

# **DATA STRUCTURES & ALGORITHMS**

**M.C.A.  
I Semester**



**Bharathidasan University**  
**Centre for Distance and Online Education**

**Chairman:**

Dr. M. Selvam  
Vice-Chancellor  
Bharathidasan University  
Tiruchirappalli-620 024  
Tamil Nadu

**Co-Chairman:**

Dr. G. Gopinath  
Registrar  
Bharathidasan University  
Tiruchirappalli-620 024  
Tamil Nadu

**Course Co-Ordinator:**

Dr. A. Edward William Benjamin  
Director-Centre for Distance and Online Education  
Bharathidasan University  
Tiruchirappalli-620 024  
Tamil Nadu

The Syllabus is Revised from 2021-22 onwards

**Author:**

**Dr. S. Chellammal, Asst Professor, Dept of Computer Science, CDOE,  
Bharathidasan University, Trichy**

"The copyright shall be vested with Bharathidasan University"

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Publisher.

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Publisher, its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.



Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE  
PVT LTDE-28, Sector-8, Noida -  
201301 (UP)

Phone: 0120-4078900 • Fax: 0120-4078999

Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44

Website: [www.vikaspublishing.com](http://www.vikaspublishing.com) • Email: [helpline@vikaspublishing.com](mailto:helpline@vikaspublishing.com)

---

# SYLLABI-BOOK MAPPING TABLE

## Data Structures & Algorithms

---

Syllabi	Mapping in Book
<b>BLOCK I: INTRODUCTION</b> <b>UNIT 1: Introduction to Data Structure:</b> Types of Data Structure , Primitive Data Types <b>Algorithms:</b> Time and Space Complexity of Algorithms	<b>Unit 1:</b> Introduction to Data Structure (Pages 1-20)
<b>UNIT 2: Arrays:</b> Array Initialization, Definition of Array, Characteristic of Array, One-dimensional Array, Two-dimensional Array and Multi Dimensional Array	<b>Unit 2:</b> Arrays (Pages 21-36)
<b>BLOCK II: LINEAR DATA STRUCTURE</b> <b>UNIT 3: Stack:</b> Stack Related Terms, Operations on a Stack	<b>Unit 3:</b> Stack (Pages 37-45)
<b>UNIT 4: Representation of Stack:</b> Implementation of a Stack - Application of Stack. Expression Evaluation Polish Notation.	<b>Unit 4:</b> Representation of Stack (Pages 46-67)
<b>UNIT 5: Queues:</b> Operations on Queue Circular Queue, Representation of Queues, Application of Queues	<b>Unit 5:</b> Queues (Pages 68-89)
<b>UNIT 6: List:</b> Merging Lists, Linked List, Single Linked List, Double Linked List, Header Linked List	<b>Unit 6:</b> Lists (Pages 90-98)
<b>UNIT 7: Operation on Linked List:</b> Insertion and Deletion of Linked List	<b>Unit 7:</b> Operation on Linked Lists (Pages 99-118)
<b>UNIT 8: Traversal:</b> Traversing a Linked List, Representation of Linked List.	<b>Unit 8:</b> Traversal (Pages 119-124)
<b>BLOCK III: NON-LINEAR DATA STRUCTURE</b> <b>UNIT 9: Trees:</b> Binary Trees, Types of Binary Trees, Binary Tree Representation	<b>Unit 9:</b> Trees (Pages 125-133)
<b>UNIT 10:</b> Binary Tree Operations / Applications : Traversing Binary Trees, Binary Search Tree	<b>Unit 10:</b> Binary Tree Operations/ Applications (Pages 134-156)
<b>UNIT 11: Operations on Binary Tree:</b> Insertion and Deletion Operations, Hashing Techniques.	<b>Unit 11:</b> Operations on Binary Tree (Pages 157-175)

---

## **Program Outcomes:**

1. **Technical Knowledge and Understanding:** Graduates of an MCA program will demonstrate a strong foundation in computer science and information technology, including programming languages, algorithms, data structures, and software development methodologies.
2. **Software Development and Implementation:** Graduates will have the skills to develop, implement, and test software applications using various programming languages, frameworks, and development tools.
3. **Database Management:** Graduates will possess knowledge of database concepts, database design, and database management systems to efficiently store, retrieve, and manipulate data.
4. **System and Network Administration:** Graduates will have an understanding of system administration, network management, and security protocols to effectively manage and maintain computer systems and networks.
5. **Information Systems Management:** Graduates will possess knowledge of information systems, IT project management, and organizational processes to effectively manage and align technology solutions with business goals.
6. **Communication and Collaboration:** Graduates will possess effective communication skills, both oral and written, and the ability to collaborate with multidisciplinary teams to solve complex problems and present technical information.

## **Program-Specific Outcomes:**

1. **Software Engineering:** On completion of the program, the Graduates will be specializing in Software Engineering should have advanced knowledge and skills in software development methodologies, software testing, software quality assurance, and software project management.
2. **Web Development:** On completion of the program, the Graduates will be specializing in Web Development should possess expertise in web programming languages, web design principles, web development frameworks, and the ability to create dynamic and interactive web applications.
3. **Data Science:** On completion of the program, the Graduates will be specializing in Data Science should have proficiency in data analysis, data mining, machine learning algorithms, and statistical modeling to extract insights and solve complex data-related problems.
4. **Mobile Application Development:** On completion of the program, the Graduates will be specializing in Mobile Application Development should have expertise in mobile programming languages, mobile application design, mobile user interface development, and the ability to create innovative mobile applications.
5. **Artificial Intelligence:** On completion of the program, the Graduates will be specializing in Artificial Intelligence should possess knowledge and skills in AI algorithms, natural language processing, machine learning, and the ability to develop AI-powered applications.
6. **Enterprise Resource Planning (ERP) Systems:** On completion of the program, the Graduates will be specializing in ERP Systems should have knowledge of ERP concepts, ERP implementation methodologies, ERP modules, and the ability to manage and integrate ERP systems in organizations.

---

# CONTENTS

---

## INTRODUCTION

### BLOCK I: INTRODUCTION

#### UNIT 1 INTRODUCTION TO DATA STRUCTURE 1-20

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Primitive and Composite Data Types
- 1.3 Abstract Data Types
- 1.4 Data Structures
  - 1.4.1 Linear Data Structures
  - 1.4.2 Non-Linear Data Structures
- 1.5 Operations on Data Structures
- 1.6 Algorithms
- 1.7 Algorithm Design Techniques
  - 1.7.1 Brute Force Algorithm
  - 1.7.2 Divide and Conquer
  - 1.7.3 Dynamic Programming
  - 1.7.4 Greedy Algorithm
- 1.8 Time and Space Complexity of Algorithms
- 1.9 Big O Notation
- 1.10 Recurrences
- 1.11 Answers to Check Your Progress Questions
- 1.12 Summary
- 1.13 Key Words
- 1.14 Self Assessment Questions and Exercises
- 1.15 Further Readings
- 1.16 Learning Outcomes

#### UNIT 2 ARRAYS 21-36

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Arrays (Ordered Lists)
  - 2.2.1 Single-Dimensional Arrays
  - 2.2.2 Multi-Dimensional Arrays
- 2.3 Representation of an Array
  - 2.3.1 Memory Representation of a Single-Dimensional Array
  - 2.3.2 Memory Representation of a Two-Dimensional Array
- 2.4 Answers to Check Your Progress Questions
- 2.5 Summary
- 2.6 Key Words
- 2.7 Self Assessment Questions and Exercises
- 2.8 Further Readings
- 2.9 Learning Outcomes

## **BLOCK II: LINEAR DATA STRUCTURE**

### **UNIT 3     STACK** **37-45**

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Stack Related Terms and Operations on Stack
- 3.3 Answers to Check Your Progress Questions
- 3.4 Summary
- 3.5 Key Words
- 3.6 Self Assessment Questions and Exercises
- 3.7 Further Readings
- 3.8 Learning Outcomes

### **UNIT 4     REPRESENTATION OF STACK** **46-67**

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Application and Implementation of Stack
  - 4.2.1 Converting Infix Notation to Postfix and Prefix or Polish Notations
- 4.3 Answers to Check Your Progress Questions
- 4.4 Summary
- 4.5 Key Words
- 4.6 Self Assessment Questions and Exercises
- 4.7 Further Readings
- 4.8 Learning Outcomes

### **UNIT 5     QUEUES** **68-89**

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Queues
- 5.3 Representation of Queues
- 5.4 Circular Queue and Deque
- 5.5 Priority Queue
- 5.6 Applications of Queues
- 5.7 Answers to Check Your Progress Questions
- 5.8 Summary
- 5.9 Key Words
- 5.10 Self Assessment Questions and Exercises
- 5.11 Further Readings
- 5.12 Learning Outcomes

### **UNIT 6     LISTS** **90-98**

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Linked List
- 6.3 Singly-Linked Lists
- 6.4 Circular Linked Lists

- 6.5 Doubly-Linked Lists
- 6.6 Merging Lists and Header Linked List
- 6.7 Answers to Check Your Progress Questions
- 6.8 Summary
- 6.9 Key Words
- 6.10 Self Assessment Questions and Exercises
- 6.11 Further Readings
- 6.12 Learning Outcomes

## **UNIT 7      OPERATION ON LINKED LISTS** **99-118**

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Insertion and Deletion Operations in Linked List
- 7.3 Insertion and Deletion in Circular Linked List
- 7.4 Insertion and Deletion in Doubly-Linked Lists
- 7.5 Answers to Check Your Progress Questions
- 7.6 Summary
- 7.7 Key Words
- 7.8 Self Assessment Questions and Exercises
- 7.9 Further Readings
- 7.10 Learning Outcomes

## **UNIT 8      TRAVERSAL** **119-124**

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Traversing Linked Lists
- 8.3 Representation of Linked List
- 8.4 Answers to Check Your Progress Questions
- 8.5 Summary
- 8.6 Key Words
- 8.7 Self Assessment Questions and Exercises
- 8.8 Further Readings
- 8.9 Learning Outcomes

## **BLOCK III: NON-LINEAR DATA STRUCTURE**

## **UNIT 9      TREES** **125-133**

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Binary Trees
  - 9.2.1 Forms/Types of Binary Tree
  - 9.2.2 Binary Tree Representations
- 9.3 Answers to Check Your Progress Questions
- 9.4 Summary
- 9.5 Key Words
- 9.6 Self Assessment Questions and Exercises
- 9.7 Further Readings
- 9.8 Learning Outcomes

## **UNIT 10 BINARY TREE OPERATIONS/APPLICATIONS**

**134-156**

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Traversing Binary trees
- 10.3 Binary Search Tree
- 10.4 Traversing a Binary Search Tree
- 10.5 Answers to Check Your Progress Questions
- 10.6 Summary
- 10.7 Key Words
- 10.8 Self Assessment Questions and Exercises
- 10.9 Further Readings
- 10.10 Learning Outcomes

## **UNIT 11 OPERATIONS ON BINARY TREE**

**157-175**

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Insertion and Deletion Operations
- 11.3 Hashing Techniques
- 11.4 Answers to Check Your Progress Questions
- 11.5 Summary
- 11.6 Key Words
- 11.7 Self Assessment Questions and Exercises
- 11.8 Further Readings
- 11.9 Learning Outcomes

## **BLOCK IV: SEARCHING TECHNIQUES**

### **UNIT 12 SEARCHING**

**176-191**

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Searching
  - 12.2.1 Linear Search
  - 12.2.2 Binary Search
- 12.3 Answers to Check Your Progress Questions
- 12.4 Summary
- 12.5 Key Words
- 12.6 Self Assessment Questions and Exercises
- 12.7 Further Readings
- 12.8 Learning Outcomes

## **BLOCK V: SORTING TECHNIQUES**

### **UNIT 13 SORTING**

**192-205**

- 13.0 Introduction
- 13.1 Objectives
- 13.2 Definition
- 13.3 Bubble Sort



- 13.4 Insertion Sort
- 13.5 Radix Sort
- 13.6 Answers to Check Your Progress Questions
- 13.7 Summary
- 13.8 Key Words
- 13.9 Self Assessment Questions and Exercises
- 13.10 Further Readings
- 13.11 Learning Outcomes

## **UNIT 14 OTHER SORTING TECHNIQUES**

**206-218**

- 14.0 Introduction
- 14.1 Objectives
- 14.2 Selection Sort
- 14.3 Quick Sort
- 14.4 Tree Sort
- 14.5 Answers to Check Your Progress Questions
- 14.6 Summary
- 14.7 Key Words
- 14.8 Self Assessment Questions and Exercises
- 14.9 Further Readings
- 14.10 Learning Outcomes

---

## INTRODUCTION

---

### NOTES

A data structure is defined as a set of data elements that represents operations, such as insertion, deletion, modification and traversal of the values present in the data elements. Data elements are the data entries that are stored in the memory for organizing and storing data in an ordered and controlled way. The commonly used data structures in various programming languages, such as C, are arrays, linked lists, stacks and trees. Data structures are of two types, linear and non-linear.

This book, *Data Structures & Algorithms*, introduces you to the basic concepts of data structures. It explains arrays, which can be used to store lists of elements and discusses stacks – a linear data structure, which includes memory representation and the various applications of stacks. Queues, including their representation and trees; linked list and the various types of linked lists; trees including the binary tree, binary search tree and threaded binary tree, along with the various applications of trees; searching and sorting data; and hashing are some of the other topics explained in this book.

The book follows the self-instructional mode wherein each Unit begins with an Introduction followed by an outline of the Objectives. The detailed content is then presented in a simple and structured format interspersed with Check Your Progress questions for testing the understanding of the students. A Summary, a list of Key Words and a set of Self Assessment Questions and Exercises is also provided at the end of each unit for effective recapitulation.

---

## BLOCK - I

### INTRODUCTION

---

*Introduction to Data  
Structure*

## NOTES

---

# UNIT 1 INTRODUCTION TO DATA STRUCTURE

---

### Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Primitive and Composite Data Types
- 1.3 Abstract Data Types
- 1.4 Data Structures
  - 1.4.1 Linear Data Structures
  - 1.4.2 Non-Linear Data Structures
- 1.5 Operations on Data Structures
- 1.6 Algorithms
- 1.7 Algorithm Design Techniques
  - 1.7.1 Brute Force Algorithm
  - 1.7.2 Divide and Conquer
  - 1.7.3 Dynamic Programming
  - 1.7.4 Greedy Algorithm
- 1.8 Time and Space Complexity of Algorithms
- 1.9 Big O Notation
- 1.10 Recurrences
- 1.11 Answers to Check Your Progress Questions
- 1.12 Summary
- 1.13 Key Words
- 1.14 Self Assessment Questions and Exercises
- 1.15 Further Readings
- 1.16 Learning Outcomes

---

## 1.0 INTRODUCTION

---

A data structure is a specialized format for organizing and storing data. Some general data structure types include the array, file, record, table, tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In this unit, you will learn about data types, algorithms and complexities associated with them.

---

## 1.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss Primitive data types

## NOTES

- Understand composite data types
- Explain Linear and non-linear data structure
- Analyze abstract data types
- Understand algorithms and their design techniques
- Understand time and space complexity of algorithms

---

### 1.2 PRIMITIVE AND COMPOSITE DATA TYPES

---

Each programming language provides various data types and each data type is represented differently within the computer's memory. The memory requirement of a data type determines the permissible range of values for that data type. The data types can be classified into several categories, including primitive data types and composite data types.

The data types provided by a programming language are known as *primitive data types* or *in-built data types*. Different programming languages provide different set of primitive data types. For example, the primitive data types in the C language are int (for integers), char (for characters), and float (for floating point numbers). The data types that are derived from primitive data types of the programming language are known as **composite data types** or **derived data types**. Various data types in the C language include arrays, functions, and pointers.

In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements. For example, the various user-defined data types as provided by C are structures, unions, and enumerations.

---

### 1.3 ABSTRACT DATA TYPES

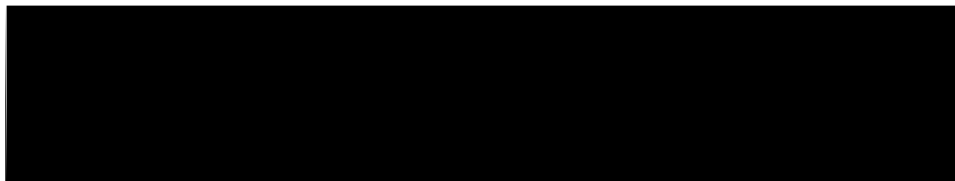
---

Generally, handling small problems is much easier than handling comparatively larger problems. The same rule is applicable to programming also. Therefore, a large program is decomposed into small logical units or modules, each of which does a well-defined sub task of the whole program. The size of each module is kept as small as possible and if required, other modules are invoked from it. This modular design provides several advantages. First, several people can be employed to work on a single program, which will increase the speed of completing the given task. Second, a well-designed modular program has modules independent of each others, implementation, which will make the program easily modifiable.

An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented. The data type integer is an example the abstract data type. We frequently perform

operations on integers that are associated with them like addition, subtraction, division, multiplication, modulus, etc. However, we do not know how these operations are actually performed on integers. We only know the syntax of how to perform these operations in some programming language. For example, C language defines  $+$ ,  $-$ ,  $/$ ,  $*$ ,  $\%$  to perform some basic arithmetic operations on integers.

The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation. If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it. The data structures, namely, *linked lists*, *stacks*, and *queues* are some examples of ADTs.



## NOTES

---

## 1.4 DATA STRUCTURES

---

The logical or mathematical model used to organize the data in main memory is called a *data structure*. Various data structures are available, each having certain special features. These features should be kept in mind while choosing a data structure for a particular situation. Generally, the choice of a data structure depends on its simplicity and effectiveness in processing of data. In addition, we also consider how well it represents the actual relationship of the data in the real world. Data structures are divided into two categories, namely, *linear data structure* and *non-linear data structure*.

### 1.4.1 Linear Data Structures

A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor. An array is the simplest linear data structure. Various other linear data structures are linked into lists, stacks, and queues.

#### Arrays

A finite collection of homogenous elements is termed as an array. Here, the word 'homogenous' indicates that the data type of all the elements in the collection should be same, that is, int or char or float or any other built-in or user-defined data type. However, an array cannot have elements of two or more data types together.

## NOTES

The elements of an array are always stored in a contiguous memory locations irrespective of the array size. The elements of an array can be referred to by using one or more *indices* or *subscripts*. An index or a subscript is a positive integer value which indicates the position of a particular element in the array. If the number of subscripts required to access any particular element of an array is one, then it is called a, *single-dimensional array*. Otherwise, it is a *multi-dimensional array*. A multi-dimensional array can be a *two-dimensional array* or even more.

Consider a single-dimensional array  $Arr$  with size  $n$ , where  $n$  is the maximum number of elements that  $Arr$  can store. Mathematically, the elements of  $Arr$  are denoted as  $Arr_1, Arr_2, Arr_3, \dots, Arr_n$ . In different programming languages, array elements are denoted by different notations such as by parenthesis notation or by bracket notation. Table 1.1 shows the notation of elements of a single-dimensional array  $Arr$  with size  $n$  in different programming languages as follows:

**Table 1.1** Different Notations of a Single-dimensional Array

S. Number	Notation	Programming Language(s)
1	$Arr(1), Arr(2), Arr(3), \dots, Arr(n)$	BASIC and FORTRAN
2	$Arr[1], Arr[2], Arr[3], \dots, Arr[n]$	PASCAL
3	$Arr[0], Arr[1], Arr[2], \dots, Arr[n-1]$	C, C++ and Java

Note that in the languages, BASIC, PASCAL, and FORTRAN, the smallest subscript value is 1 and the largest subscript value is  $n$ . On the other hand, in languages like C, C++ and Java, the smallest subscript value is 0 and the largest subscript value is  $n-1$ . In general, the smallest subscript value that is used to access an array element is the lower bound ( $L_b$ ) and the largest subscript value that is used is upper bound ( $U_b$ ).

In two-dimensional arrays, the elements can be viewed as arranged in the form of rows and columns (matrix form). To access an element of a two-dimensional array, two subscripts are used—first one represents the row number and second one represents the column number. For example, consider a two-dimensional array  $Arr$  with size  $m \times n$ , where  $m$  and  $n$  represent the number of rows and columns, respectively. Mathematically, the array  $Arr$  is denoted as  $Arr_{ij}$ , where  $i$  and  $j$  indicate the row numbers and the column number with  $i \leq m$  and  $j \leq n$ . Table 1.2 shows the notation of elements of a two-dimensional array  $Arr$  in different programming languages as follows:

**Table 1.2** Different Notations of Two-dimensional Array

S. Number	Notation	Programming Language(s)
1	Arr(i, j) with $0 < i \leq m$ and $0 < j \leq n$	BASIC and FORTRAN
2	Arr[i, j] with $0 < i \leq m$ and $0 < j \leq n$	PASCAL
3	Arr[i][j] with $0 \leq i < m$ and $0 \leq j < n$	C, C++ and Java

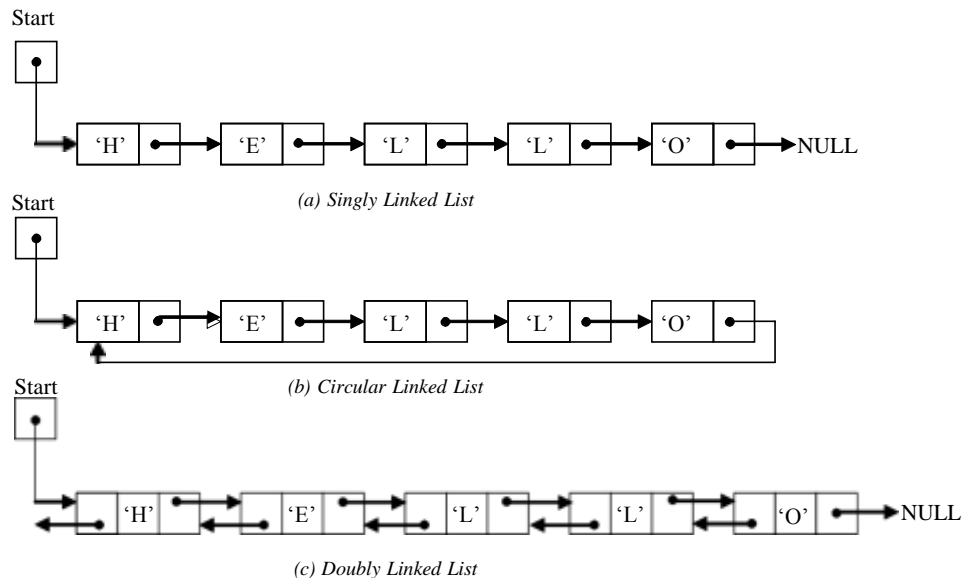
## NOTES

### Linked lists

Another commonly used linear data structure is a linked list. A linked list is a linear collection of similar data elements, called nodes, with each node containing some data and pointer(s) pointing to other node(s) in the list. Nodes of a linked list are not constrained to be at contiguous memory locations; instead they can be stored anywhere in the memory. The linear order of the list is maintained by the pointer field(s) in each node.

Depending on the pointer field(s) in each node, linked lists can be of different types. If each node of a linked list contains only one pointer and it points to the next node, then it is called a linear linked list or singly linked list. In such type of lists, the pointer field in the last node contains NULL. However, if the pointer in the last node is modified to point to the first node of the list, then it is called a circular linked list.

In addition to the pointer to the next node, each node of a linked list can also contain a pointer to its previous node. This type of a linked list is called doubly linked list. Figure 1.1 shows a singly, circular, and doubly linked list with five nodes each as follows:



**Fig. 1.1** Various Types of Linked Lists

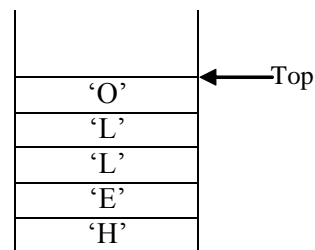
**Note:** One major reason behind the popularity of linked lists is that it can expand or shrink during its life.

## NOTES

As mentioned earlier, linked lists allow the elements of the list to be stored non-contiguously. Thus, in order to access an element in the list, one has to start with the first node and follow the pointers in the nodes until the required element is found or till the end of list. In other words, linked lists allow only sequential access to their elements, meaning, in order to access the  $n^{\text{th}}$  node of the list, the  $n-1$  preceding nodes need to be traversed.

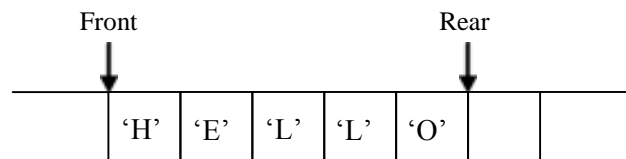
### Stacks and queues

A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end. This end is called, 'Top' of the stack. The operation of adding a new element in the stack and deleting an element from the stack is called push and pop respectively. Since the addition and deletion of elements always occurs at one end of the stack, the last element that is pushed onto the stack is the first one to come out. Therefore, a stack is also known as a Last-In-First-Out (LIFO) list. Figure 1.2 shows a stack with five elements and the position of top as follows:



**Fig. 1.2** A Stack with Five Elements

A *queue* is a linear data structure in which the addition or insertion of a new element occurs at one end, called '**Rear**', and deletion of an element occurs at other end, called '**Front**'. Since the insertion and deletion occur at opposite ends of the queue, the first element that is inserted in the queue is the first one to come out. Therefore, a queue is also called a **First-In-First-Out (FIFO)** list. Figure 1.3 shows a queue with five elements and the position of Front and Rear as follows:



**Fig. 1.3** A Queue with Five Elements

### 1.4.2 Non-Linear Data Structures

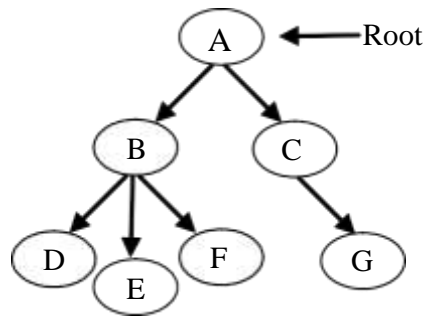
A non-linear data structure is one in which its elements do not form a sequence. It means, unlike a linear data structure, each element is not constrained to have a



unique the predecessor and a unique successor. Trees and graphs are the two data structures which come under this category. These data structures have been discussed in the subsequent paragraphs.

## Trees

Many-a-times, we observe a hierarchical relationship between various data elements. This hierarchical relationship between data elements can be easily represented using a non-linear data structure called *trees*. A tree consists of multiple nodes, with each node containing zero, one or more pointers to other nodes called child nodes. Each node of a tree has exactly one parent except a special node at the top of the tree called a *root node*. An example tree with A as the root of the tree is shown in Figure 1.4 as follows:



**Fig. 1.4** An Example Tree

In this tree, the root node has two child nodes B and C. In turn, the node B has three child nodes D, E and F, and the node C has one child G. The nodes D, E, F, and G have no child. The nodes without any child node are called external nodes or leaf nodes, whereas, the nodes having one or more child nodes are called internal nodes.

## Graphs

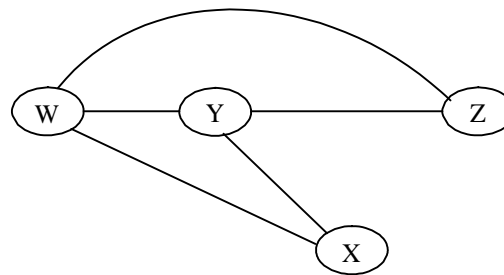
Formally, a graph  $G(V, E)$  consists of a pair of two non-empty sets  $V$  and  $E$ , where  $V$  is a set of vertices or nodes and  $E$  is a set of edges. The graph is used to represent the non-hierarchical relationship among pairs of data elements. The data elements become the vertices of the graph and the relationship is shown by edges between the two vertices. For example, assume four places W, X, Y, and Z such that

- ☐ There exists some path from X to Y, X to W, Y, to W, Y to Z, and Z to W.
- ☐ There is no direct path from X to Z.

We can simply represent this situation using a graph where the places W, X, Y, and Z are represented as the nodes of the graph and a path from one place to another place is represented by an edge between them (see Figure 1.5).

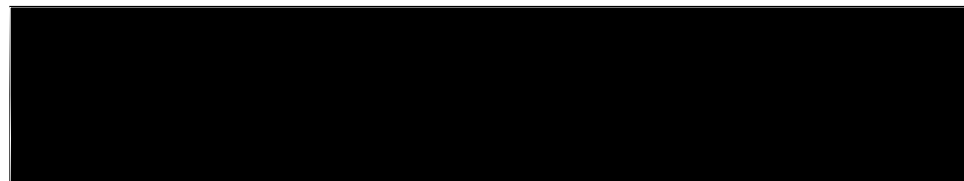
## NOTES

## NOTES



*Fig. 1.5 An Example Graph*

It is clear from Figure 1.5, that each node can have links with multiple other nodes. This analogy suggests a that it is similar to a tree, however, unlike trees, there is no root node in a graph. Further, graphs show relationships which may be non-hierarchical in nature. It means there is no parent and child relationship. But, a tree can be considered as a variant or a special type of graph.



---

## 1.5 OPERATIONS ON DATA STRUCTURES

---

As discussed earlier, the data needs to be processed by applying certain operations on it. Different data structures allow us to perform different types of operations on the data stored in them. The operations that need to be performed frequently on the data play an important role in the choice of a data structure for a particular situation. In this section, we will discuss various operations that can be performed on the data structures. They are listed as follows:

- **Traversing:** It means accessing all the data elements one by one to process all or some of them. For example, if we need to count or display the elements in a data structure, then traversing needs to be done.
- **Searching:** It is the process of finding the location of a given data element in the data structure. This operation involves one or more condition(s), and locates all the elements that satisfy the condition(s). If no element in the structure satisfies the condition(s), then the appropriate message is displayed.
- **Insertion:** It means adding a new data element in the data structure. Anew element can be inserted anywhere in the structure, such as in the beginning, in the end, or in the middle. Insertion in the beginning or in the end is usually simple or straightforward. However, inserting an element somewhere in the middle requires specifying the location or the data element after (or before) which the insertion is to be done.

- **Deletion:** It means removing any existing data element from the data structure. Deletion can be performed anywhere in the structure, in the beginning, in the end, or in the middle. Deleting an element may involve first locating it by applying the search operation and then deleting it.
- **Sorting:** It is the process of arranging all the elements of a data structure in a logical order such as ascending or descending order. There are various methods that can be applied for sorting the elements.
- **Merging:** It is the process of combining the elements of two sorted data structures into a single sorted data structure. Note that both the structures to be merged should be similar.

## NOTES

---

## 1.6 ALGORITHMS

---

In general, a problem may be defined as a state of mind of living beings to which they are not satisfied. Out of these problems, some of them can be solved with the use of computers. A solution to any solvable problem may be defined as a sequence of steps which when followed with the available (or allowed) resources lead to the satisfactory situation. A description of such a sequence of steps in some specific notation is called an algorithm.

Formally, an algorithm refers to a finite set of steps, when followed, solves a particular problem. Here, the word 'finite' means that the algorithm must terminate after performing a finite number of steps. An algorithm that goes on performing a set of steps infinitely is not of any use. Other than finiteness, an algorithm must have the following characteristics:

- It must take some input values supplied externally.
- It must produce some result or output.
- It must be definite, which means that all the steps in the algorithm must be clear and unambiguous.
- It must be effective, which means each step in the algorithm must be simple and basic so that any person can carry out these steps easily and effectively by using a pen and paper.

---

## 1.7 ALGORITHM DESIGN TECHNIQUES

---

There are various techniques which can be used for designing algorithms. Some commonly used algorithm design techniques have been discussed in this section.

### 1.7.1 Brute Force Algorithm

Brute force is a general technique which is used for finding solutions to various problems. In this technique, all possible candidates for the solution are listed and then examined to check whether each candidate satisfies the problem. For example,

## NOTES

to find the factors of a natural number  $n$ , it will determine the number of integers from 1 to  $n$  and then check all possible combinations of integers from 1 to  $n$ , that will form a product (equivalent to number  $n$ ) when multiplied together.

There are various algorithms where brute force technique can be applied; some of which are selection sort, pattern matching in strings, knapsack problem, and travelling salesman problem. Let us discuss how this technique is used in pattern matching. Pattern matching is the process of determining whether a given pattern of string (say, **P**) occurs in another string (say, **S**) or not, provided the length of string **P** is not greater than that of **S**. In other words, pattern matching determines whether or not **P** is a sub-string of **S**. Using this algorithm, the string **S** is scanned character by character. Starting from the first character, each character of **S** is compared with the first character of **P**. When the match for the first character is found, the next character from the pattern string **P** is compared with the character adjacent to the searched character in string **S**. This process continues till the complete pattern string is found. If the next character does not match, string **S** is searched again for other occurrences of the first character and also the subsequent characters of the pattern string **P** in the similar manner. This process continues until a match is found or the end of pattern string **P** is reached. If a match is found, this algorithm returns the position in **S** where the pattern string **P** occurs. For example, consider a pattern string **P** = “**ways**” that is to be searched in string **S** = “**hard work always pays**”, then this algorithm will return 13 as result.

The main advantage of brute force is that it is simple to implement. The algorithm will definitely find a solution if it exists, since it examines all possible solutions to a problem. However, the execution time of this algorithm is directly proportional to the number of solutions, that is, it increases rapidly with an increase in the size of the problem. Therefore, it is used in situations where the size of the problem is small or when some problem-specific heuristics are available that can be used to limit the number of possible solutions to a controllable size. It is also used as a baseline (an imaginary standard by which things are measured or compared) method to develop heuristics for other search algorithms.

### 1.7.2 Divide and Conquer

The divide and conquer technique is one of the widely used technique is to develop algorithms will for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently. The technique follows a top-down approach. To solve the problem, it recursively divides the problem into a number of sub-problems, to the extent where they cannot be sub-divided any further into more sub-problems. It then solves the sub-problems to find solutions that are then combined together to form a solution to the actual problem.

Some of the algorithms based on this technique are sorting, multiplying large numbers, syntactic analysis, etc. For example, consider the merge sort algorithm

that uses the divide and conquer technique. The algorithm is composed of steps, which are as follows:

**Step 1:** Divide the  $n$ -element list, into two sub-lists of  $n/2$  elements each, such that both the sub-lists hold half of the element in the list.

**Step 2:** Recursively sort the sub-lists using merge sort.

**Step 3:** Merge the sorted sub-lists to generate the sorted list.

Note that the merging of sub-lists starts, only when the length of sorted sub-lists (through recursive application) reach to 1. At this point, two sub-lists each of length 1 are merged (combined) by placing all the elements of the list in a sorted order.

### 1.7.3 Dynamic Programming

Dynamic programming is a technique that is generally used for solving optimization problems where the best (optimal) solution out of the available possible solutions is to be found. One example of such a problem is the shortest path problem where, if a person in city X has to reach city Z there would be many possible routes to reach the city Z. The aim is to select the shortest route from all the available routes so as to reach the destination in minimum possible time. Note that in the given problem, all possible routes represent different solutions to the problem.

Using dynamic programming, when the problem is solved, there is a possibility that sub-problems of the same type may arise. The basic idea behind the technique is that it stores the solutions to such sub-problems. This helps in repeated calculation and hence, improves the efficiency of the algorithm. The algorithms that are designed using this technique consist of three steps, which are as follows:

- **Dividing the problem into simpler sub-problems:** The problem is divided into sub-problems, such that each sub-problem has a similar structure to the original problem.
- **Finding optimal solutions to sub-problems:** The solution to an original problem is computed by combining the solutions of the sub-problems. Therefore, for finding optimal solution to the original problem, the solutions to sub-problems should also be optimal.
- **Storing solutions to overlapping sub-problems:** The identified sub-problems consist of either unrelated sub-problems (each having an independent solution) or common sub-problems (having similar optimal solution). The solutions to these sub-problems are stored in a table. So, while finding optimal solutions, if any overlapping (recurring) sub-problems are found, the solutions stored in the table can be used. This increases the efficiency of the dynamic programming algorithm.

Note that dynamic programming applies a bottom-up approach to solve the problem. That is, it first finds a solution to the simplest sub-instances of the problem and then solves the more complex instances, using the results of earlier

## NOTES

## NOTES

computed (sub) instances. Some of the well-known optimization problems where the dynamic programming technique is used are knapsack problem, problem of making change, shortest path problem, and chained matrix multiplication problem.

### 1.7.4 Greedy Algorithm

We have discussed the use of dynamic programming to solve optimization problems. However, there are many optimization problems such as the famous minimal spanning tree problem by Kruskal, minimum number of notes problem, and activity-selection problem that can be solved more efficiently using a greedy algorithm. As we know that the algorithm for optimization problems consist of stages, having a set of choices at each stage. The basic idea of the greedy technique is to make locally optimal choices (i.e., the best choice at a particular stage) assuming that these choices will result in a globally optimal solution.

The technique follows a top-down approach. To find solution to a problem, sequence of choices is made recursively (based on minimum or maximum value criterion) from the set of given choices at each stage. After a choice is made, the problem reduces to a sub-problem (similar to the actual problem). The solution to the actual problem is found by combining the sequence of choices that are made.

The greedy technique does not always lead to an optimal solution. However, the problems that are solvable using this technique are said to possess the greedy-choice property.

---

## 1.8 TIME AND SPACE COMPLEXITY OF ALGORITHMS

---

It is quite possible that there are one or more algorithms for solving a particular problem. If there are, they must be carefully analysed before choosing any one algorithm to be followed. The analysis of an algorithm gives us a general idea that how long it will take to solve a particular problem and what resources are required for it. Although, our aim is to choose the best algorithm, it is not always possible because of limited resources. The two main resources we consider for an algorithm are memory space and the processor time it requires.

Space complexity and time complexity are the two main measures for the efficiency of an algorithm that is considered during analysis. The space complexity of an algorithm is the maximum amount of memory space required by it at the time of execution. Frequently, the memory space required by an algorithm is the multiple of the size of input. On the other hand, the time complexity of an algorithm is the amount of time required to execute it. Here, by time we are not referring to the number of seconds or minutes required, instead it is a representation of the number of operations to be performed while executing the algorithm. This is because, if time for an algorithm is measured in seconds or in such time units on one computer, then upon moving to a faster or slower computer, this analysis will become invalid.

## NOTES

While measuring the time for an algorithm, only significant operations are taken into consideration. Significant operations are those operations which take much time in execution. Moreover, the number of times that they are to be executed gets affected significantly with change in the size of input. For example, at the time of searching algorithms, the significant operation is comparison, which is performed to check whether the value is the one we are searching for. The number of comparisons increase with the size of the list in which the search is to be done. Other operations, like prompting whether the search is successful or not, returning the position of searched value, etc., are performed fixed number of times, irrespective of the size of the list.

Note that in a search algorithm, the number of times that the comparison is to be done is also affected by the location of value to be searched in the list. If the value is found at the first location then only one comparison needs to be done. This is the best case for an algorithm, and in this case, it has to do the least amount of work. On the other hand, if the value is found at the last location or is not present in the list at all,  $N$  comparisons will be required, where  $N$  is the size of the list. This is the worst case for the algorithm, and in this case, it has to do maximum amount of work. Another case that is also considered is an average case in which an algorithm gives average performance. For example, in the simplest search algorithm, on an average, half the elements need to be compared.

### Time-space tradeoff

The best or efficient algorithm is the one which utilizes minimum processor time and requires minimum memory space during its execution. However, unfortunately, it is not always possible to develop an algorithm which is efficient in terms of both space and time. Therefore, while designing an algorithm, we may have to compromise on one at the cost of the other.

Suppose, for a given problem, an algorithm is developed which takes  $S$  amount of memory and utilizes  $T$  amount of processor time during its execution. Note that  $S$  and  $T$  are not always as least as possible. However, the algorithm may be modified to minimize  $T$  at the cost of some extra memory space. Similarly, the algorithm may also be modified to minimize  $S$  at the cost of extra processor time. Therefore, if memory is the major constraint, then an algorithm which takes minimum memory space can be chosen. On the other hand, if time is the major constraint, then an algorithm which will utilize the minimum processor time can be chosen. Since with modern computers, memory is not a constraint, we mainly focus on the algorithms that utilize the minimum processor time, unless or otherwise stated.

---

## 1.9 BIG O NOTATION

---

In the analysis of algorithms, the growth factor plays an important role. This is because it is quite possible that an algorithm will perform lesser number of operations than the other when the size of input is small, however, many more when the size of

## NOTES

input gets larger. Therefore, during the analysis of algorithms, we would not be interested in determining the number of operations to be performed for some specific input, say  $N$ . Instead, we are interested in determining the equation that relates the number of operations to be performed to the size of the input. Once we are ready with the equations for two algorithms, we can determine the rate at which the equations will grow with growth in the size of input. Table 1.3 shows the rate of growth for common standard functions with the increase in the input size.

**Table 1.3** Rate of Growth for Common Standard Functions

Size of Input (n)	f(n)	f(n <sup>2</sup> )	f(n <sup>3</sup> )	f(2 <sup>n</sup> )	f(log <sub>2</sub> n)	f(n log <sub>2</sub> n)
1	1	1	1	2	0	0
5	5	25	125	32	2.3	11.6
10	10	100	1000	1024	3.3	33.2
20	20	400	8000	1048576	4.3	86.4
50	50	2500	125000	1125899906842624	5.6	282.2

From Table 1.3, it is clear that the function  $\log_2 n$  is the slowest growing function and the function  $2^n$  is the fastest growing function. One can observe that the function  $2^n$  does not have much difference in the values with other functions when the size of the input is small. However, as the input size grows, there becomes a huge difference. To verify whether a function will grow faster or slower than the other function, we have some asymptotic or mathematical notations, which are as follows:

- **Big Omega  $\Omega(f)$** : A function  $f(n)$  is  $\Omega(g(n))$ , if there exists positive values  $k$  and  $c$  such that  $f(n) \geq c \cdot g(n)$ , for all  $n \geq k$ . This notation defines a lower bound for a function  $f(n)$ .
- **Big Theta  $\Theta(f)$** : A function  $f(n)$  is  $\Theta(g(n))$ , if there exists positive values  $k$ ,  $c_1$ , and  $c_2$  such that  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ , for all  $n \geq k$ . This notation defines both a lower bound as well as an upper bound for a function  $f(n)$ .
- **Big Oh  $O(f)$** : A function  $f(n)$  is  $O(g(n))$ , if there exists positive values  $k$  and  $c$  such that  $f(n) \leq c \cdot g(n)$ , for all  $n \geq k$ . This notation defines an upper bound for a function  $f(n)$ .
- **Little oh  $o(f)$** : A function  $f(n)$  is  $o(g(n))$ , if  $f(n)$  is  $O(g(n))$  and  $f(n)$  is not  $\Omega(g(n))$  (that means, there exists no positive values  $k$  and  $c$  such that  $f(n) \geq c \cdot g(n)$ , for all  $n \geq k$ .)

While comparing any two algorithms, the algorithm whose equation (that relates to the number of operations to the size of the input) grows slowly than the other, is considered better. It means that we are interested in finding that algorithm (out of the two) whose equation is in the Big Oh of another one. Such algorithm will definitely perform better than the other when the input size is large. To understand this, suppose the equation for the first and second algorithms are  $f(n) = 14n^2 + 8n$  and  $g(n) = 5n^3 + 3$  respectively. In order to find which algorithm



works better, the values of  $f(n)$  and  $g(n)$  are computed for some sample values of  $n$ , which is shown in Table 1.4 as follows:

**Table 1.4** Values of  $f(n)$  and  $g(n)$  for Sample Values of  $n$

Value of $n$	$f(n)$	$g(n)$
1	22	8
2	72	43
3	150	138
4	256	323
5	390	628
6	552	1083

From Table 1.4, it is clear that for all positive values of  $n \geq 4$ , the value of  $g(n)$  is larger than  $f(n)$ . It implies that  $f(n)$  is in  $O(g(n))$  for all  $n \geq 4$ . Since computer programs generally deal with large number of values, the first algorithm with complexity  $f(n) = 14n^2 + 8n$  is chosen, since it works better for all  $n \geq 4$ .

These asymptotic notations also facilitate the recognition of essential characters of a complexity function through some simpler functions. For example, consider the function  $f(n) = 2n^3 + 3n^2 + 1$ . We know,

$$\begin{aligned} f(n) &= 2n^3 + 3n^2 + 1 \\ &<= 2n^3 + 3n^3 + 1n^3 = 6n^3 \text{ for all } n \geq 1 \end{aligned}$$

It is found that  $f(n) = O(n^3)$  with  $c = 6$  and  $k = 1$ . It means that the function  $f(n)$  has essentially the same behaviour as that of  $n^3$ , when the size of  $n$  grows and becomes larger and larger. However, computing the value of  $n^3$  is much easier than computing the value of function  $f(n)$ .

