LOVELY PROFESSIONAL UNIVERSITY

DATA STRUCTURE

Edited By
Dr. Pooja Gupta

ISBN: 978-93-87034-70-9

Printed by
EXCEL BOOKS PRIVATE LIMITED
Regd. Office: E-77, South Ext. Part-I, Delhi-110049
Corporate Office: 1E/14, Jhandewalan Extension, New Delhi-110055
+91-8800697053, +91-011-47520129
info@excelbooks.com/projects@excelbooks.com
internationalalliance@excelbooks.com
www.excelbooks.com

for
Lovely Professional University
Phagwara

CONTENTS

Unit 1:	Introduction to Data Structures *Anil Sharma, Lovely Professional University*	1
Unit 2:	Complexity Analysis *Rishi Chopra, Lovely Professional University*	19
Unit 3:	Arrays *Mandeep Kaur, Lovely Professional University*	37
Unit 4:	Pointers *Pawan Kumar, Lovely Professional University*	59
Unit 5:	Introduction to Linked List *Mithilesh Kumar Dubey, Lovely Professional University*	79
Unit 6:	Linked List Operations *Manpreet Kaur, Lovely Professional University*	91
Unit 7:	Stacks *Anil Sharma, Lovely Professional University*	119
Unit 8:	Queues *Sarabjit Kumar, Lovely Professional University*	129
Unit 9:	Recursion *Sahil Rampal, Lovely Professional University*	149
Unit 10:	Trees *Kamlesh Lakhwani, Lovely Professional University*	177
Unit 11:	Introduction to Binary Trees *Kumar Vishal, Lovely Professional University*	199
Unit 12:	Binary Tree Traversals and Operations *Balraj Kumar, Lovely Professional University*	213
Unit 13:	Binary Search Trees *Yadwinder Singh, Lovely Professional University*	235
Unit 14:	Heaps *Pooja Gupta, Lovely Professional University*	249

SYLLABUS

Data Structure

Objectives: The objectives for this course are to gain a solid understanding of the following topics:

- The fundamental design, analysis, and implementation of basic data structures and algorithms
- Principles for good program design, especially the uses of data abstraction and modular program composition
- Basic concepts in the specification and analysis of programs

S. No.	Description
1.	**Basic concepts and notations**: Data structures and data structure operations
2.	**Complexity Analysis**: Mathematical notation and functions, algorithmic complexity and time space trade off, Big O Notation, The best, average & Worst cases analysis of various algorithms.
3.	**Arrays:** Linear & Multidimensional Arrays, Representation & traversal
4.	**Pointers:** Array Pointers, Records and Record Structures, Representation of Records in Memory; Parallel Arrays
5.	**Linked list**: Representation, traversal, searching, Insertion, deletion of linked list. Two way/multi linked structures, Header Lists, Circular Lists
6.	**Stacks**: Basic operation of Stack, Memory Representation, Traversal. Queues: Operations, Representation & Types.
7.	**Recursion**: Definition, Function Call & Recursion implementation, Anatomy of Recursive Call, Complexity issues
8.	**Trees**: Definition, Representation in memory.
9.	**Binary trees**: Binary tree traversal, Insertion, Deletion & Searching
10.	**Binary Search Trees**: Search, Insertion, deletion Intro to Heaps

Unit 1: Introduction to Data Structures

CONTENTS

Objectives

Introduction

1.1 Basic Concepts and Notations of Data Structures

1.1.1 Data Structures and Algorithms

1.1.2 The Concept of Data Type

1.1.3 Data Structure Notations

1.2 Need for Data Structures

1.2.1 Goals of Data Structure

1.2.2 Features of Data Structure

1.3 Classification of Data Structure

1.3.1 Primitive Data Structure

1.3.2 Non-primitive Data Structure

1.4 Abstract Data Type

1.4.1 Abstract Data Type (ADT) Model

1.5 Data Structure Operations

1.5.1 Operations on Primitive Data Structure

1.5.2 Operations on Non-primitive Data Structure

1.6 Summary

1.7 Keywords

1.8 Self Assessment

1.9 Review Questions

1.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the basic concepts and notations of data structures
- Discuss the need for data structures
- Explain the classification of data structures
- Discuss abstract data types
- Explain data structure operations

Introduction

The term data refers to numerical or other information represented in a form, suitable for processing by a computer. Structure can be defined as the way in which parts are arranged or put together to form a whole. A data structure can be defined as the combination of data and all the potential operations, which are required for those set of data. The basic data items include bits, characters and integers. Data structures deal with manipulation and assembling of data. However, the data available is usually in the

amorphous form. When different types of such amorphous data are related to each other, we refer to it as a data structure.

Example: Queues, Stacks, Trees etc.

A data structure can be described as a set of domains d, a set of functions F and a set of rules A.

D = {d, F, A}

Where,

D refers to Data structure

d refers to Domain variable

F refers to a set of functions or procedures operating on domain variables.

A is a set of rules governing the operations of these functions on the domain variable.

The instructions of a computer program use data to perform certain tasks. Some programs generate data without using any inputs. Some programs generate data using a set of inputs, while some programs use a data set to manipulate the given data. Thus, the data is processed efficiently only by organizing them in a particular structure.

The study of data structures helps to understand the basic concepts involved in organizing and storing data as well as the relationship among the data sets. This in turn helps to determine the way information is stored, retrieved and modified in a computer's memory. The study of data structures is not limited to the study of data sets. It further extends to the study of representation of data elements. This means that it explains how different types of data are placed in the computer's memory using the binary number system, which forms the storage structure or memory representation. Data structure is implemented in computer programs to manage data. The data is managed using certain logical or mathematical models or concepts. A complex data structure can also be built using simple data structures.

1.1 Basic Concepts and Notations of Data Structures

Data structure is a branch of computer science. The study of data structure helps you to understand how data is organized and how data flow is managed to increase efficiency of any process or program. Data structure is the structural representation of logical relationship between data elements. This means that a data structure organizes data items based on the relationship between the data elements.

Example: A house can be identified by the house name, location, number of floors and so on. These structured set of variables depend on each other to identify the exact house. Similarly, data structure is a structured set of variables that are linked to each other, which forms the basic component of a system

1.1.1 Data Structures and Algorithms

A data structure is basically an arrangement of data within a computer's memory in computer-understandable language. In other words, data is stored in 0 and 1 format and is retrieved in ASCII (American Standard Code for Information Interchange) codes, which is human-understandable format.

The structural and functional aspects of the program depend on the design of the data structure. Thus, a data structure forms the basic building block of a program. Different data structures are used in applications for efficient operation of these applications. The programmers must select the correct data structure to write more efficient programs. This helps to solve the complexity of the problems at a rapid rate.

In computer science, an algorithm is defined as a finite list of distinct instructions for calculating a function. Algorithms are used for data processing, calculation and automated reasoning. An algorithm can also be defined as a set of rules that accurately defines a series of operations.

Example: Instructions for assembling a puzzle can be an example of an algorithm. If you are given a preliminary set of marked pieces, you can follow the instructions given to complete the puzzle.

According to Levitin, algorithms can be defined as, "A sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time."

Most computer programs involve an algorithm and one or more data structures. An appropriate data structure needs to be selected for an algorithm as the efficiency of the algorithm depends on the data structure chosen. By increasing the data storage space, you may not be able to reduce the time needed for processing the data and vice versa.

Example: When we want to print a mailing list alphabetically we need to use a data structure and an algorithm. We first arrange all the names in a data structure (array) and then sort the names alphabetically using an algorithm (sorting).

1.1.2 The Concept of Data Type

A data type comprises a set of data with values and consists of predefined set of characteristics. To be specific, data is usually stored in a variable, where the value of the variable changes according to the program being executed. The four commonly used data types in C language include int (integer), float (real number), char (character) and pointer. The keywords int, float, char and so on always take lowercase letters. Generally, a data type includes constants and variables. A constant is considered as an entity that does not change in any given program. A variable is an entity that may change from one program to another. It is necessary to specify the variables that will be used in a program. Therefore, type declaration is made by giving the data type and then the variable names. The syntax for declaring the data type and the variable name is as given below:

Syntax:

<(data type)><variable names>;

Example: int x;
Where, int is a data type and **x** is a variable

Integer Data Type

An integer data type includes only whole numbers. It does not contain any fractional data. It is denoted by the keyword **int**. It occupies 2 bytes of memory space. Integer data type can either be signed or unsigned. The signed type integer takes both positive and negative values. The range of integer constant is from -32768 to +32767 (-2^15 to +2^15 - 1) for a 16-bit compiler and -128 to 127 (-2^7 to +2^7 - 1) for an 8-bit complier. A 16-bit compiler uses one bit for storing sign and the remaining 15-bits for storing numbers. An 8-bit complier uses one bit for storing sign and the remaining 7-bits for storing numbers. The unsigned integer ranges from 0 to 65535. The signed and unsigned integers are specified as follows:

Syntax:

Unsigned int value;

Signed int value;

The integer data type is denoted by placeholder format string % d, which indicates that the data being used is of integer values.

Example: int x;
scanf ("%d", &x);

Generally, there are three classes of integers namely, short int, int and long int. As shown in table 1.1, short integers occupy only 1 byte, int occupies 2 bytes and the long integers occupy 4 bytes of memory. Long integers can store longer range of values when compared to integer and short integer.

Table 1.1: Integer Data Type Memory Allocation

Short int	Int	Long int
1 Byte	2 Bytes	4 Bytes

Floating Point Data Type

The floating point data type contains fractional numbers/real numbers and stores a maximum of six digits after decimal point. The keyword used to denote floating point number is 'float'. It occupies 4 bytes of memory space. The floating point data type is denoted by the placeholder %f, which indicates that the data being used is of floating point values. The three classes of floating point data type are float, double and long double.

Example:

```
float x;
scanf ("%f", &x);
```

As shown in table 1.2, float occupies 4 bytes of memory space. Double has longer precision than float and occupies 8 bytes of memory space. The long double further extends the precision and occupies 10 bytes of memory space.

Table 1.2: Floating Point Data Type Memory Allocation

Float	Double	Long double
4 Byte	8 Bytes	10 Bytes

Character Data Type

Character data type consists of a single character. It can store a single special symbol or alphabet placed within single inverted commas. It is denoted by the keyword char. It occupies only 1 byte of memory space. The character data type is denoted by placeholder %c, which indicates that the data being used is of character values.

Example:

```
char x;
scanf ("%c", &x);
```

Table 1.3 gives the syntax and examples of the different data types.

Table 1.3: Data Types

Data type	Syntax	Examples	Explanation
Int	int<variable name>	int x; x = 5; short int x; long int x; unsigned int x; signed int x;	In the example, int is a data type and x is a variable name. Variable can also hold a value. The value of variable x is 5.
Float	float <variable name>	float x; x = 6; double x; long double x;	In the example, float is a data type and x is a variable name. 'float' will interpret the integer value 6 as 6.0.
Char	char <variable name>	char x; x = 'a'; char x; x = '5'; char x; x = '+';	In the example, char is a data type and x is a variable name. The character to be assigned to the char variables is specified within the single quotes.

From the table 1.3, it is clear that each data type is capable of holding a particular type of value. This helps to determine the possible operations that can be performed using that data.

Pointers

A pointer is a reference data structure. A pointer is actually a variable that stores the address of another variable or structure in a program. The pointer variable holds only the memory location and not the actual content. The pointer normally uses the address operator represented by '&' and the indirection operator represented by '*'. The address operator provides the address of the variable and the indirection operator provides the value of the variable which is being pointed by the pointer variable.

Notes

The majority of microcomputers in this world use the ASCII (American Standard Code for Information Interchange) Character Set which has established values for 0 to 127.

1.1.3 Data Structure Notations

To find the best solution for a particular process, we must know which solution will take less time to run. To make the selection process easier, few notations are used, which are:

1. ***Big-O Notation:*** Big-O is the technique of expressing the upper bound of the running time of an algorithm. It is an evaluation of the maximum amount of time it could possibly take for the algorithm to complete.

Example: The running time for printing a list of n items looking at each item once can be expressed as O (n).

2. ***Big-Omega Notation:*** This notation describes the lower bound of the running time of an algorithm. It describes the best case running time of an algorithm for a given data size.

Example: The running time for finding the total number of prime factors of n, counting prime factors with multiplicity can be expressed as Ω (n).

3. ***Big-Theta Notation:*** Big-Theta notation gives both the upper and lower bounds of the running time of an algorithm. This type of notation is used for comparing run-times or growth rates between two growth functions.

Example: In a linear search algorithm, the lower bound is expressed as Θ (1) and the upper bound as Θ (n).

1.2 Need for Data Structures

The study of data structure helps programmers to store and manipulate data efficiently. Data structures help to understand the relationship of a data element with other data elements. They also provide various methods to organize and represent the data within the computer's memory.

Data structure is imperative since it governs the types of operations we perform on the data, and the competency of the operations carried out. It also governs how dynamic we can be in dealing with our data.

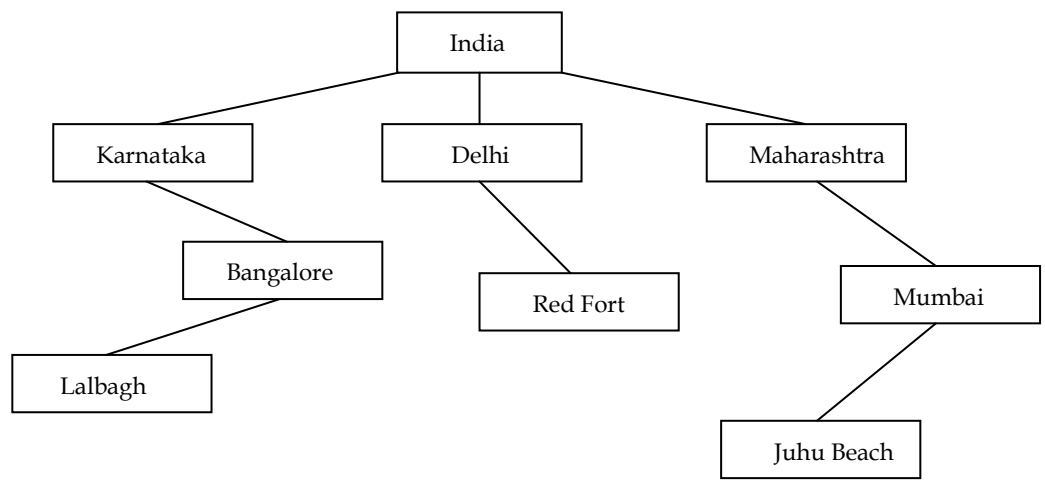
Notes

Data structures specify if we can add additional data on the fly, or if we need to know about all the data initially. We determine which data structures to use for storing our data only after we have vigilantly analyzed the problem and know what we are expected to do with the data.

There are different ways to organize data, for which there is a need for different kinds of data structure.

Consider a set of data that represents points of interest in a country. For each point of interest, the location is represented by country name in the first level, then the state name and finally the point of interest. Here, data is organized in a hierarchical form and hence it needs a hierarchical type of data structure as shown in figure 1.1.

Figure 1.1: Hierarchical Type of Data Structure



There are more real world scenarios, where the data structure is implemented effectively to manage the data.

Example: Banks perform many operations such as opening accounts, closing accounts, adding money, and withdrawing money from an account. Hence, banking applications need appropriate data structures for their database system. The data structure should be such that it can efficiently handle search operations as well as basic functions like randomly inserting and deleting the data items from anywhere in the structure.

From the preceding discussion it is clear that data structures are needed to analyze, store and organize data in a logical manner.

Task Find some real world data which can be represented as a hierarchical data structure.

1.2.1 Goals of Data Structure

Data structure basically implements two complementary goals. The first goal of data structure is to develop mathematical entities and operations, which can be used to solve particular classes of problems. The second goal is to find out representations for these entities and then implement the operations on those representations. This goal considers implementing the high level data type to solve the problems, which in turn uses existing data types.

The fundamental goal of data structure is to produce solutions that are correct and efficient. This in turn helps to produce quality software. Generally, the production of quality data structure to have quality software implementation involves the following additional goals:

Correctness

Data structure is designed such that it operates correctly for all kinds of input, which is based on the domain of interest. In other words, correctness forms the primary goal of data structure, which always depends on the specific problems that the data structure is intended to solve.

Example: A data structure designed to store a collection of numbers, in a specific order, must make sure that the numbers are not stored in a haphazard way.

Efficiency

Data structure also needs to be efficient. It should process the data at high speed without utilizing much of the computer resources such as memory space. In a real time state, the efficiency of a data structure is an important factor that determines the success and failure of the process.

Example: NASA space shuttle requires a high level data structure design, so that it reacts quickly to any changing conditions during a lift-off.

1.2.2 Features of Data Structure

Some of the important features of data structures are:

1. Robustness
2. Adaptability
3. Reusability

Robustness

Generally, all computer programmers wish to produce software that generates correct output for every possible input provided to it, as well as execute efficiently on all hardware platforms. This kind of robust software must be able to manage both valid and invalid inputs.

Example: A program is developed which takes an integer as an input. However, if a floating point number is given as an input to it, then the program must be able to manage this error and recover gracefully from this error.

Similarly, software must produce correct solution in spite of any limitation of the computer.

Example: If a user wants to store more data items in a data structure than expected, then the software must be capable of extending the capacity of the structure to store the additional data.

Adaptability

Developing software projects such as word processors, Web browsers and Internet search engine involves large software systems that work or execute correctly and efficiently for many years. Moreover, software evolves due to ever changing market conditions or due to emerging technologies.

Example: Software evolves to adapt to the increase in speed of CPU or network, or it evolves to add new functionality as per the market demand.

Thus, the goal of data structure is to develop quality software which is capable of adapting to any given situation.

Reusability

Reusability and adaptability go hand-in-hand.

Example: In reusable software, the code of the particular software developed can easily be incorporated or adapted in the component of different systems or application domains.

It is a known fact that the programmer requires many resources for developing any software, which makes it an expensive enterprise. However, if the software is developed in a reusable and adaptable way, then it can be implemented in most of the future applications. Thus, by implementing quality data structures, it is possible to develop reusable software, which tends to be cost effective and time saving.

Caselet

Data Structures – Making Things Easy

Einstein@Home is a distributed computing software project. It used NVIDIA's CUDA (CUDA is NVIDIA's parallel computing architecture) architecture to enable drastic increase in computing performance. But, with the existing data structure, it performed poorly when used with graphics processing units (GPUs). The whole data structure was redesigned. A novel spatial data structure called dynamic grid was optimized for CUDA usage. This resulted in a three-fold improvement in the performance of the GPUs. This improvement in the performance was achieved without even optimizing the code executed on the device. Thus the proper usage of data structure proved beneficial for the project.

Source: http://gpgpu.org/2008/08/11/case-studies-on-gpu-usage-and-data-structure-design

1.3 Classification of Data Structure

A data structure provides a structured set of variables that are associated with each other in different ways. It forms a basis of programming tool that represents the relationship between data elements and helps programmers to process the data easily.

Data structure can be classified into two categories:

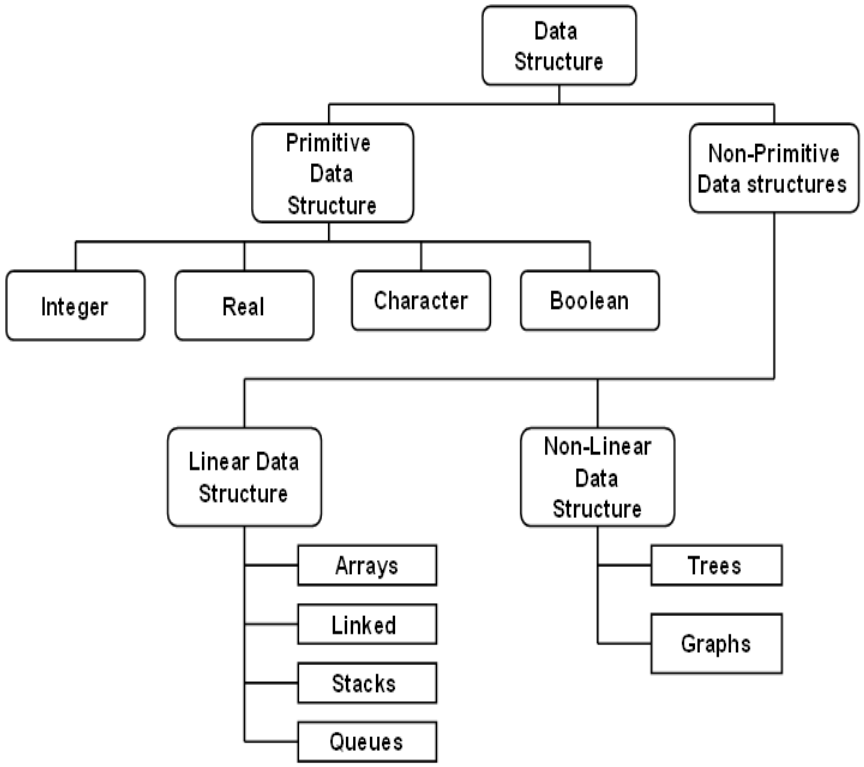
1. Primitive data structure
2. Non-primitive data structure

Notes

Many different names are used for the data elements of a data structure. Some examples are "data element", "data object", "node" and "record". The specific name that is used depends on the type of data structure.

Figure 1.2 shows the different classifications of data structures.

Figure 1.2: Classification of Data Structure



1.3.1 Primitive Data Structure

Primitive data structures consist of the numbers and the characters which are built in programs. These can be manipulated or operated directly by the machine level instructions. Basic data types such as integer, real, character, and Boolean come under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.

Integer is used for integral or fixed-precision values. It is denoted as int. The type INTEGER includes a subset of the whole numbers whose size may differ in different computer systems. It is considered that all operations on data of this type are precise and correspond to the ordinary laws of arithmetic, and if the result lies outside the representable subset, the computation might fail.

The primitive data type REAL designates a subset of the real numbers. Arithmetic done with operands of the type INTEGER is considered to yield accurate results, whereas, arithmetic on values of type REAL is permitted to be incorrect within the limits of round-off errors. This is the main reason for the apparent distinction between the types INTEGER and REAL. The four basic arithmetic operations which are also the standard operators are addition (+), subtraction (-), multiplication (*), and division (/).

Character is used for character values. It is denoted as Char. The standard type CHAR consists of a set of printable characters. The type CHAR comprises 26 upper-case letters, 26 lower-case letters, 10 decimal digits, and a number of other graphic characters such as, punctuation marks. The subsets of letters and digits are structured and contiguous. Every single computer stores character data in a one byte field as an integer value. A byte consists of 8 bits so the one byte field has 256 possibilities using the positive values of 0 to 255.

The type Boolean is used for Boolean values. The standard type BOOLEAN values are denoted by the two identifiers TRUE and FALSE. The Boolean operators comprise logical conjunction, disjunction, and negation. The logical conjunction is denoted by the symbol &, the logical disjunction by OR, and negation by "~".

Notes Based on the language and its implementation, primitive data types may or may not have a one-to-one connection with objects in the computer's memory.

1.3.2 Non-primitive Data Structure

Non-primitive data structures are those that are derived from primitive data structures. These data structures cannot be operated or manipulated directly by the machine level instructions. They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type). These are further divided into linear and non-linear data structure based on the structure and arrangement of data.

Linear Data Structure

A data structure that maintains a linear relationship among its elements is called a linear data structure. Here, the data is arranged in a linear fashion. But in the memory, the arrangement may not be sequential.

Example: Arrays, linked lists, stacks, queues.

Array

Array, in general, refers to an orderly arrangement of data elements. Array is a type of data structure that stores data elements in adjacent locations. Array is considered as linear data structure that stores elements of same data types. Hence, it is also called as a linear homogenous data structure.

When we declare an array, we can assign initial values to each of its elements by enclosing the values in braces { }.

Example: int Paul [5] = { 26, 7, 67, 50, 66 };

This declaration will create an array as shown below:

	0	1	2	3	4
Paul	26	7	67	50	66

The number of values inside braces { } should be equal to the number of elements that we declare for the array inside the square brackets [ ]. In the example of array Paul, we have declared 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element. After this declaration, array Paul will have five integers, as we have provided 5 initialization values.

Arrays can be classified as one-dimensional array, two-dimensional array or multidimensional array.

1. ***One-dimensional Array*:** It has only one row of elements. It is stored in ascending storage location.
2. ***Two-dimensional Array:*** It consists of multiple rows and columns of data elements. It is also called as a matrix.

3. ***Multidimensional Array:*** Multidimensional arrays can be defined as array of arrays. Multidimensional arrays are not bounded to two indices or two dimensions. They can include as many indices as required.

Linked List

A linked list is a data structure in which each data element contains a pointer or link to the next element in the list. Through linked list, insertion and deletion of the data element is possible at all places of a linear list. Also in linked list, it is not necessary to have the data elements stored in consecutive locations. It allocates space for each data item in its own block of memory. Thus, a linked list is considered as a chain of data elements or records called nodes. Each node in the list contains information field and a pointer field. The information field contains the actual data and the pointer field contains address of the subsequent nodes in the list.

Figure 1.3: A Linked List

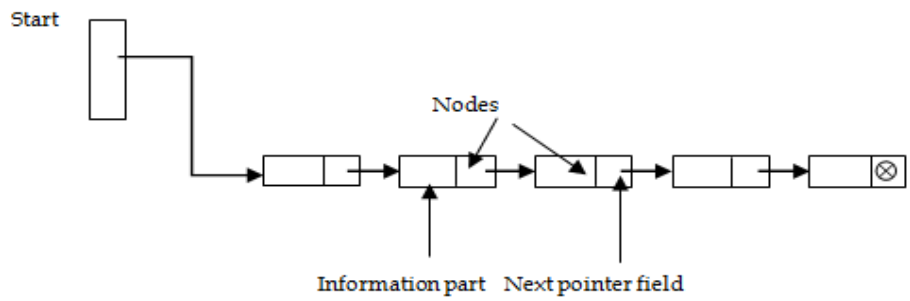


Figure 1.3 represents a linked list with 4 nodes. Each node has two parts. The left part in the node represents the information part which contains an entire record of data items and the right part represents the pointer to the next node. The pointer of the last node contains a null pointer.

Stacks

A stack is an ordered list in which data items are inserted and deleted only from one end. It is also known as Last-In First-Out list (LIFO) because the last element which enters the stack will be on top of the stack and is the first one to come out.

Figure 1.4: A Stack

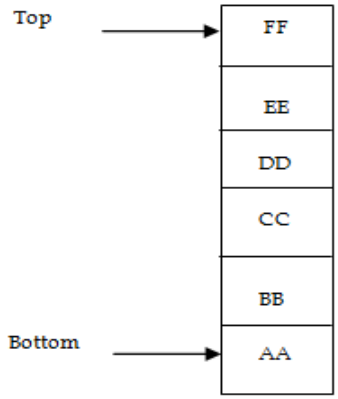


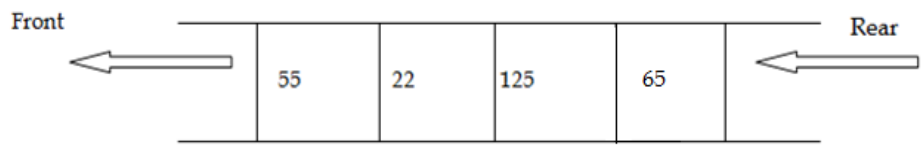
Figure 1.4 is a schematic diagram of a stack. Here, element FF is the top of the stack and element AA is the bottom of the stack. Elements are added to the stack from the top. Since it follows LIFO pattern, EE

cannot be deleted before FF is deleted, and similarly DD cannot be deleted before EE is deleted and so on.

Queues

Queue is a non-primitive linear data structure, where the homogeneous data elements are stored in sequence. In queue, data elements are inserted from one end and deleted from the other end. Hence, it is also called as First-In First-Out (FIFO) list. Figure 1.5 shows a queue with 4 elements, where 55 is the front element and 65 is the rear element. Elements can be added from the rear and deleted from the front.

Figure 1.5: A Queue



Non-linear Data Structure

Non-linear data structure is a kind of data structure in which data elements are not arranged in a sequential order. There is a hierarchical relationship between individual data items. Here, the insertion and deletion of data is not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

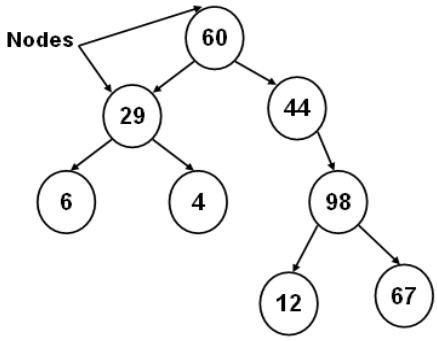
Example: Trees and graphs are examples of non-linear data structures.

Trees

A tree is a non-linear data structure in which data is organized in branches. The data elements in tree are arranged in a sorted order. It imposes a hierarchical structure on the data elements.

Figure 1.6 represents a tree which consists of 8 nodes. The root of the tree is the node 60 at the top. Node 29 and 44 are the successors of the node 60. The nodes 6, 4, 12 and 67 are the terminal nodes as they do not have any successors.

Figure 1.6: A Tree



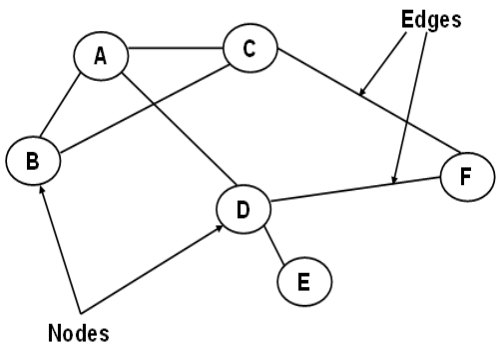
Did you know? Each value or key in a tree occurs only once, i.e., there are no duplicates.

Graphs

A graph is also a non-linear data structure. In a tree data structure, all data elements are stored in definite hierarchical structure. In other words, each node has only one parent node. While in graphs, each data element is called a vertex and is connected to many other vertexes through connections called edges.

Thus, a graph is considered as a mathematical structure, which is composed of a set of vertexes and a set of edges. Figure 1.7 shows a graph with six nodes A, B, C, D, E, F and seven edges [A, B], [A, C], [A, D], [B, C], [C, F], [D, F] and [D, E].

Figure 1.7: A Graph



1.4 Abstract Data Type

According to National Institute of Standards and Technology (NIST), a data structure is an organization of information, usually in the memory, for better algorithm efficiency. Data structures include queues, stacks, linked lists, dictionary, and trees. They could also be a conceptual entity, such as the name and address of a person.

From the above definition, it is clear that the operations in data structure involve higher-level abstractions such as, adding or deleting an item from a list, accessing the highest priority item in a list, or searching and sorting an item in a list. When the data structure does such operations, it is called an abstract data type.

An Abstract Data Type [ADT] is a technique that is used to specify the logical properties of a data type. ADT can be considered as a basic mathematical concept used to define the data types. An ADT consists of two parts, namely, a value definition and an operator definition. A value definition consists of a definition clause and a condition clause.

Notes Two basic structures, namely array and linked list can be used to implement an ADT list.

The operator definition consists of three parts: a header, preconditions, and post-conditions. The preconditions and post-conditions are optional and can be used depending on the program requirement.

Example: Complex numbers consists of a real part and an imaginary part and both parts are represented by real numbers. Different operations like addition, subtraction, multiplication, and division can be performed on complex numbers. As we do not have a true data type for complex numbers, we implement them using an ADT.

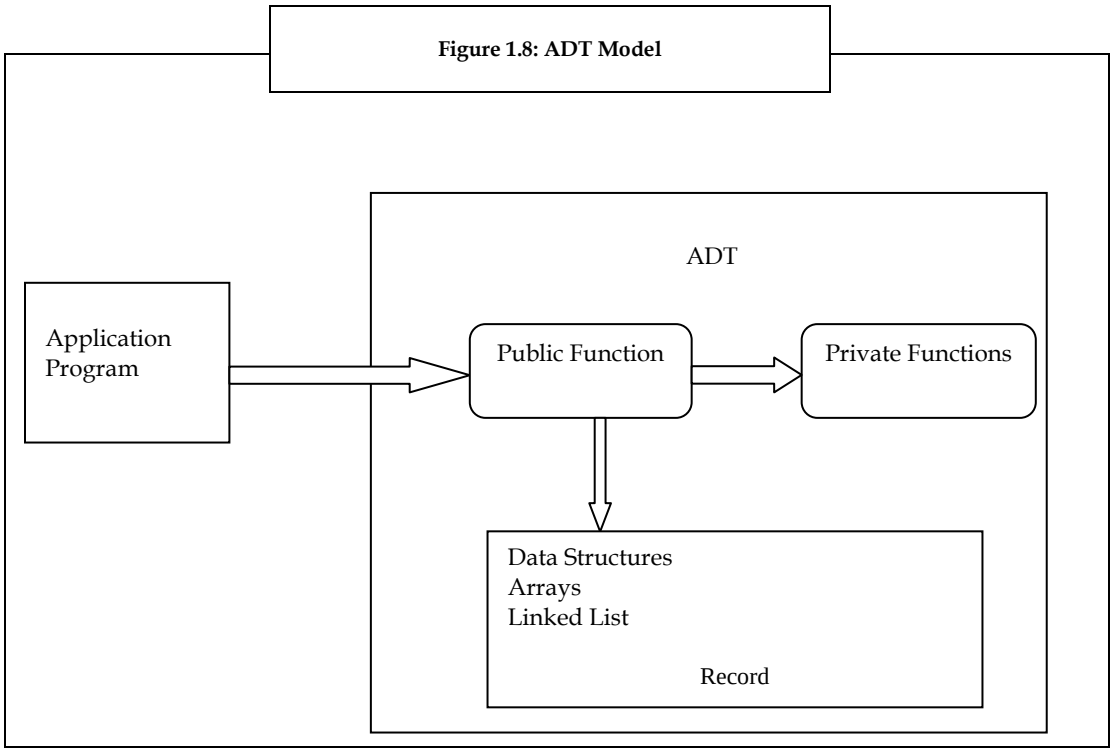
Thus, ADT can be considered as the mathematical model for data structures that have similar semantics.

Did you know? Applying an ADT means providing one procedure or function for every abstract operation. The ADT instances are illustrated by some concrete data structure that is manipulated by those procedures, according to the ADT's specifications.

1.4.1 Abstract Data Type (ADT) Model

As shown in figure 1.8, there are two different parts of the ADT model - functions (public and private) and data structures. Both, the functions and data structures are contained inside an ADT model and are not part of the application program. Data structures are available to all the ADT functions when required and a function may call any other function to accomplish its task. This shows that data structures and functions are within the scope of each other. Data is entered, accessed, modified and deleted through the external application programming interface. This interface can access only the public functions. Every ADT operation has an algorithm to perform a specific task. The operation name and parameters are available to the application, and they provide the interface to the application. Simple structures are used to implement a program that completely controls a list.

Figure 1.8: ADT Model



Caution It is not enough if you just encapsulate the structure in an ADT. It is also necessary that multiple versions of the structure coexist.

1.5 Data Structure Operations

Data which appears in the data structures are processed with the help of certain operations. Sometimes two or more of the operations may be used in a given situation.

Example: When you want to delete a record with a given key, you first need to use the search operation to find the location of the record and then use the delete operation.

1.5.1 Operations on Primitive Data Structure

Various operations can be performed on primitive data structures. Some of these operations are:

1. ***Creation Operation:*** The creation operation creates a data structure.

Example: Consider an example of integer type data structure.

int a;
Here, declaration of int creates 2 bytes of memory space for variable 'a'. This variable is used to store only integer value.

2. ***Destroy Operation:*** The destroy operation destroys the data structure. In C language, a function called 'free()' is used to destroy the data structure. This helps in efficient use of memory.
3. ***Selection Operation:*** The selection operation is used to access data within a data structure. The significance of selection operation is provided in file data structure. Files provide the option of sequential and random access, which totally depend on the nature of files.
4. ***Update Operation:*** The update operation is used to modify data in data structure.

1.5.2 Operations on Non-primitive Data Structure

The operations on non-primitive data structure depend on the logical organization of data and their storage structure. Non-primitive data focuses on formation of a set of data elements that are either homogeneous (same data type) or heterogeneous (different data type). Therefore, non-primitive data cannot be operated or manipulated directly by the machine level instructions. Some of the operations on non-primitive data structure are:

1. ***Traversing:*** Traversing is the method of processing each element exactly once. Traversing is generally done to check the availability of data elements in an array. After carrying out an insertion or deletion operation, you would want to check whether the operation has been successful or not. We use traversing to check if the element is successfully inserted or deleted.
2. ***Sorting:*** Sorting is the technique of arranging the data elements in some logical order, either ascending or descending order. Some algorithms make use of sorted lists. Therefore, efficient sorting is essential for optimizing these algorithms to ensure that they work correctly.
3. ***Merging*:** Merging is the method of combining the elements in two different sorted lists into a single sorted list. It is based on the divide-and-conquer algorithm. Merge sort can be considered as the best choice for sorting a linked list as it is easy to implement.
4. ***Searching:*** Searching is the method of finding the location of an element with a given key value, or finding the location of an element which satisfies a given condition. Searching a data structure allows the efficient retrieval of unambiguous items from a set of items, such as a particular record from a database.
5. ***Insertion:*** Insertion is the method of adding a new element to the data structure. The insertion process may add a new element in the ith position of the data structure. If sorting also needs to be performed, first we need to assign an item to the given elements and compare it with the previous elements. If the assigned element is smaller than the previous element, we need to swap the positions of both these items. This process is repeated until the correct position of the item is identified.

6. ***Deletion:*** Deletion refers to removing an item from the structure. When a node is not required in the data structure, it can be removed using the delete operation.

Task

A Company has a membership file in which each record contains the following data for a given member: Name, Sex, Age, and Phone Number.

Suppose the company wants to send email invitations for a party to all the members, which operation will be used to obtain the name and phone number of each member? Discuss.

1.6 Summary

- A data structure is the organization or arrangement of data in a computer's memory or disk storage, so that it can be used efficiently.
- The study of data structure helps students to understand how data is organized and how the data flow is managed to increase efficiency of any process or program.
- Algorithms are used for data processing, calculations, and automated reasoning. An algorithm can be defined as a set of rules that accurately define a series of operations.
- A data type comprises a set of data with values and consists of predefined set of characteristics. The four commonly used data types in C are int (integer), float (real number), char (character), and pointer.
- Two fundamental goals of data structure are correctness and efficiency. Some of the important features of data structures are robustness, adaptability and reusability.
- Data structure can be classified into two categories: primitive data structure and non-primitive data structure.
- Basic data types such as integer, real, character, and Boolean are categorized under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.
- Non-primitive data structures are further divided into linear and non-linear data structure based on the structure and arrangement of data.
- Arrays, linked lists, stacks, queues are examples of linear data structure. Trees and graphs are examples of non-linear data structure.
- An Abstract Data Type (ADT) is a technique that is used to specify the logical properties of a data type. It can be considered as a basic mathematical concept used to define the data types.
- Data that appears in the data structures are processed with the help of certain operations. Sometimes two or more operations may be used for a data structure in a given situation.

1.7 Keywords

Amorphous: Not having a definite form; shapeless.

Application Program: A program designed to perform a particular function directly for the user or for another application program.

Private Functions: Functions that can be accessed only by the members of the same data structure.

Public Functions: Functions that can be accessed by the members internal to the data structure as well as by members of other data structures.

1.8 Self Assessment

1. State whether the following statements are true or false:
   (a) A data structure is the organization or arrangement of data in a computer's disk storage.
   (b) Data structure organizes data items based on the relationship between the data elements.
   (c) The study of microcomputers helps to create many complex applications, which involve many programmers and designers.
   (d) Efficiency forms the primary goal of the data structure.
   (e) Reusability and robustness go hand-in-hand in a software program.
   (f) The significance of selection operation is provided in file data structure.
   (g) Insertion operation is not possible if the sorting operation is not carried out beforehand.
   (h) Merging is based on the divide-and-conquer algorithm.
2. Fill in the blanks:
   (a) In computer science, …….. is a useful method defined as a finite list of distinct instructions for calculating a function.
   (b) Practically, a data type includes a constant and …….
   (c) Floating point data type can store a maximum of …… digits after decimal point.
   (d) Arithmetic done with operands of the type …… is considered to yield accurate results.
   (e) …… is a kind of data structure that stores elements of same data types.
   (f) Last-In First-Out list (LIFO) process happens in a …… data structure.
   (g) …… is considered as the mathematical model for data structures that have similar semantics.
   (h) The two different parts of the ADT model are functions and ……

1.9 Review Questions

1. ”Data structure is important in building a software program.” Discuss.
2. “The structural and functional aspects of the program depend on the design of the data structure”. Comment.
3. By increasing the amount of space for storing the data, you may not be able to reduce the time needed for processing the data. Why?
4. Why do you think long integers can store longer range of values when compared to integer and short integer?
5. “The pointer variable holds only the memory location and not the actual content”. Discuss.
6. Analyze why efficiency is one of the major goals of data structure.
7. “Reusability and adaptability go hand-in-hand”. Discuss.
8. What is the chief reason for the apparent distinction between the data types - integer and real?
9. “Non-linear data structures do not permit the insertion and deletion of data in a linear fashion.” Comment.

10. Can we consider abstract data type as a basic mathematical concept which is used to define the data types? Justify your answer.
11. "Files provide the option of sequential and random access". Comment.
12. "Non-primitive data cannot be operated or manipulated directly by the machine level instructions". Discuss.

Answers: Self Assessment

1. (a) True (b) True (c) False (d) False (e) False (f) True (g) False (h) True
2. (a) Algorithm (b) Variables (c) Six (d) Integer (e) Array (f) Stack (g) Abstract data type (h) Data structure

1.10 Further Readings

Books

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill
Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Online link

http://www-old.oberon.ethz.ch/WirthPubl/AD.pdf
http://msdn.microsoft.com/en-us/library/06bkb8w2%28v=vs.71%29.aspx
http://www.cplusplus.com/doc/tutorial/variables/

Unit 2: Complexity Analysis

CONTENTS

Objectives

Introduction

2.1 Mathematical Notation and Functions

2.1.1 Asymptotic Notations

2.1.2 Mathematical Functions

2.2 Algorithmic Complexity and Time Space Tradeoff

2.2.1 Algorithmic Complexity

2.3 Algorithmic Analysis

2.3.1 Types of Analysis

2.4 Summary

2.5 Keywords

2.6 Self Assessment

2.7 Review Questions

2.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain mathematical notation and functions
- Analyze the algorithmic complexity and time space tradeoff
- Discuss algorithmic analysis

Introduction

Each computer program is a series of instructions that are arranged in a specific order to perform a specific task. A computer program is written to instruct a computer to perform a specific task in order to obtain the desired result. Irrespective of the language used to develop a program, there are some generic steps that can be followed to solve a problem. These generic steps are called algorithms. According to H. Cormen, "Before there were computers, there were algorithms." An algorithm is a set of instructions that performs a particular task. It is considered as a tool that helps to solve a specific computational problem.

Mathematical notation is a system of symbolic representations of mathematical objects and ideas. Mathematical functions appear quite often in the analysis of algorithm along with their notation. Some of the mathematical functions are floor and ceiling functions, summation symbol, factorial, Fibonacci numbers, and so on.

The complexity of an algorithm is a function that describes the efficiency of an algorithm in terms of the amount of data the algorithm must process.

The two main complexity measures of efficiency of an algorithm are:

1. ***Time Complexity***: It is a function that describes the time taken by an algorithm to solve a problem.

Example: Big-O notation is used to express the time complexity of an algorithm.

2. ***Space Complexity:*** It is a function that describes the amount of memory or space required by an algorithm to run. A good algorithm has minimum number of space complexity.

Algorithm analysis is an important part of computational complexity theory. It provides theoretical estimates for the resources that are needed for any algorithm to solve a given problem. These estimates provide an insight into the measures that determine algorithm efficiency. It is necessary to check the efficiency of each of the algorithms in order to select the best algorithm. We can easily measure the efficiency of algorithms by calculating their time complexity. The shorthand way to represent time complexity is asymptotic notation.

Example: Consider the algorithm for sorting a deck of cards. This algorithm continues by repeatedly searching through the deck for the lowest card. The square of the number of cards in the deck is the asymptotic complexity of this algorithm.

2.1 Mathematical Notation and Functions

Algorithms are widely used in various areas of study. We can solve different problems using the same algorithm. Therefore, all algorithms must follow a standard. The mathematical notations use symbols or symbolic expressions, which have a precise semantic meaning.

According to Lancelot Hogben, "Every meaningful mathematical statement can also be expressed in plain language. Many plain language statements of mathematical expressions would fill several pages, while to express them in mathematical notation might take as little as one line. One of the ways to achieve this remarkable compression is to use symbols to stand for statements, instructions, and so on."

2.1.1 Asymptotic Notations

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.

To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity. The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as the fastest possible, slowest possible or average possible.

Asymptotic notation within the limit deals with the character of a function that is a parameter with large values. The main characteristic of this approach is that, importance is given to the terms while neglecting the constant factors present in the expression. This helps in the classification of run-time functions into broad efficiency classes.

The notations such as O (Big-O), Ω (Omega), and θ (Theta) are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

Big-O Notation

'O' is the representation for Big-O notation. Big-O is the method used to express the upper bound of the running time of an algorithm. It is used to describe the performance or time complexity of the algorithm. Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm.

Table 2.1 gives some names and examples of the common orders used to describe functions. These orders are ranked from top to bottom.

Table 2.1: Common Orders

Time complexity		Examples
O(1)	Constant	Adding to the front of a linked list
O(log n)	Logarithmic	Finding an entry in a sorted array
O(n)	Linear	Finding an entry in an unsorted array
O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
O(n2)	Quadratic	Shortest path between two nodes in a graph
O(n3)	Cubic	Simultaneous linear equations
O(2n)	Exponential	The Towers of Hanoi problem

Big-O notation is generally used to express an ordering property among the functions. This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem. Big-O is defined as:

$$f(n) \leq c * g(n)$$

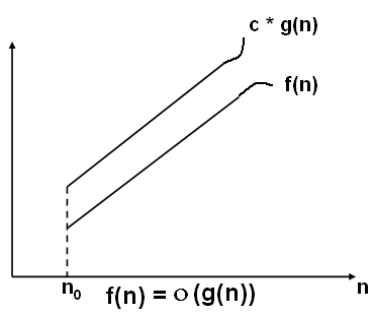
where, **n** can be any number of inputs or outputs and **f(n)** as well as **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer $\mathbf{n_0}$ such that, $n \geq n_0$.

The Big-O can also be denoted as f(n) = O(g(n)), where **f(n)** and **g(n)** are two non-negative functions and f(n) < g(n) if **g(n)** is multiple of some constant **c**. The graphical representation of f(n) = O(g(n)) is shown in figure 2.1, where the running time increases considerably when **n** increases**.**

Example: Consider $f(n)=15n^3+40n^2+2n\log_n+2n$. As the value of **n** increases, n^3 becomes much larger than n^2, $n\log_n$, and n. Hence, it dominates the function f(n) and we can consider the running time to grow by the order of n^3. Therefore, it can be written as $f(n)=O(n^3)$.

The values of n for f(n) and C* g(n) will not be less than n_0. Therefore, the values less than n_0 are not considered relevant.

Figure 2.1: Big-O Notation f(n) = O(g(n))



Source: Puntambekar, A., A. (2010). Design and Analysis of Algorithms, Technical Publications Pune.

Let us take an example to understand the Big-O notation more clearly.

Example: Consider function f(n) = 2(n)+2 and g(n) = n^2.
We need to find the constant c such that $f(n) \leq c * g(n)$.
Let n = 1, then
f(n) = 2(n)+2 = 2(1)+2 = 4
g(n) = $n^2 = 1^2 = 1$
Here, f(n)>g(n)
Let n = 2, then
f(n) = 2(n)+2 = 2(2)+2 = 6
g(n) = $n^2 = 2^2 = 4$
Here, f(n)>g(n)
Let n = 3, then
f(n) = 2(n)+2 = 2(3)+2 = 8
g(n) = $n^2 = 3^2 = 9$
Here, f(n)<g(n)
Thus, when n is greater than 2, we get f(n)<g(n). In other words, as **n** becomes larger, the running time increases considerably. This concludes that the Big-O helps to determine the 'upper bound' of the algorithm's run-time.

Notes

1. Big-O notation ignores all the constant factors and lower order factors of n.
2. Big-O notation does not determine how quickly or slowly the algorithms actually execute for a given input.

Omega Notation

'Ω' is the representation for Omega notation. Omega describes the manner in which an algorithm performs in the best case time complexity. This notation provides the minimum amount of time taken by an algorithm to compute a problem. Thus, it is considered that omega gives the "lower bound" of the algorithm's run-time. Omega is defined as:

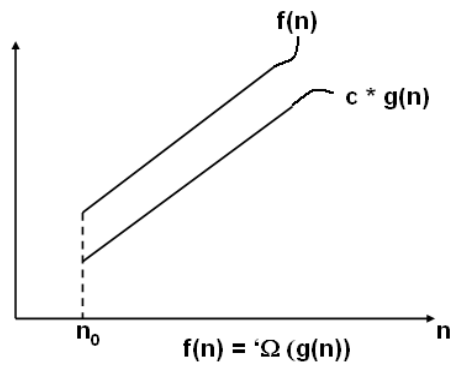
$$f(n) \geq c * g(n)$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer $\mathbf{n_0}$ such that $n>n_0$.

Omega can also be denoted as f(n) = Ω (g(n)) where, **f** of **n** is equal to Omega of **g** of **n**. The graphical representation of f(n) = Ω (g(n)) is shown in figure 2.2. The function **f(n)** is said to be in Ω (g(n)), if **f(n)** is bounded below by some constant multiple of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c** and some non-negative integer $\mathbf{n_0}$, such that $f(n) \geq c * g(n)$ for all $n \geq n_0$.

Figure 2.2 shows Omega notation.

Figure 2.2: Omega Notation f(n) = Ω (g(n))



Source: Puntambekar, A., A. (2010). Design and Analysis of Algorithms, First ed. Technical Publications Pune. Page. 23.

Let us take an example to understand the Omega notation more clearly.

Example: Consider function f(n) = 2n2+5 and g(n) = 7n.

We need to find the constant c such that $f(n) \geq c * g(n)$.

Let n = 0, then

f(n) = 2n2+5 = 2(0)2+5 = 5

g(n) = 7(n) = 7(0) = 0

Here, f(n)>g(n)

Let n = 1, then

f(n) = 2n2+5 = 2(1)2+5 = 7

g(n) = 7(n) = 7(1) = 7

Here, f(n)=g(n)

Let n = 2, then

f(n) = 2n2+5 = 2(2)2+5 = 13

g(n) = 7(n) = 7(2) = 14

Here, f(n)<g(n)

Thus, for n=1, we get $f(n) \geq c * g(n)$. This concludes that Omega helps to determine the "lower bound" of the algorithm's run-time.

Theta Notation

'θ' is the representation for Theta notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:

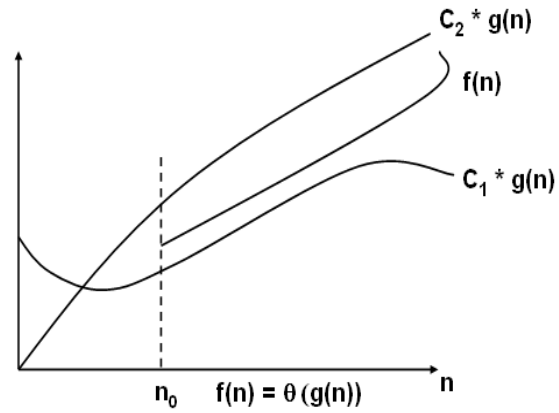
$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \text{for all } n > n_0$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there are two constants namely, **c1**, **c2,** and a non-negative integer $\mathbf{n_0}$.

Theta can also be denoted as f(n) = θ(g(n)) where, **f** of **n** is equal to Theta of **g** of **n**. The graphical representation of f(n) = θ(g(n)) is shown in figure 2.3. The function **f(n)** is said to be in θ (g(n)) if **f(n)** is bounded both above and below by some positive constant multiples of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c1** and **c2** and some non-negative integer $\mathbf{n_0}$, such that $C_2 g(n) \leq f(n) \leq C_1 g(n)$ for all $n \geq n_0$.

Figure 2.3 shows Theta notation.

Figure 2.3: Theta Notation f(n) = θ(g(n))



Source: Puntambekar, A., A. (2010). Design and Analysis of Algorithms, First ed. Technical Publications Pune. Page. 24.

Let us take an example to understand the Theta notation more clearly.

Example: Consider function f(n) = 4n + 3 and g(n) = 4n for all n ≥ 3; and f(n) = 4n + 3 and g(n) = 5n for all n ≥ 3.
Then the result of the function will be:
Let n = 3
f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 4n =4(3) = 12 and
f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 5n =5(3) = 15 and
here, c1 is 4, c2 is 5 and n0 is 3
Thus, from the above equation we get c1 *g*(n) *f*(n) c2 *g*(n). This concludes that Theta notation depicts the running time between the upper bound and lower bound.

Task Determine a constant p for a given function f(n) ≥ c * g(n) where f(n)=2n+3 and g(n)=2n.

2.1.2 Mathematical Functions

Mathematical functions express the idea that an input completely determines an output. A function provides exactly one value to each input of a specified type. The value can be real numbers or can be elements from any given sets: the domain and the codomain of the function.

Example: Function f(x)=2x
In this case, the function is assigned to every real number, the real number with twice its value.
Assume x=5, then we can write f(5) = 10.

Some of the mathematical functions are described below:

Floor and Ceiling Functions

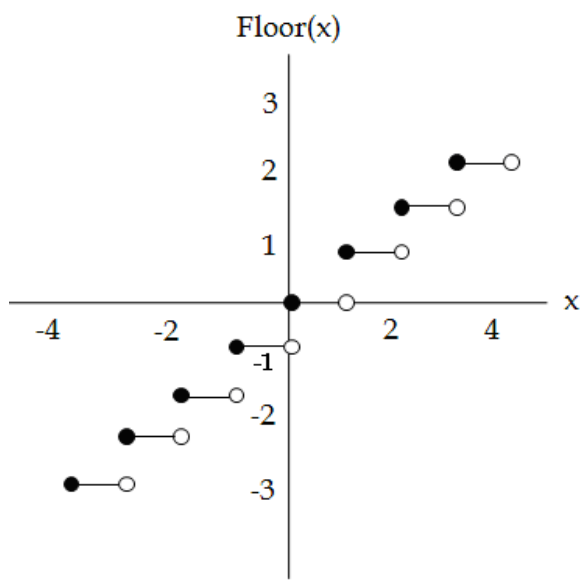
Floor function is represented as floor(x). Floor function which is also called greatest integer function gives the largest integer less than or equal to x. The range of floor(x) is the set of all integers, but the domain of floor(x) is the set of all real numbers.

Let us take an example to understand the floor function more clearly.

Example: floor(1.01)=1
floor(0)=0
floor(2.9)=2
floor(-3)=-3
floor(-1.1)=-2
Find out floor(x) for various values of x.

Figure 2.4 shows the graph for floor(x).

Figure 2.4: Floor Function



Source: http://mathworld.wolfram.com/FloorFunction.html

Ceiling function is represented as ceiling(x). It gives the smallest integer value greater than or equal to x. The domain of ceiling(x) is the set of all real numbers. The range of ceiling(x) is the set of all integers.

