

DATA STRUCTURES WITH C PROGRAMMING

Dr. Anil Kumar Yadav and
Vinod Kumar Yadav

```
val future = for {
    jsonStr <- loadURL("https://www.metaweather.com/api/location/se
    Some(woeid) <- getWoeid(jsonStr)
    daysJson <- getDaysJSON(woeid, days)
} {
    daysJson.foreach(println)
}

def loadURL(url: String): Future[String] = Future { io.Source.fro
def getDaysJSON(woeid: Int, days: Seq[Int]): Future[Seq[String]] =
    val dayFutures = days.map(day => loadURL(s"https://www.metaweb
    Future.sequence(dayFutures)
}
```

AP | ARCLER
PRESS

DATA STRUCTURES WITH C PROGRAMMING

DATA STRUCTURES WITH C PROGRAMMING

Dr. Anil Kumar Yadav and Vinod Kumar Yadav



www.arclerpress.com

Data Structures with C Programming

Dr. Anil Kumar Yadav and Vinod Kumar Yadav

Arcler Press

2010 Winston Park Drive,

2nd Floor

Oakville, ON L6H 5R7

Canada

www.arcлерpress.com

Tel: 001-289-291-7705

001-905-616-2116

Fax: 001-289-291-7601

Email: orders@arcлерeducation.com

e-book Edition 2019

ISBN: 978-1-77361-658-2 (e-book)

This book contains information obtained from highly regarded resources. Reprinted material sources are indicated and copyright remains with the original owners. Copyright for images and other graphics remains with the original owners as indicated. A Wide variety of references are listed. Reasonable efforts have been made to publish reliable data. Authors or Editors or Publishers are not responsible for the accuracy of the information in the published chapters or consequences of their use. The publisher assumes no responsibility for any damage or grievance to the persons or property arising out of the use of any materials, instructions, methods or thoughts in the book. The authors or editors and the publisher have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission has not been obtained. If any copyright holder has not been acknowledged, please write to us so we may rectify.

Notice: Registered trademark of products or corporate names are used only for explanation and identification without intent of infringement.

© 2019 Arcler Press

ISBN: 978-1-77361-337-6 (Hardcover)

Arcler Press publishes wide variety of books and eBooks. For more information about Arcler Press and its products, visit our website at www.arcлерpress.com

ABOUT THE AUTHORS



Dr. Anil Kumar Yadav has done his B.Tech (CSE), M.Tech (I.T) and PhD (Computer Science and Engineering) in the area of Machine Learning specialization in Reinforcement Learning and Artificial Intelligence. He has filed patent on topic Machine Learning, “Method for Reinforcement Learning.”. He has been teaching data structures and Artificial intelligence over the last 13 years at under graduate and post graduate levels. With over a decade of teaching experience in the engineering institutions, he has exemplified his work at various designations. His research interests include data structures, deep learning , machine learning algorithms specially reinforcement learning, computational intelligence, computer vision and image processing, brain computer/machine interface, intelligent informatics, soft-computing in modeling and control, cognitive science. He has presented papers at national and international conferences and published several papers in international journals and also attended several workshops. He is also a reviewer of some of the international journal and conferences.



Vinod Kumar Yadav has done his B.Tech (I.T), M.Tech (CSE). His area of interest is Data mining, Artificial Intelligence and Machine Learning. He has been teaching data structures over the last 7 years at under graduate and post graduate levels. He is also GATE Qualified in Information Technology in 2007. He has published many papers at national, international conferences and international Journals. He also attended several national workshops.

TABLE OF CONTENTS

<i>List of Figures</i>	<i>xi</i>
<i>Preface.....</i>	<i>xvii</i>
Chapter 1 Data Structures	1
1.1. Introduction.....	2
1.2. Data Structure Basic Terminology	2
1.3. Data Structure.....	3
1.4. Introduction To Algorithm	8
1.5. Basic Concept of Function.....	13
1.6. Basic Concept of Pointers	19
1.7. Introduction To Structure.....	22
1.8. Dynamic Memory Allocation In Data Structure.....	28
Chapter 2 Array.....	35
2.1. Introduction.....	36
2.2. Application of Array.....	36
2.3. Definition of Array	37
2.4. Representation of Array.....	37
2.5. Ordered List.....	56
2.6. Sparse Matrices.....	57
2.7. Garbage Collection.....	60
Chapter 3 Recursion	63
3.1. Introduction.....	64
3.2. Recursion	64
3.3. Tower of Hanoi	72
3.4. Backtracking	74

Chapter 4	Stack	77
4.1.	Introduction.....	78
4.2.	Definition of Stack	78
4.3.	Operations on Stack	80
4.4.	Disadvantages of Stack	94
4.5.	Applications of Stack	94
4.6.	Expressions (Polish Notation)	95
4.7.	Evaluation of Postfix Expression	98
4.8.	Decimal To Binary Conversion.....	99
4.9.	Reversing The String.....	102
Chapter 5	Queue	105
5.1.	Introduction.....	106
5.2.	Definition And Structure of Queue.....	106
5.3.	Operations on Queue	106
5.4.	Circular Queue.....	114
5.5.	D-Queue (Double Ended Queue)	119
5.6.	Priority Queues.....	124
5.7.	Application of Queue	128
Chapter 6	Linked List	129
6.1.	Introduction.....	130
6.2.	Definition And Structure of Linked List.....	130
6.3.	Characteristics of Linked List.....	134
6.4.	Types of Linked List.....	135
6.5	Polynomial Representation of Linked List.....	158
Chapter 7	Tree.....	161
7.1.	Introduction.....	162
7.2.	Definition of Trees	162
7.3.	Binary Tree.....	165
7.4.	Binary Tree Representation.....	169
7.5.	Binary Tree Traversal	172
7.6.	Binary Search Tree (BST)	179
7.7.	Height Balanced (AVL) Tree	196