## 1.10 RECURRENCES

**Recurrence** can be described as is an equation or inequality that defines a function in terms of its own values. It is used to express the complexity of algorithms. For example, the recurrence for the merge-sort algorithm can be expressed as follows:

$$f(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(f(n/2)) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solving recurrences makes it easy to compare the complexity of any two algorithms. The recurrence can be solved by using any of the following methods:

- **Substitution method:** In this method, a reasonable guess for the solution is made and it is proved through mathematical induction.
- **Recursion tree:** In this method, recurrences are represented as a tree whose nodes indicate the cost that is incurred at the various levels of recursion.

## NOTES

## NOTES

- **Master method:** This method is used to determine the asymptotic solutions to recurrences of the specific form.

Let us see how recurrence can be solved through the master method. The master method uses the master theorem, which is as follows:

Let  $a \geq 1$  and  $b > 1$ ,  $g(n)$  is asymptotically positive and let  $f(n)$  be defined by

$$f(n) = af(n/b) + g(n)$$

Then  $f(n)$  can be bounded asymptotically as follows:

1. If  $g(n) = O(n^{\log_b a})$  for some  $\epsilon > 0$ , then  $f(n) = O(n^{\log_b a})$ .
2. If  $g(n) = \Theta(n^{\log_b a})$ , then  $f(n) = \Theta(n \lg n)$ .
3. If  $g(n) = \Omega(n^{\log_b a + \epsilon})$  for some  $\epsilon > 0$ , and if  $ag(n/b) \leq cg(n)$  for some constant  $c < 1$  and for all sufficiently large  $n$ , then  $f(n) = \Theta(g(n))$ .

In each of the above three cases, the function  $f(n)$  is compared with the function  $g(n)$ . While the above comparison, the following three cases may arise:

1. If the function  $n^{\log_b a}$  is greater than  $g(n)$ , then the solution is  $f(n) = \Theta(n^{\log_b a})$ .
2. If function is equal to  $g(n)$ , then the solution is  $f(n) = \Theta(n \lg n) = \Theta(g(n) \lg n)$ .
3. If function  $g(n)$  is greater than  $n^{\log_b a}$ , then the solution is  $f(n) = \Theta(g(n))$ .

For example, consider the following recurrences:

(i)  $f(n) = 4f(n/2) + n$

Here,  $a = 4$ ,  $b = 2$ , and  $g(n) = n$

Therefore,

$n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$ . Since,  $g(n) = O(n^{\log_2 4})$  where  $\epsilon = 1$ , apply the first case of the master theorem to conclude the solution. The solution is  $f(n) = \Theta(n^2)$ .

(ii)  $f(n) = f(3n/4) + 1$

Here,  $a = 1$ ,  $b = 4/3$ ,  $g(n) = 1$

Therefore,  $n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = 1$ . Since,  $g(n) = \Theta(n^{\log_b a}) = \Theta(1)$ , apply the second case of the master theorem to conclude the solution. The solution is  $f(n) = \Theta(n \lg n)$ .

(iii)  $f(n) = 4f(n/5) + n \lg n$

Here,  $a = 4$ ,  $b = 5$ ,  $g(n) = n \lg n$

Therefore,

$n^{\log_5 2} = n^{\log_5 4}$ . Since  $g(n) = \Theta(n^{\log_5 4 + 1})$ , where  $\Theta(1) > 0$ , apply the third case of the master theorem to conclude the solution. The solution is  $f(n) = \Theta(n \lg n)$ .

## NOTES

---

### 1.11 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. Primitive data types are the data types provided by a programming language.
2. The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT.
3. Data structures are divided into two categories, namely linear data structure and non-linear data structure.
4. A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor.
5. A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end.
6. A queue is a linear data structure in which the addition or insertion of a new element occurs at one end, called 'Rear', and deletion of an element occurs at other end, called 'Front'.

---

### 1.12 SUMMARY

---

- Each programming language provides various data types and each data type is represented differently within the computer's memory.
- The memory requirement of a data type determines the permissible range of values for that data type.
- The data types can be classified into several categories, including primitive data types and composite data types.
- The data types provided by a programming language are known as primitive data types or in-built data types.

## NOTES

- In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements.
- Generally, handling small problems is much easier than handling comparatively larger problems.
- The size of each module is kept as small as possible and if required, other modules are invoked from it.
- Second, a well-designed modular program has modules independent of each other's, implementation, which will make the program easily modifiable.
- An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented.
- The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation.
- If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it.
- The logical or mathematical model used to organize the data in main memory is called a data structure.
- These features should be kept in mind while choosing a data structure for a particular situation.
- The choice of a data structure depends on its simplicity and effectiveness in processing of data.
- Data structures are divided into two categories, namely, linear data structure and non-linear data structure.
- A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor.
- A finite collection of homogenous elements is termed as an array.
- The elements of an array are always stored in contiguous memory locations irrespective of the array size.
- A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end.
- A queue is a linear data structure in which the addition or insertion of a new element occurs at one end, called 'Rear', and deletion of an element occurs at other end, called 'Front'.
- A tree consists of multiple nodes, with each node containing zero, one or more pointers to other nodes called child nodes.

---

### 1.13 KEY WORDS

---

- **Traversing:** It means accessing all the data elements one by one to process all or some of them.
- **Substitution method:** In this method, a reasonable guess for the solution is made and it is proved through mathematical induction.
- **Recursion tree:** In this method, recurrences are represented as a tree whose nodes indicate the cost that is incurred at the various levels of recursion.
- **Searching:** It is the process of finding the location of a given data element in the data structure.
- **Insertion:** It means adding a new data element in the data structure. A new element can be inserted anywhere in the structure, such as in the beginning, in the end, or in the middle.
- **Deletion:** It means removing any existing data element from the data structure.
- **Sorting:** It is the process of arranging all the elements of a data structure in a logical order such as ascending or descending order.
- **Merging:** It is the process of combining the elements of two sorted data structures into a single sorted data structure.

### NOTES

---

### 1.14 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. Write a short note on abstract data types.
2. Write some points of differences between linear and non-linear data structures.
3. What are the different operations on data structures?

#### Long-Answer Questions

1. “The data types provided by a programming language are known as primitive data types or in-built data types. Different programming languages provide different set of primitive data types.” Discuss in detail.
2. “If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it.” Explain.
3. Write a detailed note on Algorithm Design Techniques.
4. What do you mean by time and space complexity of algorithms?

## NOTES

---

### 1.15 FURTHER READINGS

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

---

### 1.16 LEARNING OUTCOMES

---

- Understand composite data types
- Linear and non-linear data structure
- Abstract data types
- Understand algorithms and their design techniques
- Understand time and space complexity of algorithms

## UNIT 2    ARRAYS

### Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Arrays (Ordered Lists)
  - 2.2.1 Single-Dimensional Arrays
  - 2.2.2 Multi-Dimensional Arrays
- 2.3 Representation of an Array
  - 2.3.1 Memory Representation of a Single-Dimensional Array
  - 2.3.2 Memory Representation of a Two-Dimensional Array
- 2.4 Answers to Check Your Progress Questions
- 2.5 Summary
- 2.6 Key Words
- 2.7 Self Assessment Questions and Exercises
- 2.8 Further Readings
- 2.9 Learning Outcomes

### NOTES

## 2.0 INTRODUCTION

An Array is a collection of variables that belong to the same data type. You can also store groups of data of the same data type within an array. An Array might belong to any of the data types. It is in fact a data structure which can store a fixed-size collection of elements of the same data type. An array can also store a collection of data, and can be called a collection of variables of the same kind. The simplest type of a data structure is a linear array, which is also called a one-dimensional array. In computer science, an array type is a data type that is meant to describe a collection of elements. In this unit, you will learn about the arrays and its kinds.

## 2.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain single dimensional arrays
- Analyze multi-dimensional arrays
- Discuss the memory representation of a single-dimensional arrays
- Describe the memory representation of a two-dimensional array

## 2.2 ARRAYS (ORDERED LISTS)

Array is one of the data types that can be used for storing a list of elements. When programmers want to store a list of elements under a single variable name, but still want to access and manipulate an individual element of the list, then arrays are

**NOTES**

used. Arrays can be defined as a fixed-size sequence of elements of the same data type. These elements are stored at contiguous memory locations and can be accessed sequentially or randomly. The programmer can access a particular element of an array by using one or more indices or subscripts. If only one subscript is used then the array is known as a single-dimensional array. If more than one subscript is used then the array is known as a multi-dimensional array.

**2.2.1 Single-Dimensional Arrays**

A single-dimensional array is defined as an array in which only one subscript value is used to access its elements. It is the simplest form of an array. Generally, a single dimensional array is denoted as follows:

```
array_name[L:U]
```

where,

array\_name = the name of the array

L = the lower bound of the array

U = the upper bound of the array

Before using an array in a program, it needs to be declared. The syntax of declaring a single-dimensional array in C is as follows:

```
data_type array_name[size];
```

where,

data\_type = data type of elements to be stored in array

array\_name = name of the array

size = the size of the array indicating that the lower bound of the array is 0 and the upper bound is size-1. Hence, the value of the subscript ranges from 0 to size-1.

For example, in the statement `int abc[5]`, an integer array of five elements is declared and the array elements are indexed from 0 to 4. Once the compiler reads a single-dimensional array declaration, it allocates a specific amount of memory for the array. Memory is allocated to the array at the compile-time before the program is executed.

**Initializing and accessing single-dimensional array**

An array can be initialized in two ways. It can be done by declaring and initializing it simultaneously or by accepting elements of the already declared array from the user. Once an array is declared and initialized, the elements stored in it can be accessed any time. These elements can be accessed by using a combination of the name of an array and subscript value.

**Example 2.1:** A program to illustrate the initialization of two arrays and display their elements is as follows:

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



```

#define MAX 5
void main()
{
    int A[MAX]={1,2,3,4,5};
    int B[MAX], i;
    clrscr();
    printf("Enter the elements of array B:\n");
    for (i=0;i<MAX;i++)
    {
        printf("Enter the element: ");
        scanf("%d", &B[i]);
    }
    printf("Elements of array A: \n");
    for (i=0;i<MAX;i++)
        printf("%d\t", A[i]);
    printf("\nElements of array B: \n");
    for (i=0;i<MAX;i++)
        printf("%d\t", B[i]);
    getch();
}

```

## NOTES

### The output of the program is as follows:

```

Enter the elements of array b:
Enter a value: 6
Enter a value: 7
Enter a value: 8
Enter a value: 9
Enter a value: 10
Elements of array a:
1      2      3      4      5
Elements of array b:
6      7      8      9      10

```

In this example, an array *A* is declared and initialized simultaneously and the elements for the array *B* are accepted from the user, then the elements of both the arrays are displayed.

Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array. To perform any operation on an array, the elements of the array need to be accessed. The process of accessing each element of an array is known as **traversal**. Generally, the traversal of an array is performed from the element at position 0 to element at position *size-1*.

**NOTES****Algorithm 2.1 Traversing an Array**

```

traverse (ARR, size)
1. Set i = 0, sum = 0
2. Print "The elements of the array are: "
3. While i < size          //size indicates number of elements in the array
    Print ARR[i]
    Set sum = sum + ARR[i]
    Set i = i + 1
End While
4. Print "Sum of elements of an array: ", sum
5. End

```

**Example 2.2:** A program to illustrate the traversal of an array is as follows:

```

#include<stdio.h>
#include<conio.h>
#define MAX 10

/*Function  prototype*/
void traverse(int [], int);

void main()
{
    int ARR[MAX];
    int i, size;
    clrscr();
    printf("Enter the number of elements in array:\n");
    scanf("%d", &size);
    printf("Enter the elements of the array:\n");
    for (i=0;i<size;i++)
    {
        scanf("%d", &ARR[i]);
    }
    traverse(ARR, size);
    getch();
}

/*Function to find the sum of the elements of matrix*/
void traverse(int ARR[], int size)
{
    int i, sum=0;
    printf("The elements of the array are:\n");
    for (i=0;i<size;i++)
    {

```

```

        printf("%d ", ARR[i]);
        sum+=ARR[i];
    }
    printf("\nSum of elements of an array: %d", sum);
}

```

**The output of the program is as follows:**

```

Enter the number of elements in array: 5
Enter the elements of the array:
12 23 34 45 56
The elements of the array are:
12 23 34 45 56
Sum of elements of an array: 170

```

### 2.2.2 Multi-Dimensional Arrays

Multi-dimensional arrays can be described as ‘arrays of arrays’. A multi-dimensional array of dimension *n* is a collection of elements, which are accessed with the help of *n* subscript values. Most of the high-level languages, including C, support arrays with more than one dimension. However, the maximum limit of an array dimension is compiler dependent.

The syntax of declaring a multi-dimensional array in C is as follows:

```
element_type array_name[a][b][c] ..... [n];
```

where,

```

element_type = the data type of array
array_name = name of the array
[a][b][c] ..... [n] = array subscripts

```

The arrays of three or more dimensions are not often used because of their huge memory requirements and the complexity involved in their manipulation. Hence, only two-dimensional and three-dimensional arrays have been discussed in brief in this unit.

#### Two-dimensional arrays

A two-dimensional array is one in which two subscript values are used to access an array element. They are useful when the elements being processed are to be arranged in rows and columns (matrix form).

Generally, a two-dimensional array is represented as as follows:

```
A[Lr : Ur, Lc : Uc]
```

where,

```

Lr and Lc = the lower bounds of a row and column, respectively
Ur and Uc = the upper bounds of a row and column, respectively

```

## NOTES

**NOTES**

The number of rows in a two-dimensional array can be calculated by using the formula  $(U_r - L_r + 1)$  and the number of columns can be calculated by using the formula  $(U_c - L_c + 1)$ .

Like a single-dimensional array, a two-dimensional array also needs to be declared first. The syntax of declaring a two-dimensional array in C is as follows:

```
data_type                array_name
[ row_size ] [ column_size ] ;
```

For example, in the statement `int a[3][3]`, an integer array of three rows and three columns is declared. Once a compiler reads a two-dimensional array declaration, it allocates a specific amount of memory for this array.

**Initializing and accessing two-dimensional arrays**

A two-dimensional array can be initialized in two ways just like a single-dimensional array, i.e., by declaring and initializing the array simultaneously and by accepting array elements from the user.

Once a two-dimensional array is declared and initialized, the array elements can be accessed anytime. Same as one-dimensional arrays, two-dimensional array elements can also be accessed by using a combination of the name of the array and subscript values. The only difference is that instead of one subscript value, two subscript values are used. The first subscript indicates the row number and the second subscript indicates the column number of a two-dimensional arrays.

**Algorithm 2.2 Traversing a Two-Dimensional Array**

```
traverse (ARR, m, n)
1. Set i = 0, sum = 0      //sum stores sum of elements of two-dimensional array
2. While i < m             //m is number of rows in two-dimensional array
    Set j = 0
    While j < n //n is number of columns in two-dimensional array
        Print ARR[i][j]
        Set sum = sum + ARR[i][j]
        Set j = j + 1
    End While
    Set i = i + 1
End While
3. Print "Sum of the elements of a matrix is : ", sum
4. End
```

**Example 2.3:** A program to illustrate the traversal of a matrix (two-dimensional array) and finding the sum of its elements is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void traverse(int[][MAX], int,int);
void main()
{
```

```

    int ARR[MAX][MAX], i, j, m, n;
    clrscr();
    printf("Enter the number of rows and columns of a
matrix A: ");
    scanf("%d%d", &m, &n);
    printf("Enter the elements of matrix A: \n");
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%d", &ARR[i][j]);
    traverse(ARR, m, n);
    getch();
}
/*Function to find sum of elements of the matrix*/
void traverse(int ARR[][MAX], int m, int n)
{
    int i, j, sum=0;
    printf("Matrix A is: ");
    for(i=0;i<m;i++)
    {
        printf("\n");
        for(j=0;j<n;j++)
        {
            printf("%d ", ARR[i][j]);
            sum=sum+ARR[i][j];
        }
    }
    printf("\nSum of elements of a matrix is: %d", sum);
}

```

**The output of the program is as follows:**

```

Enter the number of rows and columns of a matrix A: 3 3
Enter the elements of matrix A:
1 2 3
4 5 6
7 8 9
Matrix A is:
1 2 3
4 5 6
7 8 9
Sum of elements of a matrix is: 45

```

## NOTES

**NOTES****Three-dimensional arrays**

A three-dimensional array is defined as an array in which three subscript values are used to access an individual array element. The three-dimensional array can be declared as follows:

```
int A[3][3][3];
```

**Algorithm 2.3 Traversing a Three-Dimensional Array**

```
traverse (ARR)
1. Set i = 0, count = 0    //count is used to count the number of zeroes in
                           //three-dimensional array
2. While i < MAX           //MAX is the size of three-dimensional array
    Set j = 0
    While j < MAX
        Set k = 0
        While k < MAX
            If ARR[i][j][k] = 0
                Set count = count + 1
            End If
            Set k = k + 1
        End While
        Set j = j + 1
    End While
    Set i = i + 1
End While
3. Print "Number of zeroes in given array are: ", count
4. End
```

**Example 2.4:** A program to illustrate the traversal of a three-dimensional array and to find the number of zeroes in it is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 3

/*Function prototype*/
void traverse(int[][MAX][MAX]);

void main()
{
    int ARR[MAX][MAX][MAX], i, j, k;
    clrscr();
    printf("Enter the elements of an array A(3x3x3):\n");
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            for(k=0;k<MAX;k++)
                scanf("%d", &ARR[i][j][k]);
    traverse(ARR);
    getch();
}
```

```

void traverse(int ARR[] [MAX] [MAX])
{
    int i, j, k, count=0;
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            for(k=0;k<MAX;k++)
                if(ARR[i][j][k]==0)
                    count++;
    printf("Number of zeroes in given array are: %d",
count);
}

```

**The output of the program is as follows:**

```

Enter the elements of an array A(3x3x3):
1 2 3 0 4 5 0 6 7
0 7 3 2 0 5 6 0 8
2 0 4 5 0 7 6 9 0
Number of zeroes in given array are: 8

```

## NOTES

## 2.3 REPRESENTATION OF AN ARRAY

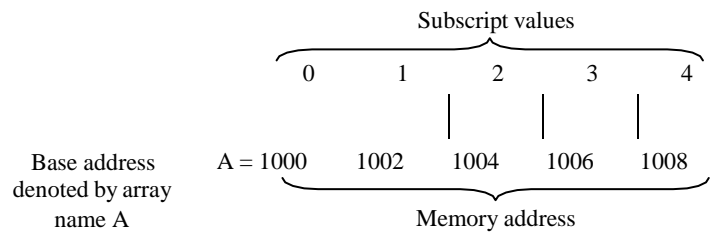
All the elements in an array are always stored next to each other. Also, the memory address of the first element of an array is contained in the name of the array. The memory location, where the first element of an array is stored, is known as the **base address**, which is generally referred to by the name of the array.

### 2.3.1 Memory Representation of a Single-Dimensional Array

Each element in a single-dimensional array is associated with a unique subscript value, starting from 0 to size-1. The subscript value for an element specifies its position starting from the base address and the difference between the memory addresses of two consecutive array elements is the size of an array's data type. For example, let us study a declaration as follows:

```
int A[5];
```

In this declaration, an integer array of five elements is declared. The array name *A* refers to the base address of the array. It must be noted that array elements, in Figure 2.1 are indexed from 0 to 4.

**NOTES****Fig. 2.1** Base Address and Subscript Values of an Array**Address calculation in a single-dimensional array**

The base address of an array can be used to calculate the address of any element in an array. The formula to calculate the address of an element of an array is as follows:

$$\text{Array}[I] = \text{Base} + (I-L) * \text{Size}$$

where,

Array = name of an array

I = position of an array element whose address needs to be calculated

L = lower bound of an array

Base = base address of an array

Size = size of each element of an array, i.e., the number of bytes of space occupied by the individual element of the array

Similarly, the length of an array, i.e., the number of elements in the array can be calculated by the formula:

$$\text{Length of Array} = U - L + 1$$

where,

U = upper bound of an array

L = lower bound of an array

The concept of address calculation in a single-dimensional array will be clear from the examples as provided here.

**Example 2.5:** Calculate the address of the fourth element of a floating point array A[10] implemented in C. The base address of the array is 1000.

**Solution:** Here,

$$\text{Base} = 1000$$

$$\text{Size} = 4 \text{ (size of a float variable)}$$

$$I = 3$$

$$L = 0 \text{ (in C)}$$

Substituting these values in the formula  $\text{Array}[I] = \text{Base} + (I-L) * \text{Size}$ , the required address can be calculated as follows :

$$\begin{aligned} \text{Address of } A[3] &= 1000 + (3-0) * 4 \\ &= 1000 + 12 \\ &= 1012 \end{aligned}$$



Thus, the array contains 9 elements.

**Address calculation in a two-dimensional array**

The address of any element of a two-dimensional array stored in the row-major order can be calculated using the following formula:

**NOTES**

$$A[I, J] = \text{Base} + \text{Size} * (C * (I - L_r) + (J - L_c))$$

where,

A = name of an array

I, J = position of the element whose address has to be calculated

Base = base address of the two-dimensional array

Size = size of an individual elements of the array

C = number of columns in each row

L<sub>r</sub> = lower bound of rows

L<sub>c</sub> = lower bound of columns

To understand the concept of calculation of an address in a two-dimensional array in the row-major order, a few examples have been provided here.

**Example 2.7:** If a two-dimensional array C[5 .. 10, -5 .. 9] is stored in a row-major order, calculate the address of C[8, -2] if the base address is 10 and each array element requires 2 bytes of memory space.

**Solution:** Here,

I = 8                      J = -2                      Base = 10                      Size = 2  
L<sub>r</sub> = 5                      U<sub>r</sub> = 10                      L<sub>c</sub> = -5                      U<sub>c</sub> = 9

$$C = (U_c - L_c + 1) = (9 - (-5) + 1) = 9 + 5 + 1 = 15$$

Substituting these values in the formula  $A[I, J] = \text{Base} + \text{Size} * (C * (I - L_r) + (J - L_c))$ , the required address can be calculated as follows:

$$\begin{aligned} \text{Address of } C[8, -2] &= 10 + 2 * (15 * (8 - 5) + (-2 - (-5))) \\ &= 10 + 2 * (15 * 3 + 3) \\ &= 10 + 2 * 48 \\ &= 10 + 96 \\ &= 106 \end{aligned}$$

**Example 2.8:** The two-dimensional array A[5][10] is stored in the memory using a row-major order. Calculate the address of the element A[2][3] if the base address is 150 and each element requires 4 bytes of memory space.

**Solution:** Here (assuming array is represented in C),

I = 2                      J = 3                      Base = 150                      Size = 4  
L<sub>r</sub> = 0                      U<sub>r</sub> = 4                      L<sub>c</sub> = 0                      U<sub>c</sub> = 9

$$C = (U_c - L_c + 1) = (9 - 0 + 1) = 10$$

Substituting these values in the formula  $A[I, J] = \text{Base} + \text{Size} * (C * (I - L_r) + (J - L_c))$ , the required address can be calculated as follows:

$$\begin{aligned} \text{Address of } A[2][3] &= 150 + 4 * (10 * (2 - 0) + (3 - 0)) \\ &= 150 + 4 * 23 \\ &= 150 + 92 \\ &= 242 \end{aligned}$$

The address of any element of a two-dimensional array stored in the column-major order can be calculated using the following formula:

$$A[I, J] = \text{Base} + \text{Size} * ((I - L_r) + r * (J - L_c))$$

where,

- A = the array name
- I, J = the position of the element whose address has to be calculated
- Base = the base address of the two-dimensional array
- Size = the size of the individual elements of the array
- r = the number of rows in each column
- Lr = the lower bound of rows
- Lc = the lower bound of columns

NOTES

To understand the concept of calculation in a two-dimensional array in a column-major order, a few examples have been provided here.

**Example 2.9:** If a two-dimensional array `C[5 .. 10, -5 .. 9]` is stored in a column-major order, calculate the address of `C[8, -2]` if the base address is 10 and each array element requires 2 bytes of memory space.

**Solution:** Here,

<code>I = 8</code>	<code>J = -2</code>	<code>Base = 10</code>	<code>Size = 2</code>
<code>Lr = 5</code>	<code>Ur = 10</code>	<code>Lc = -5</code>	<code>Uc = 9</code>

$r = (Ur - Lr + 1) = (10 - 5 + 1) = 6$

Substituting these values in the formula  $A[I, J] = Base + Size * ((I - Lr) + r * (J - Lc))$ , the required address can be calculated as follows:

Address of `C[8, -2]`

- $= 10 + 2 * ((8 - 5) + 6 * (-2 - (-5)))$
- $= 10 + 2 * (3 + 18)$
- $= 10 + 2 * 21$
- $= 10 + 42$
- $= 52$

**Example 2.10:** The two-dimensional array `A[5][10]` is stored in memory using the column-major order. Calculate the address of the element `A[2][3]` if the base address is 150 and each element requires 4 bytes of memory space.

**Solution:** Here (assuming array represented in C),

<code>I = 2</code>	<code>J = 3</code>	<code>Base = 150</code>	<code>Size = 4</code>
<code>Lr = 0</code>	<code>Ur = 4</code>	<code>Lc = 0</code>	<code>Uc = 9</code>

$r = (Ur - Lr + 1) = (4 - 0 + 1) = 5$

Substituting these values in the formula  $A[I, J] = Base + Size * ((I - Lr) + r * (J - Lc))$ , the required address can be calculated as follows:

Address of `A[2][3]`

- $= 150 + 4 * ((2 - 0) + 5 * (3 - 0))$
- $= 150 + 4 * 17$
- $= 150 + 68$
- $= 218$



**NOTES**

---

**2.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS**

---

1. A single-dimensional array is defined as an array in which only one subscript value is used to access its elements.
2. Two-dimensional arrays are represented in a linear form to enable the storage of elements in the contiguous memory locations.
3. To perform any operation on an array, the elements of the array need to be accessed.

---

**2.5 SUMMARY**

---

- Array is one of the data types that can be used for storing a list of elements.
- Arrays can be defined as a fixed-size sequence of elements of the same data type.
- The programmer can access a particular element of an array by using one or more indices or subscripts.
- If only one subscript is used then the array is known as a single-dimensional array.
- A single-dimensional array is defined as an array in which only one subscript value is used to access its elements.
- Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array.
- To perform any operation on an array, the elements of the array need to be accessed.
- The process of accessing each element of an array is known as traversal.
- Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension  $n$  is a collection of elements, which are accessed with the help of  $n$  subscript values.
- Most of the high-level languages, including C, support arrays with more than one dimension.
- All the elements in an array are always stored next to each other.
- Each element in a single-dimensional array is associated with a unique subscript value, starting from 0 to size-1.

- Two-dimensional arrays are represented in a linear form to enable the storage of elements in the contiguous memory locations.
- There are two ways in which a two-dimensional array can be represented in the linear form which are row-major order and column-major order.
- In a row-major order representation of a two-dimensional array, first the elements of the first row are stored sequentially in the memory, then the elements of the second row are stored sequentially, and so on.

## NOTES

### 2.6 KEY WORDS

- **Base Address:** It is the memory location, where the first element of an array is stored.
- **A three-dimensional array:** It is defined as an array in which three subscript values are used to access an individual array element.

### 2.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. Write a short note on single dimensional arrays.
2. Write in brief about multi-dimensional arrays
3. What do you mean by memory representation of a single-dimensional arrays?
4. Describe the memory representation of a two-dimensional array.

#### Long-Answer Questions

1. "Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array." Discuss. Also explain how can an operation be performed on an array?
2. Write a program to illustrate the traversal of an array.
3. "Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension n is a collection of elements, which are accessed with the help of n subscript values. " Discuss.
4. "A two-dimensional array is one in which two subscript values are used to access an array element." Discuss two-dimensional arrays in detail.
5. Write a program to illustrate the traversal of a matrix (two-dimensional array) and finding the sum of its elements.

**NOTES**

---

**2.8 FURTHER READINGS**

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

---

**2.9 LEARNING OUTCOMES**

---

- Explain single dimensional arrays
- Analyze multi-dimensional arrays
- Discuss the memory representation of a single-dimensional arrays
- Describe the memory representation of a two-dimensional array

---

## BLOCK - II

### LINEAR DATA STRUCTURE

---

## NOTES

## UNIT 3    STACK

---

### Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Stack Related Terms and Operations on Stack
- 3.3 Answers to Check Your Progress Questions
- 3.4 Summary
- 3.5 Key Words
- 3.6 Self Assessment Questions and Exercises
- 3.7 Further Readings
- 3.8 Learning Outcomes

---

### 3.0 INTRODUCTION

---

A **stack** is a linear data structure in which an element can be added or removed only at one end called the **top** of the stack. In the terminology related to stacks, the insert and delete operations are known as **PUSH** and **POP** operations respectively. The last element added to the stack is the first element to be removed, that is, the elements are removed in the opposite order in which they are added to the stack. Hence, a stack works on the principle of last in first out, and is also known as a **last-in-first out (LIFO)** list. In this unit, you will learn about the stack organisation and operations on stack.

---

### 3.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss stack and its definition
- Explain the various stack related terms
- Analyze the operations on stack

---

### 3.2 STACK RELATED TERMS AND OPERATIONS ON STACK

---

A stack can be organized (represented) in the memory either as an array or as a singly-linked list. In both the cases, insertion and deletion of elements is allowed only at one end. Insertion and deletion at the middle of an array or a linked list is not allowed. An array representation of a stack is static, but linked list representation

## NOTES

is dynamic in nature. Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization. This is because if the number of elements to be stored in a stack is less than the allocated memory then the memory space will be wasted. Conversely, if the number of elements to be handled by a stack is more than the size of the stack, then it will not be possible to increase the size of the stack to store these elements. In this section, only the array organization of a stack will be discussed.

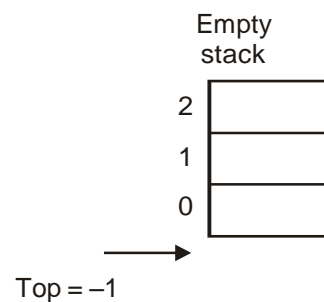
When a stack is organized as an array, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set as -1 to indicate an empty stack. Before inserting a new element onto a stack, it is necessary to test the condition of overflow. **Overflow** occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element. If a stack is not full then the push operation can be performed successfully. To push an item onto a stack, Top is incremented by one and the element is inserted at that position.

Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow. **Underflow** occurs when a stack is empty and an attempt is made to pop an element. If a stack is not empty, POP operation can be performed successfully. To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one.

The total number of elements in a stack at a given point of time can be calculated from the value of Top as follows:

$$\text{number of elements} = \text{Top} + 1$$

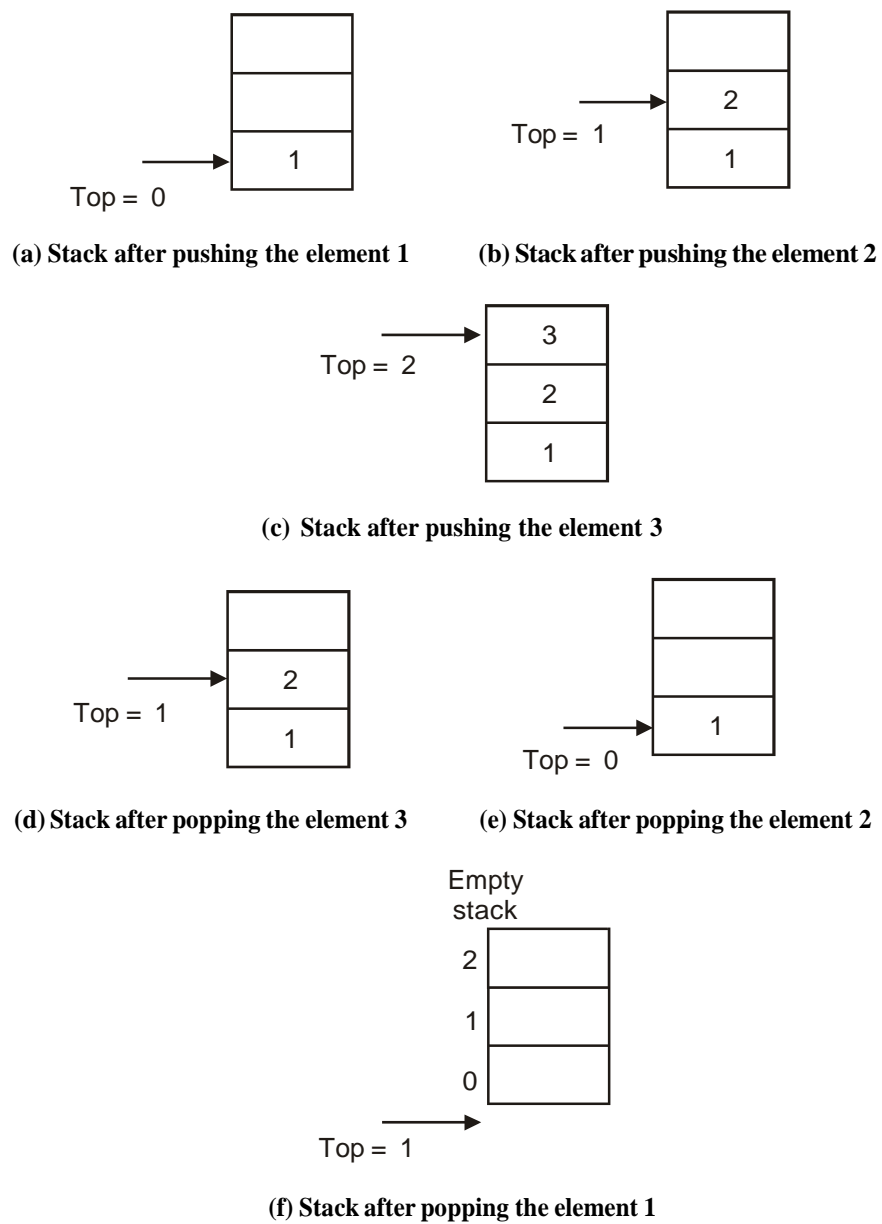
Figure 3.1 shows an empty stack with size 3 and Top = -1.



**Fig. 3.1** An Empty Stack

To insert an element 1 in a stack, Top is incremented by one and the element 1 is stored at stack[Top]. Similarly, other elements can be added to the same stack until Top reaches 2, as shown in Figure 3.2. To POP an element from the stack (data element 3), Top is decremented by one, which removes the element 3 from the stack. Similarly, other elements can be removed from the stack until Top reaches -1. Figure 3.2 shows different states of stack after performing PUSH and POP operations on it.





**Fig. 3.2** Various States of Stack after Push and Pop Operations

To implement a stack as an array in C language, the following structure named Stack needs to be defined as follows:

```
struct stack
{
    int item[MAX];    /*MAX is the maximum size of the
array*/
    int Top;
```

## NOTES

## NOTES

**Algorithm 3.1 Push Operation on Stack**

```

push(s, element)           //s is a pointer to stack

1. If s->Top = MAX-1         //checking for stack overflow
   Print "Overflow: Stack is full!" and go to step 5
   End If
2. Set s->Top = s->Top + 1    //incrementing Top by 1
3. Set s->item[s->Top] = element //inserting element in the stack
4. Print "Value is pushed onto the stack..."
5. End

```

**Algorithm 3.2 Pop Operation on Stack**

```

pop(s)

1. If s->Top = -1            //checking for stack underflow
   Print "Underflow: Stack is empty!"
   Return 0 and go to step 5
   End If
2. Set popped = s->item[s->Top] //taking off the top element from the stack
3. Set s->Top = s->Top - 1     //decrementing Top by 1
4. Return popped
5. End

```

**Example 3.1:** A program to implement a stack as an array is as follows:

```

#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
typedef struct stack
{
    int item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);    /*to create an empty stack*/
void push(stk *, int);      /*to push an element
onto the stack*/
int pop(stk *);             /*to pop the top element from
the stack*/
int isempty(stk *);         /*to check for the underflow
condition*/
int isfull(stk *);          /*to check for the
overflow condition*/
void main()
{
    int choice;
    int value;
    stk s;
    createstack(&s);

```

```

do{
    clrscr();
    printf("\n\tMain Menu");
    printf("\n1. Push");
    printf("\n2. Pop");
    printf("\n3. Exit\n");
    printf("\nEnter your choice: ");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1: printf("\nEnter the value to be inserted:
");
                scanf("%d", &value);
                push(&s, value);
                getch();
                break;
        case 2: value=pop(&s);
                if (value==0)
                    printf("\nUnderflow: Stack is empty!");
                else
                    printf("\nPopped item is: %d", value);
                getch();
                break;
        case 3: exit();
        default: printf("\nInvalid choice!");
    }
}while(1);
}
void createstack(stk *s)
{
    s->Top=-1;
}
void push(stk *s, int element)
{
    if (isfull(s))
    {
        printf("\nOverflow: Stack is full!");
        return;
    }
    s->Top++;
}

```

## NOTES

**NOTES**

```

        s->item[s->Top]=element;
        printf("\nValue is pushed onto the stack...");
    }
int pop(stk *s)
{
    int popped;
    if (isempty(s))
        return 0;
    popped=s->item[s->Top];
    s->Top--;
    return popped;
}
int isempty(stk *s)
{
    if (s->Top==-1)
        return True;
    else return False;
}
int isfull(stk *s)
{
    if (s->Top==MAX-1)
        return True;
    else return False;
}

```

**The output of the program is as follows:**

```

        Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 23
Value is pushed onto the stack...
        Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 35
Value is pushed onto the stack...
        Main Menu
1. Push

```

```
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 40
Value is pushed onto the stack...
```

Main Menu

```
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 40
```

Main Menu

```
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 35
```

Main Menu

```
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 23
```

Main Menu

```
1. Push
2. Pop
3. Exit
Enter your choice: 2
Underflow: Stack is empty!
```

Main Menu

```
1. Push
2. Pop
3. Exit
Enter your choice: 3
```

## NOTES

**NOTES**

---

### 3.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. A stack can be organized (represented) in the memory either as an array or as a singly-linked list.
  2. To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one
- 

### 3.4 SUMMARY

---

- A stack can be organized (represented) in the memory either as an array or as a singly-linked list.
  - Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization.
  - When a stack is organized as an array, a variable named Top is used to point to the top element of the stack.
  - An array representation of a stack is static, but linked list representation is dynamic in nature
  - When a stack is organized as an array, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set as -1 to indicate an empty stack.
  - Overflow occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element.
  - When a stack is organized as an array, a variable named Top is used to point to the top element of the stack.
  - Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow.
  - To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one.
- 

### 3.5 KEY WORDS

---

- **Overflow:** It occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element.
- **Stack:** It is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.

### 3.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. What is a stack?
2. How is a stack organized?
3. What do you mean by overflow in a stack?

#### Long-Answer Questions

1. Write a program to implement a stack as an array.
2. How can you insert elements in a stack? Explain.
3. Write a note on the operations on stack.

#### NOTES

### 3.7 FURTHER READINGS

- Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.
- Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.
- Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.
- Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.
- McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.
- Louise I. Shelly 2020 Dark Commerce

### 3.8 LEARNING OUTCOMES

- Stack and its definition
- The various stack related terms
- The operations on stack

# UNIT 4 REPRESENTATION OF STACK

## NOTES

### Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Application and Implementation of Stack
  - 4.2.1 Converting Infix Notation to Postfix and Prefix or Polish Notations
- 4.3 Answers to Check Your Progress Questions
- 4.4 Summary
- 4.5 Key Words
- 4.6 Self Assessment Questions and Exercises
- 4.7 Further Readings
- 4.8 Learning Outcomes

## 4.0 INTRODUCTION

In the previous unit, you have learnt about stacks. This unit will teach you about representation of stacks. A stack is an abstract data type and it serves as a collection of elements, with two primary principal operations: push, and pop. Push adds an element to the collection and pop removes the most recently added element. As you go through this unit, you will be able to discuss the application and implementation of stacks.

## 4.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain what are stacks
- Discuss the application of stacks
- Analyse the implementation of stacks

## 4.2 APPLICATION AND IMPLEMENTATION OF STACK

Stacks are used where the last-in-first-out principle is required like reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, implementing recursion and function calls, etc. This section discusses some of these applications.



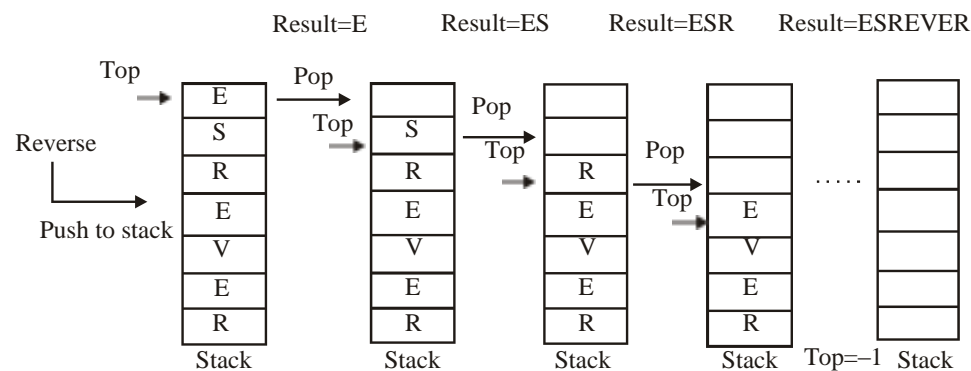
## Reversing Strings

Representation of Stack

A simple application of stacks is reversing strings. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one by one. Since the character last pushed in comes out first, subsequent POP operations result in reversal of the string.

For example, to reverse a string 'REVERSE', the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E, and so on, as shown in Figure 4.1.

## NOTES



**Fig. 4.1** Reversing a String using a Stack

Once all the letters are stored in a stack, they are popped one by one. Since the letter at the top of the stack is E, it is the first letter to be popped. The subsequent POP operations take out the letters S, R, E, and so on. Thus, the resultant string is the reverse of original one as shown in Figure 4.1.