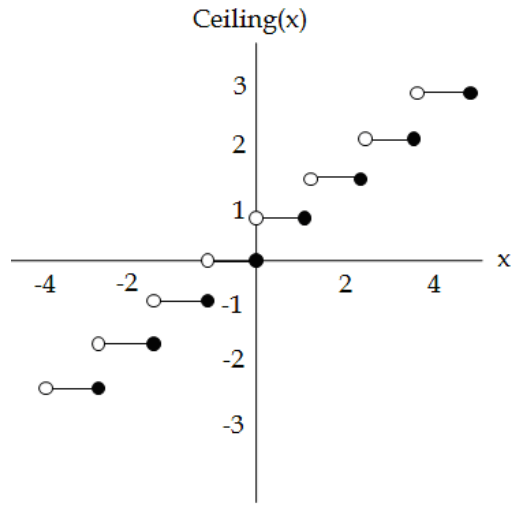
Let us consider the following example.

Example:

ceiling (1.5)=2
ceiling(0)=0
ceiling(2)=2
ceiling(-3)=-3
ceiling(-1.1)=-1
Find out ceiling(x) for various values of **x**.

Figure 2.5 shows the graph for ceiling(x).

Figure 2.5: Ceiling Function



Source: http://mathworld.wolfram.com/CeilingFunction.html

Did you know?

The name and symbol for the floor function and ceiling function was invented by K. E. Iverson (Graham et. al. 1994).

Summation Symbol

Summation symbol is Σ. Summation is the operation of combining a sequence of numbers using addition. The result is the sum or total of all the numbers. Apart from numbers, other types of values such as, vectors, matrices, polynomials, and elements of any additive group can also be added using summation symbol.

Example: Consider a sequence x1, x2, x3……x10. The simple addition of this sequence is x1+x2+x3+x4+x5+x6+x7+x8+x9+x10. Using mathematical notation we can shorten the addition. It can be done by using a symbol to denote "all the way up to" or "all the way down to".
Then, the expression will be x1+x2+x3+…..+x10. We can also represent the expression using Greek letter Σ as shown below:

$$\sum_{index\ variable=a}^{b} variable_{index\ variable}$$

Here, **a** is the first index and **b** is the last index. The variables are the numbers that appear constantly in all terms. In the expression,
x1+x2+x3+x4+x5+x6+x7+x8+x9+x10
1 is the first index, 10 is the last index, and **x** is the variable. If we use **i** as the index variable then the expression will be

$$\sum_{i=1}^{10} x_i$$

Exponent and Logarithm

Exponential function has the form f(x) =a^x+B where, **a** is the base, **x** is the exponent, and **B** is any expression.

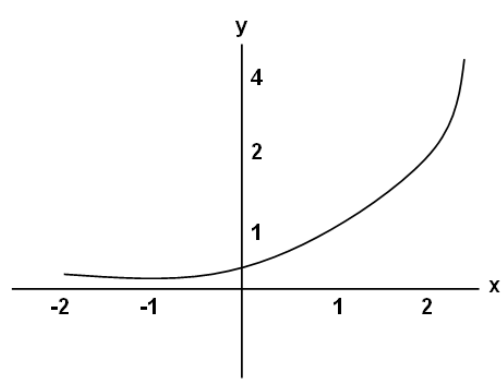
If **a** is positive, the function continuously increases in value. As **x** increases, the slope of the function also increases.

Example: Consider a function. f(x)=2x
Here, we have an exponential function with base 2. Some typical values for this function are:

x	-2	-1	0	1	2
f(x)	1/4	1/2	1	2	4

The graph for y=2x is shown in figure 2.6. In the graph as **x** increases, **y** also increases, and as **x** increases the slope of the graph also increases.

Figure 2.6: Graph for y=2^x



Source: http://www.themathpage.com/aprecalc/logarithmic-exponential-functions.htm

A logarithm is an exponent. The logarithmic function is defined as $f(x)= \log_b x$. Here, the base of the algorithm is ***b***. The two most common bases which we use are base 10 and base ***e***

Example: Consider the exponential equation $5^2=25$ where 5 is base and 2 is exponent.
The logarithmic form of this equation is:
$\log_5 25=2$

Here, we can say that the logarithm of 25 to the base 5 is 2.

Factorial

The symbol of the factorial function is '**!**'. The factorial function multiplies a series of natural numbers that are in descending order. The factorial of a positive integer **n** which is denoted by **n!** represents the product of all the positive integers is less than or equal to n.

$$n!=n*(n-1)*(n-2)\ldots\ldots 2*1$$

Example: 5!=5*4*3*2*1=120

Fibonacci Numbers

In the Fibonacci sequence, after the first two numbers i.e. 0 and 1 in the sequence, each subsequent number in the series is equal to the sum of the previous two numbers. The sequence is named after Leonardo of Pisa, also known as Fibonacci.

Fibonacci numbers are the elements of Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765…….

Sometimes this sequence is given as 0, 1, 1, 2, 3, 5….. There are also other Fibonacci sequences which start with the numbers:

3, 10, 13, 23, 36, 59…….

Fibonacci numbers are the example of patterns that have intrigued mathematicians through the ages. In mathematical terms, the sequence $\mathbf{F_n}$ of Fibonacci numbers is defined as:

$$F_n = F_{n-1} + F_{n-2}$$

Example: Beginning with a single pair of rabbits, if every month each productive pair bears a new pair, who become productive when they are 1 month old, how many rabbits will there be after n months?
Assume that there are x^n pairs of rabbits after n months. The number of pairs in n+1 month is x^{n+1}. Each pair produces a new pair every month but no rabbit dies within that period. New pairs are only born to pairs which are at least 1 month old, so there is an x^{n-1} new pair.
$X^{n+1} = x^n + x^{n-1}$
This equation shows the rules for generating the Fibonacci numbers.

Did you know? Fibonacci was the greatest mathematician of his age. He eliminated the use of complex Roman numerals and made mathematics more accessible to the public by bringing the Hindu-Arabic system (including zero) to Western Europe.

Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers. In the modular arithmetic, numbers wrap around and reach a given fixed quantity, which is known as the modulus. This is 12 in the case of hours and 60 in the case of minutes or seconds in a clock. In the 12 hour clock, the day is divided into two 12

hour periods. If the time is 6:00 now, then 9 hours later it will be 3:00. Usual addition suggests that the later time must be 6 + 9 = 15, but this is not true because clock time "wraps around" every 12 hours; there is no "15 o'clock". Likewise, if the clock starts at 12:00 noon then after 20 hours the time will be 8:00 the next day, rather than 32:00. The hour number starts all over again after it reaches 12. Therefore, this is arithmetic modulo 12.

Example: Two numbers **x** and **y** are said to be equal or congruent module **N** if their difference is exactly divisible by **n** i.e. n/(x-y).
Generally, **x** and **y** are non-negative and **N** is a positive integer. Thus, we can write
x=y (mod n)
If the difference between **x** and **y** is an integer multiple of **n**, then the number **n** is called the modulus of congruence.

2.2 Algorithmic Complexity and Time Space Tradeoff

Complexity is a measure of performance of an algorithm. The complexity of computation is a characterization of time and space requirements, which helps to solve a problem using a specific algorithm. Computational complexity is mostly concerned with the lower bound.

2.2.1 Algorithmic Complexity

We can determine the efficiency of an algorithm by calculating its performance. Following are the two factors that help us to determine the efficiency of an algorithm:

1. Total time required by an algorithm to execute.
2. Total space required by an algorithm to execute.

Thus, the two main considerations required to analyze the program are:

1. Time complexity
2. Space complexity

The amount of computer time required to solve a problem is the time complexity of an algorithm and the amount of memory required to compute the problem is the space complexity of an algorithm.

Time Complexity

Time complexity of an algorithm is the amount of time required by an algorithm to execute. It is always measured using the frequency count of all important statements or the basic instructions. This is because the clock limitation and multiprogramming environment makes it difficult to obtain a reliable timing figure.

The time taken by an algorithm is the sum of compile time and run time. The compile time does not depend on the instance characteristics, as a program once compiled can be run many times without recompiling. Thus, only the run-time of the program matters while calculating time complexity. Let us take an example to get a clear idea of how time complexity of an algorithm is computed.

Table 2.2 shows the analysis of time complexity.

Table 2.2: Analysis of Time Complexity

Algorithm Step	Statements/Instructions
A	x = x+1
B	for a = 1 to n step 1 x = x+1 Loop
C	for a = 1 to n step 1 for b = 1 to n step 1 x = x+1 Loop

In the table 2.2:

1. In step A, there is one independent statement 'x= x+1' and it is not within any loop. Hence, this statement will be executed only once. Thus, the frequency count of step A of the algorithm is **1**.
2. In step B, there are three statements out of which 'x = x+1' is an important statement. As the statement 'x = x+1' is contained within the loop, the statement will be executed n number of times. Thus, the frequency count of algorithm is **n**.
3. In step C, the inner and outer loop runs **n** number of times. Thus, the frequency count is $\mathbf{n^2}$.

During the analysis of algorithm, the focus is on determining those statements that provide the greatest frequency count. The formulas used to calculate the steps executed by an algorithm are:

$$1 + 2 + \ldots\ldots + n = n(n+1)/2$$

$$1^2+2^2+\ldots..+ n^2 = n(n+1)(2n+1)/6$$

If an algorithm has input of size **n** and performs **f(n)** basic functions, then the time taken to execute those functions will be **cf(n)**, where **c** is a constant that depends upon the algorithm design.

The time complexity of an algorithm can be further analyzed as best case, worst case and average case time complexity.

1. In best case time complexity, an algorithm will take minimum amount of time to solve a particular problem. In other words, the algorithm runs for a short time.

Example: Bubble sort has a best case time complexity of **n**.

2. In worst case time complexity, an algorithm will take maximum amount of time to solve a particular problem. In other words, algorithm runs for a long time.

Example: Quicksort has a worst case time complexity of n^2.

3. In average case time complexity, only certain sets of inputs to the algorithm get the time complexity. It specifies the behavior of an algorithm on a particular input.

Example: Quicksort has an average case time complexity of **n * log(n)**.

In general, time complexity helps to estimate the number of functions required to solve a problem of size **n**.

Space Complexity

Space complexity is the amount of memory an algorithm requires to run. The space complexity of an algorithm can be determined by relating the size of a problem (**n**) to the amount of memory (**s**) needed to solve that problem. Thus, the space complexity can be computed by using the below two components:

1. ***Fixed Space Requirement:*** It is the amount of space acquired by fixed sized structure, variables, and constants.
2. ***Variable Space Requirement:*** It is the amount of space required by the structured variables, whose size depends on particular problem instance.

Therefore, to calculate the space complexity of an algorithm we have to consider the following two factors:

1. Constant characteristics
2. Instant characteristics

Thus, the space requirement S(p) is given as:

$$S(p) = C + Sp$$

Here, **C** is the constant (required fixed space) and **Sp** is the space that depends on a particular instance of variables.

Let us take an example to determine the space complexity of the variables used in a program.

Example:

```
Algorithm: To compute the sum of three elements
//Input: x, y, and z are of integer type
Input x, y, z
//Output: The sum of three integers is returned
return x+y+z
Thus, if each of the input elements occupies 2 bytes of memory space,
then the inputs x, y, z will require a total memory size of 6 bytes.
```

In general, space complexity helps to define the amount of memory required to solve a particular problem.

Task Write a searching algorithm and find out the best, worst, and average case time complexity of that algorithm.

Time Space Tradeoff

Most of the algorithms are constructed to work with inputs of arbitrary length. Usually, the efficiency of an algorithm is stated as a function relating to time complexity or space complexity.

Time space tradeoff in context with algorithms relates to the execution of an algorithm. The execution of an algorithm can be done in a short time by using more memory, because execution time increases with less memory. Therefore, proper selection of one alternative over the other is the tradeoff.

Problems like sorting or matrix-multiplication have many choices of algorithms. Some of the choices are extremely space-efficient and some are extremely time-efficient. Research in time-space tradeoff lower

bounds seeks to prove that for certain problems, no algorithms exist that achieve lesser time complexity and space complexity simultaneously.

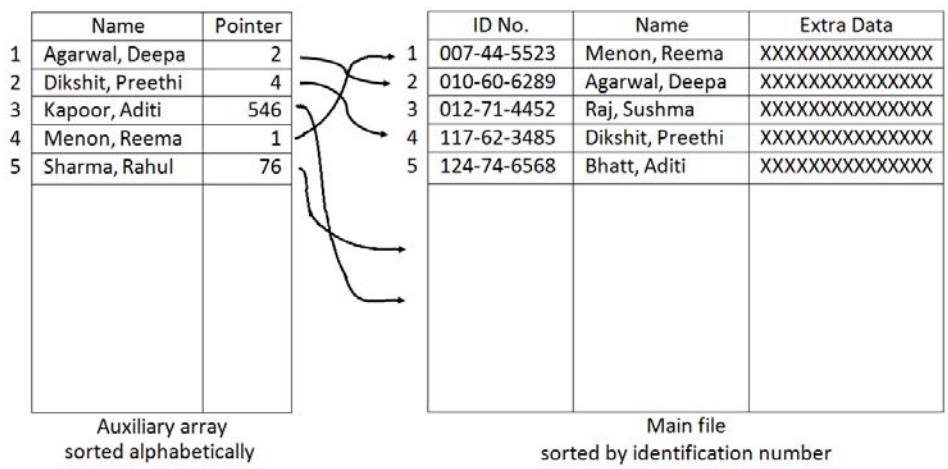
Let us take an example to understand time space tradeoff more clearly.

Example: Assume that you are given a file of records which contains names, social security numbers, and other additional information among its fields. You can easily sort the file alphabetically and run a binary search to find the record for a given name. Suppose you are given only the social security number of a person and you are required to find the record for a given name. Then to solve such a problem, you can create a file which will be sorted numerically according to the social security number. But, this process increases the space required for sorting the data.

Another way is to have the main file sorted numerically by social security number and to have an auxiliary array with only two columns as shown in figure 2.7. The first column contains an alphabetized list of the names and the second column contains pointers which give the locations of the corresponding records in the main file. This method is used frequently, since the required additional space is minimal when compared to the amount of extra information it provides.

Figure 2.7 shows Time-Space tradeoff.

Figure 2.7: Time Space Tradeoff



Source: Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw-Hill.

2.3 Algorithmic Analysis

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big-O notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

2.3.1 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size **n**. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

To analyze the best case efficiency, we have to first determine the kind of inputs for which the count **C(n)** will be the smallest among all possible inputs of size **n**. Then, we ascertain the value of **C(n)** on the most convenient inputs.

Example: In case of sequential search, the best case for lists of size **n** is when their first elements are equal to the search key. Then,

$$C_{best}(n) = 1$$

Notes Best case does not mean the smallest input. It means the input of size **n** for which the algorithm runs the fastest.

Average Case Analysis

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size n to analyze the average case efficiency of algorithm.

Example: Assume that in case of sequential search, the probability of successful search is equal to t i.e. $0 \le t \le 1$, and the probability of the first match occurring in the **i**th position of the list is the same for all values of **i**. From these assumptions we can easily find out the average number of key comparisons C_{avg} (n).

In case of successful search, the probability of the first match occurring in the **i**th position of the list is $\frac{t}{n}$ for all values of **i** and the comparison made by the algorithm is also **i**.

In case of unsuccessful search, the number of comparison is **n** with the probability of $(1-t)$. Therefore, we can write:

$$C_{avg}(n) = [1.\frac{t}{n} + 2.\frac{t}{n} + ... + i.\frac{t}{n} + ... + n.\frac{t}{n}] + n.(1-t)$$

$$= \frac{t}{n}[1+2+...i+...+n] + n(i-t)$$

$$= \frac{t}{n}\frac{n(n+2)}{2} + n(1-t)$$

$$= \frac{t(n+1)}{2} + n(1-t)$$

For t=1, the average number of key comparisons made by sequential search is $\frac{(n+1)}{2}$ which means the algorithm inspects on an average about half of the

list's elements.

For t=0, the average number of key comparisons is **n** which means the algorithm inspects all **n** element on all such inputs.

Notes

The investigation of average case efficiency is more difficult compared to the best case efficiency and worst case efficiency. The direct approach for computing the average case efficiency involves the divison of all instances of size **n** into several classes. Thus, for each instance of the class, the number of times the algorithm's basic operation executed is same.

Worst Case Analysis

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size **n**. The algorithm runs the longest among all the possible inputs of the similar size because of this input of size **n**.

To determine the worst case efficiency of an algorithm, we have to analyze the algorithm to identify the kind of input suitable for the largest value of the basic operation's count *C*(n) among all possible inputs of size **n**. Then, we can compute the worst case value $C_{worst}(n)$.

Example:

In case of sequential search, if the search element key is present at the $\mathbf{n^{th}}$ position of the list, then the basic operations and time required to execute the algorithm is more. Thus, it gives the worst case time complexity. Worst case time complexity is represented as:

$$Cworst(n)=n$$

Worst case efficiency guarantees that for any instance of size **n**, the running time will not exceed $C_{worst}(n)$, the running time on the worst-case inputs.

2.4 Summary

- A computer program is written as a sequence of steps that needs to be performed to obtain the desired result.
- Mathematical notation is a system of symbolic representations of mathematical objects and ideas.
- Some of the mathematical functions are floor and ceiling functions, summation symbol, factorial, Fibonacci numbers, and so on.
- The complexity of an algorithm is a function which describes the efficiency of an algorithm in terms of the amount of data the algorithm must process.
- The efficiency of each algorithm can be checked by computing its time complexity.
- The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as fastest possible, slowest possible, or average possible.
- The floor and ceiling functions give the nearest integer up or down.
- In the Fibonacci sequence, after the first two numbers, i.e., **0** and **1** in the sequence, each subsequent number in the series is equal to the sum of the previous two numbers.
- Analysis of an algorithm is required to determine the amount of resources such as, time and storage required to execute the algorithm.

- Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.
- The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

2.5 Keywords

Lower Bound: A mathematical argument which means that you can't hope to go faster than a certain amount.

Memory: An internal storage area in the computer.

Notation: The activity of representing something by a special system of characters.

Upper Bound: A number equal to or greater than any other number in a given set.

2.6 Self Assessment

1. State whether the following statements are true or false:
   (a) An algorithm is a set of instructions that performs a particular task.
   (b) The efficiency of each algorithm can be checked by computing its space complexity.
   (c) Summation is the operation of combining a sequence of numbers using addition.
   (d) Factorial function means to multiply a series of descending natural numbers.
   (e) Time space tradeoff in context with algorithms relates to the execution of an algorithm.
   (f) Variable space requirement is the amount of space acquired by fixed size structure, variables and constants.
2. Fill in the blanks:
   (a) The …… help to represent the time complexity in a shorthand way.
   (b) …… is generally used to express an ordering property among the functions.
   (c) A good algorithm must have …… number of space complexity.
   (d) The ……describes the way algorithm performs in the best case time complexity.
   (e) If an algorithm takes maximum amount of time to execute a specific set of input, then it is called the ……
3. Select a suitable choice for every question:
   (a) Which of the following is used to express the space complexity of an algorithm?
      (i) Theta notation (ii) Big-O notation (iii) Omega notation (iv) Factorial function
   (b) Which of the following provides an algorithm's behavior on a typical or random input?
      (i) Best case analysis (ii) Average case analysis (iii) Floor function (iv) Ceiling function
   (c) Which of the following is called a greatest integer function?
      (i) Ceiling function (ii) Floor function (iii) Modular arithmetic (iv) Factorial
   (d) Which of the following is an important part of computational complexity theory?
      (i) Algorithm analysis (ii) Floor function (iii) Ceiling function (iv) Time space tradeoff
   (e) Which of the following can be defined as "$f(n) \geq c * g(n)$"?
      (i) Big-O notation (ii) Theta notation (iii) Omega notation (iv) Floor function

2.7 Review Questions

1. "Mathematical notation is a system of symbolic representations of mathematical objects and ideas." Discuss.
2. "To select the best algorithm, it is necessary to check the efficiency of each algorithm." Justify.
3. "Big-O notation describes the performance or time complexity of an algorithm." Comment.
4. "The omega notation can be defined as $f(n) \geq c * g(n)$." Describe.
5. "Floor function gives the largest integer lesser than or equal to **x**." Describe with an example.
6. Describe modular arithmetic with the help of a 12 hour clock.
7. "Efficiency of an algorithm can be determined by calculating its performance." Comment.
8. "Time complexity of an algorithm is the amount of time required by an algorithm to execute." Discuss with an example.
9. "Time space tradeoff in context of algorithms relates to the execution of an algorithm." Comment.
10. "Analysis of an algorithm is required to determine the amount of resources it requires." Discuss.
11. "The best case efficiency of an algorithm is the efficiency of the algorithm for the best case input of size **n**." Discuss with an example.
12. "Complexity is a measure of the performance of an algorithm." Comment.

Answers: Self Assessment

1. (a) True (b) False (c) True (d) True (e) True (f) False
2. (a) Asymptotic notations (b) Big-O notation (c) Minimum
   (d) Omega (e) Worst case time complexity
3. (a) Big-O notation (b) Average case analysis (c) Floor function
   (d) Algorithm analysis (e) Omega notation

2.8 Further Readings

Books

Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw-Hill.

Reddy, P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www-old.oberon.ethz.ch/WirthPubl/AD.pdf

http://msdn.microsoft.com/en-us/library/06bkb8w2%28v=vs.71%29.aspx

http://www.cplusplus.com/doc/tutorial/variables/

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-1-administrivia-introduction-analysis-of-algorithms-insertion-sort-mergesort/lec1.pdf

Unit 3: Arrays

CONTENTS

Objectives

Introduction

3.1 Fundamentals of Arrays

3.2 Types of Arrays

3.2.1 Linear Array

3.2.2 Multidimensional Array

3.3 Types of Array Operations

3.3.1 Adding Operation

3.3.2 Sorting Operation

3.3.3 Searching Operation

3.3.4 Traversing Operation

3.4 Summary

3.5 Keywords

3.6 Self Assessment

3.7 Review Questions

3.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Recall the fundamentals of arrays
- Explain the types of arrays
- Describe the types of array operations

Introduction

A data structure consists of a group of data elements bound by the same set of rules. The data elements also known as members are of different types and lengths. We can manipulate data stored in the memory with the help of data structures. The study of data structures involves examining the merging of simple structures to form composite structures and accessing definite components from composite structures. An array is an example of one such composite data structure that is derived from a primitive data structure.

Did you know? APL (named after the book A Programming Language), designed by Ken Iverson, was the first programming language to provide array programming capabilities.

An array is a set of similar data elements grouped together. Arrays can be one-dimensional or multidimensional. Arrays store the entries sequentially. Elements in an array are stored in continuous locations and are identified using the location of the first element of the array.

3.1 Fundamentals of Arrays

An array is a data type, much like a variable as both array and variable hold information. However, unlike a variable, an array can hold several pieces of data called elements. Arrays can hold any type of

data, which includes string, integers, Boolean, and so on. An array can also handle other variables as well as other arrays. An integer index identifies the individual elements of an array.

Declaring an Array

An array is declared before it is accessed. The syntax for accessing an array component is:

data-type array_name[size];

Here, **data-type** can be of various types like int, float, or char. The array name defines the name of the array and size defines its element storage capacity. In C, the array index starts from 0.

Example: Declaring an array

int a[14];

Here, 14 memory locations are reserved for the variable **a** where the items have index ranging from 0-13. The array starts from the index 0, which is the lower bound and ends at the upper bound 13. Therefore, **a**[4] would refer to the fifth element in the array **a,** where 4 is the array index or subscript.

Name of an array without array index refers to the address of the first element.

Example: **a** or **a**[0] refers to the first element of the array **a**.

Initializing an Array

We can initialize an array by assigning values to the elements during declaration. We can access the element by specifying its index. While initializing an array, the initial values are given sequentially separated by commas and enclosed in braces.

Example: Consider the elements 10, 20, 30, and 40. The array can be represented as:
a[4]={10, 20, 30, 40}
The elements can be stored in an array as shown below:
a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
The element 20 can be accessed by referencing a[1].
Now, consider n number of elements in an array. Hence, to access any element within the array, we use a[i], where i is the value between 0 to n-1.
The corresponding code used in C language to read n number of integers in an array is:

```
for(i= 0; i<n; i++)
{
    scanf("%d",&a[i]);
}
```

Example: Consider a class of 10 students whose weight is recorded as {24, 28, 26, 30, 34, 36, 42, 44, 50, 45}. The array can be represented as a[10] = {24, 28, 26, 30, 34, 36, 42, 44, 50, 45}; and the element 26 can be accessed by referencing a[2].

Array Initialization in its Declaration

A variable is initialized in its declaration.

Example: int value = 10;
Here, the value 10 is called an initializer.

Similar to a variable, we can initialize an array at the time of its declaration. The following example shows an array initialization.

Example: int a[5] = {10, 11, 12, 13, 14};

In this declaration, a[0] is initialized to 10, a[1] is initialized to 11, and so on. There must be at least one initial value between braces. If the number of initialized array elements is lesser than the declared size, then the remaining array elements are assigned the value 0.

If we provide all the array elements during initialization, it is not necessary to specify the array size. The compiler automatically counts the number of elements and reserves the space in the memory for the array.

Example: int a[] = {10, 20, 30, 40};
Here the compiler reserves four spaces for array a.

Caution

The number of values specified during initialization cannot exceed the array size. If too many initial values are specified, a syntax error occurs.
int weight[6] = {45, 48, 54, 58, 59, 62, 33, 21,43, 19, 4, 77};
Here, the array size is 6 but the number of initial values is 12. This leads to a syntax error.

Notes

1. While defining an array size, it is better to use a symbolic constant than specifying a fixed quantity.

   # define GVAL 30 //GVAL is the maximum size of the array which is 30

   int p[GVAL]; //Declaring the array p to GVAL

2. A string consists of characters enclosed within double quotes. They are considered as an array of characters. A string is terminated by a null character '\0'. String input is read using either scanf() or gets().

Example: In a string array, a user can enter any number of characters including blank characters. Each string is terminated by a null character.

Consider the declaration:
Char str[GVAL];

If scanf("%s",str) is used to read the string 'GLOBAL WARMING', only the string until the blank character (space between the words) is stored and then the string is terminated by a null character. Here, the blank character is treated as a terminator.
The following syntax is used to read the string consisting of a blank, which acts as a character.
scanf("%[^\n]",str);

3.2 Types of Arrays

The elements in an array are referred either by a single subscript or by two or more subscripts. Hence, the arrays are of two types namely, one-dimensional array and multidimensional array, based on the subscript referred. A two-dimensional array is also a type of multidimensional array. When the array is referred by a single subscript, then it is known as one-dimensional array or linear array. When the array is referred by two subscripts, it is known as a two-dimensional array. Some programming languages allow more than two or three subscripts and these arrays are known as multidimensional arrays.

3.2.1 Linear Array

A linear or one-dimensional array is a structured collection of elements (often called array elements). It can be accessed individually by specifying the position of each element by an index value.

Example: A linear array can be anything from a row of trees or a street full of lampposts. Any sequence with repeated objects or shapes forms a linear array.

Now let us see how individual elements of linear array are accessed. The syntax for accessing an array component is:

ArrayName[IndexExpression]

The **IndexExpression** must be an integer value. The integer value can be of char, short int, long int, or Boolean value because these are integral data types. The simplest form of index expression is a constant.

Example: If we consider an array number[25], then,
number[0] specifies the 1st component of the array
number[1] specifies the 2nd component of the array
number[2] specifies the 3rd component of the array
number[3] specifies the 4th component of the array
number[4] specifies the 5th component of the array
.
.
.
number[23] specifies the 2nd last component of the array
number[24] specifies the last component of the array

To store and print values from the number array, we can perform the following:

```
for(int i=0; i < 25; i++)
{
    number[i]=i;                    // Storing a number in each array element
        printf("%d", number[i]);    //Printing the value
    }
```

To store values in a **number** array we use a **for** loop. For every iteration of the **for** loop, the value of **i** is assigned to each element of the array and then the values are printed using a **printf** statement.

Each element of an array is treated as a simple variable. Each array element is declared to hold a value of integer data type.

Example:

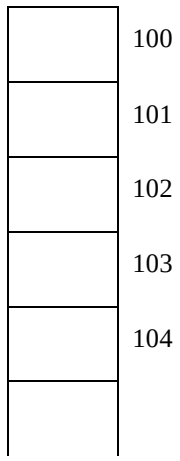
```
for(int i=0; i < 25; i++)
{                                   //Iterations
    /* Double the value in each array element and store it in the array element*/
    number[i]=2*number[i];
    printf("The output of Linear array is = %d", number[i]);
}
```

For the number array, the valid index range is from 0 to 24. Index is considered to be out-of-bounds, if the index expression results in a value less than 0 or greater than the array_size.

Representation of Linear Arrays in Memory

We know that the memory of a computer is just a sequence of address locations. Let **A** be a linear array in the memory of a computer. Figure 3.1 depicts computer memory addresses of the array **A**.

Figure 3.1: Computer Memory Addresses of Array A



The following notation is used for the address of an element of array A:

LOC (A[P]) = address of the element A[P] of the array A.

We know that the elements of **A** are stored in successive memory cells. The computer keeps track of the address of **A**'s first element and not the address of every element. This is denoted by **Base (A)** and is known as the base address of **A**. The address of any element of **A** is calculated using the following formula:

LOC (A[P]) = Base (A) + w (P – lower bound)

Here, **A** is a linear array, **w** is the size of each element of the array **A**, **LOC** is a variable used to store the location of the linear array, and **P** is the index of the element. The time required to calculate **LOC (A[P])** is fundamentally the same for any value of **P**. Suppose the array **A** has the capacity to store 4 elements. The size of the array elements is as shown below:

A[0] = 2 bytes

A[1] = 2 bytes

A[2] = 2 bytes

A[3] = 2 bytes

Hence, the memory occupied by Array **A** is 8 bytes.

Example: Consider an array **A** having a base address of 100, so Base(A)= 100.
Let us now calculate the address of A[1]. Here, index of the element is, **P**=1, size of each integer, **w**=2 and lower bound is 0.
The formula is: LOC (A[P]) = Base (A) + w (P – lower bound)
Therefore, LOC (A[1] = 100 + 2(1-0) = 102

Let us consider an example wherein we declare an array, access the array using pointers, and print the array.

Example:

```
#include<stdio.h>
int i;
void printarr(int b[])
{
        for(i=0;i<5;i++)                        //Iterations using for loop
          {
            printf("Value in the array is %d\n",b[i]);  //Printing the values in the
array
          }
}
void printdetail(int b[])
{
        for(i=0;i<5;i++)                        //Iterations using for loop
          {
   /*Printing the  values and addresses of the array elements*/
            printf("value in array is %d and its address is %8u\n", b[i],&b[i]);

          }
}

void main()
{
        int b[5];                       //Declaring the array
        clrscr();
        for(i=0;i<5;i++)                //Iterations using for loop
          {
            b[i]=i;                     //Assign the value of i to each array element
          }
    printarr(b);                        //Call the printarr function
    printdetail(b);                     //Call the printdetail function
        getch();
}
```

Output:

Value in the array is 0

Value in the array is 1

Value in the array is 2

Value in the array is 3

Value in the array is 4

Value in the array is 0 and its address is 65516

Value in the array is 1 and its address is 65518

Value in the array is 2 and its address is 65520

Value in the array is 3 and its address is 65522

Value in the array is 4 and its address is 65524

In this example:

1. First, the header files are included using #include directive and function **printarr** is defined.
2. The function **printarr(int b[])** accepts an array as a parameter and using a **for** loop, prints the values of the array.
3. The function **printdetail(int b[])**, prints the values of the array along with their addresses using **for** loop.
4. Variable **i** is declared globally.
5. Inside the **main()** function, array **b** is declared.
6. Using a **for** loop, the value of **i** is assigned to each element in an array.
7. The functions **printarr(b)** and **printdetail(b)** are called. The **getch()** function prompts the user to press a key and the program terminates.

Task

Write an algorithm to add all the elements of an array.

3.2.2 Multidimensional Array

Multidimensional arrays are also known as "arrays of arrays." Programming languages often need to store and manipulate two or more dimensional data structures such as, matrices, tables, and so on. When programming languages use two subscripts they are known as two-dimensional arrays. One subscript denotes a row and the other denotes a column.

The declaration of two-dimension array is as follows:

data_type array_name[row_size][column_size];

Example: int m[5][10]
Here, m is declared as a two dimensional array having 5 rows (numbered from 0 to 4) and 10 columns (numbered from 0 to 9). The first element of the array is m[0][0] and the last row last column is m[4][9]

Now let us discuss a three-dimensional array. A three-dimensional array is considered as an array of two-dimensional arrays.

Example: A three dimensional array is created as follows:
int bigArray [ ][ ][ ] = new int [10][10][4];
This will create an array named **bigArray** containing 400 integers. We can access any element of this array by using 3 indices.

Example: Suppose we want to assign a value 312 to the element at position 3 down, 7 across, and 2 in, then we write it as:
bigArray [2][6][1] = 312;

The general form of an n-dimensional array is as follows:

Consider an n-dimensional m1 x m2 x m3…..mn **M** containing elements m1, m2, m3,….mn. Each element is specified by a list of n integers k1, k2, k3….kn known as index.

Where,

1<=k1<=m1, 1<=k2<=m2,…….. 1<=kn<=mn.

An array **M** with index k1, k2, k3….kn is denoted by

$M_{k1, k2, k3.....kn}$ or M[k1, k2, …….kn]

A two-dimensional array **marks**[2][3] is given in the example.

Example:

```
marks[0][0]
15.5
marks[0][1]
20.5
marks[0][2]
25.5
marks[1][0]
30.5
marks[1][1]
35.5
marks[1][2]
40.5
```

The first element is given by marks[0][0] which contains 15.5, the second element marks[0][1] contains 20.5, and so on.

Initialization of Multidimensional Arrays

Like the one dimension arrays, two-dimensional arrays are also initialized by declaring a list of initial values enclosed in braces.

Example:

```
int table[2][3]={0,0,0,1,1,1};
```

The table array initializes the elements of first row to 0 and the second row to 1. The initialization is done row by row. The above statement can be equivalently written as:

```
int table[2][3]={{0,0,0},{1,1,1}}
```

Three or four-dimensional arrays are more complicated. They can also be initialized by declaring a list of initial values enclosed in braces.

Example:

```
int table[3][3][3]={1,2,3,4,5 6,7,8,…….27 };
```

This will create an array named **table** containing 27 integers. We can access any element of this array by using 3 indices.

The method to access table[1][1][1], is as shown below:

The values for array - table[3][3][3] are as follows:

```
{1, 2, 3}
{4, 5, 6}
{7, 8, 9}

{10, 11, 12}
{13, 14, 15}
{16, 17, 18}

{19, 20, 21}
{22, 23, 24}
{25, 26, 27}
```

The values in the array can be accessed using three **for** loops. The loop contains three variables i, j, and k respectively. This is as shown below:

```
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
```

```
{
for(k=0;k<3;k++)
{
printf("%d\t",table[i][j][k]);
}
printf("\n");
}
}
printf("%d", table[1][1][1]);
```

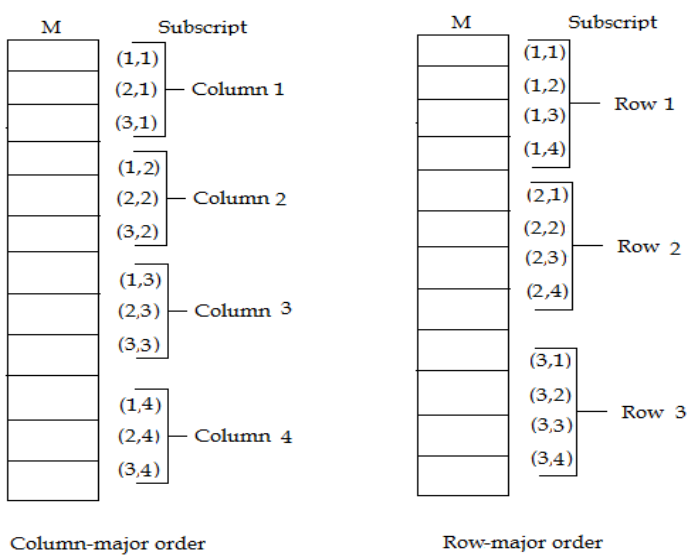
For every iteration of the i, j and k loops, the values printed are:
[0][0][0] = 1
[0][0][1] =2
[0][0][2] =3
[1][1][1] =14

Representing Two-Dimensional Arrays in Memory

Let **M** be a two-dimensional **a** x **b** array. Although, **M** is pictured as a rectangular array of elements with **a** rows and **b** columns, the array will be represented in the memory as a block of **a** x **b** sequential memory locations. Specifically, the programming language will store the array **M** either column wise or row wise. When the programming language stores an array column wise, it is known as column-major order and when it stores row wise it is known as row-major order.

The figure 3.2 depicts the two storage ways when **M** is a two-dimensional 3 x 4 array.

Figure 3.2: Two-dimensional 3 x 4 Array



Source: Lipschutz, S. Data Structures with C. Delhi: Tata McGraw-Hill. Page 4.31.

Now, consider the two-dimensional **a** x **b** array **M**. The computer keeps track of Base (A) – the address of the first element **M[1,1]** of **M** and computes the address **LOC (M[J,K])** of **M[J,K]** using the formula:

For column-major order,

LOC (M[J,K]) = Base(M) = w[P(K-1) +(J-1) --- (1)

For row-major order,

LOC (M[J,K]) = Base(M) = w[Q(J-1) +(K-1) --- (2)

Again, **w** denotes the size of memory location for each element of the array **M**. **P** and **K** denotes the row major order and column major order respectively for the array **M.**

Notes

In the two-dimensional a x b array **M**, the formulas are linear in **J** and **K**, and the address LOC (M[J,K]) is time independent of **J** and **K**.

Example:

Consider that 25 students are given 4 tests. The students are numbered from 0-25 and the test score is assigned in a 25 x 4 matrix array - MARKS. Thus, MARKS[13,2] contains the marks of the second test of the 14th student.

In particular, the third row of the array,
MARKS[3, 1], MARKS[3, 2], MARKS[3, 3], MARKS[3, 4]. This gives the score for all the four tests of the third student.

Suppose Base (MARKS) = 100 and w=4 bytes, then, the program stores two-dimensional arrays using row-major order. In this case, the row-major is 4. The address of MARKS[10,2], that is the marks scored by the tenth student in the second test are as per the formula:

LOC (M[J,K]) = Base(M) + w[Q(J-1) +(K-1)]
LOC (MARKS[13,2]) = 100 + 4[4(10-1) +(2-1)]
= 100 + 4[36+1]
LOC (MARKS[13,2]) = 248

3.3 Types of Array Operations

The operations performed on an array, are:

1. Adding operation
2. Sorting operation
3. Searching operation
4. Traversing operation

3.3.1 Adding Operation

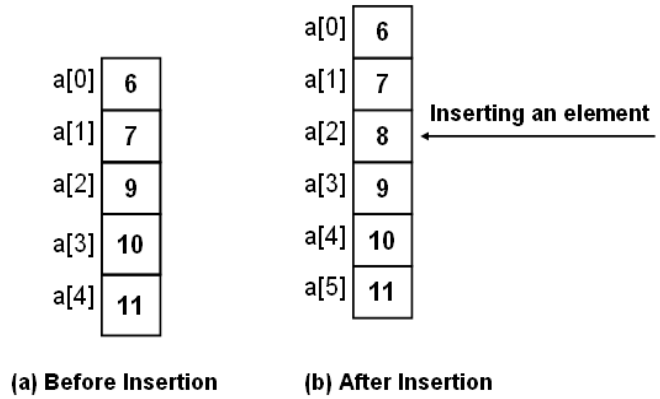
Adding elements into an array is known as insertion. The insertion of data elements is done at the end of an array. This is possible only if there is enough space in the array to add the additional elements. The elements can also be inserted in the middle of the array. Here, the average half of the array elements is moved to the next location to empty the block of memory, and to accommodate the new element.

Example:

Consider an array **a** of size 5. If you need to add an element 8 at **a[2]** position, then all the elements from **a[3]** have to be moved down (i.e. to the next location).

Figure 3.3 shows how to add elements to an array.

Figure 3.3: Adding Elements to an Array



Algorithm for Inserting an Element into an Array

Let **a** be an array of size **N** and **I** be the array index. Algorithm to insert an element in the **M**th position of the array **a** is as follows:

1. Start
2. read a[N], I<-0
3. repeat for I=N to M (Decrement I by one)
4. a[I+1]<- a[I]
5. a[M]<-ELEMENT
6. M<-M+1
7. Stop

The below program illustrates the concept of inserting an element into a one-dimensional array.

Example:

```
#include<stdio.h>
#include<conio.h>

void main()
{
int n, i, data, po_indx, a[50];       //Variable declaration
clrscr();

printf("Enter number of elements in the array\n");

/*Get the number of elements to be added to the array from the user*/

scanf("%d", &n);
printf("\nEnter %d elements\n\n", n);      //Print the number of elements
for(i=0;i<n;i++)                           //Iterations using for loop
  scanf("%d",&a[i]);                       //Accepting the values in the array

printf("\nEnter a data to be inserted\n");
  scanf("%d",&data);                       //Reads the data added by user
```

```
printf("\nEnter the position of the item \n");
  scanf("%d",&po_indx);            //Reads the position where the data is inserted

/* Checking if the position is greater than the size of the array*/
if(po_indx-1>n)
printf("\nposition not valid\n");   //If the condition is true this will be printed
else                          //If the condition is false the 'else' part will get executed
  {

            for(i=n;i>=po_indx;i--)  //Iterations using for loop
               a[i]=a[i-1];              //Value of a[i-1] is assigned to a[i]
           /*Value of data will be assigned to [po_indx-1] position*/
            a[po_indx-1]=data;
              n=n+1;                 //Incrementing the value of n

printf("\nArray after insertion\n");          //Print the array list after insertion
            for(i=0;i<n;i++)                   //Use for loop and
            printf("%d\t",a[i]);               //Print the final array after insertion
  }

 getch();                                      //Display characters on screen
}
```

Output:

```
Enter number of elements in the array
5
Enter 5 elements
15 20 32 45 62
Enter a data to be inserted
77
Enter the position of the item
2
Array after insertion
15 77 20 32 45 62
```

In this example:

1. First, the header files are included using #include directive.
2. Then, the index, array, and the variables are declared.
3. The program accepts the number of elements in the array.
4. Using a **for** loop, the values are accepted and stored in the array.
5. Then, the program accepts the **data** along with the position where it needs to be inserted.
6. If the position to be inserted is greater than the number of elements **(po_indx-1>n)** then the program displays "position is not valid". Otherwise, the program by means of a for loop, checks whether **i>=po_indx** is true and assigns the **a[i-1]** value to **a[i].**
7. Then **data** is assigned to **a**[po_indx-1].
8. Then, the program increments the number of elements and prints the array after insertion.
9. **getch()** prompts the user to press a key and the program terminates.

Task Write an algorithm to delete an element at a specific position from an array.

3.3.2 Sorting Operation

Sorting operation arranges the elements of a list in a certain order. Efficient sorting is important for optimizing the use of other algorithms that require sorted lists to work correctly.

Sorting an array efficiently is quite complicated. There are different sorting algorithms to perform the task of sorting, but here we will discuss only Bubble Sort.

Bubble Sort

Bubble sort is a simple sorting technique when compared to other sorting techniques. The bubble sort algorithm starts from the very first element of the data set. In order to sort elements in the ascending order, the algorithm compares the first two elements of the data set. If the first element is greater than the second, then the numbers are swapped.

This process is carried out for each pair of adjacent elements to the end of the data set until no swaps occur on the last pass. This algorithm's average and worst case performance is O (2n) as it is rarely used to sort large, unordered data sets.

Bubble sort can always be used to sort a small number of items where efficiency is not a high priority. Bubble sort may also be effectively used to sort a partially sorted list.

Example: Even when one element is not in order, bubble sort takes 2n of time. If two elements are not in order, bubble sort takes at most 3n time.

Algorithm for Sorting an Array

Let **A** be an array containing data with **N** elements. This algorithm sorts the elements in **A** as follows:

1. Start
2. Repeat Steps 3 and 4 for K= 1 to N-1
3. Set PTR :=1 [Initializes pass pointer PTR]
4. Repeat while PTR $\leq$ N – K: [Executes pass]

   If A[PTR] > A[PTR+1], then:

   Interchange A[PTR] and A[PTR +1]

   [End of If structure]

   Set PTR := PTR+1

   [End of inner loop]

   [End of Step 2 outer loop]
5. Exit

In the algorithm, there is an inner loop, which is controlled by the variable **PTR**, and an index **K** controls the outer loop. **K** is used as a counter and **PTR** is used as an index.

The below program illustrates the concept of sorting an array using bubble sort.

Example:

```
#include <stdio.h>
#include <conio.h>

int A[8] = {55, 22, 2, 43, 12, 8, 32, 15};  //Declaring the array with 8 elements
int N = 8;                                  //Size of the array
```

```
void BUBBLE (void);                    //BUBBLE Function declaration

void main()
{
int i;   //Variable declaration
clrscr();

/*Printing the values in the array*/
printf("\n\nValues present in array A =");
for (i=0; i<8; i++)                    //Iterations using for loop
   printf(" %d, ", A[i]);              //Printing the array

BUBBLE();                              //BUBBLE function is called

/*Printing the values from the array after sorting*/

printf("\n\nValues present in the array after sorting =");

for (i=0; i<8; i++)                    //Iterations
  printf(" %d, ", A[i]);               // Printing the array after sorting


getch();                               // waits for a key to be pressed
}
void BUBBLE(void)                      //BUBBLE Function definition

{
  int K, PTR, TEMP;                    //Declaration variables

  for(K=0; K <= (N-2); K++)            //Iterations
  {
          PTR = 0;                          //Assign 0 to variable PTR
          while(PTR <= (N-K-1-1))           //Checking if PTR <= (N-K-1-1)
          {
          /* Checking if the element at A[PTR] is greater than A[PTR+1]*/
             if(A[PTR] > A[PTR+1])
             {
                     TEMP = A[PTR];
                     A[PTR] = A[PTR+1];
                     A[ PTR +1] = TEMP;
              }
/*Increment the array index*/
              PTR = PTR+1;
          }
  }
}
```

Output:

Values present in A[8] = 55, 22, 2, 43, 12, 8, 32, 15

Values present in A[8] after sorting = 2, 8, 12, 15, 22, 32, 43, 55

In this example:

1. First, the header files are included using #include directive.
2. Then, the array **A** is declared globally along with the array elements and the size.
3. Then, inside the main function the variable **i** is declared.

4. The values in the array are printed using a **for** loop.
5. Next, the **Bubble** function is called. The sorting operation is carried out and values present in the array are printed.
6. **getch()** prompts the user to press a key. Then the program terminates.
7. In The **BUBBLE** function the variables **K, PTR and TEMP** are declared as integers.
8. **PTR** is set to 0.
9. Within the **while** loop the adjacent array elements are compared. If the element at a lower position is greater than the element at the next position, both the elements are interchanged.
10. The array index is then incremented.

Did you know? There is no algorithm that can sort **n** items in time of order less than O(n log n).

Task

Consider the array NUM[10] = {11,55,71,37,55,29,8,13, 32,6}

Write an algorithm to sort the array in descending order.

3.3.3 Searching Operation

Searching is an operation used for finding an item with specified properties among a collection of items. In a database, the items are stored individually as records, or as elements of a search space addressed by a mathematical formula or procedure. The mathematical formula or procedure may be the root of an equation containing integer variables.

Search operation is closely related to the concept of dictionaries. Dictionaries are a type of data structure that support operations such as, search, insert, and delete.

Computer systems are used to store large amounts of data. From these large amount of data, individual records are retrieved based on some search criterion. The efficient storage of data is an important issue to facilitate fast searching.

There are many different searching techniques or algorithms. The selection of algorithm depends on the way the information is organized in memory. Now, we will discuss linear searching technique.

Example: Suppose **A** is a linear array with **n** elements. If no information on A is specified, then the most spontaneous way to search for a given **ITEM** in **A** is to compare **ITEM** with each element of **A,** one by one. First, we test whether **A**[1] = **ITEM**, and then we test whether **A**[2] = **ITEM**, and so on. This method of sequentially traversing array **A** is called linear search.

Algorithm for Linear search

Let **A** be a linear array with **N** elements and **ITEM** be the given item of information. The search algorithm will find the location **LOC** of **ITEM** in **A** or sets **LOC :=0** if the search fails. The algorithm is as follows:

1. Start
2. [Insert ITEM at the end of A.] Set A[N+1] :=ITEM
3. [Initialize counter.] Set LOC :=1
4. [Search for ITEM.]
   (a) Repeat while A[LOC] ≠ ITEM:
   (b) Set LOC := LOC + 1
   [End of loop]
5. [Successful?] If LOC = N + 1, then: Set LOC := 0
6. Exit

The following example illustrates the concept of searching an element in a linear array.

Example:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

void main()
{
/* Declaring array M that can store 20 integers and the variables ele, i and num
as integer variables */

int M[20], i, ele, num;01
clrscr();                    //Clears the previously entered data
printf("\nEnter the number of elements to insert in an array: ");

/*Accepts the number of elements from the user to store in the array M*/
scanf("%d", &num);

/*Accepts input from the user until i value is less than the num entered*/
for(i=0;i<num; i++)
{
printf("\nEnter Element %d: ", i+1);
scanf("%d",&M[i]); //accepts the input from the user and stores in array M
}
printf("\nEnter the element to be searched: ");
scanf("%d", &ele);  //Accepts the element to be searched from the user
for(i=0;i<num;i++){
/* Check if the element to be searched is equal to the value stored in the array */
if(M[i] == ele){
/* Display the position of the element in the array */
printf("\nElement is found at position %d",i+1);
getch(); //wait until a key is pressed
exit(1);
}
}
printf("\nElement not found");
getch();
}
```

Output:
Enter the number of elements to insert in an array:
6
Enter element 1: 10
Enter element 2: 20
Enter element 3: 30
Enter element 4: 40
Enter element 5: 50
Enter element 6: 60
Enter the element to be searched:
40
Element is found at position 4

In this example:

1. First, the header files are included using #include directive.
2. An array named **M** with an element storage capacity of 20 is declared.
3. The variable **i**, **ele**, and **num** are declared.
4. The user enters the number of elements to be inserted in the array **M.**
5. The first **for** loop accepts the input and stores the elements in the array **M**.
6. Then, the user enters the element to be searched in the array **M.**
7. The second **for** loop checks if the condition **M[i]** is equal to the search element entered. If the condition is true, then the position of the element in the array **M** is displayed. Otherwise, the program displays a message "Element not found".
8. The **getch()** prompts the user to press any key and then the program is terminated.

Task

Write an algorithm to search an element in an array using binary search technique.

3.3.4 Traversing Operation

Traversing an array refers to moving in inward and outward direction to access each element in an array. To traverse an array, one can use **for** loop. The array elements are accessed using an array index or a pointer of type similar to that of array elements. To access the elements using a pointer, the pointer must be initialized with the base address of the array. Traversing operation also involves printing the elements in an array.

Algorithm for Traversal Operation

Let **X** be an array of size **N**. You need to traverse through the array and perform the required operations on each element of the array. Let the required operation be **OP**. Here**, i** is the array index and the lower bound starts with 0. The algorithm for traversing a given array is as follows:

1. Start
2. read X[N], i=0
3. repeat for I = 0, 1, 2…..N

   OP on X[i]
4. Stop

Example:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20                          //Define array size

void main()
{

float sum(float[], int);                 //Function declaration
float x[SIZE], Sum_total=0.0;
int i, n;                                //Variable declaration
clrscr();

printf("Enter the number of elements in array\n");
scanf(" %d", &n);                        //Reads the data added by user

printf("Enter %d elements:\n", n);       //Printing the values in the array
for(i=0; i<n; i++)                       //Iterations using for loop

/* Input the elements of the array (Traverse operation)*/

   scanf(" %f", &x[i]);
printf("The elements of array are:\n\n");  //Printing the elements of the array
for(i=0; i<n; i++)                         //Iterations using for loop

/*print the elements of array in floating point form(Traverse operation)*/

  printf(" %.2f\t", x[i]);
/*Call the function sum and store the value returned in Sum_total*/
Sum_total = sum(x, n);

/*Printing the sum*/

printf("\n\nSum of the given array is: %.2f\n", Sum_total);
getch();                                 //wait until a key is pressed
}
float sum(float x[], int n)              //Function declaration
{

int i;                                   //Variable declaration
float total=0.0;                         //the variable total is set to 0.0
for(i=0; i<n; i++)                       //Iterations
total+=x[i];                             //each element x[i] is added to the value of total
return(total);                           //Returning the total value
}
```

Output:

```
Enter the number of elements in array
5
Enter 5 elements
14 15 16 17 18
The elements of array are:
14.00 15.00 16.00 17.00 18.00
Sum of the given array is: 80.00
```

In this example:

1. First, the header files are included using #include directive.
2. Using the #define directive, the array size, SIZE, is set to 20.
3. In the **main()** function, the function sum and the variables are declared.
4. A **for** loop is used accept the elements of the array.
5. The next **for** loop prints the elements of the array.
6. The program calls the **sum()** function to add all the elements of the array. The value returned by the **sum()** function is stored in the variable **Sum_total.**
7. The program then prints the sum of the elements of the array.
8. **getch()** prompts the user to press a key to exit the program.
9. The function **sum** that accepts two arguments and returns a float value is defined. The function **sum** does the following steps:
   (a) Initializes an integer variable i.
   (b) Initializes a float variable total and assigns 0.0 to it.
   (c) Adds the elements of the array using a for loop and the result is stored in total.
   (d) Finally, it returns the value of total.

Lab Exercise

1. Write a C program to accept a two-dimensional array, print the values and sum up the elements of the array.
2. Write a C program to perform matrix addition and multiplication.

3.4 Summary

- An array is a set of same data elements grouped together. Arrays can be one-dimensional or multidimensional.
- The fundamentals of arrays include the concepts of declaring and initializing an array.
- A linear or one-dimensional array is a structured collection of elements (often called as array elements) that are accessed individually by specifying the position of each element with a single index value.
- Multidimensional arrays are nothing but "arrays of arrays". Two subscripts are used to refer to the elements.
- The operations that are performed on an array are adding, sorting, searching, and traversing.
- Adding of elements is done at the end of an array or in the middle of an array.
- Sorting operation arranges the elements of a list in a certain order.
- Searching operation is used for finding an item with specified properties among a collection of items.
- Traversing an array refers to moving in inward and outward direction to access each element in an array.

3.5 Keywords

Average Case Performance: Average case of a given algorithm denotes the average usage of the resources. Usually the resource considered is run time, but it could also be memory or any other resources.

Composite Structures: Data structures made of distinguishable data types.

Index: A non-negative integer used to identify an array element.

Worst Case Performance: Worst case of a given algorithm denotes the maximum usage of resources. Usually the resource considered is run time, but it could also be memory or any other resources.

3.6 Self Assessment

1. State whether the following statements are true or false:
   (a) The data stored in the memory can be manipulated with the help of data structures.
   (b) An array is initialized by assigning values to the elements.
   (c) When an array is referred by multiple subscripts then it is called a linear array.
   (d) Linear arrays cannot be indexed.
   (e) An array can be initialized at the time of its declaration.
   (f) Programming languages need to store and manipulate two or more dimensional data structure such as matrices.
2. Fill in the blanks:
   (a) …… condition occurs if you try to insert the data to an array when there is no space available for insertion.
   (b) Any sequence with repeated objects or shapes forms a ……
   (c) A …… character terminates every string.
   (d) …… sort is the simplest of all sorting algorithms.
   (e) …… operation involves printing each element in an array.
3. Select a suitable choice for every question:
   (a) Which of the following operations arranges the elements of a list in a certain order?
      (i) Traversing
      (ii) Declaration
      (iii) Initialization
      (iv) Sorting
   (b) Which among the following algorithms have an average and worst case performance of O(n2)?
      (i) Bubble sort
      (ii) Merge sort
      (iii) Heap sort
      (iv) Quick sort

(c) Which of the following searching techniques sequentially traverses an array to search an element?