Chapter 8	Graph.....	213
8.1.	Introduction.....	214
8.2.	Definition of Graph	214
8.3.	Representation of Graphs.....	221
8.4.	Graph Traversal.....	224
8.5.	Spanning Tree	229
8.6.	Shortest Path Problem	240
8.7.	Application Of Graph	243
Chapter 9	Sorting.....	245
9.1.	Introduction.....	246
9.2.	Types of Sorting	246
9.3.	Basic Terms of Sorting.....	246
9.4.	Sorting Techniques.....	247
Chapter 10	Searching and Hashing.....	291
10.1.	Introduction.....	292
10.2.	Searching	292
10.3.	Hashing.....	296
10.4.	Collision	300
10.5.	Collision Handling Method	301
10.6.	Rehashing.....	306
10.7.	Application of Hashing	307
Index.....		309

LIST OF FIGURES

- Figure 1.1** Part of abstract data type.
- Figure 1.2** Classification of data structures.
- Figure 1.3** One-dimensional array.
- Figure 1.4** Stack data structure
- Figure 1.5** Queue structure.
- Figure 1.6** Representation of link list.
- Figure 1.7** Tree data structure.
- Figure 1.8** A graph.
- Figure 1.9** Big O notation.
- Figure 1.10** Ω Notation.
- Figure 1.11** ' Θ ' Notation
- Figure 1.12** Output a user-defined function.
- Figure 1.13** Classification of function argument.
- Figure 1.14** Output Call by Value to Pass Arguments.
- Figure 1.15** Output of Call by Reference to Pass Arguments.
- Figure 1.16** Output of Accessing variable through Pointers.
- Figure 1.17** Output of Null Pointers.
- Figure 1.18** Output of Structure Initialization.
- Figure 1.19** Output structure pointer
- Figure 1.20** Output of an Array of Structure.
- Figure 1.21** Output of malloc and free function.
- Figure 1.22** Output of calloc and free function.
- Figure 1.23** Output of realloc Function.
- Figure 2.1** One-dimensional Array.
- Figure 2.2** Array initialization and accessing.
- Figure 2.3** Output of Program Greater Number.

- Figure 2.4** Output of Program 2.3.
- Figure 2.5** Array positions.
- Figure 2.6** 3 x 4 array two-dimensional.
- Figure 2.7** Row major store in an array.
- Figure 2.8** Column major store in an array.
- Figure 2.9** Output of Program 2.4.
- Figure 2.10** Output of Program 2.5
- Figure 2.11** Output of Searching.
- Figure 2.12** Output of Program Insertion.
- Figure 2.13** Output of Program Deletion.
- Figure 2.14** Output of Program Deletion.
- Figure 2.15** Polynomial representation.
- Figure 2.16** Sparse matrix.
- Figure 2.17** Output of Sparse Matrix or not.
- Figure 3.1** Output of a Factorial number Using Iteration Method.
- Figure 3.2** Output of a Factorial number Using Recursion Method.
- Figure 3.3** Output of Fibonacci Series.
- Figure 3.4** Output for Check Prime number.
- Figure 3.5** Initial Setup of Tower of Hanoi
- Figure 3.6** After applying first Step position of Tower
- Figure 3.7** After applying second step position of tower
- Figure 3.8** After applying third step position of tower.
- Figure 3.9** After applying fourth step position of tower.
- Figure 3.10** Tree Example.
- Figure 4.1** Stack Containing Items.
- Figure 4.2** Stack using a 1-dimensional array.
- Figure 4.3** Stack empty condition
- Figure 4.4** Stack full condition.
- Figure 4.5** Performing Push Operation.
- Figure 4.6** Performing Pop Operation.
- Figure 4.7** Stack Operation.

- Figure 4.8** Output Example 4.1.
- Figure 4.9** Output of stack operation using linked list.
- Figure 4.10** Output of Decimal to Binary
- Figure 4.11** Output Reverse a String.
- Figure 5.1** Queue structure.
- Figure 5.2** Array Implementation of Queue.
- Figure 5.3 (a)** Queue in Memory.
- Figure 5.3 (b)** Queue after deleting the first element.
- Figure 5.3 (c)** Queue after inserting an element.
- Figure 5.4** Output of Queue Operations.
- Figure 5.5** Output of Queue operation using Linked List.
- Figure 5.6** Linear Queue.
- Figure 5.7** Circular Queue.
- Figure 5.8** Output of Circular Queue Operations
- Figure 5.9** D-Queue.
- Figure 5.10 (a)** Input-restricted D-queue.
- Figure 5.10 (b)** Output-restricted D-queue.
- Figure 5.11** Output of D-Queue Operations.
- Figure 5.12** Output of Priority Queue.
- Figure 6.1** Structure of a Node
- Figure 6.2** Representation of link list.
- Figure 6.3** Output of Linked List.
- Figure 6.4** A singly-linked list containing three integer values.
- Figure 6.5** Output of Single linked list operations.
- Figure 6.6** Output of Circular Linked List Operations.
- Figure 6.7** Doubly Linked List.
- Figure 6.8** Output of Doubly linked list Operations.
- Figure 6.9** Structure of a polynomial node.
- Figure 6.10** List structure of polynomial.
- Figure 6.11** Linked representation of a polynomial two variable.
- Figure 7.1** Tree.
- Figure 7.2** a, b and c represent a forest.