### Algorithm 4.1 String Reversal Using Stack

```
reversal(s, str)
1. Set i = 0
2. While(i < length_of_str)
    Push str[i] onto the stack
    Set i = i + 1
End While
3. Set i = 0
4. While(i < length_of_str)
    Pop the top element of the stack and store it in str[i]
    Set i = i + 1
End While
5. Print "The reversed string is: ", str
6. End
```

**Example 4.1:** The following is a program to reverse a given string using stacks:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 101
typedef struct stack
```

**NOTES**

```

{
    char item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void reversal(stk *, char *);
void push(stk *, char);
char pop(stk *);
void main()
{
    stk s;
    char str[MAX];
    int i;
    createstack(&s);
    clrscr();
    do
    {
        printf("Enter any string (max %d characters): ",
MAX-1);
        for(i=0;i<MAX;i++)
        {
            scanf("%c", &str[i]);
            if(str[i]=='\n')
                break;
        }
        str[i]='\0';
    }while(strlen(str)==0);
    reversal(&s, str);
    getch();
}
/*Function definitions*/
void createstack(stk *s)
{
    s->Top=-1;
}
void reversal(stk *s, char *str)
{
    int i;
    for (i=0;i<strlen(str);i++)

```

```

        push(s, str[i]);
    for(i=0;i<strlen(str);i++)
        str[i]=pop(s);
    printf("\nThe reversed string is: %s", str);
}
void push(stk *s, char item)
{
    s->Top++;
    s->item[s->Top]=item;
}
char pop(stk *s)
{
    char popped;
    popped=s->item[s->Top];
    s->Top--;
    return popped;
}

```

**The output of the program is as follows:**

```

Enter any string (max 100 characters): Hello World
The reversed string is: dlroW olleH

```

#### 4.2.1 Converting Infix Notation to Postfix and Prefix or Polish Notations

Another important application of stacks is the conversion of expressions from infix notation to postfix and prefix notations. The general way of writing arithmetic expressions is known as **infix notation** where the binary operator is placed between two operands on which it operates. For simplicity, expressions containing unary operators have been ignored. For example, the expressions ‘a+b’ and ‘(a–c)\*d’, ‘[(a+b)\*(d/f)–f]’ are in infix notation. The order of evaluation of these expressions depends on the parentheses and the precedence of operators. For example, the order of evaluation of the expression ‘(a+b)\*c’ is different from that of ‘a+(b\*c)’. As a result, it is difficult to evaluate an expression in an infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation which can be easily evaluated by a computer system to produce a correct result.

A Polish mathematician Jan Lukasiewicz suggested two alternative notations to represent an arithmetic expression. In these notations, the operators can be written either before or after the operands on which they operate. The notation in which an operator occurs before its operands is known as the **prefix notation** (also known as **Polish notation**). For example, the expressions ‘+ab’ and ‘\*–acd’ are in prefix notation. On the other hand, the notation in which an operator occurs after its operands is known as the **postfix notation** (also known as **Reverse Polish** or **suffix notation**). For example, the expressions ‘ab+’ and ‘ac–d\*’ are in postfix notation.

## NOTES

**NOTES**

A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression. That is, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of operators is insignificant. As a result, they are compiled faster than the expressions in infix notation. Note that the expressions in an infix notation can be converted to both prefix and postfix notations. The subsequent sections will discuss both the types of conversions.

**Conversion of infix to postfix notation**

To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators should always be kept in mind. The operators of the same precedence are evaluated from left to right. This conversion can be performed either manually (without using stacks) or by using stacks. Following are the steps for converting the expression manually:

- (i) The actual order of evaluation of the expression in infix notation is determined. This is done by inserting parentheses in the expression according to the precedence and associativity of operators.
- (ii) The expression in the innermost parentheses is converted into postfix notation by placing the operator after the operands on which it operates.
- (iii) Step 2 is repeated until the entire expression is converted into a postfix notation.

For example, to convert the expression ' $a+b*c$ ' into an equivalent postfix notation, the steps will be as follows:

- (i) Since the precedence of  $*$  is higher than  $+$ , the expression  $b*c$  has to be evaluated first. Hence, the expression is written as follows:  
 $(a + (b * c))$
- (ii) The expression in the innermost parentheses, that is,  $b*c$  is converted into its postfix notation. Hence, it is written as  $bc*$ . The expression now becomes as follows:  
 $(a + bc*)$
- (iii) Now the operator  $+$  has to be placed after its operands. The two operands for  $+$  operator are  $a$  and the expression  $bc*$ . The expression now becomes as follows:  
 $(abc*+)$

Hence, the equivalent postfix expression will be as follows:

$abc*+$

When expressions are complex, manual conversion becomes difficult. On the other hand, the conversion of an infix expression into a postfix expression is simple when it is implemented through stacks. In this method, the infix expression

NOTES

is read from left to right and a stack is used to temporarily store the operators and the left parenthesis. The order in which the operators are pushed on to and popped from the stack depends on the precedence of operators and the occurrence of parentheses in the infix expression. The operands in the infix expression are not pushed on to the stack, rather they are directly placed in the postfix expression. Note that the operands maintain the same order as in the original infix notation.

Algorithm 4.2 Infix to Postfix Conversion

```
infixtopostfix(s, infix, postfix)
1. Set i = 0
2. While (i < number_of_symbols_in_infix)
    If infix[i] is a whitespace or comma
        Set i = i + 1 and go to step 2
    If infix[i] is an operand, add it to postfix
    Else If infix[i] = '(', push it onto the stack
    Else If infix[i] is an operator, follow these steps:
        i. For each operator on the top of stack whose precedence is greater
           than or equal to the precedence of the current operator, pop the
           operator from stack and add it to postfix
        ii. Push the current operator onto the stack
    Else If infix[i] = ')', follow these steps:
        i. Pop each operator from top of the stack and add it to postfix
           until '(' is encountered in the stack
        ii. Remove '(' from the stack and do not add it to postfix
    End If
    Set i = i + 1
End While
3. End
```

For example, consider the conversion of the following infix expression to a postfix expression:

a - (b + c) \* d / f

Initially, a left parenthesis ‘ ( ’ is pushed onto the stack and the infix expression is appended with a right parenthesis, ‘ ) ’. The initial state of the stack, infix expression and postfix expression are shown in Figure 4.2.

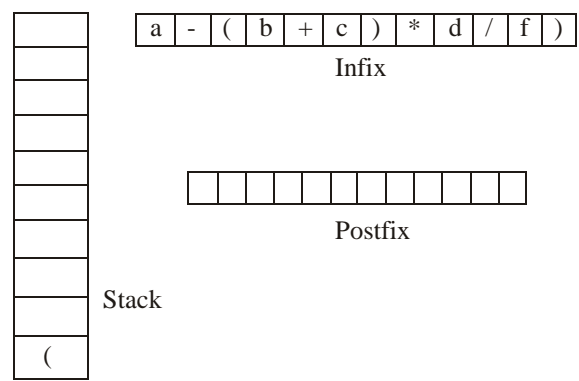
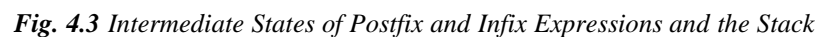


Fig. 4.2 Initial State of the Stack, Infix Expression, and Postfix Expression

infix is read from left to right and the following steps are performed:

- 1. The operand a is encountered, which is directly put to postfix.
- 2. The operator – is pushed on to the stack.

3. The left parenthesis '(' is pushed onto the stack.
4. The next element is b, which being an operand is directly put to postfix.
5. +, being an operator, is pushed onto the stack.
6. Next, c is put to postfix.
7. The next element is the right parenthesis ')' and hence, the operators at the top of the stack are popped until '(' is encountered in the stack. Till then, the only operator in the stack above the '(' is +, which is popped and put to postfix. '(' is popped and removed from the stack, as shown in Figure 4.3(a). Figure 4.3(b) shows the current position of stack.



**Fig. 4.3** *Intermediate States of Postfix and Infix Expressions and the Stack*

8. Then, the next element  $*$  is an operator and hence, it is pushed onto the stack.
9. Then,  $d$  is put to `postfix`.
10. The next element is  $/$ . Since the precedence of  $/$  is same as the precedence of  $*$ , the operator  $*$  is popped from the stack and  $/$  is pushed onto the stack, as shown in Figure 4.4.
11. The operand  $f$  is directly put to `postfix` after which,  $'$  is encountered.
12. On reaching  $'$ , the operators in stack before the next  $'$  is reached and popped. Hence,  $/$  and  $-$  are popped and put to `postfix` as shown in Figure 4.4.

13. '(' is removed from the stack. Since the stack is empty, the algorithm is terminated and postfix is printed.

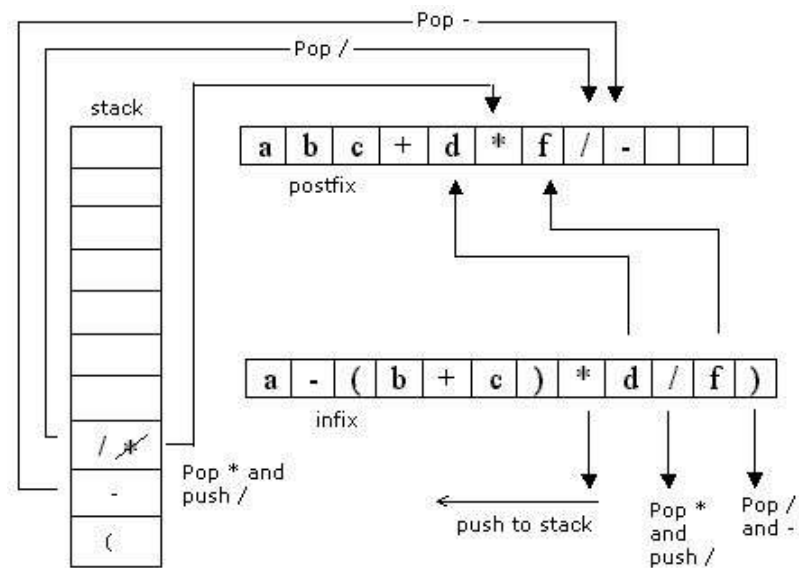


Fig. 4.4 The State when - and / are Popped

The step-wise conversion of expression  $a - (b + c) * d / f$  into its equivalent postfix expression is shown in Table 4.1.

Table 4.1 Conversion of Infix Expression into Postfix

Element	Action Performed	Stack Status	Postfix Expression
A	Put to postfix	(	A
-	Push	( -	a
(	Push	( - (	a
b	Put to postfix	( - (	ab
+	Push	( - ( +	ab
c	Put to postfix	( - ( +	abc
)	Pop +, put to postfix, pop (	( -	abc+
*	Push	( - *	abc+
d	Put to postfix	( - *	abc+d
/	Pop *, put to postfix, push /	( - /	abc+d*
f	Put to postfix	( - /	abc+d*f
)	Pop / and -	Empty	abc+d*f/-

### Conversion of infix to prefix notation

The conversion of an infix expression to a prefix expression is similar to the conversion of infix to postfix expression. The only difference is that the expression in an infix notation is scanned in reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

### NOTES

## NOTES

## Algorithm 4.3 Infix to Prefix Conversion

```

infixtoprefix(s, infix, prefix)
1. Set i = 0
2. While (i < number_of_symbols_in_infix)
   If infix[i] is a whitespace or comma
       Set i = i + 1 go to step 2
   If infix[i] is an operand, add it to prefix
   Else If infix[i] = ')', push it onto the stack
   Else If infix[i] is an operator, follow these steps:
       i. For each operator on the top of stack whose precedence is greater
          than or equal to the precedence of the current operator, pop the
          operator from stack and add it to prefix
       ii. Push the current operator onto the stack
   Else If infix[i] = '(', follow these steps:
       i. Pop each operator from top of the stack and add it to prefix until
          ')' is encountered in the stack
       ii. Remove ')' from the stack and do not add it to prefix
   End If
   Set i = i + 1
End While
3. Reverse the prefix expression
4. End

```

For example, consider the conversion of the following infix expression to a prefix expression:

$$a - (b + c) * d / f$$

The step-wise conversion of the expression  $a - (b + c) * d / f$  into its equivalent prefix expression is shown in Table 4.2. Note that initially ')' is pushed onto the stack, and '(' is inserted in the beginning of the infix expression. Since the infix expression is scanned from right to left, but elements are inserted in the resultant expression from left to right, the prefix expression needs to be reversed.

**Table 4.2** Conversion of Infix Expression into Prefix Expression

Element	Action Performed	Stack Status	Prefix Expression
f	Put to expression	)	f
/	Push	) /	f
d	Put to expression	) /	fd
*	Push	) / *	fd
)	Push	) / *)	fd
c	Put to expression	) / *)	fdc
+	Push	) / *) +	fdc
b	Put to expression	) / *) +	fdcb
(	Pop and + and put to expression, pop )	) / *	fdcb+
-	Pop *, / and push -	) -	fdcb+*/
a	Put to expression	) / *-	fdcb+a
(	Pop - and put to expression, pop (	Empty	fdcb+*/a-
	Reverse the resultant expression		-a/*+bcd f

The equivalent prefix expression is  $-a/*+bcd f$ .

### Evaluation of Postfix Expression

In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation. The equivalent postfix expression is then evaluated. Evaluation of postfix expressions is also implemented



through stacks. Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression. During evaluation, a stack is used to store the intermediate results of evaluation.

Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right. Each element in the expression is checked to find out whether it is an operator or an operand. If the element is an operand, it is pushed onto the stack. On the other hand, if the element is an operator, the first two operands are popped from the stack and an operation is performed on them. The result of this operation is then pushed back to the stack. This process is repeated until the entire expression is evaluated.

## NOTES

### Algorithm 4.4 Evaluation of a Postfix Expression

```
evaluationofpostfix(s, postfix)
1. Set i = 0, RES=0.0
2. While (i < number_of_characters_in_postfix)
   If postfix[i] is a whitespace or comma
       Set i = i + 1 and continue
   If postfix[i] is an operand, push it onto the stack
   If postfix[i] is an operator, follow these steps:
       i. Pop the top element from stack and store it in operand2
       ii. Pop the next top element from stack and store it in operand1
       iii. Evaluate operand2 op operand1, and store the result in
           RES (op is the current operator)
       iv. Push RES back to stack
   End If
   Set i = i + 1
End While
3. Pop the top element and store it in RES
4. Return RES
5. End
```

For example, consider the evaluation of the following postfix expression using stacks:

$abc+d*f/-$

where,

$a=6$

$b=3$

$c=6$

$d=5$

$f=9$

After substituting the values of  $a$ ,  $b$ ,  $c$ ,  $d$  and  $f$ , the postfix expression becomes as follows:

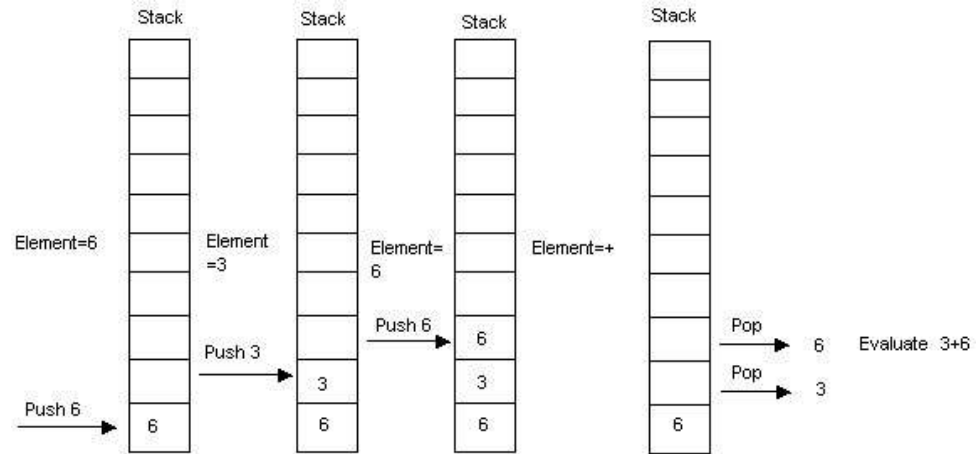
$636+5*9/-$

The following are the steps performed to evaluate an expression:

1. The expression to be evaluated is read from left to right and each element is checked to find out if it is an operand or an operator.
2. First element is 6, which being an operand is pushed onto the stack.

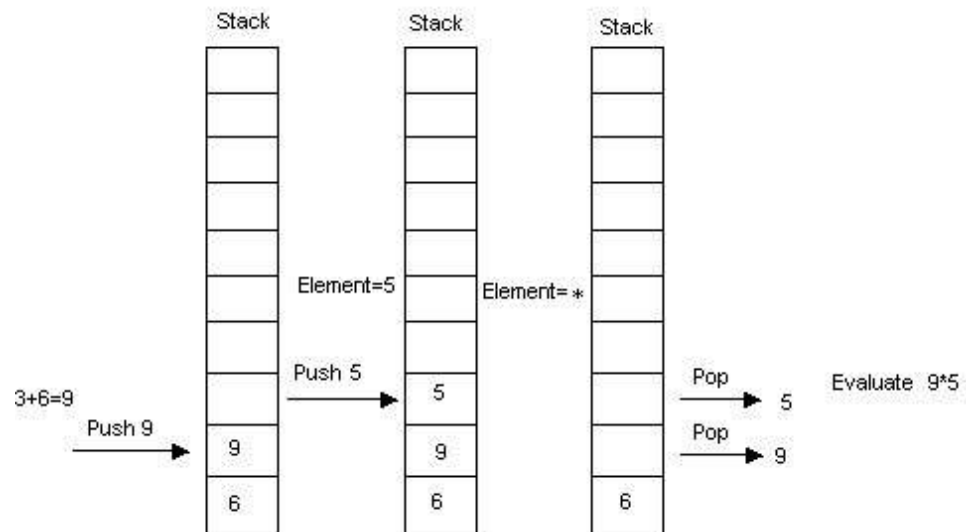
## NOTES

3. Similarly, the operands 3 and 6 are pushed onto the stack.
4. Next element is +, which is an operator. Hence, the element at the top of stack 6 and the next top element 3 are popped from the stack, as shown in Figure 4.5.



**Fig. 4.5** Evaluation of the Expression using Stacks

5. Expression 3+6 is evaluated and the result, that is 9, is pushed back to stack, as shown in Figure 4.6.
6. Next element in the expression, that is 5, is pushed to the stack.
7. Next element is \*, which is a binary operator. Hence, the stack is popped twice and the elements 5 and 9 are taken off from the stack, as shown in Figure 4.6.



**Fig. 4.6** Popping 9 and 5 from the Stack

8. Expression 9\*5 is evaluated and the result, that is 45, is pushed to the back of the stack.

- 9. Next element in the postfix expression is 9, which is pushed onto the stack.
- 10. Next element is the operator /. Therefore, the two operands from the top of the stack, that is 9 and 45, are popped from the stack and the operation  $45 / 9$  is performed. Result 5 is again pushed to the stack.
- 11. Next element in the expression is -. Hence, 5 and 6 are popped from the stack and the operation  $6 - 5$  is performed. The resulting value, that is 1, is pushed to the stack (see Figure 4.7).

NOTES

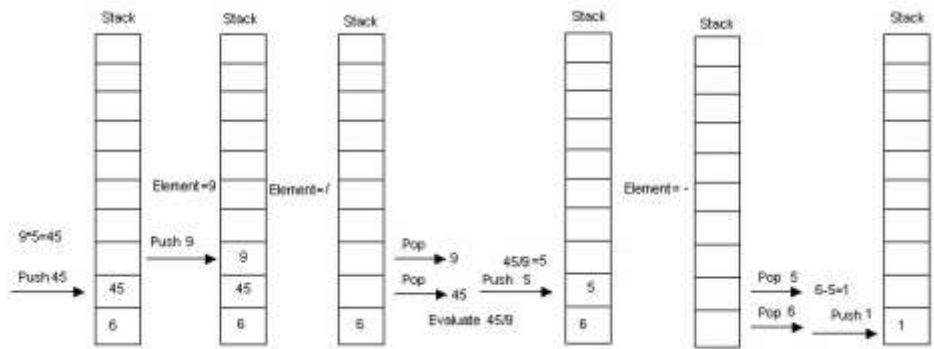


Fig. 4.7 Final State of Stack with the Result

- 12. There are no more elements to be processed in the expression. Element on top of the stack is popped, which is the result of the evaluation of the postfix expression. Thus, the result of the expression is 1.

The step-wise evaluation of the expression  $6\ 3\ 6 + 5 * 9 / -$  is shown in Table 4.3.

Table 4.3 Evaluation of the Postfix Expression

Element	Action Performed	Stack Status
6	Push to stack	6
3	Push to stack	6 3
6	Push to stack	6 3 6
+	Pop 6	6 3
	Pop 3	6
	Evaluate $3+6=9$	6
	Push 9 to stack	6 9
5	Push to stack	6 9 5
*	Pop 5	6 9
	Pop 9	6
	Evaluate $9*5=45$	6
	Push 45 to stack	6 45
9	Push to stack	6 45 9
/	Pop 9	6 45
	Pop 45	6
	Evaluate $45/9=5$	6
	Push 5 to stack	6 5
-	Pop 5	6
	Pop 6	EMPTY
	Evaluate $6-5=1$	EMPTY
	Push 1 to stack	1
	Pop VALUE=1	EMPTY

NOTES

Multi-stacks

So far, programs containing a single stack have been discussed in the unit. If two or more stacks are needed in a program, then it can be accomplished in two ways. One way is to have a separate array for each stack in the program. This approach has a disadvantage—if one stack needs to store larger number of elements than the specified size and the other stack has lesser number of elements—it is not possible to store the elements of first stack in the second stack, in spite of the vacant space. This problem can be solved by another way, that is, by having a single array of sufficient size to hold two or more stacks. The two stacks can be represented efficiently in the same array, provided, one stack grows from left to right and other grows from right to left. Also, the memory is utilized more efficiently in this case.

For example, consider an array `stack[MAX]` to hold two stacks, `stack1` and `stack2`. Two top variables `Top1` and `Top2` are required to represent the top of the two stacks. Initially, to represent the empty stacks, `Top1` is set as `-1` and `Top2` is set as `MAX`. In this case, the condition of overflow occurs when the combined size of both the stacks exceed `MAX`. For this, variable `count` is used that keeps track of the number of elements stored in the array. Initially, `count` is set to `0`. Overflow occurs when the value of `count` exceeds the value of `MAX` and an attempt is made to insert a new element.

To PUSH an element in `stack1`, `Top1` is incremented by 1 and the element is inserted in that position. On the other hand, to PUSH an element in `stack2`, `Top2` is decremented by 1 and the element is inserted in that position. To POP an element from `stack1`, the element at the position indicated by `Top1` is assigned to a local variable and then `Top1` is decremented by 1. On the other hand, to POP an element from `stack2`, the element at the position indicated by `Top2` is assigned to a local variable and then `Top2` is incremented by 1. Figure 4.9 shows an array of size `MAX` to hold two stacks `stack1` and `stack2`, where `stack1` grows from left to right and `stack2` grows from right to left.

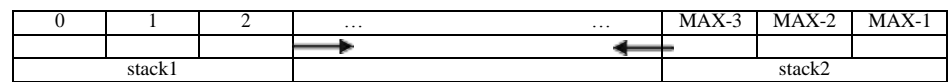


Fig. 4.8 Representing Two Stacks by an Array of Size MAX

To represent two stacks in the same array, the following structure called `multistack` needs to be defined in C language:

```
struct multistack
{
    int item[MAX];
    int Top1, Top2;
    int count;
    int sno;    /*sno indicates the stack number (1 or
```

```
2) */
};
```

Representation of Stack

#### Algorithm 4.5 PUSH Operation on Multi-Stack

```
push(s, element)           //s is a pointer to multi-stack

1. If (s->count == MAX)
    Print "Overflow: Stack is full!" and go to step 4
End If
2. If(s->sno == 1) //if element is to be inserted in stack1
    Set s->Top1 = s->Top1 + 1
    Set s->item[s->Top1] = element
    Print "Value is pushed onto stack1..."
    Set s->count = s->count + 1
End If
3. If(s->sno == 2)
    Set s->Top2 = s->Top2 - 1
    Set s->item[s->Top2] = element
    Print "Value is pushed onto stack2..."
    Set s->count = s->count + 1
End If
4. End
```

## NOTES

#### Algorithm 4.6 POP Operation on Multi-Stack

```
pop(s)                     //s is a pointer to multi-stack

1. If(s->sno == 1)
    If (s->Top1 == -1)
        Print "Underflow! Stack1 is empty"
        Return 0 and go to step 4
    Else
        Set popped = s->item[s->Top1]
        Set s->Top1 = s->Top1 - 1
        Set s->count = s->count - 1
    End If
End If
2. If(s->sno == 2)
    If (s->Top2 == MAX)
        Print "Underflow! Stack2 is empty"
        Return 0 and go to step 4
    Else
        Set popped = s->item[s->Top2]
        Set s->Top2 = s->Top2 + 1
        Set s->count = s->count - 1
    End If
End If
3. Return popped
4. End
```

**Example 4.2:** A program to implement multi-stacks using a single array is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
typedef struct multistack
{
    int item[MAX];
    int Top1, Top2;
    int count;
```

**NOTES**

```

        int sno;      /*sno is the stack number (1 or 2)*/
    }mstk;
    /*Function prototypes*/
    void createstack(mstk *);
    void push(mstk *, int);
    int pop(mstk *);
    int isempty(mstk *);
    int isfull(mstk *);
    void main()
    {
        int choice;
        int value;
        mstk s;
        createstack(&s);
        do{
            clrscr();
            printf("\n\tMain Menu");
            printf("\n1. Push");
            printf("\n2. Pop");
            printf("\n3. Exit\n");
            printf("\nEnter your choice: ");
            scanf("%d", &choice);
            switch(choice)
            {
                case 1: printf("\nEnter the stack number (1 or
2): ");
                        scanf("%d", &s.sno);
                        printf("\nEnter the value to be inserted:
");
                        scanf("%d", &value);
                        push(&s, value);
                        getch();
                        break;
                case 2: printf("\nEnter the stack number (1 or
2): ");
                        scanf("%d", &s.sno);
                        value=pop(&s);
                        if (value==0)
                        {
                            if(s.sno==1)
                                printf("\nUnderflow: Stack1 is

```

```

empty!");
                else if(s.sno==2)
                    printf("\nUnderflow: Stack2 is
empty!");
            }
            else
                printf("\nPopped element is: %d", value);
            getch();
            break;
        case 3: exit();
        default: printf("\nInvalid choice!");
    }
}while(1);
}
/*Function definitions*/
void createstack(mstk *s)
{
    s->Top1=-1;
    s->Top2=MAX;
    s->count=0;
}
void push(mstk *s, int item)
{
    if (isfull(s))
    {
        printf("\nOverflow: Stack is full!");
        return;
    }
    if(s->sno==1)
    {
        s->Top1++;
        s->item[s->Top1]=item;
        printf("\nValue is pushed onto stack1...");
        s->count++;
    }
    if(s->sno==2)
    {
        s->Top2--;
        s->item[s->Top2]=item;
        printf("\nValue is pushed onto stack2...");
    }
}

```

*Representation of Stack*

## NOTES

**NOTES**

```
s->count++;  
}  
}  
int pop(mstk *s)  
{  
    int popped;  
    if (isempty(s))  
        return 0;  
    if (s->sno==1)  
    {  
        popped=s->item[s->Top1];  
        s->Top1-;  
        s->count-;  
    }  
    if (s->sno==2)  
    {  
        popped=s->item[s->Top2];  
        s->Top2++;  
        s->count-;  
    }  
    return popped;  
}  
int isempty(mstk *s)  
{  
    int r;  
    if (s->sno==1)  
    {  
        if (s->Top1==-1)  
            r=True;  
        else  
            r=False;  
    }  
    if (s->sno==2)  
    {  
        if (s->Top2==MAX)  
            r=True;  
        else  
            r=False;  
    }  
    return r;  
}
```



```

}
int isfull(mstk *s)
{
    if (s->count==MAX)
        return True;
    else return False;
}

```

*Representation of Stack*

## NOTES

**The output of the program is as follows:**

```

Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 1
Enter the value to be inserted: 34
Value is pushed onto stack1...

Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 2
Enter the value to be inserted: 45
Value is pushed onto stack2...

Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the stack number (1 or 2): 1
Enter the value to be inserted: 23
Value is pushed onto stack1...

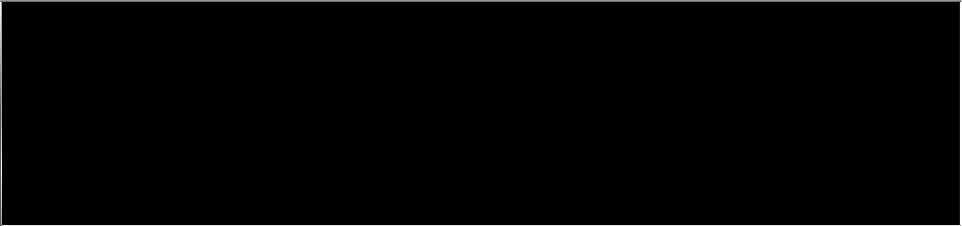
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Popped element is: 23

Main Menu

```

## NOTES

```
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 2
Popped element is: 45
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Popped element is: 34
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 1
Underflow: Stack1 is empty!
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Enter the stack number (1 or 2): 2
Underflow: Stack2 is empty!
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3
```



---

### 4.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression.
2. Stacks are used where the last-in-first-out principle is required like reversing strings.
3. During evaluation, a stack is used to store the intermediate results of evaluation.

### NOTES

---

### 4.4 SUMMARY

---

- Stacks are used where the last-in-first-out principle is required like reversing strings.
- A simple application of stacks is reversing strings. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right.
- Once all the characters of the string are pushed onto the stack, they are popped one by one.
- Since the character last pushed in comes out first, subsequent POP operations result in reversal of the string.
- The general way of writing arithmetic expressions is known as infix notation where the binary operator is placed between two operands on which it operates.
- A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression.
- To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators should always kept in mind.
- The conversion of an infix expression to a prefix expression is similar to the conversion of infix to postfix expression.
- In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation.
- Evaluation of postfix expressions is also implemented through stacks.

## NOTES

- Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression.
- During evaluation, a stack is used to store the intermediate results of evaluation.
- Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right.
- If the element is an operand, it is pushed onto the stack.
- If two or more stacks are needed in a program, then it can be accomplished in two ways.

---

### 4.5 KEY WORDS

---

- **Stack:** It is an abstract data type that serves as a collection of elements, with two principal operations- push and pop.
- **Reversing Strings:** It is a simple application of stacks. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right.

---

### 4.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. Write the algorithm for reversing a string.
2. Write a short note on stacks.
3. Discuss the application of stacks.

#### Long-Answer Questions

1. What do you mean by implementation of stack? Discuss in detail.
2. Write a program to reverse a given string using stacks.
3. Write a detailed note on Conversion of infix to postfix notation.
4. Write a program to convert an expression from infix notation to postfix notation.

---

### 4.7 FURTHER READINGS

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

*Representation of Stack*

## NOTES

### 4.8 LEARNING OUTCOMES

- What are stacks
- The application of stacks
- The implementation of stacks

---

## UNIT 5 QUEUES

---

### NOTES

#### Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Queues
- 5.3 Representation of Queues
- 5.4 Circular Queue and Deque
- 5.5 Priority Queue
- 5.6 Applications of Queues
- 5.7 Answers to Check Your Progress Questions
- 5.8 Summary
- 5.9 Key Words
- 5.10 Self Assessment Questions and Exercises
- 5.11 Further Readings
- 5.12 Learning Outcomes

---

### 5.0 INTRODUCTION

In this unit, you will learn about the queues, their representation and applications. A Queue is an abstract data structure which is somewhat similar to Stacks. But unlike stacks, a queue is open at both its ends. One end of a queue is always used to insert data (called enqueue) and the other is used to remove data (called dequeue). Queue follows the basic and simple First-In-First-Out methodology, which means that the data item stored first will be accessed first.

---

### 5.1 OBJECTIVES

After going through this unit, you will be able to:

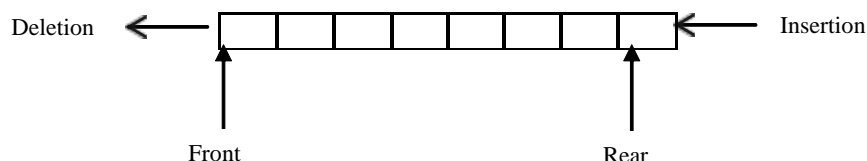
- Understand queues
- Discuss the representation of queues
- Analyze circular queues and deque
- Explain about priority queues
- List the applications of queue

---

### 5.2 QUEUES

A **queue** is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end. The end of the queue from which the

element is deleted is known as the **Front** and the end at which a new element is added is known as the **Rear**. Figure 5.1 shows a queue.



*Fig. 5.1 A Queue*

The following are the basic operations that can be performed on queues:

- **Insert Operation:** To insert an element at the rear of the queue
- **Delete Operation:** To delete an element from the front of the queue

Before inserting a new element in the queue, it is necessary to check whether there is space for the new element. If no space is available, the queue is said to be in the condition of overflow. Similarly, before deleting an element from the queue, it is necessary to check whether there is an element in the queue. If there is no element in the queue, the queue is said to be in the condition of underflow.

### 5.3 REPRESENTATION OF QUEUES

Like stacks, queues can be represented in the memory by using an array or a singly linked list. In this section, we will discuss how a queue can be implemented using an array.

#### Array Implementation of a Queue

When a queue is implemented as an array, all the characteristics of an array are applicable to the queue. Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed. As we know that a queue keeps on changing as elements are inserted or deleted, the maximum size should be large enough for a queue to expand or shrink.

The representation of a queue as an array needs an array to hold the elements of the queue and two variables `Rear` and `Front` to keep track of the rear and the front ends of the queue, respectively. Initially, the value of `Rear` and `Front` is set to `-1` to indicate an empty queue. Before we insert a new element in the queue, it is necessary to test the condition of overflow. A queue is in a condition of overflow (full) when `Rear` is equal to `MAX-1`, where `MAX` is the maximum size of the array. If the queue is not full, the insert operation can be performed. To

#### NOTES

**NOTES**

insert an element in the queue, `Rear` is incremented by one and the element is inserted at that position.

Similarly, before we delete an element from a queue, it is necessary to test the condition of underflow. A queue is in the condition of underflow (empty) when the value of `Front` is `-1`. If a queue is not empty, the delete operation can be performed. To delete an element from a queue, the element referred by `Front` is assigned to a local variable and then `Front` is incremented by one.

The total number of elements in a queue at a given point of time can be calculated from the values of `Rear` and `Front` given as follows:

$$\text{Number of elements} = \text{Rear} - \text{Front} + 1$$

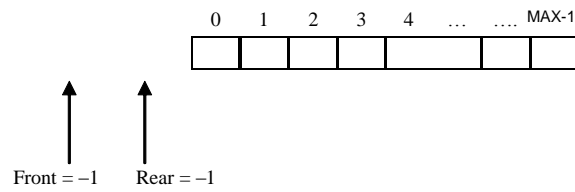
To understand the implementation of a queue as an array in detail, consider a queue stored in the memory as an array named `Queue` that has `MAX` as its maximum number of elements. `Rear` and `Front` store the indices of the rear and front elements of `Queue`. Initially, `Rear` and `Front` are set to `-1` to indicate an empty queue (refer Figure 5.2(a)).

Whenever a new element has to be inserted in a queue, `Rear` is incremented by one and the element is stored at `Queue[Rear]`. Suppose an element 9 is to be inserted in the queue. In this case, the rear is incremented from `-1` to `0` and the element is stored at `Queue[0]`. Since it is the first element to be inserted, `Front` is also incremented by one to make it to refer to the first element of the queue (refer Figure 5.2(b)). For subsequent insertions, the value of `Rear` is incremented by one and the element is stored at `Queue[Rear]`. However, `Front` remains unchanged (refer Figure 5.2(c)). Observe that the front and rear elements of the queue are the first and last elements of the list, respectively.

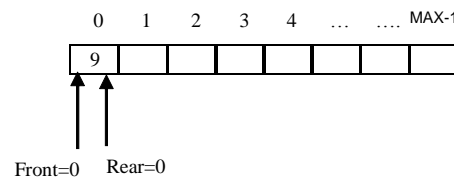
Whenever, an element is to be deleted from a queue, `Front` is incremented by one. Suppose that an element is to be deleted from `Queue`. Then, here it must be 9. It is because the deletion is always made at the front end of a queue. Deletion of the first element results in the queue as shown in Figure 5.2(d). Similarly, deletion of the second element results in the queue as shown in Figure 5.2(e). Observe that after deleting the second element from the queue, the values of `Rear` and `Front` are equal. Here, it is apparent that when values of `Front` and `Rear` are equal other than `-1`, there is only one element in the queue. When this only element of the queue is deleted, both `Rear` and `Front` are again made equal to `-1` to indicate an empty queue.

Further, suppose that some more elements are inserted and `Rear` reaches the maximum size of the array (refer Figure 5.2(f)). This means that the queue is full and no more elements can be inserted in it even though the space is vacant on the left of the `Front`.

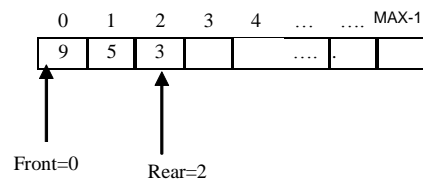




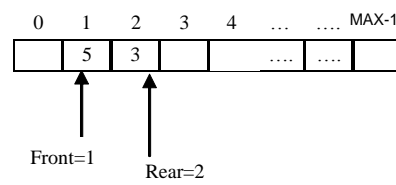
(a) An Empty Queue



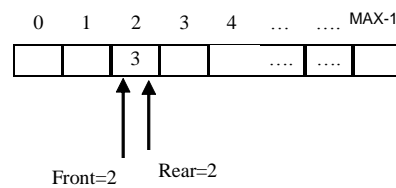
(b) Queue after Inserting the First Element



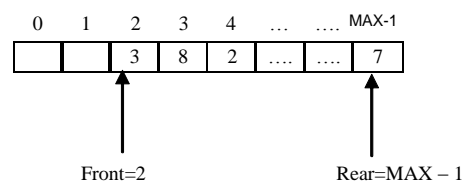
(c) Queue after Inserting a few Elements



(d) Queue after Deleting the First Element



(e) Queue after Deleting the Second Element



(f) Queue having Vacant Space though Rear = MAX - 1

**Fig. 5.2** Various States of a Queue after the Insert and Delete Operations**NOTES**

## NOTES

To implement a queue as an array in the C language, the following structure named queue is used:

```
struct queue
{
    int item[MAX];
    int Front;
    int Rear;
};
```

**Algorithm 5.1 Insert Operation on a Queue**

qinsert(q, val) //q is a pointer to structure type queue and val is the value to be inserted

1. If q->Rear = MAX-1 //check if queue is full  
Print "Overflow: Queue is full!" and go to step 5  
End If
2. If q->Front = -1 //check if queue is empty  
Set q->Front = 0 // make front to refer to first element  
End If
3. Set q->Rear = q->Rear + 1 //increment Rear by one
4. Set q->item[q->Rear] = val //insert val
5. End

**Algorithm 5.2 Delete Operation on a Queue**

qdelete(q)

1. If q->Front = -1 //check if queue is empty  
Print "Underflow: Queue is empty!"  
Return 0 and go to step 5  
End If
2. Set del\_val = q->item[q->Front] //del\_val is the value to be deleted
3. If q->Front = q->Rear //check if there is only one element  
Set q->Front = q->Rear = -1  
Else  
Set q->Front = q->Front + 1 //increment Front by oneEnd  
If
4. Return del\_val
5. End

**Linked Implementation of a Queue**

A queue implemented as a linked list is known as a **linked queue**. A linked queue is represented using two pointer variables `Front` and `Rear` that point to the first and the last node of the queue, respectively. Initially, `Rear` and `Front` are set to `NULL` to indicate an empty queue.

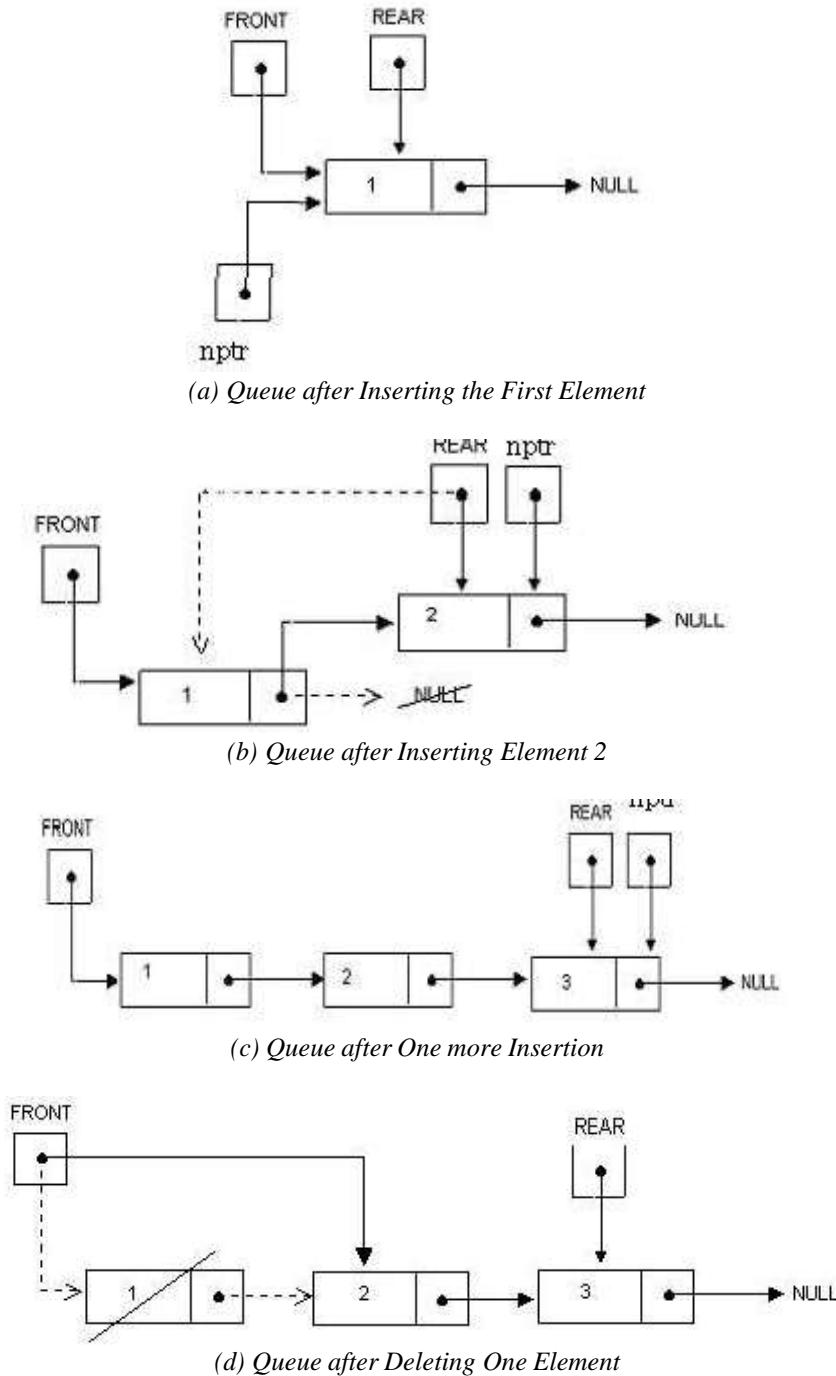
To understand the implementation of a linked queue, consider a linked queue, say `Queue`. The `info` and `next` fields of each node represent the element of the queue and a pointer to the next element in the queue, respectively. Whenever a new element is to be inserted in the queue, a new node `nptr` is created and the element is inserted into the node. If it is the first element being inserted in the queue, both `Front` and `Rear` are modified to point to this new node. On the other hand, in subsequent insertions, only `Rear` is modified to point to the new node; `Front` remains unchanged.

Whenever an element is deleted from the queue, a temporary pointer is created, which is made to point to the node pointed to by `Front`. Then `Front`

is modified to point to the next node in the queue, and the temporary node is deleted from the memory. Figure 5.3 shows the various states of a queue after the insert and delete operations.

**Note:** Since the memory is allocated dynamically, a linked queue reaches the overflow condition when no more free memory space is available to be dynamically allocated.

## NOTES



**Fig. 5.3** Various states of a Linked Queue after the Insert and Delete Operations

## NOTES

**Algorithm 5.3 Insert Operation on a Linked Queue**

qinsert(q, val) //val is the value to be inserted

1. Allocate memory for nptr //nptr is a pointer to the new node to be inserted
2. If nptr = NULL // checking for queue overflow  
Print "Overflow: Memory not allocated!" and go to step 6  
End If
3. Set nptr->info = val
4. Set nptr->next = NULL
5. If Front = NULL //check if queue is empty  
Set q->Rear = q->Front = nptr //rear and front are made to point to new node  
//node  
Else  
Set q->Rear->next = nptr  
Set q->Rear = nptr //Rear is made to point to new node  
End If
6. End

**Algorithm 5.4 Delete Operation on a Linked Queue**

qdelete(q)

1. If Front = NULL  
Print "Underflow: Queue is empty!"  
Return 0 and go to step 7  
End if
2. Set del\_val = q->Front->info //del\_val is the element pointed by the Front
3. Set temp = q->Front //temp is the temporary pointer to Front
4. If q->Front = q->Rear //checking if there is one element in the queue  
Set q->Front = q->Rear = NULL  
Else  
Set q->Front = q->Front->next //making Front point to next node  
End If
5. De-allocate temp //de-allocating memory
6. Return del\_val
7. End

---

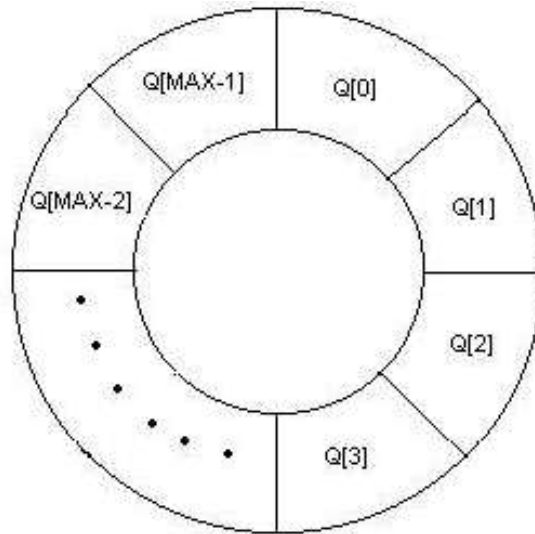
**5.4 CIRCULAR QUEUE AND DEQUE**

---

As discussed earlier, in the case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted. However, there may be the possibility that the space on the left of the front index is vacant. Hence, in spite of space on the left of front being empty, the queue is considered full. This wastage of space in the array implementation of a queue can be avoided by shifting the elements to the beginning of the array if space is available. In order to do this, the values of the Rear and Front indices have to be changed accordingly. However, this is a complex process and difficult to implement. An alternative solution to this problem is to implement a queue as a circular queue.

The array implementation of a circular queue is similar to the array implementation of the queue. The only difference is that as soon as the rear index

of the queue reaches the maximum size of the array, `Rear` is reset to the beginning of the queue, provided it is free. The circular queue is full only when all the locations in the array are occupied. A circular queue is shown in Figure 5.4.



**Fig. 5.4** A Circular Queue

**Note:** A circular queue is generally implemented as an array though it can also be implemented as a circular linked list.