(i) Binary search

(ii) Linear search

(iii) Quick search

(iv) Jump search

(d) What is the memory address of the first element of an array called as?

(i) Floor address

(ii) Base address

(iii) Foundation address

(iv) First address

(e) Which of the following is specified to access an element of an array?

(i) Syntax

(ii) Lower bound

(iii) Index

(iv) Upper bound

3.7 Review Questions

1. "Elements in an array are stored in continuous locations and are identified using the location of the first element." Discuss.
2. "Any element can be accessed by specifying the index of the element." Comment.
3. "If the number of initial values in an array is less than the actual array size then, the remaining array elements will be initialized to zero." Discuss.
4. "Multidimensional arrays are nothing but arrays of arrays." Comment.
5. "Linear arrays can be indexed." Analyze.
6. "To use the values stored in the number array, we can treat each array element as a simple variable of data type int." Comment.
7. "If too many initial values are specified, a syntax error will occur." Explain.
8. "Efficient sorting is important for optimizing the use of other algorithms that require sorted lists to work correctly." Discuss.
9. "Bubble sort is the simple sorting technique among all sorting techniques." Comment.
10. "Efficient storage of data facilitates fast searching. "Analyze.
11. "Traversing operation also involves printing each and every element in an array. "Comment.
12. "Insertion of data elements is usually done at the end of an array." Analyze.

Answers: Self Assessment

1. (a) True (b) True (c) False (d) False (e) True (f) True
2. (a) Overflow (b) Linear array (c) Null (d) Bubble (e) Traversing
3. (a) Sorting (b) Bubble sort (c) Linear search (d) Base address (e) Index

3.8 Further Readings

Books

Lipschutz, S. Data Structures with C. Delhi: Tata McGraw-Hill.

Reddy, P. (2009). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www.exforsys.com/tutorials/c-language/c-arrays.html

http://www.fearme.com/misc/alg/node19.html

http://cboard.cprogramming.com/c-programming/52894-traversing-array.html

Unit 4: Pointers

CONTENTS

Objectives

Introduction

4.1 Fundamentals of Pointers

4.2 Operations on Pointers

4.3 Dangling Pointers

4.4 Pointers to Functions

4.5 Pointers and Arrays

4.5.1 Array of Pointers

4.6 Records and Record Structures

4.6.1 Indexing Items in a Record

4.7 Representation of Records in Memory - Parallel Arrays

4.7.1 Variable Length Records

4.8 Summary

4.9 Keywords

4.10 Self Assessment

4.11 Review Questions

4.12 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the fundamentals of pointers
- Analyze the operations on pointers
- Explain dangling pointers
- Discuss the concept of pointers to functions
- Describe pointers and arrays
- Explain records and record structures
- Analyze the representation of records in memory - parallel arrays

Introduction

In C language, pointers are one of the strongest and the most powerful feature as they allow dynamic allocation of memory. Pointers are variables that point to data items. They store the reference of another variable. Although pointers provide flexibility while structuring data, you need to use pointers with caution as they can introduce bugs. These bugs cause difficulty while debugging a program.

You can use pointers to write compact codes, thereby reducing the program length and complexity. The method of passing the address of an argument in the calling function to a corresponding parameter in the called function is called "pass by reference". This method helps to access the data faster and supports dynamic allocation of memory. It can be used to access the byte or word locations and CPU registers. Dangling pointer is a pointer that points to a deleted object. Uninitialized pointers having

invalid addresses might cause the system to crash. Pointers used incorrectly can cause bugs and is difficult to identify.

4.1 Fundamentals of Pointers

The address which locates a variable within the memory is a reference to that variable and can be obtained by preceding the variable name with an ampersand (&) or the reference operator.

Example:

```
x = & a;
```

This would assign the address of **a** to **x**. Since, **&** is a reference operator, it stores the address of the memory and not the content of the variable.

Example:

```
a= 30;
b= a;
x= &a;
```

Here, **b** will contain the value of **a**, whereas **x** will contain the address of **a**.

Pointer can be declared as:

type * variablename;

Here, **type** is the data types of the value like int, char, and float that the pointer points to.

Example:

```
int *a;
char *name;
float *username;
```

*variablename is used to point to a value that the pointer will point.
An asterisk (*) acts as a dereference operator.

Example:

```
void main ()
{
   int iv, fv;
   int * m;
   m= & iv;
   *m= 5;
   m = &fv;
   *m= 10;
    printf( " Initial value is %d\n ", &fv;)
    printf( " Final value is %d\n", &sv;)
}
```

Output:
Initial value is 5
Final value is 10
In this example:

1. First, the address of **iv** is assigned to **m** using the reference operator (&).
2. Then, the value 5 is assigned to the memory location pointed by **m** because at this moment, m is pointing to the location of **iv**, which modifies the value of **iv**.
3. Then, the address of **fv** is assigned to **m** using the reference operator (&).
4. Finally, the value 10 is assigned to the memory location pointed to by **m**, i.e., **fv,** using the dereference operator (*).

A pointer to pointer can be declared as:

int ** x;

int can be accessed by dereferencing the pointer.

Example: j= **x; //assign an integer to j

Consider the following:

int b = 5;

int *c = &b

int ** d = &c;

Here, both **c** and **d** contain the address of an int, while **d** contains the address of a pointer to an int. There are three ways to update the value of variable **b** to 10. They are:

a= 10;

or

*b= 10;

or

**c=10;

All of them will set the value 10 to the variable **b**.

typedef keyword can be used to assign name to type definition and use the type name to declare the variables. Syntax for defining names using **typedef** in pointers is shown below:

typedef known_type_definition new_type_name

Here, **known_type_definition** is the data type like, int, char, and float.

new_type_name is the new variable name

Example: typedef int *Inpt
Inpt pointer1, pointer2;
The new type name is used to declare pointer variables of type pointer to integers.

Did you know? Address is a machine-level reference to a location in memory space of a process.

Notes

1. "&" is the reference operator and is read as "address of."
2. "*" is the dereference operator and is read as "value pointed by."
3. A variable name referenced with "&" can be dereferenced with "*".

When a pointer is declared, C language allocates space for the pointer. There are many functions which can be used for dynamic memory allocation and deallocation. They are:

1. ***Malloc():*** This function allocates a block of memory. A pointer of type void is returned when a block of memory with a specific size is reserved. This function can then be assigned to any type of pointers.
2. ***Calloc():*** This function allocates same sized multiple blocks of storage, when required.
3. ***Realloc():*** This function moves reserved block of memory to another location of different dimensions. This is used to change the memory allocated by functions, like calloc and malloc, when the memory is either insufficient or is in excess.
4. ***Free():*** This function releases a previously used block of memory.

Notes

Malloc() function is used when a pointer has to point to a block of memory and calloc() is used to request the memory space for an array which is initialized with zero-value blocks.

Consider the following pointer declaration:

char *y;

The variable type for **y** is declared to be of pointer to **char**. The pointer **y** must be initialized at the time of its declaration to avoid the chances of random value being used as memory address:

char *y = NULL;

4.2 Operations on Pointers

We can use pointer variables for various types of expressions. The table 4.1 specifies the operators which can be used using pointers.

Table 4.1: Pointer Operations

Operators name	**Symbols**	**Examples (x and y are pointer variables)**
Relational operator	> >= == < <= !=	x>y x>=y x==y x<y x<=y x !=y
Increment operator	++	x++
Decrement operator	--	y--

The above mentioned operators such as, >=, <=, >, <, ==, !=, ++, and -- can be applied to pointers only when both operands are pointers.

Example:

```
int *a, *b;
If (a >= b)
{

}
```

The preceding example is valid because both **a** and **b** are pointers. However, the next example is not valid because 50 is a numeric constant and not a pointer type.

Example:

```
int *a;
If (a > 50)
{

}
```

Table 4.2 shows some of the valid and invalid operations that can be performed on the pointers.

Table 4.2: Valid and Invalid Pointer Operations

Arithmetic operators	Invalid operation	Valid operation
Addition	m1 + m2	m1+2
Subtraction	m1-m2	m1-2
Multiplication	m1*m2	*m1×*m2

Caution

Equality operator (==) and inequality operator (!=) can be applied only if one of the operand is a null pointer (NULL or '\0'.)

4.3 Dangling Pointers

A dangling or wild pointer is a pointer which does not point to a valid memory location. This means that a running process which has certain restrictions on accessing the memory location does not fall under the address space. A pointer if not handled properly produces serious bugs or a bad program. Dangling pointers can be created in many ways.

Example:

```
{
        char *p = NULL;
        {
                char a;
                p = &a;
        }
//memory location which a was occupying is released
// p is now a dangling pointer
}
```

Dangling pointer can be avoided by making **p** as a null pointer after exiting from the inner block. A dangling pointer in a program always points to a memory location outside the process space. Here, the location pointed by the dangling pointer may or may not contain a valid object. If it is modified, then the valid object's value can change unexpectedly distorting the performance of the process which owns the object. This is called memory corruption. This leads to an erratic behavior of the system which will finally crash the system.

Dangling pointers can be avoided by initializing them to "NULL" during their declaration or when they are not used in the program.

A common programming error while creating a dangling pointer is returning the address of the local variable.

Example:

```
char *pointerfunction (void)
{
        char c[ ] = "pointers and arrays";
        return c;
}
```

In this example, if it is necessary to "return" the address of c then you can declare it with the "static" storage specifier.

Dangling pointers are usually created by the combination of malloc() and free() library calls. A pointer becomes dangling when the block of memory referred to by the pointer is set free.

Example:

```
#include<stdlib.h>
{
        char *c = malloc (A_CONST);
        free( c) ;
        c = NULL;
}
```

Did you know? In Java, object reference is a pointer and an address.

4.4 Pointers to Functions

Pointers can be used in function declaration. By using pointers, a complex function can be easily represented. The variables used to invoke the called function are known as arguments or actual parameters, and the variables used in the actual definition of the called function are called dummy parameters or formal parameters. Pointers used in the function definition are classified into two groups. They are:

1. Pass by value or call by value
2. Pass by reference or call by reference

Pass by Value or Call by Value

You have seen that when a function is invoked, a link has to be established between the formal and the actual parameters. A temporary storage has to be created so that the value of the actual parameters can be stored. The formal parameters will pick up the value from the storage area. This mechanism of data transfer between actual and formal parameters allows the actual parameter values to be copied into the formal parameters. This method is known as "call by value" or "pass by value". The corresponding formal parameter will represent a local variable in the called function. The current value of the corresponding actual parameter will become the initial value of the formal parameter. The value of the formal parameter might be changed in the body of the actual parameter or subprogram by using the assignment or input statements. This will not change the value of actual parameters.

Example:

```
# include<stdio.h>
#include<conio.h>
void num(int x, int y)
{
    x = 100;
   y = 200;
}
void main()
{
   int m, n;
   m = 10;
   n = 20;
   num (m, n);
  printf("m =  %d and n = %d", m, n);
```

```
 getch();
}
```

Output:

m = 10 and b = 20

In the above example, two functions - **num()** and **main()** are used. Execution always starts from the **main()** function, where **main()** function is the calling function and **num()** is the called function. The sequence of steps are as follows:

1. Execution starts from the calling function **main()**. Since, the variables **m** and **n** are defined within the function, memory is allocated for these variables during run-time and initialized with junk values '?'. Let the addresses of the variables be 1004 and 1008.

2. After executing the statements:

   m = 10;

   n = 20;

   The values **m** and **n are** copied into the memory locations identified by **m** and **n** in the addresses 1004 and 1008.

3. Function **num()** is invoked with two parameters **m** and **n**.

4. Control is then transferred to the called function **num().**

5. Memory is allocated for the formal parameters **x** and **y**. Assuming that the address of these variables is 2000 and 2004, the values of the actual parameters 10 and 20 are copied into these locations. Note that the number and the type of the actual parameters match with the number and type of formal parameters. So, the formal parameters obtain the value of actual parameters which is 10 and 20.

6. When the statements

   x=100

   y = 200

   are executed, the earlier values 10 and 20 are replaced by 100 and 200.

7. Then, control transfers from the called function to the calling function. It points from where it was invoked earlier. Before the control is transferred to the calling function, the memory allocated for the formal parameters is de-allocated.

8. Execution stops.

Notes

Advantages of Pass by Value or Call by Value

1. In Pass by value, expressions are passed as arguments instead of passing the values or variables as arguments.
2. This protects the values of the actual variables from getting altered by the called function.

Disadvantages of Pass by Value or Call by Value

1. Pass by value technique will not allow the information to be sent back to the calling portion of the program.
2. It allows only one-way transfer of information i.e., from the calling function to the called function.

Task

Write a C program to swap two numbers using pointers.

Pass by Reference or Call by Reference

When an address is passed to a function, the parameters receiving the address will have to be pointers. The process of calling a function by using pointers to pass the address of the variable is called pass by reference or call by reference. The function called by reference changes the values of the variable used in the call.

Example:

```
#include<stdio.h>
void num(int *x, int *y)
{
  *x = 100;
  *y = 200;
}
void main()
{
  int m, n;
  m = 10;
  n = 20;
  num(&m, &n);
  printf( " m = %d and n= %d", m, n);
 getch();
}
```

Output:
m = 100 and n = 200

There is a difference in output between the above two examples. The function **num()** is called again. The output will be 100 and 200. Here, instead of passing the values of actual parameters, the addresses of actual parameters, i.e., **&m** and **&n** are passed. The addresses are copied into formal parameters. Since the formal parameters contain the addresses, they are considered as pointers. The sequences of steps are as follows:

1. Execution of the calling process i.e., the main function will start. Since, the variables **m** and **n** are defined within the function, memory is allocated for these variables during run-time and initialized with junk values '?'. Assume that the addresses of the variables are 1004 and 1008.

2. After executing the statements:

   m = 10

   n = 20

   The values 10 and 20 will be copied into the memory locations recognized by **m** and **n** in the addresses 1004 and 1008.

3. Function **num()** is invoked with two parameters **m** and **n**. Here, the addresses of **m** and **n** are passed as parameters.

4. Control is transferred to the called function **num().** Memory is allocated for the formal parameters **x** and **y**. Let the addresses of the formal parameters be 2004 and 2008. Since, the address of **m** which is 1004 and address of **n** which is 1008 are passed as parameters, these addresses are copied into the location 2004 identified by **x** and 2008 identified by **y**. Here, the formal parameters **x** and **y** contain the address of the actual parameters **m** and **n,** which means that the values of actual parameters is accessed by the formal parameters through pointers.

5. When statements

   *x = 100

   *y = 200

   are executed, the memory location pointed to by **x** i.e., contents pointing to 1004 which is 10, is replaced by 100. Similarly, the memory location pointed by **y** i.e., the contents pointing to 1008 which is 20 is replaced by 200.

6. Control is transferred from the called function to the calling function to the point from where it was called.

7. The actual parameters are indirectly changed by the formal parameters.

8. Execution stops.

Hence, the values in one function can be changed by some other functions by passing addresses, and then de-referencing the addresses using the indirection operator (*) in the called function.

Notes

Advantages of Pass by Reference or Call by Reference

1. In pass by reference, information can be sent back to the calling function through parameters.
2. A two-way transfer of information can be achieved through pass by reference.

Task

Write a program to accept X elements into an array and then compute the sum using pointers.

4.5 Pointers and Arrays

Array is a group of elements (homogenous type) which is stored in contiguous memory locations. The concept of arrays is similar to pointers. An array can be considered as an internal or hidden pointer since one element in the array is stored adjacent to another. The identifier of an array is equivalent to the address of the first element since a pointer is equal to the address of the first element which it points to.

Example:

```
int  num[20];
int *p;
```

For the above example, the following assignment operation is valid:

p = num;

After the execution of this instruction, **p** and **num** are equal and will have the same properties. Here, the only difference is that, we can change the value of the pointer **p** by another value, while the **num** will always point to the first 20 elements of type **int** which was defined. Unlike **p** (ordinary pointer), numbers in an array are considered as a constant pointer. Hence, the following is not valid.

num = p;

Since num is an array, it operates as a constant operator. Therefore, it is not possible to assign value to it.

Example:

```
#include<stdio.h>
void  main()
{
   int num[5], x;
   int *p;
   p = num;
  *p = 10;
  p++;
 *p = 20;
  p = &num[2];
 *p = 30;
  p = num + 3;
  *p = 40;
   p = num;
 *(p+4) = 50;
 for (x=0;x<5; x++)
    printf (" %d ", num[x]);
   getch();
}
```

Output: 10 20 30 40 50

In this example:

1. An array **num** is assigned to pointer **p**.
2. A value is assigned to the memory location pointed by ***p.**
3. The pointer **p** is incremented to point to the next location.
4. Steps 2 and 3 are repeated for all the values.

Bracket sign operator [ ] or dereference operator is known as offset operator. Similar to *, the offset operator dereferences the variable it follows, but also adds the number between the brackets to the address being dereferenced.

Example:

b[5] = 0; //b[offset of 5] = 0

*(b+5) = 0; //pointed by (a+5) = 0

These two expressions are equal and valid if **b** is a pointer or an array.

Three points need to be remembered while using pointers with arrays. They are:

1. Elements of an array are always stored in contiguous memory locations.

2. Incrementing or decrementing pointer variables lead to incrementing or decrementing memory location, depending on the data type of the defined pointer variable.

3. The first element of an array is always numbered as zero, which makes the last element one less than the size of the array.

4.5.1 Array of Pointers

A two-dimensional array is a collection of one-dimensional arrays. In the case of two-dimensional arrays, a single pointer can be used to point to the contiguous one-dimensional arrays. Therefore, instead of defining a two–dimensional array as;

data_type array_name[ep1][ep2]

we can define the two–dimensional array as:

data_type (*pt)[ep2];

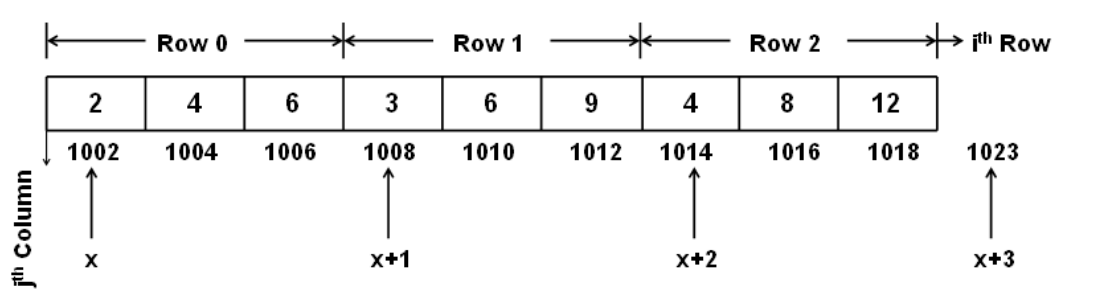
Here, **data_type** is the data type of the array, **array_name** is the name of the array, and **ep1** and **ep2** are the maximum number of elements in that row or column.

A two-dimensional vector with 3 rows and 3 columns can be initialized as:

```
int  x[3][3] = {
                {2,4,6},
                {3,6,9},
                {4,8,12}
            };
```

Figure 4.1 shows the row-wise representation of elements of the matrix **x**:

Figure 4.1: Representation of Elements in Memory



Pointers can be used to declare **x** as:

int (*x)[3];

Here, **x** is defined as a pointer to a group of one-dimensional array, each having 3 elements in an array. Initially, **x** points to the first dimensional array called the first row of matrix which is Row 0. (x+1) points to the second one-dimensional array which is Row 1. (x+2) points to the third dimensional array which is Row 2.

So, the starting address of the ith Row can be accessed using:

x+i

The entire i^{th} Row can be accessed using:

*(x+i)

The address of the first element in the jth Column can be accessed using:

*(x+i) + j

The item in the jth Column can be accessed using:

((x+i)+j)

Thus, by using &x[i][j] or *(x+i)+j address of the ith Row and j^{th} Column can be obtained and by using x[i][j] or *(*(x+i)+j) the item at the ith Row and j^{th} Column can be obtained.

A three-dimensional array of float type can be defined as:

float b[10][20][30];

A three-dimensional array using pointers can be defined as:

float (*b)[20][30];

A multidimensional array can be represented in terms of an array of pointers. The definition of a conventional array is:

data_type array_name[exp1][exp2];

Array of pointers can be used to define two-dimensional array as

data_type *array_name[exp1];

Here, **data_type** refers to the data type of an array

array_name is the name of the array

exp1 is the maximum number of elements in the row

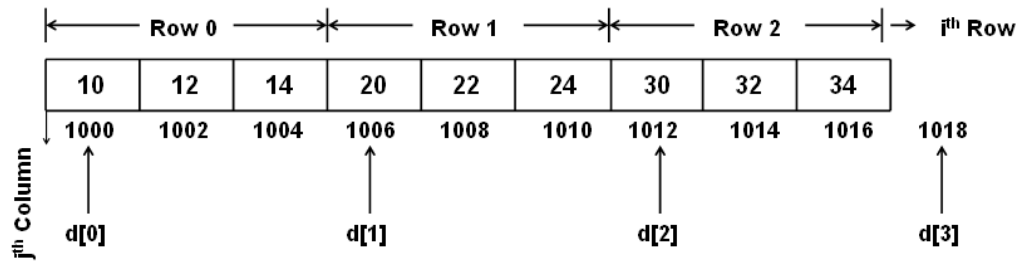
Here, **exp2** is not used while defining array of pointers.

Example: Suppose, a two-dimensional vector is initialized with 3 rows and 3 columns as

```
int d[3][3] = {
        { 10, 12, 14}
        { 20, 22, 24}
        { 30, 32, 34}
    };
```

Then, the elements of the matrix **d** are stored in memory row-wise as shown in figure 4.2:

Figure 4.2: Array of Pointers



By using array of pointers, you can declare **d** as:

int *d[3];

Here, **d** is an array of pointers, **d[0]** gives the address of Row 0, **d[1]** gives the address of Row 1, and **d[2]** gives the address of Row 2. Now, **d[0]+0** will give the address of the element in the 0th row and 0th column, **d[0]+1** will give the address of the element in the 0th row and 1st column and so on.

In general, the address of the element in the ith Row and jth Column is given by:

d[i]+j

The element in the **ith** Row and **jth** Column can be accessed using "*" (indirection operator) by specifying *(d[i]+j).

4.6 Records and Record Structures

Usually, collection of data is organized into hierarchy of fields, records, and files. A file is a collection of similar records and a record is a collection of related data items, where each data item is called a field or an attribute. Each data item can be a group item composed of sub items. The items which cannot be decomposed are called elementary items, scalars or atoms. Identifiers are used to refer to the data types, variables, and functions.

A record is a collection of data items that differs from a linear array in the following ways:

1. A record can be a collection of non-homogenous data (data items with different data types).
2. In a record, data items are indexed by attribute names. Therefore, there may not be a natural ordering of the elements.

Under the relationship of a group item to sub item, "level" numbers can be used to describe the hierarchical structure of the data items in a record, as shown in the following example:

Example: Consider a kindergarten school maintaining a record of the admission application it receives. The record contains the following data items: name, gender, date of birth, father's name, mother's name, and address. Further, date of birth is a group item with sub items month, day, and year, Father and Mother group items will have sub items Name and Age, and Address group items will have sub items Dno, Road, and Area.

The structure of the record can be as follows:

```
1 Application
    2 Name                    //data item
```

```
2 Gender
2 Birthday                //group item
   3 Month                // sub item
   3 Day
   3 Year
2 Father
   3 Name
   3 Age
2 Mother
   3 Name
   3 Age
2 Address
   3 Dno
   3 Road
   3 Area
```

The number to the left of each identifier is known as level number. Each group item is followed by its sub items. The level of sub items is 1 more than the level of the group item.

In a record structure, some of the identifiers may also be referred to as arrays of elements. In the above example, suppose the first line of the structure is replaced by

1 Application (20)

This indicates that there is a file of 20 records, and the different records in the file can be distinguished by using the usual subscript notation.

That is, we will write

$Application_1$, $Application_2$, $Application_3$

Or

Application [1], Application [2], Application [3]

to denote the different records in the file.

4.6.1 Indexing Items in a Record

Suppose, we want to access few data items in a record, then we cannot simply write the data name of the item. This is because the name might appear in different places in the record.

Example: The structure of the kindergarten application record is as follows:

```
1 Application
    2 Name
    2 Gender
    2 Birthday
       3 Month
       3 Day
       3 Year
    2 Father
       3 Name
       3 Age
    2 Mother
       3 Name
       3 Age
    2 Address
       3 Dno
       3 Road
       3 Area
```

In the above example, Gender and Year does not require any qualification as each refers to a unique item in the structure.

On the other hand, consider Age which is occurring more than once in the record. Therefore, in order to specify a particular item, we will have to qualify the name by using appropriate group names in the structure. This qualification can be done by using decimal points or periods to separate group items from sub items.

If we want to refer to the Age of the mother then, it can be done in the following way:

Application.Mother.Age or Mother.Age

This reference is said to be fully qualified. Sometimes, we can include more identifiers for clarity.

In the above example, if the first line of the record structure is replaced by:

1 Application (20)

The Application is defined to be a file with 20 records, and then every item will automatically become a 20 element array. Some languages allow the Gender in the Application to be referenced as:

Application.Gender[5] or Gender[5]

So, the name of the mother of the fifth child can be referenced by writing:

Application.Mother.Age[5] or Mother.Age[5]

Notes

Texts can use functional notation instead of dot notation to represent qualifying identifiers. For example, Age (Mother(Application)) instead of Application. Mother. Age.

4.7 Representation of Records in Memory - Parallel Arrays

Records can contain a collection of non-homogenous data. Therefore, the elements of a record may not be stored in an array. In C language, structures can be used to store such non-homogenous data records.

Example: Consider the record structure of kindergarten school Application . You can store the record in C by the following declaration, which defines the data aggregate called structure.

```
struct Application
{
    Char Name[20];
    Char Gender[1];
    struct Birthday
    {
      int Month;
      int Day;
      int Year;
    }B;
   struct Father
   {
    char Name[20];
     int Age;
    }F;
    struct Mother
    {
   char Name[20];
   int Age;
   }M;
```

```
    struct Address
    {
    int Dno;
   int Road;
   char Area;
   }A
}AN1;
```

If a programming language does not support the hierarchical structures concept, then we can assume that the record contains non-homogenous data. The record can be stored in individual variables, one for each of its elementary data items. Suppose you want to store an entire file of records, then such a file can be stored as a collection of parallel arrays. This means that elements of different arrays with the same subscript can belong to the same record. The following two examples illustrate this.

Example: Suppose, a health club membership contains the name, age, gender, and telephone number of each member. Then, you can store the file in four parallel arrays i.e., Name, Age, Gender, and Phone as shown below:

Name	Age	Gender	Phone
Rohit	20	Male	23454322
Ashwitha	25	Female	23334446
Pradeep	30	Male	23243567
Bhaskar	33	Male	23546745

Example: Consider the Kindergarten application records example. You can store the file of such records in twelve linear arrays such as, Name, Gender, Month, Day, Year, FatherName, FatherAge, MotherName, MotherAge, AddressDno, AddressRoad, AddressArea; one array for each elementary data item.

Here, you must use different variable names for the name and age of the father and mother, which was not necessary in the previous example. Again, we can assume that the arrays are parallel i.e., for a fixed subscript N, the elements Name[N], Gender[N], Month[N],……., AddressArea[N] belong to the same record.

4.7.1 Variable Length Records

Consider an elementary school which wants to keep a record of every student studying in their school. The record contains the following data:

Name, Telephone Number, Father, Mother, and Siblings.

Here, Father, Mother, and Siblings will contain the names of the student's father, mother, and brothers or sisters who attend the same school.

Two such records are represented in figure 4.3.

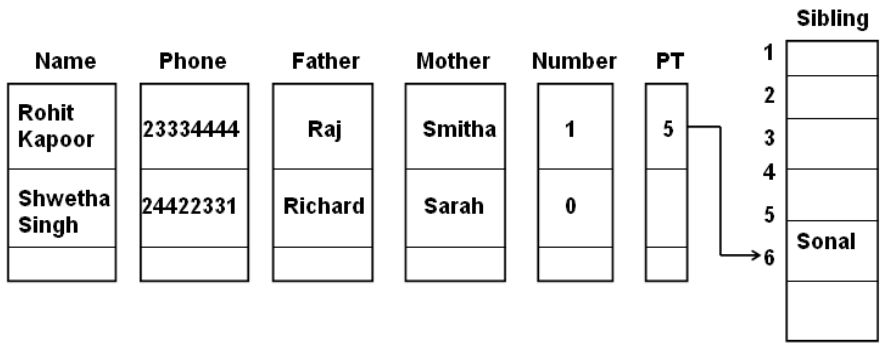
Figure 4.3: Variable Length Records

Rohit Kapoor	23334444	Raj	Smitha	Sonal
Shwetha Singh	244223311	Richard	Sarah	---------

Here, "--------"means that student has no sibling studying in the school.

This is an example of variable-length record because the data element Siblings contains zero or more names. One way of storing the file in the array is as shown in figure 4.4.

Figure 4.4: Variable Length Records



Here, the linear arrays Name, Phone, Father, and Mother are the first four data items in the records, and arrays Num and PT give the number and location of siblings in the array Sibling.

Lab Exercise

1. Write a C program using pointers that accepts two arrays of equal size and creates a third array that contains the product of the elements of the two arrays.
2. Write a C program using pointers that accepts two strings and prints the concatenation of both these strings.
3. Write a C program using pointers and arrays that accepts a string and deletes 'l' characters from the nth position of a string.

4.8 Summary

- Pointers are simple to use which help in reducing the length of the program.
- Pointer variable stores the memory address of another variable.
- Careless use of pointers may cause unexpected errors in the execution of programs.
- Pointer is a valid address stored in the pointer variable.
- Pointer variable is usually declared like an ordinary variable with * preceding each variable name.
- Pointer to pointer stores the address of the pointer.

- Memory can be allocated and deallocated dynamically by using functions such as, malloc(), calloc(), realloc(), free().
- Since pointers can produce bugs, they should be used with caution as debugging becomes complicated.
- Record is a collection of non-homogenous data.
- Structures are used to store non-homogenous data.

4.9 Keywords

Bug: A fault in the code or routine of a program

Contiguous Memory: Memory which is not fragmented into smaller blocks and arranged in different locations.

CPU Registers: Special memory locations constructed from flip-flops.

Junk Value: When no value has been assigned to a particular field in a database, the field contains a value called as junk value.

4.10 Self Assessment

1. State whether the following statements are true or false:
   (a) Pointer can be declared as int d.
   (b) Dangling pointer is a pointer which points to the data.
   (c) Memory is deallocated using malloc().
   (d) Records should not contain non-homogenous data.
   (e) Pointer operation" m1+m2" is valid.
   (f) A file is a collection of similar records.
2. Fill in the blanks:
   (a) Memory can be deallocated using ……
   (b) Pass by value allows …… transfer of data.
   (c) …… is used to allocate a block of memory.
   (d) Dangling pointer is also known as……
   (e) In pass by value, ……can be passed as arguments.
3. Select a suitable choice for every question:
   (a) Pointer to pointer is declared as:
      (i) int **p;
      (ii) int p;
      (iii) int *p;
      (iv) int p*;
   (b) Pointer is declared as:
      (i) int (p);
      (ii) int p;
      (iii) int *p;
      (iv) int **p;

(c) Two-dimensional array can be represented using pointers as:

(i) data_type(*p)[ep1];

(ii) data_type(**p)[ep2];

(iii) data_type(*p)[ep2];

(iv) data_type (*p)[*ep];

(d) In pointers, reference operator is:

(i) **&**

(ii) @

(iii) $

(iv) ^

(e) Equality operator can be applied only if:

(i) Both operators are null

(ii) One operator is null

(iii) Operators are strings

(iv) Operators are characters.

4.11 Review Questions

1. "int *a;

   int *b;

   int *c;

   c= a+b". Is this a valid operation? Justify.
2. "The address which locates a variable within the memory is a reference to that variable." Explain.
3. "There are many functions which can be used for dynamic memory allocation and deallocation." Name and explain the functions.
4. "**typedef** keyword can be used to assign name to type definition." Explain.
5. "Multidimensional array can be represented in terms of an array of pointers." Explain.
6. "Record is a collection of data items." Discuss.
7. "The function which is called by reference will change the values of the variable used in the call." Comment.
8. "Wild pointer is a pointer which does not point to a valid memory location." Discuss.
9. "A record which is a collection of data items differs from a linear array." Explain.
10. "Pointers can be used in function declaration." Describe.
11. "An array can be considered as an internal or hidden pointer." Comment.
12. "Records can contain a collection of non-homogenous data." Explain.

Answers: Self Assessment

1. (a) False (b) False (c) False (d) False (e) False (f) True
2. (a) free() (b) one-way transfer (c) malloc() (d) Wild pointer (e) Expressions
3. (a) Int *p; (b) int **p; (c) data_type(*p)[ep2]; (d) & (e) one operator is null

4.12 Further Readings

Books

Amdani, S. (2009). C Programming. New Delhi: Laxmi Publications.

Tenebaum, A. (2009). Data Structures using C. South Asia: Dorling Kindersley.

Online link

http://www.owasp.org/images/f/fa/OWASP_IL_8_Dangling_Pointer.pdf

http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

Unit 5: Introduction to Linked List

CONTENTS

Objectives

Introduction

5.1 Basics of Linked List

5.2 Representation of Linked List in Memory

5.3 Types of Linked Lists

5.3.1 Singly-Linked List

5.3.2 Doubly-Linked List

5.3.3 Circular Linked List

5.3.4 Circular Doubly-Linked List

5.4 Summary

5.5 Keywords

5.6 Self Assessment

5.7 Review Questions

5.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Understand the basics of linked list
- Analyze the representation of linked list in memory
- Discuss the types of linked lists

Introduction

Linked lists are the most common data structures. They are referred to as an array of connected objects where data is stored in the pointer fields. Linked lists are useful when the number of elements to be stored in a list is indefinite.

Example: Consider the list of five students in an attendance register - Amar, Divya, Prateek, Sunil, and Yash. If you wish to add the name "Rita" to the list, then you have to create space for the new name in the list. A linked list performs this by moving the first three names in the list to the left and the last two names to the right.

Linked lists are similar to arrays as they both are used to store a collection of data. The drawbacks of using arrays are:

1. Once the elements are stored, it becomes difficult to insert or delete an element at any position in a list.
2. Arrays have fixed size. Hence, if the memory allocated is too large than the actual data, unused portion of the memory is wasted. If the allocated memory is less, it will result in loss of data due to inadequate memory.

Did you know? Linked lists were developed in the year 1955-56 by Allen Newell, Cliff Shaw, and Herbert Simon at RAND Corporation as a primary data structure. It was developed for their Information Processing Language (IPL). IPL was used by the authors to develop several early artificial intelligent programs, including Logic Theory Machine, the General Problem Solver, and a computer chess program.

The advantage of using linked list is its ability to dynamically shrink and expand in size. This allows you to insert or delete elements efficiently at any position in the list.

There are four types of linked lists namely, singly-linked list, doubly-linked list, circular list, and header lists. Linked list is used as an Abstract Data Type (ADT) as it can store data of any type in nodes that are interconnected to each other.

5.1 Basics of Linked List

The technique of dynamically implementing a list using pointers is known as linked list. Every element in a list is represented as a node.

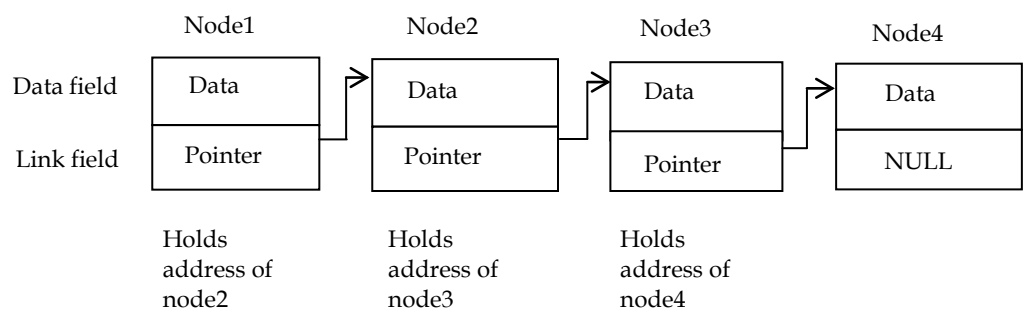
Example: A train consists of several coaches that are interconnected to one another. Here, a linked list represents a train and the coaches form the nodes in a list.

Figure 5.1 depicts a linked list consisting of four nodes.

Each node in figure 5.1 consists of two fields:

1. ***Data Field:*** It is a field that stores the element value of a specific data type in the list.
2. ***Link Field:*** It is a pointer field used to point to the next consecutive node thereby establishing a link between two nodes. For example, in figure 5.1, the pointer field of Node1 holds the address of Node2, the pointer field of Node2 holds the address of Node3, and so on.

Figure 5.1: Representation of a Linked List



Each node holds the address of the next consecutive node and the pointer points to the data item in the next node. In figure 5.1, the Node4 address is assigned a NULL value to depict the end of the list.

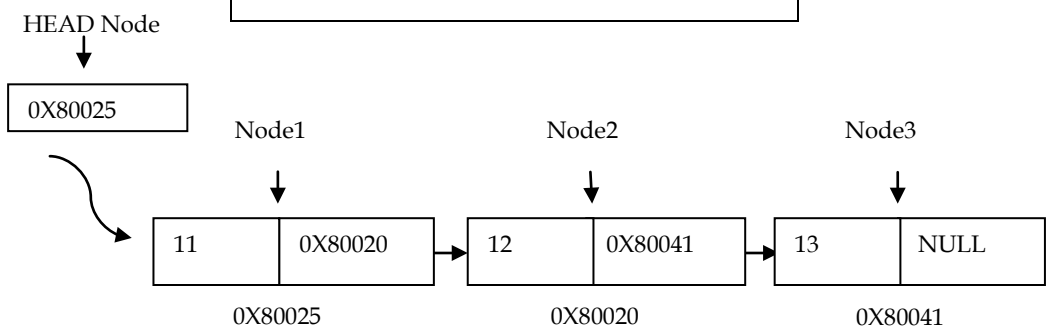
Did you know? Arrays follow a strategy of allocating memory for its elements in a single block of memory. Whereas, linked lists allocate memory for each element individually. The memory size increases as and when the data is added. This technique prevents memory wastage.

5.2 Representation of Linked List in Memory

The representation of a linked list in the memory is shown in figure 5.2. The linked list consists of three nodes. Each node contains a data field and a link field. The data field consists of any data item that is stored in a list such as, numbers, characters, strings, and so on. The link field holds the address of the next element.

In the figure 5.2, HEAD Node holds the address of Node1 (0X80020), Node1 holds the address of Node2 (0X80041), and Node2 holds the address of Node3 (Null). "NULL" denotes a null pointer which is the end of linked list or empty list.

Figure 5.2: Representation of a Linked List in Memory



A linked list can be represented in memory with the following declaration:

```
struct new_list {
        int element_value;
        struct new_list *next_element;
      }; node1, node2, node3
```

Here, a linked list named "new_list" is created. In "new_list" the data field named "element value" is declared. The "element_value" can be an integer, a character, floating point, or double type. The "new_list" also contains a link field named "*next_element" which points to the next node in the list. The end of the structure containing three nodes (node1, node2, and node3) denotes the objects. They are created to access the structure elements.

In a linked list, the HEAD node holds the address of the first node. When there are no nodes present in a list, then the HEAD node will be equal to NULL and the list is known as Empty list or NULL list.

5.3 Types of Linked Lists

The various types of linked list are:

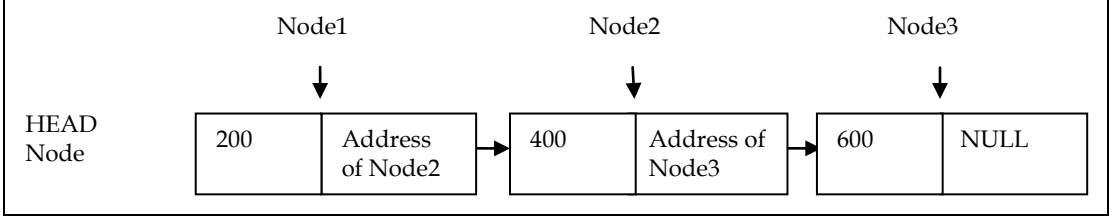
1. Singly-linked list
2. Doubly-linked list
3. Circular singly-linked list
4. Circular doubly-linked list

5.3.1 Singly-Linked List

As the name suggests, a singly-linked list consists of only one pointer that points to another node. It is also known as a linear list because the last node in a singly-linked list is assigned a NULL value and hence does not point to any other node. The first node in the list is known as a HEAD or first node.

Figure 5.3 depicts a singly-linked list. The **HEAD Node** is a dummy node pointing to Node1. Node1 holds the address of Node2, and Node2 holds the address of Node3. Node3 points to NULL to indicate that there are no additional nodes present in the list.

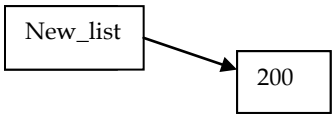
Figure 5.3: Representation of a Singly-linked List



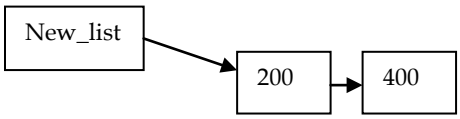
Example: The program shows the implementation of a singly-linked list consisting of four nodes. The program displays the value present in each node.

The program flow is depicted below:

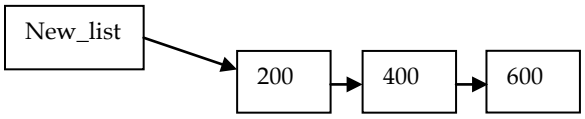
1. Creating node1 with value 200



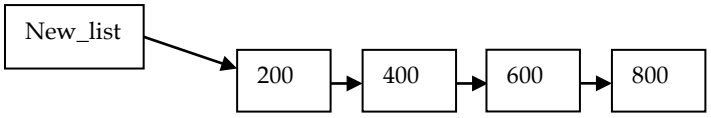
2. Creating node2 with value 400



3. Creating node3 with value 600



4. Creating node4 with value 800



```
#include<stdio.h>
struct new_list
{
        int value;
        struct new_list *next_element;
} n1, n2, n3, n4;                //Creates four nodes of type new_list

void main()
{
 int j;
n1.value = 200;                  //Assigning value to node1
n2.value = 400;                  //Assigning value to node2
n3.value = 600;                  //Assigning value to node3
n4.value = 800;                  //Assigning value to node4

n1.next_element = &n2;     //Assigning address of node2 to node1
n2.next_element = &n3;     //Assigning address of node3 to node2
n3.next_element = &n4;     //Assigning address of node4 to node3
n4.next_element = 0;       //Assigning 0 to node4 to indicate the end of the list

j = n1.next_element->value;          //Storing the value of node1 in variable j
printf("%d\n", j);
/* you can use this statement to print the value present in node1 or print j
directly as depicted in the above statement*/
printf("%d\n", n1.next_element->value);
printf("%d/n", n4.next_element->value);          //Printing the value of node4
printf("%d/n", n2.next_element->value);          //Printing the value of node2
printf("%d/n", n3.next_element->value);          //Printing the value of node3
}
```

Output:

When you run the program, the following output is obtained:

400
0
600
800

In this example:

1. First a structure named new_list is created. The list contains an integer data variable named value to store data and a pointer variable named next_element to point to next node.
2. Then, four objects namely, n1, n2, n3, and n4 are created to access the structure elements. In the program they act as nodes in a list.
3. In the main() function, the value for the four nodes n1, n2, n3, and n4 are assigned.
4. Then, the address of n2 is stored in n1, address of n3 is stored in n2, and address of n4 is stored in n3. The address of n4 is assigned zero to indicates the end of the list.
5. Finally, the value present in n1, n4, n2 and n3 are printed.

Task

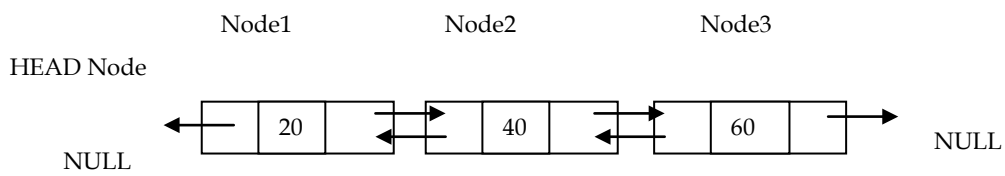
Write a simple singly linked list program to accept the elements from the user and store it in a list.

5.3.2 Doubly-Linked List

Doubly-linked list contains two pointers for each node in the list. The first pointer points to the next element and the second pointer points to the previous element. The previous pointer for the **HEAD** node points to **NULL** and the next pointer for the last node points to **NULL.** Doubly-linked list is also known as a two-way list as both forward and backward traversal is possible.

Figure 5.4 depicts a doubly-linked list. The **HEAD** Node is a dummy node pointing to Node1. Node1 has two pointers, the first pointer points to Node2 and the second pointer points to **HEAD** Node. Likewise, Node2 and Node3 also have two pointers to point to the next and the previous element in the list. The **HEAD** Node and the Node3 are assigned to **NULL.** The data field of Node1, Node2, and Node3 consists of values 20, 40, and 60 respectively. When you try to print the value of Node2's next element, the value present in Node3 which is 60, will be printed.

Fig 5.4: Representation of a Doubly-Linked List



Example: The program shows the implementation of a doubly-linked list consisting of three nodes. The program displays the value present in each node.

```
#include<stdio.h>
struct list
{
        int value;
        struct list *next;       //Creating a pointer to point to the next element
        struct list *previous;//Creating a pointer to point to the previous element

} n1, n2, n3;                    //Creating three nodes of type list

void main()
{
int j;

n1.value = 20;                   //Assigning value to node1
n2.value = 40;                   //Assigning value to node2
n3.value = 60;                   //Assigning value to node3

n1.next = &n2;                   //Assigning address of node2 to node1
n2.next = &n3;                   //Assigning address of node3 to node2
n2.previous = &n1;               //Assigning address of node1 to node2
n3.previous = &n2;               //Assigning address of node2 to node3

n3.next = 0;                     //Assigning 0 to node3 to indicate the end of the list
n1.previous = 0;                 //Assigning 0 to node1 to indicate there are no elements
present before node1

j = n1.next->value;              //Storing the value of node1 in variable j
```

```
printf("%d\n", j);
//printf("%d\n", n1.next->value); // you can use this statement to print the value
present in node1 or print j directly as depicted in the above statement

printf("%d/n", n1.next->value);          //Printing the next value of node1
printf("%d/n", n2.next->value);          //Printing the next value of node2
printf("%d/n", n1.previous->value);      //Printing the previous value of node1
printf("%d/n", n2.previous->value);      //Printing the previous value of node2
printf("%d/n", n3.previous->value);      //Printing the previous value of node3
}
```

Output:
When you run the program, the following output is obtained:
40
60
0
20
40

In this example:

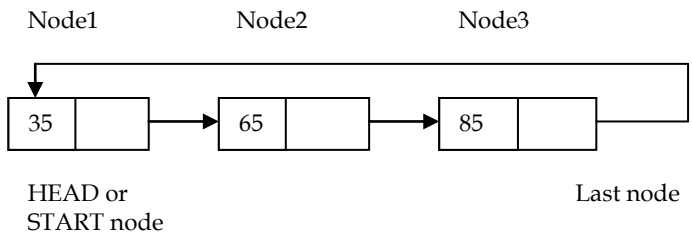
1. First, a structure named l**ist** is created. The list contains an integer data variable named **value** to store data**,** a pointer variable named **next_element** to point to next node and a pointer variable named **previous_element** to point to previous node.
2. Then, the three objects namely, **n1, n2,** and **n3** are created to access the structure elements. In the program they act as nodes in a list.
3. In the **main()** function, the value for nodes **n1, n2,** and **n3** are assigned.
4. Then, the address of **n2** is stored in **n1** and address of **n3** is stored in **n2**. In order to traverse backwards, the address of **n1** is stored in **n2** and address of **n2** is stored in **n3**. The address of **n3** is assigned a **NULL** value to depict the end of the list.
5. Finally, the values present in **n1**, **n2**, and **n3** are printed.

5.3.3 Circular Linked List

In a circular linked list, only one pointer is used to point to another node or next element. It is known as a circular list because the last node's pointer points to the HEAD node.

Figure 5.5 depicts a circular linked list. The linked list consists of four nodes like, Node1, Node2, and Node3 with values 35, 65, and 85 respectively. The last node which is Node3 points to the first node (Node1) and hence, the list continues to form a loop. When you try to print the value of Node3's next element the value present in Node1, which is 35, will be printed.

Figure 5.5: Representation of a Circular Linked List



Example:

The program shows the implementation of a circular linked list consisting of three nodes. The program displays the value present in each node.

```
#include<stdio.h>
Struct list
{
        int value;
        struct list *next_element;
} n1, n2, n3;                    //Creates four nodes of type new_list

void main()
{
 int j;

n1.value = 35;                   // Assigning value to node1
n2.value = 65;                   // Assigning value to node2
n3.value = 85;                   //Assigning value to node3

n1.next_element = &n2;           //Assigning address of node2 to
node1
n2.next_element = &n3;           //Assigning address of node3 to
node2
n3.next_element = &n1;           //Assigning address of node3 to
node1

j = n1.next_element->value;      //Storing the value of node1 in
variable j
printf("%d\n", j);               //Printing the value of j
/* you can use this statement to print the value present in node1*/
printf("%d\n", n1.next_element->value);

printf("%d/n", n2.next_element->value);      //Printing the value of
node2
printf("%d/n", n3.next_element->value);      //Printing the value of
node3
}
```

Output:

When you run the program, the following output is obtained:

65
65
85
35

In this example:

1. First, a structure named l**ist** is created. The **list** contains an integer data variable named **value** to store data and a pointer variable named **next_element** to point to next node.
2. Then, the three objects namely, **n1, n2,** and **n3** are created to access the structure elements. In this program, these objects act as nodes in a list.
3. In the **main()** function, the value for nodes **n1, n2** and **n3** are assigned.
4. Then, the address of **n2** is stored in **n1** and address of **n3** is stored in **n2**. Since, it is a circular list, the address of **n3** is assigned to **n1** instead of NULL value.
5. Finally, the values present in n1, n2, and n3 are printed.

Task

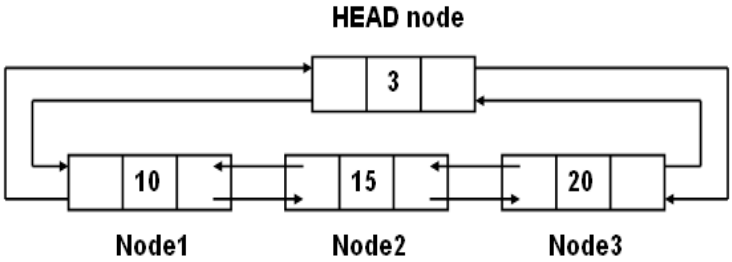
Write a simple circular linked list program to accept the elements from the user and store it in a list.

5.3.4 Circular Doubly-Linked List

In a circular doubly-linked list, the previous pointer of the first node and the next pointer of the last node point to the **HEAD** node. The **HEAD** node can have a dummy data or it can store the total number of nodes present in the list.

Figure 5.6 depicts a circular doubly-linked list. The linked list consists of four nodes such as, **HEAD** node, Node1, Node2, and Node3 with values 3, 10, 15 and 20 respectively. Each node has two pointers to point to the next and previous elements. The last node (Node3) points to the **HEAD** node and the **HEAD** node in turn points to the first node (Node1).

Figure 5.6: Representation of a Circular Doubly-Linked List



Example: The program shows the implementation of a circular doubly-linked list consisting of three nodes and a HEAD node. The program displays the value present in each node.

```
#include<stdio.h>
struct list
{
        int value;
        struct list *next;        //Creating a pointer to point to the next element
        struct list *previous;  //Creating a pointer to point to the previous element
} n1, n2, n3, h;                //Creates four nodes of type list

void main()
{
 int j;

n1.value = 10;              //Assigning value to node1
n2.value = 15;              //Assigning value to node2
n3.value = 20;              //Assigning value to node3
h.value = 3;                //Assigning value to HEAD node
n1.next = &n2;              //Assigning address of node2 to node1
n2.next = &n3;              //Assigning address of node3 to node2
n3.next = &h;               //Assigning address of HEAD node to node3
h.next = &n1;               //Assigning address of node1 to HEAD node

n1.previous = &h;           //Assigning address of node1 to HEAD node
n2.previous = &n1;          //Assigning address of node1 to node2
n3.previous = &n2;          //Assigning address of node2 to node3
```

```
h.previous = &n3;            //Assigning address of node3 to HEAD node

j = n1.next_element->value;            //Storing the value of node1 in variable j
printf("%d\n", j);
//printf("%d\n", n1.next->value);   // you can use this statement to print the value
present in node1 or print j directly as depicted in the  above statement
printf("%d/n", n2.next->value);            //Printing the value of node2
printf("%d/n", n3.next->value);            //Printing the value of node3
printf("%d/n", h.next->value);             //Printing the value of HEAD node
printf("%d/n", n1.previous->value);      //Printing the previous value of node1
printf("%d/n", n2.previous->value);      //Printing the previous value of node2
printf("%d/n", n3.previous->value);      //Printing the previous value of node3
printf("%d/n", h.previous->value);       //Printing the previous value of HEAD
node
}
```

Output:
When you run the program, the following output is obtained:
15
20
3
10
3
10
15
20

In this example:

1. First, a structure named **list** is created. The **list** contains an integer data variable named **value** to store data**,** a pointer variable named **next_element** to point to next node, and a pointer variable named **previous_element** to point to previous node.
2. Then, the four objects namely, **n1, n2, n3,** and **h** are created to access the structure elements. In this program, these objects act as nodes in a list. The HEAD node **(h)** contains the total number of nodes present in the list.
3. In the **main()** function, the value for the nodes **n1, n2, n3,** and **h** are assigned.
4. Then, the address of **n2** is stored in **n1** and the address of **n3** is stored in **n2**. In order to traverse backwards, the address of **h** is stored in **n3** and address of **n1** is stored in **h**.
5. Finally, the values present in **n1**, **n2**, **n3,** and **h** are printed.

Lab Exercise

Write a C program to store 20 integers in descending order in a linked list.

5.4 Summary

- Linked list is a technique of dynamically implementing a list using pointers. A linked list contains two fields namely, data field and link field.
- Linked lists are useful when the number of elements to be stored in a list is indefinite.
- The HEAD node or the START node depicts the beginning of the list and holds the total number of elements or nodes present in a list. The various types of linked lists are singly-linked list, doubly-linked list, circular singly-linked list, and circular doubly-linked list.
- A singly-linked list consists of only one pointer to point to another node and the last node always points to NULL to indicate the end of the list.
- A doubly-linked list consists of two pointers, one to point to the next node and the other to point to the previous node.
- In a circular singly-linked list, the last node always points to the first node to indicate the circular nature of the list.
- A circular doubly-linked list consists of two pointers for forward and backward traversal and the last node points to the first node.
- Except for circular linked list, all other types of linked lists assign a NULL value to the last node to depict the end of the list

5.5 Keywords

Abstract Data Type (ADT): It is a mathematical model for certain classes of data structure such as, lists, stacks, tress, graphs, and so on that are similar in behavior.