- Figure 7.3** An example of Tree.
- Figure 7.4** Binary tree.
- Figure 7.5** Strictly Binary Tree
- Figure 7.6** Almost Complete.
- Figure 7.7** Complete Binary Tree.
- Figure 7.8** Extended Binary Tree.
- Figure 7.9** (a) Left -skewed binary tree (b) Right – skewed binary tree.
- Figure 7.10** General Tree.
- Figure 7.11** Expression Tree.
- Figure 7.12** Sequential representation of a binary tree.
- Figure 7.13** Structure of a node binary tree.
- Figure 7.14** Linked List Representation.
- Figure 7.15** Binary Tree (a) and (b).
- Figure 7.16** Binary Tree (a) and (b).
- Figure 7.17** Binary Tree (a) and (b).
- Figure 7.18** Finally Binary Tree.
- Figure 7.19** Computing pointer in a binary tree.
- Figure 7.20** A threaded binary tree.
- Figure 7.21** Binary Search Tree.
- Figure 7.22** Binary Search Tree.
- Figure 7.23** Before deletion.
- Figure 7.24** After deletion.
- Figure 7.25** Before deletion.
- Figure 7.26** After deletion of a node from the tree.
- Figure 7.27** Before deletion.
- Figure 7.28** After the deletion of a node from the tree.
- Figure 7.29** Red-Black Tree.
- Figure 7.30** AVL tree.
- Figure 7.31** Not an AVL tree.
- Figure 7.32** Classification of Rotation.
- Figure 7.33** Balanced AVL search tree.
- Figure 7.34** AVL search tree after performing LL rotation.

- Figure 7.35** Balanced AVL search tree.
- Figure 7.36** AVL search tree after performing RR rotation
- Figure 7.37** Balanced AVL search tree.
- Figure 7.38** AVL search tree after performing LR rotation.
- Figure 7.39** Balanced AVL search tree.
- Figure 7.40** AVL search tree after performing RL rotation.
- Figure 7.41** AVL tree Exercise of 7.4
- Figure 7.42** Weight Balance Tree.
- Figure 8.1** Simple Graph.
- Figure 8.2** A Directed Graph.
- Figure 8.3** An Undirected Graph.
- Figure 8.4** A Complete Graph.
- Figure 8.5** A Sub Graph.
- Figure 8.6** A connected graph.
- Figure 8.7** A Multigraph.
- Figure 8.8** A Graph with four Vertices.
- Figure 8.9** A directed graph with four Vertices.
- Figure 8.10** A Null Graph.
- Figure 8.11** An Isomorphic Graph.
- Figure 8.12** A Homeomorphic Graph.
- Figure 8.13** An Undirected Graph.
- Figure 8.14** A Directed Graph.
- Figure 8.15** Linked list representation of Figure 8.14.
- Figure 8.16** An Undirected Graph.
- Figure 8.17** Linked list of Figure 8.16.
- Figure 8.18** Graph with six vertices.
- Figure 8.19** An undirected graph.
- Figure 8.20** An undirected graph.
- Figure 8.21** An undirected graph.
- Figure 8.22(a)** Undirected Graph.
- Figure 8.22(b)** Spanning Tree.
- Figure 8.23** Undirected Graph.

- Figure 8.24** Minimum spanning tree Graph of 8.23.
- Figure 8.25** Undirected Graph G.
- Figure 8.26** A Minimum Spanning tree of Figure 8.25.
- Figure 8.27** Undirected Graph G.
- Figure 8.28** Undirected Graph G.
- Figure 8.29** A Minimum Spanning tree of Figure 8.27
- Figure 8.30** Unweighted graph
- Figure 8.31** A Graph.
- Figure 9.1** Output of Bubble Sort.
- Figure 9.2** Output of Insertion Sort.
- Figure 9.3** Output of Selection Sort.
- Figure 9.4** Output of Merge Sort element.
- Figure 9.5** Output of merge sort for two unsorted list.
- Figure 9.6** Quick Sort.
- Figure 9.7** Output of Quick Sort element.
- Figure 9.8** Output of Heap Sort.
- Figure 9.9** Output of Radix sort elements.
- Figure 10.1** Output for Binary search.
- Figure 10.2** A small phone directory book as a hash table.
- Figure 10.3** Chaining.
- Figure 10.4** Rehashing.

PREFACE

The significant role of *Data Structures* is famous for Computer Science and Engineering. This book is about the structure, actions and the principle of a different data type that help improve the ability to write an efficient algorithm, program and Analysis algorithm and program complexity. In this book, we present the fundamental concepts, operations, and algorithms for different types of data structures. This makes an understanding of this subject more lucid and makes it more interesting. This book, intended for the first course in the data structure at the junior or senior undergraduate level.

This book prepared according to the syllabus of various Universities and conceived as a textbook for the course of Bachelor of Engineering or Technology of different branches of computer science. The contents of the book have been thoroughly organized and spread over ten chapters. Each chapter begins with the basics and describes with syntax, diagrams, and examples. Each concept of data structure has implemented in the C programming language. The book not only covers the scope of the subject but also explains the philosophy of the subject.

We trust the textbook would meet the requirements of both the teachers and the students. We would very much appreciate receiving any suggestions for improvement in the book from teachers, students, and other readers.

We wish to express our deep thanks to all those who helped in making this book a reality.

We express our gratitude to our publisher and the team of publications that have taken great pains in publishing the book with speed and accuracy.

1

CHAPTER

DATA STRUCTURES

CONTENTS

1.1. Introduction.....	2
1.2. Data Structure Basic Terminology	2
1.3. Data Structure.....	3
1.4. Introduction To Algorithm	8
1.5. Basic Concept of Function	13
1.6. Basic Concept of Pointers	19
1.7. Introduction To Structure.....	22
1.8. Dynamic Memory Allocation In Data Structure.....	28

1.1. INTRODUCTION

In the Computer Programming or Software development, Data Structures is one of the most valuable roles for computer engineers. Use of appropriate data structures enables a computer system to perform its task more efficiently, by influencing the ability of computers to store and retrieve data from any location in its memory. A data structure is a method of how storing data on a computer so that it can be used efficiently. In Computer Programming we are using different types of data to perform, store, access and sent data and information. Computers cannot do without data, so organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