To understand the operations on a circular queue, consider a circular queue represented in the memory by the array `CQueue [MAX]`. `Rear` and `Front` are used to store the indices of the rear and front elements of `CQueue`, respectively. Initially, both `Rear` and `Front` are set to `NULL` to indicate an empty queue.

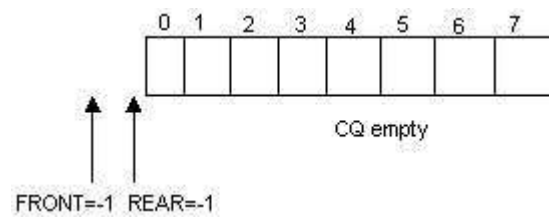
Whenever an element is to be inserted in a circular queue, `Rear` is incremented by one. However, if the value of the rear index is `MAX-1`, instead of incrementing `Rear`, it is reset to the first index of the array if space is available in the beginning. Hence, if any location to the left of the front index is empty, the elements can be added to the queue at an index starting from 0. A queue is considered full in the following cases:

- When the value of `Rear` equals the maximum size of the array and `Front` is at the beginning of the array
- When the value of `Front` is one more than the value of `Rear`

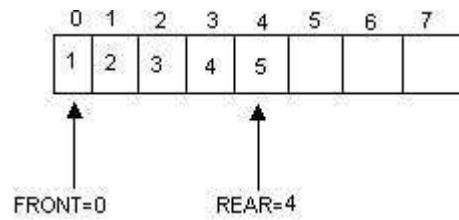
Whenever an element is to be deleted from the queue, `Front` is incremented by one. However, if the value of `Front` is `MAX-1`, it is reset to the 0th position in the array. When the value of `Front` equals the value of `Rear` (other than `-1`), it indicates that there is only one element in the queue. On deleting the last element, both `Rear` and `Front` are reset to `NULL` to indicate an empty queue. Figure 5.5 shows the various states of a queue after some insert and delete operations.

## NOTES

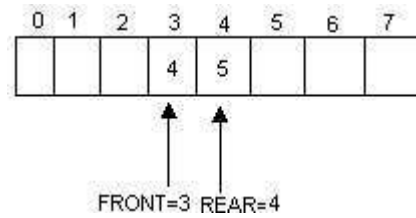
## NOTES



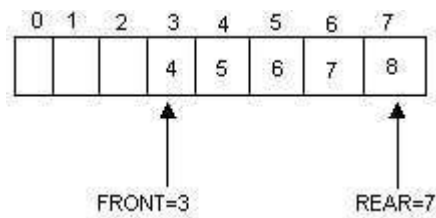
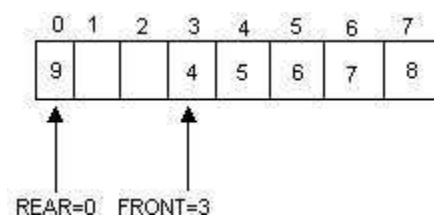
(a) An Empty Queue



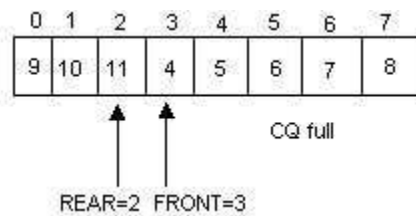
(b) Queue after Inserting a few Elements



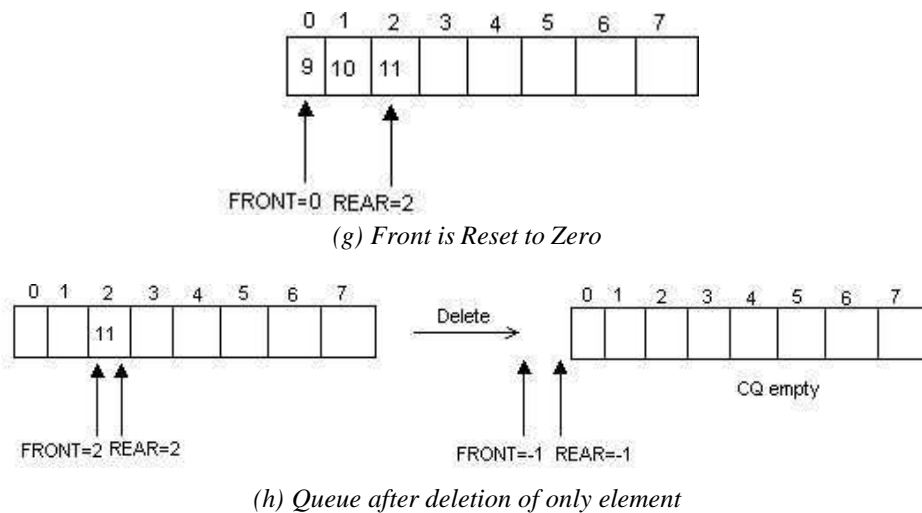
(c) Queue after Deleting a few Elements

(d) Queue when  $Rear = MAX - 1$ 

(e) Rear is Reset to Zero

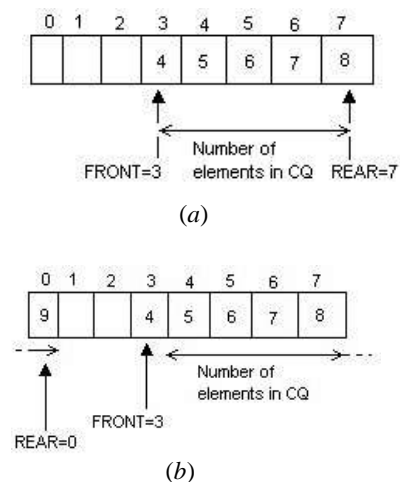


(f) Queue Full



**Fig. 5.5** Various States of a Circular Queue after the Insert and Delete Operations

The total number of elements in a circular queue at any point of time can be calculated from the current values of the rear and front indices of the queue. In case,  $\text{Front} < \text{Rear}$ , the total number of elements =  $\text{Rear} - \text{Front} + 1$ . For instance, in Figure 5.6(a),  $\text{Front} = 3$  and  $\text{Rear} = 7$ . Hence, the total number of elements in CQueue at this point of time is  $7 - 3 + 1 = 5$ . In case,  $\text{Front} > \text{Rear}$ , the total number of elements =  $\text{Max} + (\text{Rear} - \text{Front}) + 1$ . For instance, in Figure 5.6(b),  $\text{Front} = 3$  and  $\text{Rear} = 0$ . Hence, the total number of elements in CQueue is  $8 + (0 - 3) + 1$ .



**Fig. 5.6** Number of Elements in a Circular Queue

## NOTES

## NOTES

**Algorithm 5.5 Insert Operation on a Circular Queue**

qinsert(q, val)

1. If  $((q \rightarrow \text{Rear} = \text{MAX}-1 \text{ AND } q \rightarrow \text{Front} = 0) \text{ OR } (q \rightarrow \text{Rear} + 1 = q \rightarrow \text{Front}))$   
Print "Overflow: Queue is full!" and go to step 5  
End If //check if circular queue is full
2. If  $q \rightarrow \text{Rear} = \text{MAX}-1$  // check if rear is MAX-1  
Set  $q \rightarrow \text{Rear} = 0$   
Else  
Set  $q \rightarrow \text{Rear} = q \rightarrow \text{Rear} + 1$  //increment rear by one  
End If
3. Set  $q \rightarrow \text{CQueue}[q \rightarrow \text{Rear}] = \text{val}$  //val is the value to be inserted in the queue
4. If  $q \rightarrow \text{Front} = -1$  //check if queue is empty  
Set  $q \rightarrow \text{Front} = 0$   
End If
5. End

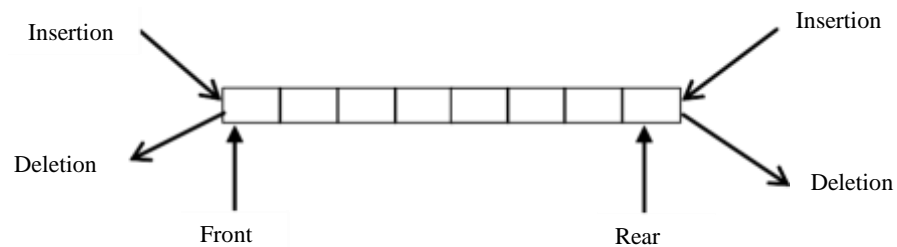
**Algorithm 5.6 Delete Operation on a Circular Queue**

qdelete(q)

1. If  $q \rightarrow \text{Front} = -1$   
Print "Underflow: Queue is empty!"  
Return 0 and go to step 5  
End If
2. Set  $\text{del\_val} = q \rightarrow \text{CQueue}[q \rightarrow \text{Front}]$  //del\_val is the value to be deleted
3. If  $q \rightarrow \text{Front} = q \rightarrow \text{Rear}$  // check if there is one element in the queue  
Set  $q \rightarrow \text{Front} = q \rightarrow \text{Rear} = -1$   
Else  
If  $q \rightarrow \text{Front} = \text{MAX}-1$   
Set  $q \rightarrow \text{Front} = 0$   
Else  
Set  $q \rightarrow \text{Front} = q \rightarrow \text{Front} + 1$   
End If  
End If
4. Return del\_val
5. End

**Deque**

A deque (short form of double-ended queue) is a linear list in which elements can be inserted or deleted at either end but not in the middle. That is, elements can be inserted/deleted to/from the rear end or the front end. Figure 5.7 shows the representation of a deque.

*Fig. 5.7 A Deque*



Like a queue, a deque can be represented as an array or a singly linked list. Here, we will discuss the array implementation of a deque.

**Algorithm 5.7 Insert Operation in the Beginning of a Deque**

```
qinsert_beg(q, val)

1. If (q->Rear = MAX-1 AND q->Front = 0)
    Print "Overflow: Queue is full!" and go to step 4
    End If
2. If q->Front = -1
    Set q->Front = q->Rear = 0
    Set q->DQueue[q->Front] = val
    End If
3. If q->Rear != MAX - 1          //check if last position is occupied
    Set num_item = q->Rear - q->Front + 1 //total number of elements
    Set i = q->Rear + 1
    Set j = 1
    While j <= num_item
        Set q->DQueue[i] = q->DQueue[i-1] //shift elements one space
        to the                          //right
        Set i = i - 1
        Set j = j + 1
    End While
    Set q->DQueue[i] = val
    Set q->Front = i
    Set q->Rear = q->Rear + 1
    Else
        Set q->Front = q->Front - 1
        Set q->DQueue[q->Front] = val
    End If
4. End
```

**NOTES**
**Algorithm 5.8 Insert Operation at the End of a Deque**

```
qinsert_end(q, val)

1. If (q->Rear = MAX-1 And q->Front = 0) //check if queue is full
    Print "Overflow: Queue is full!" and go to step 4
    End If
2. If q->Front = -1          //check if queue is empty
    Set q->Front = q->Rear = 0
    Set q->Dqueue[q->Front] = val and go to step 4
    End If
3. If q->Rear = MAX-1        // check if last position is occupied
    Set i = q->Front - 1
    While i < q->Rear          //shift elements one place to the left of queue
        Set q->Dqueue[i] = q->Dqueue[i+1]
        Set i = i + 1
    End While
    Set q->Dqueue[q->Rear] = valSet
    q->Front = q->Front - 1
    Else
        Set q->Rear = q->Rear + 1
        Set q->Dqueue[q->Rear] = val
    End If
4. End
```

## NOTES

**Algorithm 5.9 Delete Operation in the Beginning of a Deque**

qdelete\_beg(q)

1. If q->Front = -1  
    Print "Underflow: Queue is empty!"  
    Return 0 and go to step 5  
    End If
2. Set del\_val = q->DQueue[q->Front]
3. If q->Front = q->Rear  
    Set q->Front = q->Rear = -1  
Else  
    Set q->Front = q->Front + 1End  
If
4. Return del\_val
5. End

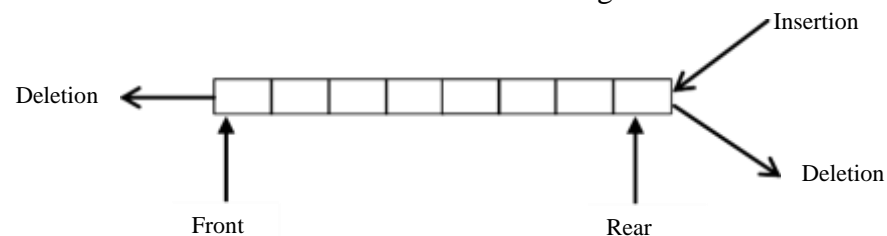
**Algorithm 5.10 Delete Operation at the End of a Deque**

qdelete\_end(q)

1. If q->Front = -1  
    Print "Underflow: DeQueue is empty!"  
    Return 0 and go to step 5  
    End If
2. Set del\_val = q->DQueue[q->Rear]
3. If q->Front = q->Rear  
    Set q->Front = q->Rear = -1  
Else  
    Set q->Rear = q->Rear - 1  
    If q->Rear = -1  
        Set q->Front = -1  
    End If  
End If
4. Return del\_val
5. End

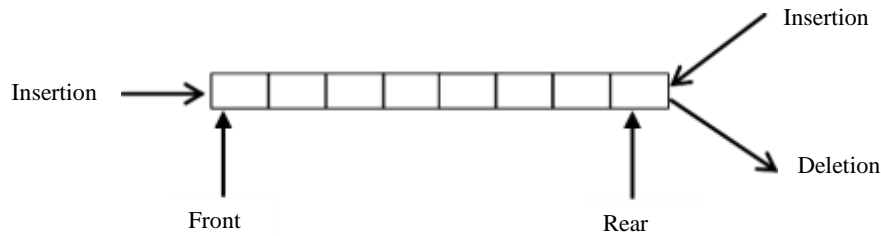
The two variations of a deque are as follows:

- **Input Restricted Deque:** It allows insertion of elements only at one end but deletion can be done at both ends refer Figure 5.8.



*Fig. 5.8 An Input Restricted Deque*

- **Output Restricted Deque:** It allows deletion of elements only at one end but insertion can be done at both ends refer Figure 5.9.



*Fig. 5.9 An Output Restricted Deque*

The implementation of both these queues is similar to the implementation of a deque. The only difference is that in an input restricted queue, the function for insertion in the beginning is not needed, whereas in an output restricted queue, the function for deletion in the beginning is not needed.

## NOTES

### 5.5 PRIORITY QUEUE

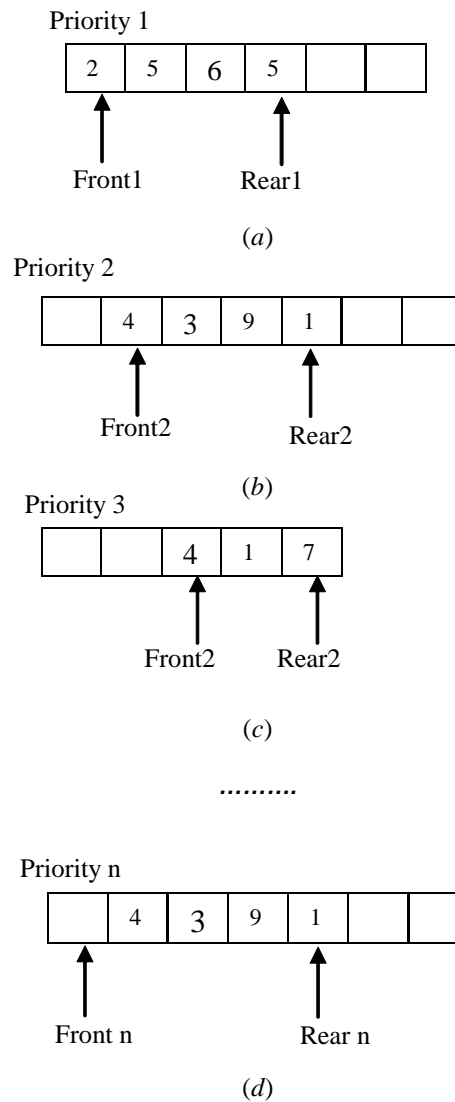
A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority. While implementing a priority queue, the following two rules are applied:

- The element with higher priority is processed before any element of lower priority.
- The elements with the same priority are processed according to the order in which they were added to the queue.

A priority queue can be represented in many ways. Here, we will discuss the implementation of a priority queue using multiple queues.

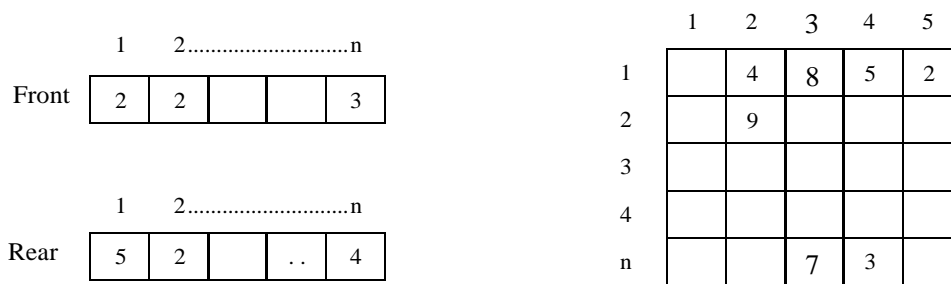
In the multiple queue representation of a priority queue, a separate queue for each priority is maintained. Each queue is implemented as a circular array and has its own two variables, `Front` and `Rear` (refer Figure 5.10). The element with the given priority number is inserted in the corresponding queue. Similarly, whenever an element is to be deleted from the queue, it must be the element from the highest priority queue. Note that lower priority number indicates higher priority.

## NOTES



**Fig. 5.10 Multiple Queues According to the Priority**

If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where the row number shows the priority and the column number shows the position of the element within the queue. In addition, two arrays to keep track of the front and rear positions of each queue corresponding to each row are maintained (refer Figure 5.11).



**Fig. 5.11** A Priority Queue as a Two-Dimensional Array

**Algorithm 5.11 Insert Operation in a Priority Queue**

```
qinsert(q, val, prno) //prno is the priority of val

1. If (q->Rear[prno] = MAX-1 AND q->Front[prno] = 0) OR (q->Rear[prno]+1 =
   q->Front[prno]) Print "Overflow: Queue full!" and go to step 5
   End If
2. If q->Rear[prno-1] = MAX-1
   Set q->Rear[prno-1] = 0
   Else
   Set q->Rear[prno-1] = q->Rear[prno-1] + 1
   End If
3. Set q->CQueue[prno-1][q->Rear[prno-1]] = val
4. If q->Front[prno-1] = -1
   Set q->Front[prno-1] = 0
   End If
5. End
```

**Algorithm 5.12 Delete Operation in a Priority Queue**

```
qdelete(q)

1. Set flag = 0, i = 0
2. While i <= MAX-1
   If NOT (q->Front[prno] = -1) //check if not empty
   Set flag = 1
   Set del_val = q->CQueue[i][q->Front[i]]
   If q->Front[i] = q->Rear[i]
   Set q->Front[i] = q->Rear[i] = -1
   Else If q->Front[i] = MAX-1
   Set q->Front[i] = 0
   Else
   Set q->Front[i] = q->Front[i] + 1
   End If
   End If
   Break //jump out of the while loop
   End If
   Set i = i + 1
   End While
3. If flag = 0
   Return 0 and go to step 4
   Else
   Return del_val
   End If
4. End
```

**NOTES**

## 5.6 APPLICATIONS OF QUEUES

### NOTES

There are numerous applications of queues in computer science. Various real-life applications such as railway ticket reservation and the banking system are implemented using queues. One of the most useful applications of a queue is in simulation. Another application of a queue is in the operating system, to implement various functions like CPU scheduling in a multiprogramming environment, device management (printer or disk), etc. Besides, there are several algorithms like level-order traversal of binary tree, etc., that use queues to solve problems efficiently. This section discusses some of the applications of queues.

#### Simulation

Simulation is the process of modelling a real-life situation through a computer program. Its main use is to study a real-life situation without actually making it occur. It is mainly used in areas like military operations, scientific research, etc., where it is expensive or dangerous to experiment with the real system. In simulation, corresponding to each object and action, there is a counterpart in the program. The objects that are studied are represented as data and the actions are represented as operations on the data. By supplying different data, we can observe the result of the program. If the simulation is accurate, the result of the program represents the behaviour of the actual system accurately.

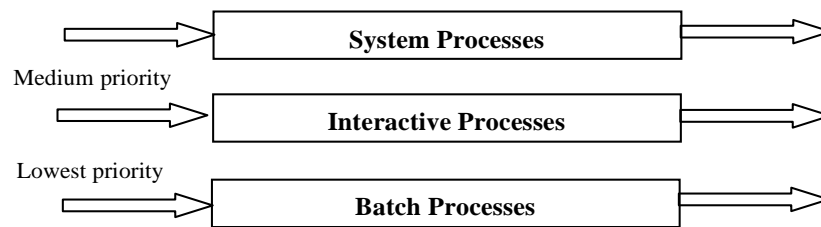
Consider a ticket reservation system having four counters. If a customer arrives at time  $t_a$  and a counter is free, then the customer will get the ticket immediately. However, it is not always possible that a counter is free. In that case, a new customer goes to the queue having fewer customers. Assume that the time required to issue the ticket is  $t$ . Then the total time spent by the customer is equal to the time  $t$  (time required to issue the ticket) plus the time spent waiting in line. The average time spent in the line by the customer can be computed by a program simulating the customer action. This program can be implemented using a queue, since while one customer is being serviced, the others are kept waiting.

#### CPU Scheduling in a Multiprogramming Environment

As we know, in a multiprogramming environment, multiple processes run concurrently to increase CPU utilization. All the processes that are residing in the memory and are ready to execute are kept in a list referred to as a **ready queue**. It is the job of the scheduling algorithm to select a process from the processes and allocate the CPU to it.

Let us consider a multiprogramming environment where the processes are classified into three different groups, namely system processes, interactive processes and batch processes. Some priority is associated with each group of processes. The system processes have the highest priority, whereas the batch processes have the least priority. To implement a multiprogramming environment, a **multi-level**

**queue scheduling** algorithm is used. In this algorithm, the ready queue is partitioned into multiple queues (refer Figure 5.12). The processes are assigned to the respective queues. The higher priority processes are executed before the lower priority processes. For example, no batch process can run unless all the system processes and interactive processes are executed. If a batch process is running and a system process enters the queue, then batch process would be preempted to execute this system process.



*Fig. 5.12 Multi-Level Queue Scheduling*

In this algorithm, the processes of a lower priority may starve if the number of processes in a higher-priority queue is high. Starvation can be prevented by two ways. One way is to time-slice between the queues, that is, each queue gets a certain interval of time. Another way is using a **multi-level feedback queue** algorithm. In this algorithm, processes are not assigned permanently to a queue; instead, they are allowed to move between the queues. If a process uses too much CPU time, it is moved to lower priority. Similarly, a process that has been waiting for too long in a lower-priority queue is moved to the higher-priority queue. To implement multiple programming environments, a priority queue using multiple queues can be used.

### Round Robin Algorithm

The Round Robin algorithm is one of the CPU scheduling algorithms designed for time-sharing systems. In this algorithm, the CPU is allocated to a process for a small time interval called **time quantum** (generally from 10 to 100 milliseconds). Whenever a new process enters, it is inserted at the end of the ready queue. The CPU scheduler picks the first process from the ready queue and processes it until the time quantum elapses. Then, the CPU switches to the next process in the queue and the first process is inserted at the end of the queue if it has not been finished. If the process is finished before the time quantum, the process itself releases the CPU voluntarily and the process gets deleted from the ready queue. This process continues until all the processes are finished. When a process is finished, it is deleted from the queue. To implement the Round Robin algorithm, a circular queue can be used.

Suppose there are  $n$  processes, such as  $P_1, P_2, \dots, P_n$  served by the CPU. Different processes require different execution time. Suppose, sequence of processes arrivals is arranged according to their subscripts, i.e.,  $P_1$  comes first, then  $P_2$ . Therefore,  $P_i$  comes after  $P_{i-1}$  where  $1 < i \leq n$ . Round Robin algorithm first decides a small unit of

## NOTES

NOTES

time called time quantum or time slice represented by  $\tau$ . A time quantum generally starts from 10 to 100 milliseconds. CPU starts services from  $P_1$ . Then,  $P_1$  gets CPU for  $\tau$  instant of time; afterwards CPU switches to process  $P_2$  and so on. Now, during time-sharing, if a process finishes its execution before the finding of its time quantum, the process then simply releases the CPU and the next process waiting will get the CPU immediately. When CPU reaches the end of time quantum of  $P_n$  it returns to  $P_1$  and the same process will be repeated. For an illustration, consider Table 5.1 for the set of processes.

Table 5.1 Table for Process and Burst Time

Process	Burst Time
$P_1$	7
$P_2$	18
$P_3$	5

The total required CPU time keeps 30 units for burst time as summarized in Table 5.1 and depicted in Figure 5.13.

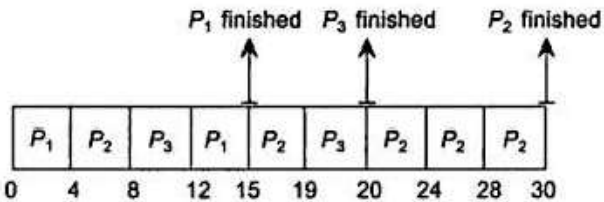


Fig. 5.13 Round Robin Scheduling

The advantage of Round Robin algorithm is in reducing the average turn-around time. The turn-around time of a process is the time of its completion, i.e., time of its arrival. Thus, Round Robin algorithm uses first come first Served or FCFS strategy.

5.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. If there is no element in the queue, the queue is said to be in the condition of underflow.
2. When a queue is implemented as an array, all the characteristics of an array are applicable to the queue.



3. A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.
4. Simulation is the process of modelling a real-life situation through a computer program.

## NOTES

### 5.8 SUMMARY

- A queue is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end.
- The end of the queue from which the element is deleted is known as the Front and the end at which a new element is added is known as the Rear.
- Before inserting a new element in the queue, it is necessary to check whether there is space for the new element.
- If there is no element in the queue, the queue is said to be in the condition of underflow.
- Like stacks, queues can be represented in the memory by using an array or a singly linked list.
- When a queue is implemented as an array, all the characteristics of an array are applicable to the queue.
- Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed.
- Whenever a new element has to be inserted in a queue, `Rear` is incremented by one and the element is stored at `Queue[Rear]`.
- Whenever, an element is to be deleted from a queue, `Front` is incremented by one.
- Whenever an element is deleted from the queue, a temporary pointer is created, which is made to point to the node pointed to by `Front`.
- The `info` and `next` fields of each node represent the element of the queue and a pointer to the next element in the queue, respectively.
- In the case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted.
- The array implementation of a circular queue is similar to the array implementation of the queue. The only difference is that as soon as the rear index of the queue reaches the maximum size of the array, `Rear` is reset to the beginning of the queue, provided it is free.
- A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

**NOTES**

- In the multiple queue representation of a priority queue, a separate queue for each priority is maintained.
- If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where the row number shows the priority and the column number shows the position of the element within the queue.
- One of the most useful applications of a queue is in simulation.
- Simulation is the process of modelling a real-life situation through a computer program.
- If the simulation is accurate, the result of the program represents the behaviour of the actual system accurately.
- To implement a multiprogramming environment, a multi-level queue scheduling algorithm is used.
- If a batch process is running and a system process enters the queue, then batch process would be preempted to execute this system process.
- To implement multiple programming environments, a priority queue using multiple queues can be used.
- The advantage of Round Robin algorithm is in reducing the average turn-around time.

---

## 5.9 KEY WORDS

---

- **Input Restricted Deque:** It allows insertion of elements only at one end but deletion can be done at both ends.
- **Output Restricted Deque:** It allows deletion of elements only at one end but insertion can be done at both ends.
- **Priority queue:** It is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

---

## 5.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

### Short-Answer Questions

1. What are queues?
2. Write a short note about representation of queues.
3. What is a circular queue?
4. List few basic operations performed on queues.

**Long-Answer Questions**

1. Write a program to implement a queue as an array.
2. Differentiate between circular queue and deque.
3. Write a program to implement a circular queue.
4. Write a program to illustrate the insertion and deletion operations on a simple deque.

**NOTES****5.11 FURTHER READINGS**

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

**5.12 LEARNING OUTCOMES**

- Understand queues
- The representation of queues
- Circular queues and deque
- Learn about priority queues
- The applications of queue

---

## UNIT 6    LISTS

---

### NOTES

#### Structure

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Linked List
- 6.3 Singly-Linked Lists
- 6.4 Circular Linked Lists
- 6.5 Doubly-Linked Lists
- 6.6 Merging Lists and Header Linked List
- 6.7 Answers to Check Your Progress Questions
- 6.8 Summary
- 6.9 Key Words
- 6.10 Self Assessment Questions and Exercises
- 6.11 Further Readings
- 6.12 Learning Outcomes

---

### 6.0    INTRODUCTION

---

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. A linked list is a sequence of data structures, which are held together by links. A Linked List is a sequence of links which contains items. Each link contains a connection to another link. A Linked list is the second most-used data structure after an array.

---

### 6.1    OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss linked list
- Explain singly-linked list
- Analyze doubly-linked list
- Understand merging list and header linked list

---

### 6.2    LINKED LIST

---

A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time. The successive elements of a dynamic data structure need not be stored in contiguous memory locations but they are still linked together by means of some linkages or references. Whenever a new element is inserted,

the memory for the same is allocated dynamically and is linked to the data structure. The elements can be inserted as long as memory is available. Thus there is no upper limit on the number of elements in the data structure. Similarly, whenever an element is deleted from the data structure, memory is de-allocated so that it can be reused in the future. Linked list is an example of a dynamic data structure. It has been explained in this section.

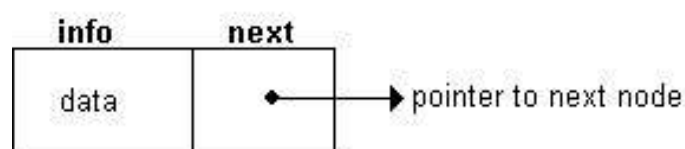
### Linked List

A **linked list** is a linear collection of homogeneous elements called **nodes**. Successive nodes of a linked list need not occupy adjacent memory locations. The linear order between nodes is maintained by means of **pointers**. In linked lists, insertion or deletion of nodes do not require shifting of existing nodes as in the case of arrays; they can be inserted or deleted merely by adjusting the pointers or links.

Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as singly-linked list, circular-linked list and doubly-linked list. The unit will discuss these types in detail in the subsequent sections.

## 6.3 SINGLY-LINKED LISTS

A singly-linked list is also known as a linear linked list. In it, each node consists of two fields, viz. 'info' and 'next', as shown in Figure 6.1. The 'info' field contains the data and the 'next' field contains the address of memory location where the subsequent node is stored. The last node of the singly-linked list contains NULL in its 'next' field which indicates the end of the list.



*Fig. 6.1 Node of a Linked List*

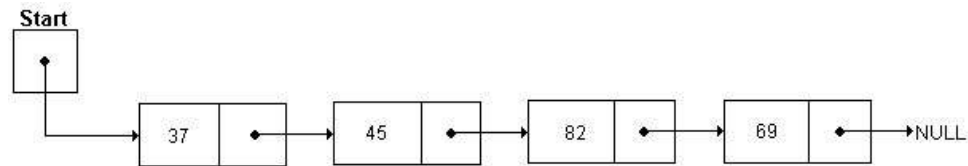
**Note:** The data stored in the 'info' field may be a single data item of any data type or a complete record representing a student, or an employee, or any other entity. In this unit, however, it is assumed that the 'info' field contains an integer data.

A linked list contains a list pointer variable 'Start' that stores the address of the first node of the list. In case, the 'Start' node contains NULL, the list is called an **empty list** or a **null list**. Since each node of the list contains only a single pointer pointing to the next node, not to the previous node—allowing traversing in

### NOTES

only one direction—hence, it is also referred to as a **one-way list**. Figure 6.2 shows a singly-linked list with four nodes.

## NOTES



*Fig. 6.2 A Singly-Linked List with Four Nodes*

## Operations

A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists. Before implementing these operations, it is important to understand how the node of a linked list is created.

Creating a node means defining its structure, allocating memory to it, and its initialization. As discussed earlier, the node of a linked list consists of data and a pointer to the next node. To define a node containing an integer data and a pointer to next node in C language, a self-referential structure can be used whose definition is as follows:

```

typedef struct node
{
    int info;                /*to store integer type data*/
    struct node *next;       /*to store a pointer to next
node*/
}Node;
Node *nptr;                 /*nptr is a pointer to node*/
  
```

After declaring a pointer `nptr` to new node, the memory needs to be allocated dynamically to it. If the memory is allocated successfully (means no overflow), the node is initialized. The `info` field is initialized with a valid value and the `next` field is initialized with `NULL`.

### Algorithm 6.1 Creation of a Node

```

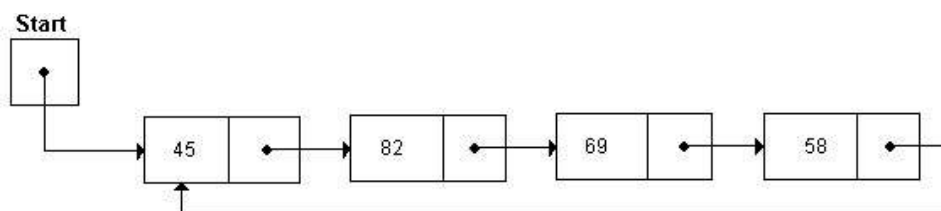
create node()
1. Allocate memory for nptr           //nptr is a pointer to new node
2. If nptr = NULL
    Print "Overflow: Memory not allocated!" and go to step 7
    End If
3. Read item                         //item is the value to be inserted in the
   new node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Return nptr                       //returning pointer nptr
7. End
  
```

Now, the linked list can be formed by creating several nodes of type `Node` and inserting them either in the beginning or at the end or at a specified position in the list.

## NOTES

## 6.4 CIRCULAR LINKED LISTS

A linear linked list, in which the `next` field of the last node points back to the first node instead of containing `NULL`, is termed as a **circular linked list**. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached. This is because when a circular linked list is traversed, starting with a particular node, the same node is reached at the end. Figure 6.3 shows an example of a circular linked list.



*Fig. 6.3 A Circular Linked List*

All the operations that can be performed on linear linked lists can be easily performed on circular linked lists but with some modifications. Some of these operations have been discussed in this section.

**Note:** The process of creating a node of a circular linked list is same as that of linear linked list.

### Traversing

A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list. A circular linked list is traversed until a node in the list is reached at, which contains address of the first node in its `next` field rather than `NULL` as in case of a linear linked list.

#### Algorithm 6.2 Traversing a Circular Linked List

```

display(Start)
1. If Start = NULL
   Print "List is empty!!" and go to step 4
   End If
2. Set temp = Start           //initialising temp with Start
3. Do
   Print temp->info           //displaying value of each node
   Set temp = temp->next
   While temp != Start
4. End
  
```

## NOTES

## 6.5 DOUBLY-LINKED LISTS

In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, one can traverse only in one direction, *i.e.*, from beginning to end. However, sometimes it is required to traverse in the backward direction, *i.e.*, from end to beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such type of a linked list is called **doubly-linked list**.

Each node of a doubly-linked list consists of three fields—`prev`, `info`, and `next` (see Figure 6.4). The `info` field contains the data, the `prev` field contains the address of the previous node, and the `next` field contains the address of the next node.

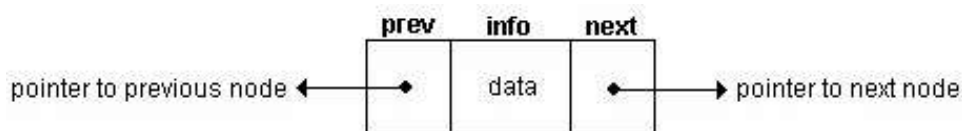


Fig. 6.4 Node of a Doubly-Linked List

Since a doubly-linked list allows traversing in both forward and backward directions, it is also referred to as a **two-way list**. Figure 6.5 shows an example of a doubly-linked list having four nodes. It must be noted that the `prev` field of the first node and `next` field of the last node in a doubly-linked list points to `NULL`.

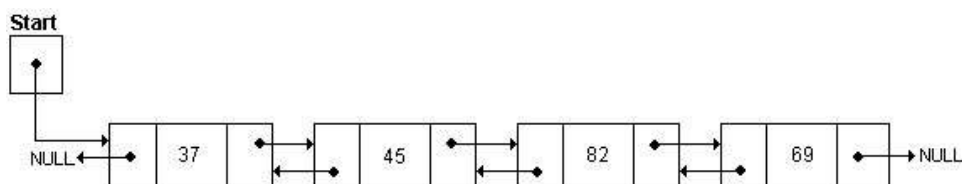


Fig. 6.5 A Doubly-Linked List with Four Nodes

To define the node of a doubly-linked list in C language, the structure used to represent the node of a singly-linked list is extended to have an extra pointer which points to previous node. The structure of a node of a doubly-linked list is as follows:

```
typedef struct node
{
    int info;                /*to store integer type data*/
    struct node *next;       /*to store a pointer to next
node*/
    struct node *prev;       /*to store a pointer to
previous node*/
}Node;
Node *nptr;                 /*nptr is a pointer to node*/
```



When memory is allocated successfully to a node, *i.e.*, when there is no condition of overflow, the node is initialized. The `info` field is initialized with a valid value and the `prev` and `next` fields are initialized with `NULL`.

#### Algorithm 6.3 Creating a Node of a Doubly Linked List

```
create_node()

1. Allocate memory for nptr           //nptr is a pointer to new node
2. If nptr = NULL
   Print "Overflow: Memory not allocated!" and go to step 8
3. Read item                          //item is the value stored in the node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Set nptr->prev = NULL
7. Return nptr
8. End
```

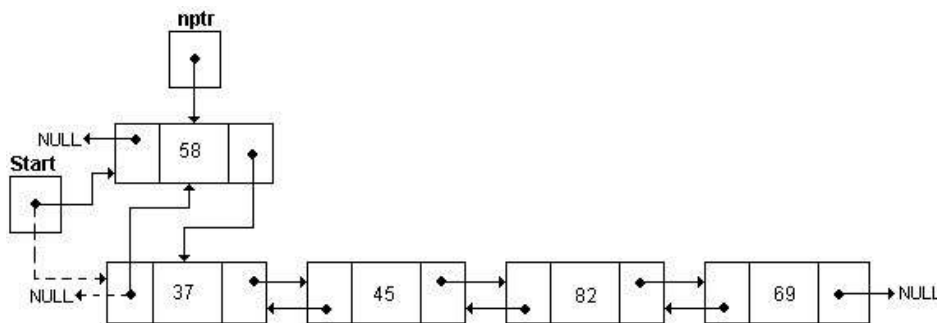
## NOTES

It must be brought to notice that all the operations that are performed on singly-linked lists can also be performed on doubly-linked lists. In the subsequent sections, only insertion and deletion operations on doubly-linked lists have been discussed.

### Insertion

#### Insertion in the beginning

To insert a new node in the beginning of a doubly-linked list, a pointer, for example `nptr` to new node is created. The `next` field of the new node is made to point to the existing first node and `prev` field of the existing first node (that has become the second node now) is made to point to the new node. After that, `Start` is modified to point to the new node. Figure 6.6 shows the insertion of node in the beginning of a doubly-linked list.



**Fig. 6.6** Insertion in the Beginning

#### Algorithm 6.4 Insertion in the Beginning

```
insert_beg(Start)

1. Call create_node()           //creating a new node pointed to by nptr
2. If Start != NULL
   Set nptr->next = Start       //inserting node in the beginning
   Set Start->prev = nptr
   End If
3. Set Start = nptr             //making Start to point to new node
4. End
```

**NOTES**


---

## 6.6 MERGING LISTS AND HEADER LINKED LIST

---

**Merging Lists**

Merge lists or algorithms are a family of algorithms that take multiple sorted lists as a medium of input and in turn produce a single list as an output. This output contains all the elements of the input lists in a neatly sorted order. These algorithms are then used as subroutines in various sorting algorithms, which most famously merge sort.

**Header Linked List**

A header linked list is a linked list that contains a special node at the front of the list. This special node is called a headed node and it does not contain any actual data item that is included in the list but generally contains some useful information about the entire linked list.




---

## 6.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time.
2. A singly-linked list is also known as a linear linked list.
3. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached.
4. A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list.

---

## 6.8 SUMMARY

---

- A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time.
- Whenever a new element is inserted, the memory for the same is allocated dynamically and is linked to the data structure.
- A linked list is a linear collection of homogeneous elements called nodes.

- In linked lists, insertion or deletion of nodes do not require shifting of existing nodes as in the case of arrays; they can be inserted or deleted merely by adjusting the pointers or links.
- Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as singly-linked list, circular-linked list and doubly-linked list.
- As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.
- While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size
- A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists. Creating a node means defining its structure, allocating memory to it, and its initialization.
- A linear linked list, in which the next field of the last node points back to the first node instead of containing NULL, is termed as a circular linked list.
- The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached.
- A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list.
- In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, one can traverse only in one direction, i.e., from beginning to end.

## NOTES

### 6.9 KEY WORDS

- **Linked list:** It is a linear collection of homogeneous elements called nodes.
- **Dynamic data structure:** It is one in which the memory for elements is allocated dynamically during run-time.

### 6.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. What is a linked list?
2. Explain singly-linked list.
3. What do you mean by doubly-linked list?
4. Differentiate between merging list and header linked list.

**NOTES****Long Answer Questions**

1. “A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time.” Explain.
2. “The last node of the singly-linked list contains NULL in its ‘next’ field which indicates the end of the list.” Explain with examples.
3. “A number of operations can be performed on singly-linked lists.” Elaborate.
4. “All the operations that are performed on singly-linked lists can also be performed on doubly-linked lists.” Explain.

**6.11 FURTHER READINGS**

- Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.
- Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.
- Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.
- Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.
- McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.
- Louise I. Shelly 2020 Dark Commerce

**6.12 LEARNING OUTCOMES**

- Linked list
- Singly-linked list
- Doubly-linked list
- Understand merging list and header linked list

# UNIT 7 OPERATION ON LINKED LISTS

## NOTES

### Structure

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Insertion and Deletion Operations in Linked List
- 7.3 Insertion and Deletion in Circular Linked List
- 7.4 Insertion and Deletion in Doubly-Linked Lists
- 7.5 Answers to Check Your Progress Questions
- 7.6 Summary
- 7.7 Key Words
- 7.8 Self Assessment Questions and Exercises
- 7.9 Further Readings
- 7.10 Learning Outcomes

## 7.0 INTRODUCTION

In the previous unit you have studied about lists. Now that you have got an understanding of the basic concepts behind linked list and their types, its time to dive into the common operations that can be performed. This unit will basically discuss the insertion and deletion operations on the linked lists.

## 7.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss lists
- Analyze insertion and deletion of operators in linked lists
- Understand insertion and deletion in circular and doubly-linked lists

## 7.2 INSERTION AND DELETION OPERATIONS IN LINKED LIST

To insert a node in the beginning of a list, the `next` field of the new node (pointed to by `nptr`) is made to point to the existing first node and the `Start` pointer is modified to point to the new node as shown in Figure 7.1.