Iteration: In programming, iteration is an act of repeating a certain process to obtain the desired output.

Null Pointer: It is a pointer not pointing to any element in the list. The link field of the last node is assigned a NULL value instead of any address.

Start Node: It is also known as an external pointer. A start node consists of the address of the first node. Using the first node's address the next consecutive nodes can be accessed.

5.6 Self Assessment

1. State whether the following statements are true or false:
   (a) A linked list follows a strategy of allocating memory in a single block.
   (b) The data field is a pointer that points to the data present in the consecutive nodes.
   (c) The disadvantage of using arrays is its ability to store a fixed number of elements.
   (d) A singly-linked list consists of a single pointer to point to the next node in the list.
   (e) The HEAD node in a linked list can hold a dummy data or store the total nodes present in the list.
2. Fill in the blanks:
   (a) When there are no nodes present in a list, the list is known as …….
   (b) ……. consists of two pointers for each node in a list.
   (c) In a ……. list does the last node's pointer points to the head node.
   (d) ……. are useful when you are not clear of the number of elements to be stored in a list.

3. Select a suitable choice for every question.
   (a) Which of the following field holds the address of next element?
      (i) Data field (ii) Node field (iii) Link field (iv) Address field
   (b) Which of the following value is assigned to the last node to depict the end of a list?
      (i) Address of middle node (ii) NULL
      (iii) Address of first node (iv) Address of previous node
   (c) Which of the following list is also known as a two-way list?
      (i) Singly-linked list (ii) Circular linked list
      (iii) Doubly-linked list (iv) Header list
   (d) Which of the following node is referred to as a dummy node in a linked list?
      (i) Head node (ii) Circular node (iii) End node (iv) First node

5.7 Review Questions

1. "Linked lists are useful when the number of elements to be stored in a list is indefinite." Discuss.
2. "Linked lists require additional storage for references as they use pointers. This makes linked lists impractical for those lists that store data of Character or Boolean types." Discuss.
3. "Doubly-linked lists are more advantageous than singly-linked lists." Discuss.
4. "Dynamic arrays can be used instead of linked lists." Analyze.
5. "Circular linked lists are better than linear lists. "Comment.
6. "Linked lists allow only sequential access to elements. Whereas, arrays allow random access to its elements". Analyze.

Answers: Self Assessment

1. (a) False (b) False (c) True (d) True (e) True
2. (a) Empty list or null list (b) Doubly linked list (c) Circular singly-linked list
   (d) Linked list
3. (a) Link field (b) Loss of data (c) NULL
   (d) Doubly-linked list (e) Head node

5.8 Further Readings

Books

Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw Hill.

Reddy, P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www.scribd.com/doc/26312970/39/REPRESENTATION-OF-LINKED-LIST

http://cslibrary.stanford.edu/103/LinkedListBasics.pdf

Unit 6: Linked List Operations

CONTENTS

Objectives

Introduction

6.1 Traversing a Linked List

6.2 Searching a Linked List

6.3 Inserting a Node into a Linked List

6.3.1 Inserting a Node at the Beginning of a List

6.3.2 Inserting a Node after a Given Node

6.3.3 Inserting a Node in a Circular Linked List

6.4 Deleting a Node from a Linked List

6.4.1 Deleting a Node Following a Given Node

6.5 Summary

6.6 Keywords

6.7 Self Assessment

6.8 Review Questions

6.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain the traversing of a linked list
- Describe the process of searching a linked list
- Explain the method of inserting a node into a linked list
- Discuss the process of deleting a node from a linked list

Introduction

In the previous unit, we discussed the fundamentals of linked list. This unit deals with linked list operations for performing different functions. Data processing involves organizing data into lists. One way to store such data is by means of arrays. But, in arrays the process of insertion and deletion are expensive. Arrays occupy only a block of memory space and it is not possible to extend the size of the array.

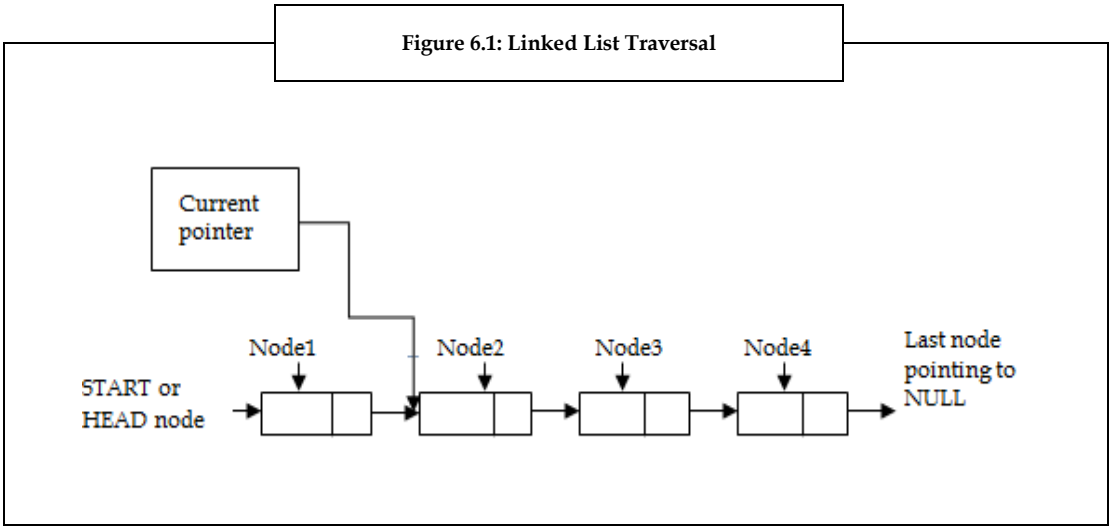
The alternative way of storing the data can be done in a list format. The elements are stored in a list that contains a field called a link or a pointer. The pointer contains the address of the next element in the list. In linked list, it is not essential that consecutive elements occupy contiguous space in memory. Hence it is easy to traverse and insert and delete items in the linked list.

6.1 Traversing a Linked List

Traversing a linked list involves processing every node present in a list. It is useful to perform various operations such as, reversing the order of elements present in a list, sorting the elements in ascending or descending order, and so on.

Figure 6.1 depicts the traversal of a list. The figure shows a list consisting of **HEAD** or the **START** node pointing to the first node (Node1). The last node (Node4) points to **NULL** indicating the end of the list. It also includes a pointer variable named **current** to point to the next node.

Figure 6.1: Linked List Traversal



The list traversal can be achieved by using a **while** loop for every loop iteration. The pointer variable points to the next node and thus all the nodes in the list can be processed.

Example: The program shows the implementation of a traversal of a linked list and displays the elements present in a list.

```
#include<stdio.h>
#include<conio.h>
int LIST[20];
int LIST1[20];
int HEAD;
void MULT(int);
void main()
{
Int PTR;
Clrscr();

LIST[0] =55;
LIST[2]=15;
LIST[3]=20;
LIST[5]=65
LIST[7]=35;
LIST[8]=66;
LIST[9]=12;
LIST[11]=6;
LIST[13]=75;
LIST[14]=80;
LIST[16]=79;
LIST[18]=39;

LIST1[0]=3;
LIST1[2]=13;
LIST1[3]=2;
LIST1[5]=8;
LIST1[7]=14;
```

```
LIST1[8]=9;
LIST1[9]=18;
LIST1[11]=16;
LIST1[13]=5;
LIST1[14]=-1;
LIST1[16]=0;
LIST1[18]=7;

HEAD=11;
PTR=HEAD;
printf("Initially entered list\n");
while(PTR!=-1)
{
printf("%d\t", LIST[PTR]);
PTR=LIST1[PTR];
}
PTR=HEAD;
while(PTR!=-1)
{
MULT(PTR);
PTR=LIST1[PTR];
}
PTR=HEAD;
printf("\n \n List after traversal:\n");
while(PTR!=-1)
{
printf("%d\t", LIST[PTR]);
PTR=LIST1[PTR];
}
getch();
}
void MULT(int p1)
{
LIST[p1]=LIST[p1]*10;
}
```

Output:
When you run the program, the following output is obtained:
Initially entered list:
6
79
55
20
15
75
65
66
12
39
35
80

List after traversal:
60
790
550
200

```
150
750
650
660
120
390
350
800
```

In this example:

1. Two integer lists, named **LIST[ ]** and **LIST1[ ]** are declared with an element storage capacity of 20.
2. An integer pointer named **PTR** is declared to point to the next element.
3. An integer variable named **HEAD** is declared to specify the current position of the **PTR**.
4. A list of values are assigned for **LIST[ ]** and **LIST1[ ]**.
5. The **PTR** is set at position 11 in the list.
6. The first **while** loop, prints the value present in **LIST[11] (6)** and then traverses to the eleventh element of **LIST1[11]** whose value is **16**. The list then traverses back to **LIST[16]** and prints its value **(79)** as shown in the output.
7. The steps 5 and 6 are repeated for each element until the **PTR** is not equal to **-1**, which depicts the end of the elements in a list.
8. The second **while** loop calls the function **MULT()**, which multiplies the traversed data by **10** and prints the value. For example, the value in **LIST[11]=6** is multiplied by **10** and the result **60** is printed.
9. Step 8 is repeated until the end of the loop is reached, i.e., until **PTR** is not equal to **-1**.

Task Write a function to print the reverse of elements present in a list.

6.2 Searching a Linked List

Searching is the most common operation performed in a linked list. The search operation involves traversing through the list to search for a specific item. Several iterations may be performed until the desired number is found or until the end of the list is reached. In search operation, every element in the list is associated with a key element. The element is searched in the list by using a key.

Example: The program searches for a specific number entered by the user by traversing through each node present in a list. The program also displays the number's position in the list.

```
#include<stdio.h>
#include<conio.h>
int LIST[20];
int LIST1[20];
int HEAD;
int SEARCH(int);
void main()
{
```

```
int PTR, NUM, NUM_LOC;
Clrscr();
LIST[0] =55;
LIST[2]=10;
LIST[3]=19;
LIST[5]=60;
LIST[7]=35;
LIST[8]=20;
LIST[9]=8;
LIST[11]=12;
LIST[13]=45;
LIST[14]=68;
LIST[16]=75;
LIST[18]=80;

LIST1[0]=3;
LIST1[2]=13;
LIST1[3]=2;
LIST1[5]=8;
LIST1[7]=14;
LIST1[8]=9;
LIST1[9]=18;
LIST1[11]=16;
LIST1[13]=5;
LIST1[14]=-1;
LIST1[16]=0;
LIST1[18]=7;

HEAD = 16;
PTR=HEAD;
printf("Traversed List:\n);
While(PTR!=-1)
{
printf(" %d/t", LIST[PTR]);
PTR=LIST1[PTR];
}
printf("\n\n Enter the number to search");
scanf(" %d", &NUM);
NUM_LOC=SEARCH(NUM);
if(NUM_LOC==-1)
printf("\n number is not present in the list");
else
printf("\n number %d is present at index location %d in the list", NUM,
NUM_LOC);
getch();
}
int SEARCH(int I)
{
int p=HEAD;
int L=-1;
while(p!=-1)
{
If(I==LIST[p])
{
L=p;
break;
}
```

```
else
p=LIST1[p];
}
return(L);
}
```

Output:
When you run the program, the following output is obtained:
Traversed List:
75 55 19 10 45 60 20 8 80 35 12 68
Enter the number to search
8
Number 8 is present at index location 9 in the list

In this example:

1. Two integer lists named **LIST[ ]** and **LIST1[ ]** are declared with an element storage capacity of **20**.
2. An integer pointer named **PTR** is declared to point to the next element
3. An integer variable named **HEAD** is declared to specify the current position of the **PTR**.
4. A list of values are assigned for **LIST[ ]** and **LIST1[ ]**.
5. The **PTR** is set at position **16** in the list.
6. The first **while** loop prints the value present in **LIST[16]** which is **(75)** and traverses to the third element of **LIST1[16]** whose value is **0**. The list then traverses back to **LIST[0]** and then prints its value **(55)** as shown in the output.
7. Steps 5 and 6 are repeated for each element until the **PTR** is not equal to **-1**, which depicts the end of the elements in a list.
8. The program prompts the input from the user to search a specific number in the list.
9. If the number entered is equal to **HEAD,** then the **while** loop is terminated. Otherwise, the **LIST1[p]** is stored in **p**. The **while** loop iterates again and again until **p** reaches the end of the list**.**
10. If the **NUM_LOC** has not reached the end of the list, then the **NUM** and **NUM_LOC** will be printed. Otherwise, a message saying "number not present in the list" is displayed.

Did you know? Accessing data in a linked list takes more time when compared to arrays because to access a specific element, the list has to be traversed from starting point to the end point of the list, which consumes more time.

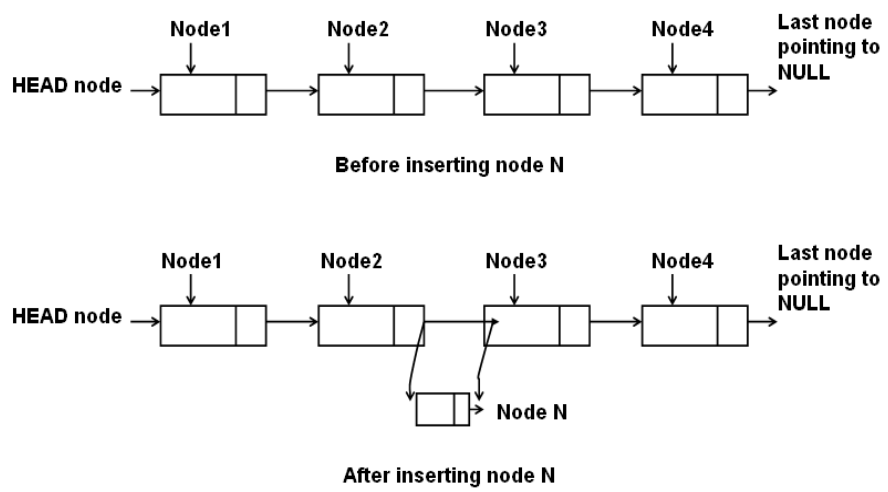
6.3 Inserting a Node into a Linked List

Insertion is one of the basic operations in linked lists. There are three types of insertions that can be carried out:

1. Inserting a node at the beginning of a list
2. Inserting a node at the end of the list
3. Inserting a node after a given node

Figure 6.2 depicts the insertion of a new node. The linked list consists of five nodes such as, **HEAD** node, Node1, Node2, Node3, and Node4 respectively. Suppose, we insert node **N** between Node2 and Node3, then Node2 will point to new node **N** and node **N** will point to Node3.

Figure 6.2: Insertion of a New Node



6.3.1 Inserting a Node at the Beginning of a List

Let us consider **HEAD** as the initial position in a linked list, **num** as the input entered by the user, **PTR** as the pointer pointing to the next node address, and **newnode** as the node being inserted in the list.

Algorithm to insert a new node at the beginning of a list is as follows:

1. Input the value to be inserted
2. Create a new node **N**
3. Store the **DATA** in the data field of the node
4. If **HEAD** is equal to **NULL**, then assign link field of **newnode** to **NULL**. Otherwise, assign **HEAD** to link field of **newnode**. The new node now points to **HEAD.**
5. Exit

Example: The following program inserts a node at the beginning of a list:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
#include<ctype.h>
struct list
{
    int value;
```

```
    struct list *next;
}*HEAD,*newnode,*PTR;
void menu();
void createnode();
void display();
int emptylist();
void insert_beg(int);

void main()
{
   clrscr();
   menu();
}
void menu()
{
   int choice, num;
   printf("MENU");
   printf("\n1. Create node");
   printf("\n2. Display");
   printf("\n3. Insert node at the beginning");
   printf("\n4.Exit");
   printf("\n Enter your choice: ");
   scanf("%d",&choice);
switch(choice)
{
case 1:
  createnode();
  clrscr();
  printf("The created linked list is:\n");
  display();
  getch();
  clrscr();
  menu();
  break;
case 2:
  clrscr();
  if(emptylist()==1)
```

```
{
  printf("The linked list is:\n");
  display();
}
  getch();
  clrscr();
  menu();
  break;
case 3:
   clrscr();
   printf("\n Enter the number to be inserted: ");
   scanf("%d",&num);
   insert_beg(num);
   clrscr();
   printf("\n After insertion at the beginning the linked list is:\n");
  display();
  getch();
  clrscr();
  menu();
  break;
case 4:
  exit(1);
default:
  clrscr();
  printf("Your choice is wrong\n\n");
  menu();
}
}
void createnode()
{
  int num;
  char ch;
  clrscr();
  newnode=(struct list*)malloc(sizeof(struct list));
  HEAD=newnode;
  do
{
```

```
 printf("\n Enter data: ");
 scanf("%d",&num);
newnode->value=num;
printf("\n Do you want to create another node:(y/n)");
fflush(stdin);
scanf("%c",&ch);
if(tolower(ch)=='y')
{
newnode->next=(struct list*)malloc(sizeof(struct list));
newnode=newnode->next;
}
else
{
newnode->next=NULL;
}
}
while(tolower(ch)!='n');
}
void display()
{
int i;
PTR=HEAD;
i=1;
while(PTR!=NULL)
{
printf("\nNode %d : %d",i,PTR->value);
PTR=PTR->next;
i++;
}
}
int emptylist()
{
if(HEAD==NULL)
{
printf("\nLinked List is empty");
return(0);
}
```

```
else
{
return(1);
}
}
void insert_beg(int num)
{
newnode=(struct list*)malloc(sizeof(struct list));
newnode->value=num;
if(HEAD==NULL)
{
HEAD=newnode;
newnode->next=NULL;
}
else
{
newnode->next=HEAD;
HEAD=newnode;
}
}
```

Output:
When you run the program, the following output is obtained:

```
Menu
1.    Create node
2.    Display
3.    Insert node at the beginning
4.    Exit
Enter your choice:
1
Enter data: 40
Do you want to create another node(y/n)
y
Enter data: 50
Do you want to create another node(y/n)
n
```

The created linked list is:

Node1: 40

Node2: 50

Menu

1. Create node
2. Display
3. Insert node at the beginning
4. Exit

Enter your choice:

2

The linked list is:

Node1: 40

Node2: 50

Menu

1. Create node
2. Display
3. Insert node at the beginning
4. Exit

Enter your choice:

3

Enter the number to be inserted: 60

After insertion at the beginning, the list is:

Node1: 60

Node2: 40

Node3: 50

In this example:

1. A list is created with a data field named **value** and link field named ***next**.
2. Three pointers are created namely, **HEAD**, **PTR** and **newnode. HEAD** points to the first node, **PTR** points to the next element, and **newnode** signifies the new node yet to be created.
3. Four functions are created namely, **menu()**, **createnode()**, **display()**, **emptylist(),** and **insert_beg()**. The function **menu()** displays the operations that a user can perform. The operations are:

   (a) Creating a node

   (b) Displaying the list items

   (c) Inserting a node at the beginning of the list

   When the user tries to display an empty list, the function **emptylist()** is called.

4. The **switch** statement is used to implement the three operations. The details of the actions performed in each switch case is as follows:
   (a) Case 1 and case 2 call the function **display()** to display the elements present in a list or to display a message "linked list is empty" in case there are no elements.
   (b) Case 3 prompts the input from the user to insert a new node at the beginning of the list by calling **insert_beg()** function and also calls the **display()** function to display the list.
   (c) Case 4 results in the exit of the program.
   (d) The **default** prints a message "your choice is wrong" in case the user enters input greater than 4.
5. The program terminates when the user enters 4.

6.3.2 Inserting a Node after a Given Node

Let us consider **START** as the initial position in a list, **data** as the input entered by a user, **pos** as any position in the list where the data will be inserted, and **q** and **temp** as the temporary pointers to hold the node address.

Algorithm to insert a new node after a given node is as follows:

1. Input the **data**
2. Specify the **pos** where the **data** is to be inserted
3. Initialize **q** to **START.**
4. If **q** is equal to **NULL**, then print "there are less elements present in the list than the entered position number"
5. for (i=0;i<pos-1;i++). q=q->link;

   temp=(struct node*)malloc(sizeof (struct node));

   temp->link=q->link;

   temp->value=data;

   q->link=temp;
6. Repeat step 5 until **i** value is not less than **pos-1**
7. Exit.

Example: The following program inserts a node after a given node:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>

void createlist(int);
void add_beginning(int);
void add_after(int, int);
void Display();
struct node
{
int value;
struct node *link;
}*START;
```

```
void createlist(int data)
{
struct node *q,*temp;
temp= (struct node*)malloc(sizeof(struct node));
temp->value=data;
temp->link=NULL;
if(START==NULL)
START=temp;
else
q=START;
while(q->link!=NULL)
q=q->link;
q->link=temp;
}
void add_beginning(int data)
{
struct node *temp;
temp=(struct node*)malloc(sizeof(struct node));
temp->value=data;
temp->link=START;
START=temp;
}
void add_after(int data,int pos)
{
struct node *temp,*q;
int i;
q=START;
for(i=0;i<pos-1;i++)
{
q=q->link;
if(q==NULL)
{
printf ("\n\n There are less than %d elements",pos);
getch();
return;
}
}
temp=(struct node*)malloc(sizeof (struct node));
temp->link=q->link;
temp->value=data;
q->link=temp;
}
void Display()
{
struct node *q;
if(START == NULL)
{
printf ("\n\nList is empty");
return;
}
q=START;
printf("\n\nList is : ");
while(q!=NULL)
{
printf ("%d ", q->value);
q=q->link;
}
```

```
printf ("\n");
getch();
}
void main()
{
int choice,n,m,position,i;
START=NULL;
while(1)
{
clrscr();
printf ("1.Create List\n");
printf ("2.Add at beginning\n");
printf ("3.Add after \n");
printf ("4.Display\n");
printf ("5.Exit\n");
printf ("\nEnter your choice:");
scanf ("%d",&choice);
switch (choice)
{
case 1:
printf ("\n\n Enter the number of nodes you want add:");
scanf ("%d",&n);
for(i = 0;i<n;i++)
{
printf ("\nEnter the element:");
scanf ("%d",&m);
createlist(m);
}
break;

case 2:
printf ("\n\nEnter the element : ");
scanf ("%d",&m);
add_beginning(m);
break;

case 3:
printf ("\n\nEnter the element:");
scanf ("%d",&m);
printf ("\nEnter the position after which you want to insert the element:");
scanf ("%d",&position);
add_after(m,position);
break;

case 4:
Display();
break;
case 5:
exit(0);

default:
printf ("\n\n Wrong choice");
}
}
}
```

Output:
When you run the program, the following output is obtained:

1. Create List
2. Add at beginning
3. Add after
4. Display
5. Exit

Enter your choice: 1

Enter the number of nodes you want to add: 2

Enter the element: 10 20

1. Create List
2. Add at beginning
3. Add after
4. Display
5. Exit

Enter your choice: 4

List is: 10 20

1. Create List
2. Add at beginning
3. Add after
4. Display
5. Exit

Enter your choice: 3

Enter the element: 30

Enter the position after which you want to insert the element:2

1. Create List
2. Add at beginning
3. Add after
4. Display
5. Exit

Enter your choice: 4

List is: 10 20 30

In this example:

1. A list named **node** is created with a data field named **value** and link field named ***link**
2. Four functions are created namely, **createlist()**, **add_beginning()**, **add_after()** and **Display().** The user has to select the appropriate function to be performed.

3. The function **createlist()** allocates the memory for list **node** and stores the elements entered by the user.
4. The function **add_beginning()** allocates the memory for each element entered by the user and stores the elements at the beginning of the list.
5. The function **add_after()** prompts the user to enter the input for the element data and the position number where the data is to be stored.
6. The **for** statement in the **add_after()** function searches for the position to insert the new element.
7. If the user enters a position that is less than the actual elements, then a message saying "there are less elements present than the entered position number" is displayed.
8. The **Display()** function displays the element data and the position where the data is inserted.
9. The program terminates when the user enters 5.

Notes

In a doubly-linked list, the **HEAD** node is set to **NULL**. Suppose, you want to create a node with a value of 20, then the head will now point to HEAD->next = new_node and the new node will point to newnode->previous=HEAD. This depicts how two pointers can be used to point to next and previous elements. Now, if you wish to add another node with a value 30, then the node having the value 20 will become the head node. You can create the second node by setting HEAD->next! =NULL and HEAD=HEAD->next.

Task

Write a program to insert a node at the beginning of a doubly-linked list.

6.3.3 Inserting a Node in a Circular Linked List

Let us consider **num** as the data entered by the user, **tmp** as the temporary variable holding the address of the first node and **last** as the last node in the list.

Algorithm to insert a new node after a given node is as follows:

1. Input the data
2. Call the **getnode()** function. The **getnode()** allocates the memory for the first node.
3. Save the node data in the **tmp** variable
4. Assign the data entered by the user to the data file of the **tmp**
5. If **last** node is equal to NULL, then assign the data present in **tmp** to **last** node. Otherwise, assign address of last node to the address of **tmp.** Now, the **tmp** holds the address of **last** node.
6. Return the value present in **last** node
7. Exit.

Example: The following program inserts a node in a circular linked list.

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<process.h>
struct list
{
int value;
struct list *link;
};
typedef struct list* node;
node getnode()
{
 node p;
 p= (node) malloc(sizeof(struct list));
 if(p==NULL)
{
            printf("Out of memory\n");
            exit(0);
 }
 return p;
}
node insert_beginning(int num, node last)
{
node tmp;
tmp=getnode();
tmp->value=num;
if(last==NULL)
last=tmp;
else
tmp->link=last->link;
last->link=tmp;
return last;
}
void display(node last)
{
```

```
node tmp;
if(last==NULL)
{
printf("list is empty\n");
return;
}
printf("contents of the list:\t");
for(tmp=last->link;tmp!=last;tmp=tmp->link)
printf("%d/n", tmp->value);
printf("%d\n", tmp->value);
}
void main()
{
node last;
int choice, num;
last=NULL;
for(;;)
{
printf("1. Insert at beginning\n");
printf("2. display\n");
printf("3. Exit\n");
printf("enter the choice/n");
scanf("%d", &choice);
switch(choice)
{
case 1:
printf("enter the number to be inserted\n");
scanf("%d", &num);
last=insert_beginning(num, last);
break;
case 2:
display(last);
break;
default:
exit(0);
}
}
```

}

Output:

1. Insert at beginning

2. Display

3. Exit

Enter the choice

1

Enter the number to be inserted

50

1. Insert at beginning

2. Display

3. Exit

Enter the choice

2

Contents of the list:

50

1. Insert at beginning

2. Display

3. Exit

Enter the choice

1

Enter the number to be inserted

40

1. Insert at beginning

2. Display

3. Exit

Enter the choice

2

Contents of the list:

40

50

In this example:

1. A structure named **list** is created with a data field named **value** and link field named ***link.**

2. Three functions are created namely, **insert_beginning(), display(),** and **exit.** The user has to select the appropriate function to perform.

3. The function **getnode()** creates and allocates memory for first node. If the memory allocation is unsuccessful, then the function displays a message "out of memory".

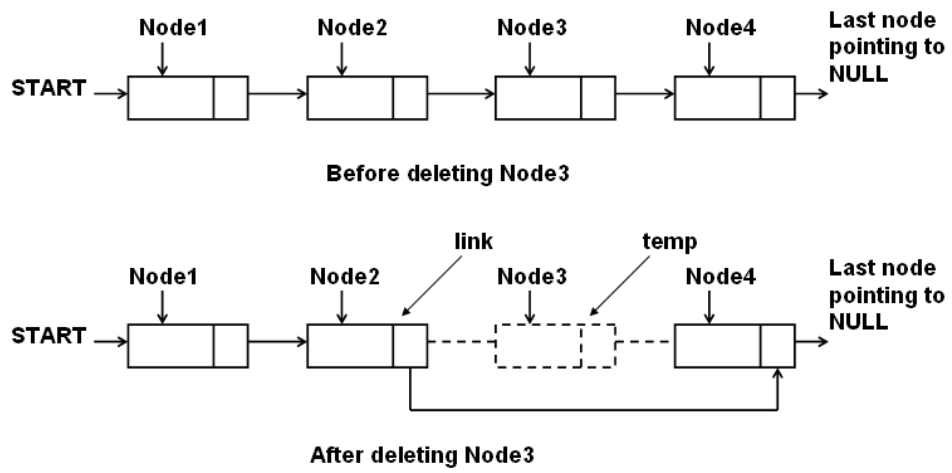
4. The function **insert_beginning()** prompts input from user and inserts the entered number in the first position of the list.
5. The function **display()** displays the list every time a new element is added to the list.
6. The program terminates when the user enters 3.

6.4 Deleting a Node from a Linked List

Figure 6.3 depicts the deletion of a node from the linked list. The linked list consists of five nodes such as, **START**, Node1, Node2, Node3, and Node4 respectively. If Node3 has to be deleted from the list, then assign the address of Node4 to Node2. This is represented as:

Node2->next=&Node4

Figure 6.3: Deleting a Node from a Linked List



6.4.1 Deleting a Node Following a Given Node

Let us consider **START** as the initial position in a list, **data** as the element to be deleted, and **temp** and **q** as the temporary pointers to hold the address of the node.

Algorithm for deleting a node is as follows:

1. Input the **data** to be deleted
2. If **START** is equal to the **data** entered by the user, then
   (a) temp=START
   (b) START=START->link
   (c) free the deleted **temp** node
   (d) exit
3. Assign **START** to **q**
4. While (q->link->link) is not equal to **NULL**, check if(q->link->data) is equal to data. If true then execute the statements in step 5.
5. temp=q->link

   q->link=temp->link

   free the deleted temp node

exit

6. Assign q=q->link
7. To delete the last element, check if(q->link->data) is equal to data. If true, then execute the statements in step 8.
8. temp=q->link

   free the deleted temp node

   q->link=NULL

   exit
9. Display entered element not found
10. Exit.

Example: The following program deletes a node following the given node:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>

void createlist(int);
void delete(int);
void Display();

struct node
{
int value;
struct node *link;
}*START;

void createlist(int data)
{
struct node *q,*temp;
temp= (struct node*)malloc(sizeof(struct node));
temp->value=data;
temp->link=NULL;
if(START==NULL)
START=temp;
else
q=START;
while(q->link!=NULL)
q=q->link;
q->link=temp;
}

void delete(int data)
{
struct node *temp,*q;
if (START->value == data)
{
temp=START;
START=START->link; //to delete first element
free(temp);
return;
```

```
}
q=START;
while(q->link->link != NULL)
{
if(q->link->value == data) //to delete element in between
{
temp=q->link;
q->link=temp->link;
free(temp);
return;
}
q=q->link;
}
if(q->link->value==data) //to delete last element
{
temp=q->link;
free(temp);
q->link=NULL;
return;
}
printf ("\n\nElement %d not found",data);
getch();
}

void Display()
{
struct node *q;
if(START == NULL)
{
printf ("\n\nList is empty");
return;
}
q=START;
printf("\n\nList is : ");
while(q!=NULL)
{
printf ("%d ", q->value);
q=q->link;
}
printf ("\n");
getch();
}

void main()
{
int choice,n,m,position,i;
START=NULL;
while(1)
{
clrscr();
printf ("1.Create List\n");
printf ("2.Delete element\n");
printf ("3.Display\n");
printf ("4.Exit\n");
printf ("\nEnter your choice:");
scanf ("%d",&choice);
switch (choice)
```

```
{
case 1:
printf ("\n\n Enter the number of nodes you wish to add:");
scanf ("%d",&n);
for(i = 0;i<n;i++)
{
printf ("\nEnter the element:");
scanf ("%d",&m);
createlist(m);
}
break;
case 2:
if (START == NULL)
{
printf("\n\n List is empty");
continue;
}
printf ("\n\n Enter the element for deletion:");
scanf ("%d",&m);
delete(m);
break;
case 3:
Display();
break;
case 4:
exit(0);
default:
printf ("\n\nWrong choice");
}
}
}
```

Output:
When you run the program, the following output is obtained:

1. Create List
2. Delete element
3. Display
4. Exit

Enter your choice:1

Enter the number of nodes you wish to add: 4

Enter the element: 10 20 30 40

1. Create List
2. Delete element
3. Display
4. Exit

Enter your choice: 2

Enter the element for deletion: 40

1. Create List
2. Delete element

3. Display
4. Exit

Enter your choice: 3

List is: 10 20 30

In this example:

1. A structure named **node** with a data field named **value** and link field named ***link** are created.
2. Three functions are created namely, **createlist()**, **delete()**, and **Display().** The user has to select the appropriate function to be performed.
3. The function **createlist()** allocates the memory for list **node** and stores the elements entered by the user.
4. The **delete()** function compares the data entered by the user with the START element if both are equal then the first element is deleted. Otherwise, the elements in the other nodes are verified with the entered data and the deletion of the element takes place accordingly.
5. The **Display()** function displays the list after the deletion operation.
6. The program terminates when the user enters 4.

Task

Write a program to delete an element at the beginning of a circular linked list.

Notes

Deleting a Node in a Circular Linked List

Assume **p** as first node and **q** as last node, **link** as the pointer field pointing to the next node and **value** as the data stored in each node.

To delete an element at the beginning of a circular linked list, follow the statements listed below:

```
p=q->link
q->link=p->link
printf("the deleted element is %d\n", p->value)
freenode(p);
```

Suppose, only one node exists in the list then assign **NULL** value to **q** to indicate the empty list.

The same is depicted in the following statements:

```
If(q->link==q) {
printf("the deleted element is %d\n", q->value)
freenode(q)
q=NULL
return q
}
```

Caselet

Use of Singly-Linked Lists in Development of Processors

Technical Systems Consultants is a U.S. based software company, a supplier of software for Southwest Technical Products Corporation (SWTPC). The operating systems developed by Technical Systems Consultants used singly-linked lists as file structures, a directory entry pointed to the first sector of a file, and succeeding portions of the file were located by traversing pointers. The systems that used this technique are Flex (Motorola 6800 CPU), mini-Flex, and Flex9 (Motorola 6809 CPU).

Source: http://www.absoluteastronomy.com/topics/Linked_list

Lab Exercise

1. Write a program to delete a node at the end of a doubly-linked list. The program should prompt users to enter the data and must contain three functions create_node(), delete_end(), and display().
2. Write a program to insert a node at the beginning of a circular doubly-linked list. The program should prompt users to enter the data and must contain three functions create_node(), delete_end(), and display().

6.5 Summary

- We can perform many operations on a linked list like traversing, searching, inserting and deleting.
- The traversal operation in a linked list involves processing every node present in a list to obtain the desired output.
- Searching operation involves searching for a specific element in the list using an associated key.
- Insertion operation involves inserting a node at the beginning or end of a list.
- Deletion operation involves deleting a node at the beginning or following a given node or at the end of a list.

6.6 Keywords

Key Element: An element from a given list which is made as a base reference to search other elements in that list.

Link Field: The initial word of a message buffer which is used to point to the next buffer on the message row.

6.7 Self Assessment

1. State whether the following statements are true or false:
   (a) It is possible to insert an element anywhere in the linked list.
   (b) Linked list stores only a fixed set of data in the list.
   (c) Traversing operation is used to search a specific element in the list.
   (d) Deletion of a node can be done only at the beginning or end of a list.
   (e) Search operation is used for reversing and sorting the list elements.

2. Fill in the blanks:
   (a) …… operation involves iterating through every element in a list to arrange the data in a specific order.
   (b) While inserting an element in the list …… operation has to be performed.
   (c) …… keyword is used to clear the data present in a variable.
   (d) In search operation, every element in the list is associated with a……
3. Select a suitable choice for every question:
   (a) In search operation, every element in the list is searched using a ……
      (i) Key
      (ii) Pointer
      (iii) Address
      (iv) Link
   (b) Which of the following operations are performed if the list is not empty?
      (i) The address of the next node is assigned to the HEAD node.
      (ii) The address of the next node is assigned to NULL.
      (iii) The address of the next is assigned to temp variable.
      (iv) The address of the previous node is assigned to NULL.

6.8 Review Questions

1. "List traverse operation forms the basis for other operations such as, reversing and sorting of list elements." Analyze.
2. "In a doubly-linked list, insertion and deletion takes more time than linear linked list." Analyze.
3. "Using a head node is essential while performing linked list operations." Discuss.
4. "It is possible to insert an element anywhere in the linked list". Justify

Answers: Self Assessment

1. (a) True
   (b) False
   (c) False
   (d) False
   (e) False
2. (a) Traversing
   (b) Incrementing
   (c) Free
   (d) Key element
3. (a) Key
   (b) The address of the next node is assigned to the HEAD node

6.9 Further Readings

Books

Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw Hill.

Reddy, P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www.cs.rpi.edu/~musser/gp/List/lists1.html

http://www.stsci.edu/~bsimon/linked_list.html

Unit 7: Stacks

CONTENTS

Objectives

Introduction

7.1 Fundamentals of Stacks

7.1.1 Stack Structure

7.2 Basic Operations of Stack

7.2.1 Push Operation

7.2.2 Pop Operation

7.3 Representing Stacks in Memory

7.4 Stack Implementation using Arrays

7.5 Summary

7.6 Keywords

7.7 Self Assessment

7.8 Review Questions

7.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the fundamentals of stacks
- Explain the basic operations of stack
- Analyze representing stacks in memory
- Discuss stack implementation using arrays

Introduction

Stacks are simple data structures and an important tool in programming language. Stacks are linear lists which have restrictions on the insertion and deletion operations. These are special cases of ordered list in which insertion and deletion is done only at the ends.

The basic operations performed on stack are ***push*** and ***pop***. Stack implementation can be done in two ways - static implementation or dynamic implementation. Stack can be represented in the memory using a one-dimensional array or a singly linked list.

7.1 Fundamentals of Stacks

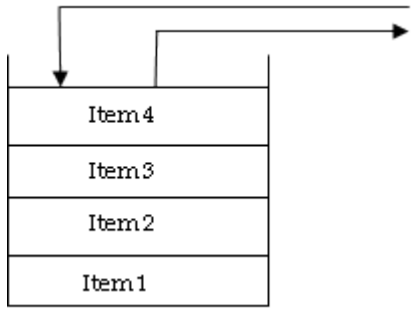
A stack is a linear data structure in which allocation and deallocation are made in a last-in-first-out (LIFO) method. In the LIFO method, the insertions and deletions are done at one end which is known as the top of the stack (TOS). In a stack, the top is a variable which points to the top element in the stack. Consider a stack of books or stack of coins. The items can be added or removed only from the top. This means that the last item that is added to the stack is the first item to be removed.

Did you know? The stack was first proposed in 1955 and then patented in 1957 by the German Friedrich L. Bauer. The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin.

7.1.1 Stack Structure

The stack data structure is used to maintain records of a file in which the order among the records of file is not important. Figure 7.1 displays the structure of a stack where stack is like a hollow cylinder with a closed bottom end and an open top end. In the stack data structure, the records are added and deleted at the top end. Last-In-First-Out (LIFO) principle is followed to retrieve records from the stack. The records added last are accessed first. In Figure 7.1, the order of entry of the records in the stack is item 1, 2, 3, 4 and the order of retrieval of the records from the stack is item 4, 3, 2, 1.

Figure 7.1: Stack Data Structure



7.2 Basic Operations of Stack

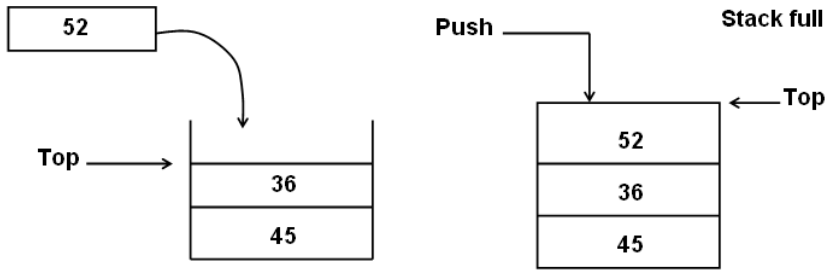
The basic operations of stack are to:

1. Insert an element in the stack (Push operation)
2. Delete an element from the stack (Pop operation)

7.2.1 Push Operation

The procedure to insert a new element to the stack is called push operation. The push operation adds an element on the top of the stack. '**Top**' refers to the element on the top of stack. Push makes the '**Top**' point to the recently added element on the stack. After every push operation, the '**Top**' is incremented by one. When the array is full, the status of stack is **FULL** and the condition is called stack overflow. No element can be inserted when the stack is full. Figure 7.2 illustrates the push operation in the stack data structure.

Figure 7.2: Push Operation in the Stack Data Structure



In the figure 7.2, the stack has two elements 45 and 36. The '**Top**' points to '36' as it is the last item in the stack. Element 52 is added on the stack through push operation. The '**Top**' points to '52' after the push

operation as it is the last item recently added. After adding 52, the stack is full or it is in stack overflow condition. No more items can be added in this stack.
The syntax used for Push operation is PUSH (stack, item).

Algorithm to Implement Push Operation on Stack

```
PUSH (STACK, n, top, item)        /* n = size of stack*/
if (top = n) then STACK_FULL;  /* checks for stack overflow  */
else
   { top = top+1;                 /* increases the top by 1 */
   STACK [top] = item ;}   /* inserts item in the new top position */
end PUSH
```

7.2.2 Pop Operation

The procedure to delete an element from the top of the stack is called pop operation. After every pop operation, the **'Top'** is decremented by one. When there is no element in the stack, the status of the stack is called empty stack or stack underflow. The pop operation cannot be performed when it is in stack underflow condition.

Figure 7.3: Pop Operation

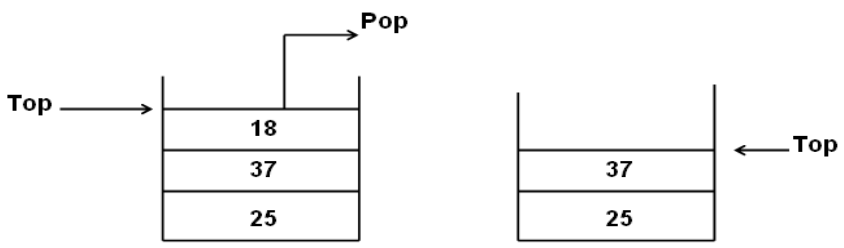


Figure 7.3 shows the pop operation in stack. The stack initially has three items, 25, 37 and 18. The '**Top**' points to the last item, 18. After the pop operation, item 18 is deleted from stack. Now, the '**Top**' points to 37.
The syntax used for Pop operation is POP (stack).

Algorithm to Implement Pop Operation in a Stack

```
POP (STACK, top, item)
if (top = 0) then STACK_EMPTY;      /* check for stack underflow*/
else { item = STACK [top];          /* remove top element*/
       top = top – 1;               /* decrement stack top*/
     }
end POP
```

7.3 Representing Stacks in Memory

Stacks are represented in main memory by using one-dimensional arrays or by using a singly linked list.

Array Representation of Stacks

First, a memory block of sufficient size is allocated to accommodate the full capacity of the stack. The items of the stack are stored in a sequential manner.

Using C language, stack is declared as an array of variable

int stack[Max_STACK_SIZE];

or

char stack[Max_STACK_SIZE];

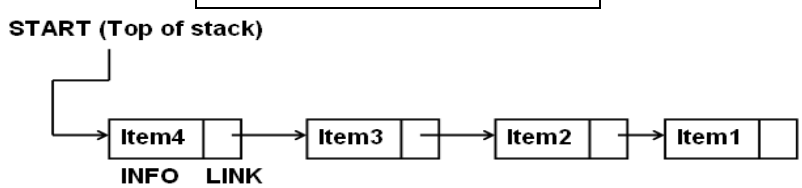
where Max_STACK_SIZE is the maximum number of elements that can be inserted in a stack. Max_STACK_SIZE should be defined by an appropriate value.

Example: In C language, we can define constants using #define statement. The stack size with a capacity of storing 6 elements can be defined as
#define MAX_STACK_SIZE 6

Linked List Representation of Stacks

The array representation of stacks is easy and convenient. However, it allows the representation of only fixed sized stacks. The size of the stack varies during program application for different applications. Representing stack using linked list can solve this problem. A singly linked list can be used to represent any stack. In a singly linked list, the data field represents the **ITEM** and the **LINK** field points to the next item. Figure 7.4 shows the linked representation of the stack.

Figure 7.4: Linked List Representation of a Stack



In figure 7.4, the **INFO** field of the nodes holds the elements of the stack. The **LINK** field holds pointers to the next element in the stack. The **START** pointer of the linked list is the Top pointer variable of stack.

7.4 Stack Implementation using Arrays

A stack is a sequence of data elements. To implement a stack structure, an array can be used as it is a storage structure. Each element of the stack occupies one array element. Static implementation of stack can be achieved using arrays. The size of the array, once declared, cannot be changed during the program execution. Memory is allocated according to the array size. The memory requirement is determined before the compilation. The compiler provides the required memory. This is suitable when the exact number of elements is known. The static allocation is an inefficient memory allocation technique because if fewer elements are stored than declared, the memory is wasted and if more elements need to be stored than declared, the array cannot expand. In both the cases, there is inefficient use of memory.

Example: Program for Stack Implementation Using Arrays

```
 #include <stdio.h>
#include <conio.h>
 #define MAX 10
 void push();
 int pop();
 void display();
 int stack_arr[MAX];
 int top= -1;
                void  main()            /* program execution  begins in the main
method */
                {
                    int ch, pop_value;
                    clrscr();
                     do                 /* do …while loop displays  three choices and
accepts the choice                entered*/
                   {
                     printf("\n 1. Push/n");
                     printf("\n 2. Pop/n");
                     printf("\n 3. Display/n");
                     printf("\n Enter the choice/n");
                     scanf("%d", &ch);
                   switch (ch)                    /* switch case compares the value
entered with each case */
                   {
                      case 1: push();
                      break;
                      case 2: pop_value = pop();
                      break;
                      case 3: display();
                      break;
                     case 4: exit(1);
                   /* If any choice other than 1,2 or 3 is entered, the message is
displayed*/
                      default: printf("\n Wrong choice");
                   }
                   } while(1);
                   }
                  void push()       /* If the value entered is 1, the push method is
called*/
                  {
                     int item;
                     if(top==(MAX-1))     /* if condition checks whether the stack
is full or not*/
                  {
                     printf("\n Overflow. Stack is full");      /* If the stack is full,
the message is displayed*/
                    getch();
                    exit(0);
                 }
                    else
                 {
                /* If the stack is not full, a message is displayed and values are
```

```
accepted*/
                    printf("\n Enter the element to insert in the stack");
                    scanf("%d",&item);              /* accepts the value entered in
variable item*/
                    top=top+1;                  /*after the insertion, top is incremented
by 1*/
                   stack_arr[top]=item;
                  }
                  }
                  int pop()          /* If the value entered is 2, the pop method is
called*/
                  {
                    int i  = 0;
                    if(top==-1)     /* if condition checks whether the stack is empty
or not*/
                  {
                   printf("\n Stack underflow");    /* If the stack is empty, the
message is displayed*/
                   getch();
                   exit(0);
                  }
                 else
                 {
                   i=stack_arr[top];       /* If the stack is not empty, the item is
deleted*/
                   top=top-1;            /* top is decremented by 1*/
                 }
                return i;
                }
        void display()         /* display method displays the status of the stack*/
        {
          int i;
          if(top==-1)           /* if condition checks whether the stack is empty or
not*/
        {
          printf("\n Stack is empty");
          getch();
          exit(0);
        }
        else
        {
        for(i=top;i>=0;i--)
        {
          printf("\n The item is %d",stack_arr[i]);  /* displays all the items in the
stack*/
        }
        }
        }
```

Output:

```
Push
Pop
Display
Exit
Enter the choice
 1
```

```
Enter the element to insert in the stack
50
Push
Pop
Display
Exit
Enter the choice
 1
Enter the element to insert in the stack
60
Push
Pop
Display
Exit
Enter the choice
3
The item is 60
The item is 50
Push
Pop
Display
Exit
Enter the choice
 2
Push
Pop
Display
Exit
Enter the choice
3
The item is 50
```

In this example:

1. The header files are included and a constant value 10 is defined for variable MAX using #define header.
2. An integer stack array named stack_arr[MAX] is declared. The stack array can hold 10 elements.
3. Three functions are created namely, push() , pop() and display(). The user has to select an appropriate function to be performed.
4. The switch statement is used to call the push(), pop(), and display() functions.
5. When the user enters 1, the push() function is called. In the push() function, the **if** loop checks for the stack size. If the stack array is full, the program displays a message "Overflow. Stack is full". Else, it takes the input from the user and inserts into the stack.
6. When the user enters 2, the function pop() is called. In the pop() function, the if loop checks for the stack size. If the stack array is empty, the program displays a message "Stack underflow". Else, the pop() function pops the elements present in the stack array.

7. When the user enters 3, the function display() is called. In the display() function, the if loop checks for the stack size. If the stack array is empty, the program displays a message "Stack is empty". Else, the for loop displays each element present in the array.
8. When the user enters 4, the program terminates.

Did you know? In Computer Science, a call stack stores information about the active subroutines of a computer program. This stack is known as control stack, machine stack or execution stack. It is usually shortened to "stack". The details of the call stack are hidden and automatic in high-level programming language.

Lab Exercise

Write a C program to convert the following expression into prefix and postfix expressions using stacks.

(A + B) * (C – D)/ (E – F)

7.5 Summary

- Stacks are simple data structures and important tools in programming language.
- A stack is a linear data structure in which allocation and deallocation are made in a last-in-first-out (LIFO) method.
- The basic operations of stack are inserting an element on the stack (push operation) and deleting an element from the stack (pop operation).
- Stacks are represented in main memory by using one-dimensional array or a singly linked list.
- To implement a stack structure, an array can be used as its storage structure. Each element of the stack occupies one array element. Static implementation of stack can be achieved using arrays.

7.6 Keywords

Deallocation: A process by which memory is reclaimed.

Dynamic Allocation: Automatic memory allocation where memory is allocated as required at run-time.

Linear Lists: A sequential list.

Static Allocation: Process of allocating memory at compile-time before the associated program is executed.

7.7 Self Assessment

1. State whether the following statements are true or false:
   (a) Stacks are ordered linear list in which insertion and deletion is done at both the ends.
   (b) A stack is a linear data structure in which allocation and deallocation are made in a last-in-last-out (LILO) method.
   (c) The insertions and deletions in a stack is done at one end which known as top of the stack.
   (d) The size of the array once declared, cannot be changed during the program execution.
   (e) When the array is full, the condition is called stack overflow.

2. Fill in the blanks:
   (a) No element can be inserted when the status of the stack is …….
   (b) In a stack, the items can be added or removed only from the ……..
   (c) The …… operation adds an element on the top of the stack.
3. Select a suitable choice for every question:
   (a) After every push operation the top is incremented by ……
      (i) 2
      (ii) 3
      (iii) 1
      (iv) 4
   (b) Pushing an element into a stack that is full results in a condition called ……..
      (i) Stack full
      (ii) Stack overfull
      (iii) Stack overflow
      (iv) Stack underflow
   (c) Static stack implementation can be achieved using ……
      (i) Linked list
      (ii) Array
      (iii) Queue
      (iv) Structure

7.8 Review Questions

1. "Static implementation of stack can be achieved using arrays". Explain with a program.
2. "Stacks are represented in main memory using two ways". Discuss with a program.
3. "The static allocation in a stack is an inefficient memory allocation technique". Provide a solution to this problem with an example.
4. "In static stack implementation, when the array is full, the status of stack is FULL and the condition is called stack overflow". Discuss.
5. "The pop operation cannot be performed when the status of the stack is underflow". Discuss with an example.

Answers: Self Assessment

1. (a) False (b) False (c) True (d) True (e) False
2. (a) Overflow (b) Top (c) Push
3. (a) 1 (b) Stack overflow (c) Array

7.9 Further Readings

Books

Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw-Hill.

Reddy, P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www.brucemerry.org.za/manual/structures/circular.html

http://www.niitcrcs.com/btpc/btpc-08%20papers%5CIsha-Arrays.pdf

http://www.sqa.org.uk/e-learning/ArrayDS02CD/page_19.htm

Unit 8: Queues

CONTENTS

Objectives

Introduction

8.1 Fundamentals of Queues

8.2 Basic Operations of Queue

8.2.1 Insert at Rear End

8.2.2 Delete from the Front End

8.3 Representing Queue in Memory

8.4 Types of Queue

8.4.1 Double Ended Queue

8.4.2 Circular Queue

8.4.3 Priority Queue

8.5 Summary

8.6 Keywords

8.7 Self Assessment

8.8 Review Questions

8.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe the fundamentals of queues
- Discuss the basic operations of queues
- Analyze representing queues in memory
- Explain the types of queues

Introduction

A queue is a linear list of elements that consists of two ends known as front and rear. We can delete elements from the front end and insert elements at the rear end of a queue. A queue in an application is used to maintain a list of items that are ordered not by their values but by their sequential value.

Example: If the users of a particular website want to select a list of reports throughout the day, and during idle time wants to print those reports. The website can be designed in a way that information stored internally can be retrieved easily by implementing the queue mechanism. If a user needs to find the name of the next report to print, the information can be obtained from the top of the queue and new addition of the reports get stored easily at the rear end.

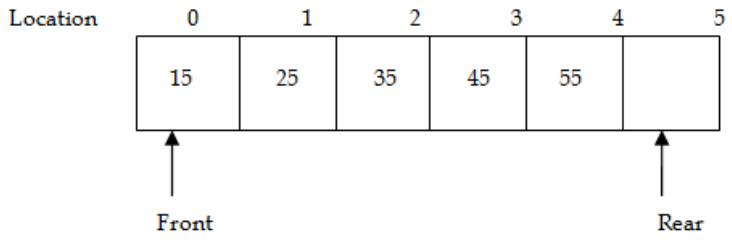
8.1 Fundamentals of Queues

A queue is an ordered collection of items in which deletion takes place at one end, which is called the front of the queue, and insertion at the other end, which is called the rear of the queue. The queue is a 'First In First Out' system (FIFO). In a time-sharing system, there can be many tasks waiting in the

queue, for access to disk storage or for using the CPU. The queues in a bank, or railway station counter are examples of queue. The first person in the queue is the first to be attended.

The two main operations in the queue are insertion and deletion of items. The queue has two pointers, the front pointer points to the first element of the queue and the rear pointer points to the last element of the queue. Figure 8.1 shows the structure of a queue. A new item is inserted at the rear end and elements are removed from the queue from the front end.

Figure 8.1: Queue Structure



8.2 Basic Operations of Queue

The basic operations of queue are insertion and deletion of items which are referred as enqueue and dequeue respectively. In enqueue operation, an item is added to the rear end of the queue. In dequeue operation, the item is deleted from the front end of the queue.

8.2.1 Insert at Rear End

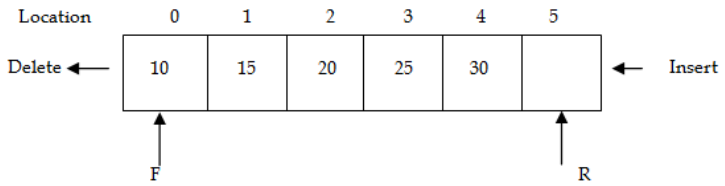
To insert an item into the queue, first it should be verified whether the queue is full or not. If the queue is full, a new item cannot be inserted into the queue. The condition FRONT=NULL indicates that the queue is empty. If the queue is not full, items are inserted at the rear end. When an item is added to the queue, the value of rear is incremented by 1.

8.2.2 Delete from the Front End

To delete an item from the stack, first it should be verified that the queue is not empty. If the queue is not empty, the items are deleted at the front end of the queue. When an item is deleted from the queue, the value of the front is incremented by 1.