1.2. DATA STRUCTURE BASIC TERMINOLOGY

1.2.1. Data and Data Types

Data are a group of specific facts, numbers, character, and figures. A data item refers to a single unit of values. Data is an unprocessed form of information. Data is plural and datum is singular form. Data items that are divided into subitems are called group items; those are not called elementary items. For examples, an employee's name may divide into three subitems: first name, middle and last name. Data is numerical, character, symbols or any other kind of information. A Data Types consists of a set of values and a set of operations. A data type is a name which refers to kinds of data that variables may hold in the programming language. The data is stored in the memory at some location. By using the name of the variable, one can access the data from that memory location easily. For example in 'C' language, the data types are **int** (integer value), **float** (decimal value), **char** (character), **double** (real value of large range) etc. Data types divided into categories: **Built-in data types (Primitive Data)** and **User define data types (Non-primitive Data)**. Generally, a programming language supports a set of built-in data types and allow the user to define a new type those are called user-defined data types.

Information: Information is related to an attribute or a set of attributes of an object. Examples are the number of students in a class, length of the room, and the component of a computer. The fundamental component of information is the Data; "Information is a collection of data. When data

are processed or organized it gives meaningful and logical knowledge it becomes information.”

Abstract Data Type: The abstract data type is a triple of D-Set of domains, F-Set of functions, A-Axioms in which only what is to be done is mentioned but how is to be done is not mentioned. At ADT, all the all the implementation details are hidden.

In short: **ADT = Type + Function Names + Behaviors of Each function**

An abstract data type can declare the data type and without explanation operations to how it will be implemented. Applications that use the data type are unaware of implementation; they only make use of the operations that defined abstractly. If we want to change implementations, we have to do is rewrite the operations. No matter how large our application is, the cost of changing implementations is the same. The abstract data type is written with the help of the instances and operations.

Implementation: The part that implements the abstract data type.

These two pieces are completely independent. It should be possible to take the implementation developed for one application and use it for a completely different application with no changes.

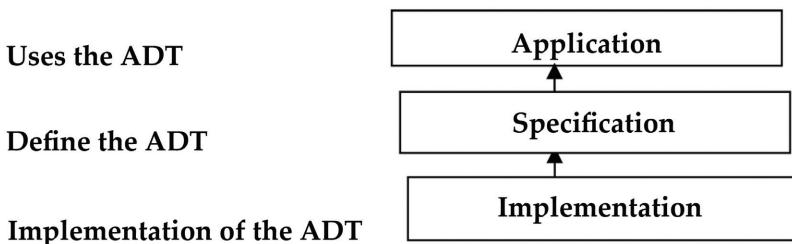


Figure 1.1: Part of abstract data type.

1.3. DATA STRUCTURE

Definition: The data structure can be defined as the organization of data or elements and all possible operations, which are required for those set of data. It is a logical or mathematical representation and organization of any data is known as a data structure. Some of the data structures are arrays, stacks, queues, linked lists, trees, and graphs, etc.

“The well-organized collection of data and all feasible operations on data is called a data structure.”

Data Structure = Organized data (mathematical or logical) + Acceptable operations

1.3.1. Basic Operation of Data Structures

The following operations play a major role in data processing on data structures:

- **Traversing:** Accessing each record exactly once so that certain items in the record may be processed. This process is also called visiting the record.
- **Inserting:** adding a new record to the structure.
- **Deleting:** Removing a record from the structure.
- **Searching:** Finding the location of record with a given key value, or finding the locations of all records which satisfy one or more conditions.
- **Sorting:** Arranging the records in some logical order.
- **Merging:** Combining the records in two different files into a single file.

1.3.2. Classification of Data Structures

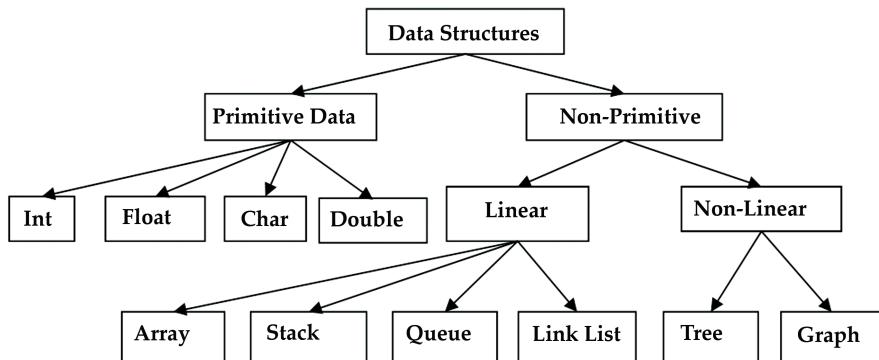


Figure 1.2: Classification of data structures.

The data structures are generally divided into two categories.

1. **Primitive Data type:** These are basic data that are directly operated upon machine instructions. These have different

representations on different computers. Such as int, float, char, double which are defined by programming language itself.

- a) **Integer Data Type:** This type of data is used to store the integer value in a variable. The value should not contain any fraction data.

Example: int a;
 a = 10; is acceptable but
 a = 10.05 is not acceptable.

- b) **Float Data Type:** This type of data is used to store the real values in a variable. The value may have some fractional or decimal data.

Example: float a;
 a = 10; is acceptable but it will be interpreted as a = 10.0
 a = 10.05 is acceptable.

- c) **Character Data Type:** This type of data is used to store some text or alphabets in a variable. The character is always written within single quotes.

Example: char answer;
 answer = "T";
d) **Double Data Type:** This type of data is used to store the numeric value in the variable. It is same as of float data type but the capability of this data type is larger than the float data type size.

Example: double a;
 a = 10.0000000100

2. **Non-Primitive Data type:** These are more sophisticated data, which are derived from the primitive data. The user defines data emphasize the structuring of a group of homogeneous or heterogeneous data items.