## NOTES

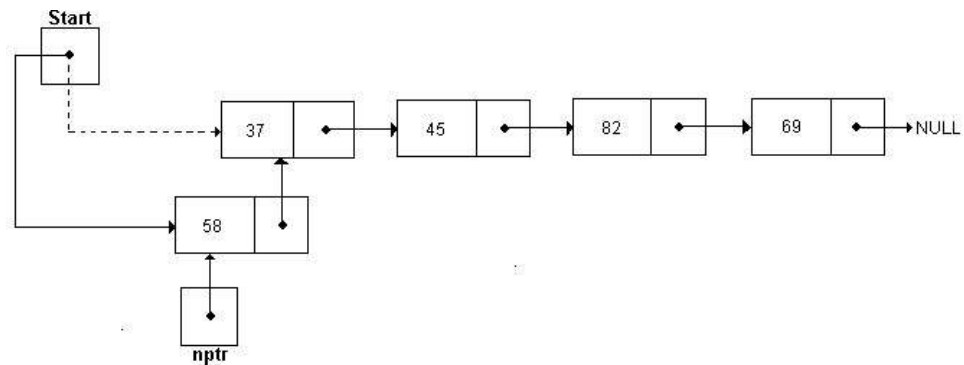


Fig. 7.1 Insertion in the Beginning of a Linked List

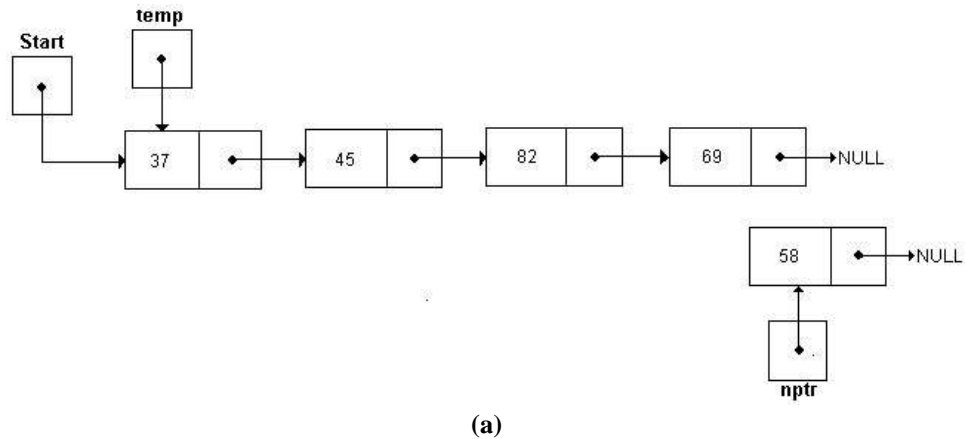
**Algorithm 7.1 Insertion in Beginning**

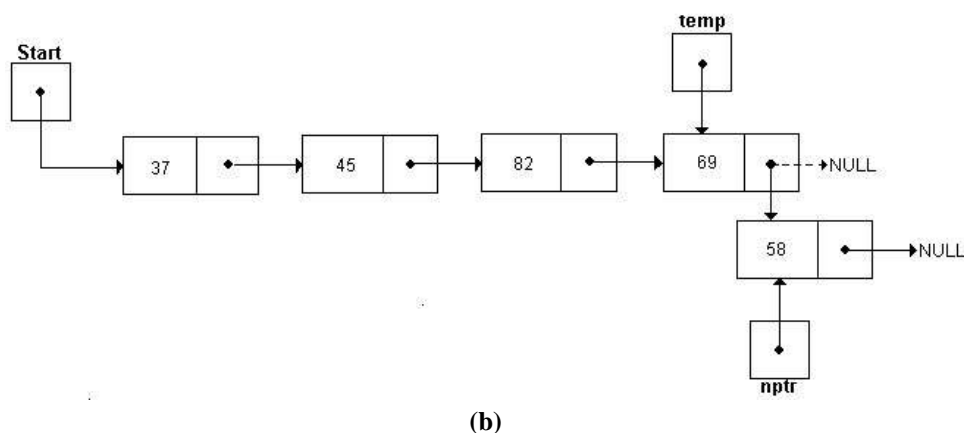
insert beg(Start)

1. Call create node() //creating a new node pointed to by nptr
2. Set nptr->next = Start
3. Set Start = nptr //Start pointing to new node
4. End

**Insertion at end**

To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node. However, if the linked list is initially empty then the new node becomes the first node and Start points to it. Figure 7.2(a) shows a linked list with a pointer variable temp pointing to its first node and Figure 7.2(b) shows temp pointing to the last node and the next field of last node pointing to the new node.





(b)

**Fig. 7.2** Insertion at the End of a Linked List**Algorithm 7.2 Insertion at the End**

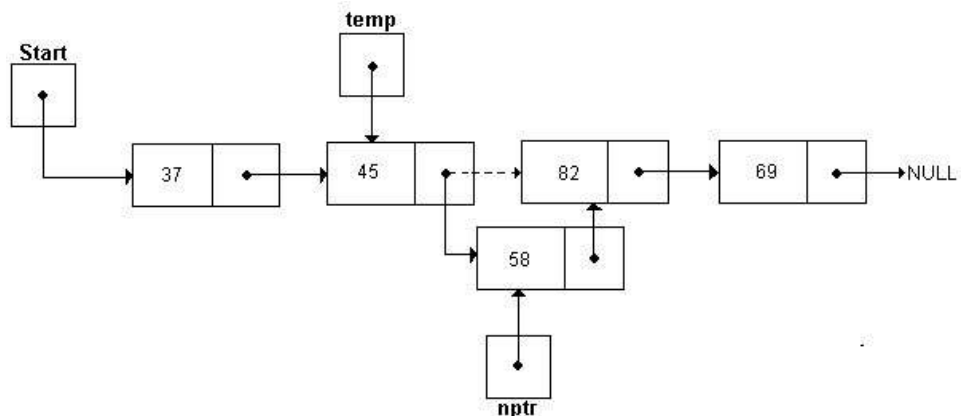
```

insert end(Start)
1. Call create_node()           //creating a new node pointed to by
   nptr
2. If Start = NULL              //checking for empty list
   Set Start = nptr             //inserting new node as the first node
Else
   Set temp = Start
   While temp->next != NULL //traversing up to the last node
   Set temp = temp->next
   End While
   Set temp->next = nptr       //appending new node at the end
End If
3. End

```

**Insertion at a specified position**

To insert a node at a position `pos` as specified by the user, the list is traversed up to `pos-1` position. Then the `next` field of the new node is made to point to the node that is already at the `pos` position and the `next` field of the node at `pos-1` position is made to point to the new node. Figure 7.3 shows the insertion of the new node pointed to by `nptr` at the third position.

**Fig. 7.3** Insertion at a Specified Position in a Linked List**NOTES**

**NOTES****Algorithm 7.3 Insertion at a Specified Position**

```

insert_pos(Start)
1. Call create_node()           //creating a new node pointed to by
   nptr
2. Set temp = Start
3. Read pos                     //position at which the new node is to be
   inserted
4. Call count_node(temp)        //counting total number of nodes in count
   variable
5. If (pos > count + 1 OR pos = 0)
   Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
   Set nptr->next = Start
   Set Start = nptr             //inserting new node as the first node
   Else
   Set i = 1
   While i < pos - 1             //traversing up to the node at pos-1
   position
       Set temp = temp->next
       Set i = i + 1
   End While
   Set nptr->next = temp->next    //inserting new node at pos
   position
   Set temp->next = nptr
   End If
7. End

```

**Deletion**

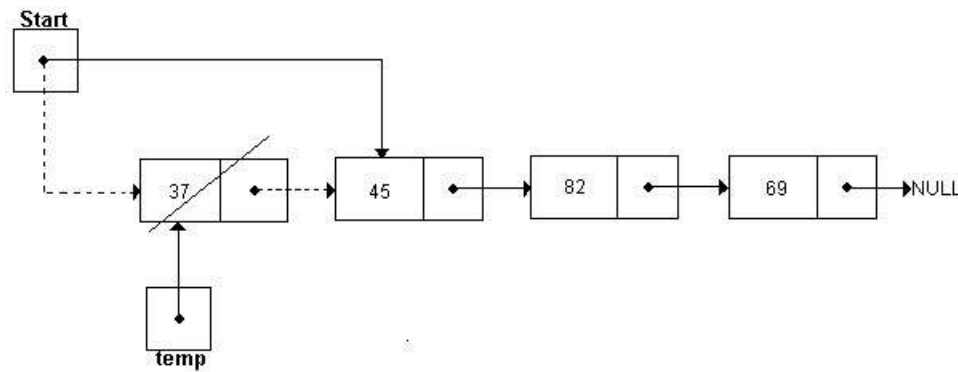
Like insertion, nodes can be deleted from the linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is de-allocated. It must be noted that while performing deletions, the immediate predecessor of the node to be deleted must keep track of. Thus, two temporary pointer variables are used (except in case of deletion from beginning), while traversing the list.

**Note:** A situation where the user tries to delete a node from an empty linked list is termed as underflow.

**Deletion from beginning**

To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable `temp` and `Start` is modified to point to the second node in the linked list. After this, the memory occupied by the node pointed to by `temp` is de-allocated. Figure 7.4 shows the deletion of a node from the beginning of a linked list.





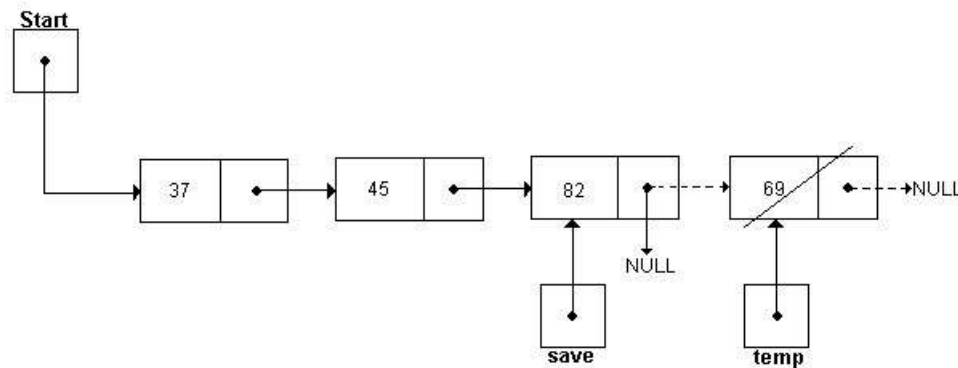
**Fig. 7.4** Deletion from the Beginning of a Linked List

**Algorithm 7.4 Deletion from the Beginning**

```
delete beg(Start)
1. If Start = NULL           //checking for underflow
   Print "Underflow: List is empty!" and go to step 5
   End If
2. Set temp = Start          //temp pointing to the first node
3. Set Start = Start->next    //moving Start to point to the second node
4. Deallocate temp           //deallocating memory
5. End
```

**Deletion from end**

To delete a node from the end of a linked list, the list is traversed up to the last node. Two pointer variables *save* and *temp* are used to traverse the list where *save* points to the node as previously pointed to by *temp*. At the end of traversing, *temp* points to the last node and *save* points to the second last node. Then the *next* field of the node pointed to by *save* is made to point to *NULL* and the memory occupied by the node pointed to by *temp* is de-allocated. Figure 7.5 shows the deletion of a node from the end of a linked list.



**Fig. 7.5** Deletion from the End of a Linked List

**NOTES**

**NOTES****Algorithm 7.5 Deletion from the End**

```

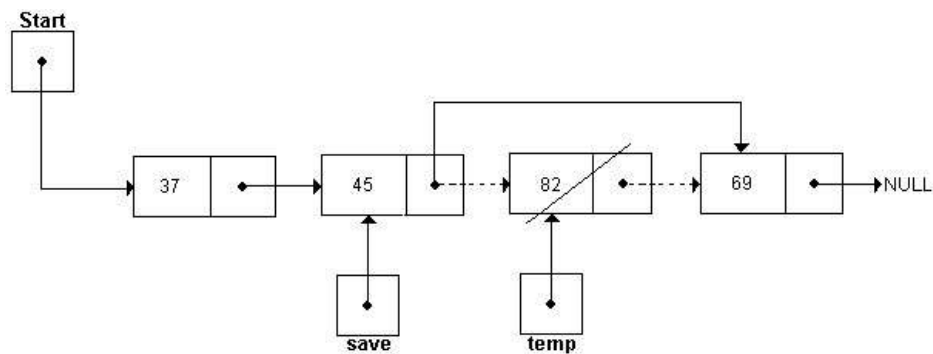
delete end(Start)

1. If Start = NULL //checking for underflow
   Print "Underflow: List is empty!" and go to step 6
End If
2. Set temp = Start //temp pointing to the first node
3. If temp->next = NULL //deleting the only node of the list
   Set Start = NULL
Else
   While (temp->next) != NULL //traversing up to the last node
       Set save = temp //save pointing to node previously
                           //pointed to by temp
       Set temp = temp->next //moving temp to point to next node
   End While
End If
4. Set save->next = NULL //making new last node to point to
   NULL
5. Deallocate temp //deallocating memory
6. End

```

**Deletion from a specified position**

To delete a node from a position *pos* as specified by the user, the list is traversed up to *pos* position using pointer variables *temp* and *save*. At the end of traversing, *temp* points to the node at *pos* position and *save* points to the node at *pos-1* position. Then the *next* field of the node pointed to by *save* is made to point to the node at *pos+1* position and the memory occupied by the node as pointed to by *temp* is de-allocated. Figure 7.6 shows the deletion of a node at the third position.



**Fig. 7.6** Deletion from a Specified Position in a Linked List

**Algorithm 7.6 Deletion from a Specified Position**

```

delete pos(Start)

1. If Start = NULL           //checking for underflow
   Print "Underflow: List is empty!" and go to step 8
   End If
2. Set temp = Start
3. Read pos                  //position of the node to be deleted
4. Call count_node(Start)    //counting total number of nodes in count
   variable
5. If pos > count OR pos = 0
   Print "Invalid position!" and go to step 8
   End If
6. If pos = 1
   Set Start = temp->next    //deleting the first node
   Else
   Set i = 1
   While i < pos             //traversing up to the node at position pos
       Set save = temp
       Set temp = temp->next
       Set i = i + 1
   End While
   Set save->next = temp->next //deleting the node at position
pos
   End If
7. Deallocate temp          //deallocating memory
8. End

```

**NOTES****Searching**

Searching a value for example `item` in a linked list means finding the position of a node, which stores `item` as its value. If `item` is found in the list, the search is successful and the position of that node is displayed. However, if `item` is not found till the end of list, then search is unsuccessful and an appropriate message is displayed. It must be noted that the linked list may be in a sorted or an unsorted order. Therefore, two search algorithms are discussed, one for sorted and another for an unsorted linked list.

**Note:** Only linear search can be performed on linked lists.

**Searching in an unsorted list**

If the data in a linked list is not arranged in a specific order, the list is traversed completely, starting from the first node to the last node and the value of each node (`node->info`) is compared with the value to be searched.

**Algorithm 7.7 Searching in an Unsorted List**

```

search unsort(Start)

1. If Start = NULL
   Print "List is empty!!" and go to step 7
   End If
2. Set ptr = Start           //ptr pointing to the first node
3. Set pos = 1
4. Read item                 //item is the value to be searched
5. While ptr != NULL         //traversing up to the last node
   If item = ptr->info
       Print "Value found at position", pos and go to step 7
   Else
       Set ptr = ptr->next //moving ptr to point to next node
       Set pos = pos + 1
   End If
   End While
6. Print "Value not found"    //search unsuccessful
7. End

```

## Searching in a sorted list

### NOTES

The process of searching an item in a sorted (ascending order) linked list is similar to that of an unsorted linked list. However, while comparing, once the value of any node exceeds *item* (the value to be searched), the search is stopped immediately. In such a case, the list is not required to be traversed completely.

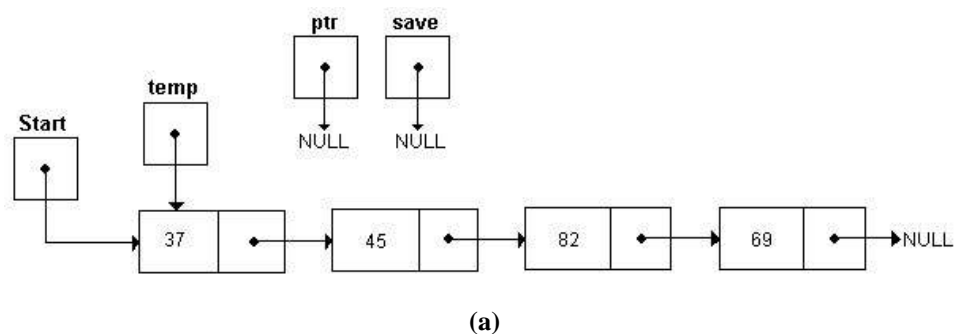
#### Algorithm 7.8 Searching in a Sorted List

```

search sort(Start)
1. If Start = NULL
    Print "List is empty!!" and go to step 7
    End If
2. Set ptr = Start           //ptr pointing to the first node
3. Set pos = 1
4. Read item
5. While ptr->next != NULL   //traversing up to the last node
    If item < ptr->info       //comparing item with the value of current
    node
        Print "Value not found" and go to step 7
    Else If item = ptr->info
        Print "Value found at position", pos and go to step 7
    Else
        Set ptr = ptr->next //moving ptr to point to next node
        Set pos = pos + 1
    End If
    End While
6. Print "Value not found"   //search unsuccessful
7. End
  
```

## Reversing

To reverse a singly-linked list, the list is traversed up to the last node and links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first. For this, three pointer variables like *save*, *ptr* and *temp* are used. Initially, *temp* points to *Start* and both *ptr* and *save* point to *NULL*. While traversing the list, *temp* points to the current node, *ptr* points to the node previously pointed to by *temp* and *save* points to the node previously pointed to by *ptr*. The links between the nodes are reversed by making the *next* field of the node pointed to by *ptr* to point to the node pointed to by *save*. At the end of traversing, *temp* points to *NULL*, *ptr* points to last node, and *save* points to the second last node of the list. Then *Start* is made to point to the node pointed to by *ptr* in order to make the last node as the first node of the list. Figure 7.7 shows the process of reversing a linked list.



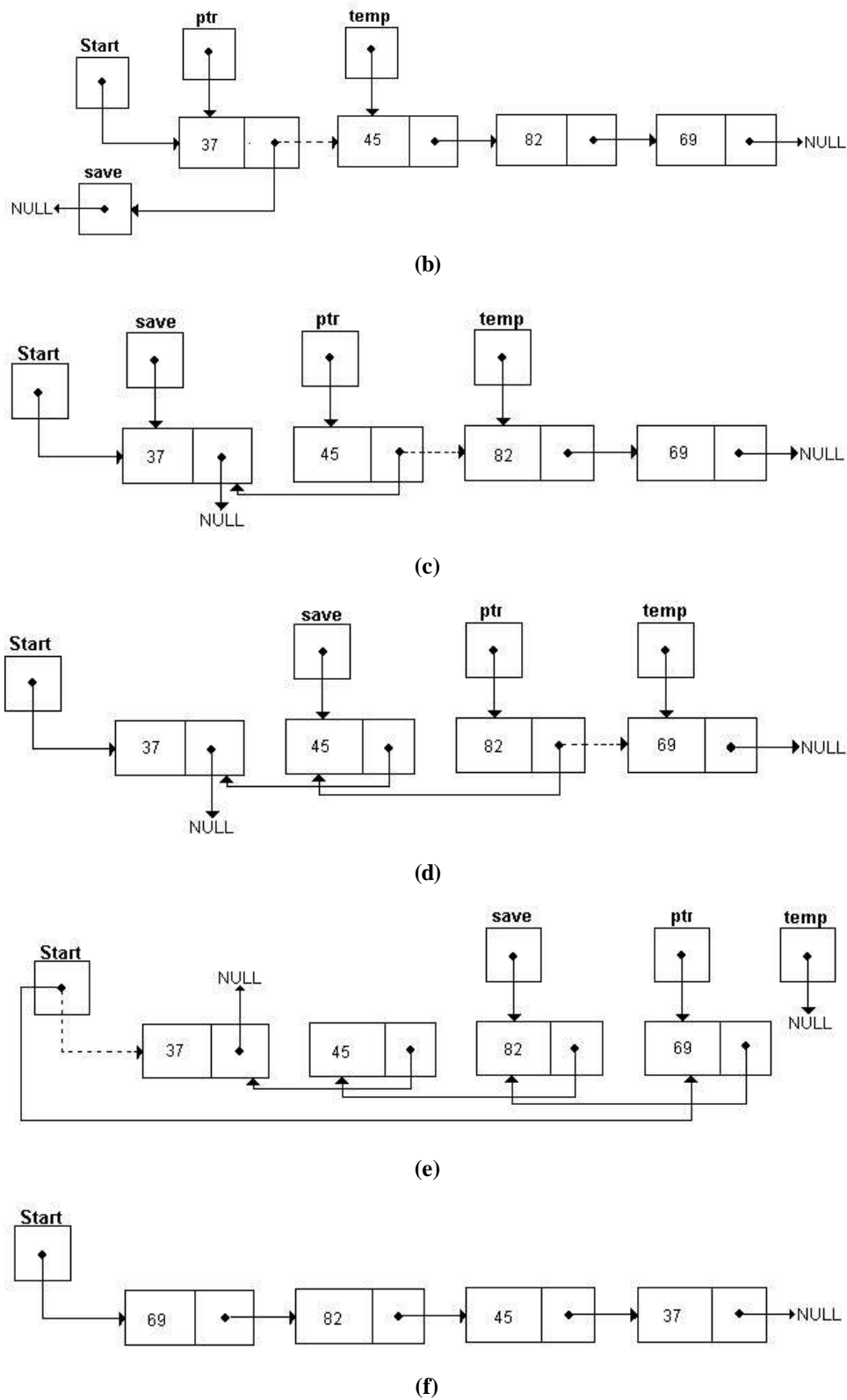


Fig. 7.7 Reversal of a Linked List

## NOTES

**NOTES****Algorithm 7.9 Reversing a Singly-Linked List**

```

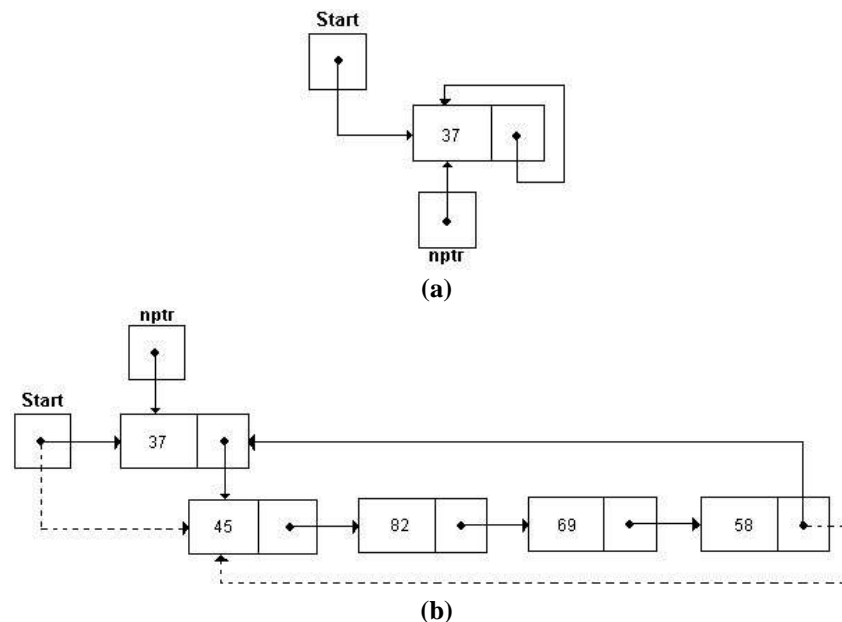
reverse(Start)
1. Set temp = Start
2. Set ptr = NULL
3. Set save = NULL
4. While temp != NULL                               //traversing up to the last node
    Set save = ptr
    Set ptr = temp
    Set temp = temp->next
    Set ptr->next = save
End While
5. Set Start = ptr
6. End

```

The following program shows the searching and reversing operations on a singly-linked list. It must be noted that to simplify a program the linked list is built by creating nodes and inserting them at the end of a list.

### 7.3 INSERTION AND DELETION IN CIRCULAR LINKED LIST

Like linear linked lists, nodes can be inserted at any position in a circular linked list, at the beginning, end or at a specified position. Insertion of a new node at various positions has been discussed in this section. To insert a new node (pointed to by *nptr*) at the beginning of a circular linked list [see Figure 7.8(b)], the *next* field of the new node is made to point to the existing first node and the *Start* pointer is modified to point to the new node. Since the first node of the list is changed, the *next* field of the last node also needs to be modified to point to a new node. However, if initially the list is empty, a new node is inserted as the first node and its *next* field is made to point to itself as shown in Figure 7.8(a).



**Fig. 7.8** Insertion in the Beginning

**Algorithm 7.10 Insertion in the Beginning**

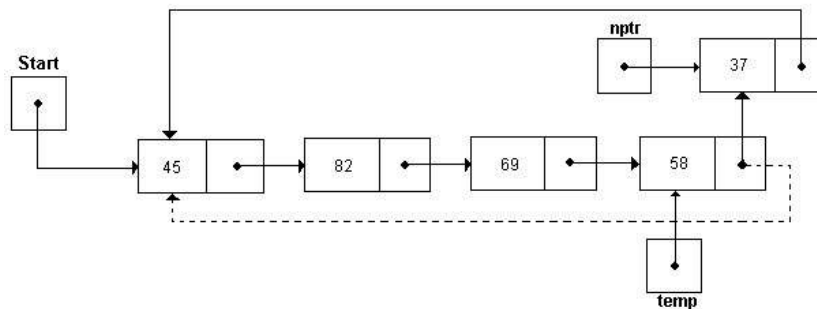
```

insert beg(Start)
1. Call create_node()           //creating a new node pointed to by
   nptr
2. If Start = NULL               //checking for empty list
   Set Start = nptr             //inserting new node as the
first node
   Set Start->next = Start
Else
   Set temp = Start
   While temp->next != Start //traversing up to the last node
   Set temp = temp->next
   End While
   Set nptr->next = Start       //inserting new node in the
beginning
   Set Start = nptr            //Start pointing to new node
   Set temp->next = Start       //next field of last node pointing to new
node
End If
3. End

```

**NOTES****Insertion at the end**

While inserting a new node (pointed to by `np_ptr`), at the end of a circular linked list, the list is traversed up to the last node. The `next` field of the last node is made to point to a new node and the `next` field of the new node is made to point to `Start`. However, if the circular linked list is empty, a new node becomes the first node and `Start` points to it. In addition, the `next` field of the new node points to itself as it is the only node in the list. Figure 7.9 shows the insertion of a new node pointed to by `np_ptr` at the end of a circular linked list.

**Fig. 7.9 Insertion at the End****Algorithm 7.11 Insertion at the End**

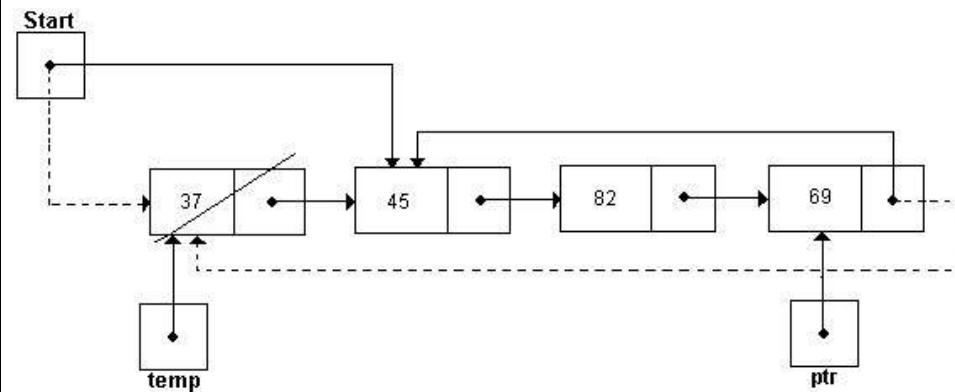
```

insert end(Start)
1. Call create_node()           //creating a new node pointed to by
   nptr
2. If Start = NULL               //checking for empty list
   Set Start = nptr             //inserting new node in the
empty linked list
   Set Start->next = Start       //next field of first node pointing to
itself
Else
   Set temp = Start
   While temp->next != Start //traversing up to the last node
   Set temp = temp->next
   End While
   Set temp->next = nptr         //next field of last node pointing
to new node
   Set nptr->next = Start        //next field of new node pointing
to Start
End If
3. End

```

**Deletion****NOTES**

To delete a node from the beginning of a circular linked list, *Start* is modified to point to the second node and field *next* of the last node is made to point to the new first node. For this, two pointer variables *temp* and *ptr* are used. Pointer *temp* stores the address of the node to be deleted (address of the first node) and *Start* is modified to point to the second node. Pointer *ptr* is used for traversing the list and at the end of traversing, it stores the address of the last node. Then field *next* of the last node is made to point to the new first node. Also, memory occupied by the node pointed to by *temp* is de-allocated. Figure 7.10 shows the deletion of a node from the beginning of a circular linked list.



*Fig. 7.10 Deletion from the Beginning*

**Algorithm 7.12 Deletion from the Beginning**

```

delete beg (Start)
1. If Start = NULL
    Print "Underflow: List is empty!" and go to step 8
    End If
2. Set temp = Start
3. Set ptr = temp
4. While ptr->next != Start           //traversing up to the last node
    Set ptr = ptr->next
    End While
5. Set Start = Start->next           //Start pointing to the next node
6. Set ptr->next = Start             //last node pointing to new first node
7. Deallocate temp                  //deallocating memory
8. End

```

**Deletion from the end**

To delete a node from the end of a circular linked list, two pointer variables *save* and *temp* are used. Pointer variable *temp* is used to traverse the list and *save* points to the node previously pointed to by *temp*. At the end of traversing, *temp* points to the last node and *save* points to the second last node. Then the next



field of `save` is made to point to `Start` and the memory occupied by the last node, i.e., pointed to by `temp` is de-allocated. Figure 7.11 shows the deletion of a node from the end of a circular linked list.

Operation on Linked Lists

## NOTES

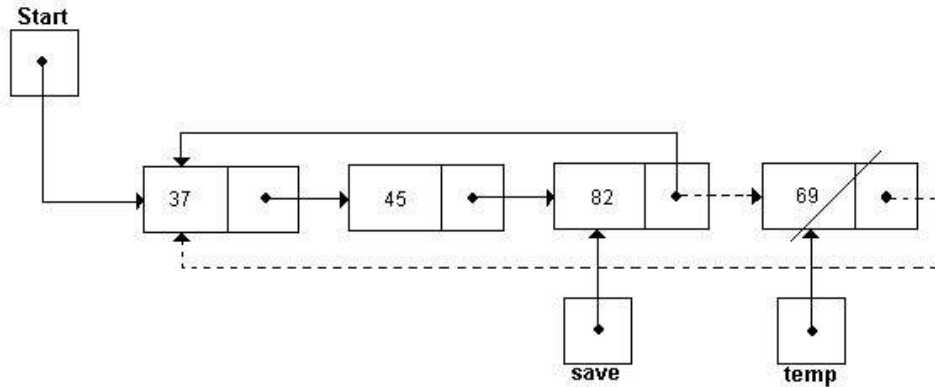


Fig. 7.11 Deletion from the End

### Algorithm 7.13 Deletion from the End

```
delete end(Start)
1. If Start = NULL           //checking for underflow
   Print "Underflow: List is empty!" and go to step 5
End If
2. Set temp = Start
3. If temp->next = Start     //deleting the only node of the list
   Set Start = NULL
Else
   While temp->next != Start //traversing up to the last node
       Set save = temp
       Set temp = temp->next
   End While
   Set save->next = Start    //second last node becomes the last
   node
End If
4. Deallocate temp          //deallocating memory
5. End
```

**Note:** The process of deleting a node from a specified position in a circular linked list is same as that of a singly-linked list.

## 7.4 INSERTION AND DELETION IN DOUBLY-LINKED LISTS

To insert a new node in the beginning of a doubly-linked list, a pointer, for example `nptr` to new node is created. The `next` field of the new node is made to point to the existing first node and `prev` field of the existing first node (that has become the second node now) is made to point to the new node. After that, `Start` is modified to point to the new node. Figure 7.12 shows the insertion of node in the beginning of a doubly-linked list.

## NOTES

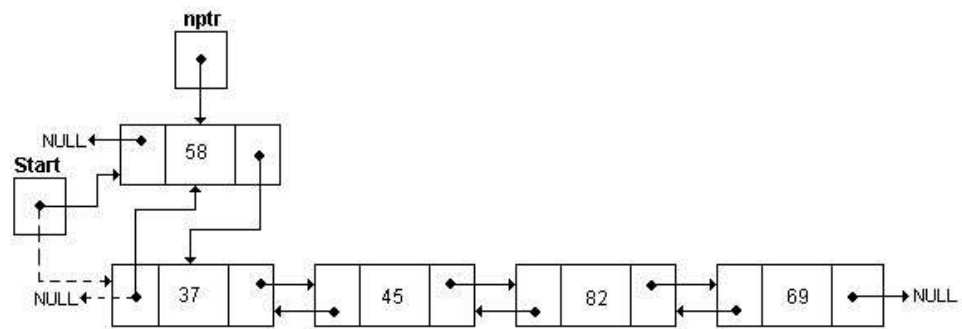


Fig. 7.12 Insertion in the Beginning

**Algorithm 7.14 Insertion in the Beginning**

```

insert beg(Start)
1. Call create_node()           //creating a new node pointed to by nptr
2. If Start != NULL
    Set nptr->next = Start       //inserting node in the beginning
    Set Start->prev = nptr
    End If
3. Set Start = nptr             //making Start to point to new node
4. End

```

**Insertion at the end**

To insert a new node at the end of a doubly-linked list, the list is traversed up to the last node using some pointer variable, for example, the `temp`. At the end of traversing, `temp` points to the last node. Then, field `next` of the last node (pointed to by `temp`), is made to point to the new node and the field `prev` of the new node is made to point to the node pointed to by `temp`. However, if a list is empty, the new node is inserted as the first node in the list. Figure 7.13 shows the insertion of a new node at the end of a doubly-linked list.

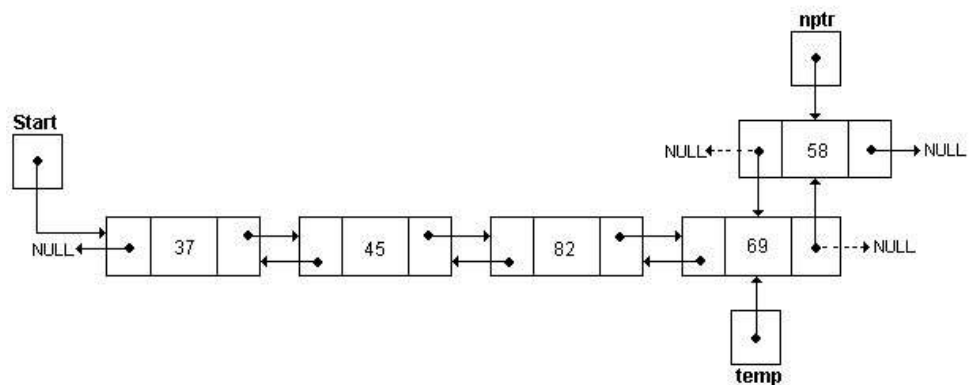


Fig. 7.13 Insertion at the End

**Algorithm 7.15 Insertion at the End**

```

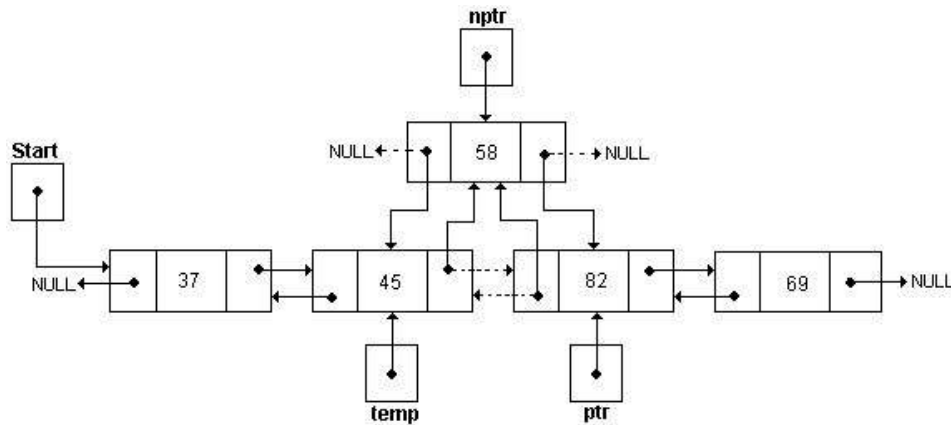
insert_end(Start)

1. Call create_node()           //creating a new node pointed to by
   nptr
2. If Start = NULL
   Set Start = nptr             //inserting new node as the first
   node
   Else
   Set temp = Start              //pointer temp used for traversing
   While temp->next != NULL
   Set temp = temp->next
   End While
   Set temp->next = nptr
   Set nptr->prev = temp
   End If
3. End

```

**NOTES****Insertion at a specified position**

To insert a new node (pointed to by `nptr`) at a specified position, for example, `pos` in a doubly-linked list, the list is traversed up to `pos-1` position. At the end of traversing, `temp` points to the node at `pos-1` position. For simplicity, another pointer variable, `ptr` is used to point to the node that is already at position `pos`. Then, field `prev` of the node, pointed to by `ptr` is made to point to the new node and field `next` of the new node is made to point to the node pointed to by `ptr`. Also, field `prev` of the new node is made to point to the node pointed to by `temp` and field `next` of the node pointed to by `temp` is made to point to the new node. Figure 7.14 shows the insertion of a new node at the third position in a doubly-linked list.



**Fig. 7.14** Insertion at a Specified Position

## NOTES

**Algorithm 7.16 Insertion at a Specified Position**

```

insert_pos(Start)
1. Call create_node()           //creating a new node pointed to by
   nptr
2. Set temp = Start
3. Read pos
4. Call count_node(temp)        //counting number of nodes in count
5. If pos = 0 OR pos > count + 1
   Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
   Set nptr->next = Start        //inserting node at the beginning
   Set Start = nptr             //Start pointing to new node
   Else
   Set i = 1
   While i < pos-1               //traversing up to the node at pos-1
   position
       Set temp = temp->next
       Set i = i + 1
   End While
   Set ptr = temp->next
   Set ptr->prev = nptr
   Set nptr->next = ptr
   Set nptr->prev = temp
   Set temp->next = nptr
   End If
7. End

```

**Deletion**

To delete a node from the beginning of a doubly-linked list, a pointer variable, for example, `temp` is used to point to the first node. Then `Start` is modified to point to the next node and the `prev` field of this node is made to point to `NULL`. After that, the memory occupied by the node pointed to by `temp` is de-allocated. Figure 7.15 shows the deletion of a node from the beginning of a doubly-linked list.

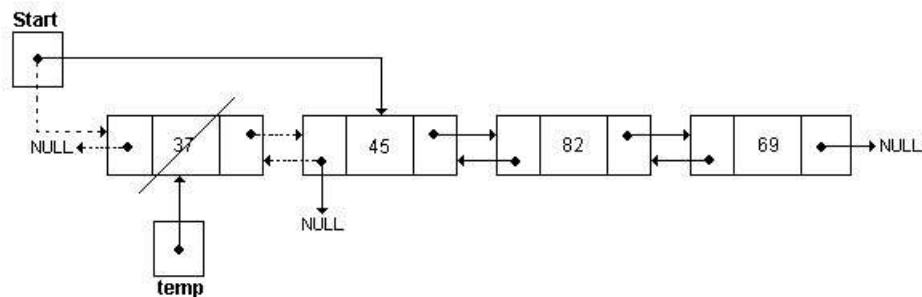


Fig. 7.15 Deletion from the Beginning

**Algorithm 7.17 Deletion from the Beginning**

```

delete_beg(Start)
1. If Start = NULL
   Print "Underflow: List is empty!" and go to step 6
   End If
2. Set temp = Start             //temp points to the node to be deleted
3. Set Start = Start->next      //making Start to point to next node
4. Set Start->prev = NULL
5. Deallocate temp              //de-allocating memory
6. End

```

**Note:** The process of deleting node from the end of a doubly-linked list is same as that of singly-linked list.

## Deletion from a specified position

Operation on Linked Lists

To delete a node from a position, for example, `pos`, as specified by the user, the list is traversed up to the position `pos`, using pointer variables `temp` and `save`. At the end of traversing, `temp` points to the node at `pos` position and `save` points to the node at `pos-1` position. For simplicity, another pointer variable `ptr` is used to point to the node at `pos+1` position. Then the `next` field of the node at `pos-1` position (pointed to by `save`) is made to point to the node at `pos+1` position (pointed to by `ptr`). In addition, the field `prev` of the node at `pos+1`, position (pointed to by `ptr`) is made to point to the node at `pos-1` position (pointed to by `save`). After that, the memory occupied by the node pointed to by `temp` is de-allocated. Figure 7.16 shows the deletion of a node at the third position from a doubly-linked list.

## NOTES

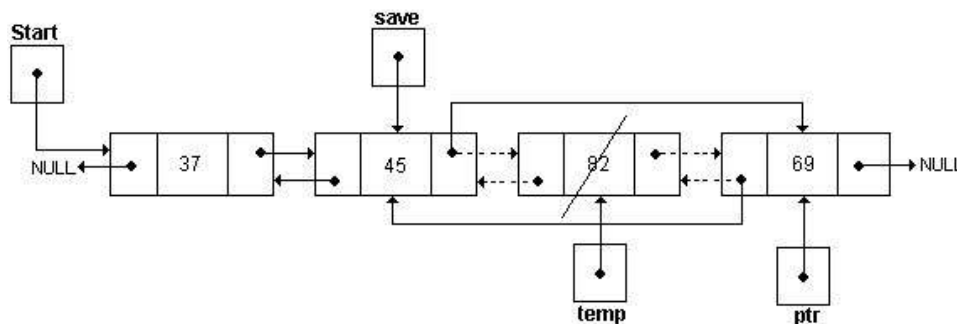


Fig. 7.16 Deletion from a Specified Position

### Algorithm 7.18 Deletion from a Specified Position

```
delete pos(Start)
1. If Start = NULL
   Print "Underflow: List is empty!" and go to step 8
   End If
2. Set temp = Start
3. Read pos
4. Call count_node(temp)           //counting total number of nodes in count
   variable
5. If pos > count OR pos = 0
   Print "Invalid position!" and go to step 6
   End If
6. If pos = 1
   Set Start = Start->next          //deleting the first node
   Start->prev = NULL
   Else
   Set i = 1
   While i < pos                    //traversing up to the node at pos
   position
       Set save = temp              //save pointing to the node at pos-1
   position
       Set temp = temp->next         //making temp to point to next node
       Set i = i + 1
   End While
   Set ptr = temp->next
   Set save->next = ptr
   Set ptr->prev = save
   End If
7. Deallocate temp                 //de-allocating memory
8. End
```

**Note:** A doubly-linked list, in which the next field of the last node points to the first node instead of 'NULL', is termed as a **doubly circular linked list**.

## NOTES

### 7.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Whenever a node is deleted, the memory occupied by the node is de-allocated.
2. To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node.
3. To delete a node from the end of a circular linked list, two pointer variables save and temp are used.

### 7.6 SUMMARY

- To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node.
- To insert a node at a position pos as specified by the user, the list is traversed up to pos-1 position
- Like insertion, nodes can be deleted from the linked list at any point of time and from any position.
- Whenever a node is deleted, the memory occupied by the node is de-allocated.
- It must be noted that while performing deletions, the immediate predecessor of the node to be deleted must be keep track of.
- To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and Start is modified to point to the second node in the linked list.
- To delete a node from the end of a linked list, the list is traversed up to the last node.
- To delete a node from a position pos as specified by the user, the list is traversed up to pos position using pointer variables temp and save.

- Searching a value for example item in a linked list means finding the position of a node, which stores item as its value. If item is found in the list, the search is successful and the position of that node is displayed.
- To reverse a singly-linked list, the list is traversed up to the last node and links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first.
- The links between the nodes are reversed by making the next field of the node pointed to by ptr to point to the node pointed to by save.
- To delete a node from the beginning of a circular linked list, Start is modified to point to the second node and field next of the last node is made to point to the new first node.
- Pointer ptr is used for traversing the list and at the end of traversing, it stores the address of the last node.
- To delete a node from the end of a circular linked list, two pointer variables save and temp are used.
- To insert a new node in the beginning of a doubly-linked list, a pointer, for example nptr to new node is created.

## NOTES

### 7.7 KEY WORDS

- **Underflow:** It is a situation where the user tries to delete a node from an empty linked list.
- **Doubly-linked list:** It is a list in which the next field of the last node points to the first node instead of 'NULL'.

### 7.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. What are lists?
2. Write an algorithm to insert a node at the end in linked lists.
3. Write an algorithm to delete a node in the linked list?

#### Long-Answer Questions

1. Write a detailed note on insertion and deletion of operations in linked list.
2. What do you mean by insertion and deletion in circular linked list? Explain in detail.
3. Write a detailed report on insertion and deletion in doubly-linked lists.

## NOTES

---

### 7.9 FURTHER READINGS

---

- Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.
- Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.
- Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.
- Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.
- McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

---

### 7.10 LEARNING OUTCOMES

---

- Insertion and deletion of operators in linked lists
- Understand insertion and deletion in circular and doubly-linked lists



## UNIT 8 TRAVERSAL

### Structure

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Traversing Linked Lists
- 8.3 Representation of Linked List
- 8.4 Answers to Check Your Progress Questions
- 8.5 Summary
- 8.6 Key Words
- 8.7 Self Assessment Questions and Exercises
- 8.8 Further Readings
- 8.9 Learning Outcomes

### NOTES

### 8.0 INTRODUCTION

In the previous unit, you have learnt about the insertion and deletion operations on linked lists. In this unit, you will learn about the traversing and representation of linked lists. Traversing means to access the elements of the lists for searching an element, find the position for insertion etc.