Figure 8.2 is a representation of the basic operations of a queue. The first element inserted into the queue is 10, the second element inserted is 15 and so on. 30 is the last inserted element. The first element to be deleted from the queue is 10. If a new element is added, it is inserted after 30 and it will be the last element in the queue. A new element cannot be inserted in the queue when the queue is full.

Figure 8.2: Operations in Queue



Example:

```
/*Program of queue using array*/
/*insertion and deletion in a queue*/
/*insertion and deletion in a queue*/
# include <stdio.h>
# define MAX 50
int queue_arr[MAX];
int rear = -1;
int front = -1;
void ins_delete();
void insert();
void display();
void main()
{
int choice;
while(1)
{
printf("1.Insert\n");
printf("2.Delete\n");
printf("3.Display\n");
printf("4.Quit\n");
printf("Enter your choice : \n");
scanf("%d",&choice);
switch(choice)
{
case 1 : insert();
break;
case 2 : ins_delete();
break;
case 3: ins_display();
break;
case 4: exit(1);
default:
printf("Wrong choice\n");
}/*End of switch*/
        }/*End of while*/
   }/*End of main()*/
void insert()
```

```
{
int added_item;
if (rear==MAX-1)
printf("Queue overflow\n");
else
{
if (front==-1)                          /*If queue is initially empty */
front=0;
printf("Enter an element to add in the queue : ");
scanf("%d", &added_item);
rear=rear+1;
queue_arr[rear] = added_item ;
}
      }                                  /*End of insert()*/
void ins_delete()
{
if (front == -1 || front > rear)
{
printf("Queue underflow\n");
return ;
}
else
{
printf("Element deleted from queue is : %d\n", queue_arr[front]);
front=front+1;
}
}                        /*End of delete() */
void display()
{
int i;
if (front == -1)
printf("Queue is empty\n");
else
{
printf("Elements in the queue:\n");
for(i=front;i<= rear;i++)
printf("%d ",queue_arr[i]);
```

```
printf("\n");
}
}                    /*End of display() */
```

Output:

1. Insert

2. Delete

3. Display

4. Quit

Enter your choice: 1

Enter an element to add in the queue: 25

Enter your choice: 1

Enter an element to add in the queue: 36

Enter your choice: 3

Elements in the queue: 25, 36

Enter your choice: 2

Element deleted from the queue is: 25

In this example:

1. The preprocessor directives #include are given. MAXSIZE is defined as 50 using the #define statement.
2. The queue is declared as an array using the declaration **int queue_arr[MAX].**
3. In the **while** loop, the different options are displayed on the screen and the value entered in the variable **choice** is accepted.
4. The **switch** case compares the value entered and calls the method corresponding to it. If the value entered is invalid, it displays the message **"Wrong choice".**
5. **Insert method**: The **insert** method inserts item in the queue. The **if** condition checks whether the queue is full or not. If the queue is full, the "**Queue overflow"** message is displayed. If the queue is not full, the item is inserted in the queue and the **rear** is incremented by 1.
6. **Delete method**: The **delete** method deletes item from the queue. The **if** condition checks whether the queue is empty or not. If the queue is empty, the "**Queue underflow"** message is displayed. If the queue is not empty, the item is deleted and **front** is incremented by 1.
7. **Display method**: The **display** method displays the contents of the queue. The **if** condition checks whether the queue is empty or not. If the queue is not empty, it displays all the items in the queue.

8.3 Representing Queue in Memory

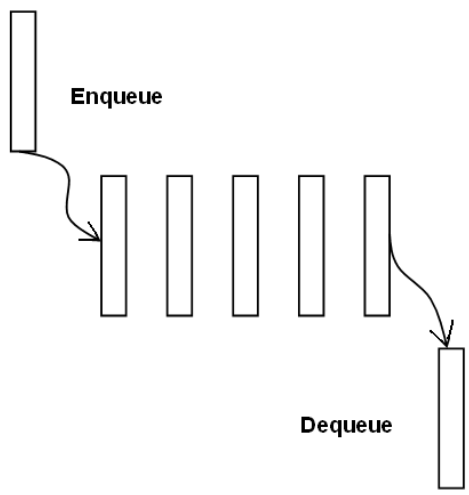
The main attribute of a queue data structure is the fact that it permits accessibility only to the front and back of the structure. Additionally, elements can be removed only from the front and added to the back. In this way, the appropriate metaphor used to characterize queues is the idea of a checkout line.

Example:

Cars moving in line at a gas station or machine parts on the assembly line are the real-life examples where queues are prevalent.

Thereby, queues in data structures are the same as queues that you would see in any shop while waiting to pay at the checkout counter. In each of the cases, the object or customer at the front of the line was the first to enter and the one at the end of the line is the last to have entered. Each time a customer makes payment for their goods (or the machine part is removed from the line) the customer leaves the queue. This represents the "dequeue" function of the queue. Each time another customer or object enters the line to wait, they join the end of the line. This represents the "enqueue" function of the queue. The "size" function of the queue returns the length of the line and "empty" function returns true only if the line is empty. Figure 8.3 depicts how a queue is represented.

Figure 8.3: Representation of a Queue

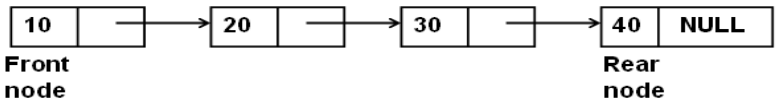


Notes

Queue can be implemented using arrays and linked list too. The main benefit in linked lists is that the size of the queue is not much of a concern. In linked lists, we can add as many nodes as possible and the queue will never have a full condition. The queue that uses linked list would be similar to that of a linked list. The only difference between the two of them is that, in queues, the leftmost node and the rightmost node is called as front and rear node respectively. Also, we cannot remove any of the arbitrary nodes from the queue. Always the front node needs to be removed.

Task

Use the diagram depicted below to implement queue using linked list:



8.4 Types of Queue

The different types of queue are:

1. Double ended queue
2. Circular queue
3. Priority queue

8.4.1 Double Ended Queue

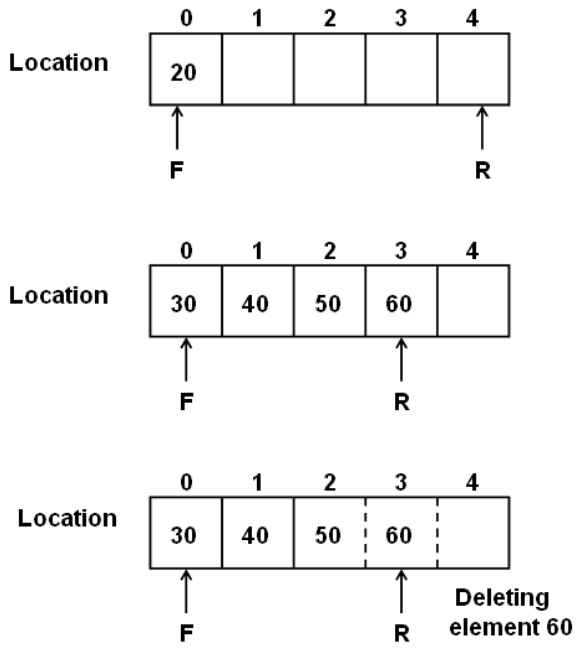
Double ended queue is also known as *deque*. It is a type of queue where the insertions and deletions happen at the front or the rear end of the queue. The various operations that can be performed on the double ended queue are:

1. Insert an element at the front end
2. Insert an element at the rear end
3. Delete an element at the front end
4. Delete an element at the rear end

Let us now discuss how an element can be inserted at the front end and deleted at the rear end of a deque. Figure 8.4 depicts inserting and deleting an element from deque. The front end of the queue is identified by **F** and the rear end is identified by **R.** If we want to insert an element 20 at the front end, we can do this by checking if **F** is equal to zero and then increment **F** and insert the element. We cannot insert an element at the front if an element is already present at the first position, as queue follows 'First In First Out' method. In the figure 8.4, the element 30 is already present in the first position of the queue. Hence, we cannot insert an element at the front end. If we want to delete element 60 at the rear end, access the element at the rear end and then decrement the pointer **R**. When the elements are deleted and queue becomes empty, reset the pointer **F** to 0 and rear end pointer **R** to -1. An element can be deleted only if the queue is not empty.

Figure 8.4 depicts inserting and deleting an element from deque.

Figure 8.4: Inserting and Deleting an Element from Deque



Example: Program for the Insertion and Deletion of an Element in a Dequeue.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int Q_F(int R)
{
    return (R==SIZE-1)?1:0;
}

int Q_E(int F, int R)
{
    return(F>R)?1:0;
}

void front_insert(int num, int Q[], int *F, int *R)
{
    if(*F==0 || *R==-1)
    {
       Q[++(*r)]=item;
       return;
     }
     if(*F!=0)
     {
        Q[--(*F)]=item;
         return;
```

```
      }
      printf("Front inertion not possible\n");
}

void rear_delete(int Q[], int *F, int *R)
{
    if(Q_E(*F, *R))
    {
       printf("Queue underflow\n");
       return;
     }
     printf("The element deleted is %d\n", Q[(*R)--]);

    if(*F>*R)
    {
       *F=0, *R=-1;
     }
}

void display(int Q[], int F, int R)
{
     int i;
     if(Q_E(F, R))
     {
        printf("Queue is empty\n");
        return;
     }
     printf("Contents of the queue is:\n");
     for(i=F;i<=R; i++)
     {
        printf("%d\n", Q[i]);
      }
}
void main()
{
    int choice, num, F, R, Q[10];
    F=0;
    R=-1;
     for(;;)
     {
        printf("1. Insert at front/n");
        printf("2. Delete at rear end/n");
        printf("3. Display/n");
        printf("4. Exit/n");
        scanf("%d", &choice);

      switch(choice)
      {
         case 1: printf("Enter the number to be inserted\n");
              scanf("%d", &num);
              front_insert(num, Q, &F, &R);
              break;
        case 2: rear_delete(Q, &F, &R);
              break;
        case 3: display(Q, F, R);
              break;
        default: exit(0);
```

```
    }
  }
}
```

Output:

1. Insert at front end
2. Delete at rear end
3. Display
4. Exit

1

Enter the number to be inserted

30

1. Insert at front end
2. Delete at rear end
3. Display
4. Exit

1

Enter the number to be inserted

40

1. Insert at front end
2. Delete at rear end
3. Display
4. Exit

3

The contents of the queue is 30 40

1. Insert at front end
2. Delete at rear end
3. Display
4. Exit

2

The element deleted is 40

In this example:

1. The header files are defined and a constant value 5 is defined for variable SIZE using #define header. The SIZE defines the size of the queue.
2. Four functions are created namely, **Q_F()**, **Q_E()**, **front_insert(), rear_delete(), and display().** The user has to select an appropriate function to be performed.
3. The **switch** statement is used to call the **front_insert(), rear_delete(), and display()** functions.
4. When the user enters 1, the **front_insert()** function is called. In the **front_insert()** function the **if loop** checks if the **F** pointer is equal to 0 or **R** pointer is equal to -1. If the result is true, then the **R** pointer is

incremented and the value entered by the user (**num**) is assigned to **Q**. The value of **R** is returned. The second **if loop** checks if the **F** pointer is not equal to 0. If the result is true, then the **F** pointer is decremented and the value entered by the user (**num**) is assigned to **Q**. The value of **F** is returned**.** Else, the program prints the message "front insertion not possible"

5. When the user enters 2, **rear_delete()** function is called. In the rear_**delete()** function the **if loop** calls the **Q_E()** function with the current pointer values of **F** and **R.** If the condition is true, the program prints the message "Queue underflow". It returns the value of **F** and **R.** The program prints the deleted element.
6. When the user enters 3, the function **display()** is called. In the function **display()** the **if loop** checks for the queue size. If the queue is not empty the program displays the elements.
7. When the user enters 4, the program terminates.

Task

Write an algorithm to perform insertion of an element at the rear end and deletion of an element at the front end for a deque.

8.4.2 Circular Queue

In a circular queue, the rear end is connected to the front end forming a circular loop. An advantage of circular queue is that, the insertion and deletion operations are independent of one another. This prevents an interrupt handler from performing an insertion operation at the same time when the main function is performing a deletion operation. The figure 8.5 depicts a circular queue. The queue elements are stored in an array. The front end of the queue is represented as **F** and the rear end is represented as **R.** Before inserting an element into the queue, the **R** pointer should be set to -1. The value of **R** is then incremented to insert the elements. In the first figure of figure 8.5, only one element (20) is present in the queue. Hence, the value of **F** and **R** pointer will be 0. In the second figure of figure 8.5, two elements are added (40 and 60) to the queue. This can be done by incrementing the **R** pointer. The following statement depicts the increment operation:

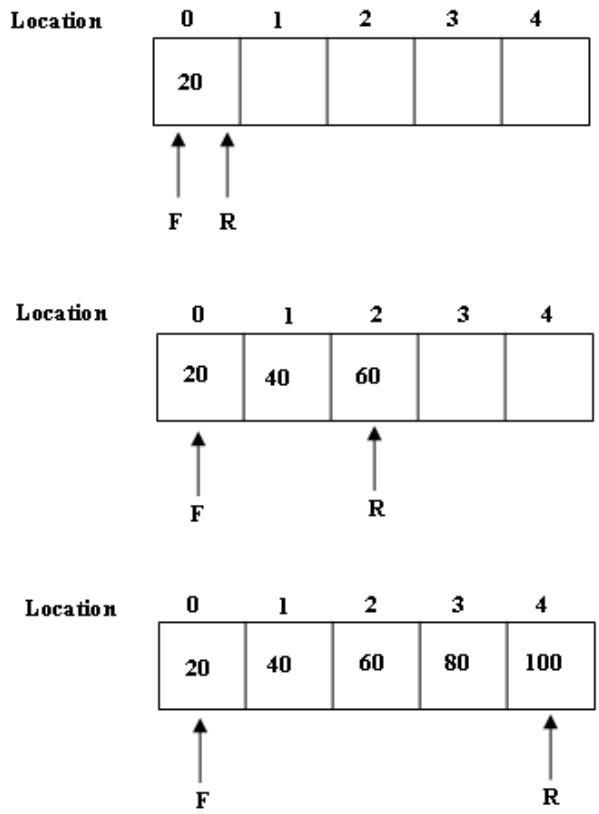
R=(R+1) % SIZE

Here,

SIZE is the queue size. In this case, the size is 5.

In the third figure of figure 8.5, elements 80 and 100 are added to the queue. Now the **R** value will be 5. Since, the value of SIZE is also 5, **R** will point to 0.

Figure 8.5: Circular Queue



Example: Program for Implementation of Circular Queue.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5

int Q_F(int COUNT)
{
    return (COUNT==SIZE)? 1:0;
}

int Q_E(int COUNT)
{
    return (COUNT==0)? 1:0;
}

void rear_insert(int item, int Q[], int *R, int *COUNT)
{
```

```
    if(Q_F(*COUNT))
    {
       printf("Queue overflow");
       return;
    }
    *R=(*R+1) % SIZE;
    Q[*R]=num;
    *COUNT+=1;
}
void front_delete(int Q[], int *F, int *COUNT)
{
    if(Q_E(*COUNT))
    {
       printf("Queue underflow");
       return;
     }
    printf("The deleted element is %d\n", Q[*F]);
    *F=(*F+1) % SIZE;
    *COUNT-=1;
}

void display(int Q[], int F, int COUNT)
{
    int i,j;
    if(Q_E(COUNT))
    {
       printf("Queue is empty\n");
       return;
    }
    printf("The contents of the queue are:\n");
     i=F;
     for(j=1;j<=COUNT; j++)
    {
       printf("%d\n", Q[i]);
       i=(i+1) % SIZE;
    }
    printf("\n");
}
void main()
{
    int choice, num, COUNT, F, R, Q[20];
    clrscr();
    F=0;
    R=-1;
    COUNT=0;
    for(;;)
    {
      printf("1. iInsert at front\n");
      printf("2. Delete at rear end\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      scanf("%d", &choice);

      switch(choice)
      {
        case 1: printf("Enter the number to be inserted\n");
            scanf("%d", &num);
```

```
            rear_insert(num, Q, &R, &COUNT);
            break;
        case 2: front_delete(Q, &F, &COUNT);
            break;
        case 3: display(Q, F, COUNT);
            break;
        default: exit(0);
      }
    }
}
```

Output:

1. Insert at rear end
2. Delete at front end
3. Display
4. Exit

1

Enter the number to be inserted

50

1. Insert at rear end
2. Delete at front end
3. Display
4. Exit

1

Enter the number to be inserted

60

1. Insert at rear end
2. Delete at front end
3. Display
4. Exit

3

The contents of the queue are 50 60

1. Insert at rear end
2. Delete at front end
3. Display
4. Exit

2

The element deleted is 50

In this example:

1. The header files are included and a constant value 5 is defined for variable SIZE using #define statement. The SIZE defines the size of the queue.

2. A queue is created using an array named **Q** with an element capacity of 20. A variable named **COUNT** is declared to store the count of number elements present in the queue.
3. Four functions are created namely, **Q_F()**, **Q_E()**, **rear_insert(), front_delete(),**and **display().** The user has to select an appropriate function to be performed.
4. The **switch** statement is used to call the **rear_insert(), front_delete(), and display()** functions.
5. When the user enters 1, **rear_insert()** function is called. In the **rear_insert()** function, the **if loop** checks if the **count** is full. If the condition is true, then the program prints a message "Queue is empty". Else, it checks for the value of **R** and assigns the element (**num**) entered by the user to **R**. Initially, when there are no elements in the queue, the value of **R** will be **0**. After every insertion, the variable **COUNT** is incremented.
6. When the user enters 2, the **front_delete()** function is called. In this function, the **if loop** checks if the variable **COUNT** is empty. If the condition is true, then the program prints a message "Queue underflow". Else, the element in the **0th** position will be deleted. The size of **F** is computed and the **COUNT** is set to 1.
7. When the user enters 3, the **display()** function is called. In this function, the **if loop** checks if the value of **COUNT** is **0**. If the condition is true, the program prints a message "Queue is empty". Else, the value of **F** is assigned to the variable **i.** The **for loop** then displays the elements present in the queue.
8. When the user enters 4, the program terminates.

8.4.3 Priority Queue

In priority queue, the elements are inserted and deleted based on their priority. Each element is assigned a priority and the element with the highest priority is given importance and processed first. If all the elements present in the queue have the same priority, then the first element is given importance.

Example: Program for Implementation of Priority Queue

```
#include<stdio.h>
#include<malloc.h>
struct queue
{
    int PRI;
    int value;
    struct queue *next;
}*F, *q, *tmp, *new;
typedef struct queue *P;

void ins()
{
   int num, el_pri;
   new = ( P ) malloc(10 );
   printf( "Enter the element to be inserted:" );
   scanf( "%d", &num );
   printf( "Enter a priority:" );
   scanf( "%d", &el_pri );
   new->value = num;
   new->PRI = el_pri;
```

```
   if ( F == NULL || el_pri < F->PRI )
{
   new->next = F;
   F = new;
}
   else
{
   q = F;
   while ( q->next != NULL && q->next->PRI <= el_pri )
   q = q->next;
   new->next = q->next;
   q->next = new;
      }
}
void del()
{
   if ( F == NULL )
      {
         printf( "\n QUEUE UNDERFLOW\n" );
      }
   else
      {
         new = F;
         printf( "\nDeleted number is %d\n", new->value );
         F = F->next;
         free( F );
      }
}
void disp()
{
   tmp = F;
   if ( F == NULL )
      printf( "QUEUE IS EMPTY\n" );
   else
      {
      printf( "QUEUE IS:\n" );
      while ( tmp != NULL )
      {
       printf( "\n%d[PRI=%d]", tmp->value, tmp->PRI );
       tmp = tmp->next;
       }
      }
}
int main()
{
   int choice;
   clrscr();
   while(1)
   {
     printf( "\n 1. INSERT \n 2. DELETE \n 3. DISPLAY \n 4. EXIT" );
     printf( "\n Enter your choice" );
     scanf( "%d", &choice );
     switch ( choice )
     {
        case 1:
          ins();
         break;
```

```
        case 2:
          del();
          break;
        case 3:
          disp();
          break;
        default: exit(1);
      }
  }
}
```

Output:

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

1

Enter the element to be inserted

10

Enter a priority

1

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

1

Enter the element to be inserted

20

Enter a priority

2

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

3

QUEUE IS:

10[PRI=1]

20[PRI=2]

1. INSERT

2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

2

Deleted number is 10

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

2

Deleted number is 20

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT

Enter your choice:

2

QUEUE UNDERFLOW

In this example:

1. The header files namely, **stdio** and **malloc** are included.
2. A structure named **queue** is created which consists of three variables namely, **value, PRI,** and **next**. The variable **value** holds the value of the element. The **PRI** holds the element priority value and next is a pointer variable that points to the next element in the queue. Four objects are declared namely, **F, q, tmp,** and **new** to access the structure elements.
3. Three functions are created namely, **ins(), del(),** and **disp()**. The user has to select an appropriate function to perform.
4. The **switch** statement is used to call the **ins(), del(),** and **disp()** functions.
5. When the user enters 1, the **ins()** function is called. This function allocates memory of capacity **10** for the **queue** using **malloc** function. Then, the user enters the element to be inserted into the queue, and its priority. The value of the element entered is stored in value and its priority is stored in **PRI**. The **if** loop checks if the **F** value is equal to NULL or the priority of the entered element is less than the priority of the first element. If either of the condition is true, then the element entered is stored in the second position of the queue. Else, the value of **F** is assigned to **q.** The **if loop** checks if the third position of the queue is not equal to NULL and the priority of the second element is less than the third. If the condition is true, then the second element is stored in the third position and the third element is stored in the second position. The loop continues to check for the priority of all the elements in the queue and stores them accordingly.

6. When the user enters 2, the d**el()** function is called. In this function, the **if loop** checks if the value of **F** is equal to NULL. If the condition is true, then program prints the message "QUEUE UNDERFLOW". Else, **F** is assigned to **new,** and the element in **F** is deleted. The pointer **F** is set free.
7. When the user enters 3, the function **disp()** is called. In this function, the **if loop** checks if the value of **F** is equal to NULL. If the condition is true, then the program prints the message "QUEUE is EMPTY". Else, it displays the elements present in the **queue** along with their priority.
8. When the user enters 4, the program terminates.

Lab Exercise

1. Create a circular queue having an element storage capacity of 5. Insert 4 elements into the queue. Delete first two elements and insert an element at the position F=1 and R=3.
2. Create a priority queue having an element capacity of 3. Insert the elements 100 having priority 2, 200 having priority 1, and 300 having priority 3. Try deleting element with priority 2. Analyze the result.

8.5 Summary

- A queue is an ordered collection of items in which deletion takes place at the front and insertion at the rear of the queue.
- The basic operations performed on a queue include inserting an element at the rear end and deleting an element at the front end.
- In a memory, a queue can be represented in two ways; by representing the way in which the elements are stored in the memory, and by naming the address to which the front and rear pointers point to.
- The different types of queues are double ended queue, circular queue, and priority queue.

8.6 Keywords

Dequeue: Process of deleting elements from the queue.

Enqueue: Process of inserting elements into queue.

Front End: Refers to the first node in the queue.

Rear End: Refers to the last node in the queue.

8.7 Self Assessment

1. State whether the following are true or false.
   (a) Insertions and deletions can happen at the front or the rear end of the priority queue.
   (b) A double ended queue is also known as deque.
   (c) If the queue is not empty, the items are deleted at the rear end of the queue.
2. Fill in the blanks.
   (a) The end of the queue from which the element is deleted is termed as ……. and the end at which the new element is added is termed as …….
   (b) While inserting an element in a basic queue, the rear pointer is always……
   (c) In a ……. queue, insertion and deletion operations are independent of one another.

(d) In priority queue, the elements are inserted and deleted based on their…….

3. Select a suitable choice for the following questions.

(a) What should be the value of **R** pointer, before inserting elements into the queue?

(i) -1

(ii) 0

(iii) 1

(iv) Is not set to any value

(b) Which among the following is a principle of queue?

(i) Last in First Out

(ii) First in First Out

(iii) Last in Last Out

(iv) First in Last Out

(c) Using which of the following operation, an item is added to the rear end of the queue?

(i) Enqueue

(ii) Dequeue

8.8 Review Questions

1. "In circular queue the insertion and deletion operations are independent of each other." Analyze.
2. "Using double ended queues is more advantageous than using circular queues. "Discuss
3. "Stacks are different from queues." Justify.
4. "Using priority queues is advantageous in job scheduling algorithms. "Analyze
5. Is it possible to insert an element in the front end in a queue? If yes, which method is followed? Discuss
6. Can a basic queue be implemented to function as a dynamic queue? Discuss
7. "It is more advantageous to implement queues using linked lists. "Analyze

Answers: Self Assessment

1.	(a) False	(b) True	(c) False	
2.	(a) Front, Rear	(b) Decremented	(c) Circular	(d) Priority
3.	(a) -1	(b) First in First Out	(c) Dequeue	

8.9 Further Readings

Books

Lipschutz, S. (2011). Data Structures with C. Delhi: Tata McGraw-Hill.

Reddy, P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications.

Online link

http://www.brucemerry.org.za/manual/structures/circular.html

http://www.niitcrcs.com/btpc/btpc-08%20papers%5CIsha-Arrays.pdf

http://www.sqa.org.uk/e-learning/ArrayDS02CD/page_19.htm

Unit 9: Recursion

CONTENTS

Objectives

Introduction

9.1 Fundamentals of Recursion

9.1.1 Definition of Recursion

9.1.2 Types of Recursion

9.2 Anatomy of Recursive Call

9.3 Function Call and Recursion Examples

9.3.1 Factorial of a Number

9.3.2 Fibonacci Series

9.3.3 Tower of Hanoi

9.4 Complexity Issues

9.5 Iteration vs. Recursion

9.6 Summary

9.7 Keywords

9.8 Self Assessment

9.9 Review Questions

9.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the fundamentals of recursion
- Explain the anatomy of recursive call
- Describe function call and recursion examples
- Analyze the complexity issues
- Compare iteration and recursion

Introduction

Recursion is one of the methods that can be used to solve problems related to mathematics, gaming, and so on. The conventional problem solving methods decompose the solution into steps and execute each step one by one. The recursive method of solving a problem breaks the problem into a smaller instance of the same type and solves the smaller instances. High level languages implement recursion by allowing a function to call itself.

Let us consider a real life problem and analyze how recursion can be used to solve it.

Figure 9.1: A Multi-Story Building

Figure 9.1 depicts a multi-story building. When a ten story building is built, nine stories are first built, and then an extra story is added. Similarly, to construct nine stories, the builder first needs to build eight stories and then add an extra story. We can generalize and say that to build an n-story building, n-1 stories need to be built and then one more story must be added. This is a simple example of recursion. This is like saying, "build story" function considers an n-story building and if the value of 'n' is greater than one, it first calls itself to build a lower story and then adds one to build the higher ones.

Recursion is a function that directly or indirectly invokes an instance of itself. In C language, almost all the library functions can be used recursively. Recursion is considered to be an advanced kind of control flow.

Recursive algorithms are mainly used for manipulating data structures, which are defined recursively. When a data object is recursively defined, it is easy to state algorithms that work recursively on such objects. Although some languages like BASIC and COBOL do not have the facility to provide recursion, all the latest programming languages use recursion as their basic iterative control structure.

If our programming language does not permit recursion, it should not matter as we can easily translate a recursive program into a nonrecursive one. Recursion is an inherent property of C language. This unit helps you in understanding how recursion can be implemented using C language.

This unit explains the concept of recursion, which is a problem solving technique. It provides various examples and explanation to design and understand recursion.

Did you know? Recursion is a familiar concept in Mathematics and Logic. For instance, the natural numbers are defined recursively as follows:
'0' is a natural number
If '**n**' is a natural number, then s(n) = (n+1) is a natural number, wherein, '**s**' is a successor function. In this context, recursion is closely related to mathematical induction.

It also discusses about recursive call and complexity issues. It also describes the difference between iteration and recursion.

9.1 Fundamentals of Recursion

The ability of a function to call itself is called recursion. Recursion is an essential concept in Computer Science. Recursion makes it convenient to solve various problems that would be cumbersome to solve using iterative constructs such as for, while, and do while.

In Computer Science and Mathematics, recursion just means self reference. Therefore, a recursive function is a function whose definition is based upon itself. In other words, a function that contains a call statement to itself or a call statement to another function that may eventually result in a call back to the original function is termed as a recursive function.

Recursion defines a problem in terms of itself. A recursive solution repeatedly divides a problem into smaller sub problems till a solvable sub problem is attained. After obtaining the answer for the solvable sub problem, the answer is fed back into the larger sub problem to obtain the solution. This process goes on until we solve the main problem.

Let us now discuss the concept of recursion.

Let 'P' be the original problem. P can be redefined into sub problems like P_1, P_2 and so on. Let P_n be the last sub problem. If P_n sub problem can be defined without any subdivision, then the solution of P_n can be used to solve P_{n-1} sub problem. Further, this solution is fed into P_{n-2}, … P_1 till the original problem 'P' has been solved.

Example: Program to count till the entered number and to display information about the current recursion level

```
/* A preprocessor directive */
#include <stdio.h>

/* Declare an  integer variable named 'level' */
int level;

/* Function declaration of count function that accepts an integer val */
void count (int val)
{
/* Print the count value */
printf ("\n Start counting at level % 2d : val = %2d\n", ++level, val);

/* Checking if variable 'val' is greater than 1 */
if (val>1)

/* A recursive call is made passing the value  (val-1) */
count (val-1);

/* Print variable 'val' value */
printf ("Display val", val);

/* Print level and val. Then decrement the value of level */
printf ("Stop counting at level %2d : val = %2d\n", level--, val);
}

/*Main Program*/
void main ()
{
/* Function count and variable int are declared*/
int val;
void count (int);
printf ("Count till what value?");

/* Accept an integer value and store it in val */
scanf ("%d", &val);
/* Initialize level value to zero */
level = 0;
/* Calling count function passing val as the parameter */
```

```
count (val);
 }
```

Output:

```
Count till what value? 4
Start counting at level 1: val =4
Start counting at level 2: val =3
Start counting at level 3: val =2
Start counting at level 4: val =1
Stop counting at level 4: val =1
Stop counting at level 3: val =2
Stop counting at level 2: val =3
Stop counting at level 1: val =4
```

In this example:

1. First, the **stdio.h** header file is included, using the **include** keyword.
2. Then, an integer variable named level is declared.
3. Then, a function named **count** is declared.
4. In the **count()** function,
   (a) First, increment the **level** value and print the values of **level** and **val**.
   (b) Then, check the value of **val** using an 'if' condition to determine whether it is greater than 1. If the condition is true, a recursive call is made by passing the **value (val-1)**.
   (c) Then, print the value of **val**.
   (d) Finally, decrement the value of **level** and print the values of **level** and **val**.
5. Execution begins at the function **main ()** which does not return any value.
6. In the **main()** function,
   (a) First, initialize an integer variable **val**.
   (b) Then, declare a function named **count**.
   (c) Then, accept an integer value and store it in **val**.
   (d) Then, set the value of **level** to zero.

Finally, call the **count** function using the parameter **val**.

Now let us understand how the recursion process functions. The program starts executing in the main function. Consider that the function count () has val = 4. The count (4) function starts executing while the main function is on hold. This function exhibits the level information which is nothing but the level of recursion. As value of val is equal to 4, which is greater than 1, the function count () is called after decrementing val to 3. Then count (3) exhibits details about the level of recursion. As value of val is equal to 3, which is greater than 1, the function count () is once again called with val=2.

At this point, the main (), count (4) and count (3) are all on hold, and count (2) starts executing. Similar to the previous calls, count (2) exhibits details about the level of recursion. Then, the function checks whether its val value is greater than 1. As the condition is true, the main (), count (4), count (3), and count (2) are on hold, while count (1) starts executing. count (1) exhibits details about the level of recursion. Then the 'if' condition fails and control shifts to the next statement, i.e., to printf () statement to display the value of **val**.

The best feature of the recursive function call is that, the last version called is the first to be performed. Also, the last one called is the first to finish its task. We can see this in the output shown.

9.1.1 Definition of Recursion

A function that calls itself is termed as a recursive function and the process involved in doing this is called recursion. Recursion is a methodology that permits breaking down of a problem into one or many sub problems that are similar to the original problem.

The parameters defined in the function definition are termed as formal arguments. The parameters defined in a caller function and provided in the function call are termed as local variables Every time a function is invoked, a new set of formal parameters and local variables are allocated on the stack. Then, these new values are used to execute from the beginning of the function. A call to itself repeats until the base or terminal condition is achieved. Once the base condition is achieved, the function returns the result of the earlier function. A series of returns ensure that the output to the original problem is obtained.

Example: A simple recursive example

```
/* A preprocessor directive */
#include<stdio.h>
/* printnum function definition */
int printnum (int begin)
{
  /* If begin value is less than 9 */
   if (begin < 9)
    {
      /* Print begin value */
       printf (%d", begin);
      /* printnum function calls itself */
      printnum (begin + 1);
     }
/* Return begin value */
return (begin);
}

/* Main function */
void main ()
{
    /* Initialize integer variables a and c and assign 1 to a */
    int a=1, c;
   /* Clears the output screen */
    clrscr ();
   /* Calls printnum function passing the parameter 1*/
    c = printnum (a);
    /* Waits until a key is pressed */
    getch();
}
```

Output:

12345678

In this example:

1. First, the **stdio.h** header file is included using the **include** keyword.
2. Then, a function named **printnum** is declared which returns an integer value stored in **begin**.
3. In this method,
   (a) First, the value of **begin** using an 'if' condition is checked to determine whether it is less than 9. If the condition is true, **begin**

is printed and the **printnum (begin + 1)** function is invoked.

(b) Finally, the method returns the value of **begin**.

4. Execution begins with the function **main ()** which does not return any value.
5. In the main() function,
   (a) First, integer variables **a** and **c** are initialized and value 1 is assigned to **a**.
   (b) Then, output screen is cleared using **clrscr ()**.
   (c) Then, a parameter named **a,** which holds the value 1, is passed into the **printnum** function. This value is stored in variable **c**.

Finally, the program terminates when any key is pressed.

Notes

The functions that are generally defined recursively are:

1. Factorial
2. Greatest Common Divisor (GCD)
3. Fibonacci
4. Games
   (a) Chess
   (b) Towers of Hanoi

To successfully apply recursion to a problem, you should be able to divide the problem into subparts, which is as shown in the problem given below.

For instance, we can compute the positive exponential power of a^b using recursion.

The recursive definition for finding a positive exponential power of a given number is as follows:

$$Power\ (a,b) = \begin{Bmatrix} 1; if\ b = 0 \\ a * power(a,(b-1)); if\ b > 0 \end{Bmatrix}$$

Here, 'a' is the mantissa and 'b' is the exponent.

In the same manner, you can determine the negative exponential power of number, such as, a^{-b}. The definition of recursion is as follows:

$$Power\ (a,-b) = \begin{Bmatrix} 1; if\ b = 0 \\ 1/(a * power(a,(b-1))); if\ b > 0 \end{Bmatrix}$$

Here, the power is negative, therefore, the inverse return value of a function is considered to be the exponential power of the number. That is, $a^{-b} = 1 / a^b$.

Let us now consider an example that recursively computes power (x, y) such that the value contained in x is raised to y^{th} power.

Example:

A recursive routine that computes the value of 5 raised to the power of 2, i.e., power (5, 2).

```
/* A recursive function that returns the result of the value raised to the power
contained in the variable raised*/
/* It returns 0 or -1 if the raised value is negative*/
/* A preprocessor directive */
```

```
# include <stdio.h>
/* power function definition */
int power (int x, int y)
{
/* Any value that is raised to zero is one */
if (y == 0) return 1;
/* Any value that is raised to 1 returns the same value */
if (y==1) return x;
/* power function calls itself */
return (x * power (x, y – 1));

}
/* Main function */
void main ()
{
/* Initialize integer variables x, y and z */
int x, y, z;
/* Clears the output screen */
clrscr();
/* Print enter the base value */
printf ("\n Enter the base value: ");
/* Accept the base value and store it in x*/
scanf ("%d", &x);
/* Print enter the power */
printf ("\n Enter the Power: ");
/* Accept the exponent value and store it in y*/
scanf ("%d", &y);
/* The power function is called with the arguments x and y*/
/* The value returned is assigned to z*/
z = power (x, y);
/* The value of z is printed*/
printf ("\n %d raised to %d is %d", x, y, z);
/* Waits until a key is pressed */
getch ();
}
```

Output:

The result of power (5, 1) is 5.
The result of power (5, 2) is 25.

In this example:

1. First the **stdio.h** header file is included using the **include** keyword.
2. Then, a function named **power** is declared which holds integer variables **x** and **y**.
3. In the function **power()**,
   (a) First, using an 'if' condition, the value of **y** is checked. If y is equal to zero, the value 1 is returned.
   (b) Then, using an 'if' condition, the value of **y** is checked. If **y** is equal to 1, the value of **x** is returned.
   (c) Finally, **power** function calls itself to return **x** to the **power (x, y-1)**.
4. Execution begins at the function **main ()** which does not return any value.

5. In **main()**,
   (a) First, the integer variables **x**, **y** and **z** are initialized.
   (b) The output screen is cleared using **clrscr ()**.
   (c) The base value is accepted and stored in **x**.
   (d) The exponent value is accepted and stored in **y**.
   (e) The **power** function is called with the arguments **x** and **y**. The value returned is assigned to **z**.
   (f) The value of **z** is printed.
   (g) Finally getch() prompts the user to press any key and the program terminates.

Now let us consider how we solve 5 * power (5, 2). If power() function is called for power(5, 2), the processing performed is as shown below.

power (5, 2) return (5 * power (5, 2-1))
power (5, 1) return (5 * power (5, 1-1))
power (5, 0) return 1 as any value that is raised to zero is one

Task

The expression a % b yields the remainder of 'a' divided by 'b'. Greatest common divisor (GCD) of integers x and y is defined as:

If (y <=x && x % y == 0); gcd (x, y) = y

If (x < y); gcd (x, y) = gcd (y, x)

Otherwise; gcd (x, y) = gcd (y, x % y)

Write a recursive function in C to determine gcd (x, y). Also determine an iterative method to compute this function.

9.1.2 Types of Recursion

When a function invokes itself, it is called as recursion. There are two types of recursion. They are:

1. ***Direct Recursion***: In this kind of recursion, the function invokes itself. Direct recursion involves only one function.
2. ***Indirect Recursion***: Indirect recursion occurs when one method invokes the other, which eventually results in the original method being called again.

Direct Recursion

Direct recursion involves only one function that invokes itself till the specified condition is true. Let us analyze the following program:

Example: A program that performs sum of first five numbers using a call function

```
/* A preprocessor directive to include standard input and output operations */
# include <stdio.h>
/* A preprocessor directive that contains macros and function declaration used in
working with processes and threads */
# include <process.h>
/* Globally declare the main function */
void main (int);
/* Initialize integer variables x and s and assign 0 to s */
int x, s = 0;
```

```
/* Variable x is passed onto the main function */
void main (x)
{
    /* Add values of variable s and x and store in variable s */
    s = s + x;
   /* Print the value of x */
    printf ("\n x = %d s = %d", x, s);
   /* If x value is equal to 5 then terminate successfully */
   if (x==5) exit (0);
    /* main function calls itself which holds post-incremented value of x */
   main (++x);
}
```

Output:

```
x=1 s=1
x=2 s=3
x=3 s=6
x=4 s=10
x=5 s=15
```

In this example:

1. First, the **stdio.h** header file is included using the **include** keyword.
2. Then, the **process.h** header file that contains macros and function declaration are included using the **include** keyword.
3. Then, the **main()** function is declared globally.
4. Then, integer variables **x** and **s** are initialized and 0 is assigned to **s**.
5. Execution begins with the **void main (x)** function which does not return any value, but a variable **x** is passed onto it.
6. In this **main()** function,
   (a) First, the values of variable **s** and **x** are added and stored in variable **s**.
   (b) Then, the value of **x** is printed.
   (c) Then, using an 'if' condition the value of **x** is checked. If x is equal to 5, the loop is terminated successfully.
   (d) Finally, the **main** function calls itself with the post-incremented value of **x**.

Indirect Recursion

In indirect recursion, two or more functions are involved in recursion. When control passes from one function and enters into another function, the former function's local variables are destroyed.

Example: A program that demonstrates indirect recursion between two functions.

```
/* A preprocessor directive that contains standard input and output operations */
# include <stdio.h>
/* A preprocessor directive that creates text user interfaces */
# include <conio.h>
/* A preprocessor directive that contains macros and function declaration used in
working with processes and threads */
# include <process.h>

/* Initialize integer variable s globally*/
```

```
int s;
/* The show function is defined globally which does not return any value */
void show (void);
/* main function */
main ()
{
   /* If s value is equal to 5, the program execution comes to an end*/
   if (s==5) exit (0);
   /* show function is called */
   show ();
}
/* show function definition */
void show ()
{
   /* Print the value of s */
   printf ("%d", s);
  /* Increment the value of s */
   s++;
   /* main function is called */
   main ();
 }
```

Output:
0 1 2 3 4

In this example:

1. First the **stdio.h** header file is included using the **include** keyword.
2. Then, the **conio.h** header file that creates text user interfaces are included, using the **include** keyword.
3. Then, the **process.h** header file is included using the **include** keyword.
4. Then, the integer variable **s** is declared globally and initialized.
5. Then, **show** function is defined globally which does not return any value.
6. Execution begins in the **main ()** function.
7. In **main()**,
   (a) First, using an 'if' condition, the value of **s** is checked. If the value is equal to 5, the program execution comes to an end.
   (b) Finally, the **show** function is invoked.
8. In the **show** function,
   (a) First, the value of **s** is printed.
   (b) Then, the value of **s** is incremented.
   (c) Finally, the **main** function is invoked.

9.2 Anatomy of Recursive Call

Recursive function comprises two types of cases. They are:

1. A base case
2. A recursive case

A base case is the smallest instance of the problem whose solution is not recursive. It guarantees the termination of a function. Even a small problem can have many base cases.

A recursive case defines a problem in terms of smaller problems of similar type. Recursive case comprises a recursive function call. A problem can consist of many recursive cases.

To determine recursive solution for any kind of problem, follow the steps given below:

1. Define a problem in terms of a smaller problem of similar type.
2. Define recursive part. In the power() function shown in 9.1.1, the statement 'return (x * power (x, y – 1))'; is the recursive part.
3. Define base case even for a small problem where solution can be calculated easily.

To attain recursive solutions, follow the steps given below:

1. Analyze how the problem can be defined in terms of smaller problems.
2. Determine how each of the recursive calls help to divide the problem into sub-problems.
3. Analyze the base case which can be solved without recursion.
4. Find whether the base case can be attained or not when the problem is broken.

Caution Be careful while using recursive step. If the terminal or base condition is omitted and the recursive step is wrongly used, then it will result in infinite recursion that exhausts the memory slowly.

Let us design a recursive solution for writing a string backwards.

Problem: Write a string of characters in reverse order.

Recursive solution: Write the last character of string and solve the problem by writing the first (n-1) character backward. Each recursive solution should increment the string pointer of the string that needs to be written backwards. Then, define the base case by writing an empty string backward which does not do anything. As the problem reduces, the base case is attained.

Example: Program to write a string of characters in reverse order

```
/* A preprocessor directive that contains standard input and output operations */
#include<stdio.h>
/* print_reverse is a recursive function which takes variable S */
print_reverse (char *S)
/* S is a pointer which points to a character */
{
   /* Check whether the string pointer value is not equal to NULL*/
   if (*S != NULL)
     {
          /* print_reverse function calls itself after incrementing the pointer by 1
*/
          print_reverse (++S);
         /* putchar writes a single character to the standard output stream, that
is, the value of S is decremented and that string pointer is written*/
          putchar (* (--S));
     }
/* Main function */
```

```
main ()
{
    /* A character array S whose size is 80 is declared */
    char S[80];
    printf ("\n Enter a string: \n");
    /* gets function declared in stdio.h header file reads a line from standard input and stores it in S. */
    gets (S);
    printf("\nThe reversed string: \n");
    /* print_reverse function is called with the argument S */
    print_reverse (S);
    /* Waits until a key is pressed */
    getch();
}
```

Output:

Enter the string to be reversed: english

Output String : hsilgne

In this example:

1. First, the **stdio.h** header file is included using the **include** keyword.
2. Then, a recursive function named **print_reverse** is declared. The function takes a parameter **S,** which is a character pointer.
3. In the **print_reverse** () method:
   (a) First, using an 'if' condition, the value of string pointer **S** is checked. If the value is not equal to NULL, a recursive call is made by passing the incremented value of **S**.
   (b) Finally, the value of **S** is decremented and the string pointer value is written using **putchar()**.
4. Then, the **main ()** function is executed.
5. Then, a character array **S** whose size is 80, is declared.
6. Then, **gets** function is used to read a line from the standard input and this is stored in **S.**
7. Then, **print_reverse** function is invoked with the argument **S**.
8. Finally, getch() prompts the user to press any key and when the key is pressed, the program is terminated.

Task

Use the function explained below to identify the following:

1. The base case of the function Sample
2. The recursive case of the function Sample

```
int Sample (int base, limit)
{
   if (base > limit)  return -1;
   else
      if (base == limit) return 1;
      else
      return base * Sample (base+1, limit);
}
```

Notes

1. Base case computes one or more specific numbers (usually 0 or 1) for which the result can be attained immediately.
2. Recursive case computes result by calling recursively the function with a small argument and using the result to obtain the final answer.

9.3 Function Call and Recursion Examples

A function is a block of statements that can be utilized to perform a particular task. A function comprises the following parts:

1. Function prototype declaration
2. Function declarator
3. Actual and formal arguments
4. Return statement
5. Invoking or calling function

Function Prototype Declaration

The functional prototypes are provided in the beginning of the program after the # include statement. The function prototype declaration comprises the return type, arguments list, and name of the function.

Function Call

A function is activated only when a function call is invoked. A function should be invoked by its name along with the argument list enclosed within parenthesis and terminated with a semi-colon.

Actual and Formal Argument

The arguments defined in a caller function and provided in the function call are termed as actual arguments. The arguments defined in the function definition are termed as formal arguments.

Return Statement

Return statement returns value to the caller function. Return statement returns just one value at a time. When the compiler encounters a return statement, the control of the program is transferred to the caller function.

Let us see how recursion is implemented.

When a function is invoked, the run-time system allocates memory spaces dynamically for storing parameters, variables, and constants that are defined within the function. Every function stores a return address.

Example: Program to illustrate the usage of stack

```
/* Recursive Call */
/* A preprocessor directive that contains standard input and output operations */
# include <stdio.h>
/* Main function */
main ()
{
   /* Initialize x as an integer variable with static as its storage class and assign
   0 to it. Static storage refers to the memory locations which persist
   Throughout the lifetime of the program. */
   static int x = 0;
   /* Increment the value of x */
   x++;
   /* Condition statement: If x is less than 7, the main function itself is
   invoked otherwise  x value is incremented */
   x < 7 ? main () :  x++;
   /* Print the value of x */
   printf ("%2d", x);
 }
```

Output:

8 8 8 8 8 8 8

In this example:

1. First the **stdio.h** header file is included using the **include** keyword.
2. Then, the **main** function is defined.
3. In **main()**:
   (a) First, x is initialized as an integer variable with static as its storage class and 0 is assigned to it.
   (b) Then, x is incremented.
   (c) Then, using an 'if' condition, the value of x is checked. If x is less than 7, the **main** function itself is invoked. Otherwise, the value of **x** is incremented.
   (d) Finally, the value of x is printed.

9.3.1 Factorial of a Number

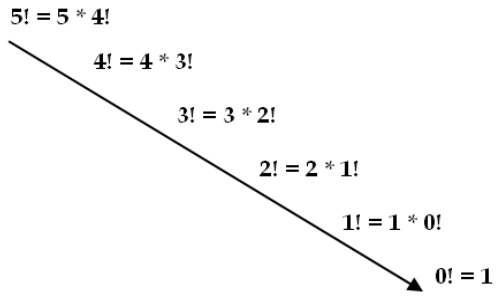
Calculation of factorial of a number is another program that can be written recursively. Let us discuss how to write a program to determine the factorial of a number. Factorial of a number 'n' can be determined using the series: 1*2*3*…n = n!

The recursive definition to determine factorial of n is given below:

$$\text{Fact (n)} = \begin{Bmatrix} 1; \text{ if } n = 0 \\ n * \text{fact}(n-1); \text{ otherwise} \end{Bmatrix}$$

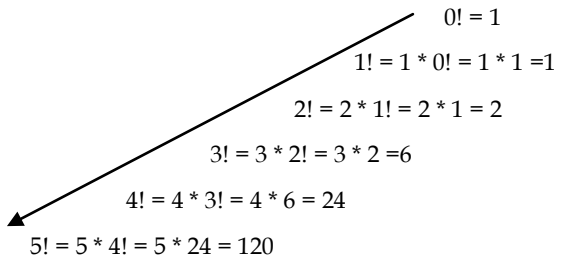
By definition, 1 is the value for 0!. Let us compute the factorial of 5. Figure 9.2 depicts the factorial of 5.

Figure 9.2: Factorial of 5



But computations will be done in reverse order as depicted in figure 9.3.

Figure 9.3: Computations for Factorial of 5 in Reverse Order



Example: Program that computes the factorial of a number using recursion.

```
#include<stdio.h>
/* fact is a recursive function which takes an integer variable num and returns
an integer value */
int fact (int num)
{
   /* Check if num value is equal to zero */
   if (num==0)
        /* Return the  value 1 */
        return 1;
   /* Otherwise */
   else
       /* Return num * fact (num-1) value */
       return (num * fact (num-1));
 }
/* Main function */
```

```
void main ()
{
   /* Initialise an integer variable num */
   int num;
    printf ("Enter the number: \n");
   /* Accept an integer value num */
   scanf ("%d", &num);
   /* Print the value of fact (num) */
   printf (" The factorial of %d = %d\n", num, fact (num));
}
```

Output:
Enter the number: 5
The factorial of 5 = 120

In this example:

1. First, the **stdio.h** header file is included using the **include** keyword.
2. Then, a recursive function **fact** is declared which takes an integer variable **num** and returns an integer value.
3. In this method,
   (a) The value of **num** is checked using an 'if' condition to determine whether it is equal to 0. If the condition is true, value 1 is returned.
   (b) Otherwise, the fact function invokes itself by return **(num * fact (num-1))**.
4. The **main ()** program is then defined. It does not return any value.
5. In **main ():**
   (a) First, an integer variable **num** is initialized.
   (b) Then, an integer value **num** is accepted.
   (c) Finally, the function **fact (num)** is invoked and the value is printed and returned.

Suppose the value of 'num' is 4 when fact (num) is executed, then the value of 'num' along with a return address of say 1000, stored in the program counter is pushed onto the stack. Table 9.1 (a) depicts the stack at this point. As 'num' is not zero, (num-1) is set to 3. fact () function is invoked with (num-1) as the parameter. Prior to entering into the fact () function, assume that the return address is 2000 and the local parameters are pushed onto the stack as depicted in table 9.1 (b). Each time the fact () function is invoked recursively, a new set of local parameters will be pushed onto the stack. When fact () is invoked recursively till the value of 'num' becomes zero, the value of num is decremented by 1. Table 9.1 (a) through 9.1 (e) depicts the stack every time the fact () function is called.

When the value of 'num' becomes zero, the return statement with value 1 is returned. This is termed as base or terminal condition. Once the return statement executes, the top most stack variables are popped and the control moves to the original point from where it was invoked and value 1 is copied into fact (num-1).

The new values of num, num-1, fact (num-1) and (num * fact (num-1)) are depicted in table 9.1 (f) as the execution proceeds.

Table 9.1: Illustrations of Stack while Determining fact (4)

4	…	…	…	1000
num	num-1	fact(num-1)	num*fact(num-1)	Program Counter

fact(4) in main()
(a)

4	3	…	…	2000
4	…	…	…	1000
num	num-1	fact(num-1)	num*fact(num-1)	Program Counter

fact(4)
(b)

3	2	…	…	2000
4	3	…	…	2000
4	…	…	…	1000
num	num-1	fact(num-1)	num*fact(num-1)	Program Counter

fact(3)
(c)

2	1	…	…	2000
3	2	…	…	2000
4	3	…	…	2000
4	…	…	…	1000
num	num-1	fact(num-1)	num*fact(num-1)	Program Counter

fact(2)
(d)

1	0	…	…	2000
2	1	…	…	2000
3	2	…	…	2000
4	3	…	…	2000
4	…	…	…	1000
num	num-1	fact(num-1)	num*fact(num-1)	Program Counter

(e)

num	num-1	fact(num-1)	num*fact(num-1)
1	0	1	1
2	1	1	2
3	2	2	6
4	3	6	24

(f)

Finally, the control passes onto the main () program. The value 24 is copied into (num * fact (num-1)). A function can be invoked many times to solve a problem which is termed as depth of recursion. 'num' is called the depth of recursion to calculate fact (num).

9.3.2 Fibonacci Series

Fibonacci series is a series of numbers represented as 0, 1, 1, 2, 3, 5, 8, 13, 21, …. Fibonacci series starts with 0 and 1. Each of the subsequent Fibonacci number is computed as the sum of the previous two Fibonacci numbers.

Did you know? The ratio of consecutive Fibonacci numbers is equal to 1.618. This repeatedly occurs in nature. This number has been termed as golden ratio or golden mean. Architects often design buildings, rooms and windows whose width and length are in ratio of the golden mean.

Fibonacci series is defined recursively as:

fibonacci (0) = 0

fibonacci (1) = 1

fibonacci (n) = fibonacci (n-1) + fibonacci (n-2)

Let us calculate recursively the nth Fibonacci number using Fibonacci () function.

Example: Program to calculate recursively the nth fibonacci number using fibonacci () function.

```
/* Recursive fibonanacci function */
/* A preprocessor directive to include standard input and output operations */
# include <stdio.h>
/* Globally declare the fibonacci function */
long fibonacci (long n);
/* Main function */
int main (void)
{
/*Declare a long variable named result */
long result;
/*Declare a long variable named number */
long number;
printf (“Input an integer: “);
/* Accept a long variable number */
scanf (“%d”, & number);
/* Fibonacci function is called and its value is stored in a variable named result */
result = fibonacci (number);
/* Print the value of result */
printf (“Fibonacci (%1d) = %ld\n”, number, result);
/* Return 0 */
return 0;
}
```

```
/* Fibonacci recursion function */
long fibonacci (long n)
{
/* Check if the value of n is equal to 1 */
if (n==1)
{
/* Return 0 */
return 0;
}
/* Check if the value of n is equal to 2 */
if (n==2)
{
/* Return 1 */
return 1;
}
/* Otherwise */
else
{
/* Fibonacci function is invoked within itself and the value is returned */
return fibonacci (n-1) + fibonacci (n-2);
}
}
```

Output:

Input an integer: 5

Fibonacci (5) = 0, 1, 1, 2, 3

In this example:

1. First, the **stdio.h** header file is included using the **include** keyword.
2. Then, **fibonacci** function is declared globally. It returns and accepts a long data type.
3. Then, the **int main (void)** program is defined. This returns an integer value.
4. In **main ()**:
   (a) First, initialize the long variables **result** and **number.**
   (b) Then, accept the value for **number.**
   (c) Then, call the **fibonacci** function and store its value in a variable **result.**
   (d) Then, print the value of **result**.
   (e) Finally, return the value 0.
5. Then the function **fibonacci** is defined.