Linear Data Structure: Linear data structures are the data structures in which data is arranged or accessed (read or write) in straight sequence or the consecutive way one by one in a list.

Examples: Arrays, Stacks, Queues, and List.

- a) **Array:** it is a collection of all the elements with similar data types. The array is the collection of a finite number of homogenous data element such that the elements of the array are referenced respectively by an index

set consisting of ‘n’ consecutive numbers and stored respectively in successive memory locations.

Example: int A[4]; The computer reserved four storage locations as shown below:

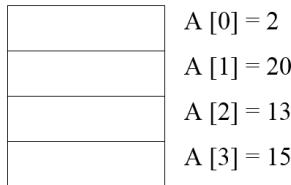


Figure 1.3: One-dimensional array.

- b) **Stack:** it is a data structure which follows last in first out (LIFO) mechanism. It means the first element will insert is to remove the last one. **Example:** if a stack of elements 40, 60, 80, and 100. Here 20 will be the bottommost element and 100 will be the topmost element in a stack. A stack is shown in Figure 1.4.

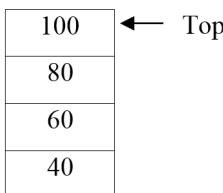


Figure 1.4: Stack data structure.

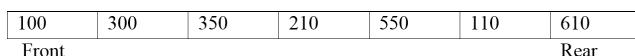


Figure 1.5: Queue structure.

- c) **Queue:** It is a data structure, which follows, first in first out (FIFO) mechanism. It means the first element inserted is the first one to remove. Queue uses two variables rear and front.
- d) **Linked List:** A linked list is a set of nodes where each node has two fields: an information or data and link or next address field. The ‘data’ field store actual piece of

information, which may be an integer, a character, a string or even a large record and ‘link’ field is used to point to next node. Hence link list of integer 20, 40, 60, 80 is shown in Figure 1.6.



Figure 1.6: Representation of link list.

Non-Linear data structures: Non-Linear data structures are the data in which data may be arranged in unordered or hierarchical manner; for example, Trees, Graphs.

- a) **Tree:** it is a non-linear data structure in which data are arranged in a sorted sequence. It is used to represent the hierarchical relationship of several data items.

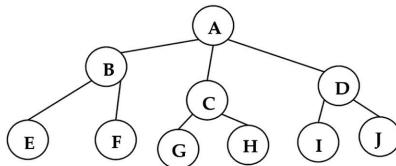


Figure 1.7: Tree data structure.

- b) **Graph:** A graph is a non-linear data structure which consists of set of nodes called vertices V and set of edges E which links vertices. From Figure 1.8, $V(G) = \{1, 2, 3, 4, 5, 6\}$, $E(G) = \{(1,2), (2,1), (2,3), (3,2), (1,4), (4,1), (4,5), (5,4), (5,6), (6,5), (3,6), (6,3)\}$

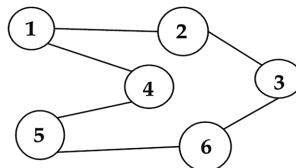


Figure 1.8: A graph.

1.4. INTRODUCTION TO ALGORITHM

An algorithm is composed of a finite set of steps, each of which may require one or more operations. An algorithm is a finite set of instructions that, if followed, accomplished a particular task. **Example:** The algorithm makes the sum of two numbers 20 and 30.

Step 1: Start

Step 2: Store 20 into X and 30 into Y.

Step 3: Calculate SUM = X + Y.

Step 4: Print the value of SUM.

Step 5: Stop.

Basic Characteristics of An algorithm:

- **Input:** an algorithm should have zero or more inputs are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** if we trace out the instructions of the algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Effectiveness:** Every instruction must be very basic so that it can be carried out, in principle, by a person using only a pencil and paper.

A program is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for the program.

Program = Data structure + Algorithms

1.4.1. Analysis of Programs

The analysis of the program does not mean simply working of the program, but to check whether for all possible situations program works or not. The analysis also involves working of the program efficiently. Efficiency in the sense,

1. The program requires less amount of storage space.
2. The programs get executed in very less amount of time.

The time and space are factors which determine the efficiencies of the program. The time required for execution of the program cannot be computed in terms of seconds because of the following factors.

1. The hardware of the machine.
2. The amount of time required by each machine instruction.
3. The amount of time required by the compilers to execute the instruction.
4. The instruction set.

1.4.2. Complexity of an Algorithm

The analysis of algorithms is a major task in computer science. In order to compare algorithms, there must be some criteria for the major effects of algorithms. The algorithm can be evaluated by a variety of criteria. The rate of growth of the time or space required to solve larger and larger instance of a program. There are three conditions for complexity theory is as follows:

- **Worst case:** The worst-case time complexity is the function defined by the maximum amount of time needed by the algorithm for an input of size, ‘n.’ Therefore, it is a function which defined by the maximum number of steps taken for any instance that size ‘n.’
- **Average case:** The average case time complexity is the execution of an algorithm having typical input data of size ‘n.’ Therefore, it is a function which defined by the average number of steps taken for any instance that size ‘n.’
- **Best case:** The best case time complexity is the minimum amount of time that an algorithm requires for an input of size ‘n.’ Therefore, it is a function which defined by the minimum number of steps taken for any instance that size ‘n.’

Space Complexity: The space complexity of a program is the amount of memory it needs to run to completion. The space needed by a program is the sum of the following components-

- A fixed part that includes space for the code, space for simple variable and fixed size component variables.
- The variable part that consists of the space needed by the component variable where size is dependent on the particular problem.

The space requirement $S(P)$ of any algorithm P may, therefore, be written as

$$S(P) = c + S_p$$

where c is a constant and S_p instance characteristics.

Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion. The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the characteristics. We assume that the compiled program will run several times without recompilation. We concern ourselves with just time of a program. This runtime is denoted by T_p (instance characteristics). If we knew the compiler characteristics to be used, we could continue to determine the number of additions, subtractions, multiplications, divisions, compares, stores and so on, that would be made by the code for P .

$$T_p(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_m \text{MUL}(n) + \dots$$

Here n denotes the instance characteristics, and c_a, c_s, c_m and so on.

Efficiency of Algorithms

If we have two algorithms that perform the same task, and the first one has a computing time of $O(n)$ and the second of $O(n^2)$, then will usually prefer the first one. The reason for this is that as n increases the time required for the execution of second algorithms will get far more than the time required for the execution of the first. We will study various values for computing function for the constant values.

$$\log_2 n > n > n \log_2 n > n^2 > n^3 > 2^n$$

Notice how the times $O(n)$ and $O(n \log_2 n)$ grow much more slowly than the others. For large datasets algorithms with a complexity greater than $O(n \log_2 n)$ are often impractical. The very slow algorithm will be the one having time complexity 2^n .

1.4.3. Complexity Notations of Algorithm

To choose the best algorithm, we need to check the efficiency of each algorithm. The efficiency can be measured by the computing time complexity of each algorithm asymptotic notation is a shorthand way to represent the time complexity. Various notations such as Ω , Θ , and O used are called asymptotic notions.

- Big (O) Notation:** The big oh notation is denoted by ‘O.’ It is a method of representing the upper bound of algorithm’s running time. Using big oh notation we can give the longest amount of time taken by the algorithm to complete.

Definition: Let $f(n)$ and $g(n)$ be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of the input and $n > n_0$. Similarly, c is some constant such that $c > 0$. We can write

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

Then $F(n)$ is big oh of $g(n)$. It is also denoted as $F(n) \in O(g(n))$. In other words, $F(n)$ is less than $g(n)$ is multiple of some constant c .

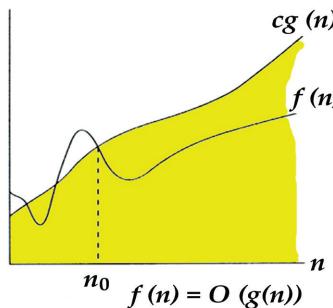


Figure 1.9: Big O notation.

Example: Consider function $F(n) = 2n + 2$ and $g(n) = n^2$. We have to find some constant c , so that $F(n) \leq c * g(n)$. As $F(n) = 2n+2$ and $g(n) = n^2$.

Find for $n = 1$, $F(n) = 2n + 2 = 2(1) + 2$, $F(n) = 4$; and $g(n) = n^2 = (1)^2$, $g(n) = 1$

i.e., $F(n) > g(n)$

if $n = 2$ then, $F(n) = 2n + 2 = 2(2) + 2$, $F(n) = 6$; and $g(n) = n^2 = (2)^2$, $g(n) = 4$

i.e., $F(n) > g(n)$

if $n = 3$ then, $F(n) = 2n + 2 = 2(3) + 2$, $F(n) = 8$; and $g(n) = n^2 = (3)^2$, $g(n) = 9$

i.e., $F(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain, $F(n) < g(n)$. Thus always upper bound of existing time is obtained by big O notation.

2. **Omega ‘ Ω ’ Notation:** Omega notation is denoted by ‘ Ω .’ This notation is used to represent the **lower bound** of algorithm’s running time. Using omega notation we can denote the shortest amount of time taken by the algorithm.

Definition: A function $F(n)$ is said to be in $\Omega(g(n))$ if $F(n)$ is bounded below by some positive constant multiple of $g(n)$ such that $F(n) \geq c * g(n)$ for all $n \geq n_0$.

It is denoted as $F(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

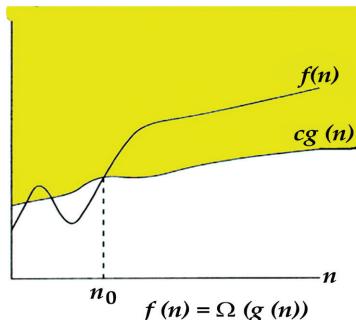


Figure 1.10: Ω notation.

Example: consider $F(n) = 2n^2 + 5$ and $g(n) = 7n$.

if $n = 0$, then $F(n) = 2(0)^2 + 5 = 5$ and $g(n) = 7(0) = 0$ i.e., $F(n) > g(n)$

if $n = 1$, $F(n) = 2(1)^2 + 5 = 7$ and $g(n) = 7(1) = 7$ i.e., $F(n) = g(n)$

If $n = 2$, $F(n) = 2(2)^2 + 5 = 13$ and $g(n) = 7(2) = 14$ i.e., $F(n) < g(n)$

If $n = 3$, $F(n) = 2(3)^2 + 5 = 23$ and $g(n) = 7(3) = 21$ i.e., $F(n) > g(n)$

Hence we can conclude that $n > 3$, we obtain $F(n) > c * g(n)$. It can be represented as $2n^2 + 5 \in \Omega(n)$. Thus always lower bound of existing time is obtained by Ω notation.

3. **‘ Θ ’ Notation:** The theta notation is denoted by Θ . By this method the running time is between upper bound and lower bound.

Definition: Let $F(n)$ and $g(n)$ be two nonnegative functions. There are two positive constants namely c_1 and c_2 such that $c_1 g(n) \leq F(n) \leq c_2 g(n)$.

Thus we can say that, $F(n) \in \Theta(g(n))$

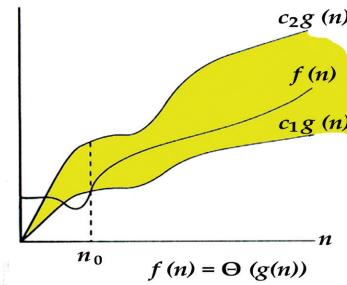


Figure 1.11: ‘ Θ ’ Notation

Example: consider $F(n) = 2n + 8$ and $g(n) = 7n$, where $n \geq 2$.