### 8.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss traversing linked lists
- Explain the representation of linked lists
- Analyze memory allocation

### 8.2 TRAVERSING LINKED LISTS

#### Traversing

Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements. For example, to display values of the nodes, the number of nodes counted, or a particular item in the list is searched, then traversing is required. A list can be traversed by using a temporary pointer variable `temp`, which will point to the node that is currently being processed. Initially, `temp` points to the first node, processes that element, moves `temp` point to the next node using the statement `temp=temp->next`, processes that element, and moves on as long as the last node is not reached, that is, until `temp` becomes `NULL`.

## NOTES

**Algorithm 8.1 Traversing a List**

```

display(Start)
1. If Start = NULL           //Start points to the first node of list
   Print "List is empty!!" and go to step 4
   End If
2. Set temp = Start          //initialising temp with Start
3. While temp != NULL
   Print temp->info           //displaying value of each node
   Set temp = temp->next      //moving temp to point to next node
   End While
4. End

```

Another example of traversing a linked list is counting the number of nodes in the linked list, which is described in the algorithm as illustrated here.

**Algorithm 8.2 Counting the Number of Nodes**

```

count_node(Start)
1. Set count = 0
2. Set temp = Start          //initialising temp with Start
3. While temp != NULL
   Set count = count + 1     //traversing the list
   Set temp = temp->next      //incrementing count
   End While
4. Return count              //returning total number of nodes in the list
5. End

```

### 8.3 REPRESENTATION OF LINKED LIST

To maintain a linked list in the memory, two parallel arrays of equal size are used. One array (INFO) is used for the 'info' field and another array (NEXT) is used for the 'next' field of the nodes of a list. Values in the arrays are stored such that the 'i<sup>th</sup>' location in arrays 'INFO' and 'NEXT' contain the 'info' and 'next' fields of a node of the list respectively. In addition, a pointer variable 'Start' is maintained in memory that stores the starting address of the list. Figure 8.1 shows the memory representation of a linked list where each node contains an integer.

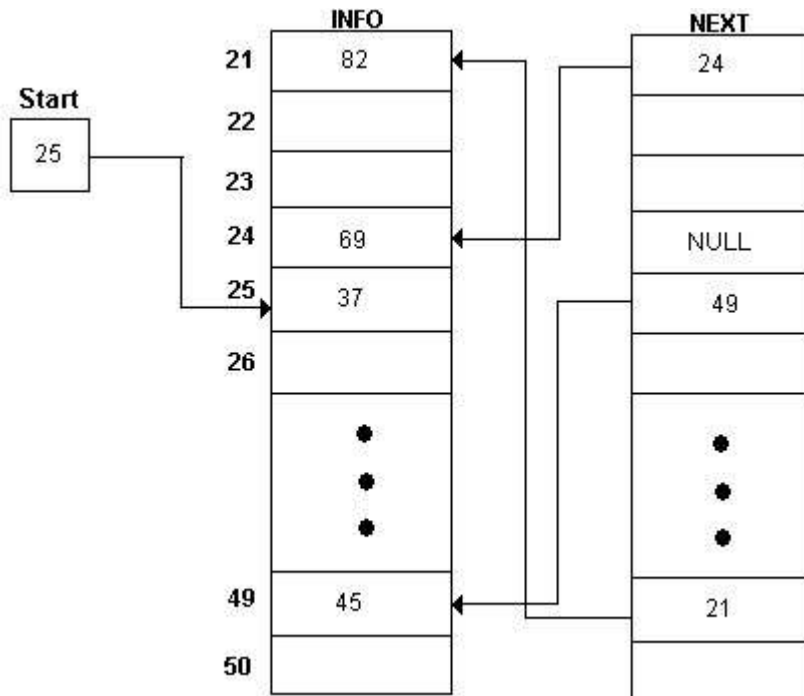
In Figure 8.1, the pointer variable *Start* contains 25, which is the address of first node of the list. This node stores the value 37 in array *INFO* and its corresponding element in array *NEXT* stores 49 which is the address of next node in the list and so on. Finally, the node at address 24 stores value 69 in array *INFO* and *NULL* in array *NEXT*, thus, it is the last node of the list. It must be noted that values in array *INFO* are stored randomly and array *NEXT* is used to keep a track of the values in the list.

#### Memory allocation

As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list. For this, the **memory manager** maintains a special linked list known as a **free-storage list** or **memory bank** or **free pool** which consists of unused memory cells. This list keeps a track of the free space available in the

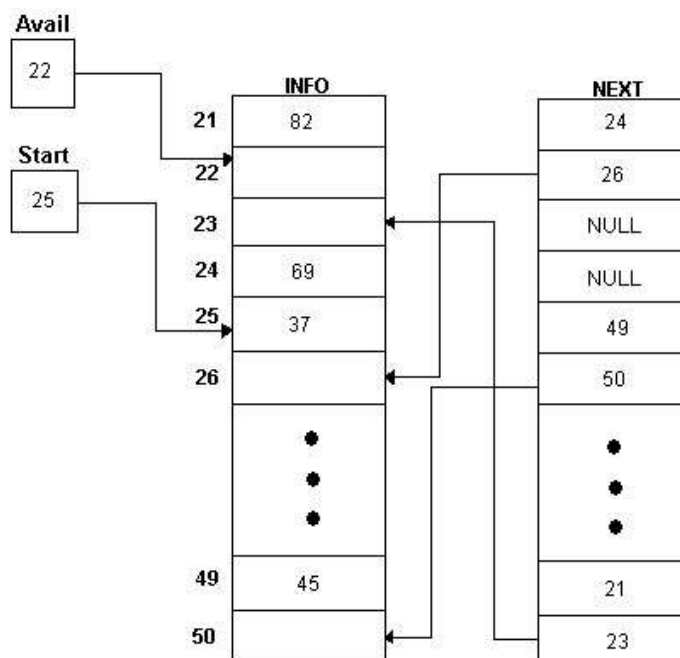
memory and a pointer to this list is stored in a pointer variable `Avail` (see Figure 8.2). Note that the end of the free-storage list is also denoted by storing `NULL` in the last available block of memory.

*Traversal*



## NOTES

*Fig. 8.1 Memory Representation of a Linked List*



*Fig. 8.2 Free-Storage List*

**NOTES**

In Figure 8.2, `Avail` contains 22, hence, `INFO[22]` is the starting point of the free-storage list. Since `NEXT[22]` contains 26, `INFO[26]` is the next free memory location. Similarly, other free spaces can be accessed and the `NULL` in `NEXT[23]` indicates the end of free-storage list.

While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free-storage list for the block of desired size. If the block of desired size is found, it returns a pointer to that block. However, sometimes there is no space available, i.e., the free-storage list is empty. This situation is termed as **overflow** and the memory manager replies accordingly.




---

## 8.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. To maintain a linked list in the memory, two parallel arrays of equal size are used.
2. Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements.
3. As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.

---

## 8.5 SUMMARY

---

- A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists.
- Creating a node means defining its structure, allocating memory to it, and its initialization.
- To define a node containing an integer data and a pointer to next node in C language, a self-referential structure can be used whose definition is as follows:
- Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements.

- To maintain a linked list in the memory, two parallel arrays of equal size are used.
- As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.
- While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size.
- If the block of desired size is found, it returns a pointer to that block.

## NOTES

### 8.6 KEY WORDS

- **Traversing a list:** It means accessing the elements of a linked list, one by one, to process all or some of the elements.
- **Linked List:** It is an ordered set of data elements, each containing a link to its successor (and sometimes its predecessor).

### 8.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. Write a short note on linked lists.
2. How are linked lists represented?
3. List the operations performed on linked lists.

#### Long-Answer Questions

1. “Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements.” Explain.
2. “As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.” Discuss.
3. “While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size.” Discuss.

### 8.8 FURTHER READINGS

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

## NOTES

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

---

## 8.9 LEARNING OUTCOMES

---

- Traversing linked lists
- The representation of linked lists
- Memory allocation

---

## BLOCK - III

*Trees*

---

### NON-LINEAR DATA STRUCTURE

---

#### NOTES

---

## UNIT 9 TREES

---

### Structure

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Binary Trees
  - 9.2.1 Forms/Types of Binary Tree
  - 9.2.2 Binary Tree Representations
- 9.3 Answers to Check Your Progress Questions
- 9.4 Summary
- 9.5 Key Words
- 9.6 Self Assessment Questions and Exercises
- 9.7 Further Readings
- 9.8 Learning Outcomes

---

### 9.0 INTRODUCTION

---

A tree is a widely used abstract data type also called an ADT, or data structure. This ADT simulates a hierarchical tree structure that has a root value and subtrees of children with a parent node; these are represented as a set of linked nodes. A binary tree is made up of nodes, where each node contains a 'left' and 'right' reference, and a data element. The topmost node in the tree is called the root. Nodes that go with the same parent are called siblings. In this unit you will learn in detail about trees.

---

### 9.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss binary trees
- Analyze the nature of binary trees
- Explain the forms of binary trees
- Understand the concept of extended binary tree

---

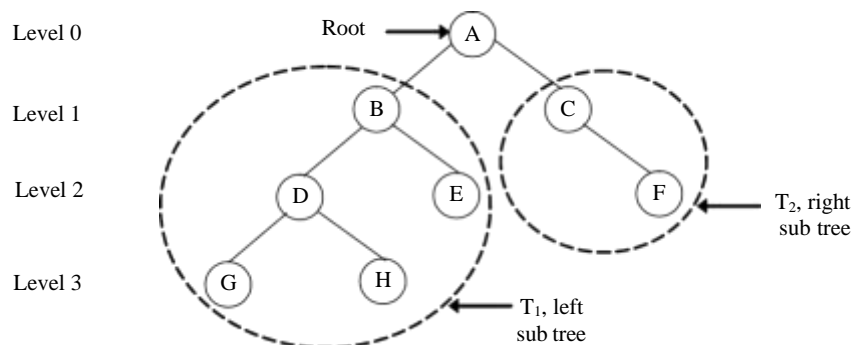
### 9.2 BINARY TREES

---

A binary tree is a special type of tree, which can either be empty or has finite set of nodes such that one of the nodes is designated as root node and remaining nodes are partitioned into two sub trees of root node known as left sub tree and right sub

tree. The nonempty left sub tree and the right sub tree are also binary trees. Unlike general tree each node in binary tree is restricted to have only two child nodes. Consider a sample binary tree T shown in Figure 9.1.

## NOTES



**Fig. 9.1** A Binary Tree

In Figure 9.1, the topmost node A is the root node of the tree T. Each node in this tree has zero or at the most two child nodes. The nodes A, B and D have two child nodes, node C has only one child node, and the nodes G, H, E and F are leaf nodes having no child nodes. The nodes B, C, D are internal nodes having child as well as parent nodes. Some basic terms associated with binary trees are:

- **Ancestor and descendant:** Node N1 is said to be an ancestor of node N2. N1 is the parent node of N2 or so on, whereas, node N2 is said to be the descendant of node N1. The node N2 is said to be left descendant of node N1 if it belongs to left sub tree of N1 and is said to be the right descendant of N1 if it belongs to right sub tree of N1. In binary tree shown in Figure 9.1, node A is ancestor of node H and node H is left descendant of node A.
- **Degree of a node:** Degree of a node is equal to the number of its child nodes. In binary tree shown in Figure 9.1, the nodes A, B and D have degree 2, node C has degree 1 and nodes G, H, E and F have degree 0.
- **Level:** Since binary tree is a multilevel data structure, each node belongs to a particular level number. In binary tree shown in Figure 9.1, the root node A belongs to level 0, its child nodes belong to level 1, child nodes of nodes B and C belong to level 2, and so on.
- **Depth (or height):** Depth of the binary tree is the highest level number of any node in a binary tree. In binary tree shown in Figure 9.1, the nodes G and H are nodes with highest level number 3. Hence, the depth of the binary tree is 3.
- **Siblings:** The nodes belonging to the same parent node are known as sibling nodes. In binary tree shown in Figure 9.1, nodes B and C are sibling nodes as they have same parent node, that is, A. Similarly, the nodes D and E are also sibling nodes.



- **Edge:** Edge is a line connecting any two nodes. In binary tree shown in Figure 9.1, there exists an edge between nodes A and B, whereas, there is no edge between the nodes B and C.
- **Path:** Path between the two nodes  $x$  and  $y$  is a sequence of consecutive edges being followed from node  $x$  to  $y$ . In binary tree shown in Figure 9.1, the path between the nodes A and H is  $A \rightarrow B \rightarrow D \rightarrow H$ . Similarly, the path from A to F is  $A \rightarrow C \rightarrow F$ .

## NOTES

### 9.2.1 Forms/Types of Binary Tree

There are various forms of binary trees that are formed by imposing certain restrictions on them. Some of the variations of binary trees are—complete binary tree and extended binary tree.

#### Complete Binary Tree

A binary tree is said to be complete binary tree if all the leaf nodes of the tree are at the same level. Thus, the tree has maximum number of nodes at all the levels (Figure 9.2).

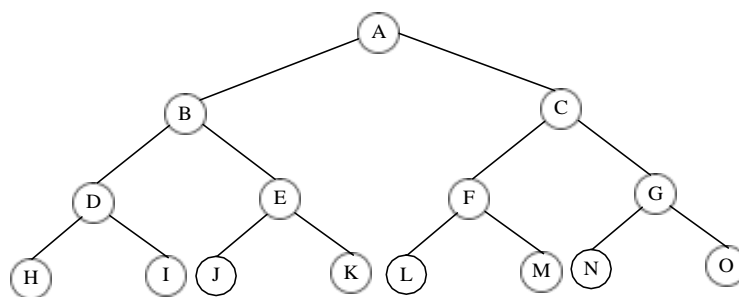
At any level  $n$  of binary tree, there can be at the most  $2^n$  nodes. That is,

At  $n = 0$ , there can be at most  $2^0 = 1$  node.

At  $n = 1$ , there can be at most  $2^1 = 2$  nodes.

At  $n = 2$ , there can be at most  $2^2 = 4$  nodes.

At level  $n$ , there can be at most  $2^n$  nodes.



**Fig. 9.2** Complete Binary Tree

#### Extended Binary Tree

A binary tree is said to be extended binary tree (also known as 2-tree) if all of its nodes are of either zero or two degree. In this type of binary trees, the nodes with degree two (also known as internal nodes) are represented as circles and nodes with degree zero (also known as external nodes) are represented as squares (see Figure 9.3).

## NOTES

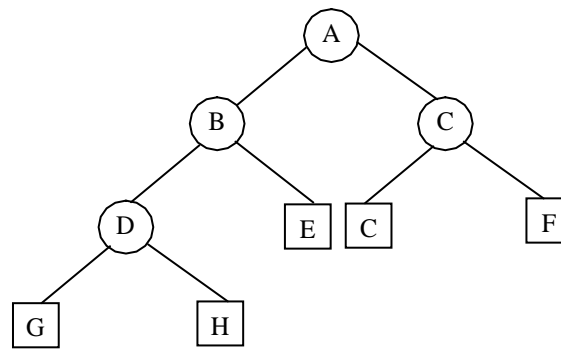


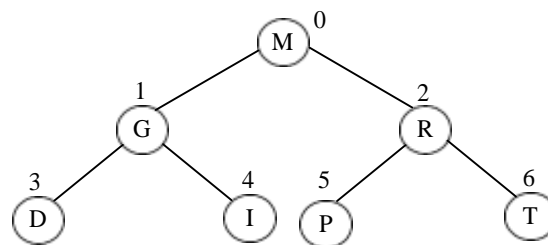
Fig. 9.3 Extended Binary Tree

## 9.2.2 Binary Tree Representations

Like stacks and queues, binary trees can also be represented in two ways in memory—array (sequential) representation and linked representation. In array representation, memory is allocated at compile time and in linked representation, memory is allocated dynamically.

## Array representation

In array representation binary tree is represented sequentially in memory by using single one-dimensional array. A binary tree of height  $n$  may comprise at most  $2^{(n+1)} - 1$  nodes, hence an array of maximum size  $2^{(n+1)} - 1$  is used for representing such a tree. All the nodes of the tree are assigned a sequence number (from 0 to  $(2^{(n+1)} - 1) - 1$ ) level by level. That is, the root node at level 0 is assigned a sequence number 0, and then nodes at level 1 are assigned sequence number in ascending order from left to right, and so on. For example, the nodes of a binary tree of height 2, having 7 ( $2^{(2+1)} - 1$ ) nodes can be numbered as shown in Figure 9.4 (a).



(a) Ordering of Nodes of Binary Tree

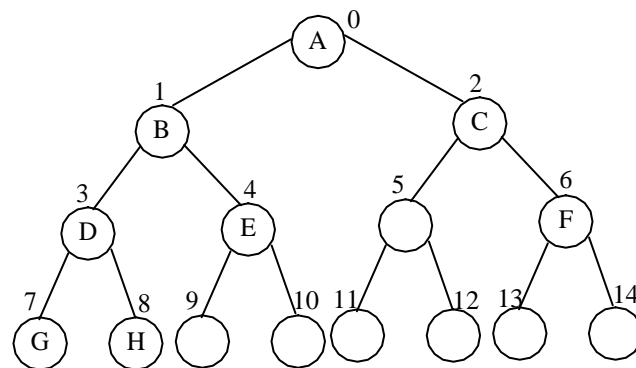
0	1	2	3	4	5	6
M	G	R	D	I	P	T

(b) Nodes of Binary Tree Stored in an Array

Fig. 9.4 Array Representation of Binary Tree

The numbers assigned to the nodes indicates the position (index value) of an array at which that particular node is stored. The array representation of this tree is shown in Figure 9.4(b). It can be observed that if any node is stored at position  $p$ , then its left child node is stored at  $2 * p + 1$  position and its right child node is stored at  $2 * p + 2$  position. For example, in Figure 9.4(b), the node G is stored at position 1, its left child node D is stored at position 3 ( $2 * 1 + 1$ ) and its right child node is stored at position 4 ( $2 * 1 + 2$ ). Notice that if any of the nodes in the tree has empty sub trees (except the leaf nodes), the nodes forming the part of these empty sub trees are also numbered and their values in the corresponding position in the array is NULL. For example, consider a binary tree shown in Figure 9.5(a), its array representation is shown in Figure 9.5(b).

## NOTES



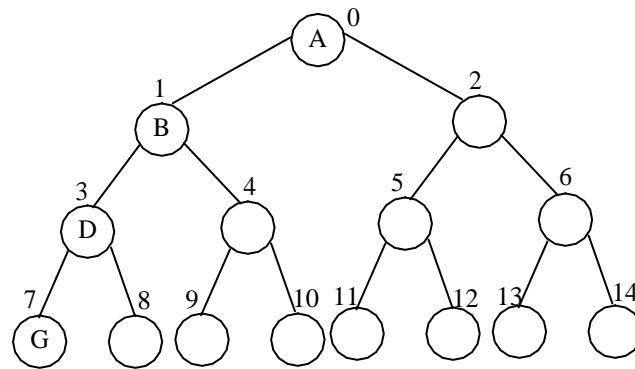
(a) Ordering of Nodes with Empty Sub Trees

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E		F	G	H						

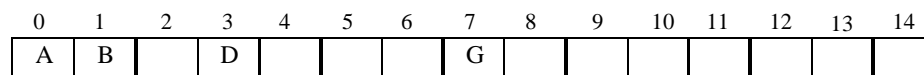
(b) Nodes with Empty Sub Trees Stored in an Array

**Fig. 9.5** Array Representation of Binary Tree with Empty Sub Trees

In this representation, an array of maximum size is declared (to accommodate maximum number of nodes for a binary tree of a given height) before run-time which leads to wastage of lot of memory space in case of unbalanced trees. In unbalanced trees the number of nodes is very small as compared to the maximum number of nodes for a given height. For example, consider an unbalanced tree shown in Figure 9.6(a). Since, this tree is of height 3, an array of size 14 ( $2^{(3+1)} - 1$ ) will be declared to store nodes of this tree. The array representation of this tree is shown in Figure 9.6(b).

**NOTES**

(a) An Unbalanced Binary Tree



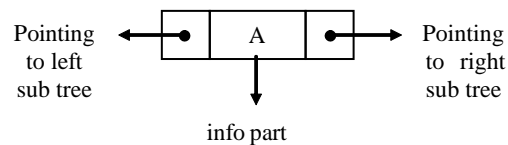
(b) Nodes of an Unbalanced Binary Tree Stored in an Array

**Fig. 9.6** Array Representation of an Unbalanced Binary Tree

It can be observed from this array representation that most of the array positions are NULL, leading to wastage of memory space. Due to this disadvantage of array representation of binary trees, the linked representation of binary trees is preferred.

**Linked Representation**

Linked representation is one of the most common and important way of representing a binary tree in memory. The linked representation of a binary tree is implemented by using a linked list having info part and two pointers. The `info` part contains the data value and two pointers, `left` and `right` are used to point to the left and right sub tree of a node, respectively. The structure of such a node is shown in Figure 9.7.

**Fig. 9.7** Structure of a Node of Binary Tree

To define a node of a binary tree in 'C' language, a self-referential structure can be used whose definition is shown as follows:

```
typedef struct node
{
    int info;
```

```

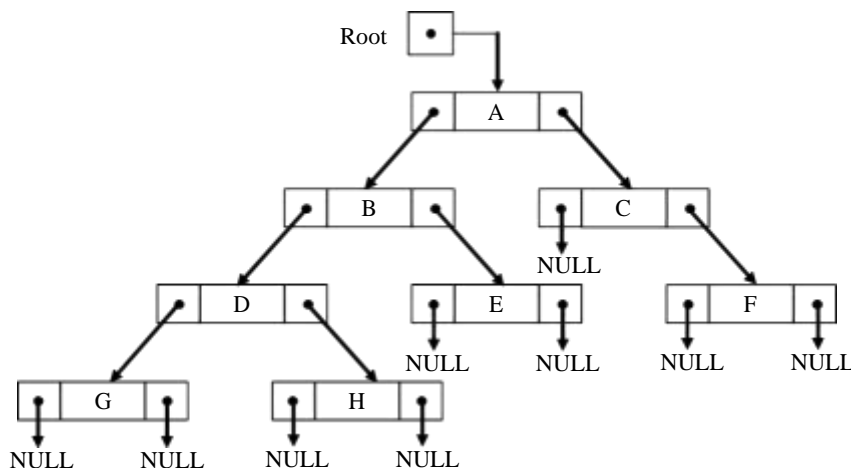
struct node *left;
struct node *right;
}Node;

```

Trees

## NOTES

In linked representation, a pointer variable `Root` of `Node` type is used to point to the root node of a tree. `Root` variable is used for accessing the root and the subsequent nodes of a binary tree. Since binary tree is empty in the beginning, the pointer variable `Root` is initialized with `NULL`. The linked representation of a sample binary tree (see Figure 9.1) is shown in Figure 9.8.



*Fig. 9.8 Linked Representation of a Binary Tree*

## 9.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Depth of the binary tree is the highest level number of any node in a binary tree.
2. The nodes belonging to the same parent node are known as sibling nodes.

## 9.4 SUMMARY

- A binary tree is a special type of tree, which can either be empty or has finite set of nodes. The nonempty left sub tree and the right sub tree are also binary trees.

## NOTES

- Degree of a node is equal to the number of its child nodes.
- Since binary tree is a multilevel data structure, each node belongs to a particular level number.
- Depth of the binary tree is the highest level number of any node in a binary tree.
- The nodes belonging to the same parent node are known as sibling nodes.
- There are various forms of binary trees that are formed by imposing certain restrictions on them. Some of the variations of binary trees are—complete binary tree and extended binary tree.
- A binary tree is said to be complete binary tree if all the leaf nodes of the tree are at the same level.
- A binary tree is said to be extended binary tree (also known as 2-tree) if all of its nodes are of either zero or two degree.
- Like stacks and queues, binary trees can also be represented in two ways in memory—array (sequential) representation and linked representation.
- In array representation binary tree is represented sequentially in memory by using single one-dimensional array.
- The numbers assigned to the nodes indicates the position (index value) of an array at which that particular node is stored.
- Linked representation is one of the most common and important way of representing a binary tree in memory.

---

## 9.5 KEY WORDS

---

- **Complete Binary tree:** It is a binary tree if all the leaf nodes of the tree are at the same level.
- **Extended Binary tree:** It is a binary tree if all of its nodes are of either zero or two degree.

---

## 9.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

### Short-Answer Questions

1. Write a short note on binary trees.
2. Show a diagrammatic representation of a binary tree.
3. What do you mean by ancestor and descendent?

## Long-Answer Questions

1. Write a detailed note on the forms of binary trees.
2. How is a complete binary tree different from an extended binary tree? Explain.
3. Write a detailed note on Binary tree representations.
4. “Linked representation is one of the most common and important way of representing a binary tree in memory.” Explain.

## NOTES

### 9.7 FURTHER READINGS

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

### 9.8 LEARNING OUTCOMES

- Binary trees
- The nature of binary trees
- The forms of binary trees
- Understand the concept of extended binary tree

## NOTES

---

# UNIT 10 BINARY TREE OPERATIONS/ APPLICATIONS

---

### Structure

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Traversing Binary trees
- 10.3 Binary Search Tree
- 10.4 Traversing a Binary Search Tree
- 10.5 Answers to Check Your Progress Questions
- 10.6 Summary
- 10.7 Key Words
- 10.8 Self Assessment Questions and Exercises
- 10.9 Further Readings
- 10.10 Learning Outcomes

---

## 10.0 INTRODUCTION

---

In the previous unit you have learnt about binary trees. This unit will explain about binary tree operations and applications. A binary tree can be traversed in three different ways— in-order, pre-order and post-order traversal. You will learn these in detail along with the basic concepts of binary search tree.

---

## 10.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss about traversing a binary tree
  - Analyze what happens when a root node is visited in pre-order traversal
  - Understand the different ways in which a tree can be traversed
  - Explain binary search tree

---

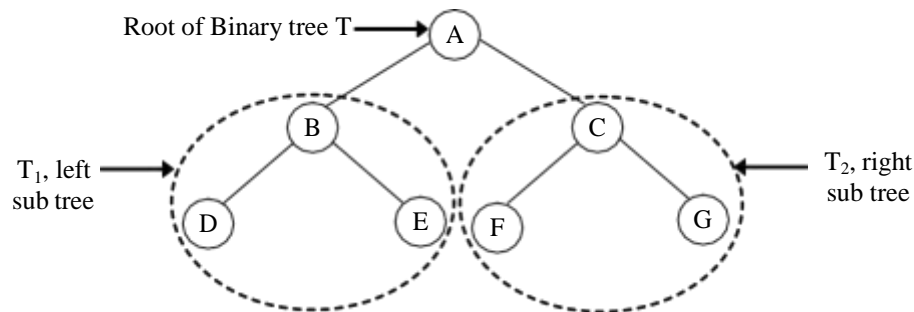
## 10.2 TRAVERSING BINARY TREES

---

Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once. The three different ways in which a tree can be traversed are—



in-order, pre-order and post-order traversal. The main difference in these traversal methods is based on the order in which the root node is visited. Note that in all the traversals the left sub tree is always traversed before the traversal of the right sub tree. To understand these traversal methods, consider a simple binary tree T, shown in Figure 10.1.



**Fig. 10.1** A Binary Tree

### Pre-order

In pre-order traversal, the root node is visited before traversing its left and right sub trees. Steps for traversing a nonempty binary tree in pre-order are as follows:

1. Visit the root node R .
2. Traverse the left sub tree of root node R in pre-order.
3. Traverse the right sub tree of root node R in pre-order.

For example, in binary tree T (shown in Figure 10.1), the root node A is traversed before traversing its left sub tree and the right sub tree. In the left sub tree T1, the root node B (of left sub tree T1) is traversed before traversing the nodes D and E. After traversing the root node of binary tree T and traversing the left sub tree T1, the right sub tree T2 is also traversed following the same procedure. Hence, the resultant pre-order traversal of the binary tree T is A, B, D, E, C, F, G.

### In-order

In in-order traversal, the root node is visited after the traversal of its left sub tree and before the traversal of its right sub tree. Steps for traversing a nonempty binary tree in in-order are as follows:

1. Traverse the left sub tree of root node R in in-order.
2. Visit the root node R .
3. Traverse the right sub tree of root node R in in-order.

## NOTES

## NOTES

For example, in binary tree  $T$  (shown in Figure 10.1), the left sub tree  $T_1$  is traversed before traversing the root node  $A$ . In the left sub tree  $T_1$ , the node  $D$  is traversed before traversing its root node  $B$  (of left sub tree  $T_1$ ). After traversing the node  $D$  and  $B$ , the node  $E$  is traversed. Once the traversal of left sub tree  $T_1$  and the root node  $A$  of binary tree  $T$  is complete, the right sub tree  $T_2$  is traversed following the same procedure. Hence, the resultant in-order traversal of the binary tree  $T$  is  $D, B, E, A, F, C, G$ .

### Post-order

In post-order traversal, the root node is visited after traversing its left and right sub trees. Steps for traversing a nonempty binary tree in post-order are as follows:

1. Traverse the left sub tree of root node  $R$  in post-order.
2. Traverse the right sub tree of root node  $R$  in post-order.
3. Visit the root node  $R$ .

For example, in binary tree  $T$  (shown in Figure 10.1), the root node  $A$  is traversed after traversing its left sub tree and the right sub tree. In the left sub tree  $T_1$ , the root node  $B$  (of left sub tree  $T_1$ ) is traversed after traversing the nodes  $D$  and  $E$ . Similarly, the nodes of right sub tree  $T_2$  are traversed following the same procedure. After traversing the left sub tree ( $T_1$ ) and right sub tree ( $T_2$ ), the root node  $A$  of binary tree  $T$  is traversed. Hence, the resultant post-order traversal of the binary tree  $T$  is  $D, E, B, F, G, C, A$ .

---

## 10.3 BINARY SEARCH TREE

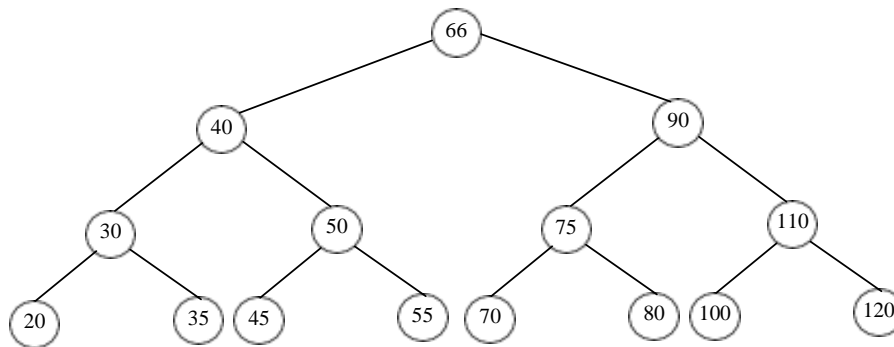
---

Binary search tree, also known as binary sorted tree, is a kind of a binary tree, which satisfies the following conditions (Figure 10.2):

1. The data value in each node is a key (unique) value, that is, no two nodes can have identical values.
2. The data values in the nodes of the left sub tree, if exists, is smaller than the value in the root node.
3. The data values in the nodes of the right sub tree, if exists, is greater than the value in the root node.
4. The left and the right sub trees, if exists, are also binary search trees.

In other words, values in the left sub tree of a root node are smaller than the value of the root node, and the values in the right sub tree are greater than the value of the root node. This rule is applicable on all the subsequent sub trees in a

binary search tree. In addition, each and every value in binary search tree is unique, that is, no two nodes in it can have identical values.



**Fig. 10.2** Binary Search Tree

There are various operations that can be performed on the binary search trees. Some of these are searching a node, insertion of a new node, traversal of a tree and deletion of a node.

### Searching a Node

Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order. The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left sub tree and if the element is greater than the value in the root node, then the searching will proceed to the right sub tree. This process is repeated until either the element to be searched is found or NULL value is encountered.

For example, consider a sample binary search tree given in Figure 10.2. The steps to search element 45 are given as follows:

1. Compare the element 45 with the value in root node (66). Since 45 is smaller than 66, move it to its left sub tree.
2. Compare the element 45 with the value (40) appearing in the left sub tree. Since 45 is greater than the 40, move it to its right sub tree.
3. Now, compare the element 45 with the value (50) appearing in the right sub tree. Since 45 is smaller than 50, move it to its left sub tree.
4. In the next step, compare the element 45 with the value (45) appearing in the left sub tree. Since 45 is equal to the value (45) stored in this node, the required element is found. Therefore, terminate the procedure.

## NOTES

NOTES

In case the value 48 is to be searched, the first four steps are same. After the step 4, the right sub tree of 45 will be accessed, this is NULL indicating the end of the tree. Therefore, the element is not found in the tree and the search is unsuccessful.

Algorithm 10.1 Searching in a Binary Search Tree

search(item, ptr)  
  
1. If !(ptr)  
    Print "Element not found!" and go to step 3End  
    If  
2. If item < ptr->info  
    Call search(item, ptr->left)  
    Else If item > ptr->info  
    Call search(item, ptr->right)  
    Else  
    Print "Element found."  
    End If  
3. End

10.4 TRAVERSING A BINARY SEARCH TREE

Traversing a binary search tree is same as traversing a binary tree. That is, binary search tree can also be traversed in three different ways—pre-order, in-order and post-order. For example, consider the tree shown in Figure 10.2. The pre-order, in-order and post-order traversal of this tree is as follows:

Pre-order traversal: 66 40 30 20 35 50 45 55 90 75 70 80 110 100 120

In-order traversal: 20 30 35 40 45 50 55 66 70 75 80 90 100 110 120

Post-order traversal: 20 35 30 45 55 50 40 70 80 75 100 120 110 90 66

It can be observed that when a binary search tree is traversed in in-order, it results in the sequence of elements in ascending order. The algorithms for traversing tree in pre-order, in-order and post-order are recursive in nature, which are given in Algorithms 10.2 to 10.4:

**Algorithm 10.2 Pre-order Traversal**

```
preorder(ptr)

1. If ptr != NULL
    Print ptr->info          //ptr is temporary pointer initialised with Root
    Call preorder(ptr->left)
    Call preorder(ptr->right)
    End If
2. End
```

**Algorithm 10.3 In-order Traversal**

```
inorder(ptr)

1. If ptr != NULL
    Call inorder(ptr->left) //ptr is temporary pointer initialised with Root
    Print ptr->info
    Call inorder(ptr->right)
    End If
2. End
```

**Algorithm 10.4 Post-order Traversal**

```
postorder(ptr)

1. If ptr != NULL
    Call postorder(ptr->left) //ptr is temporary pointer initialised with Root
    Call postorder(ptr->right)
    Print ptr->info
    End If
2. End
```

**Example 10.1:** A program to illustrate the various operations performed on binary search tree (in case of deletion of a node with two child nodes, largest value from left sub tree (in-order predecessor) is used for replacement).

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

typedef struct node {
    int info;
    struct node *left;
    struct node *right;
}Node;

int nodes, leaves;
```

**NOTES**

## NOTES

```
/*Function prototypes*/
void insert_node(int, Node **);
void search(int, Node *);
void print_treeform(Node *, int);
void preorder(Node *);
void inorder(Node *);
void postorder(Node *);
void count_nodes(Node *);
void count_leaves(Node *);
void del(Node **, Node *);
void del_node(int, Node **);

void main()
{
    int choice, n;
    Node *root=NULL;
    do
    {
        clrscr();
        printf("\nMain Menu");
        printf("\n1. Insert");
        printf("\n2. Display in tree form");
        printf("\n3. Pre-order traversal of tree");
        printf("\n4. In-order traversal of tree");
        printf("\n5. Post-order traversal of tree");
        printf("\n6. Number of nodes");
        printf("\n7. Number of leaves");
        printf("\n8. Searching");
        printf("\n9. Delete");
        printf("\n10.Exit");
        printf("\nEnter your choice . . . ");
        scanf("%d", &choice);
        switch(choice)
        {
```

```
case 1 : printf("\nEnter data for new node :
            ");
        scanf("%d", &n);
        insert_node(n, &root);
        break;
case 2 : printf("\nTree in tree form ->\n");
        if(!root)
            print_treeform(root, 1);
        else
            printf("Tree is empty!!");
        break;
case 3 : printf("\nPre-order traversal of tree
            ->\n\n");
        if(!root)
            preorder(root);
        else
            printf("Tree is empty!!");
        break;
case 4 : printf("\nIn-order traversal of tree
            ->\n\n");
        if(!root)
            inorder(root);
        else
            printf("Tree is empty!!");
        break;
case 5 : printf("\nPost-order traversal of
            tree ->\n\n");
        if(!root)
            postorder(root);
        else
            printf("Tree is empty!!");
        break;
case 6 : if(root==NULL)
        nodes=0;
        else
```

## NOTES

## NOTES

```
        nodes=1;
        count_nodes(root);
        printf("\nNumber of nodes are : %d",
            nodes);
        break;
    case 7 : leaves=0;
        count_leaves(root);
        printf("\nNumber of leaves are : %d",
            leaves);
        break;
    case 8 : printf("\nEnter value of node to be
        searched : ");
        scanf("%d", &n);
        search(n, root);
        break;
    case 9 : printf("\nEnter value of node to be
        deleted : ");
        scanf("%d", &n);
        del_node(n, &root);
        break;
    case 10 : printf("\nNormal termination of
        program.");
        break;
    default : printf("\nWrong Choice !!");
}
getch();
}while(choice!=10);
}

/*Function to insert node in a tree*/
void insert_node(int item, Node **ptr)
{
    if(!(*ptr))
    {
```



```
        (*ptr)=(Node*) malloc(sizeof(Node));
        (*ptr)->info=item;
        (*ptr)->left=NULL;
        (*ptr)->right=NULL;
    }
    if(item<(*ptr)->info)
        insert_node(item,&((*ptr)->left));
    else if(item>(*ptr)->info)
        insert_node(item,&((*ptr)->right));
}

/*Function to print tree in tree format*/
void print_treeform(Node *ptr, int level)
{
    int i;
    if(ptr)
    {
        print_treeform(ptr->right, level+1);
        printf("\n");
        for(i=0;i<level;i++)
            printf("    ");
        printf("%d", ptr->info);
        print_treeform(ptr->left, level+1);
    }
}

/*Funtion to print tree in pre-order*/
void preorder(Node *ptr)
{
    if(ptr)
    {
        printf("%d ", ptr->info);
        preorder(ptr->left);
        preorder(ptr->right);
    }
}
```

## NOTES

## NOTES

```
/*Function to print tree in in-order*/
void inorder(Node *ptr)
{
    if(ptr)
    {
        inorder(ptr->left);
        printf("%d ", ptr->info);
        inorder(ptr->right);
    }
}

/*Function to print tree in post-order*/
void postorder(Node *ptr)
{
    if(ptr)
    {
        postorder(ptr->left);
        postorder(ptr->right);
        printf("%d ", ptr->info);
    }
}

/*Function to count number of nodes in a tree*/
void count_nodes(Node *ptr)
{
    if(ptr != NULL)
    {
        if(ptr->left != NULL)
        {
            nodes++;
            count_nodes(ptr->left);
        }
    }
}
```

```
        if(ptr->right != NULL)
        {
            nodes++;
            count_nodes(ptr->right);
        }
    }
}

/*Function to count number of leaves in a tree*/
void count_leaves(Node *ptr)
{
    if(ptr != NULL)
    {
        if((ptr->left==NULL) && (ptr->right==NULL))
            leaves++;
        else
            count_leaves(ptr->left);
            count_leaves(ptr->right);
    }
}

/*Function to search a node in a tree*/
void search(int item, Node *ptr)
{
    if(!ptr)
    {
        printf("Element not found.");
        return;
    }
    else if(item<ptr->info)
        search(item, ptr->left);
    else if(item>ptr->info)
        search(item, ptr->right);
    else
    {

```

## NOTES

## NOTES

```
        printf("Element found.");
    }
}

/*Funtion to delete a node form tree*/
void del_node(int item, Node **ptr)
{
    Node *save;
    if(!(*ptr))
    {
        printf("\nItem does not exist.");
        return;
    }
    else
    {
        if(item<(*ptr)->info)
            del_node(item, &((*ptr)->left));
        else
            if(item>(*ptr)->info)
                del_node(item, &((*ptr)->right));
            else if(item==(*ptr)->info)
            {
                save=*ptr;
                if(save->right==NULL)
                {
                    *ptr=save->left;
                    free(save);
                }
                else
                    if(save->left==NULL)
                    {
                        *ptr=save->right;
                        free(save);
                    }
                else
            }
    }
}
```

```
        del(&(save->left), save);
    }
}
return;
}

/*Called from Del_node() function to delete nodes with
child nodes*/
void del(Node **p, Node *q)
{
    Node *delnode;
    if((*p)->right != NULL)
        del(&((*p)->right), q);
    else
    {
        delnode=*p;
        q->info=(*p)->info;
        *p=(*p)->left;
        free(delnode);
    }
    return;
}
```

**The output of program is**

```
Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 1
```

**NOTES**

Enter data for new node : 66

## NOTES

```
: /* Similarly insert values 40 90 30 50 75 110 20
35 45 55 70 80 100 120 in this
: order*/
```

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 2

Tree in tree form ->

```

                120
            110
        100
    90
        80
    75
        70
66
        55
    50
        45
    40
        35
    30
        20
```

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 3

Pre-order traversal of tree ->

66 40 30 20 35 50 45 55 90 75 70 80 110 100  
120

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 4

In-order traversal of tree ->

20 30 35 40 45 50 55 66 70 75 80 90 100 110  
120

## NOTES

## NOTES

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 5

Post-order traversal of tree ->

20 35 30 45 55 50 40 70 80 75 100 120 110 90  
66

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 6

Number of nodes are : 15

Main Menu

1. Insert
2. Display in tree form



```
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 7
```

Number of leaves are : 8

Main Menu

```
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 8
```

```
Enter value of node to be searched : 120
Element found.
```

Main Menu

```
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
```

## NOTES

## NOTES

```
8. Searching
9. Delete
10. Exit
Enter your choice . . . 9

Enter value of node to be deleted : 70

Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 9

Enter value of node to be deleted : 75

Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 2
```

Tree in tree form ->

```

                120
              110
            100
          90
        80
      66
    55
  50
  45
40
  35
  30
  20
```

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 9

Enter value of node to be deleted : 40

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree

## NOTES

## NOTES

```
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 2
```

Tree in tree form ->

```

                120
              110
            100
          90
        80
      66
        55
      50
        45
    35
      30
        20
```

Main Menu