6. In this function:

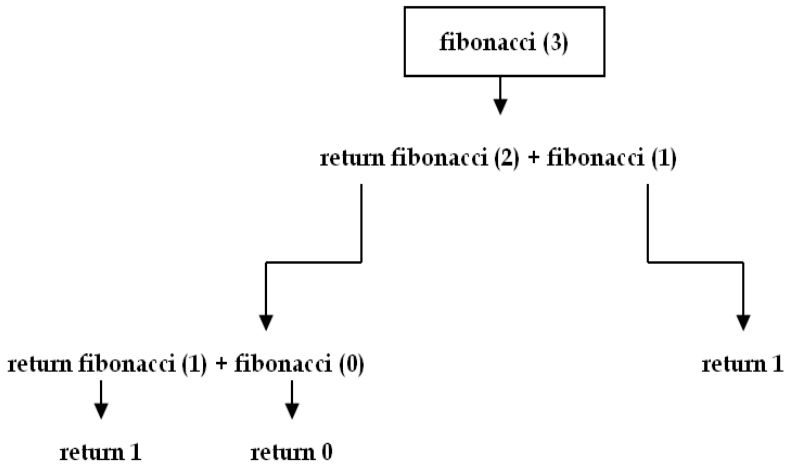
   (a) First, check the value of **n** using an 'if' condition to determine whether it is equal to 1. If the condition is true, the value 0 is returned.

   (b) Then, check the value of **n** using an 'if' condition to determine whether it is equal to 2. If the condition is true, the value 1 is returned.

   (c) Otherwise, finally invoke the **fibonacci** function within itself and return the value of **fibonacci (n-1) + fibonacci (n-2)** to the **main** function from where it is called.

In fibonacci () function, the statement 'if (n==1) return 0;' is the base case and the statement 'if (n>1), then return fibonacci (n-1) + fibonacci (n-2);' is the recursive step.

Example: Illustration of how fibonacci () function evaluates fibonacci (3).



Caution In situations where performance is a criterion, avoid the use of recursion. Recursive calls consume more time and additional memory.

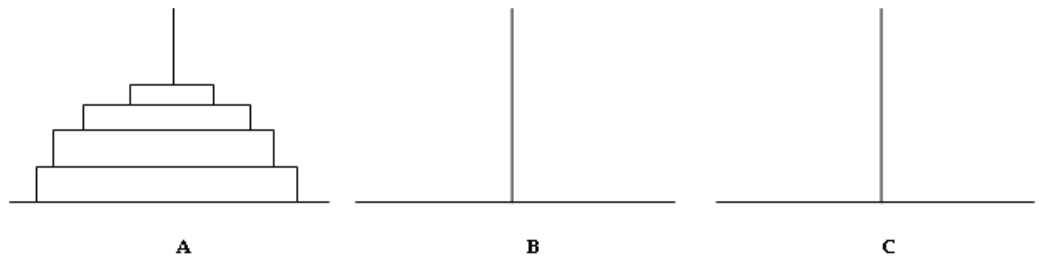
9.3.3 Tower of Hanoi

Tower of Hanoi is a traditional game, which is an example of a problem that can be solved using recursion. In the simple Tower of Hanoi problem, there exist three pegs namely, A, B and C. On peg A, there are 'n' discs of varied diameters that are placed one above the other, such that the smaller disc is always placed above the larger disc. Two pegs 'B' and 'C' are empty. Then, all discs in peg A are transferred to peg C with the help of peg B as the temporary storage. Following are the rules that need to be considered while moving the discs:

- Only one disc can be transferred at a time.
- The smaller disc must be on the top of a larger disc every time.

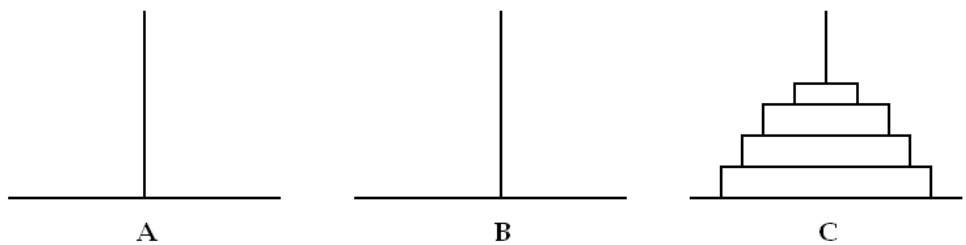
The initial set up of the Tower of Hanoi problem is depicted in figure 9.4. Let us analyze the set up.

Figure 9.4: Initial Set up of Tower of Hanoi



Once all the discs are transferred from 'A' to 'C', we get the setup as shown in Figure 9.5.

Figure 9.5: Set up of Tower of Hanoi After Transferring Discs



To transfer 'n' discs from 'A' to 'C', the recursive method comprises three steps:

1. Transfer (n-1) discs from 'A' to 'B'.
2. Transfer n^{th} disc from 'A' to 'C'.
3. Transfer (n-1) discs from 'B' to 'C'.

Example: Program that computes the Tower of Hanoi using recursion.

```
/* A preprocessor directive to include functions related to user interfaces
*/
# include <conio.h>
/* A preprocessor directive to include functions related to standard
input and output operations */
# include <stdio.h>

/* Hanoi function is defined globally which holds four arguments and
does not return any value */
void Hanoi (int, char, char, char);
/* Main function */
main ()
{
   /* Declare an integer variable n */
   int n;
   printf ("\n Input the number of discs:");
   /* Accept the integer value n */
   scanf ("%d", &n);
   /* Hanoi function is invoked */
```

```
   Hanoi (n, 'A', 'B', 'C');
 }

/* Hanoi function definition holds four arguments namely n, source,
dest and spare which are of integer and character data types respectively
*/
void Hanoi (int n, char source, char dest, char spare)
{
    /* Check whether n value is equal to 1 */
   if (n == 1)
    {
       /* Prints the values of source and dest */
       printf ("\n Transfer disc 1 from needle %c to needle %c", source,
dest);

    }
   /* Otherwise */
    else
   {
     /* Hanoi function is invoked within itself with n-1, source, spare and
dest variables*/
     Hanoi (n-1, source, spare, dest);
     /* Print the values of variables n, source and dest */
     printf ("\n Transfer disk %d from needle %c to needle %c", n,
source, dest);
     /* Hanoi function is invoked within itself with n-1, spare, dest and
source variables*/
     Hanoi (n-1, spare, dest, source);
   }
}
```

Output:

```
Input the number of discs: 3
Transfer disk 1 from needle A to needle C
Transfer disk 2 from needle A to needle B
Transfer disk 1 from needle C to needle B
Transfer disk 3 from needle A to needle C
Transfer disk 1 from needle B to needle A
Transfer disk 2 from needle B to needle C
Transfer disk 1 from needle A to needle C
```

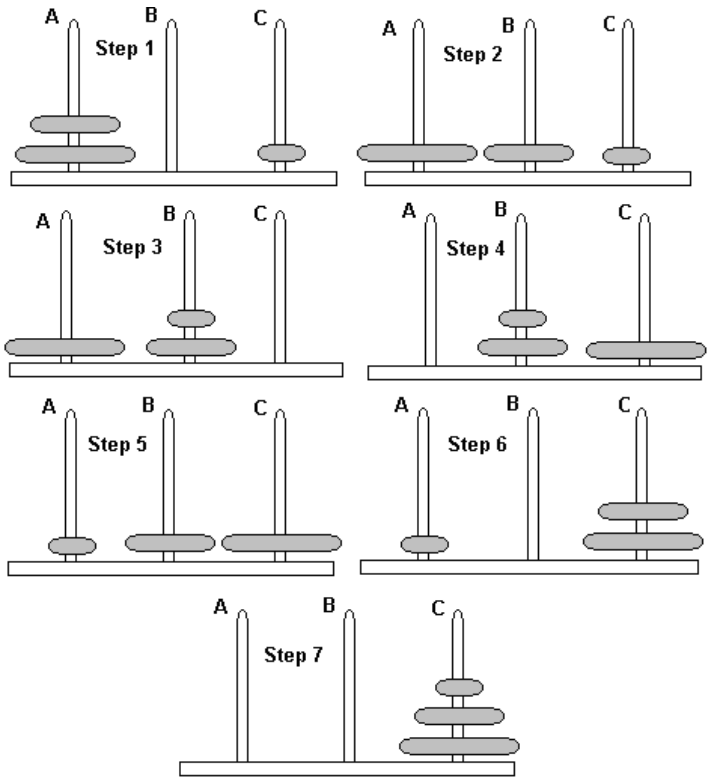
In this example:

1. First, the **conio.h** header file that creates text user interfaces are included using the **include** keyword.
2. Then, the **stdio.h** header file is included using the **include** keyword.
3. Then, **Hanoi** function is defined globally which holds four arguments and does not return any value.
4. Then, the **main ()** program is defined.
5. In **main ()**:
   (a) First, an integer variable n is initialized.
   (b) Then, the number of discs is accepted and stored in **n**.
   (c) Finally, the **Hanoi** function is invoked.

6. Then, a function named **Hanoi** is defined.
7. In this function,
   (a) First, using an 'if' condition, the value of n is checked to determine whether it is equal to 1. If the condition is true, the values of **source** and **dest** are printed.
   (b) Otherwise, the **Hanoi** function is invoked within itself with **n-1**, **source**, **spare** and **dest** variables. Then, the values of variables **n**, **source** and **dest** are printed. Then, Hanoi function is invoked within itself with **n-1**, **spare**, **dest** and **source** variables.

The Hanoi () function performs as shown in figure 9.6.

Figure 9.6: Working of Hanoi Function



9.4 Complexity Issues

Recursive procedures or algorithms invoke themselves and therefore their running time is described using a recursive equation.

The recursive equation is as given below:

T(n) = Time_for (iterative part, n) + Time_for (recursive part, n)

Where,

T(n) is the running time for computing any recursive function.

Time_for is the running time for the parts of the algorithm. The Time_for depends on the input data size for the computed recursive function.

Therefore, the recurrence relation is as given below:

T (n) = Time_for (Preprocess, n) + Time_for (Divide, n) + Time_for (Combine, n) + T(n1) + …. + T (nk).

Where, n1 to nk is less than n.

Example: If a recursive algorithm takes 'c' steps and decreases the parameter by **1**, then we may express the running time as a function, i.e., time (N). Here, time (N) represents the running time on **N** number of input elements.

time (N) = c + time (N-1)

If N =1, then time (1) = c

This equation is easy to solve and the solution is as follows:

time (N) = c + time (N-1)

c + c + time (N-2)

= c + c + … + time (1)

= c + c + … + c (N times)

= c * N

9.5 Iteration vs. Recursion

Let us study iteration and recursion. They are applied to a program as per the situation. Table 9.2 discusses the difference between iteration and recursion.

Table 9.2: Iteration vs. Recursion

Iteration	Recursion
Uses repetitive structures like **for**, **while** or **do while** loop and uses them explicitly.	Uses selection structures like **if**, **if else** or **switch** statement and achieves repetition with the help of repeated function calls.

The body loop terminates when the termination condition fails. Each time control passes into the loop, the counter is updated.
Example:
int factorial (int n)
{
int j, prod = 1;
for (j = 0; j < n; j++)
prod = prod * j;
return prod;
}
In the example program, **int factorial (int n)**, the function prototype is
	The body loop terminates when the base condition is satisfied. The base condition is achieved by invoking the same function. Every time the function is invoked, a simple version of the original problem is obtained until base condition is achieved.
	Example:
	int factorial (int n)
	{
	If (n==0)
	return 1;
	else
	return (n*factorial(n-1));
	}
	In the example program, **int factorial**

Contd..

declared. Then variable '**j**' and '**prod**' are defined as integer. And prod is assigned a value '1'. **prod = (prod * j)** is calculated until the value of '**j**' is lesser than the value of '**n**'. Finally, the program returns the value of '**prod**'.	**(int n)**, the function prototype is declared. If the value of '**n**' entered is '0', then value '1' is returned. Otherwise, **(n*factorial (n-1))** value is returned until the value of '**n**' becomes zero.
Iterative functions can be designed easily. They occupy less memory and execute much faster.	Every time a function is invoked, all the formal parameters, local variables, and return address are pushed onto the stack. Therefore, it occupies more space in the stack and more time is consumed in pushing and popping. Hence, recursion is costly in terms of memory usage and processor time.
Iteration is not suitable for problems like tree traversals, Tower of Hanoi and so on. Although these problems can be solved with the help of iterative functions, they are difficult to design, time-consuming and more error prone.	Recursion is suitable for problems like tree traversal techniques, Tower of Hanoi, and so on. Recursive functions are easily understandable and efficient.

Lab Exercise

1. Write a recursive function to add the first 'n' natural numbers. [Hint: (addition) = 1 + 2 + … + n]].
2. Write a recursive function to determine the sum of digits of a number that is entered through a keyboard.

9.6 Summary

- Recursion is a function that directly or indirectly invokes an instance of itself.
- Recursion procedures solve a given problem by breaking down the problem into an instance of the same problem.
- The two types of recursion are - direct and indirect recursion. Direct recursion is a kind of recursion in which the function invokes itself. Indirect recursion is a kind of recursion in which two functions invoke each other.
- Recursive function comprises two types of cases namely, base case and recursive case. Base case is the smallest instance of the problem and its solution should not be recursive to guarantee function termination. Recursive case comprises a recursive function call.
- Function is a block of statements that can be utilized to perform a particular task. A function is activated only when a function call is invoked.
- Factorial, Fibonacci series and Tower of Hanoi are few examples of problems that can be solved using recursion.
- Recursive procedures or algorithms invoke themselves. Therefore, the running time is described using a recursive equation.
- Iterative functions can be designed easily. They occupy less memory and execute much faster, whereas, recursion is costly in terms of memory usage and processor time.

9.7 Keywords

Actual Arguments: The arguments defined in a caller function and provided in the function call are termed as actual arguments.

Formal Arguments: The arguments defined in the function definition are termed as formal arguments.

Mantissa: It is the part of a floating-point number that contains its significant digits.

Stack: Stack is an abstract or a last in first out (LIFO) data type and data structure. It is characterized by two fundamental operations like push and pop.

9.8 Self Assessment

1. State whether the following statements are true or false:
   (a) The actual arguments are defined in the function definition.
   (b) Recursive algorithms are mainly used for manipulating data structures which are defined recursively.
   (c) If a function is invoked with a base case, a result is returned as an output from the function.
   (d) In the Tower of Hanoi problem, multiple discs can be transferred at a time.
   (e) To transfer 'n' discs from 'A' to 'C' in Tower of Hanoi, the recursive method firstly involves the transfer of (n-1) discs from 'A' to 'B'.
   (f) When a function is invoked, the run-time system allocates memory spaces dynamically for storing parameters, variables, and constants that are defined within it.
2. Fill in the blanks:
   (a) …… starts with 0 and 1 and computes the subsequent numbers using the sum of the previous two numbers.
   (b) …… is a traditional game and an example of recursion.
   (c) In…… the body loop terminates when the termination condition fails.
   (d) …… is a block of statements that can be utilized to perform a particular task.
   (e) …… is costly in terms of memory usage and processor time.
3. Select a suitable choice for every question:
   (a) Which among the following is a function that invokes an instance of itself directly or indirectly?
      (i) Iteration
      (ii) Recursion
      (iii) Repetition
      (iv) Duplication
   (b) What does recursion mean?
      (i) Self reference
      (ii) Self iteration
      (iii) Self repetition
      (iv) Self rehearsal

(c) Which among the following cannot be defined recursively?

(i) Factorial

(ii) GCD

(iii) Fibonacci

(iv) Matrix multiplication

(d) Which one of the following must be done to terminate recursion?

(i) The smaller problems must match with the base case.

(ii) The smaller problems must match with the recursive case.

(e) Which of the following involves only one function that invokes itself till the specified condition is true?

(i) Indirect recursion

(ii) Direct recursion

(iii) Tail recursion

(iv) Mutual recursion

9.9 Review Questions

1. Analyze various situations where recursion can be used.
2. "Recursion is an entirely different method of solving problems." Justify.
3. "Recursive function comprises two types of cases namely, a base case and a recursive case." Elaborate.
4. What is the functional aspect of return keyword in a recursive step? Discuss in brief.
5. "float sum (float, int) is a function prototype declaration that comprises the return type, arguments list and name of the function." Discuss.
6. "Recursion occurs when a function invokes itself recursively." Discuss the types of recursion and its usefulness while programming.
7. "Function gets activated only when a function call is invoked." Discuss.
8. Using recursion in Tower of Hanoi game, how many steps would you need to shift the 'n' discs from 'A' to 'C'?
9. "T (n) = Time_for (iterative part, n) + Time_for (recursive part, n) is a recursive equation that describes the running time of recursive procedures or algorithms". Elaborate.
10. "For a small problem where solution can be calculated easily, the solution should never be recursive." Elaborate.
11. "Recursive algorithms are mainly used for manipulating data structures which are defined recursively." Justify.
12. "If a function is called with a complex problem, the function breaks the problem into a sub-problem that is similar to the original problem." Discuss.

Answers: Self Assessment

1. (a) False (b) True (c) True (d) False (e) True (f) True
2. (a) Fibonacci series (b) Tower of Hanoi (c) Iteration (d) Function (e) Recursion
3. (a) Recursion (b) Self reference (c) Matrix multiplication (d) The smaller problems must match with the base case. (e) Direct recursion

9.10 Further Readings

Books

Reddy. P. (1999). Systematic Approach to Data Structures Using C. Bangalore: Sri Nandi Publications.

Bandyopadhyay. S. K., Dey. K. N. (2009). Data Structures Using C. New Delhi, India: Dorling Kindersley.

Kamthane. (2007). Introduction to Data Structures in C. New Delhi, India: Dorling Kindersley.

Gupta. P., Agarwal. V., Vashney. M. (2007). Data Structure Using C. New Delhi: Firewall Media.

Online link

my.safaribooksonline.com/book/programming/c/9780136085881

www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-3

www.devshed.com/c/a/Practices/Solving-Problems-with-Recursion

www.cs.sfu.ca/~tamaras/recursion/Direct_vs_Indirect.html/

Unit 10: Trees

CONTENTS

Objectives

Introduction

10.1 Trees

10.1.1 Representation of Tree in Graphs

10.1.2 Types of Graphs

10.2 Types of Trees

10.2.1 Binary Tree

10.2.2 Binary Search Tree

10.2.3 2-3 Trees

10.2.4 Huffman Trees

10.3 Representation of Tree in Memory

10.4 Application of Trees

10.4.1 Expression Trees

10.4.2 Game Trees

10.4.3 Decision Trees

10.5 Summary

10.6 Keywords

10.7 Self Assessment

10.8 Review Questions

10.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Define trees
- Discuss the types of trees
- Explain the representation of tree in memory
- Discuss the application of trees

Introduction

We know that data structure is a set of data elements grouped together under one name. A data structure can be considered as a set of rules that hold the data together. Almost all computer programs use data structures. Data structures are an essential part of algorithms. We can use it to manage huge amount of data in large databases. Some modern programming languages emphasize more on data structures than algorithms.

There are many data structures that help us to manipulate the data stored in the memory, which we have discussed in the previous units. These include array, stack, queue, and linked-list.

Choosing the best data structure for a program is a challenging task. Similar tasks may require different data structures. We derive new data structures for complex tasks using the already existing ones. We

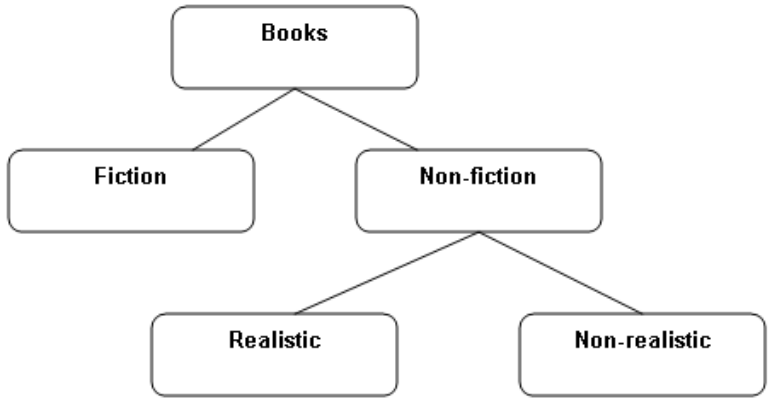
need to compare the characteristics of the data structures before choosing the right data structure. A tree is a hierarchical data structure suitable for representing hierarchical information. The tree data structure has the characteristics of quick search, quick inserts, and quick deletes.

10.1 Trees

A tree is a very important data structure as it is useful in many applications. A tree structure is a method of representing the hierarchical nature of a structure in a graphical form. It is termed as "tree structure" since its representation resembles a tree. However, the chart of a tree is normally upside down compared to an actual tree, with the root at the top and the leaves at the bottom.

The figure 10.1 depicts a tree structure, which represents the hierarchical organization of books.

Figure 10.1: Tree Structure of Books



In the hierarchical organization of books shown in figure 10.1, **Books** is the root of the tree. **Books** can be classified as **Fiction and Non-fiction. Non-fiction** books can be further classified as **Realistic and Non-realistic,** which are the leaves of the tree. Thus, it forms a complete tree structure.

Trees are primarily treated as data structures rather than as data types.

A tree is a widely-used data structure that depicts a hierarchical tree structures with a set of linked nodes. The elements of data structure in a tree are arranged in a non-linear fashion i.e., they use two dimensional representations. Thus, trees are known as non-linear data structures. This data structure is more efficient in inserting additional data, deleting unnecessary data, and searching new data.

Notes

Genealogies and organizational charts are the most familiar applications of trees.

Few other applications of tree data structures are as follows:

1. To analyze electrical circuits
2. To represent the structure of mathematical formulas
3. To organize information in database systems
4. To represent syntactic structure of source programs in compilers

10.1.1 Representation of Tree in Graphs

A graph **G** consists of a set of objects **V** = {v1, v2, v3 …} called vertices (points or nodes) and a set of objects **E** = {e1, e2, e3 ….} called edges (lines or arcs).

The set V (G) is called the vertex set of **G** and E (G) is the edge set.

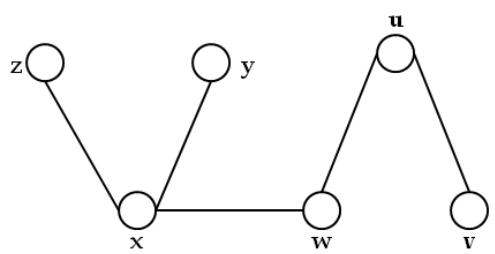
The graph is denoted as G = (V, E)

Let **G** be a graph and {u, v} an edge of **G**. Since {u, v} is 2-element set, we write {v, u} instead of {u, v}. This edge can be represented as uv or vu.

If e = uv is an edge of a graph **G**, then u and v are adjacent in **G** and e joins u and v.

Consider the graph in figure 10.2.

Figure 10.2: Graph



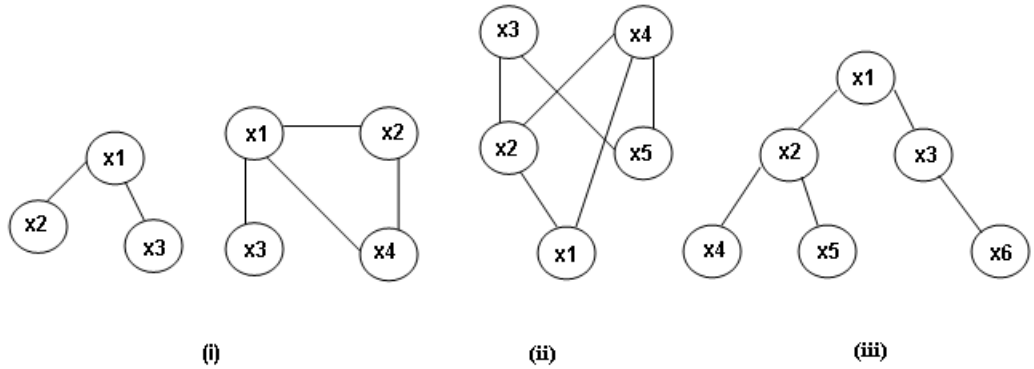
This graph G is defined by the sets:

V (G) = {u, v, w, x, y, z} and E(G) = { uv, uw, wx, xy, xz}

Every graph has a diagram associated with it. The vertex **u** and an edge **e** are incident with each other as are **v** and **e**. If two distinct edges **e** and **f** are incident with a common vertex, then they are adjacent edges.

Figure 10.3 depicts three examples of graphs. Graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes.

Figure 10.3: Examples of Graph



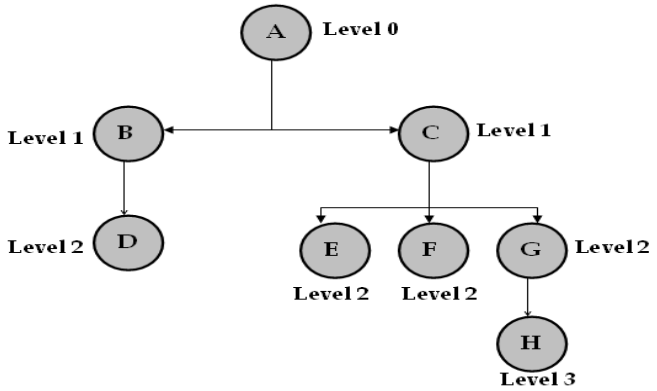
In figure 10.3, the graph (i) has two different, unconnected set of nodes. Graphs can also contain cycles. Graph (ii) has several cycles. One such path is from x1 to x2 to x4 and back to x1. Another one is from x1 to x2 to x3 to x5 to x4 and back to x1. (There are also cycles in graph (ii).) Graph (iii) does not have any cycles and all nodes are reachable. Therefore, it is a tree.

Did you know? Programs like Microsoft MapPoint that can generate driving directions from one city to another, use graphs, where the modeling cities are represented as nodes in a graph and the roads connecting the cities as edges.

Trees can be represented as graphs.

Figure 10.4 represents a tree, which is a non-linear representation of data.

Figure 10.4: A Tree Structure



Let us now discuss some of the common terminologies used with respect to trees.

Root Node

The root of a tree is called a root node. In figure 10.4, **A** is the root node. A root node occurs only once in the whole tree.

Parent Node

The parent of a node is the immediate predecessor of that node. In the figure 10.4, **A** is the parent node of **B** and **C**, **B** is the parent node of **D,** and **C** is the parent node of **E**, **F** and **G. G** is the parent node of **H**.

Child Node

Child nodes are the immediate successors of a node.

In the figure 8.4, **B** and **C** are the child nodes of **A**, **D** is the child node of **B; E, F,** and **G** are the child nodes of C and finally, **H** is the child node of **G**.

Leaf Node

A node which does not have any child nodes is known as a leaf node. In figure 10.4, **D**, **E**, **F** and **H** are the leaf nodes.

Link

The pointer to a node in the tree is known as a link. A node can have more than one link. In figure 10.4, node **A** has two links. Node **C** has three links. Nodes **B** and **G** have only one link and nodes **D**, **E**, **F** and **H** have no links.

Path

Every node in the tree is reachable from the root node through a unique sequence of links. This sequence of links is known as a path. The number of links in a path is considered to be the length of the path.

In figure 10.2, the length of the path from **A** to **D** = (the link from **A** to **B**) + (the link from **B** to **D**)

= 1 + 1

= 2

Levels

The level of a node in the tree is considered to be its hierarchical rank in the tree. In figure 10.4, the root node **A**, is at level 0.

Consider a node which is at level L. Then,

The level of its parent node = L-1

The level of its child node = L+1

This rule does not apply to a root node as it does not have a parent node. In figure 10.4, node **A** is at level 0, nodes **B** and **C** are at level 1. Nodes **D**, **E**, **F,** and **G** are at level 2 and node **H** is at level 3.

Number of nodes in the path = (the length of the path from the root to the node) +1

In figure 10.4, the number of nodes in the path from **A** to **D**

= (the length of the path from the root to the node) + 1

= {(the link from **A** to **B**) + (the link from **B** to **D**)} + 1

= {(1) + (1)} + 1

= {2} + 1

= 3

Height

The height of a non-empty tree is the maximum level of a node in the tree. The height of an empty tree (no node in a tree) is 0. The height of a tree containing a single node is 1. The longest path in the tree has to be considered to measure the height of the tree.

Height of a tree (h) = I_{max} + 1, where I_{max} is the maximum level of a tree.

In figure 10.4, the maximum level of the tree is 3.

So, height of the tree (h) = I_{max} + 1

= 3 + 1

= 4

Degree (Arity)

The degree or arity of a node is the maximum number of children that a node has. In figure 10.4, the degree of node **A** is 2; the degree of node **B** is 1; and the degree of node **C** is 3.

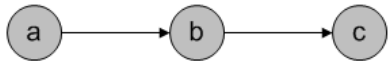
Siblings

The nodes which have the same parent node are known as siblings. In figure 10.4, nodes **B** and **C** are siblings as they have have the same parent node, A.

Graphs consist of a set of nodes and edges, just like trees. But for graphs, there are no rules for the connections between nodes. In graphs, there is no concept of a root node, nor a concept of parents and children. Rather, a graph is just a collection of interconnected nodes. All trees are graphs. A tree is a special case of a graph, in which the nodes are all reachable from some starting node.

Figure 10.5 shows a graph with arcs.

Figure 10.5: A Graph



An arc is an ordered pair of vertices. In figure 8.5, the arcs are (**a**,**b**) and (**b**,**c**). In the arc (**a**, **b**), **a** is called the tail of the arc and **b** is called the head of the arc. Similarly, in the arc (**b**,**c**), **b** is called the tail of the arc and **c** is called the head of the arc. If (a,b) is a directional arc, then (a,b) is not equal to (b,a).

In figure 10.5, Vertices (V) = {a, b, c} and Arcs (E) = {(a,b), (b,c)}

The vertices of the graph are used to represent objects and the arcs are used to represent the relationship between the objects.

10.1.2 Types of Graphs

A graph consists of a set of vertices and edges/arcs.

Graph (G) = (V, E); where V represents vertices and E represents edges/arcs.

Vertices are also known as nodes.

The three types of graphs are:

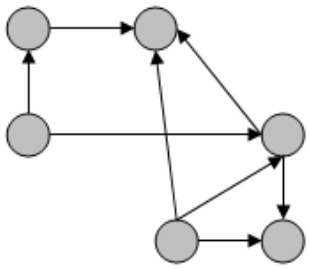
1. Directed graphs
2. Undirected graphs
3. Mixed graphs

Directed Graphs

A graph in which each edge is assigned a direction is called a directed graph.

The figure 10.6 shows a directed graph. Directed arcs connect the nodes.

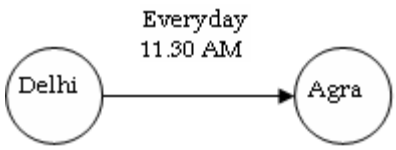
Figure 10.6: A Directed Graph



A directed graph (G) consists of vertices (V) and a set of arcs (E). Directed graphs are also known as digraphs. In a tree, the nodes are considered as vertices and the directed lines are considered as arcs.

Figure 10.7 shows example of directed graph.

Figure 10.7: Example for Directed Graph



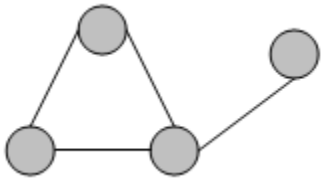
In figure 10.7, the nodes of the tree represent cities. The arc represents the day and time of the flight from Delhi to Agra. The time of the flight from Delhi to Agra is not equal to the time of the flight from Agra to Delhi. Here, (Delhi, Agra) is not equal to (Agra, Delhi).

Undirected Graphs

An undirected graph is a graph in which each edge has no direction.

The figure 10.8 shows an undirected graph.

Figure 10.8: An Undirected Graph



An undirected graph (G) consists of a set of vertices (V) and a set of edges (E). Each edge in undirected graphs has a disorganized pair of vertices.

An edge is an unordered pair of vertices. If (**a**,**b**) is an undirected edge, then (**a**,**b**) = (**b**,**a**).

Figure 10.9: Example for Undirected Graph

In figure 10.9, the nodes of the tree represent cities. The edge represents the distance between the cities. The distance from Delhi to Agra is equal to the distance from Agra to Delhi. Here, (Delhi, Agra) is equal to (Agra, Delhi).

Mixed Graph

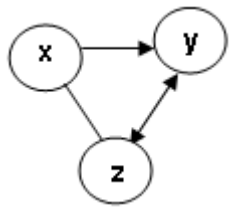
A mixed graph G contains both directed and undirected edges. It is represented as:

G = (V, E, A)

Where, **V** refers to a set of vertices, **E** refers to a set of edges and **A** refers to a set of arcs.

The figure 10.10 shows a mixed graph.

Figure 10.10: Mixed Graph



Task

Construct a graph G for given vertices V = {2, 3, 4, 5} and edges E = {(2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 3), (4, 5)}.

10.2 Types of Trees

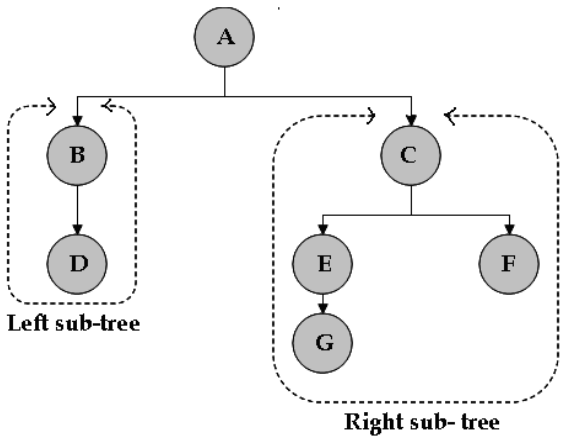
There are different kinds of trees and it is important to understand the difference between them. We will discuss the following types of trees in this section:

1. Binary tree
2. Binary search tree
3. 2-3 tree
4. Huffman tree

10.2.1 Binary Tree

A tree data structure in which every node has a maximum of two child nodes is known as a binary tree. It is the most commonly used non-linear data structure. A binary tree could either have only a root node or two disjoint binary trees called the left sub-tree or the right sub-tree. An empty tree could also be a binary tree.

Figure 10.11: A Binary Tree

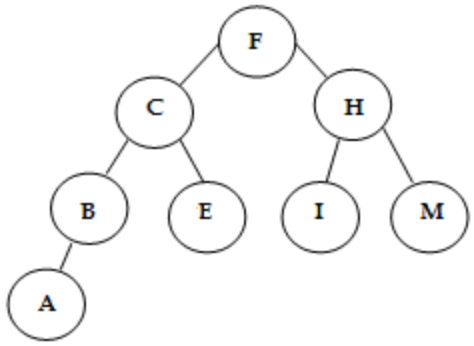


In the figure 10.11, the node **A** is the root node. The nodes **B** and **D** belong to the left sub-tree and nodes **C, E, F** and **G** belong to the right sub-tree.

10.2.2 Binary Search Tree

A binary search tree is a tree in which for a given node **n**, each node to the left is smaller than **n** and each node to the right is larger than **n**. This applies recursively down the left and right sub-trees. Figure 10.12 shows a binary search tree.

Figure 10.12: Binary Search Tree



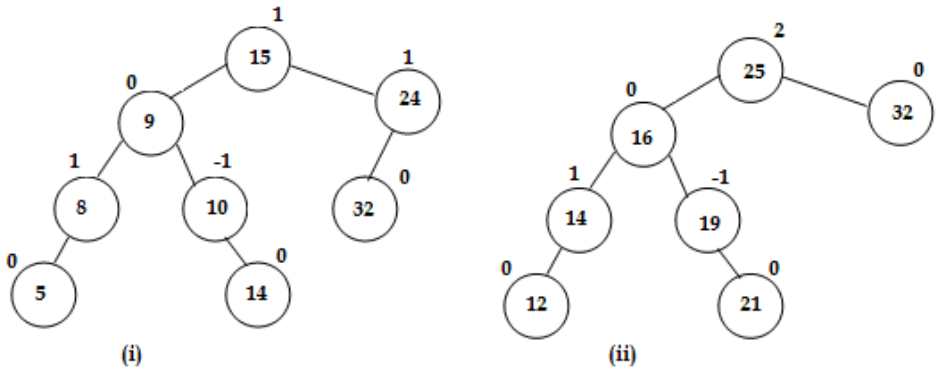
In figure 10.12, the nodes **C, B, E** and **A** belonging to left sub-tree are smaller than **F** and the nodes **H, I,** and **M** belonging to right sub-tree are larger than **F**.

AVL Tree

An AVL tree is a binary search tree having a balance factor of every node as 0 or +1 or −1. The balance factor of a node is defined as the difference between the heights of the node's left and right sub-trees. The height of an empty tree is −1.

The binary search tree in figure 10.13(i) is an AVL tree but the binary search tree in 10.13 (ii) is not an AVL tree. The numbers above the nodes indicate the balance factors of the nodes.

Figure 10.13: (i) AVL Tree
(ii) Binary Search Tree that is not an AVL Tree

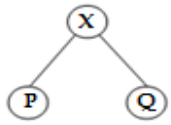


10.2.3 2-3 Trees

A 2-3 tree is another type of tree which has 2 types of nodes, 2-node and 3-node.

The figure 10.14 represents a 2-node tree.

Figure 10.14: A 2-Node Tree

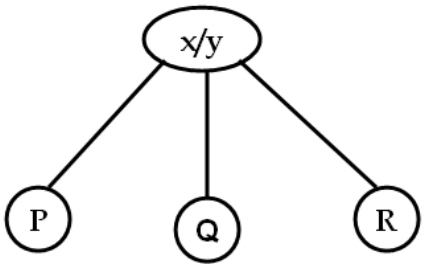


The 2-node structure shown in figure 8.14 has one data element and two children. Every 2-node must have the following properties:

1. Every value appearing in the child **P** must be ≤X.
2. Every value appearing in the child **Q** must be ≥X.
3. The length of the path from the root of a 2-node to every leaf in its child must be the same.

The figure 10.15 represents a 3-node tree.

Figure 10.15: A 3-Node Tree



The 3-node structure shown in figure 10.15 has two data elements and three children. Every 3-node must have the following properties:

1. Every value appearing in child **P** must be ≤ X.
2. Every value appearing in child **Q** must be in between X and Y.
3. Every value appearing in child **R** must be ≥ Y.
4. The length of the path from the root of a 3-node to every leaf in its child must be the same.

10.2.4 Huffman Trees

Huffman codes are digital data compression codes which were devised by Prof. David A. Huffman (1925-1999). Huffman codes provide good compression ratios. Even today, after 50 years, Huffman codes are very useful. Huffman compression is a compression technique in which there is no loss of information when the data is compressed i.e., after we decompress the data, the original information can be retrieved. Hence, it is named as 'lossless compression'. Lossless compression is desired in compressing text documents, bank records, and so on.

There are two data encoding schemes and Huffman Encoding scheme falls under one of the two schemes. The following are the two data encoding schemes:

1. ***Fixed Length Encoding***: In fixed length encoding, all symbols are encoded using the same number of bits. An example of fixed length encoding is ASCII code which uses 7 bits to encode a total of 128 different symbols. The difficulty with fixed length codes is that the probability of occurrence of the symbols to be encoded is not considered. A symbol that occurs 1000 times is encoded with the same number of bits as a symbol which occurs only 10 times. This disadvantage makes fixed length encoding inefficient for data compression.
2. ***Variable Length Encoding***: Variable length encoding assigns less number of bits to symbols which occur more often and more number of bits to symbols whose frequency is less. The Huffman Encoding scheme falls in the category of variable length encoding i.e., code for the symbol depends on the frequency of occurrence of that symbol.

Algorithm for Constructing Huffman tree

The following sequence of steps needs to be followed to construct a Huffman tree:

1. Input all symbols along with their respective frequencies.
2. Create leaf nodes representing the scanned symbols.
3. Let **S** be a set containing all the nodes created in step 2

When there is only one node in **S,** the following steps need to be followed:

1. Sort the nodes (symbols) in **S** with respect to their frequencies.
2. Create a new node to combine the two nodes with least frequencies.
3. Frequency of this new combined node will be equal to the sum of frequencies of nodes which were combined. This newly created combined node will be the parent of two nodes which were combined.
4. Replace the two nodes which were combined with the new combined node in **S**.
5. After the 4th step, you will be left with only one node, which is the root of the Huffman tree, having frequency equal to the sum of all frequencies of all symbols. Thus, a tree is generated with leaf nodes containing the basic symbols whose code is to be found.

With the help of an example we will learn how to construct a Huffman tree. The table 10.1 shows the frequency of occurrence of different symbols.

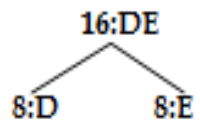
Table 10.1: Symbol Frequency Table

Symbol	Frequency of Occurrence
A	24
B	12
C	10
D	8
E	8

Using the symbols and frequencies from the table 10.1, we can create the leaf nodes and then sort them. Symbols **D** and **E** have the least frequency, 8; these two nodes are combined to make a node **DE** having

frequency 8+8=16. This new node **DE** is the parent node of the nodes **D** and **E**, and **DE** replaces **D** and **E** as shown in figure 10.16.

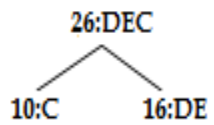
Figure 10.16: Step 1 for Constructing Huffman Tree



Again we sort the nodes based on their frequency of occurrence. Now **DE** and **C** have the least frequencies i.e., 16 and 10 each. This time we combine **DE** and **C** to create a new node **DEC** having frequency 26.

Nodes **DE** and **C** are replaced by their parent **DEC** as depicted in figure 10.17.

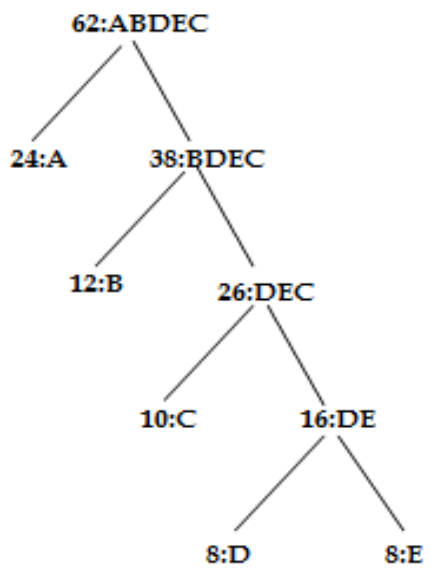
Figure 10.17: Step 2 for Constructing Huffman Tree



Similarly, combine **B** with frequency 12 and **DEC** with frequency 26 to create **BDEC. BDEC** becomes the parent of **B** and **DEC** with frequency 38. At last only two nodes are left namely, **BDEC** and **A.** We again sort them and combine both to form **ABDEC** which has a frequency count of 62.

Figure 10.18 depicts the Huffman tree.

Figure 10.18: The Huffman Tree



After making **ABDEC** parent of **A** and **BDEC** and replacing them with **ABDEC**, we have created the Huffman tree for the symbols in Table 10.1. Node **ABDEC** is the root of the tree. The figure 10.18 shows the Huffman tree thus constructed.

10.3 Representation of Tree in Memory

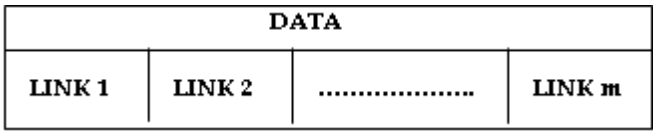
Since a node in a general tree has a number of children, the implementation of this tree becomes complex than that of a binary tree. There are two ways to represent trees. They are:

1. Linked representation
2. Array representation

Linked Representation of Trees

In linked representation of trees, if the maximum number of children is restricted to **m**, then the structure of the node can be represented as shown in figure 10.19.

Figure 10.19: Structure of Node



With **LINK** and **DATA** fields, you can represent a tree using the fixed node size linked structure. However, the null links in a node result in wastage of memory space.

Did you know? If T is an m-ary tree with n nodes, then n(m-1) +1 of the nm link fields are null.

Array Representation of Trees

A tree can also be represented using arrays. In this case, you need to have three arrays namely, DATA, LEFTCHILD and SIBLING. The information content of the node is stored in DATA array, the left-most children of the node are stored in LEFTCHILD array, and the immediate sibling of the node is stored in SIBLING array. The array representation of the tree shown in figure 10.20 is shown in the table 10.2.

Figure 10.20: A Tree

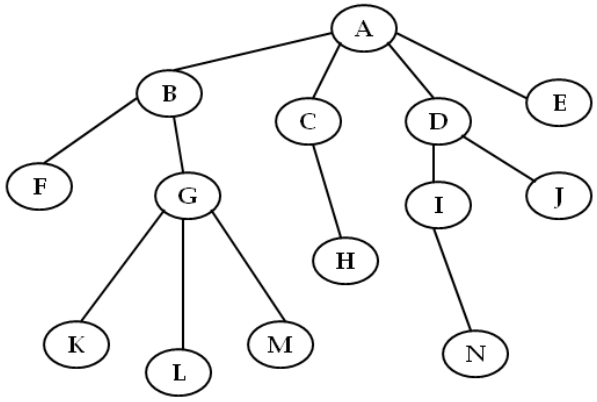


Table 10.2 shows array representation of the tree.

Table 10.2: Array Representation of the Tree

	Data	Left child	Sibling
1	A	2	0
2	B	6	3
3	C	8	4
4	D	9	5
5	E	0	0
6	F	0	7
7	G	11	0
8	H	0	0
9	I	14	10
10	J	0	0
11	K	0	12
12	L	0	13
13	M	0	0
14	N	0	0

In the above representation, **0** indicates the null link. The major advantage of array representation of trees is that it helps in getting direct access to any node. However, the drawback is that it is necessary to fix the maximum depth of the tree since the array size has already been decided. If the array size is quite larger than the depth of the tree, then it is considered to be memory wastage. If the array size is lesser than the depth of the tree, then it is not possible to represent some parts of the tree.

10.4 Application of Trees

Trees are used in operating systems, compiler design, and searching. Expression tree is an example of general structure known as parse tree, which is a central data structure in compiler design. Parse trees are not binary but are comparatively simple extensions of expression trees. Another remarkable application of trees is in designing of computer games such as Nim, Tic-tac-toe, Chess, Kalah, Checkers, and so on.

A decision tree is a binary tree where a node represents a decision and edges represent the outcome of the decision. A decision tree is thus a powerful method for classification and prediction, and facilitates decision making in sequential decision problems.

Let us discuss in detail the various applications of trees.

10.4.1 Expression Trees

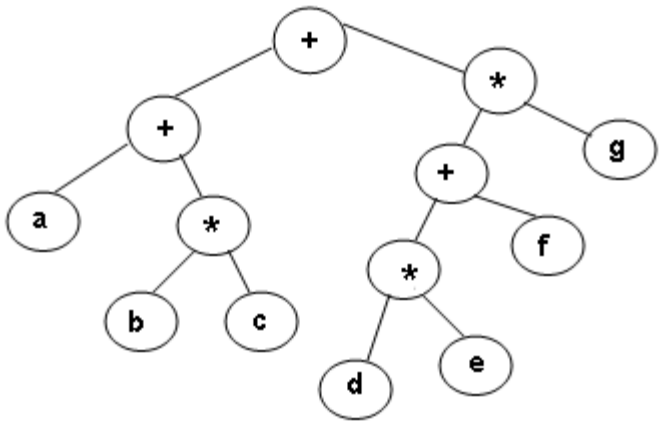
Expression trees are special kind of binary trees. An expression tree provides a method to translate executable code into data. This translation of data is very useful to modify or transform the code before executing it. The leaves of an expression tree are operands such as constants or variable names, and the other nodes contain operators. This specific tree is binary because all the operations are binary, and it is not possible for nodes to have more than two children.

To evaluate an expression tree, we recursively evaluate the left and right sub-trees and then apply the operator at the root. We can produce an infix expression by recursively producing a parenthesized left expression, and then printing out the operator at the root, and later recursively producing a parenthesized right expression. This approach (left, node, right) is known as an inorder traversal.

An alternate traversal strategy is to recursively print out the left sub-tree, the right sub-tree, and then the operator. This traversal approach (left, right, node) is known as a postorder traversal.

Figure 10.21 shows an example of an expression tree. In this example, the left sub-tree evaluates to a + (b * c) and the right sub-tree evaluates to ((d *e) + f)*g. The entire tree therefore represents (a + (b*c)) + (((d * e) + f)* g).

Figure 10.21: Expression Tree for (a + b * c) + ((d * e + f) * g)



An alternate traversal strategy is to recursively print the left sub-tree, the right sub-tree, and then the operator. If we apply this strategy to the tree above, the output is a b c * + d e * f + g * +. This traversal strategy is generally known as a post order traversal.

A third traversal strategy is to print the operator first and then recursively print the left and right sub-trees. The resulting expression, + + a * b c * + * d e f g, is the less useful prefix notation and the traversal strategy is called a preorder traversal.

Task Construct an expression tree for the expression (2 * (5 + (6 + 4))).

10.4.2 Game Trees

A game tree is a graphical representation of a sequential game. It is a directed graph whose nodes are positions in a game and whose edges are moves. The complete game tree for a game starts at the initial position and contain all possible moves from each position.

The game Nim is played by two players **A** and **B**. A board which initially contains a pile of **n** toothpicks describes the game. The players **A** and **B** make moves alternately with **A** making the first move. A legal move consists of removing either 1, 2, or 3 of the toothpicks from the pile. A player cannot remove more toothpicks than that present in the pile. The player who picks the last toothpick loses the game and the other player wins. The board configuration at any time is absolutely specified by the number of toothpicks remaining in the pile. At any time of the game, the status is determined by the board configuration together with the player whose turn it is to make the next move. A terminal board configuration is one which denotes both a win, lose, or draw situation and the further configurations are non-terminal. The only terminal configuration in Nim is the one in which there are no toothpicks in the pile. This configuration is a win for player **A** if **B** makes the last move, or else it is a win for **B**. The game of Nim cannot end in a draw.
A sequence C_1, ..., C_m of board configurations is assumed to be valid if:

1. C_1 is the starting configuration of the game
2. C_i, $0 < i < m$, are non-terminal configurations
3. C_{i+1} is obtained from C_i by a legal move made by player **A** if i is odd and by player **B** if i is even.

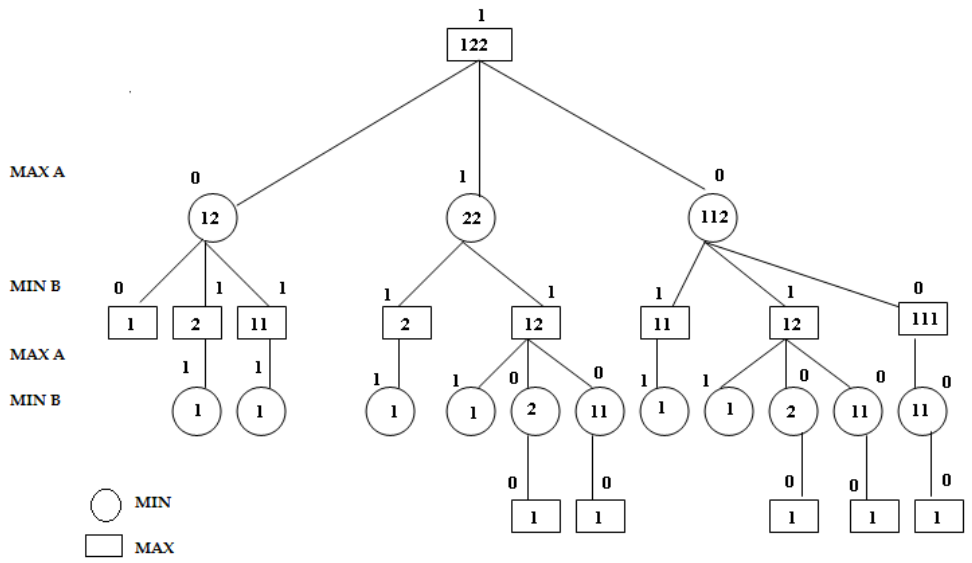
A valid sequence C_1, ...,C_m of board configurations with C_m a terminal configuration is an example of the game. The length of the order C_1,C_2, ...,C_m is m. A finite game is the game in which there are no valid sequences of infinite length. All possible instances of a finite game may be represented by a game tree. In a game tree for nim, each node of the tree represents a board configuration. The root node denotes the starting configuration C_1. The transitions from one level to the next level are made because of a move of A or B. Transitions from an odd level represents moves made by A. All other transitions are the result of moves made by B. Square nodes are used to represent the board configurations when it is A's turn to move. Circular nodes are used for other configurations. The edges from various levels are labeled with the move made by A and B respectively (for example, an edge labeled 1 means 1 toothpick is to be removed).

The terminal configurations are represented by leaf nodes. Leaf nodes are labeled by the name of the player who wins when that configuration is reached. In the game of Nim, player A can win only at leaf nodes on odd levels while B can win only at leaf nodes on even levels. The degree of any node in a game tree is equal to the number of distinct legal moves. In Nim, there are maximum 3 legal moves from any configuration. The number of legal moves from any configuration is finite.

Example: In Nim, we represent the configuration of the piles by a monotone sequence of integers, such as (1,3,5,7) or (2,2,3,9,110). A player may remove, in one turn, any number of toothpicks from one pile of his/her choice. Thus, (1,3,5,7) would become (1,1,3,5) if the player were to remove 6 toothpicks from the last pile. The player who takes the last toothpicks loses. The Nim game (1, 2, 2) is shown in figure 10.22.

Figure 10.22 depicts game tree for (1,2,2) NIM.

Figure 10.22: Game Tree for (1, 2, 2) NIM



In figure 10.22, the number in the root shows that at first, there are five toothpicks which consist of three sets, 1, 2, and 2. Assume **A** is the player who makes the first move. **A** may take one or two toothpicks. After **A**'s move, it is **B**'s turn and the numbers in the nodes represent the toothpicks left. Then B moves one or two toothpicks and the status is shown in the next nodes and so on until there is one toothpick left.

We can use this game tree to analyze the best possible move. For each player, the best move is to make the opponent lose in order to win. So, one must make the move to get the MAX score and force their opponent to get the MIN score. A loss is presented by "0" and a win is presented by "1". The MAX nodes denote the position of the current player and the MIN nodes denote the position of the opponent. Since the goal of this game is that the player who removes the last toothpick loses, the scores are assigned to "0" if the leaves are at MAX nodes and the scores are assigned to "1" if the leaves are MIN. Then, the scores are totaled from the bottom nodes and assigned to the internal nodes. At MAX nodes, choose the MAXIMUM score of the children and at MIN nodes, choose the MINIMUM score of the children. In this manner, we may compute the scores of the non-leaf nodes from the bottom up. In the example shown in figure 10.22, the root node is "1" and thus, corresponds to a win for the first player. The first player must choose a child position that corresponds to a "1".

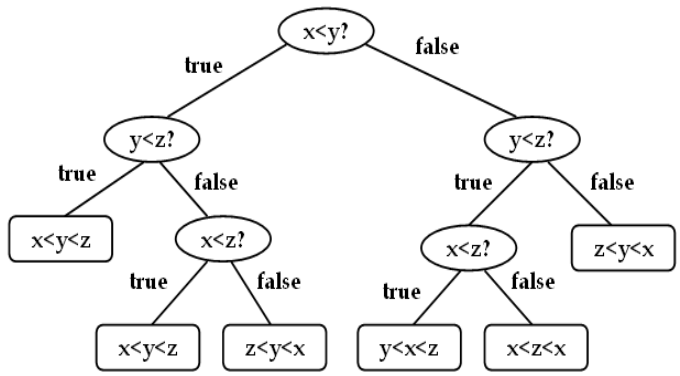
The depth of a game tree is the length of a longest instance of the game. Similarly, it is easy to construct game trees for other finite games such as chess, Tic-tac-toe, Kalah, and so on. Chess is not a finite game because it is possible to repeat board configurations in the game. Chess can be considered as a finite game when this possibility is not allowed. In games with large game trees, the decision as to which move to make next can be made only by looking at the game tree for the next few levels. A game tree is useful in determining the next move a player should make.