Similarly $F(n) = 2n + 8$ and $g(n) = 7n$

i.e., $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$. The theta notation is more precise with big oh and omega notation.

The complexity of Notations:

Notation	Best Case	Worst Case	Average Case
Big ‘O’	(Upper) Yes		
Omega ‘Ω’		(Lower) Yes	
Theta ‘ Θ ’			(Middle) Yes

1.5. BASIC CONCEPT OF FUNCTION

When we are writing a program in C language, each program uses at least one function, which is known as **main()** function. If we are writing code for an application in C language, you require performing the same task more than one time. In such case you have two options:

1. Use the same set of code each time when you want to perform the task.
2. Create a function to perform that task, and just call it every time when you need to perform that task.

Definition: A function is a collection of instruction or code that performs together and completes a particular task. A function can also be referred to as a method or a sub-routine or a procedure, etc.

Functions are used because of the following reasons:

- To get better readable and writable of code.
- Introduce the concept of code reusability, the same function can be used in any program rather than writing the same code.
- Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- Reduces the size of the code, a duplicate set of statements are replaced by function calls.

1.5.1. Types of Functions

There are basically two types of functions.

- **Predefined Standard Library Functions:** In the C language we use some predefined functions such as main(), puts(), gets(), printf(), scanf(), getch() etc. These functions which are already define and declare in header files (.h files like stdio.h, conio.h), so we call them whenever you like to use these functions.
- **User Defined Functions:** The functions that we create in a program are known as user-defined functions. So, we will study how to build user-defined functions and how to use them in C Programming.

1.5.2. Build a User Define Function

The common form of a function definition in C programming language is as follows.

Syntax: return_type function_name(parameter list) //Header of function

```
{
    //Body of the function
}
```

Example:

```
int greater(int n1, int n2) {
    int result; /* local variable declaration */
    if (n1 > n2)
        result = n1;
    else
        result = n2;
    return result; }
```

A function consists of a header and a body. Some parts of a function described below:

Function Return Type: A function may return a value. The `return_type` is the type of value that function returns. Several functions are performing the operations without returning a value. In this condition, the `return_type` is the void.

Function Name: it is the identifier of a function. The name and the parameter list compose together and make the function name.

Function Parameters: A parameter is like a placeholder. When a function is executed, we pass a value to the defined parameter. This value is referred to as the actual parameter or argument. The parameter list refers to the type, order, and a number of the parameters in a function. Parameters are optional in a function may contain or not.

Function Body: The function body contains the statement or collection of code that defines what the function performs and return.

1.5.3. Function Declarations and Call

A function declaration identifies the compiler about a function name, return type, parameters and how to call the function during compilation of the program. The body of the function can be defined independently.

Syntax: `return_type function_name(parameter list);`

Example: `int greater(int n1, int n2);`

Parameter names are not essential in function declaration only their type is required, so the following is also a valid declaration:

```
int greater(int, int);
```

Declaration of a function is required when we define a function in one program and we call that function in another program. In this case, we should declare the function at the top of the program and be calling the function.

Calling a Function: any program when we call a function, the control of the program is transferred to the called function. This called function performs a defined task and its return when the statement is executed or when the closing brace is reached. The program control returns back to the main program. Calling a function, you only need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

Program 1.1: Build a User Defined Function

```
#include<stdio.h>
#include<conio.h>
int greater(int n1, int n2); /* function declaration */
void main ()
{
    clrscr();                  /* local variable definition */
    int a = 18;
    int b = 29;
    int ret;
    ret = greater(a,b);        /* calling a function to greater value
*/
    printf("Greater value is : %d\n", ret);
    getch();  }
int greater(int n1, int n2) /* function returning the greater between two
numbers */
{
    int result;                /* local variable declaration */
    if (n1 > n2)
        result = n1;
    else
        result = n2;
    return result;
}
```

The output of the program is given in Figure 1.12.



Greater value is : 29

-

Figure 1.12: Output a user-defined function.

1.5.4. Function Arguments

On the basis of function, parameter arguments are categorized. There are two types of function are available in the C language, as shown in Figure 1.13.

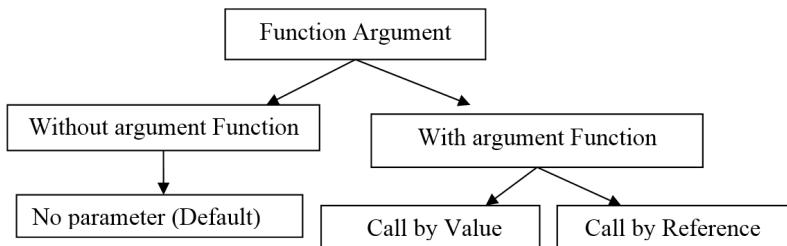


Figure 1.13: Classification of function argument.

There are two ways in which arguments can be passed to a function:

- a) **Call by value:** This method copies the actual value of an argument into the formal parameter of the function. New memory area created for the passed parameters can be used only within the function. The actual parameters cannot be modified here. In this condition, changes are made to the inside parameter the function have no effect on the argument. By default, C uses call by value to pass arguments in the parameter. In general, it means the code within a function cannot change the arguments used to call the function.

Program 1.2: Call by Value to Pass Arguments

```

#include<stdio.h>
#include<conio.h>
int sum(int n1, int n2); /* function declaration */
void main()
{
    clrscr();
    int ans;
    int n1 = 100;
    int n2 = 500;
    ans = sum(n1,n2);           /* calling a function to get addition value */
    printf("The sum of two numbers is: %d\n",ans);
    getch();
}
int sum(int a,int b)          /* function returning the sum of two numbers */
  
```

```

{
return a + b;
}

```

The output of the program is given in Figure 1.14.



```
The sum of two numbers is: 600
```

Figure 1.14: Output Call by Value to Pass Arguments.