```
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 10
```

Normal termination of program.

## NOTES

### 10.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once.
2. In pre-order traversal, the root node is visited before traversing its left and right sub trees.
3. Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order.

### 10.6 SUMMARY

- Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once.
- The three different ways in which a tree can be traversed are— in-order, pre-order and post-order traversal.
- The main difference in these traversal methods is based on the order in which the root node is visited.
- In pre-order traversal, the root node is visited before traversing its left and right sub trees.
- In in-order traversal, the root node is visited after the traversal of its left sub tree and before the traversal of its right sub tree.
- In post-order traversal, the root node is visited after traversing its left and right sub trees.
- There are various operations that can be performed on the binary search trees. Some of these are searching a node, insertion of a new node, traversal of a tree and deletion of a node.
- Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order.

## NOTES

---

### 10.7 KEY WORDS

---

- **Traversing:** It is the process of visiting each and every data item of the data structure exactly once.
- **Binary tree:** It is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

---

### 10.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. What is binary search tree?
2. What do you mean by traversing binary trees?
3. How do you search a node in binary tree?

#### Long-Answer Questions

1. “The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left sub tree and if the element is greater than the value in the root node, then the searching will proceed to the right sub tree.” Explain in detail.
2. Write a program to illustrate the various operations performed on binary search tree. NOTE: In case of deletion of a node with two child nodes, largest value from left sub tree (in-order predecessor) is used for replacement.
3. Write the algorithms for Pre-order, In-order and Post order traversal.

---

### 10.9 FURTHER READINGS

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

---

### 10.10 LEARNING OUTCOMES

---

- Learn about traversing a binary tree
- What happens when a root node is visited in pre-order traversal
- Understand the different ways in which a tree can be traversed
- Binary search tree

---

# UNIT 11 OPERATIONS ON BINARY TREE

---

*Operations on  
Binary Tree*

## NOTES

### Structure

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Insertion and Deletion Operations
- 11.3 Hashing Techniques
- 11.4 Answers to Check Your Progress Questions
- 11.5 Summary
- 11.6 Key Words
- 11.7 Self Assessment Questions and Exercises
- 11.8 Further Readings
- 11.9 Learning Outcomes

---

## 11.0 INTRODUCTION

---

In the previous unit, we considered a particular kind of a binary tree called a Binary Search Tree (BST). A binary tree is a binary search tree (BST) if and only if an in order traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently. This unit will explain the various types of operations on binary tree.

---

## 11.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss the operations in binary tree
  - Understand the insertion and deletion operations
  - List the hashing techniques
  - Analyze the division remained method

---

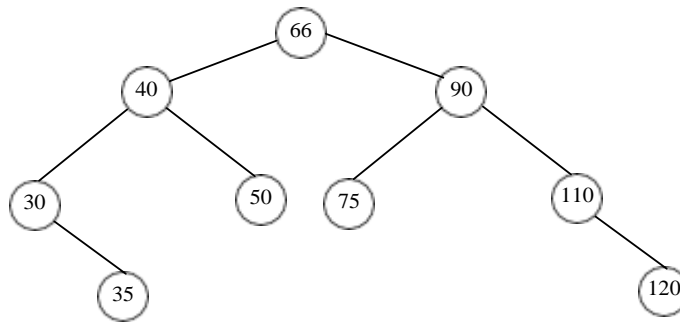
## 11.2 INSERTION AND DELETION OPERATIONS

---

Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree. The difference is that in case of insertion, appropriate null pointer is searched where new node can be inserted. The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.

## NOTES

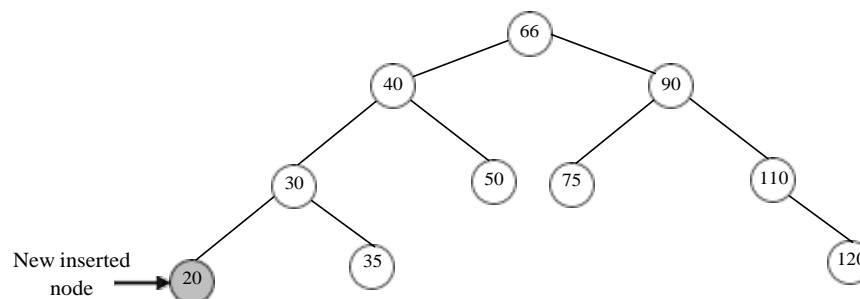
Here are two cases of insertion in a tree—first, insertion into an empty tree and second, insertion into a nonempty tree. In case the tree is initially empty, the new node to be inserted becomes its root node. In case the tree is nonempty, appropriate position is determined for insertion. For this, first of all the value in the new node is compared with the root node of the tree. If the value in the new node is less than the value in the root node, the new node is added as the left leaf if the left sub tree is empty, otherwise the search continues in the left sub tree. On the other hand, if the value in the new node is greater than the value in the root node, the new node is added as the right leaf if the right sub tree is empty, otherwise the search continues in the right sub tree.



**Fig. 11.1** A Sample Binary Search Tree

For example, consider a sample binary search tree shown in Figure 11.1. For inserting elements 20 and 80, follow the steps given:

1. Compare 20 with the value in the root node, that is, 66. Since 20 is smaller than 66, move to the left sub tree.
2. Finding that the left pointer of root node is non-null, compare 20 with the value (40) in this node. Since 20 is smaller than 40, move to the left sub tree.
3. Again, the left pointer of the current node is non-null, compare 20 with the value (30) in this node. Since 20 is smaller than 30, move to the left sub tree.
4. Now, the left pointer is null, thus 20 will be inserted at this position. After insertion, the tree will appear as shown in Figure 11.2.



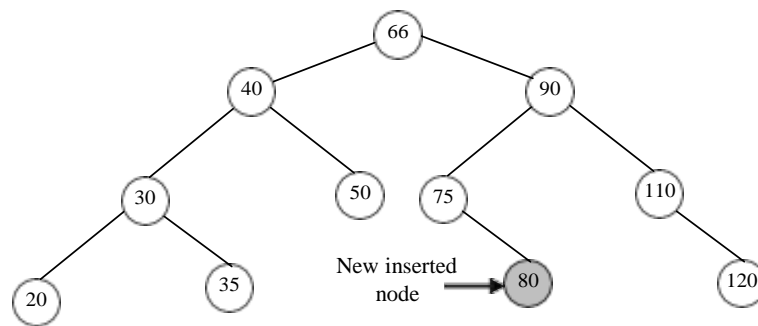
**Fig. 11.2** Insertion of a Node with Value 20



Steps for inserting the element 80 are as follows:

1. Compare 80 with the value in root node 66. Since 80 is greater than 66, move to the right sub tree.
2. Finding that the right pointer of root node is non-null, compare 80 with the value (90) in this node. Since 80 is smaller than 90, move to the left sub tree.
3. Again, the left pointer of the current node is non-null, compare 80 with the value (75) in this node. Since 80 is greater than 75, move to the right sub tree.
4. Now, the right pointer is null, thus 80 will be inserted at this position. After insertion, the tree will appear as shown in Figure 11.3.

## NOTES



**Fig. 11.3** Insertion of a Node with Value 80

### Algorithm 11.1 Insertion into a Binary Search Tree

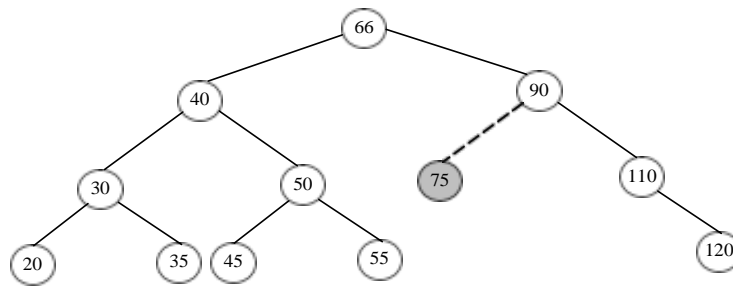
```
insert_node(item, ptr)
1. If !(ptr)
    Allocate memory for ptr
    Set ptr->info = item
    Set ptr->left = NULL
    Set ptr->right = NULL
Else
    If item < ptr->info
        Call insert_node(item, ptr->left)
    Else
        Call insert_node(item, ptr->right)
    End If
End If
2. End
```

## Deleting a Node

Deletion of a node from a binary search tree involves two steps—first, searching the desired node and second, deleting the node. Whenever a node is deleted from a tree, it must be ensured that the tree remains a binary search tree, that is, the sorted order of the tree must not be disturbed. The node being deleted may have zero, one or two child nodes. On the basis of the number of child nodes to be deleted, there are three cases of deletion which are discussed as follows:

**Case 1:** If the node to be deleted has no child node, it is deleted by making its parent's pointer pointing to NULL and de-allocating memory allocated to it.

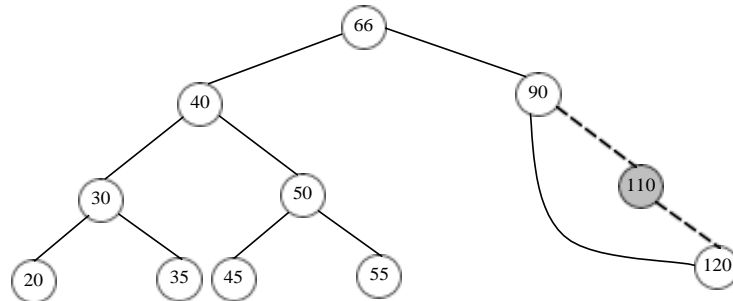
## NOTES



**Fig. 11.4** Deletion of a Node Having No Child Node

For example, the node with value 75 is to be deleted from the tree shown in Figure 11.4. Since this node has no child node, its parent's (90) left pointer will be made to point to NULL and the memory space of the node (75) is de-allocated.

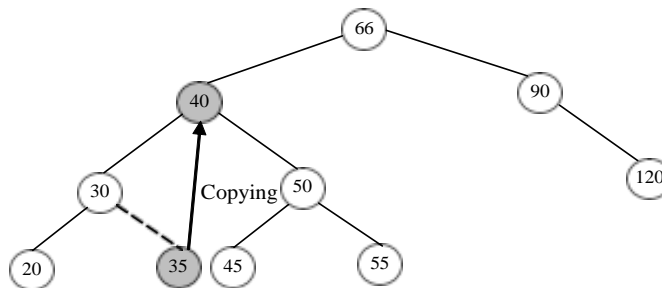
**Case 2:** If the node to be deleted has only one child node, it is deleted by adjusting its parent's pointer pointing to its only child and de-allocating memory allocated to it.



**Fig. 11.5** Deletion of a Node Having Only One Child

For example, the node with value 110 is to be deleted from the tree shown in Figure 11.5. Since this node has one child node, its parent's (90) right pointer will be made to point to its child node (120) and memory space of the node (110) is de-allocated.

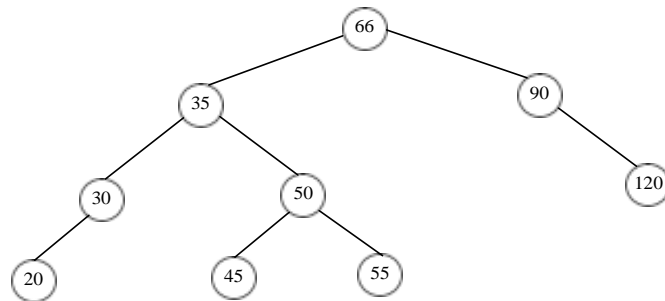
**Case 3:** If the node to be deleted has two child nodes, it is deleted by replacing its value by largest value in the left sub tree (in-order predecessor) or by smallest value in the right sub tree (in-order successor). The node whose value is used for replacement is then deleted using case 1 or case 2.



**Fig. 11.6** Deletion of a Node Having Two Child Nodes

For example, the node with value 40 is to be deleted from the tree shown in Figure 11.6. Since this node has two sub trees or child nodes, a value has to be searched from its sub trees which can be used for its replacement. The value that will be used for replacement can either be largest value from its left sub tree (35) or smallest value from its right sub tree (45). Suppose, the value 35 is selected for this purpose, then value 35 is copied in the node with the value 40. After this, the right pointer of parent node (30) of the node used for replacement (35) is made to point to NULL and memory allocated to the node with value 35 is de-allocated. As a result of deletion of this node the order of tree is maintained. The final structure of the tree after deletion of node 40 will be as shown in Figure 11.7.

## NOTES



**Fig. 11.7** Binary Search Tree after Deletion

### Algorithm 11.2 Deletion from Binary Search Tree

del\_node(item, ptr)

1. If !(ptr)
  - Print "Item does not exist." and go to step 3
2. If item < ptr->info
  - Call del\_node(item,&(ptr->left))
- Else
  - If item > ptr->info
    - Call del\_node(item,&(ptr->right))
  - Else
    - If item = ptr->info
      - Set save = ptr
      - If save->right = NULL
        - Set ptr = save->left
        - Deallocate save
      - Else
        - If save->left = NULL
          - Set ptr = save->right
          - Deallocate save
        - Else
          - Call del(&(save->left),save)
      - End If
    - End If
  - End If
  - End If
3. End

del(p, q) // q is the node to be deleted, p is the node whose value  
//is used for replacing the value in q and p is de-allocated

1. If p->right != NULL
  - Call del(&(p->right),q)
- Else
  - Set delnode = p
  - Set q->info = p->info
  - Set p = p->left
  - Deallocate delnode
- End If
2. End

## NOTES

---

### 11.3 HASHING TECHNIQUES

---

Hashing (also known as hash addressing) is generally applied to a file  $F$  containing  $R$  records. Each record contains many fields, out of these one particular field may uniquely identify the records in the file. Such a field is known as primary key (denoted by  $k$ ). The values  $k_1, k_2, \dots$  in the key field are known as *keys* or *key values*.

Whenever a key is to be inserted in the hash table, a hash function is applied on it, which yields an index for the key. It is then inserted at that index in the hash table. However, since there is a finite number of locations in the hash table and virtually an infinite number of keys to be stored, it is quite possible that two distinct keys hash to the same index in the hash table. The situation in which a key hashes to an index which is already occupied by another key is called collision. It should be resolved by finding some other location to insert the new key. This process of finding another location is called collision resolution.

Although, a variety of collision resolution techniques have been developed, however, the possibility of collision should be minimized. It makes the study of hash function an implied requirement because the hash function is responsible for specifying the location for a new key. Therefore, the topic of hashing comprises two major sub-parts, namely, hash functions and collision resolutions.

Since, the keys are inserted by applying hash functions on them, searching a key in the hash table is straightforward. Simply, the same hash function is applied on the key to be searched, which yields the index at which it may be found. Then the key at that location is compared with the desired key and if they are matched, the search is successful.

#### Hash Functions

A hash function  $h$  is simply a mathematical formula that maps the key to some slot in the hash table  $T$ . Thus, we can say that the key  $k$  hashes to slot  $h(k)$ , or  $h(k)$  is the hash value of key  $k$ . If the size of the hash table is  $N$ , then the index of the hash table ranges from 0 to  $N-1$ . A hash table with  $N$  slots is denoted by  $T[N]$ .

If the input keys are integers, then applying hash function on them is simple. However, if the input keys are strings, then they are first converted into integers before applying the hash function. For this, the numeric (ASCII) code associated with characters can be used in converting character values into integers.

There are a number of hash functions available, however, the one which is easy to compute and ensures that two distinct values hash to different location in the hash table is desirable. It is quite simple to design a hash function which is efficient and easy to compute. However, no hash function guarantees to achieve the second condition always. However, a good hash function should keep the number of collisions as minimum as possible. The following are some of the commonly used hash functions.

**Division-remainder method**

The division-remainder is the simplest and most commonly used method. In this method, the key  $k$  is divided by the number of slots  $N$  in the hash table, and the remainder obtained after division is used as an index in the hash table. That is, the hash function is

$$h(k) = k \bmod N$$

where,  $\bmod$  is the modulus operator. Different languages have different operators for calculating the modulus. In C/C++, % operator is used for computing the modulus.

Note that this function works well if the index ranges from 0 to  $N-1$  (like in C/C++). However, if the index ranges from 1 to  $N$ , the function will be

$$h(k) = k \bmod N + 1$$

This technique works very well if  $N$  is either a prime number not too close to a power of two. Moreover, since this technique requires only a single division operation, it is quite fast.

For example, consider a hash table with  $N=101$ . The hash value of the key value 132437 can be calculated as follows:

$$h(132437) = 132437 \bmod 101 = 26$$

**Multiplication method**

In multiplication method, first the key  $k$  is multiplied by a constant  $C$ , where  $0 < C < 1$ , and the fractional part of the product  $kC$  is extracted. In the second step, this fractional part is multiplied by  $N$ , and the floor of the result is taken as the hash value. The floor of a value  $x$  denoted by  $\lfloor x \rfloor$  is the largest integer less than or equal to  $x$ . That is, the hash function is

$$h(k) = \lfloor N (kC \bmod 1) \rfloor$$

where,  $kC \bmod 1$  represents the fractional part of  $kC$ , calculated as  $kC - \lfloor kC \rfloor$

Though, the constant  $C$  can take any value between 0 and 1, but still the function gives better performance for some values of  $C$ . In his study, Knuth has suggested that the following value of  $C$  is likely to work reasonably well.

$$C = (\sqrt{5} - 1) / 2 = 0.618033988749894848...$$

For example, consider a hash table with  $N=101$  and  $C=0.6180339$ . The hash value of the key 132437 can be calculated as follows:

$$\begin{aligned} h(132437) &= \lfloor 101 * ((132437 * 0.6180339...) \bmod 1) \rfloor \\ &= \lfloor 101 * (81850.5673680698... \bmod 1) \rfloor \\ &= \lfloor 101 * 0.5673680698... \rfloor \\ &= \lfloor 57.3041750498... \rfloor \\ &= 57 \end{aligned}$$

**NOTES**

## NOTES

### Folded key method

The folded key is also a two-step process. In the first step, the key  $k$  is divided into several groups from the left most digits, where each group contains  $n$  number of digits, except the last one which may contain lesser number of digits. In the next step, these groups are added together, and the hash value is obtained by ignoring the last carry (if any).

For example, if the hash table has 100 slots, then each group will have two digits; and the sum of the groups after ignoring the last carry will also be a 2-digit number between 0 and 99. The hash value for the key value 132437 is computed as follows:

1. The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 13, 24, and 37.
2. These groups are then added like  $13 + 24 + 37 = 74$ . The sum 74 is now used as the hash value for the key value 132437.

Similarly, the hash value of another keyvalue, say 6217569, can be calculated as follows:

1. The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 62, 17, 56, and 9.
2. These groups are then added, like,  $62 + 17 + 56 + 9 = 144$ . The sum 44 after ignoring the last carry 1 is now used as the hash value for the key value 6217569.

### Midsquare method

This method also operates in two steps. First, the square of the key  $k$  (that is,  $k^2$ ) is calculated, and then some of the digits from left and right ends of  $k^2$  are removed. The number obtained after removing the digits is used as the hash value. Note that the digits at the same position of  $k^2$  must be used for all the keys.

For example, consider a hash table with  $N=1000$ . The hash value of the key value 132437 can be calculated as follows:

1. The square of the key value is calculated, which is, 17539558969.
2. The hash value is obtained by taking 5<sup>th</sup>, 6<sup>th</sup> and 7<sup>th</sup> digits counting from right, which is, 955.

Similarly, the hash value of another keyvalue, say 6217569, can be calculated as follows:

1. The square of the key value is calculated, which is, 38658164269761
2. The hash value is obtained by taking 5<sup>th</sup>, 6<sup>th</sup> and 7<sup>th</sup> digits counting from right, which is, 426.

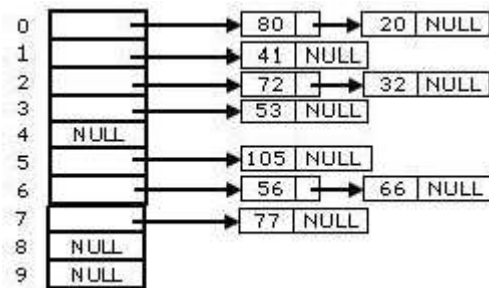
## Collision Resolution Techniques

The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys, because the number of key values is much larger than the number of available locations in the hash table. Due to this, sometimes the problem called collision occurs. Since one cannot eliminate collisions altogether, one needs some mechanisms to deal with them. There are several ways for resolving collisions, the two most common techniques used are separate chaining and open addressing.

### Separate chaining

In this technique, a linked list of all the key values that hash to the same hash value is maintained. Each node of the linked list contains a key value and the pointer to the next node. Each index  $i$  ( $0 \leq i < N$ ) in the hash table contains the address of the first node of the linked list containing all the keys that hash to the index  $i$ . If there is no key value that hashes to the index  $i$ , the slot contains NULL value. Therefore, in this method, a slot in the hash table does not contain the actual key values; rather it contains the address of the first node of the linked list containing the elements that hash to this slot.

Consider the key values 20, 32, 41, 66, 72, 80, 105, 77, 56, and 53 that need to be hashed using the simple hash function  $h(k) = k \bmod 10$ . The keys 20 and 80 hash to index 0, key 41 hashes to index 1, keys 32 and 72 hashes to index 2, key 53 hashes to index 3, key 105 hashes to index 5, keys 66 and 56 hashes to index 6 and finally the key 77 hashes to index 7. The collision is handled using the separate chaining (also known as synonyms chaining) technique as shown in Figure 11.8.



**Fig. 11.8** Collision resolution by Separate Chaining

Note that a new element can be inserted either at the beginning or at the end of the list. Generally, the elements are inserted in the beginning of the list because it is simpler to implement, and moreover, it frequently happens that the elements which are added recently are the most likely to be accessed in the near future.

The main disadvantage of separate chaining is that it makes use of pointers, which slows down the algorithm a bit because of the time required in allocating and deallocating the memory. Moreover, maintaining another data structure (that is, linked list) in addition to the hash table causes extra overheads.

## NOTES

## NOTES

### Open addressing

Unlike the separate chaining method, no separate data structure is used in open addressing because all the key values are stored in the hash table itself. Since, each slot in the hash table contains the key value rather than the address value, a bigger hash table is required in this case as compared to separate chaining. Some value is used to indicate an empty slot. For example, if it is known that all the keys are positive values, then -1 can be used to represent a free or empty slot.

To insert a key value, first the slot in the hash table to which the key value hashes, is determined, using any hash function. If the slot is free, the key value is inserted into that slot. In case the slot is already occupied, then the subsequent slots, starting from the occupied slot, are examined systematically in the forward direction, until an empty slot is found. If no empty slot is found, then an overflow condition occurs.

In case of searching, first the slot in the hash table to which the key value hashes is determined, using any hash function. Then, the key value stored in that slot is compared with the key value to be searched. If they match, the search operation is successful; otherwise alternative slots are examined systematically in the forward direction to find the slot containing the desired key value. If no such slot is found, then the search is unsuccessful.

The process of examining the slots in the hash table to find the location of a key value is known as probing. The various types of probing are linear probing, quadratic probing, and double hashing that are used in open addressing method.

### Linear probing

Linear probing is the simplest approach for resolving collisions. It uses the following hash function:

$$h(k, i) = [h'(k) + i] \bmod N$$

where,

$h'(k)$  is any hash function (for simplicity we use  $k \bmod N$ )

$i$  is the probe number ranging from 0 to  $N-1$

To insert a key  $k$  in the hash table, first the slot  $T[h'(k)]$  is probed. If this slot is empty, the key is inserted into the slot. Otherwise, the slots  $T[h'(k) + 1]$ ,  $T[h'(k) + 2]$ ,  $T[h'(k) + 3]$ , and so on (up to  $T[N-1]$ ) are probed sequentially until an empty slot is found. If no empty slot is found up to  $T[N-1]$ , we wrap around to slots  $T[0]$ ,  $T[1]$ ,  $T[2]$ , and so on until an empty slot is found or we finally reach the slot  $T[h'(k) - 1]$ . The main advantage of linear probing is that as long as the hash table is not full, a free slot can always be found, however, the time taken to find an empty slot can be quite large.

To understand linear probing, consider the insertion of the following keys into the hash table with  $N=10$ .



126, 75, 37, 56, 29, 154, 10, 99

Further consider that the basic hash function is  $h'(k) = k \bmod N$ .

**Step 1:** The key value 126 hashes to the slot 6 as follows:

$$h(126, 0) = (126 \bmod 10 + 0) \bmod 10 = (6 + 0) \bmod 10 = 6 \bmod 10 = 6$$

Since slot 6 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
						126			

**Step 2:** Next, the key value 75 hashes to the slot 5 as follows:

$$h(75, 0) = (75 \bmod 10 + 0) \bmod 10 = (5 + 0) \bmod 10 = 5 \bmod 10 = 5$$

Since slot 5 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126			

**Step 3:** Next, the key value 37 hashes to the slot 7 as follows:

$$h(37, 0) = (37 \bmod 10 + 0) \bmod 10 = (7 + 0) \bmod 10 = 7 \bmod 10 = 7$$

Since slot 7 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37		

**Step 4:** Now, the key value 56 hashes to the slot 6 as follows:

$$h(56, 0) = (56 \bmod 10 + 0) \bmod 10 = (6 + 0) \bmod 10 = 6 \bmod 10 = 6$$

Since slot 6 is not empty, the next probe sequence is computed as follows:

$$h(56, 1) = (56 \bmod 10 + 1) \bmod 10 = (6 + 1) \bmod 10 = 7 \bmod 10 = 7$$

Slot 7 is also not empty, the next probe sequence is computed as follows:

$$h(56, 2) = (56 \bmod 10 + 2) \bmod 10 = (6 + 2) \bmod 10 = 8 \bmod 10 = 8$$

Since slot 8 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37	56	

**Step 5:** Next, the key value 29 hashes to the slot 9 as follows:

$$h(29, 0) = (29 \bmod 10 + 0) \bmod 10 = (9 + 0) \bmod 10 = 9 \bmod 10 = 9$$

Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37	56	29

## NOTES

## NOTES

**Step 6:** Now, the key value 154 hashes to the slot 4 as follows:

$$h(154, 0) = (154 \bmod 10 + 0) \bmod 10 = (4 + 0) \bmod 10 = 4 \bmod 10 = 4$$

Since slot 4 is empty, 154 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
				154	75	126	37	56	29

**Step 7:** Now, the key value 10 hashes to the slot 0 as follows:

$$h(10, 0) = (10 \bmod 10 + 0) \bmod 10 = (0 + 0) \bmod 10 = 0 \bmod 10 = 0$$

Since slot 0 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
10				154	75	126	37	56	29

**Step 8:** Now, the key value 99 hashes to the slot 9 as follows:

$$h(99, 0) = (99 \bmod 10 + 0) \bmod 10 = (9 + 0) \bmod 10 = 9 \bmod 10 = 9$$

Since slot 9 is not empty, the next probe sequence is computed as follows:

$$h(99, 1) = (99 \bmod 10 + 1) \bmod 10 = (9 + 1) \bmod 10 = 10 \bmod 10 = 0$$

Slot 0 is also not empty, the next probe sequence is computed as follows:

$$h(99, 2) = (99 \bmod 10 + 2) \bmod 10 = (9 + 2) \bmod 10 = 11 \bmod 10 = 1$$

Since slot 1 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
10	99			154	75	126	37	56	29

In case of searching also, the same process is followed. The only difference is that instead of finding an empty slot to store a given key value, you find the slot containing the desired key value. The number of probes required in both the cases (insertion and searching) is not more than the number of slots in the hash table.

Linear probing is easy to implement, but it has a disadvantage that if the hash table is relatively empty, then blocks (clusters) of occupied slots start forming. This problem is known as primary clustering in which many such blocks are separated by free slots. For example, in the previous example, the slots 0 and 1 form one cluster of occupied slots, slots 4 to 9 form another cluster of occupied slots. These two clusters are separated by free slots 2 and 3.

Once the clusters are formed, there are more chances that subsequent insertions will also end up in one of the cluster. This further increases the size of the cluster, thereby increasing the number of probes required to find a free slot. The performance gets worse as you insert more and more values in the table. To avoid this problem, two techniques, namely, quadratic probing and double hashing are used.

**Quadratic probing**

In quadratic probing, the collision function is quadratic instead of linear function of  $i$  as in linear probing. That is, it uses the following hash function:

$$h(k, i) = [h'(k) + i^2] \bmod N$$

where,

$h'(k)$  is any hash function (for simplicity we use  $k \bmod N$ )

$i$  is the probe number ranging from 0 to  $N-1$

To insert a key  $k$  in the hash table, first the slot  $T[h'(k)]$  is probed. If this slot is empty, the key is inserted into the slot. Otherwise, the slots  $h'(k) + i^2$  are probed. That is, the indexes  $h'(k) + 1, h'(k) + 4, h'(k) + 9$ , and so on are considered until an empty slot is found. Quadratic probing can also guarantee a successful insertion of a key as long as the hash table is at most half full, and the size of the table is a prime number. The same probe sequences are followed to search a desired key value in the hash table.

To understand quadratic probing, consider the insertion of the following keys into the hash table with  $N=11$ .

126, 75, 37, 56, 29, 154, 10, 99

Further consider that the basic hash function is  $h'(k) = k \bmod N$ .

**Step 1:** The key value 126 hashes to the slot 5 as follows:

$$h(126, 0) = (126 \bmod 11 + 0^2) \bmod 11 = (5 + 0) \bmod 11 = 5$$

Since slot 5 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
					126					

**Step 2:** Next, the key value 75 hashes to the slot 9 as follows:

$$h(75, 0) = (75 \bmod 11 + 0^2) \bmod 11 = (9 + 0) \bmod 11 = 9$$

Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
					126				75	

**Step 3:** Next, the key value 37 hashes to the slot 4 as follows:

$$h(37, 0) = (37 \bmod 11 + 0^2) \bmod 11 = (4 + 0) \bmod 11 = 4$$

Since slot 4 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
				37	126				75	

**Step 4:** Now, the key value 56 hashes to the slot 1 as follows:

$$h(56, 0) = (56 \bmod 11 + 0^2) \bmod 11 = (1 + 0) \bmod 11 = 1$$

**NOTES**

## NOTES

Since slot 1 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
	56			37	126				75	

**Step 5:** Next, the key value 29 hashes to the slot 7 as follows:

$$h(29, 0) = (29 \bmod 11 + 02) \bmod 11 = (7 + 0) \bmod 11 = 7$$

Since slot 7 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
	56			37	126		29		75	

**Step 6:** Now, the key value 154 hashes to the slot 0 as follows:

$$h(154, 0) = (154 \bmod 11 + 02) \bmod 11 = (0 + 0) \bmod 11 = 0$$

Since slot 0 is empty, 154 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56			37	126		29		75	

**Step 7:** Now, the key value 10 hashes to the slot 10 as follows:

$$h(10, 0) = (10 \bmod 11 + 02) \bmod 11 = (10 + 0) \bmod 11 = 10$$

Since slot 10 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56			37	126		29		75	10

**Step 8:** Now, the key value 99 hashes to the slot 0 as follows:

$$h(99, 0) = (99 \bmod 11 + 02) \bmod 11 = (0 + 0) \bmod 11 = 0$$

Since slot 0 is not empty, the next probe sequence is computed as follows:

$$h(99, 1) = (99 \bmod 11 + 12) \bmod 11 = (0 + 1) \bmod 11 = 1$$

Slot 1 is also not empty, the next probe sequence is computed as follows:

$$h(99, 2) = (99 \bmod 11 + 22) \bmod 11 = (0 + 4) \bmod 11 = 4$$

Slot 4 is also not empty, the next probe sequence is computed as follows:

$$h(99, 3) = (99 \bmod 11 + 32) \bmod 11 = (0 + 9) \bmod 11 = 9$$

Slot 9 is also not empty, the next probe sequence is computed as follows:

$$h(99, 4) = (99 \bmod 11 + 42) \bmod 11 = (0 + 16) \bmod 11 = 5$$

Slot 5 is also not empty, the next probe sequence is computed as follows:

$$h(99, 5) = (99 \bmod 11 + 52) \bmod 11 = (0 + 25) \bmod 11 = 3$$

Since slot 3 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56		99	37	126		29		75	10

Though quadratic probing eliminates primary clustering, it sometimes results in a milder form of clustering known as secondary clustering where the key values that initially hash to the same position will probe the same sequence of slots. For example, consider a key value 88 that is to be inserted into the hash table. Initially,

it hashes to slot 0 (as that of the key 99). Therefore, it will follow the same probe sequence, that is, 1, 4, 9, 5, 3.

### Double hashing

As you have seen that both the linear and quadratic probing add increments to the initial hash value  $h'(k)$  to define a probe sequence. Linear probing adds  $i$ , and quadratic probing adds  $i^2$  to the initial hash value to find an alternative slot. Both these increments are independent of the key  $k$ . The double hashing method, on the other hand, uses a different hash function  $h''(k)$  to compute these increments. Therefore, the increments are dependent on the key. Double hashing uses the following hash function:

$$h(k, i) = [h'(k) + i \cdot h''(k)] \bmod N$$

where,

$h'(k)$  is any hash function (for simplicity we use  $k \bmod N$ )

$h''(k)$  is another hash function (for simplicity we use  $k \bmod N'$  where  $N'$  is slightly less than  $N$  (say  $N-1$  or  $N-2$ ))

$i$  is the probe number ranging from 0 to  $N-1$

Initially, when a key  $k$  is to be inserted into the hash table, the first slot probed is  $T[h'(k)]$ . If this slot is empty, the key is inserted into the slot. Otherwise, alternative slots are searched using another independent hash function (hence the name double hashing). In case of searching also, the same process is followed until the desired key value is found, or all the key values in the table are examined.

To understand double hashing, consider the insertion of the following keys into the hash table with  $N=13$ .

126, 75, 37, 56, 29, 152, 35, 99

Further, consider that the basic hash function is  $h'(k) = k \bmod N$  and  $h''(k) = k \bmod (N-2)$ .

**Step 1:** The key value 126 hashes to the slot 9 as follows:

$$h(126, 0) = (126 \bmod 13 + 0 \cdot (126 \bmod 11)) \bmod 13 = (9 + 0) \bmod 13 = 9$$

Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126			

**Step 2:** Next, the key value 75 hashes to the slot 10 as follows:

$$h(75, 0) = (75 \bmod 13 + 0 \cdot (75 \bmod 11)) \bmod 13 = (10 + 0) \bmod 13 = 10$$

Since slot 10 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126	75		

### NOTES

## NOTES

**Step 3:** Next, the key value 37 hashes to the slot 11 as follows:

$$h(37, 0) = (37 \bmod 13 + 0 * (37 \bmod 11)) \bmod 13 = (11 + 0) \bmod 13 = 11$$

Since slot 11 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126	75	37	

**Step 4:** Now, the key value 56 hashes to the slot 4 as follows:

$$h(56, 0) = (56 \bmod 13 + 0 * (56 \bmod 11)) \bmod 13 = (4 + 0) \bmod 13 = 4$$

Since slot 4 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
				56					126	75	37	

**Step 5:** Next, the key value 29 hashes to the slot 3 as follows:

$$h(29, 0) = (29 \bmod 13 + 0 * (29 \bmod 11)) \bmod 13 = (3 + 0) \bmod 13 = 3$$

Since slot 3 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
			29	56					126	75	37	

**Step 6:** Now, the key value 152 hashes to the slot 9 as follows:

$$h(152, 0) = (152 \bmod 13 + 0 * (152 \bmod 11)) \bmod 13 = (9 + 0) \bmod 13 = 9$$

Since slot 9 is not empty, the next probe sequence is computed as follows:

$$h(152, 1) = (152 \bmod 13 + 1 * (152 \bmod 11)) \bmod 13 = (9 + 9) \bmod 13 = 5$$

Since slot 5 is empty, 152 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
			29	56	152				126	75	37	

**Step 7:** Now, the key value 35 hashes to the slot 9 as follows:

$$h(35, 0) = (35 \bmod 13 + 0 * (35 \bmod 11)) \bmod 13 = (9 + 0) \bmod 13 = 9$$

Since slot 9 is not empty, the next probe sequence is computed as follows:

$$h(35, 1) = (35 \bmod 13 + 1 * (35 \bmod 11)) \bmod 13 = (9 + 2) \bmod 13 = 11$$

Slot 11 is also not empty, the next probe sequence is computed as follows:

$$h(35, 2) = (35 \bmod 13 + 2 * (35 \bmod 11)) \bmod 13 = (9 + 4) \bmod 13 = 0$$

Since slot 0 is empty, 35 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
35			29	56	152				126	75	37	

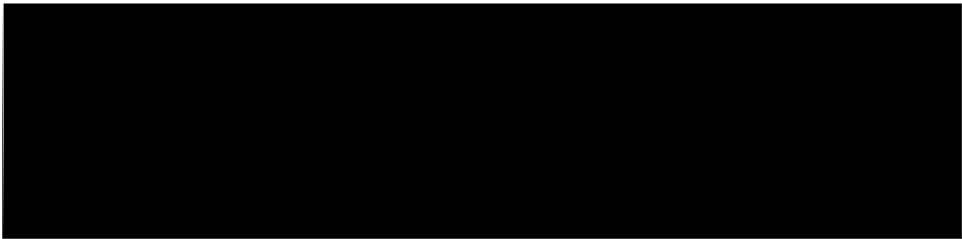
**Step 8:** Now, the key value 99 hashes to the slot as follows:

$$h(99, 0) = (99 \bmod 13 + 0 * (99 \bmod 11) \bmod 13 = (8 + 0) \bmod 13 = 8$$

Since slot 8 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
35			29	56	152			99	126	75	37	

Since the increment in double hashing depends on the value of key  $k$ , the values that hash to the same initial slot may have different probe sequences. Thus, double hashing almost eliminates the problem of primary and secondary clustering and its performance is very close to the ideal hashing. For example, the key value 35 initially hashes to slot 9 (as that of the key 152). However, the next probe sequence for 35 is 11 (not 5 as in case of 152).



## NOTES

### 11.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Hashing is also known as hash addressing.
2. A hash function  $h$  is simply a mathematical formula that maps the key to some slot in the hash table  $T$ .
3. The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys

### 11.5 SUMMARY

- Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree.
- The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.
- Here are two cases of insertion in a tree—first, insertion into an empty tree and second, insertion into a nonempty tree.

## NOTES

- The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.
- Deletion of a node from a binary search tree involves two steps—first, searching the desired node and second, deleting the node.
- If the node to be deleted has two child nodes, it is deleted by replacing its value by largest value in the left sub tree (in-order predecessor) or by smallest value in the right sub tree (in-order successor).
- Hashing (also known as hash addressing) is generally applied to a file  $F$  containing  $R$  records.
- Whenever a key is to be inserted in the hash table, a hash function is applied on it, which yields an index for the key.
- Since, the keys are inserted by applying hash functions on them, searching a key in the hash table is straightforward.
- A hash function  $h$  is simply a mathematical formula that maps the key to some slot in the hash table  $T$ .
- There are a number of hash functions available, however, the one which is easy to compute and ensures that two distinct values hash to different location in the hash table is desirable.
- The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys, because the number of key values is much larger than the number of available locations in the hash table.
- There are several ways for resolving collisions, the two most common techniques used are separate chaining and open addressing.
- In this technique, a linked list of all the key values that hash to the same hash value is maintained.
- The main disadvantage of separate chaining is that it makes use of pointers, which slows down the algorithm a bit because of the time required in allocating and deallocating the memory.
- Unlike the separate chaining method, no separate data structure is used in open addressing because all the key values are stored in the hash table itself.

---

## 11.6 KEY WORDS

---

- **Hash table:** It is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.



- **Division-remainder method:** It is the simplest and most commonly used method. In this method, the key  $k$  is divided by the number of slots  $N$  in the hash table, and the remainder obtained after division is used as an index in the hash table.

## NOTES

---

### 11.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. What do you mean by division remainder method?
2. List a few hashing techniques.
3. Draw a diagram of deletion of a node having two child nodes.
4. Write a short note about insertion and deletion operations.

#### Long-Answer Questions

1. “Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree.” Explain in detail.
2. What are the various cases of insertion in a binary search tree? Explain.
3. What do you mean by deleting a node?
4. Explain the Division-remainder method in detail.

---

### 11.8 FURTHER READINGS

---

- Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.
- Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.
- Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.
- Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.
- McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

---

### 11.9 LEARNING OUTCOMES

---

- The operations in binary tree
- Understand the insertion and deletion operations
- The hashing techniques
- The division remained method

## NOTES

---

## BLOCK - IV

### SEARCHING TECHNIQUES

---



---

## UNIT 12 SEARCHING

---

### Structure

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Searching
  - 12.2.1 Linear Search
  - 12.2.2 Binary Search
- 12.3 Answers to Check Your Progress Questions
- 12.4 Summary
- 12.5 Key Words
- 12.6 Self Assessment Questions and Exercises
- 12.7 Further Readings
- 12.8 Learning Outcomes

---

### 12.0 INTRODUCTION

---

In this unit, you will learn about various types of searching techniques. Searching is the process of finding a given value position in a list of provided values. Searching helps to decide whether a search key is present in the data or not. It can be defined as the algorithmic process of finding a particular item in a collection of items. Searching can be done on both internal data structure and on external data structures.

---

### 12.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Learn about searching techniques
- Understand about linear search
- Explain binary search
- Discuss about interpolation search

---

### 12.2 SEARCHING

---

While solving a problem, a programmer may need to search a value in an array. The process of finding the occurrence of a particular data item in a list is known as searching. Search is said to be successful or unsuccessful depending on whether the data item is found or not. The various search techniques are linear binary, Fibonacci and interpolation searches.

### 12.2.1 Linear Search

Linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached. While traversing, each element of the array is compared with the value to be searched, and if the value is found the search is said to be successful. This technique is suitable for performing a search in a small array or in an unsorted array.

#### NOTES

#### Algorithm 12.1 Linear Search

```
linear_search (ARR, size, item)
1. Set i = 0
2. While i < size
    If ARR [i] = item      //item is the value to be searched
        Return i and go to step 4
    End If
    Set i = i + 1
End While
3. Return -1              //search unsuccessful
4. End
```

#### Example 12.1: A program to perform linear search

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
int linear_search(int ARR[] , int size , int item );

void main()
{
    int ARR[MAX];
    int item, size, pos, i;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=linear_search(ARR, size, item);
    if (pos==-1)
```

**NOTES**

```

        printf("\nElement not found");
    else
        printf("\nElement found at location: %d", pos+1);
    getch();
}

int linear_search(int ARR[], int size, int item)
{
    int i;
    for (i=0;i<size;i++)
    {
        if(ARR[i]==item)
            return (i);
    }
    return (-1);
}

```

**The output of the program is**

```
Enter the size of the array (max 20): 5
```

```
Enter elements of the array:
```

```
1
```

```
4
```

```
3
```

```
6
```

```
7
```

```
Enter the element to be searched: 4
```

```
Element found at location: 2
```

**12.2.2 Binary Search**

The binary search technique is used to search for a particular element in a sorted (in ascending or descending order) array. In this technique, the element to be searched (say, `item`) is compared with the middle element of the array. If `item` is equal to the middle element, then the search is successful. If `item` is smaller than the middle element, `item` is searched in the segment of the array before the middle element. However, if `item` is greater than the middle element, `item` is searched in the array segment after the middle element. This process is repeated until the element is found or the array segment is reduced to a single element that is not equal to `item`.

At every stage of the binary search technique, the array is reduced to a smaller segment. It searches a particular element in the lowest possible number of

comparisons. Hence, the binary search technique is used for larger and sorted arrays, as it is faster compared to linear search. For example, consider an array ARR shown here:

1	2	3	4	5	6	7
0	1	2	3	4	5	6

To search an item (say, 7) using binary search in the array ARR with  $size=7$ , the following steps are performed.

- Initially, set  $LOW=0$  and  $HIGH=size-1$ . The middle of the array is determined using the formula  $MID = (LOW + HIGH) / 2$ , that is,  $MID = (0 + 6) / 2$ , which is equal to 3. Thus,  $ARR[MID] = 4$ .

LOW			MID			HIGH
↓			↓			↓
1	2	3	4	5	6	7
0	1	2	3	4	5	6

- Since the value stored at  $ARR[3]$  is less than the value to be searched, that is 7, the search process is now restricted from  $ARR[4]$  to  $ARR[6]$ . Now  $LOW$  is 4 and  $HIGH$  is 6. The middle element of this segment of the array is calculated as  $MID = (4 + 6) / 2$ , that is, 5. Thus,  $ARR[MID] = 6$ .

LOW	MID	HIGH
↓	↓	↓
5	6	7
4	5	6

- The value stored at  $ARR[5]$  is less than the value to be searched, hence the search process begins from the subscript 6. As  $ARR[6]$  is the last element, the item to be searched is compared with this value. Since  $ARR[6]$  is the value to be searched, the search is successful.