10.4.3 Decision Trees

Decision trees represent a program in the form of tree with branches. Here, each node represents a decision. First, the root node is tested and then the control is passed to one of its sub-trees, depending on the result of the test. This flow is continued until the leaf node with the element of interest is reached.

An example for a decision tree is given in figure 10.23. This decision tree decides the ascending order among three numbers **x**, **y**, and **z**.

Figure 10.23 Decision Tree that Determines the Ascending Order for Three Numbers



First, we check if **x** is lesser than **y**. If the condition is true, then check if **y** is lesser than **z**. If yes, then the ascending order of these numbers is **x<y<z**. If not, check if **x** is lesser than **z**. If yes, then the ascending order of these numbers is **x<z<y**. If not, then the ascending order of these numbers is **z<x<y**. If the condition **x<y** is false, i.e., if **x** is not lesser than **y**, then check if **y** is lesser than **z**. If this is true, then check whether **x** is lesser than **z**. If yes, then the ascending order of these numbers is **y<x<z**. If the condition **y<z** is false, i.e., if **y** is not lesser than **z**, then the ascending order of these numbers is **z<y<x.**

Task

Construct a decision tree for finding the greatest number among a given set of numbers.

10.5 Summary

- A tree structure is a way of presenting the hierarchical nature of a structure in a graphical form.
- The trees can be represented as graphs.
- The three types of graphs are directed graphs, undirected graphs, and mixed graphs.
- The different kinds of trees include binary tree, binary search tree, 2-3 tree, and Huffman tree.
- The two ways to represent trees are linked representation and array representation.
- The applications of trees include expression trees, game trees, and decision trees.
- To evaluate an expression tree, we recursively evaluate the left and right sub-trees and then apply the operator at the root.
- A game tree is a graphical representation of a sequential game. It is a directed graph whose nodes are positions in a game and whose edges are moves.
- In decision trees, each node represents a decision.

10.6 Keywords

Compression Ratio: The measurement of compressed data.

Data Encoding: The method by which certain communication devices encode digital data into an analog signal for transmission.

Genealogy: The study of ancestry and family history.

Kalah: Kalah, also called Kalaha or Mancala, is a game in the Mancala family invented by William Julius Champion Jr in 1940.

10.7 Self Assessment

1. State whether the following statements are true or false:
   (a) Decision trees represent a program in the form of tree with branches.
   (b) The null links in a node result in wastage of memory space.
   (c) A game tree is an undirected graph whose nodes are positions in a game and whose edges are moves.
   (d) The major advantage of array representation of trees is that it helps in getting direct access to any node.
   (e) An AVL tree is a binary search tree having a balance factor of every node as 0 or +2 or −2.
   (f) A binary tree could either have only a root node or have two disjoint binary trees called the left sub-tree or the right sub-tree.
2. Fill in the blanks:
   (a) An AVL tree is a …… tree.
   (b) The two nodes of 2-3 tree are …… and ……
   (c) The …… node is tested first, in a decision tree.
   (d) The information content of the node is stored in …… array.
   (e) A ……graph contains both directed and undirected edges.
3. Select a suitable choice for every question:
   (a) A tree data structure in which every node has a maximum of two child nodes is known as a ……tree.
      (i) Binary
      (ii) AVL
      (iii) Game
      (iv) Huffman

(b) An ……tree is a binary search tree having a balance factor of every node as 0 or +1 or −1.

(i) Huffman

(ii) Binary

(iii) Binary search

(iv) AVL

(c) The …… of the graph are used to represent objects.

(i) Arcs

(ii) Edge

(iii) Vertices

(iv) Directions

(d) The nodes which have the same parent node are known as ……

(i) Root

(ii) Siblings

(iii) Left nodes

(iv) Right nodes

(e) The height of an empty tree is ……

(i) 0

(ii) 1

(iii) -1

(iv) Null

10.8 Review Questions

1. "Choosing the best data structure for a program is a challenging task". Discuss.
2. "The trees can be represented as graphs". Justify.
3. "The height of a non-empty tree is the maximum level of a node in the tree". Analyze.
4. "An undirected graph is a graph in which each edge has no direction". Discuss with example.
5. "There are different kinds of trees and it is important to understand the difference between them". Analyze.
6. "An AVL tree is a binary search tree having a balance factor of every node as 0 or +1 or −1". Explain with diagram.
7. "A game tree is a graphical representation of a sequential game". Explain with example.

8. "A graph in which each edge is assigned a direction is called a directed graph". Discuss with example.
9. "A board which initially contains a pile of **n** toothpicks describes the Nim game". Analyze.
10. "A graph consists of a set of vertices and edges/arcs". Explain with diagram.
11. "There are two ways to represent trees". Discuss in detail.
12. "There are sequences of steps to be followed to construct a Huffman tree". Explain with example.
13. "Every 3-node in a 2-3 tree must have specific properties". Discuss with diagram and example.

Answers: Self Assessment

1. (a) True

   (b) True

   (c) False

   (d) True

   (e) False

   (f) True
2. (a) Binary search tree

   (b) 2-node and 3-node

   (c) Root

   (d) Data

   (e) Mixed
3. (a) Binary

   (b) AVL

   (c) Vertices

   (d) Siblings

   (e) -1

10.9 Further Readings

Books

Lipschutz. S. (2011). Data Structures with C. Delhi: Tata McGraw hill

Reddy. P. (1999). Data Structures Using C. Bangalore: Sri Nandi Publications

Samantha. D (2009). Classic Data Structures. New Delhi: PHI Learning Private Limited

Weiss. M. (1996). Data Structures and Algorithm Analysis in C. Addison Wesley Publications

Online link

http://msdn.microsoft.com/en-us/library/ms379574(v=vs.80).aspx

http://cg.scs.carleton.ca/~luc/1997notes/topic11/

Unit 11: Introduction to Binary Trees

CONTENTS

Objectives

Introduction

11.1 Types of Binary Trees

11.2 Storage Representation of Binary Tree

11.3 Overview of Threaded Binary Trees

11.4 Summary

11.5 Keywords

11.6 Self Assessment

11.7 Review Questions

11.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the different types of binary trees
- Explain the storage representation of binary tree
- Explain linked representation of binary tree in detail
- Discuss the sequential representation of binary tree
- Provide an overview of threaded binary trees

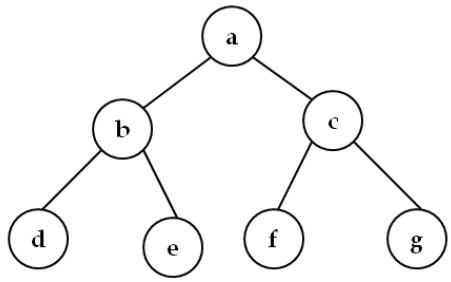
Introduction

A binary tree is an important data structure providing hierarchical representation of data in the memory. Each node in a binary tree can have a maximum of two branches. The binary tree is represented with a root node, which consists of two sub-trees at the left and right side of the root node. The two sub-trees are called as the left sub-tree and right sub-tree respectively. The nodes in the general tree can have any number of children, whereas, the nodes in the binary tree can have a maximum of two children.

We can define a binary tree as a set of nodes which is either empty or contains a root and two disjoined right sub-tree and left sub-tree.

Figure 11.1 represents a binary tree.

Figure 11.1: A Binary Tree

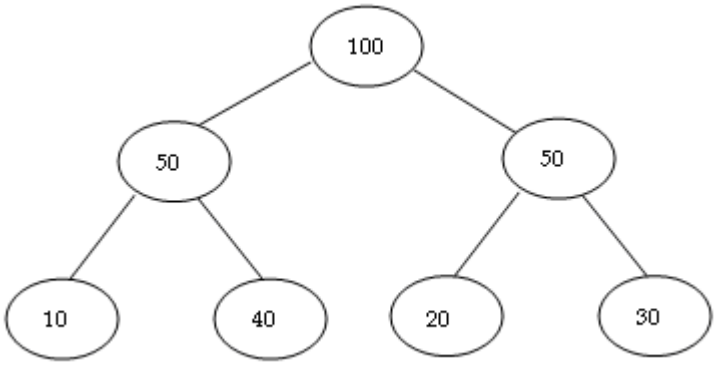


In the figure 11.1, node **a** is the root, node **b** and **c** are its child nodes. The nodes **b** and **c** form two sub-trees. Node **b** forms the left sub-tree of node **a** while node **c** forms the right sub-tree. The nodes without successors are called as terminal nodes or leaf nodes. The nodes **d**, **e**, **f** and **g** are terminal nodes.

Notes The average depth of binary tree is 0(√n).

Example: Consider the binary tree in figure 11.2 with nodes 10, 20, 30, 40, 50, 100, 50.

Figure 11.2: A Binary Tree



The binary tree of figure 11.2 is structured such that the successor node value sums up to the value of root node.

In this example, the root node is 100. The two nodes 50 and 50 form the left and the right sub-trees. Similarly, left tree 50 has two sub-trees 10 and 40, the right tree sub-tree 50 has two sub-trees 20 and 30.The tree ends with terminal nodes 10, 40, 20 and 30.

Since the terminal nodes 10, 40, 20 and 30 do not have any successor nodes, they are termed as leaves.

11.1 Types of Binary Trees

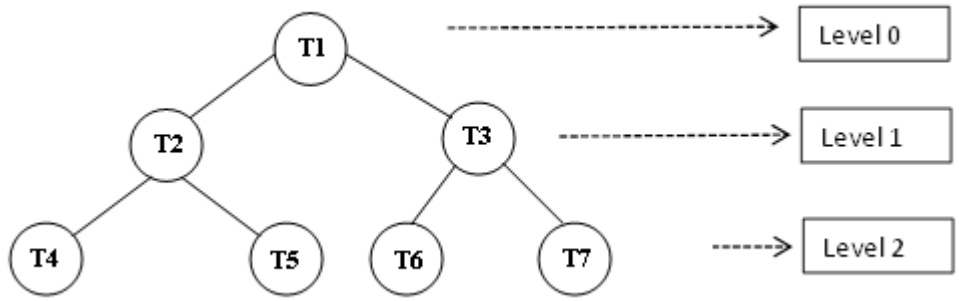
Binary trees can be classified on the basis of the level of nodes and the number of nodes that are present at each level of the tree. The types of binary trees are:

1. Complete binary tree
2. Strictly binary tree
3. Extended binary tree

Complete Binary Tree

A tree is known as a complete binary tree if each of its level, except the last level, is completely filled and all nodes are as far left as possible. Therefore, at any level 'l', the maximum number of nodes must be equal to 2^l. In a complete binary tree, at level 0 there must be only one node known as root node and it should have a maximum of two sub nodes at level 1, and at level 2 there must be a maximum of 4 sub nodes. Figure 11.3 depicts a complete binary tree. The figure depicts the nodes in the binary tree with their associated levels.

Figure 11.3: A Complete Binary Tree

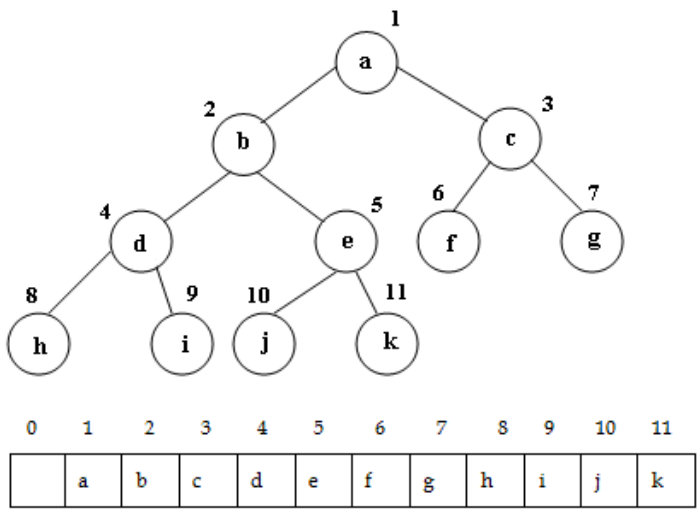


In the figure 11.3, the node **T1** at level 0 represents the root node. The two sub nodes **T2** and **T3** are the child nodes at level 1. The successor nodes **T4**, **T5**, **T6** and **T7** are the terminal nodes. The number of nodes at level 1 is equal to 2. Similarly, the number of nodes at level 2 is equal to 4.

The main advantage of a complete binary tree is that the position of the parent node and child node can be mapped easily in an array. The mapping for a binary tree is defined by assigning a number to every node in the tree. The root node is assigned the number 1.For the other nodes, if **i** is its number, then the left child node is assigned the position 2i and the right child node is assigned the position 2i+1. The mapping of binary tree provides a simple form of array storage representation. Hence the nodes in an array can be stored as a[i], where a[ ] is an array.

Figure 11.4 depicts the representation of complete binary tree in an array. An array representation of binary tree allocates nodes of the tree in a memory. Each node is indexed such that the nodes are associated with the index number of the array for allocation.

Figure 11.4: Representation of Complete Binary Tree into an Array



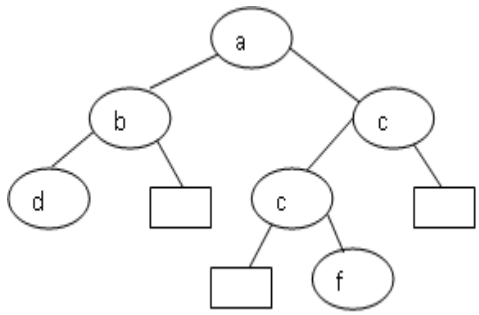
In the figure 11.4, consider the node e stored at index i=5. The parent node of node e is node b which is stored at the index i/2; 5/2=2. The left child node of node e is node j and it is stored at the index 2i; 2*5= 10 and right child node k is stored at the index 2i+1 i.e., 2*5+1=11.

Strictly Binary Tree

A binary tree is called a strictly binary tree when the non-terminal nodes have exactly two child nodes forming left and the right sub-trees.

Figure 11.5 depicts a strictly binary tree. A strictly binary tree with five nodes is provided such that the non-terminal nodes form the left and right sub-trees.

Figure 11.5: Strictly Binary Tree



In the figure 11.5, nodes **a** and **b** have two child nodes. The parent node **a** has two child nodes **b** and **c** forming the left sub-tree and right sub-tree respectively. Similarly, node **b** is the parent node for nodes

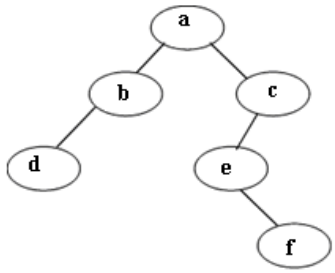
d and **e**. A binary tree in which its non-leaf nodes possess exactly two child nodes represents a strictly binary tree.

Extended Binary Tree

A binary tree is called an extended binary tree when every node in the tree either has 0 or 2 sub nodes (child nodes). The nodes with two child nodes are called internal nodes and nodes without child nodes are called external nodes. The extended binary tree is also called a 2-tree. The nodes in the binary tree possessing only one child node can be extended with one more child node. The extended binary tree plays a very important role in implementing algebraic expressions. The algebraic expressions are represented by operands and operators. Hence, the left and right child nodes represent the operands and parent node represents the operator.

Example: Consider figure 11.6 that depicts a binary tree with a single child node. The binary tree with single child node is initial phase of designing the binary tree which can be made a complete binary tree by extending the nodes.

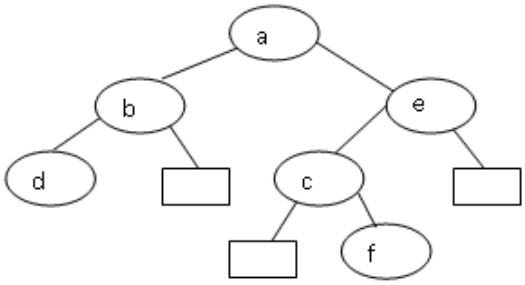
Figure 11.6: Binary Tree with Single Child Node



In the figure 11.6, the parent nodes **b** and **c** have only one child each, node **d** and **e** respectively. The node **e** also has single child node **f**. By adding another child node to the parent nodes b, c and e, we can obtain an extended binary tree.

Figure 11.7 depicts an extended binary tree.

Figure 11.7: Extended Binary Tree



In the figure 11.7, the extended binary nodes are represented by a rectangular box. The parent nodes b, c and e are provided with two child nodes by extending the nodes.

11.2 Storage Representation of Binary Tree

The binary tree can be represented by dynamic allocation technique and sequential allocation technique. In the dynamic allocation technique, memory is allocated to a node using a linked list and in the sequential allocation technique, memory is allocated to a node using an array.

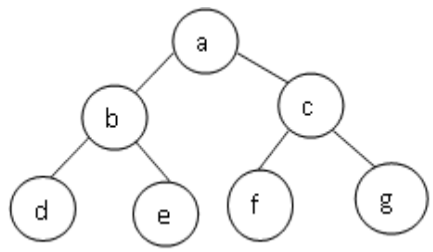
Linked Representation

The binary tree is structured mainly on three nodes -the root (parent) node, the left and the right child nodes. In a linked representation, the nodes of the tree are indicated by three fields. They are:

1. ***info*** – It contains the actual information.
2. ***llink*** – It represents the address of the left sub-tree.
3. ***rlink*** – It represents the address of right sub-tree.

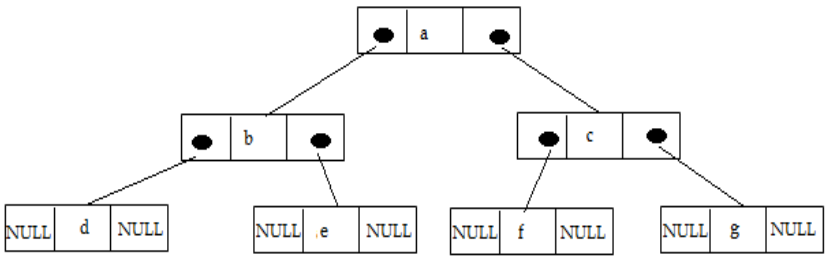
Example: Consider the tree representation shown in figure 11.8. Here each internal node has two child nodes.

Figure 11.8: A Binary Tree



The tree representation of figure 11.8 is structured in a linked list for allocating the memory to a node as shown in figure 11.9.

Figure 11.9: Linked Representation of Binary Tree



In the figure 11.9, the binary tree is structured in the form of linked list for allocating memory to the nodes. Each node consists of three fields; the left link representing the address of the left node, right link representing the address of the right node, and the data present in the node. The llink (left link) of node **a** contains the address of node **b** which is the address of its left sub-tree. Similarly, the rlink (right link) of node **a** contains the address of node **c** which is the address of its right sub-tree.

The linked form of representation is an easier way of identifying, storing and retrieving information in the memory. It represents the logical structure of the data involved.

> *Notes* If there are n nodes in the binary tree, then the number of null links in the linked representation is n+1.

Let us now discuss the algorithm for implementing the linked representation of a binary tree.

Algorithm Binary_Tree (node1, info)

Input – info is the content of the node with pointer node1

Output – A binary tree with two sub-trees of the node1

Data Structure - Linked list structure of binary tree

Steps

1. If (node1 ≠ Null) then// If tree is not empty
    (a) node1.data = info // Store data of node at node1
    (b) node1 has left sub-tree (Give option = Y/N)?
    (c) If (option =Y) then
        (i) llink = GETNODE(node)// Allocate memory to the left child
        (ii) node1.LC = llink // Assign it to left link
        (iii) Binary_Tree (llink, NEWL)// Build left sub-tree with next information as NEWL
    (d) Else
        (i) llink = Null
        (ii) node1.LC = Null // Assign for empty left sub-tree
        (iii) Binary_Tree (llink, Null)// Empty sub-tree
    (e) EndIf
    (f) Node1 has right sub-tree (Give option = Y/N)?
    (g) If (option = Y) then
        (i) rlink = GETNODE(node1) // Allocate memory to right child
        (ii) node1.RC = rlink // Assign it to right link
        (iii) Binary_Tree (rlink, NEWR) // build right sub-tree with next item as NEWR
    (h) Else
        (i) rlink = Null
        (ii) node1.RC = Null // Assign for an empty right sub-tree
        (iii) Binary_Tree (rlink, Null)
    (i) EndIf
2. EndIf
3. Stop

Notes

Advantages of Linked List Representation

1. The insertion and deletion operations can be done easily without moving the other nodes.

2. The memory representation of linked list provides dynamic memory representation. Large number of nodes can be created.

Disadvantages of Linked List Representation

1. Linked list representation does not allow direct access to the nodes in the binary tree and requires special algorithms for accessing the nodes.

Sequential Representation

The sequential or linear representation of the tree is a static representation, in which a block of memory for an array is allocated before storing the actual tree in the memory. The size of the tree is restricted according to the memory allocation.

In the linear representation, nodes are stored sequentially, one level after another. The level of the nodes starts with zero level containing the root node. In the memory allocation, root node is stored as the first element in the array.

The following principle is practiced for allocating node of a tree in the array.

(Assume that indexing of array starts from 1)

1. The root node is present at location 1.

2. For any node with index i, $1 < i \leq n$ (for any value of n)

(a) PARENT (i) = [i/2]

For the node when i = 1, there is no parent

(b) LCHILD (i) = 2 * i

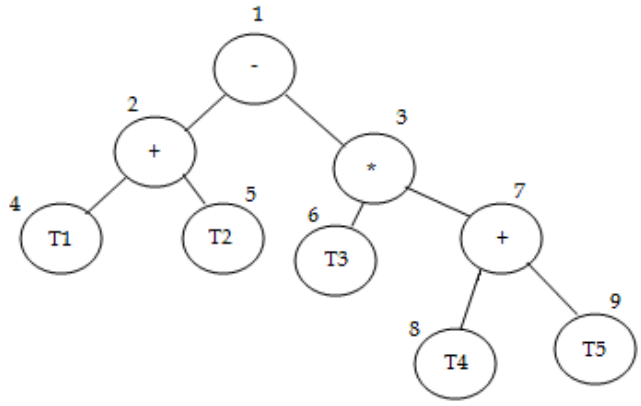
If 2 * i > n, then i has no left child

(c) RCHILD (i) = 2 * i + 1, then i has no right child

Example: In the figure 11.10, the arithmetic expression is logically structured in the form of a tree. Consider the following arithmetic expression represented in the tree structure. Memory must be allocated for each node present in the tree. The algebraic expression (T1 + T2) – T3 * (T4+T5) is represented in the tree as shown in figure 11.10. While evaluating the expression, the left sub-tree is computed first, then the right sub-tree and then finally the root.

Figure 11.10 shows an expression of binary tree.

Figure 11.10: An Expression Binary Tree



The figure 11.11 depicts the allocation of memory for the binary tree shown in figure 11.10.

In the figure 11.11, + node is stored at index i=2. The parent node of node + is node -, and is stored at the index i/2 i.e.,2/2=1. The left child node of node +is node T1 and is stored at the index 2i i.e., 2*2= 4 and right child node T2 is stored at the index 2i+1 i.e., 2*2+1=5. This form of representation indicates the array form of memory allocation of nodes of a binary tree.

Figure 11.11: Sequential Representation of Binary Tree

1	2	3	4	5	6	7	8	9
-	+	*	T1	T2	T3	+	T4	T5

Let us now discuss the algorithm for implementing the sequential representation of binary trees.

Algorithm Binary_Tree (node2, info)

Input – info refers to the data of node2

Output – A binary tree containing two sub-trees of node2

Data Structure – Array representation of tree

Steps

1. If (node2 ≠ 0) then // If tree is not empty

   (a) A[node2] = info // Store data of node2 into array A

   (b) node2 has left sub-tree (Give option = Y/N)?

   (c) If (option = Y) then // If node2 has left sub-tree

      (i) Binary_Tree (2*node2, NEWL) // Then it is 2*node2 with next item as NEWL

   (d) 4. Else

(i) Binary_Tree (0, Null) // Empty sub-tree

(e) EndIf

(f) node2 has right sub-tree (Give option = Y/N)?

(g) If (option = Y) // If node2 has right sub-tree

(i) Binary_Tree (2*node2+1, NEWR) // Then it is at 2*node2+1 with next item as NEWR

(h) Else

(i) Binary_tree (0, Null) //Empty sub-tree

(i) EndIf

2. EndIf

3. Stop

Notes

Advantages of Sequential Representation of Binary Trees

1. The nodes in the binary tree can be accessed by calculating the index of every node. It is efficient and quick.

2. The data is stored without using pointers and the information can be traced easily with the index associated with each node.

Disadvantages of Sequential Representation of Binary trees

1. It is effective only for a full binary tree. Most of the other type of trees may be empty in an array.

Did you know?

Considering the memory requirement for a tree, linked representation uses more memory than sequential representation for allocating memory to binary tree. Linked representation uses extra memory to maintain pointers.

Task

Create a binary tree for the algebraic expression ((T1* T2 + T3) + (T4 – T5) * T6) and represent the binary tree in array representation.

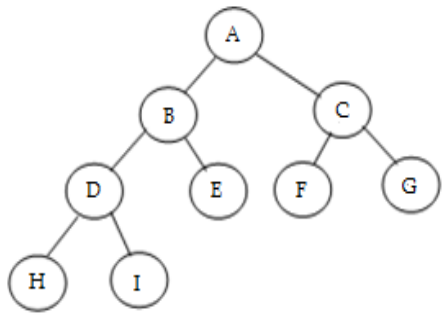
11.3 Overview of Threaded Binary Trees

In a binary tree with **n** nodes, there exists **n+1** NULL links and **2n** number of total links. However, a large value of **n, n+1** NULL and **2n** number of links results in more space wastage. Hence, the solution is to change the node structure for leaf nodes so that the nodes only consist of data field. But, this solution provides complex algorithms. Hence, the only solution is to consider the fixed node structure and use the NULL links to simplify some of the operations. This solution provides the concept of threaded binary trees.

In the threaded binary tree, the NULL links are replaced by pointers known as threads. These threads point to other nodes of a tree. When the tree is traversed in inorder, and if the left child of a node **p** in a binary tree is NULL, then it will be replaced with a thread. The thread will point to the node which appears just before the node **p**. Similarly, if the right child of node **p** is NULL, then it will be replaced with a thread. The thread will then point to a node that appears just after the node **p** after the inorder traversal of a tree. Such threads are known as inorder threads.

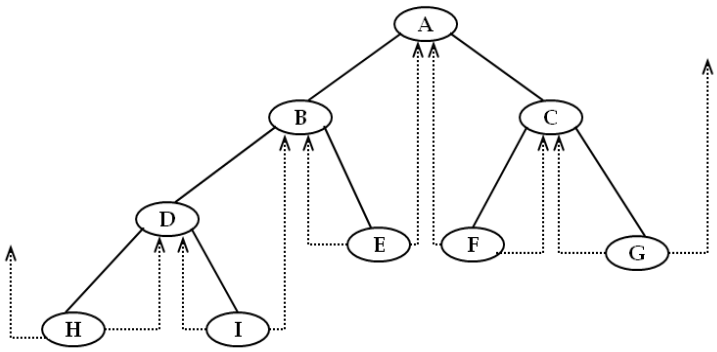
Example: Consider the binary tree in figure 11.16. The figure is a simple binary tree with four level of nodes.

Figure 11.12: Binary Tree



In the figure 11.12, the inorder traversal of the binary tree is "**H D I B E A F C G**". The equivalent threaded binary tree is shown in figure 11.13.

Figure 11.13: Threaded Binary Tree



Leaf nodes are considered for a threaded binary tree. In figure 11.17, if we consider leaf node I then the inorder predecessor of I is D and the left thread will point at node D.

Similarly, the inorder successor of I is B and the right thread will point at node B. All the nodes in the binary tree will be traversed in the similar format.

But, the left thread of node H does not have an inorder predecessor and right thread of node G does not have inorder successor. In such situations, the threads pointing to particular node are not obtained. Hence, to solve such problems, the threaded binary tree uses a node called head node. The head node will have an identical structure similar to the normal tree nodes. In such cases, if the tree is non-empty, then its left child will point at the root of the tree. Similarly, the left thread of node H and right thread of node G will point to its head node.

Lab Exercise

Write a C program to represent the linked list representation of binary tree into sequential representation.

11.4 Summary

- A binary tree is a finite set of data elements and each node contains a maximum of two branches.
- A tree is known as a complete binary tree if each of its level, except the last level, is completely filled with child nodes.
- A binary tree is called a strictly binary tree when the non-terminal nodes have exactly two child nodes forming left and the right sub-trees.
- In an extended binary tree, the nodes possessing only one child node can be extended with one more child node.
- The binary tree can be represented as array as well as linked list.
- In a threaded binary tree, the pointers are represented as threads such that the threads point to the node in the binary tree for any operations.

11.5 Keywords

Leaf Nodes: The nodes without any successors.

Node Predecessor: Node representing the parent node.

Node Successor: Node representing the child node.

Threads: The pointer linking the other nodes in the tree.

11.6 Self Assessment

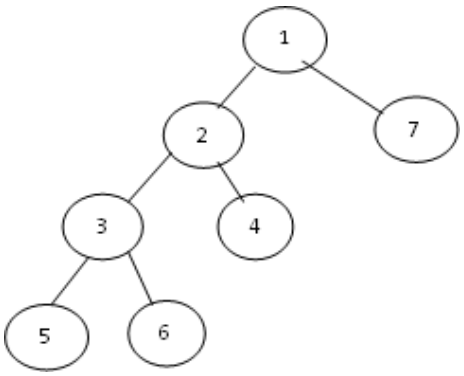
1. State whether the following statements are true or false:
   (a) The strictly binary tree can have any number of children.
   (b) In a binary tree, at level 1, there must be only one node known as root node.
   (c) In an array representation, the size of tree is restricted according to the memory allocation.
   (d) In sequential representation, the level of the nodes starts with level one that contains the root node.
2. Fill in the blanks:
   (a) The …… binary tree helps in mapping the nodes easily.
   (b) The data in …… representation is stored without any help of pointers.
   (c) In the threaded binary tree, the NULL links are replaced by pointers known as ……
3. Select a suitable choice for every question:
   (a) In which representation the memory allocation is restricted to the size of the tree?
      (i) Sequential (ii) Linked list
   (b) Which among the following technique is used to allocate memory to a node using linked list?
      (i) Dynamic allocation (ii) Sequential allocation

11.7 Review Questions

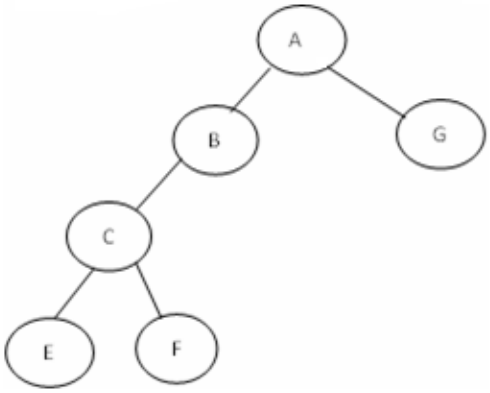
1. What is a binary tree? Construct a tree with the given string notation (T1 (T2 (T3 (T4),T5, T6), T7 (T8, (T9(T10))))).
2. Construct a tree structure for the following array representation

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T1	T2	.	T3	.	.	.	T4	.	.	.	.	.	.	.	T5

3. Write a procedure for linked list representation of binary tree. Consider the array representation provided in question 2 for constructing a linked representation.
4. "Linked list representation consumes more memory than array representation." Justify with an example.
5. Represent the following binary tree in an array.



6. Give the linked list representation of the following binary tree.



7. "In threaded binary trees, threads are used instead of pointers." Justify with example.

Answers: Self Assessment

1. (a) False (b) False (c) True (d) False
2. (a) Complete (b) Array (c) Threads
3. (a) Sequential (b) Dynamic allocation

11.8 Further Readings

Books

Reddy. P. (1999). Systematic Approach to Data Structures Using C. Bangalore: Sri Nandi Publications

D. Samantha, Classic Data Structure, Delhi: Prentice Hall of India Publications

Kamanthe A. N., Programming and Data Structures, South Asia: Dorling Kindersley (India) Publications

Online link

http://www.cis.upenn.edu/~matuszek/cit594-2005/Lectures/09-binary-trees.ppt

http://www.cprogramming.com/tutorial/lesson18.html

Unit 12: Binary Tree Traversals and Operations

CONTENTS

Objectives

Introduction

12.1 Binary Tree Traversals

12.2 Preorder Traversal

12.3 Inorder Traversal

12.4 Postorder Traversal

12.5 Binary Tree Operations

12.5.1 Insertion

12.5.2 Deletion

12.5.3 Searching

12.6 Summary

12.7 Keywords

12.8 Self Assessment

12.9 Review Questions

12.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Discuss the different kinds of binary tree traversals
- Explain in detail the preorder traversal
- Describe in detail inorder traversal
- Analyze and implement postorder traversal
- Explain binary tree operations

Introduction

In the previous chapter, we discussed the fundamentals of binary tree. In this chapter, we will discuss the traversal and other operations of a binary tree.

Traversals are the most basic operations performed on binary trees. In traversal technique, each node of a tree is systematically visited exactly once. The different traversal techniques are preorder, inorder, and post order traversal.

The binary tree operations help in the logical implementation of the structure of a binary tree. The basic operations such as insertion of a node, deletion and searching of node are performed on the binary tree structure.

12.1 Binary Tree Traversals

Binary tree traversals refer to the commonly used traversing operations on a binary tree. Traversing refers to the order in which traversing is performed on a node of the tree, its right and left sub-trees. It deals with visiting each node in a tree exactly once. A complete traversal of binary tree provides the sequential ordering of information in a tree.

A binary tree can be traversed in various ways. But the standard methods of traversal are:

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

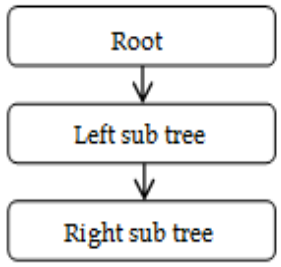
12.2 Preorder Traversal

The preorder traversal of a non-empty binary tree is defined as follows:

1. First, visit the root node
2. Next, traverse the left sub-tree of root node in preorder
3. Finally, traverse the right sub-tree of root node in preorder

The figure 12.1 depicts the functioning of preorder traversal.

Figure 12.1: Preorder Traversal



The algorithm for preorder traversal is as follows:

Input – ROOT is the pointer to the root node of binary tree

Output – Visiting all nodes in preorder manner

Data Structure – Linked structure of binary tree

Steps

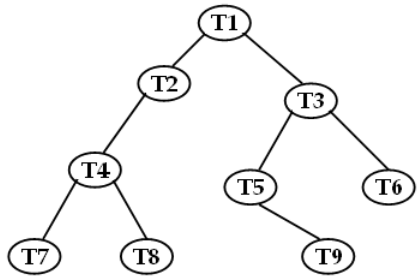
Preorder(Node *ptr)

1. ptr = ROOT //Start from ROOT
2. If (ptr ≠ null) then //If it is non-empty node
3. Visit (ptr) //Visit the node
4. Preorder (ptr→llink) //Traverse left sub-tree of node in preorder manner
5. Preorder (ptr→rlink) //Traverse right sub-tree of node in preorder manner
6. EndIf
7. Stop

Example: Consider the binary tree shown in figure 12.2. The binary tree is represented in such a way that the nodes in the binary tree are traversed in preorder traversal manner.

Figure 12.2 shows binary tree for preorder traversal.

Figure 12.2: Binary Tree for Preorder Traversal



In the figure 12.2, T1 is the root node, T2 and T3 are the sub-trees of the root node. The preorder traversal for a binary tree traverses the root node first, then the left sub-tree and finally the right sub-tree. Since the traversing process is in the order of root node, left and right sub-trees, let us assign the alphabet N for visiting root node, L for visiting left sub-tree and R for visiting right sub-tree.

The term T1NLR indicates that node T1 is the root node of the binary tree and subscript NLR indicates preorder tree traversal. The following represents the preorder traversal for binary tree present in the figure 12.2

T1NLR----->T1 T2 NLR T3 NLR //After visiting T1NLR

---->T1 T2T4 NLRμ T3 NLR //After visiting T2NLR, μis empty child

---->T1 T2 T4T7NLR T8NLR T3NLR//After visiting T4NLR

----> T1 T2 T4 T7 μμT8NLR T3NLR//After visiting T7NLR ,μis empty left/right child

---->T1 T2 T4 T7 T8 μμT3NLR//After visiting T8NLR

---->T1 T2 T4 T7 T8 T3 T5NLRT6NLR//After visiting T3NLR

---->T1 T2 T4 T7 T8 T3 T5 μT9NLRT6NLR//After visiting T5

---->T1 T2 T4 T7 T8 T3 T5 T9 μμT6NLR//After visiting T9NLR

---->T1 T2 T4 T7 T8 T3 T5 T9 T6 μμ//After visiting T6NLR

---->T1 T2 T4 T7 T8 T3 T5 T9 T6

Hence, the preorder traversal for the tree shown in figure 12.2 is T1 T2 T4 T7 T8 T3 T5 T9 T6.

The traversal in preorder starts with traversing root, right sub-tree and left sub-tree. But this traversing happens only during the downward movement of the traverse operation in a binary tree. If the upward traversing is required in a tree, then it takes place in a reverse manner. To implement upward traversing, a stack is required to save pointer variable during the tree traversal. This mode of traversing is known as iterative traverse. The general form of iterative traversal for preorder using stack is as follows:

Step 1: If the tree is empty //Check if the tree is empty

then ("empty tree") // If tree is empty write "empty tree"

return

else

Place the pointer to the root of the tree on the stack// Move the pointer to the root of stack

Step 2: Repeat step 3 while stack is not empty.

Step 3: Pop the top of the stack.

Repeat while the pointer value is not NULL.

Write (Node containing data).

If the right sub-tree is not empty, then stack the pointer to the right sub-tree and set pointer value to left sub-tree.

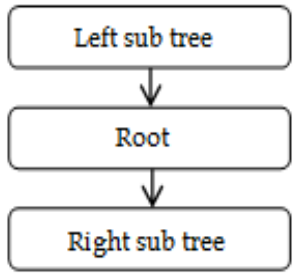
12.3 Inorder Traversal

The inorder traversal of a non-empty tree is defined as follows:

1. First, traverse the left sub-tree of the root node in inorder.
2. Next, visit the root node.
3. Finally, traverse the right sub-tree of the root node.

The figure 12.3 depicts the functioning of inorder traversal. The figure provides a functional procedure of the inorder traversal.

Figure 12.3 Inorder Traversal



The algorithm for inorder traversal is as follows:

1. Input – ROOT is the pointer to the root node of binary tree
2. Output – Visiting all nodes in inorder manner
3. Data Structure – Linked structure of binary tree

Steps:

Inorder(Node *ptr)

1. ptr = ROOT //Start from ROOT

2. If (ptr ≠ null) then //If it is non-empty node

3. Inorder (ptr→llink) //Traverse left sub-tree of node in inorder manner

4. Visit (ptr) //Visit the node

5. Inorder (ptr→rlink) //Traverse right sub-tree of node in inorder manner

6. EndIf

7. Stop

Example: Consider the binary tree shown in the figure 12.2. The inorder traversing traverses first the left sub-tree, then the node and then the right sub-tree. Since, the traversing order is left sub-tree, root node and right sub-tree, the alphabets **L, N, R** can be used for representing inorder traversal.

The term $T1_{LNR}$ indicates that node **T1** is the root node of the binary tree and subscript **LNR** indicates inorder tree traversal.

$T1_{LNR}$----->$T2_{LNR}$ T1 $T3_{LNR}$ //After visiting $T1_{LNR}$

---->$T4_{LNR}$ T2 μ T1 $T3_{LNR}$//After visiting $T2_{LNR}$

---->$T7_{LNR}$ T4 T8 $_{LNR}$ T2 μ T1 $T3_{LNR}$//After visiting $T4_{LNR}$

---->μ T7 μ T4 T8 $_{LNR}$ T2 μ T1 $T3_{LNR}$//After visiting $T7_{LNR}$

---->T7 T4 μ T8 μ T2 μ T1 $T3_{LNR}$//After visiting $T8_{LNR}$

---->T7 T4 T8 T2 T1 $T5_{LNR}$ T3 $T6_{LNR}$//After visiting $T3_{LNR}$

---->T7 T4 T8 T2 T1 μ $T5T9_{LNR}$ T3 $T6_{LNR}$//After visiting $T5_{LNR}$

---->T7 T4 T8 T2 T1 T5μ T9μ T3 $T6_{LNR}$//After visiting $T9_{LNR}$

---->T7 T4 T8 T2 T1 T5T9T3 μ T6 μ//After visiting $T6_{LNR}$

---->T7 T4 T8 T2 T1 T5T9T3 T6

Hence, the inorder traversal for the tree shown in figure 12.2 is **T7 T4 T8 T2 T1 T5T9T3 T6**

Program for inserting elements into the tree and traversing in inorder

```
#include<stdio.h>
#include<conio.h>
/*Define tree as a structure with data and pointers to the left and right sub-tree*/
struct tree
{
        long info;
        struct tree *left;
        struct tree *right;
};
/* bintree is declared as the datatype tree and initialized to Null*/
struct tree *bintree=NULL;
/*Global declaration of function insert which returns a pointer to the tree structure and accepts a pointer to tree and a long digit as parameters */
struct tree *insert(struct tree*bintree,long digit);
/*Global declaration of function inorder which does not return any value and accepts a pointer to tree as a parameter*/
void inorder(struct tree*bintree);
void main()                                          // Define main function
```

```
{
        long digit;
        clrscr();
        puts("Enter integers: and 0 to quit");
        scanf("%d",&digit);                         //Reads the first number to be inserted
        while (digit!=0)
{
        bintree=insert(bintree,digit);              // Inserts the number entered
        scanf("%d",&digit);
}
        puts("Inorder traversing of bintree:\n");
        inorder(bintree);                           //Calling inorder function to traverse
the tree
}
struct tree* insert(struct tree* bintree,long digit)        //insert function is defined
{
        if(bintree==NULL)                           //checks if the tree is empty
        {
        bintree=(struct tree*) malloc(sizeof(struct tree));     //Allocates memory for the tree
        bintree->left=bintree->right=NULL;          //Left and right sub-trees is set to NULL
        /* The digit entered is assigned to the info element of the tree node*/
        bintree->info=digit;
        }
        else
        {
        if(digit<bintree->info)             //If the entered number is less than the data of the node
        bintree->left=insert(bintree->left,digit);          //insert the digit in the left sub-tree
        else
        /*If the entered number is greater than the data of the node*/
        if(digit>bintree->info)
        bintree->right=insert(bintree->right,digit);        //insert the digit in the right sub-tree
        else
                if(digit==bintree->info)            //If entered number is equal to data of
        the node
                {
                //exits program after printing that duplicate node is present
                puts("Duplicates node:program exited");
                exit(0);
```

```
                }
        }
        return(bintree);
}
void inorder(struct tree*bintree)                    //Defining inorder function
{
        if(bintree!=NULL)                            //Checks if tree is empty
        {
        inorder(bintree->left);                      //Calls the inorder function for left sub-trees
        printf("%4ld",bintree->info);                // Prints data of the node
        inorder(bintree->right);                     // Inorder function of right sub-trees
        }
}
```

Output:

Enter integers and 0 to quit

6 1 2 3 7 8 9 0

Inorder traversing of bintree

1 2 3 6 7 8 9

In this program,

1. First the structure tree is defined. It contains a variable **info** of long type and pointers to the right and left sub-trees.
2. The variable **bintree** is declared as data type tree and initialized to NULL.
3. The function **insert** and **inorder** are globally declared.
4. In the **main()** function,
   (a) First, the numbers to be entered are declared using **long** data type
   (b) Then, the digit entered is read by the computer.
   (c) If the digit is not 0, the **insert()** function is called to insert the entered digit into the binary tree. Step b, and c are repeated until the digit entered is 0.
   (d) Finally, the **inorder()** function is called to traverse the tree.
5. The **insert()** function is defined. It accepts a pointer to a tree and a digit to be inserted in the tree as parameters. The insert function performs the following steps:
   (a) It checks if the tree is empty or non-empty.
   (b) If the tree is empty it assigns memory to the node, sets the left and right pointers of the node as **NULL** and assigns the digit to the **info** variable of the node.
   (c) If the tree is non-empty, then it performs the following steps:
      (i) If the digit entered is less than the **info** stored in the node, it recursively calls itself to enter the digit in the left sub-tree.
      (ii) If the digit entered is greater than the **info** stored in the node, it recursively calls itself to enter the digit in the left sub-tree.

(iii) If the entered digit is equal to the **info** stored in the node, then it displays a message "Duplicates node: program exited" and exits.

(d) The function **inorder()** is then defined. It accepts a pointer to a tree and a digit to be inserted in the tree as parameters

(i) It checks if the tree is empty or non-empty.

(ii) If the tree is non-empty, it traverses the left sub-tree first, then prints the value of the variable **info** stored in the node and then traverses the right sub-tree.

In the inorder traversal, the nodes traversal starts with visiting right sub-tree, root and left sub-tree. While traversing using stacks, the left sub-tree in a binary tree is traversed by moving down the tree towards left and then pushing node with data into the stack until the left sub-tree pointer node is **NULL**. Once the left sub-tree is traversed, the stack becomes non-empty, then we can pop the elements from the stack and print the data, and then traverse the pointer towards right sub-tree. This process continues until right sub-tree is **NULL.** The algorithm for inorder traversal using stacks is as follows:

Step 1: If the tree is empty then

```
{
        write( "empty tree")
        return
}
else
        Place the pointer to the root of the tree on the stack
```

Step 2: Repeat step 4 while stack is not empty.

Step 3: Repeat while pointer value is not NULL and stack the pointer to the left sub-tree.

Repeat while the pointer is not NULL.

Write (Node containing data)

If the right sub-tree is not empty, then stack the pointer to the right sub-tree and set pointer to the right sub-tree.

Notes

In the inorder traversal, if we replace left by right and right by left, converse inorder traversal is obtained. The converse inorder traversal displays the tree contents in a tree fashion.

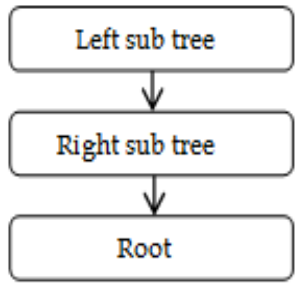
12.4 Postorder Traversal

The postorder traversal of a non-empty tree is defined as follows:

1. First, traverse the left sub-tree of the root node in postorder.
2. Next, the right sub-tree of the root node in postorder.
3. Finally, visit the root node.

The figure 12.4 depicts the functioning of postorder traversal. The figure represents the functionality of postorder traversal.

Figure 12.4: Postorder Traversal



The algorithm for postorder traversal is as follows:

Input – ROOT is the pointer to the root node of binary tree

Output – Visiting all nodes in postorder manner

Data Structure – Linked structure of binary tree

Steps

1. ptr = ROOT //Start from ROOT
2. If (ptr ≠ null) then //If it is non-empty node
3. Postorder (ptr→llink) //Traverse left sub-tree of node in postorder manner
4. Postorder (ptr→rlink) //Traverse right sub-tree of node in postorder manner
5. Visit (ptr) //Visit the node
6. EndIf
7. Stop

Example: Consider the binary tree shown in the figure 12.2. The postorder traversing depends on traversing first the left sub-tree, right sub-tree and then the node. Since, the traversing process starts with the left sub-tree, right sub-tree and root node, let us consider the alphabets **L, R, N** for postorder traversal.

The term $\mathbf{T1_{LRN}}$ indicates that node **T1** is the root node of the binary tree and subscript **LRN** indicates postorder tree traversal.

$T1_{LRN}$----->$T2_{LRN}T3_{LRN}T1$ //After visiting $T1_{LRN}$

----->$T4_{LRN}$ μ T2 $T3_{LRN}T1$//After visiting $T2_{LRN}$

----->$T7_{LRN}$ T8 $_{LRN}$T4 T2 $T3_{LRN}T1$//After visiting $T4_{LRN}$

----->μμT7 T8 $_{LRN}$T4 T2 $T3_{LRN}$ T1//After visiting $T7_{LRN}$

----->T7 μμT8 T4T2 $T3_{LRN}T1$//After visiting $T8_{LRN}$

----->T7 T8 T4 T2 $T5_{LRN}$ $T6_{LRN}T3T1$//After visiting $T3_{LRN}$

----->T7 T8 T4 T2 μT9$_{LRN}$T5T6$_{LRN}$T3 T1//After visiting T5$_{LRN}$

----->T7 T8 T4 T2 μμT9T5T6$_{LRN}$T3 T1//After visiting T9$_{LRN}$

----->T7 T8 T4 T2 T9 T5μμT6 T3 T1//After visiting T6$_{LRN}$

----->T7 T8 T4 T2 T9 T5T6 T3 T1

Hence, the postorder traversal for the tree shown in figure 12.2 is **T7 T8 T4 T2 T9 T5 T6 T3 T1**

The post order traversal starts with left sub-tree, then moves to the right sub-tree, and then finally to the root. Considering the postorder traversal using stacks, each node is stacked twice during the traversal of left sub-tree and right sub-tree. To distinguish between the left sub-tree and right sub-tree, a traversing flag is used. During the traverse of right sub-tree, flag is set to 1. This helps in checking the flag field of the corresponding node. If the flag of a node is negative, then right sub-tree is traversed, else the left sub-tree is traversed. The algorithm for postorder of binary tree using stacks is as follows:

Step 1: If the tree is empty then

{

write ("Empty tree")

return

}

else

Initialize the stack and pointer value to the root of tree

Step 2: Start an infinite loop and repeat till step 5

Step 3: Repeat while pointer value is no NULL, stack current pointer value. Set pointer value to left sub-tree

Step 4: Repeat while top pointer on stack is negative

Pop pointer off stack

write (value of pointer)

If the stack is empty

then return

Step 5: Set pointer value to the right sub-tree of the value on top of the stack.

Step 6: Stack the negative value of the pointer to right sub-tree

Program for inorder, preorder and postorder tree traversals

```
#include<stdio.h>
#include<conio.h>
struct node                          //Declare node as struct variable
{
        int data;                    //Declare data with data type int
        struct node *right, *left;   // The node stores pointers to the right and left sub-trees.
}root,*p,*q;                         // declare root as a variable of type node and p and q
as pointers to node
```

```
struct node *make(int y)
{
        struct node *newnode;                   // Declare newnode as pointer to struct node
        newnode=(struct node *) malloc(sizeof(struct node));  //Allocate space in memory
        /*Assign object data to newnode and initialize to variable y*/
        newnode->data=y;
        /*Declare right newnode and left newnode to NULL*/
        newnode->right=newnode->left=NULL;
        return(newnode);
}
void left(struct node *r,int x)                         // Define left sub-tree function
{
/*Checks if left sub-trees is not equal to NULL*/
if(r->left!=NULL)
        printf("\n Invalid !");                         // Prints invalid
else
        r->left=make(x);                                //Initialize left sub-tree
}
void right(struct node *r,int x)                        //Define right sub-tree
{
/*Checks if right sub-tree is not equal to NULL*/
if(r->right!=NULL)
        printf("\n Invalid !");                         // Prints invalid
else
        r->right=make(x);                               //Initialize right sub-tree
}
void inorder(struct node *r)                            //Define inorder traversal function
{
/*Conditional statement, check if r is not equal to NULL*/
if(r!=NULL)
{
                / *Recursively call inorder passing the address of the left sub-tree*/
                inorder(r->left);
                printf("\t %d", r->data);               //Prints the data of the node
                / *Recursively call inorder passing the address of the left sub-tree*/
                inorder(r->right);
        }
}
```

```
void preorder(struct node *r)                    //Define preorder function
{
/*Checks if r is not equal to NULL*/
if(r!=NULL)
{
        printf("\t %d",r->data);                 //Prints the data of the node
        / *Recursively call preorder passing the address of the left sub-tree*/
        preorder(r->left);
        / *Recursively call preorder passing the address of the right sub-tree*/
        preorder(r->right);
}
}
void postorder(struct node *r)                   //Define postorder function
{
if(r!=NULL)                                      //Checks if r is not equal to NULL
{
        / *Recursively call postorder passing the address of the left sub-tree*/
        postorder(r->left);
        / *Recursively call postorder passing the address of the left sub-tree*/
        postorder(r->right);
        printf("\t %d",r->data);          //Prints the data of the node
}
}
void main()
{
        int no;                           //Declare variable no
        int choice;                       //Declare variable choice
        clrscr();
        printf("\n Enter the root:");
        scanf("%d",& no);                 //Reads the number entered
        root=make(no);                    //Initialize the number to root
        p=root;                           // Value of root is then assigned to variable p
        while(1)                          //Checks the conditions provided in while loop
                {
                        /*Prints the statement "Enter another number*/
                        printf("\n Enter another number:"); scanf("%d", &no);
                        //Reads the number entered
                        /*Conditional statement, check if no is equal to -1*/
```

```
if(no==-1)
        break;              //If condition is true, the if loop breaks
p=root;                     //Assign value of root to p variable
q=root;                     //Assign value of root to q variable
/*Check if no is not equal to variable p and q not equal to NULL*/
while(no!=p->data && q!=NULL)
{
        p=q;
        if(no<p->data)              //Check if no is less than variable p
        /*Set q to the left sub-tree of p*/
                q=p->left;
        else
        /*Set q to the right sub-tree of p*/
                q=p->right;
}
/*Check if variable no is less than p variable with data*/
if(no<p->data)
{

        /*prints the node of left tree*/
        printf("\n Left branch of %d is %d",p->data,no); left(p,no);
}
else
{
        right(p,no);
        /*prints the node of right tree*/
        printf("\n Right Branch of %d is %d",p->data,no);//
}
while(1)
{
printf("\n 1.Inorder Traversal \n 2.Preorder Traversal \n 3.Postorder
Traversal \n 4.Exit");
printf("\n Enter choice:");
scanf("%d",&choice);                // Reads the choice entered
switch(choice)
{
/*Switches to inorder function and performs inorder traversal*/
        case 1 :inorder(root);
```

```
                                        break;
                /*Switches to preorder function and performs preorder traversal*/
                        case 2 :preorder(root);
                                break;
                /*Switches to postorder function and performs postorder traversal*/
                        case 3 :postorder(root);
                                break;
                        case 4 :exit(0);                        //Exits from the function
                        default:printf("Error ! Invalid Choice ");      //Prints invalid statement
                                break;
                        }
        }
        getch();
        }
}
```

Output:

```
Enter the root : 5
Enter another number: 7
Right branch of 5 is 7
1. Inorder traversal
2. Preorder traversal
3. Postorder traversal
4. Exit
Enter choice: 1
    5       7
1. Inorder traversal
2. Preorder traversal
3. Postorder traversal
4. Exit
Enter choice: 2
        5   7
1. Inorder traversal
2. Preorder traversal
3. Postorder traversal
4. Exit
Enter choice: 3
    7   5
```

In this program,

1. First, the header file **stdio.h** is included using **include** keyword.
   (a) The variable **node** is defined as a structure. It has an integer variable **data** and pointers to its **left** and **right** sub-trees, **root** is declared as a variable of type node. The variables **p** and **q** are declared as pointers to **node**.
2. Then, the **make()** function is defined. It returns a pointer to the structure **node** and accepts an integer variable **y** as a parameter. It executes the following steps:
   (a) It declares **newnode** as pointer to the structure **node** and assigns memory to it.
   (b) It assigns the integer **y** to the data variable of the **newnode**.
   (c) It sets the right and left sub-tree pointers of the **newnode** to NULL.
3. Then, the **left()** sub-tree function is defined. It accepts as parameters **r** which is a pointer to the structure **node** and an integer variable **x.** It executes the following steps:
   (a) It checks if the left pointer of **r** is empty. If it is not empty, then it calls the make function passing x as a parameter.
4. Then, the **right()** sub-tree function is defined. It accepts as parameters **r** which is a pointer to the structure **node** and an integer variable **x.**
   (a) It checks if the right pointer of **r** is empty. If it is not empty, then it calls the make function passing x as a parameter.
5. Then, the **inorder()** function is defined. It accepts as parameters **r** which is a pointer to the structure **node.**
   (a) It checks if **r** is non-empty. If it is non-empty, it then traverses the left sub-tree, prints the data and then traverses the right sub-tree.
6. Then, the **preorder()** function is defined. It accepts as parameters **r** which is a pointer to the structure **node.**
   (a) It checks if **r** is non-empty. If it is non-empty, it prints the data, traverses the left sub-tree, and then traverses the right sub-tree.
7. Then, the **postorder()** function is defined. It accepts as parameters **r** which is a pointer to the structure **node.**
   (a) It checks if **r** is non-empty. If it is non-empty, it then traverses the left sub-tree, then the right sub-tree and then prints the data.
8. In the **main()** function,
   (a) The variables **no, choice** are declared as integer variables.
   (b) The value for the **root** node is accepted and added to the tree by calling the **make()** function.
   (c) The value of **root** is then assigned to variable **p** and **q**.
   (d) The program execution enters a while loop in which the following steps are performed:
      (i) First, it accepts another integer **no.**
      (ii) Then, the while loop is exited if **no** is equal to -1.
      (iii) Then, the following steps are repeatedly performed if the no. entered is not equal to variable data of **p** and **q is** not equal to **NULL.**
         I. First, **p** is assigned the value of **q**.
         II. If the number is lesser than the data of **p, q** is assigned the address of the left sub-tree else **q** is assigned the address of the right sub-tree

(iv) If **no** is less than **data** of variable **p** the function **left()** is called passing **p** and **no** as the parameters, else the function **right ()** is called.

(e) A while loop is used to obtain the choice of traversal.

(i) If **1** is entered, inorder traversal is selected and the **inorder** function is executed.

(ii) If **2** is entered, preorder traversal is selected and **preorder** function is executed.

(iii) If **3** is selected, preorder traversal is selected and **postorder** function is executed.

(iv) If **4** is entered, the while loop is exited.

(v) If wrong digit is entered, an error message is printed on the screen.

(f) The **getch()** prompts the user to enter a key to exit the program.

Task

Consider the algebraic expression (A +(B-C)) / ((D-E) * (F+G-H))

Construct a binary tree with the above expression and traverse the tree in inorder, preorder, and postorder manner by applying the tree traversal algorithms.

12.5 Binary Tree Operations

The binary tree operations are required to perform certain functions on a binary tree. The binary tree can be structured logically and implemented using the binary tree operations. The major operations on a binary tree are:

1. Insertion
2. Deletion
3. Searching

12.5.1 Insertion

The insertion operation deals with inserting a new node at any position in the binary tree. Before inserting the node, it is important to identify the location of insertion. Hence, searching operation must be performed before performing insertion operation. Search and insertion operation are inter-related.

If an element **T1** is given, and the task is to search and insert the element, then the operation is performed as follows:

1. If element **T1** < element of root node, then select left branch node and place the element **T1** at the left side of the node.
2. If element **T1** > element of root node then place the element in the right branch of root node.

This process continues until all the nodes are visited in the tree.

Insertion is a two-step process. The first step deals with searching the node in a binary tree and the next step deals with inserting the node and providing a link for the new node. Insertion operation can be done on a sequential representation as well as linked representation of binary tree. Let us now discuss the insertion operation on sequential representation of binary tree.

In the insertion operation, the key element is identified and a new node is inserted with its data element.

The algorithm for insertion operation is as follows:

Input – KEY is the node in the binary tree to identify the location of inserting a new node.

Output – Newly inserted node with data as INFO.

Data structure – Array A stores binary tree.

Steps

1. L = SEARCH_SEQ(1, KEY) // Search for key node in the tree
2. If (L = 0) then
   (a) Print "Search is unsuccessful: No insertion"
   (b) Exit
3. EndIf // Quit execution
4. If (A[2 * L] = NULL) or (a[2 * L +1] = NULL)then //If the left and right sub-trees are empty
   (a) Read option to read as left (L) or right (R) //child (give option = L/R)
   (b) If (option = L) then
      (i) If A[2 * L] = NULL then //Left link is empty
      1. A[2 * L] = ITEM // Store it in the array **A**
      (ii) Else // Cannot be inserted as left child
      1. Print "Desired insertion is not possible" //already has left child
      2. Exit //Return to end of procedure
      (iii) EndIf
   (c) EndIf
   (d) If (option = R) then //Move to right side
      (i) If (A[2 * L +1] = NULL) then // Right link is empty
         1. A[2 * L +1] = ITEM //Store it in the array **A**
      (ii) Else //Cannot be inserted as right child
         1. Print "Desired operation is not possible" //already has a right child
         2. Exit // Return to the end of procedures
      (iii) EndIf
   (e) EndIf //Key node having both the child
5. Else //Key node does not have any empty link
   (a) Print "ITEM cannot be inserted as leaf node"
6. EndIf
7. Stop

The recursive algorithm of search operation for inserting a node is as follows:

Input – KEY is the item of search, INDEX is the index of node from which the searching process starts.

Output – LOCATION is where the item KEY is located

Data structure – Array **A** is for storing binary tree. SIZE indicates the size of **A**

Steps

1. L = SEARCH_SEQ(1, KEY)

i = INDEX //Start search from the root node

2. If (A[i] ≠ KEY) then //Present node is not key node

1. If (2 * i ≤ SIZE) then //If left sub-tree is not exhausted

(a) SEARCH_SEQ (2*i, KEY) //Search in left sub-tree

2. Else //Left sub-tree is exhausted and KEY not found

(a) If(2*i + 1 ≤ SIZE)then //If right sub-tree is not exhausted

1. SEARCH_SEQ (2*i + 1, KEY) //Search in right sub-tree

(b) Else //KEY is found neither in left or right sub-tree

1. Return(0) //Return NULL address for unsuccessful search

(c) EndIf

EndIf

3. Else

(d) Return(i) //Return address where KEY is found

4. Stop

12.5.2 Deletion

Deletion operation is used to delete a node from a non-empty binary tree. To delete a node in the binary tree, it is important to visit the parent node of the node to be deleted. Deletion operation can be done on a sequential representation as well as linked representation of binary tree. Let us discuss the deletion operation on sequential representation of binary tree.

In the deletion operation, the parent node of the deleting node must be traversed initially.

The algorithm for deletion operation is as follows:

Input – **ITEM** is the data of a node with which the node is identified for deletion.

Output – Binary tree containing a node without data as **ITEM**.

Data structure – Array **A** is for storing binary tree. **SIZE** indicates size of array **A**.

Steps

1. Flag = FALSE //Start from root node

2. L = SEARCH_SEQ (1, KEY) //Start searching from starting

3. If (A[2 * L] = NULL) and (A[2 * L +1]=NULL) //Test for leaf node

(a) flag = TRUE // If leaf node is present, then delete it

(b) A[L] = NULL

4. Else

(a) Print "The node containing ITEM is not a leaf node"

5. EndIf

6. If (flag = FALSE)

(a) Print "NODE does not exist :No deletion"

7. EndIf

8. Stop

12.5.3 Searching

Searching operation is a part of insertion and deletion operation. It is a basic step to be performed before inserting or deleting a node in the binary tree. In the insertion operation, the location for the new node to be inserted must be identified whereas, in the deletion operation, searching of the parent node of the deleting node is required. The algorithm of searching operation for inserting a node has been explained in the section 12.5.1.

Lab Exercise

1. Write a C program to traverse the following algebraic expression in inorder, postorder and pre-order traversal

   (A +B) – C + ((D * E) + (F/ G) – H)

2. Write a C program to obtain the swapped version of binary tree.

12.6 Summary

- The binary tree can be traversed in various ways. The three major binary tree traversal techniques are inorder, preorder and postorder traversals.
- The various operations performed on binary tree are insertion, deletion and searching operations.
- The searching operation is related with the insertion operation to identify the location to insert a node
- In deletion operation, searching method is used to identify the parent node of the deleting node.

12.7 Keywords

Converse Inorder Traversal: The tree obtained after the inorder traversal is represented in the original tree format.

Infinite Loop: A series of instructions in a computer program which, on execution, result in a cyclic repetition of the same instructions.

Iterative Traverse: Repetitive traversal.

Traversing Flag: Indicates if the node was visited during traversal.

12.8 Self Assessment

1. State whether the following statements are true or false:

   (a) A complete insertion operation of binary tree provides sequential ordering of information in a tree.

   (b) In the post order traversal of binary tree, first the right sub-tree is traversed.

2. Fill in the blanks:

   (a) To delete a node in the binary tree, it is important to visit the …….. node of the node to be deleted.

   (b) The…….. method deals with visiting each node in a tree exactly once.

   (c) The nodes in the binary tree can be accessed by calculating the …… of every node.

3. Select a suitable choice for every question:

   (a) Which among the following traversal of binary tree starts with traversing the root node?

      (i) Preorder

      (ii) Inorder

      (iii) Postorder

(b) Which among the following tree traversal deals with the term LRN?

(i) Inorder

(ii) Preorder

(iii) Postorder

(c) Which among the following traversals involve traversing from left tree, root and right tree?

(i) Inorder

(ii) Preorder

(iii) Postorder

(d) Which among the following operations is performed before performing insertion operation?

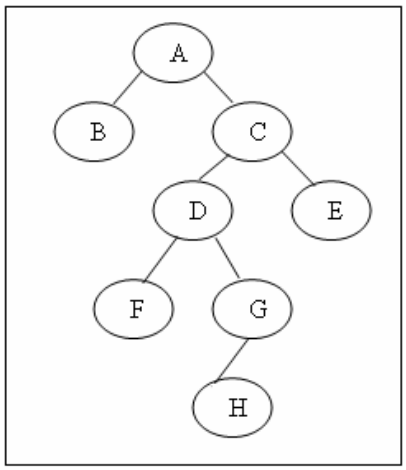
(i) Searching

(ii) Deletion

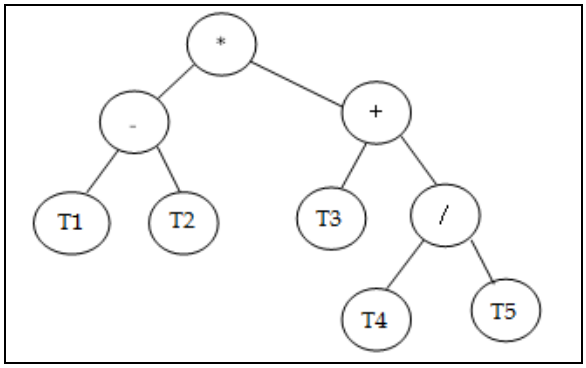
(iii) Modification

12.9 Review Questions

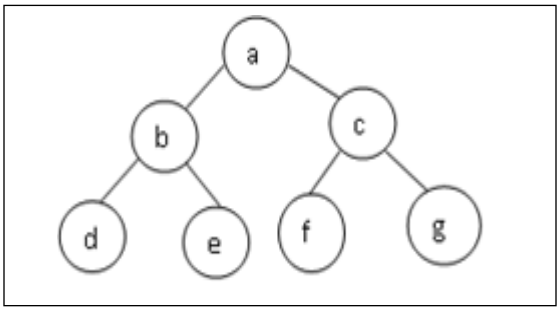
1. In the binary tree given, delete node D and insert node I.



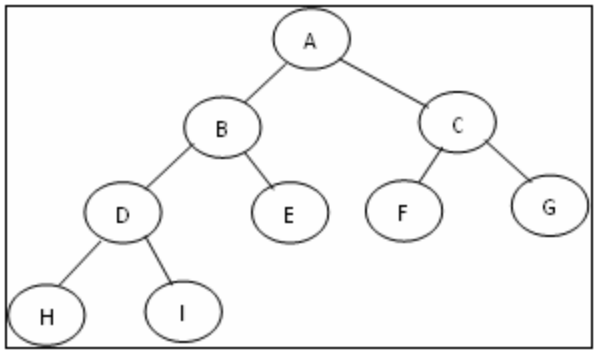
2. Perform the preorder traversal for the given binary tree.



3. Perform an inorder traversal for the given binary tree.



4. Perform a postorder traversal for the given binary tree.



5. For the following algebraic expression, construct a binary tree.

   (T1+T2) + T3 – (T5 * T6)

Answers: Self Assessment

1. (a) False (b) False
2. (a) Parent (b) Traversing (c) Index
3. (a) Preorder (b) Postorder (c) Inorder (d) Searching

12.10 Further Readings

Books

Reddy. P. (1999). Systematic Approach to Data Structures Using C. Bangalore: Sri Nandi Publications

D. Samantha, Classic Data Structure, Delhi: Prentice Hall of India Publications

Kamanthe A. N., Programming and Data Structures, South Asia: Dorling Kindersley (India) Publications

Online link

http://www.ehow.com/how_2056293_create-binary-tree-c.html

http://www.cis.upenn.edu/~matuszek/cit594-2005/Lectures/09-binary-trees.ppt

http://www.cprogramming.com/tutorial/lesson18.html

Unit 13: Binary Search Trees

CONTENTS

Objectives

Introduction

13.1 Binary Search Tree Operations

13.1.1 Searching in Binary Search Trees

13.1.2 Inserting in Binary Search Trees

13.1.3 Deleting in Binary Search Trees

13.1.4 Other Operations

13.2 Summary

13.3 Keywords

13.4 Self Assessment

13.5 Review Questions

13.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Explain insertion operation in a binary search tree
- Describe searching operation in a binary search tree
- Explain deletion operation in a binary search tree
- Find the height of a binary search tree
- Determine the minimum and maximum value in a binary search tree
- Write a C function to count the leaves and nodes of a binary search tree

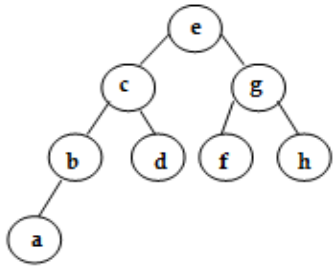
Introduction

You have already learnt about binary trees. The disadvantage of binary trees is that data is stored in these trees in any order and hence, the time taken to search these trees is longer. Search trees are data structures that support many dynamic-set operations such as searching, finding the minimum or maximum values, insertion, and deletion. Binary search trees, AVL trees and B+ trees are examples of search trees.

A binary search tree (BST) has binary nodes. In a binary search tree, for a given node with value **n**, each node to the left has a value lesser than **n** and each node to the right has a value greater than **n**. This applies recursively down the left and right sub-trees.

Figure 13.1 shows a binary search tree where characters are stored in the nodes.

Figure 13.1: A Binary Search Tree



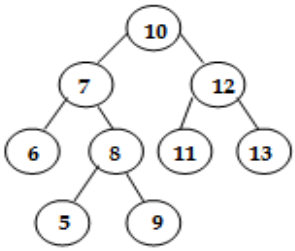
Binary search trees provide an efficient way to search through an ordered collection of objects. Consider searching an ordered list. The search must proceed successively from one end of the list to the other. On an average, **n/2** nodes must be compared for an ordered list that contains **n** nodes. In the worst case, all **n** nodes need to be compared. For a large collection of objects, this is very expensive.

Binary search tree enables searching quickly through the nodes. The longest path to search is equal to the height of the tree. Thus, the efficiency of a binary search tree depends on the height of the tree. For a tree with **n** nodes, the smallest possible height is **log n** and that is the number of comparisons that are needed on an average to search the tree. A tree must be balanced to obtain the smallest height, i.e., both the left and right sub-trees must have the same number of nodes.

Did you know? The trees which are unbalanced are called degenerate trees. For a degenerate tree with n nodes, an average of n/2 comparisons is needed, with a worst case of n comparisons.

Thus, binary search trees are node based data structures used in many system programming applications for managing dynamic sets. Another example for a binary search tree is given in Figure 13.2. As discussed, all the elements in the left sub-tree are lesser than the root node and the elements in the right sub-tree are greater than the root node.

Figure 13.2: Example for Binary Search Tree



In the binary search tree represented in figure 13.2, 10 is the root node and all the elements in the left sub-tree are lesser than 10 and the elements in the right sub-tree are greater than 10. Every node in the tree satisfies this condition for the existing left and right sub-trees.

Notes

We use binary search trees for applications such as searching, sorting, and in–order traversal.

13.1 Binary Search Tree Operations

The four main operations that we perform on binary trees are:

1. Searching
2. Insertion
3. Deletion
4. Traversal

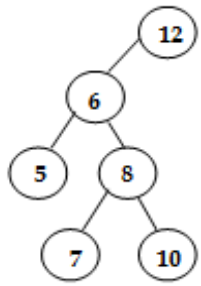
13.1.1 Searching in Binary Search Trees

In searching, the node being searched is called as key node. We first match the key node with the root node. If the value of the key node is greater than the current node, then we search for it in the right sub-tree, else we search in the left sub-tree. We continue this process until we find the node or until no nodes are left. The pseudo code for searching a binary search tree is as follows:

Pseudocode for a Binary Search Tree

```
find(X, node){
if(node = NULL)
    return NULL
if(X = node:data)
    return node
else if(X<node:data)
    return find(Y,node:leftChild)
else if(X>node:data)
    return find(X,node:rightChild)
}
```

Figure 13.3: Binary Search Tree



13.1.2 Inserting in Binary Search Trees

To insert a new element in an existing binary search tree, first we compare the value of the new node with the current node value. If the value of the new node is lesser than the current node value, we insert it as a left sub-node. If the value of the new node is greater than the current node value, then we insert it as a right sub-node. If the root node of the tree does not have any value, we can insert the new node as the root node.

Algorithm for Inserting a Value in a Binary Search Tree

1. Read the value for the node that needs to be created and store it in a node called NEW.
2. At first, if (root! =NULL) then root = NEW.
3. If (NEW->value < root->value) then attach NEW node as a left child node of root, else attach NEW node as a right child node of root.
4. Repeat steps 3 and 4 for creating the desired binary search tree completely.

When inserting any node in a binary search tree, it is necessary to look for its proper position in the binary search tree. The new node is compared with every node of the tree. If the value of the node which is to be inserted is more than the value of the current node, then the right sub-tree is considered, else the left sub-tree is considered. Once the proper position is identified, the new node is attached as the left or right child node. Let us now discuss the pseudo code for inserting a new element in a binary search tree.

Pseudocode for Inserting a Value in a Binary Search Tree

```
//Purpose: Insert data object X into the Tree
//Inputs: Data object X (to be inserted), binary-search-tree node
//Effect: Do nothing if tree already contains X;
// otherwise, update binary search tree by adding a new node containing data object X
insert(X, node){
   if(node = NULL){
     node = new binaryNode(X,NULL,NULL)
     return
    }
    if(X = node:data)
     return
   else if(X<node:data)
     insert(X, node:leftChild)
   else                          // X>node:data
      insert(X, node:rightChild)
}
```

Example: Consider figure 13.4. In this figure, 35 has to be inserted. First, 35 is compared with the value of root node i.e., 10. As 35 is greater than 10, the right sub-tree is considered. Now, 35 is compared with 22. As it is greater than 22, the search moves to the right sub-tree. 35 is then compared with 34 and again the search moves to the right sub-tree. Now, 35 is compared with 40, but as it is lesser than 40, we need to move to the left branch of 40. But since the node 40 has no left child, 35 is attached as the left child of 40. After the insertion of 35, the tree looks as shown in the figure 13.5.

Figure 13.4 shows binary tree before insertion.

Figure 13.4: Binary Tree Before Insertion

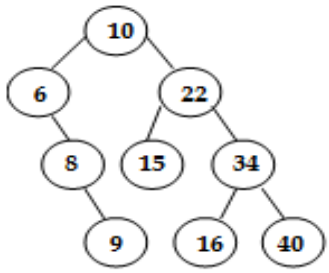
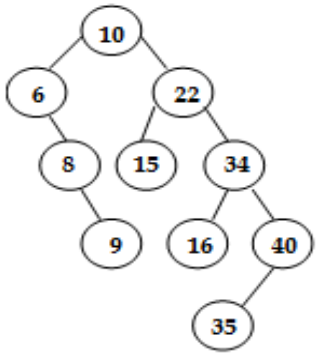


Figure 13.5 shows binary tree after insertion.

Figure 13.5: Binary Tree After Insertion



13.1.3 Deleting in Binary Search Trees

If the node to be deleted has no children, we can just delete it. If the node to be deleted has one child, then the node is deleted and the child is connected directly to the parent node.

There are mainly three cases possible for deletion of any node from a binary search tree. They are:

1. Deletion of the leaf node
2. Deletion of a node that has one child
3. Deletion of a node that has two children

We can delete an existing element from a binary search tree using the following pseudocode:

Pseudocode for Deleting a Value from a Binary Search Tree

//Purpose: Delete data object X from the Tree

//Inputs: Data object X (to be deleted), binary-search-tree node

//Effect: Do nothing if tree does not contain X;

// else, update binary search tree by deleting the node containing data object X

```
delete(X, node){
   if(node = NULL)    //nothing to do
     return
   if(X<node:data)
     delete(X, node:leftChild)
   else if(X>node:data)
     delete(X, node:rightChild)
   else {             // found the node to be deleted. Take action based on number of node children
      if(node:leftChild = NULL and node:rightChild = NULL){
         delete node
     node = NULL
     return
   }
   else if(node:leftChild = NULL){
     tempNode = node
     node = node:rightChild
     delete tempNode}
   else if(node:rightChild = NULL){
     tempNode = node
     node = node:leftChild
     delete tempNode
     }
   else { //replace node:data with minimum data from right sub-tree
      tempNode = findMin(node.rightChild)
      node:data = tempNode:data
       delete(node:data,node:rightChild)
       }
   }
```

Pseudocode for Finding Minimum Value from a Binary Search Tree

```
//Purpose: return least data object X in the Tree
//Inputs: binary-search-tree node node
// Output: bst-node n containing least data object X, if it exists; NULL otherwise
findMin(node)
{
   if(node = NULL)          //empty tree
     return NULL
   if(node:leftChild = NULL)
     return node
```

```
    return findMin(node:leftChild)
}
```

Deletion of a Leaf Node

Deletion of a leaf node is considered to be the simplest form of deletion, wherein the left or right pointer of the parent node is set as NULL. From the given tree in figure 13.6, the node with value 6 has to be deleted. Hence, the left pointer of its parent node is set as NULL, i.e., left pointer of node with value 7 is set to NULL. The figure 13.7 represents the tree after deletion of node holding value 6.

Figure 13.6: Before Deletion

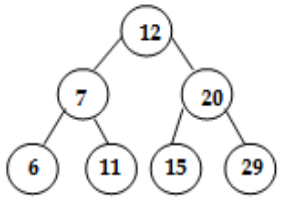
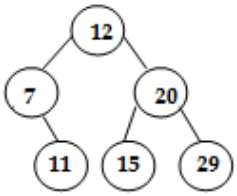


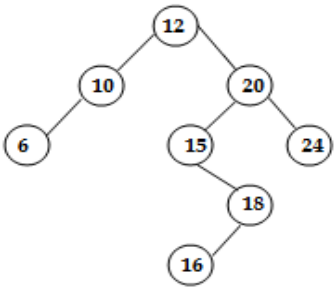
Figure 13.7: After Deletion



Deletion of a Node That Has One Child

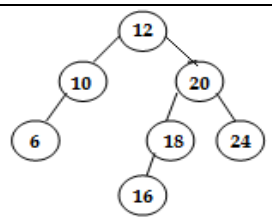
Consider figure 13.8, to understand deletion of a node that has one child node. If the node 15 has to be deleted, node 18 must be copied to the place of 15 and then the node must be set free. It is noted that the inorder successor is always copied at the position of a node being deleted.

Figure 13.8: Before Deletion



The figure 13.9 represents the tree after deletion of node 15.

Figure 13.9: After Deletion



Deletion of a Node That Has Two Children

Consider figure 13.10 to understand deletion of a node that has two children. The node with the value 8 needs to be deleted and also the inorder successor of node 8 needs to be found. The inorder successor will be copied at the location of the node. The figure 13.11 represents the tree after deletion of node holding value 8.

Figure13.10: Before Deletion

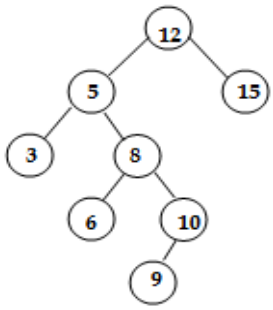
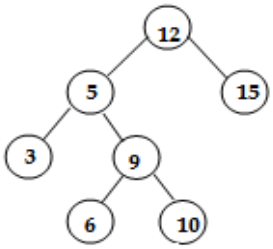


Figure 13.11: After Deletion



Thus, 9 must be copied at the position where the value of node was 8 and the left pointer of 10 must be set as NULL. This completes the entire deletion procedure.

Traversal of a binary search tree is the same as traversal of a binary tree which has been explained in Unit 12.

Did you know? The insertion, deletion and search operations have an average case complexity of O(log n), where n is the number of nodes in the binary search tree.

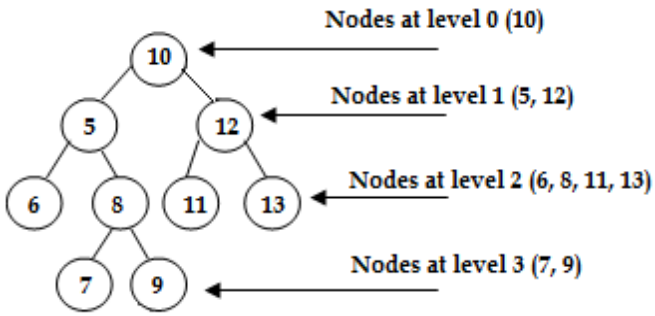
Task Write a pseudocode for traversing a binary search tree recursively.

13.1.4 Other Operations

Finding the Height of a Tree

The maximum level is referred as the height of the tree. The root is always at zero level, the adjacent nodes to the root are the first level, and so on. In the given figure 13.12, the height of the tree is 3. The height of the tree is also referred as the depth of the tree.

Figure 13.12: Binary Tree



The C function to find the height of a tree is shown in the following example.

Example:

```
struct node
{
    int element;
    struct node *L;
    struct node *R;
};
typedefstruct node* NODE;
int max(int x, int y)              // function to find maximum of two numbers
{
    return(x>y)?x:y;
  }
int height(NODE root)              // function to find the height of the tree
 {
    if(root==NULL)
       return 0;
     return 1+max(height(root->L),height(root->R));
}
```

In this example,

1. A structure named node is created. It consists of two pointer variables named **L** and **R** that point to the left node and right node.

2. An object called NODE is created to access the structure element.
3. Two functions are created namely max and height.
4. The two arguments x and y are compared by the max function, and the greater number is returned.
5. The value of root is checked by the height function. If root is NULL, 0 is returned.
6. The max function is recursively called by the height function by passing the left and right sub-trees as parameters. The value returned by the max function is incremented and returned.

To Find the Maximum and Minimum Value in a tree

In a binary search tree, a node with maximum value is found by traversing and obtaining the extreme right node of the tree. If there is no right sub-tree, then the root is returned as the node that holds the item of the highest value.

The C function to return the address of highest item in binary search tree is shown in the following example.

Example:

```
struct node
{
   int element;
   struct node *L;
   struct node *R;
 };
typedef struct node* NODE;
NODE maximum(NODE root)
{
   NODE data;
   if(root==NULL)
     return root;
   data=root;
   while(data->R!=NULL)
     data=data->R;        // find right most node in binary search tree
   return data;
  }
```

In this example:

1. A structure named node is created. It consists of two pointer variables named **L** and **R** that point to the left node and right node.
2. An object called NODE is created to access the structure element.
3. The function maximum is defined which accepts the address of the root node as the parameter.
4. In the function maximum():
   (a) A variable data is declared as type NODE.
   (b) If root is entered as NULL, then the function returns root. Else, the value of root is assigned to the variable data.
   (c) Using the while loop, the extreme right node in the tree is found and its address is returned.

In a binary search tree, a node with minimum value is found by traversing and obtaining the extreme left node of the tree. If there is no left sub-tree, then the root is returned as the node which holds the item of the least value.

The C function to return the address of least item in binary search tree is shown in following example.

Example:

```
struct node
 {
    int element;
    struct node *L;
    struct node *R;
 };
typedef struct node* NODE;
NODE minimum(NODE root)
 {
      NODE data;
      if(root==NULL)
        return root;
     data=root;
     while(data->L!=NULL)
       data=data->L;            // finds left most node in binary search tree
    return data;
}
```

In this example:

1. A structure named node is created. It consists of two pointer variables named **L** and **R** that point to the left node and right node.
2. An object called NODE is created to access the structure element.
3. The function minimum is defined which accepts the address of the root node as the parameter.
4. In the function minimum():
   (a) A variable data is declared as type NODE.
   (b) If **root** is entered as NULL, then the function **root** is returned, else, the value of **root** is assigned to the variable data.
   (c) Using the while loop, the extreme left node in the tree is found and its address is returned.

Count Nodes and Leaves in a Tree

We obtain the number of nodes in the tree by traversing the tree using any traversal technique and incrementing the counter whenever a node is visited. The variable count is taken as a global variable and it is initialized to zero. In the given example, inorder traversal is used to visit each node. The C function to obtain the number of nodes in a tree is given in the following example.

Example:

```
static count = 0;
struct node
{
    int element;
    struct node *L;
    struct node *R;
 };
typedef struct node* NODE;
int count_node(NODE root)
{
```

```
    if(root!=NULL)
    {
      count_node(root->L);
      count++;
      count_node(root->R); }
     }
     return count;
}
```

In this example:

1. The variable **count** is a global variable and it is initialized to zero. It is also declared static so that it retains its value in consequent function calls.
2. A structure named **node** is created. It consists of two pointer variables named **L** and **R** that point to the left node and right node.
3. An object called NODE is created to access the structure element.
4. The function count_node is defined and has a parameter **root** of type NODE.
5. In the function count_node(),
   (a) If **root** is not NULL, then the number of left nodes is counted by recursively calling the function and by passing the address of the left sub-tree. The count variable is also incremented.
   (b) Then, the number of right nodes is recursively counted by the function by passing the address of the right sub-tree. The count variable is also incremented.

A leaf node is a node that has zero child nodes. To obtain the number of leaves in the tree, visit each node in the given tree. Whenever a leaf is found, update the count by one. The following example shows the function to count the leaves in a binary tree.

Example:

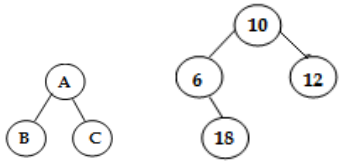
```
static count = 0;
struct node
{
   int element;
   struct node *L;
    struct node *R;
};
typedef struct node* NODE;
void count_leaf(NODE root)
{
   if(root!=NULL)
   {
      count_leaf(root->L);            // traverses recursively towards left
      if (root->L==NULL &&root->R==NULL)
     /* if a node has empty left and right child */
     count++;
     count_leaf(root->R);
    /* traverses recursively towards right */
     }
}
```

In this example:

1. The variable **count** is a global variable and it is initialized to zero. It is also declared static so that it retains its value in consecutive function calls.
2. A structure named **node** is created. It consists of two pointer variables named **L** and **R** that point to the left node and right node.
3. An object called NODE is created to access the structure element.
4. The function count_leaf is defined which accepts a parameter **root** of type NODE.
5. If **root** is not NULL, then the leaf nodes in the left sub-tree of the **root** is counted by the function. If a node has no left and right child, then the variable **count** is incremented.
6. Then, the leaf nodes in the right sub-tree of the root is counted by the function.

Task

Which of the following are binary search trees? Justify your answers.



Lab Exercise

Write a C program to implement a binary search tree.

13.2 Summary

- Search trees are data structures that support many dynamic-set operations such as searching, finding the minimum or maximum value, inserting, or deleting a value.
- In a binary search tree, for a given node **n**, each node to the left has a value lesser than **n** and each node to the right has a value greater than **n**.
- The time taken to perform operations on a binary search tree is directly proportional to the height of the tree.

13.3 Keywords

Degenerate Trees: Unbalanced trees.

Dynamic Sets: Dynamic sets are data structures that support operations such as search, insert, delete, minimum, maximum, successor and predecessor.

Inorder Successor: In binary tree, inorder successor of a node is the next node in inorder traversal of the binary tree. Inorder successor is null for the last node in inorder traversal.

Ordered List: An ordered list is a list that is maintained in some predefined order such as, alphabetical or numerical order.

13.4 Self Assessment

1. State whether the following statements are true or false:
   (a) A binary search tree has binary nodes.
   (b) The efficiency of a binary search tree depends on the height of the tree.
   (c) The smallest possible height of a tree with n nodes is log 2n.
   (d) In a binary search tree, a node with minimum value is found by traversing and obtaining the right most node of the tree.
   (e) A tree must be balanced to obtain the smallest height, where both the left and right sub-trees have the same number of nodes.
2. Fill in the blanks:
   (a) In searching, the node being search is called as a …….
   (b) We obtain the number of ……. in the tree by traversing the tree and incrementing the counter whenever a node is visited.
   (c) To insert a new element in an existing binary search tree, first we compare the value of the …… node with the current node value.
3. Select a suitable choice for every question:
   (a) The maximum level of a tree is referred as the ……. of the tree.
      (i) Height (ii) Node (iii) Leaf (iv) Root
   (b) In searching operation, the node to be searched is known as.
      (i) Key node (ii) Head node (iii) Start node (iv) Leaf node
   (c) Deletion of a leaf node involves setting left or right pointer of the parent node to?
      (i) Zero (ii) NULL (iii) Non zero (iv) One

13.5 Review Questions

1. "Binary search trees have more advantages when compared to other data structures". Justify.
2. "Performance of a binary search tree depends on its height". Explain.
3. Write a function that will search a given binary search tree for a specific key.

Answers: Self Assessment

1. (a) True (b) True (c) False (d) False (e) True
2. (a) Key node (b) Nodes (c) New
3. (a) Height (b) Key node (c) NULL

13.6 Further Readings

Books

Lipschutz.S. (2011). Data Structures with C. Delhi: Tata McGraw hill

Reddy.P. (1999). Data Structures Using C. Bangalore:Sri Nandi Publications

Online link

http://en.literateprograms.org/Binary_search_tree_(C)

http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/binarySearchTree.htm

Unit 14: Heaps

CONTENTS

Objectives

Introduction

14.1 Fundamentals of Heaps

14.2 Inserting into Heaps

14.3 Deleting the Root of a Heap

14.4 Heap Sort

14.5 Priority Queue Using Heaps

14.6 Summary

14.7 Keywords

14.8 Self Assessment

14.9 Review Questions

14.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Provide an introduction to heaps
- Explain insertion operation on heap
- Explain deletion of root of a heap
- Discuss heap sort
- Understand priority queue using heaps

Introduction

The heap data structure is a complete binary tree where each node of the tree has an orderly relationship with its successors. Binary search trees are totally ordered, but the heap data structure is only partially ordered. It is suitable for inserting and deleting minimum value operations.

Heap is an array object that is considered as a complete binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled at all levels except possibly the lowest, which is filled from the left upwards to a point. Heap data structures are suitable for implementing priority queues.

The heap serves as a foundation of a theoretically important sorting algorithm called heapsort, which we will discuss after defining the heap.

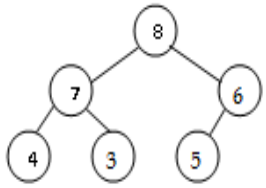
14.1 Fundamentals of Heaps

A heap can be defined as binary trees with keys assigned to its nodes (one key per node). The two types of heaps are:

1. Max heaps
2. Min heaps

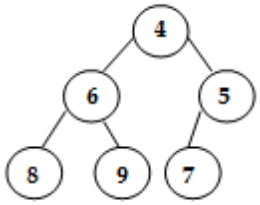
Max Heaps - A max heap is defined as a heap in which the key value in each node is greater than or equal to the keys at its children, i.e., key (parent) ≥ key (child). The figure 14.1 depicts a max heap. Here, the root node 8 is greater than its child nodes 7 and 6. Similarly, 7 and 6 are greater than their child nodes.

Figure14.1: Max Heap



Min Heaps - A min heap is a heap where the key value in each node is lesser than or equal to the keys of its children i.e., key (parent) ≤ key (child).The figure 14.2 represents a min heap. Here, the root node 4 is lesser than its child nodes 6 and 5. Similarly, 6 and 5 are lesser than their child nodes.

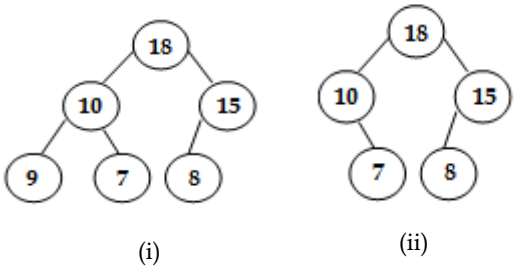
Figure 14.2: Min Heap



The Shape Requirement of Heaps

The shape requirement of heaps defines that, in a binary tree all its levels must be full except possibly the last level, where only some rightmost leaves may be missing. This requirement is valid for both max and min heaps. In figure 14.3, the first tree (i) is a heap, but the second tree (ii) is not, as the tree's shape requirement is violated. The tree in the figure 14.3 (i) satisfies the shape requirement, whereas the tree in the figure 14.3(ii) does not. The node 10 at level 1 of the figure 14.3 (ii) does not have a left child and therefore violates the shape requirement.

Figure 14.3: Illustration of Heaps



Notes

1. Heap data structure can be an n-ary tree.
2. Heap is not essentially sorted.

Task

Draw a max heap and a min heap for the sequence (20, 13, 16, 8, 15, 4, 9, 3, 12, 18)

Architectural Approach of Heaps and Algorithms

The two principal ways to construct a heap are:

1. Bottom-up heap construction algorithm
2. Top-down heap construction algorithm

Let us now discuss the bottom-up heap construction.

Bottom-up Heap Construction

Bottom-up heap construction initializes the complete binary tree with **n** nodes by placing keys in the given order and then "heapifies" the tree as follows. Starting with the last parental node and ending with the root, the algorithm checks whether the parental dominance holds over the key at this node. If the parental dominance condition is not met, the algorithm exchanges the node's key with the larger key of its children and checks for parental dominance of the key in its new position. This process continues until the parental dominance for the key is satisfied. After the "heapification" of the sub-tree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.

If parental dominance does not hold over the key at a node, the algorithm exchanges the node's key **K** with the larger key of its children and checks whether the parental dominance holds for **K** in its new position (Refer figures 14.4 and 14.5). In figure 14.4, the node with value 7 is lesser than its child node that has the value 8. Hence these two nodes are interchanged.

Figure 14.4: Checking for Parental Dominance

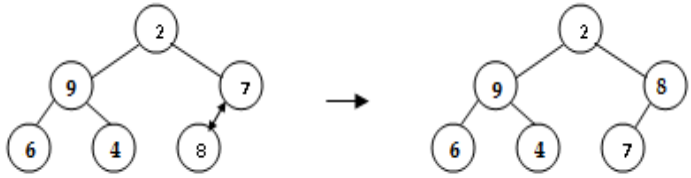
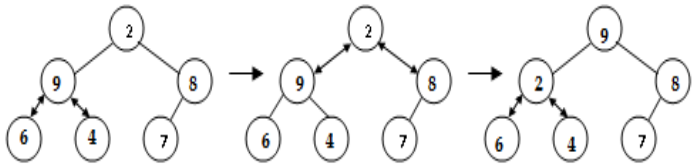


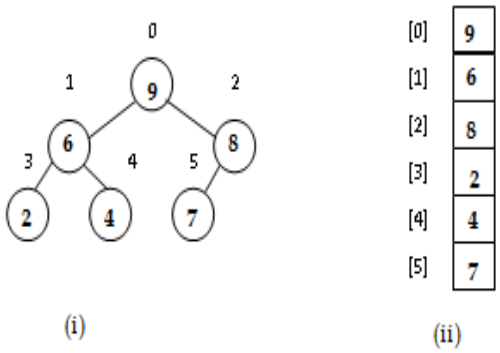
Figure 14.5 shows how checking for parental dominance continues.

Figure 14.5: Continue Checking for Parental Dominance



This process continues until the parental dominance requirement for **K** is satisfied. After completing the "heapification" of the sub-tree rooted at the current parental node, the algorithm continues to do the same for the immediate predecessor of the node as shown in figure 14.5. In figure 14.5, the node with value 9 is greater than its child nodes that have the values 6 and 4 and thereby holds the parental dominance. Then, this node with value 9 is compared with the root value 2, which is lesser than 9 and hence these two nodes are interchanged. The node with value 2 is lesser than its child nodes 6 and 4 and thus, it is interchanged with the node having larger value 6. The algorithm stops after this is done for the tree's root to give the final heap in figure 14.6(i). The numbers above the nodes in the tree indicate their position in the array which is shown in the figure 14.6(ii).

Figure 14.6: Heap and Array Representation



As the value of a node's key does not change during the process of shifting the node down the tree, it need not get involved in intermediate swaps. The empty nodes are swapped with the larger keys until a final position is reached where it accepts the deleted value again.

Let us now study the algorithm for bottom-up heap construction.

Algorithm: Heap Bottom-up (H [1...n])

//Constructs a heap from the elements of a given array

// by the bottom-up algorithm

//Input: An array H[1..n] of orderable items

//Output: A heap H[1..n]

```
fori ←n/2          down to 1 do
     k←i; v←H[k]
heap←false
while not heap and 2 * k ≤ n do
    j←2 * k
if j <n //there are two children
   if H[ j ]<H[ j + 1] j ←j + 1
   if v ≥ H[j ]
      heap←true
   else H[k]←H[j ]; k←j
     H[k]←v
```

Let us now trace the bottom-up heap construction algorithm.

Algorithm Tracing of Bottom-up Heap Construction

n=5

Algorithm: Heap bottom-up (H [1...5])

```
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..5] of orderable items
//Output: A heap H[1..5]
for i ←5/2=1        down to 1 do
   k←1; v←H[1]
heap←false
while not heap and 2 * 1 ≤ 5 do
   j ←2 * 1
if j <5 //there are five children
    if H[ 2 ]<H[ 2 + 1] 2 ←2 + 1
      if v ≥ H[2 ]
         heap←true
     else H[1]←H[2 ]; 1←2
   H[1]←v
```

Task

Construct a heap for the list 2, 9, 7, 6, 2, 8, 5 using the bottom-up construction algorithm.

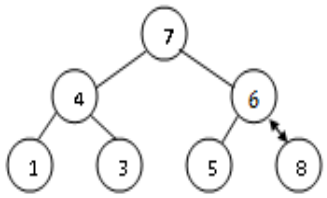
Top-down Heap Construction Algorithm

The top-down heap construction algorithm is less efficient and it constructs a heap by successive insertions of a new key into a previously constructed heap. The insertion of a new key into a heap is discussed in the section 14.2.

14.2 Inserting into Heaps

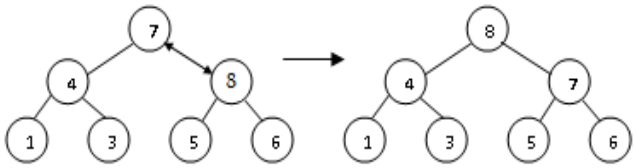
To insert a new key into a heap, add a new node with key **K** after the last leaf of the existing heap. Then, shift **K** up to its suitable place in the new heap. Consider inserting value 8 into the heap shown in the figure 14.7.

Figure 14.7: Inserting 8 into Heap



Compare 8 with its parent key. Stop if the parent key is greater than or equal to 8. Else, swap these two keys and compare 8 with its new parent (Refer to figure 14.8). This swapping continues until 8 is not greater than its last parent or it reaches the root. In this algorithm too, we can shift up an empty node until it reaches its proper position, where it acquires the value 8.

Figure 14.8: Check for Parental Dominance until Tree is Balanced

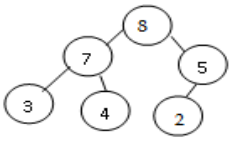


This insertion operation does not require more key comparisons than the heap's height. Since the height of a heap with **n** nodes is about log2n, the time efficiency of insertion is in O(log n).

14.3 Deleting the Root of a Heap

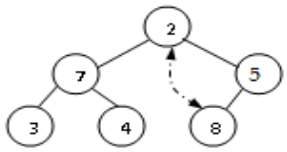
The following steps show the method to delete the root key from a heap in the figure 14.9.

Figure 14.9: Sample Heap



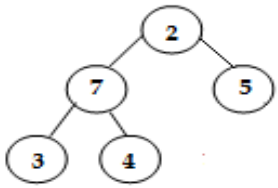
Step 1: Exchange the root's key with the last key **K** of the heaps as shown in the figure 14.10.

Figure 14.10: Exchanging the Root Key with the Last Key



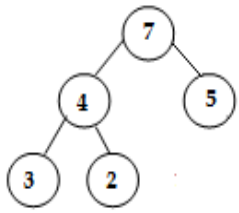
Step 2: Decrease the heap's size by 1 (Refer figure 14.11).

Figure 14.11: Delete the Key Having the Original Root Key



Step 3: "Heapify" the smaller tree by shifting **K** down the tree as we did in the bottom-up heap construction algorithm. That is, verify the parental dominance for **K** and if it holds, we complete the process (Refer figure 14.12). If not, swap **K** with the largest of its children and repeat this operation until the parental dominance condition holds for **K** in its new position.

Figure 14.12: Heapified Tree



We can determine the efficiency of deletion by the number of key comparisons required to "heapify" the tree after the swap is done, and the size of the tree is decreased by 1. Since it does not need more key comparisons than twice the heap's height, the time efficiency of deletion is in O(log n).

Task

Name some applications of heap.

14.4 Heap Sort

A heap is used to implement heapsort. Heapsort uses the data structure max-heap which is a complete binary tree where the element at any node is greater than its children. Heapsort is a comparison-based

sorting algorithm which has a worst-case O(n log n) runtime. This is a two-stage algorithm which is as follows:

Stage 1: (Heap construction): Construct a heap for a given array.

Stage 2: (Maximum deletions): Apply the root-deletion operation n−1 times to the remaining heap.

The pseudo code for the algorithm is shown below.

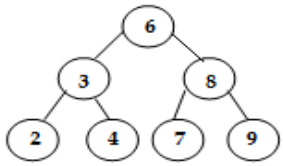
```
Heapify(A, n){                                  // heapify converts initial array into heap
   l <- left(n)
   r <- right(n)
   if l <= heapsize[A] and A[l] > A[n]          // the root is compared with its two children
     then largest <- l
   else if largest <- n
     if r <= heapsize[A] and A[r] > A[largest]
       then largest <- r
     if largest != n
       then swap A[n] <-> A[largest]              // the larger child is swapped with the root
     Heapify(A, largest)                        // heapify algorithm is applied to the  larger node
}
Buildheap(A){
   heapsize[A] <- length[A]
   for n<- |length[A]/2| down to 1
   do Heapify(A, n)
 }
Heapsort(A){
   Buildheap(A)
   for n<- length[A] downto 2
      do swap A[1] <-> A[n]
   heapsize[A] <- heapsize[A] - 1
   Heapify(A, 1)
 }
```

Heap sort first converts the initial array into a heap. The heapsort algorithm uses 'heapify' method to complete the task. The heapify algorithm, as given in the above code, receives a binary tree as input and converts it to a heap. Then, the root is compared with its two children, and the larger child is swapped with it. This may result in one of the left or right sub-trees losing the heap property. As a result, the heapify algorithm is recursively applied to the suitable sub-tree rooted at the node whose value was swapped with the root. This process continues until a leaf node is reached, or until the heap property is satisfied in the sub-tree.

Let us discuss the implementation of heapsort using the following example.

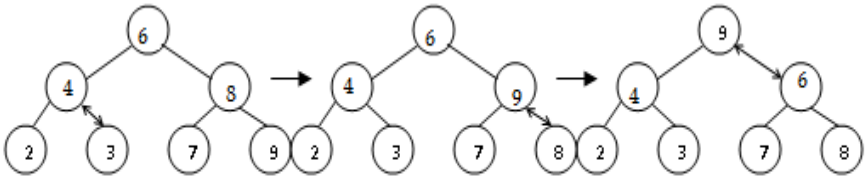
Example: Consider the following numbers 6,3,8,2,4,7,9. The tree representation of the set of numbers can be seen in the figure 14.13.

Figure 14.13 Tree for the List



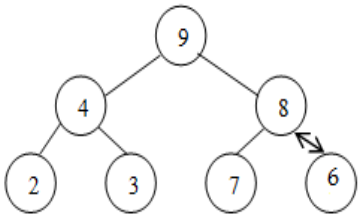
Let us now perform the first stage of heapification on the tree to make it balanced as shown in the figure 14.14.

Figure 14.14: Process of Heapification



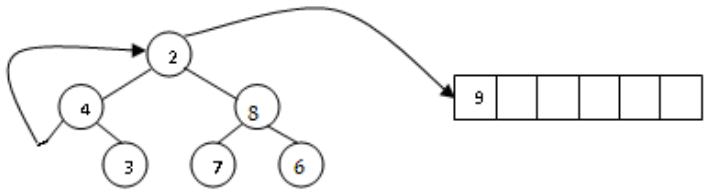
We have obtained the heapified tree now (Refer figure 14.15) and we perform stage two which includes the deletion of the nodes.

Figure 14.15: Heapified Tree



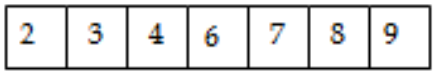
To perform the node deletion, we have to push the largest value on the top of the heap into an array and replace the value by the extreme left element in the tree as represented in figure 14.16.

Figure 14.16 Node Deletion



Now we obtain a tree which is not balanced. Therefore, we repeat the process of heapification to end with the largest element in the top node of the tree. This element is now pushed again into the array, and the extreme left bottom element replaces it. As we repeat this process, we finally obtain the sorted array as shown in figure 14.17.

Figure 14.17: Sorted Array



Notes

The time efficiency of heapsort is O(n log n) in both the worst and average cases.

14.5 Priority Queue Using Heap

A priority queue is a queue with items having an orderable characteristic called priority. The objects having the highest priority are always removed first from the priority queues.

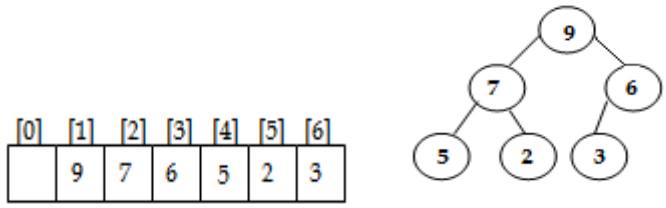
A priority queue can be obtained by creating a heap. First call a function that creates an ascending heap. After creating the heap, delete the root node and call a function to recreate the heap for the remaining elements. This method helps in implementing an ascending priority queue. In the same way, we can implement a descending priority queue.

The standard approach to implement priority queues using a heap is to use an array starting at position 1 (instead of 0), where each item in the array relates to one node in the heap:

1. Array[1] holds the root of the heap
2. Array[2] holds the left child
3. Array[3] holds the right child

Let us discuss an example. Figure 14.18 represents both the conceptual heap (the binary tree), and its array representation. As shown, the root 9 is in array [1]. Its left child 7 is in array [2] and right child is in array[3]. In general, if a node is in array[m], then its left child is in array [m*2], right child in array [m*2 + 1] and its parent is in array [m/2].

Figure 14.18: Array Representation of Heap



The shape property of the heap guarantees that there are no empty spaces in the array.

14.6 Summary

- The heap data structure is a complete binary tree where each node of the tree relates to an element of the array that stores the value in the node.
- The two principal ways to construct a heap are by using the bottom-up heap construction algorithm and the top-down heap construction algorithm
- A heap is used to implement heapsort. Heapsort is a comparison-based sorting algorithm which has a worst-case of O(n log n) runtime.
- A priority queue is a queue with items having an orderable characteristic called priority. The objects having the highest priority are always removed first from the priority queues.
- Priority queue can be attained by creating a heap.

14.7 Keywords

Ascending Heap: It is a complete binary tree in which the value of each node is greater than or equal to the value of its parent.

Heapify: Heapify is a procedure for manipulating heap data structures.

N-ary Tree: An n-ary tree is either an empty tree, or a non-empty set of nodes which consists of a root and exactly N sub-trees. The degree of each node of an N-ary tree is either zero or N.

Parental Dominance: The key at each node is greater than or equal to the keys of the children and this is fulfilled automatically for leaf nodes.

14.8 Self Assessment

1. State whether the following statements are true or false:

   (a) Heaps are data structures that are suitable for implementing priority queues.

   (b) The top-down heap construction algorithm is more efficient and it constructs a heap by successive deletions of a new key into a previously constructed heap.

   (c) To insert a new key into a heap, a new node with key **K** is added after the last leaf of the existing heap and **K** is shifted up to its suitable place in the new heap.

2. Fill in the blanks:

   (a) …… uses the data structure max-heap which is a complete binary tree where the element at any node is greater than its children.

   (b) …… construction initializes the complete binary tree with n nodes by placing keys in the given order and then "heapifies" the tree.

   (c) The heapsort algorithm uses …… method to convert the initial array into a heap.

3. Select a suitable choice for every question:

   (a) A…… is a set of items with an orderable characteristic called an item's priority.

      (i) Priority queue

      (ii) Heap

      (iii) Data structure

      (iv) BST

   (b) Heapsort is a comparison-based sorting algorithm which has a worst-case …… runtime.

      (i) O(log n

      (ii) O(log 2n)

      (iii) O(n log n)

      (iv) O(n log 2n)

   (c) Priority queue can be attained by creating a ……

      (i) Queue

      (ii) Tree

      (iii) BST

      (iv) Heap

   (d) The time efficiency of insertion in heap is ……

      (i) O(n log n)

      (ii) O(log 2n)

      (iii) O(log n)

      (iv) O(n log 2n)

14.9 Review Questions

1. "The bottom-up heap construction algorithm checks whether the parental dominance holds over the key at a node starting with the last parental node and ending with the root." Discuss.

2. "A heap can be implemented as an array by recording its elements in top-down left-to-right manner". Describe in detail.

3. "Binary search property is different from heap property". Justify.

4. "The heap data structure can be used for an efficient implementation of a priority queue". Explain.

5. Represent the max heap and min heap for the data 3, 8, 20, 28, 42, 54.

Answers: Self Assessment

1. (a) True (b) False (c) True
2. (a) Heapsort (b) Bottom-up heap (c) heapify
3. (a)Heap (b) O(n log n) (c) Heap (d) O(log n)

14.10 Further Readings

Books

Lipschutz.S. (2011). Data Structures with C. Delhi: Tata McGraw hill

Reddy.P. (1999). Data Structures Using C. Bangalore:Sri Nandi Publications

Online link

http://orion.lcg.ufrj.br/Dr.Dobbs/books/book3/chap6.htm

http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html