#### Algorithm 12.2: Binary Search

```

binary_search(ARR, size, item)
1. Set LOW = 0
2. Set HIGH = size - 1
3. While LOW <= HIGH
    Set MID = (LOW + HIGH) / 2
    If ITEM = ARR[MID]
        Return MID and go to step 5
    Else If item < ARR[MID]
        Set HIGH = MID - 1
    Else
        Set LOW = MID + 1
    End If
End While
4. Return -1
5. End
  
```

## NOTES

**NOTES****Example 12.2:** A program to perform binary search

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
int binary_search(int ARR[], int size, int item);

void main()
{
    int ARR[MAX];
    int item, size, pos, i;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=binary_search(ARR, size, item);
    if (pos==-1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d", pos+1);
    getch();
}

int binary_search(int ARR[], int size, int item)
{
    int LOW=0;
    int HIGH=size-1;
    int MID;
    while(LOW<=HIGH)
    {
        MID=(HIGH+LOW)/2;
        if(ARR[MID]==item)
            return MID;
    }
}

```

```

        else
            if (item < ARR[MID])
                HIGH = MID - 1;
            else
                LOW = MID + 1;
        }
    }
    return (-1);
}

```

### The output of the program is

Enter the size of the array (max 20): 5

Enter elements in sorted order:

11

22

33

44

55

Enter the element to be searched: 33

Element found at location: 3

### Fibonacci Search

Like binary search, Fibonacci search is also applied on sorted arrays. However, unlike, binary search it does not compare the element to be searched (say, `item`) with the middle element of the list. Instead, it uses Fibonacci series to determine the index, say `pos`, where to look in the array.

The Fibonacci series up to  $n$  terms (generally denoted by  $F_0, F_1, F_2, \dots, F_n$ ), is generated as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21...

That is, each successive term is the sum of its two preceding terms. That is,

If  $n=0$  or  $n=1$

Then

$F_n = n$ ,

Else

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

Initially, the search takes a smallest number, say  $F_m$ , of Fibonacci series that is greater than or equal to the `size`, where `size` is the number of elements in the array. Then, it compares the `item` with the element at  $F_m - 1$  position in the array. The result is as follows:

- If they are equal, search is successful, element found at position  $F_m - 1 + 1$ .

## NOTES

NOTES

- If the `item` is smaller, it is searched in the sub-list left to  $F_m-1$ .
- If the `item` is greater, it is searched in the sub-list right to  $F_m-1$ .

If `item` is not found, again a smallest number of Fibonacci series that is greater than or equal to the size of the sub-list (left or right) to be searched is taken, and the whole process is repeated until the desired element is found or the sub-list is reduced to a single element that is not equal to `item`.

To understand the Fibonacci search, consider the array `ARR` shown here. Suppose you need to search the element 13.

1	7	13	19	25	31	36
0	1	2	3	4	5	6

Since the size of the array is 7, the initial value of  $F_m$  will be 8. The search first compares the element 13 with element at index 5 ( $F_m-1$ ), that is, 26. Since 13 is less than 26, the sub-list left to index 5 is considered. The size of this sub-list is 5, therefore, new  $F_m$  will be 5. Now, the element 13 is compared with the element at index 3 ( $F_m-1$ ), that is, 13. Since, it is the desired element, the search is successful, and the element is found at position 4.

Algorithm 12.3 Fibonacci Search

```
fibonacci_search(ARR, size, item)

1. Set flag = 0, first = 0, pos = -1, last = size
2. While (flag != 1 AND first <= last)
    Set index = retfib(size)           //calling retfib() function
    If item = ARR[index+first]
        Set flag = 1
        Set pos = index
        Break                         //jump out of the loop
    Else if item > ARR[index+first]
        Set first = index + 1
        Set size = last - first
    Else
        Set last = index - 1
        Set size = last - first + 1
    End If
End While
3. If flag = 1
    Return (pos + first + 1) and go to step 4Else
    Return -1
End If
4. End

retfib(n)                             //function to generate a Fibonacci number

1. Set a = 1, b = 1, c = 1
2. If (n = 0 OR n = 1)
    Return 0 and go to step 3
Else
    While c < n
        Set c = a + b
        Set a = b
        Set b = c
    End While
    Return a
End If
3. End
```



**Example 12.3:** A program to implement Fibonacci search*Searching*

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototypes*/
int fibonacci_search(int [], int, int);
int retfib(int n);

void main()
{
    int ARR[MAX], size, item, pos, i;
do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
        }while(size>MAX);
printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=fibonacci_search(ARR, size, item);
    if (pos== -1)
        printf("\nElement not found");
    else
        printf("\nElement found at location: %d ", pos);
    getch();
}

int retfib(int n)
{
    int a=1, b=1, c=1, i;
    if(n==0 || n==1)
        return 0;
    else
    {
        while(c<n)
```

**NOTES**

Searching

## NOTES

```
{
    c=a+b;
    a=b;
    b=c;
}
return a;
}
}

int fibonacci_search(int ARR[], int size, int item)
{
    int flag=0, first=0, index, pos=-1, last=size;
    while(flag!=1 && first<=last)
    {
        index=retfib(size);
        if(item==ARR[index+first])
        {
            flag=1;
            pos=index;
            break;
        }
        else if(item>ARR[index+first])
        {
            first=index+1;
            size=last-first;
        }
        else
        {
            last=index-1;
            size=last-first+1;
        }
    }
    if(flag==1)
        return (pos+first+1);
    else
        return -1;
}
```

### The output of the program is

Enter the size of the array (max 20): 5

Enter elements in sorted order:

45  
67  
89  
100  
120

Enter the element to be searched: 67

Element found at location: 2

## NOTES

### Interpolation Search

Interpolation search is similar to binary search in the sense that it also applies on sorted arrays. However, it is based on the assumption that the elements in the list are uniformly distributed. This is done in a manner, that the probability of an element being in a particular range equals its probability of being in any other range of the same length. Thus, instead of determining the middle element using the formula  $(\text{low} + \text{high}) / 2$ , interpolation search determines the location of the item to be searched. This is done according to the magnitude of the item relative to the first and last elements of the array.

For example, if an array contains 10 elements ranging from 1 to 100 which are uniformly distributed, then according to the interpolation search, the element 50 lies in between the list.

Therefore, it uses the following formula for calculating the mid:

$$\text{mid} = \text{low} + (\text{high} - \text{low}) * ((\text{item} - \text{ARR}[\text{low}]) / (\text{ARR}[\text{high}] - \text{ARR}[\text{low}]))$$

To understand the interpolation search, consider the array ARR. Suppose you have to search the element 13. Note that the elements in this array are uniformly distributed.

1	7	13	19	25	31	36
0	1	2	3	4	5	6

Initially,  $\text{low} = 0$ ,  $\text{high} = 6$ ,  $\text{ARR}[\text{low}] = \text{ARR}[0] = 1$ , and  $\text{ARR}[\text{high}] = \text{ARR}[6] = 37$ , therefore the value of mid can be calculated as follows:

$$\begin{aligned} \text{mid} &= 0 + (6 - 0) * ((13 - 1) / (37 - 1)) \\ &= 6 * (12 / 36) \\ &= 6 * 0.33 \\ &= 2 \end{aligned}$$

Since  $\text{ARR}[\text{mid}] = \text{ARR}[2] = 13$ , which is the desired element, the search terminates successfully.

**NOTES****Algorithm 12.4 Interpolation Search**

```

interpol_search(ARR, size, item)

1. Set low = 0, high = size - 1, flag = -1
2. While low <= high
    If high = low
        If ARR[low] = item
            Set flag = low
        End If
        Break          //jump out of the loop
    End If
    Set mid = low + (high - low) * ((item - ARR[low]) / (ARR[high] - ARR[low]))
    ARR[mid] = item
    Set flag = midBreak
    Else if ARR[mid] > item
        Set high = mid - 1
    Else
        Set low = mid + 1
    End If
3. Return (flag)
4. End

```

**Example 12.4:** A program to implement interpolation search

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

int interpol_search(int [], int, int);

void main()
{
    int ARR[MAX];
    int item, size, pos, i;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter elements in sorted order:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    printf("\nEnter the element to be searched: ");
    scanf("%d", &item);
    pos=interpol_search(ARR, size, item);
    if (pos== -1)
        printf("\nElement not found");
}

```

```

    else
        printf("\nElement found at location: %d", pos+1);
        getch();
}

int interpol_search(int ARR[], int size, int item)
{
    int mid, low, high, flag=-1;
    low=0;
    high=size-1;
    while(low<=high)
    {
        if(high==low)
        {
            if(ARR[low]==item)
                flag=low;
            break;
        }
        mid=low+(high-low)*(float)(item-ARR[low]) /
(ARR[high]-ARR[low]);
        if(ARR[mid]==item)
        {
            flag=mid;
            break;
        }
        else if(ARR[mid]>item)
            high=mid-1;
        else
            low=mid+1;
    }
    return (flag);
}

```

## NOTES

### The output of the program is

Enter the size of the array (max 20): 8

Enter elements in sorted order:

20

40

65

Searching

## NOTES

80  
100  
120  
180  
200

Enter the element to be searched: 180

Element found at location: 7

### Comparison of Different Search Algorithms

This section gives the analysis of different search algorithms and determines their time complexities.

#### Analysis of Linear Search

In the best case, when the item is found at first position, the search operation terminates successfully with only one comparison. Thus, in this case the complexity of the algorithm is  $O(1)$ . In the worst case, when the item to be searched appears at the end of the list or is not present in the list, linear search requires  $n$  comparisons. In both the cases, the average complexity of linear search is  $O(n)$ .

#### Analysis of Binary Search

In each iteration, binary search algorithm reduces the array to one half. Therefore, for an array containing  $n$  elements, there will be  $\log_2 n$  iterations. Thus, the complexity of binary search algorithm is  $O(\log_2 n)$ . This complexity will be same irrespective of the position of the element, even if the element is not present in the list.

#### Analysis of Fibonacci Search

The complexity of Fibonacci search is similar to that of binary search, that is,  $O(\log_2 n)$ . However in general, the performance of Fibonacci search is always worse than binary search. Finding the middle element in binary search involves division operation  $[(low+high)/2]$ , but in Fibonacci search only addition or subtraction operation is involved. Therefore, the average performance of Fibonacci search may be better than binary search on computers where division is more time consuming than addition or subtraction.

#### Analysis of Interpolation Search

The performance of interpolation search is highly dependent on the distribution of elements in the list. In the best case when the elements are uniformly distributed, the interpolation search requires  $\log_2 n (\log_2 n)$  comparisons, as compared to binary search which requires  $\log_2 n$  comparisons. Thus, in this case

interpolation search has a very poor performance. On the other hand, if the elements are not uniformly distributed, then interpolation search gives a very poor performance. In the worst case, the value of `mid` can consistently be equal to `low+1` or `high-1`. In this case, the performance of interpolation search deteriorates to linear search. However, the performance of binary search can never exceed  $\log_2 n$ .

## NOTES

### 12.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The process of finding the occurrence of a particular data item in a list is known as searching.
2. Interpolation search is similar to binary search in the sense that it also applies on sorted arrays.
3. If the elements are not uniformly distributed, then interpolation search gives a very poor performance.

### 12.4 SUMMARY

- The process of finding the occurrence of a particular data item in a list is known as searching.
- The various search techniques are linear binary, Fibonacci and interpolation searches.
- Linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached.
- The binary search technique is used to search for a particular element in a sorted (in ascending or descending order) array.
- At every stage of the binary search technique, the array is reduced to a smaller segment.
- Interpolation search is similar to binary search in the sense that it also applies on sorted arrays.

**NOTES**

- In the best case, when the item is found at first position, the search operation terminates successfully with only one comparison.
- In each iteration, binary search algorithm reduces the array to one half. Therefore, for an array containing  $n$  elements, there will be  $\log_2 n$  iterations.
- The performance of interpolation search is highly dependent on the distribution of elements in the list.
- If the elements are not uniformly distributed, then interpolation search gives a very poor performance.

---

## 12.5 KEY WORDS

---

- **Linear search:** It is one of the simplest searching technique, where the array is traversed sequentially from the first element until the value is found or the end of the array is reached.
- **Binary search:** It is a search technique which is used to search for a particular element in a sorted (in ascending or descending order) of array.

---

## 12.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

**Short-Answer Questions**

1. Write a short note on searching.
2. What do you mean by linear search?
3. Write a note on binary search.
4. Write an algorithm for binary search.

**Long-Answer Questions**

1. Write a program to perform linear search.
2. Write a program to perform binary search.
3. “The binary search process is repeated until the element is found or the array segment is reduced to a single element that is not equal to item.” Discuss.
4. Give a detailed comparison of different search algorithms.

---

## 12.7 FURTHER READINGS

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.



Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

Searching

## NOTES

---

## 12.8 LEARNING OUTCOMES

---

- Learn about searching techniques
- Understand about linear search
- Binary search
- Learn about interpolation search

## NOTES

---

## BLOCK - V

### SORTING TECHNIQUES

---



---

## UNIT 13 SORTING

---

### Structure

- 13.0 Introduction
  - 13.1 Objectives
  - 13.2 Definition
  - 13.3 Bubble Sort
  - 13.4 Insertion Sort
  - 13.5 Radix Sort
  - 13.6 Answers to Check Your Progress Questions
  - 13.7 Summary
  - 13.8 Key Words
  - 13.9 Self Assessment Questions and Exercises
  - 13.10 Further Readings
  - 13.11 Learning Outcomes
- 

### 13.0 INTRODUCTION

---

In the previous unit, you learnt about searching. This unit will discuss sorting. Sorting refers to the way in which data is arranged in a particular order. A sorting algorithm is employed to rearrange a given array elements according to a comparison operator on the elements. This unit will first begin with the definition of sorting. It will then discuss Bubble Sort, Insertion Sort and Radix Sort.

---

### 13.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Discuss sorting
  - Describe the process of insertion sorting, bubble sorting and bucket sorting
- 

### 13.2 DEFINITION

---

The process of arranging the data in some logical order is known as sorting. The order can be ascending or descending for numeric data, and alphabetically for character data. There are two types of sorting, namely, internal sorting and external sorting. If all the data that is to be sorted fits entirely in the main memory, then internal (in-memory) sorting is used.

On the other hand, if all the data that is to be sorted do not fit entirely in the main memory, external sorting is required. An external sorting requires the use of

external memory such as disks or tapes during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any internal sorting technique and written back to the disk in some intermediate file. This process continues until all the data is sorted.

### Internal Sorting

There are different internal sorting algorithms such as insertion sort, bubble sort, selection sort, heap sort, merge sort, quick sort and bucket sort. The choice of a particular algorithm depends on the properties of the data and the operations to be performed on the data. For all these algorithms, we will consider an array `ARR` containing  $n$  elements, which are to be sorted in an ascending order.

## NOTES

### 13.3 BUBBLE SORT

The bubble sort algorithm requires  $n-1$  passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if one element is greater than the other then both the elements are swapped. After the first pass, the largest element in the list is placed at the last position. Similarly, in the second pass the second largest element is placed at its appropriate position. Thus, in each subsequent pass, the next largest element is placed at its appropriate position. Since this algorithm makes the larger values to 'bubble up' to the end of the list, it is named bubble sort.

The bubble sort algorithm possesses an important property. This property is that if a particular pass is made through the list without swapping any items, then there will be no further swapping of elements in the subsequent passes. This property can be used to eliminate the unnecessary passes once the list is sorted in the desired order. For this, a `flag` variable can be used to detect if any interchange has been made during the pass. We use `flag = 0` to indicate that no swaps have occurred in a particular pass, therefore, no further passes are required.

To understand the bubble sort technique, consider an unsorted array shown here.

8	7	65	5	43
---	---	----	---	----

The steps to sort the values stored in the array in ascending order using bubble sort are as follows:

#### First pass:

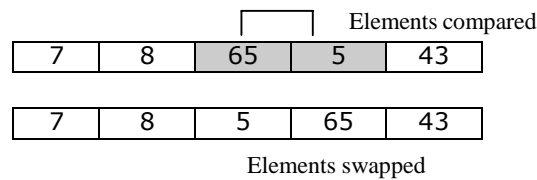
1. The values 8 and 7 are compared with each other. Since 7 is smaller than 8, both the values are swapped with each other.
2. **No swapping:** Next, the values 8 and 65 are compared with each other. Since 8 is less than 65, means they are in proper order and, hence, no swapping is required. The list remains unchanged.

7	8	65	5	43
---	---	----	---	----

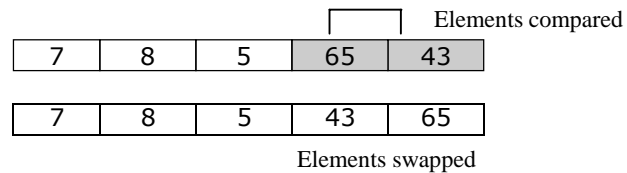
No swapping

## NOTES

3. **Elements compared:** Then the values 65 and 5 are compared with each other. Since 5 is less than 65, both the values are swapped.



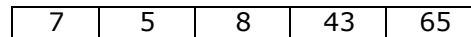
4. **Elements compared:** Next, the values 65 and 43 are compared with each other. Since 43 is less than 65, both the values are swapped.



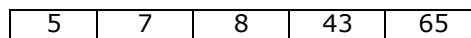
After the first pass, the largest value of the array (here, 65) is placed at last position.

**Second pass**

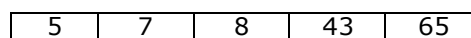
1. The values 7 and 8 are compared with each other. Since 7 is smaller than 8, no swapping is required.
2. Then the values 8 and 5 are compared. Since 8 is greater than 5, both are swapped.
3. Next, the elements 8 and 43, and 43 and 65 are compared. Since they are already in ascending order, they need not be swapped.

**Third Pass**

1. The values 7 and 5 are compared with each other. Since 7 is greater than 5, both are swapped.
2. Since the remaining elements are already in ascending order, they are not swapped.

**Fourth Pass**

1. In the fourth pass, no swapping is required as all the elements are already in ascending order. Thus, at the end of this pass, the list is sorted in ascending order as follows:



**Algorithm 13.1 Bubble Sort**

```

bubble_sort(ARR, size)
1. Set i = 0, flag = 1
2. While (i < size-1 AND flag = 1)
    Set j = 0
    Set flag = 0
    While (j < size-i-1)
        If (ARR[j] > ARR[j+1])
            Set flag = 1           //swap will occur, hence set flag = 1
            Set temp = ARR[j]      //temp is temporary variable used to swap
                                   //two values
            Set ARR[j] = ARR[j+1]
            Set ARR[j+1] = temp
        End If
        Set j = j + 1
    End While
    Print ARR after (i+1)th pass
    Set i = i + 1
End While
3. Print "No. of passes: ", i
4. End

```

**NOTES****Example 13.1:** A program to show sorting of an array using bubble sort

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void bubble_sort(int [], int);

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    bubble_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
    getch();
}

```

**NOTES**

```

    }

void bubble_sort(int ARR[], int size)
{
    int i, j, k, temp, flag=1;
    i=0;
    while (i<size-1 && flag==1)
    {
        flag=0;
        for(j=0;j<size-i-1; j++)
        {
            if (ARR[j]>ARR[j+1])
            {
                flag=1;
                temp=ARR[j];
                ARR[j]=ARR[j+1];
                ARR[j+1]=temp;
            }
        }
        printf("\nArray after pass %d: ", i+1);
        for(k=0;k<size;k++)
            printf("%d ", ARR[k]);
        i++;
    }
    printf("\nNo. of passes: %d", i);
}

```

**The output of the program is**

```

Enter the size of the array (max 20): 5
Enter the elements of the array:
8
7
65
5
43

Array after pass 1: 7  8  5  43  65
Array after pass 2: 7  5  8  43  65
Array after pass 3: 5  7  8  43  65
Array after pass 4: 5  7  8  43  65
No. of passes: 4
The sorted array is: 5  7  8  43  65

```

## 13.4 INSERTION SORT

The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list. In the first pass, the element  $ARR[1]$  is compared with  $ARR[0]$ , and if  $ARR[1]$  and  $ARR[0]$  are not sorted, they are swapped. In the second pass, the element  $ARR[2]$  is compared with  $ARR[0]$  and  $ARR[1]$ , and it is inserted at its proper position in the sorted sub-list containing the elements  $ARR[0]$ ,  $ARR[1]$ . Similarly, during  $i$ th iteration, the element  $ARR[i]$  is placed at its proper position in the sorted sub-list containing the elements  $ARR[0]$ ,  $ARR[1]$ ,  $ARR[2]$ , ...,  $ARR[i-1]$ .

In order to determine the actual position of the element (say,  $ARR[i]$ ) in the sorted sub-list containing the elements  $ARR[0]$ ,  $ARR[1]$ , ...,  $ARR[i-1]$ , the element  $ARR[i]$  is compared with all other elements to its left, until an element  $ARR[j]$  is found such that  $ARR[j] \leq ARR[i]$ . Now, to insert the element at its actual position, all the elements  $ARR[i-1]$ ,  $ARR[i-2]$ ,  $ARR[i-3]$ , ...,  $ARR[j+1]$  are shifted one position towards the right to create the space for  $ARR[i]$ , and then  $ARR[i]$  is inserted at  $(j+1)$ st position.

To understand the insertion sort algorithm, consider an unsorted array shown here. The steps to sort the values stored in the array in ascending order using insertion sort are given here.

7	33	20	11	6
---	----	----	----	---

- The first value, that is, 7 is trivially sorted by itself.
- Then the second value 33 is compared with the first value 7. Since 33 is greater than 7, no changes are made.
- Next, the third element 20 is compared with its previous elements (elements towards its left). Since 20 is smaller than 33 but greater than 7, it is inserted at second position. For this, the element 33 is shifted one position towards right and 20 is inserted at its appropriate (second) position.

7	33	20	11	6
7	20	33	11	6

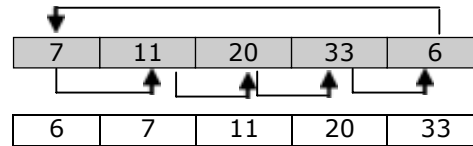
- Then, the fourth element 11 is compared with its previous elements. Since 11 is greater than 7 and less than 20 and 33, it is placed between 7 and 20. For this, the elements 20 and 33 need to be shifted one position towards the right.

7	20	33	11	6
7	11	20	33	6

### NOTES

## NOTES

- Finally, the last element 6 is compared with all the elements preceding it. Since it is smaller than all the other elements, they are shifted one position towards right and 6 is inserted at the first position in the array. After this pass, the array is sorted.



The final sorted array is as follows:

6	7	11	20	33
---	---	----	----	----

**Algorithm 13.2 Insertion Sort**

```
insertion_sort(ARR, size)
```

```

1. Set i = 1
2. While (i < size)
    Set temp = ARR[i]
    j = i - 1
    While (temp < ARR[j] AND j >= 0)
        Set ARR[j+1] = ARR[j]
        Set j = j - 1
    End While
    Set ARR[j+1] = temp
    Print ARR after ith pass
    i = i + 1
End While
3. Print "No. of passes: ", i-1
4. End
```

**Example 13.2:** A program to show sorting of an array using insertion sort

```

#include<stdio.h>
#include<conio.h>
#define MAX 20
/*Function prototype*/
void insertion_sort(int [], int);
void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0; i<size; i++)
        scanf("%d", &ARR[i]);
```



```

        insertion_sort(ARR, size);
        printf("\nThe sorted array is: ");
        for(i=0;i<size;i++)
            printf("%d ", ARR[i]);
        getch();
    }
    void insertion_sort(int ARR[], int size)
    {
        int i, j, k, temp, count=0;;
        for (i=1;i<size;i++)
        {
            temp=ARR[i];
            j=i-1;
            if(temp<ARR[j])
            {
                while(temp<ARR[j] && j>=0)
                {
                    ARR[j+1]=ARR[j];
                    j--;
                }
            }
            ARR[j+1]=temp;
            printf("\nArray after pass %d: ", i);
            for(k=0;k<size;k++)
                printf("%d ", ARR[k]);
        }
        printf("\nNo. of passes: %d", i-1);
    }
}

```

## NOTES

### The output of the program is

```

Enter the size of the array (max 20): 5
Enter the elements of the array:
35
20
4
10
5
Array after pass 1: 20  35  4  10  5
Array after pass 2: 4  20  35  10  5
Array after pass 3: 4  10  20  35  5
Array after pass 4: 4  5  10  20  35
No. of passes: 4
The sorted array is: 4  5  10  20  35

```

**NOTES****13.5 RADIX SORT**

The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left. In the first pass, the numbers are sorted according to the digits at units place. In the second pass, the numbers are sorted according to the digits at tens place, and so on. Since the base of decimal numbers is 10, the radix sort requires ten buckets, numbered 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 to store the array elements in each pass. The number of passes in the algorithm is equal to the number of digits in the largest number. Therefore, the algorithm first determines the largest number in the list and counts the number of digits in it.

In the first pass, the numbers having 0 at units place are placed in bucket 0. Numbers having 1 at their units place are placed in bucket 1, numbers having 2 at their units place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at units place. This list becomes the input for the second pass. In the second pass, the numbers having 0 at their tens place are placed in bucket 0. Numbers having 1 at their tens place are placed in bucket 1, numbers having 2 at their tens place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at tens place. Now, this array becomes the input for the third pass. This process is repeated  $d$  times, where  $d$  represents the number of digits in the largest number of the list.

To understand the radix sort algorithm, consider an unsorted array.

318	233	56	899	912	674	555	110	21	746
-----	-----	----	-----	-----	-----	-----	-----	----	-----

Since the largest element (that is, 899) consists of three digits, the array will be sorted in three passes.

**First pass**

In the first pass, the digits at the units place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
110	21	912	233	674	555	56 746		318	899

Now, the elements are retrieved from each bucket and copied into the original array. The array now becomes:

110	21	912	233	674	555	56	746	318	899
-----	----	-----	-----	-----	-----	----	-----	-----	-----

**Second pass**

In the second pass, the digits at the tens place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
	110 912 318	21	233	746	555 56		674		899

Now, the elements are retrieved from each bucket and copied into the original array. The array now becomes:

110	912	318	21	233	746	555	56	674	899
-----	-----	-----	----	-----	-----	-----	----	-----	-----

### Third pass

In the third pass, the digits at the hundredth place are considered, and the elements are placed in the corresponding buckets as follows:

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
021 056	110	233	318		555	674	746	899	912

Now, the elements are retrieved from each bucket and copied into the original array. After this pass, the array is sorted as follows:

21	56	110	233	318	555	674	746	899	912
----	----	-----	-----	-----	-----	-----	-----	-----	-----

#### Algorithm 13.3 Radix Sort

```

bucket_sort(ARR, size)
1. Set i = 1, digitcount = 0, divisor = 1, t = 0, largest = ARR[0]
2. While (i < size)
    If (ARR[i] > largest)
        Set largest = ARR[i]
    End If
    Set i = i + 1
End While
3. While (largest > 0)
    Set largest = largest / 10
    Set digitcount = digitcount + 1
End While
4. Set i = 0
5. While (i < digitcount)
    Set k = 0
    While (k < 10)
        Set buckcount[k] = 0
        Set k = k + 1
    End While
    Set j = 0
    While (j < size)
        Set r = (ARR[j] / divisor) % 10
        Set bucket[buckcount[r]][r] = ARR[j]
        Set buckcount[r] = buckcount[r] + 1
        Set j = j + 1
    End While
    Set t = 0, j = 0
    While (j < 10)
        Set k = 0
        While (k < buckcount[j])
            Set ARR[t] = bucket[k][j]
            Set t = t + 1
            Set k = k + 1
        End While
        Set j = j + 1
    End While
    Print ARR after ith pass
    Set divisor = divisor * 10
    Set i = i + 1
End While
6. Print "No. of passes: ", digitcount
7. End

```

## NOTES

**NOTES****Example 13.3:** A program to show sorting of an array using radix sort

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void bucket_sort(int [], int);

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    bucket_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
    getch();
}

void bucket_sort(int ARR[], int size)
{
    int bucket[MAX][10], buckcount[10];
    int i, j, k, r, digitcount=0, divisor=1, largest,
t=0;
    largest=ARR[0];
    for(i=1;i<size;i++)
    {
        if(ARR[i]>largest)
            largest=ARR[i];
    }
    while(largest>0)

```

```

{
    largest/=10;
    digitcount++;
}
for(i=0;i<digitcount;i++)
{
    for(k=0;k<10;k++)
        buckcount[k]=0;
    for(j=0;j<size;j++)
    {
        r=(ARR[j]/divisor)%10;
        bucket[buckcount[r]][r]=ARR[j];
    }
    t=0;
    for(j=0;j<10;j++)
        for(k=0;k<buckcount[j];k++)
        {
            ARR[t++]=bucket[k][j];
        }
    printf("\nArray after pass %d: ",i+1);
    for(j=0;j<size;j++)
        printf("%d ", ARR[j]);
    divisor*=10;
}
printf("\nNo. of passes: %d ", digitcount);
}

```

### The output of the program is

Enter the size of the array (max 20): 10

Enter the elements of the array:

318  
233  
56  
899  
912  
674  
555  
110  
21  
746

## NOTES

**NOTES**

Array after pass 1: 110 21 912 233 674 555 56 746 318 899

Array after pass 2: 110 912 318 21 233 746 555 56 674 899

Array after pass 3: 21 56 110 233 318 555 674 746 899 912

No. of passes: 3

The sorted array is: 21 56 110 233 318 555 674 746 899 912

---

### 13.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. The process of arranging the data in some logical order is known as sorting.
2. There are two types of sorting, namely, internal sorting and external sorting.
3. An important property of the bubble sort algorithm is that if a particular pass is made through the list without swapping any items, then there will be no further swapping of elements in the subsequent passes.
4. The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left.

---

### 13.7 SUMMARY

---

- The process of arranging the data in some logical order is known as sorting. The order can be ascending or descending for numeric data, and alphabetically for character data.
- There are two types of sorting, namely, internal sorting and external sorting
- If all the data that is to be sorted do not fit entirely in the main memory, external sorting is required
- The bubble sort algorithm requires  $n-1$  passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if one element is greater than the other then both the elements are swapped.
- The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list.

- The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left.

Sorting

### 13.8 KEY WORDS

- **Sorting:** It refers to arranging *data* in a particular format.
- **Bucket Sort:** It is a *sorting* algorithm that works by distributing the elements of an array into a number of *buckets*.
- **Insertion Sort:** It is a *sorting* algorithm in which the elements are transferred one at a time to the right position.

### NOTES

### 13.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

#### Short-Answer Questions

1. What are the two types of sorting?
2. What does external sorting require?
3. How does insertion sort work?

#### Long-Answer Questions

1. Describe the steps to sort the values stored in the array in ascending order using bubble sort.
2. Write a program showing sorting of an array using bubble sort and insertion sort.
3. Examine the steps to sort values using radix sort.

### 13.10 FURTHER READINGS

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. *Data Structures and Algorithms in Java*. London: John Wiley and Sons.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

### 13.11 LEARNING OUTCOMES

- Sorting
- The process of insertion sorting, bubble sorting and bucket sorting

---

## UNIT 14 OTHER SORTING TECHNIQUES

---

### NOTES

#### Structure

- 14.0 Introduction
- 14.1 Objectives
- 14.2 Selection Sort
- 14.3 Quick Sort
- 14.4 Tree Sort
- 14.5 Answers to Check Your Progress Questions
- 14.6 Summary
- 14.7 Key Words
- 14.8 Self Assessment Questions and Exercises
- 14.9 Further Readings
- 14.10 Learning Outcomes

---

### 14.0 INTRODUCTION

---

In the previous unit, you were introduced to sorting. You learnt about sorting techniques such as insertion sorting, bubble sorting and bucket sorting. In this unit, you will be introduced with other sorting techniques such as selection sort, quick sort and tree sort.

---

### 14.1 OBJECTIVES

---

After going through this unit, you will be able to:

- Define selection sort, quick sort and tree sort
- Write programs using selection, quick and tree sorting techniques

---

### 14.2 SELECTION SORT

---

In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.

Like bubble sort algorithm, the selection sort also requires  $n-1$  passes to sort an array containing  $n$  elements. However, there is a slight difference between



the selection sort and the bubble sort algorithms. In selection sort, the smallest element is the first one to be placed at its correct position, then the second smallest element takes its position, and so on. Whereas, in bubble sort, the largest element is the first one to be placed at its appropriate position, then the second largest element, and so on.

To understand the selection sort algorithm, consider an unsorted array shown here.

8	33	6	21	4
---	----	---	----	---

The steps to sort the values stored in the array in ascending order using selection sort are as follows:

1. In the first pass, the entire array is scanned for the smallest element, which is 4 in this list. It is swapped with the first element, that is, 8. Thus, 4 is placed at its correct position and is not used for any further comparisons.
2. In the second pass, the smallest element is searched from the last four elements, which is 6. It is swapped with the second element, that is, 33.

4	33	6	21	8
---	----	---	----	---

4	6	33	21	8
---	---	----	----	---

3. In the third pass, the smallest element is searched from the last three elements, which is 8. This value is swapped with the third element, that is, 33.

4	6	33	21	8
---	---	----	----	---

4	6	8	21	33
---	---	---	----	----

4. In the fourth pass, the smallest element is searched from the last two elements. Since 21 is smaller than 33, therefore, no changes are made in the list obtained after the third pass, and the list is sorted in ascending order. The sorted list is as follows.

4	6	8	21	33
---	---	---	----	----

## NOTES

**NOTES****Algorithm 14.1 Selection Sort**

```

selection_sort(ARR, size)

1. Set i = 0
2. While (i < size-1)
    Set small = ARR[i]
    pos = i
    Set j = i + 1
    While (j < size)           //searching the smallest element in unsorted list
        If (ARR[j] < small)
            Set small = ARR[j]
            pos = j
        End If
        Set j = j + 1
    End While
    Set ARR[pos] = ARR[i]    //placing the smallest element at its correct position
    Set ARR[i] = small
    Print ARR after (i+1)th pass
    i = i + 1
End While
3. Print "No. of passes: ", i
4. End

```

**Example 14.1:** A program to show sorting of an array using selection sort

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototype*/
void selection_sort(int [], int);

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0; i<size; i++)
        scanf("%d", &ARR[i]);
    selection_sort(ARR, size);
    printf("\nThe sorted array is: ");
}

```

```

        for(i=0;i<size;i++)
            printf("%d  ", ARR[i]);
        getch();
    }

void selection_sort(int ARR[], int size)
{
    int i, j, k, small, pos, count=0;
    for (i=0;i<(size-1);i++)
    {
        small=ARR[i];
        pos=i;
        for (j=i+1;j<size;j++)
        {
            if (ARR[j]<small)
            {
                small=ARR[j];
                pos=j;
            }
        }
        ARR[pos]=ARR[i];
        ARR[i]=small;
        printf("\nArray after pass %d: ", i+1);
        for(k=0;k<size;k++)
            printf("%d  ", ARR[k]);
    }
    printf("\nNo. of passes: %d", i);
}

```

**The output of the program is**

Enter the size of the array (max 20): 5

Enter the elements of the array:

8

6

33

21

5

**NOTES**

**NOTES**

Array after pass 1: 5 6 33 21 8  
 Array after pass 2: 5 6 33 21 8  
 Array after pass 3: 5 6 8 21 33  
 Array after pass 4: 5 6 8 21 33  
 No. of passes: 4  
 The sorted array is: 5 6 8 21 33

---

### 14.3 QUICK SORT

---

Quick sort algorithm also follows the principle of *divide-and-conquer*. However, it does not simply divide the list into halves. Rather it first picks up a partitioning element, called pivot that divides the list into two sub-lists. This is done in a way that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sub-list are greater than the pivot. The same process is applied on the left and right sub-lists separately. This process is repeated recursively until each sub-list contains not more than one element.

The main task in quick sort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate location in the array. The choice of pivot has a significant effect on the efficiency of quick sort algorithm. The simplest way is to choose the first element as pivot. However, the first element is not a good choice, especially if the given list is already or nearly ordered. For better efficiency, the middle element can be chosen as a pivot. Note that, the first element is as taken as pivot for simplicity.

The steps involved in quick sort algorithm are as follows:

1. Initially, three variables `pivot`, `beg` and `end` are taken, such that both `pivot` and `beg` refer to the 0th position, and `end` refers to (n-1)th position in the list.
2. Starting with the element referred to by `end`, the array is scanned from right to left, and each element on the way is compared with the element referred to by `pivot`. If the element referred to by `pivot` is greater than the element referred to by `end`, they are swapped and step 3 is performed. Otherwise, `end` is decremented by 1 and step 2 is continued.
3. Starting with the element referred to by `beg`, the array is scanned from left to right, and each element on the way is compared with the

element referred to by `pivot`. If the element referred to by `pivot` is smaller than the element referred to by `end`, they are swapped and step 2 is performed. Otherwise, `beg` is incremented by 1 and step 3 is continued.

The first pass terminates when `pivot`, `beg` and `end` all refer to the same array element. This indicates that the pivot element is placed at its final position. The elements to the left of this element are smaller than this element, and elements to its right are greater.

To understand the quick sort algorithm, consider an unsorted array:

8	33	6	21	4
---	----	---	----	---

The steps to sort the values stored in the array in ascending order using quick sort are as follows:

- Initially, the index 0 in the list is chosen as the pivot, and the index variables `beg` and `end` are initialized with index 0 and  $n-1$  respectively.

8	33	6	21	4
---	----	---	----	---

- The scanning of elements is started from the end of the list. `ARR[pivot]` (that is, 8) is greater than `ARR[end]` (that is, 4). Therefore, they are swapped.

8	33	6	21	4
↑				↑
pivot				end
beg				

- Now, the scanning of elements is started from the beginning of the list. Since `ARR[pivot]` (that is, 8) is greater than `ARR[beg]` (that is, 33), therefore `beg` is incremented by 1, and the list remains unchanged.

4	33	6	21	8
↑				↑
beg				pivot
				end

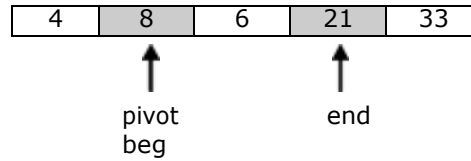
- Next, the element `ARR[pivot]` is smaller than `ARR[beg]`, they are swapped.

4	8	6	21	33
	↑			↑
	pivot			end
	beg			

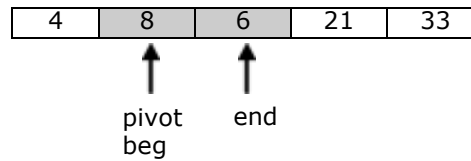
## NOTES

**NOTES**

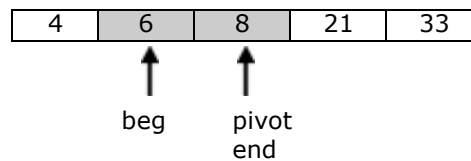
5. Again, the list is scanned from right to left. Since,  $ARR[pivot]$  is smaller than  $ARR[end]$ , therefore the value of  $end$  is decremented by 1, and the list remains unchanged.



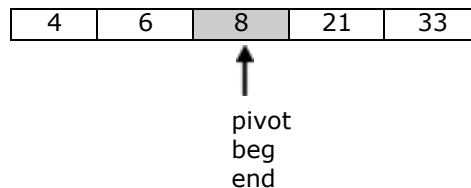
6. Next, the element  $ARR[pivot]$  is smaller than  $ARR[end]$ , value of  $end$  is decremented by 1, and the list remains unchanged.



7. Now,  $ARR[pivot]$  is greater than  $ARR[end]$ , they are swapped.



8. Now, the list is scanned from left to right. Since,  $ARR[pivot]$  is greater than  $ARR[beg]$ , value of  $beg$  is incremented by 1. The list remains unchanged.



At this point since the variables `pivot`, `beg` and `end` all refer to the same element, the first pass is terminated and the value 8 is placed at its appropriate position. The elements to its left are smaller than 8, and elements to its right are greater than 8. The same process is applied on the left and right sub-lists.

**Algorithm 14.2 Quick Sort**

```

quick_sort(ARR, size, lb, ub)
1. Set i = 1           //i is a static integer variable
2. If lb < ub
    Call splitarray(ARR, lb, ub)    //returning an integer value pivot
    Print ARR after ith pass
    Set i = i + 1
    Call quick_sort(ARR, size, lb, pivot - 1)    //recursive call to quick_sort() to
                                                //sort left sub list
    Call quick_sort(ARR, size, pivot + 1, ub);    //recursive call to quick_sort() to
                                                //sort right sub list
    Else if (ub=size-1)
        Print "No. of passes: ", iEnd
    If
3. End

splitarray(ARR, lb, ub)
//splitarray partitions the list into two sub lists such that the elements in left sub list are
smaller than ARR[pivot], and elements in the right sub list are greater than ARR[pivot]
1. Set flag = 0, beg = pivot = lb, end = ub
2. While (flag != 1)
    While (ARR[pivot] <= ARR[end] AND pivot != end)Set
        end = end - 1
    End While
    If pivot = end
        Set flag = 1
    Else
        Set temp = ARR[pivot]
        Set ARR[pivot] = ARR[end]
        Set ARR[end] = temp
        Set pivot = end
    End If
    If flag != 1
        While (ARR[pivot] >= ARR[beg] AND pivot != beg)
            Set beg = beg + 1
        End While
        If pivot = beg
            Set flag = 1
        Else
            Set temp = ARR[pivot]
            Set ARR[pivot] = ARR[beg]
            Set ARR[beg] = temp
            Set pivot = beg
        End If
    End If
    End While
3. Return pivot
4. End

```

**NOTES****Example 14.2:** A program to show sorting of an array using quick sort

```

#include<stdio.h>
#include<conio.h>
#define MAX 20

/*Function prototypes*/
void quick_sort(int [], int, int, int);
int splitarray(int [], int, int);

```

**NOTES**

```

void main()
{
    int ARR[MAX], i, size;
    do
    {
        clrscr();
        printf("\nEnter the size of the array (max %d): ",
MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    quick_sort(ARR, size, 0, size-1);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
    getch();
}

void quick_sort(int ARR[], int size, int lb, int ub)
{
    int pivot, k;
    static int i=0;
    if (lb<ub)
    {
        pivot=splitarray(ARR, lb, ub);
        printf("\nArray after pass %d: ", i+1);
        for (k=0;k<size;k++)
            printf("%d ", ARR[k]);
        i++;
        quick_sort(ARR, size, lb, pivot-1);
        /*recursive call to function to sort left sub-list*/
        quick_sort(ARR, size, pivot+1, ub);
        /*recursive call to function to sort right sub-list*/
    }
    else if (ub==(size-1))
        printf("\nNo. of passes: %d", i);

}

int splitarray(int ARR[], int lb, int ub)
{

```



```

int pivot, beg, end, temp, flag=0;
beg=pivot=lb;
end=ub;
while(!flag)
{
    while ((ARR[pivot]<=ARR[end]) && (pivot!=end))
        end--;
    if (pivot==end)
        flag=1;
    else
    {
        temp=ARR[pivot];
        ARR[pivot]=ARR[end];
        ARR[end]=temp;
        pivot=end;
    }
    if (!flag)
    {
        while ((ARR[pivot]>=ARR[beg]) && (pivot!=beg))
            beg++;
        if (pivot==beg)
            flag=1;
        else
        {
            temp=ARR[pivot];
            ARR[pivot]=ARR[beg];
            ARR[beg]=temp;
            pivot=beg;
        }
    }
}
return pivot;
}

```

**The output of the program is**

Enter the size of the array (max 20): 5

Enter the elements of the array:

6  
5  
4  
3  
2

**NOTES**

**NOTES**

```

Array after pass 1: 2 5 4 3 6
Array after pass 2: 2 5 4 3 6
Array after pass 3: 2 3 4 5 6
Array after pass 4: 2 3 4 5 6
No. of passes: 4
The sorted array is: 2 3 4 5 6

```

---

## 14.4 TREE SORT

---

A tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted. It then traverses the tree so that the elements come out in a sorted order. Its typical use is to sort elements online. As an insertion is completed, the set of elements seen so far is available in sorted order.

The following tree sort algorithm in pseudocode accepts a collection of comparable items and outputs the items in ascending order

```

STRUCTURE BinaryTree
  BinaryTree:LeftSubTree
  Object:Node
  BinaryTree:RightSubTree

PROCEDURE Insert(BinaryTree:searchTree, Object:item)
  IF searchTree.Node IS NULL THEN
    SET searchTree.Node TO item
  ELSE
    IF item IS LESS THAN searchTree.Node THEN
      Insert(searchTree.LeftSubTree, item)
    ELSE
      Insert(searchTree.RightSubTree, item)

PROCEDURE InOrder(BinaryTree:searchTree)
  IF searchTree.Node IS NULL THEN
    EXIT PROCEDURE
  ELSE
    InOrder(searchTree.LeftSubTree)
    EMIT searchTree.Node
    InOrder(searchTree.RightSubTree)

```

```
PROCEDURE TreeSort(Collection:items)
    BinaryTree:searchTree
```

```
    FOR EACH individualItem IN items
        Insert(searchTree, individualItem)
```

```
    InOrder(searchTree)
```

*Other Sorting Techniques*

## NOTES

---

### 14.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

---

1. In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position).
2. Quick sort algorithm also follows the principle of divide-and-conquer.
3. *Tree sort* is a *sorting* algorithm that is based on Binary Search.

---

### 14.6 SUMMARY

---

- In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.
- Like bubble sort algorithm, the selection sort also requires  $n-1$  passes to sort an array containing  $n$  elements.
- Quick sort algorithm also follows the principle of divide-and-conquer. However, it does not simply divide the list into halves. Rather it first picks up a partitioning element, called pivot that divides the list into two sub-lists.
- A tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree so that the elements come out in sorted order.

## NOTES

---

## 14.7 KEY WORDS

---

- **Selection Sort:** It is a sorting algorithm that starts by finding the minimum value in the array and moving it to the first position.
- **Quick Sort:** It is a popular sorting algorithm utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.
- **Tree Sort:** It is a *sort* algorithm that builds a binary search *tree* from the elements to be *sorted*, and then traverses the *tree* (in-order) so that the elements come out in *sorted* order.

---

## 14.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

### Short-Answer Questions

1. Differentiate between bubble sort and selection sort.
2. How does the quick sort algorithm sort data?

### Long-Answer Questions

1. Describe the steps to sort the values stored in the array in ascending order using selection sort.
2. Write a program showing sorting of an array using selection sort and quick sort.

---

## 14.9 FURTHER READINGS

---

Storer, J.A. 2012. *An Introduction to Data Structures and Algorithms*. New York: Springer Publishing.

Preiss, Bruno. 2008. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. *Data Structures and Algorithms*. New Delhi: Laxmi Publications.

McMillan, Michael. 2007. *Data Structures and Algorithms Using C#*. Cambridge, UK: Cambridge University Press.

Louise I. Shelly 2020 Dark Commerce

---

## 14.10 LEARNING OUTCOMES

---

- Define selection sort, quick sort and tree sort
- Write programs using selection, quick and tree sorting techniques