- b) **Call by Reference:** This method copies address of an argument into the formal parameter. Inside the function, Address operator (&) is used in the parameter for the called function. This means that changes made to the parameter affect the original variables.

Program 1.3: Call by Reference to pass Arguments

```

#include<stdio.h>
#include<conio.h>
void swap(int *n1, int *n2)
{
int temp;
temp = *n1;
*n1 = *n2;
*n2 = temp;
}
void main()
{
int n1 = 10, n2 = 20;
clrscr();
printf("\nBefore swap value of n1:%d,"n1);
printf("\nBefore swap value of n2:%d,"n2);
swap(&n1, &n2); // passing value to function
printf("\nafter swap value of n1: %d,"n1);
printf("\nafter swap value of n2: %d,"n2);
getch();
}

```

The output of the program is given in Figure 1.15.

```
Before swap value of n1:10
Before swap value of n2:20
after swap value of n1: 20
after swap value of n2: 10
```

Figure 1.15: Output of Call by Reference to Pass Arguments.

1.5.5. Main Features of Function

- The basic intention of the function is the reuse of code.
- Use of any function we can invoke (call) any another functions.
- All the time compilation will take place from top to bottom.
- All the time execution process will starts from main() and ends with main() function only.
- In function definition first line is called function declaration or function header.
- Always function declaration should be matched with function declaratory.
- In implementation whenever a function does not return any values back to the calling place then specify the return type.
- **void** means nothing that is no return value.
- In implementation whenever a function returns other than void then specify the return type as return value type that is one type of return value it is returning the same type of return statement should be mentioned.
- Default return type of any function is an **int**.
- Default parameter type of any function is **void**.

1.6. BASIC CONCEPT OF POINTERS

A pointer is a variable which stores the address of another variable, of the memory location. Similar to any variable or constant, we must declare a pointer before using it to store any variable address. If the variable name is preceded by an asterisk or * then such variable is called pointer variable. The Syntax of a pointer variable declaration is:

Syntax: Data type * variable_name;

Here, the data type is the type of pointers it must be valid in C data and variable _name is the name of the pointer variable. The asterisk * used to declare a pointer.

Some examples are given that are the valid pointer declarations:

```
int *i; /* pointer to an integer */
float *f; /* pointer to a float */
char *c /* pointer to a character */
```

The * symbol informs the compiler that we use a pointer variable, the actual data type of the value of all pointers.

1.6.1. Operations on Pointers

There are some essential operations used for pointers initialization and accessing the variable through pointers.

- a. We define and declare a pointer variable;
- b. Assign the address of a variable to a pointer; and
- c. Lastly, access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. We have used two important operators * and &. The * means ‘value at a specified memory address’ and & means the ‘memory address at.’ The following example makes use of these operations.

Example:

```
int *ip; /* pointer variable define */
int a,b; /* normal variable define */
a = 100; /* store value in a variable */
ip = &a; /* store memory address of a in ip */
```

```
b = *ip; /* getting value from memory address in ip and store in b
variable */
```

Example:

```
int *ip; float a,b;
a = 4.3;
b = 3.8;
ip = &a /* this is wrong*/
```

Program 1.4: Accessing variable through Pointers

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

```
#include<conio.h>
void main ()
{
clrscr();
int a = 20; /* actual variable declaration */
int *ip; /* pointer variable declaration */
ip = &a; /* store address of a in pointer variable*/
printf("\nAddress of a variable: %x\n," &a); /*address stored in pointer
variable*/
printf("\nAddress stored in ip variable: %x\n," ip); /*access value using
pointer*/
printf("\nValue of *ip variable: %d\n," *ip);
getch();
}
```

The output of the program is given in Figure 1.16.

```
Address of a variable: fff4
Address stored in up variable: fff4
Value of ip variable: 20
```

Figure 1.16: Output of Accessing variable through Pointers.

1.6.2. NULL Pointers

The NULL pointer means a constant with a value of zero defined in several standard libraries. This is done at the time of variable declaration. A pointer variable that is assigned by NULL is called a null pointer.

Program 1.5: For Null Pointer

```
#include<stdio.h>
#include<conio.h>
void main ()
{
clrscr();
int *ptr = NULL;
printf("The value of ptr is : %x\n," ptr );
getch();
}
```

The output of the program is given in Figure 1.17.



```
The value of ptr is : 0
```

Figure 1.17: Output of Null Pointers.

1.7. INTRODUCTION TO STRUCTURE

The structure is a non-primitive or user-defined data type in C languages which allows joining together different data types to store a particular type of record. The structure is slightly related to an Array. The only difference is that array is used to store a collection of homogeneous or similar data types, even as the structure can store a collection of the dissimilar type of data. The structure is used to represent a record. Suppose you want to store records of Student which consists of student name, address, roll number and age. You can define a structure to hold this information.

1.7.1. Defining a Structure

To define a structure **struct** keyword is used. **struct** defines a new user-defined data type which is a collection of different types of data that is known as members of the structure. The closing braces in the structure type declaration must be followed by a semicolon “;”.

Syntax:

```
struct structure_name {  
    data_type member1;  
    data_type member2;  
    .  
    .  
    data_type member n; //declaration of different data types  
};
```

Example:

```
struct Library  
{  
    char Bookname[15];  
    char Authorname[15];  
    int volume;  
    float price;    };
```

Here the above example **struct Library** declares a structure to hold the details of the book in the library which consists of four data fields,

namely Bookname, Authorname, volume, and price. These fields are called **structure elements or members**. Each data member can have a different data type, like Bookname and Author name is of char type, the volume is int type and price is of float type etc. **The library** is the name of the structure and is called structure tag.

1.7.2. Declaring Structure Variables

After a structure is defined, it is feasible to declare variables of a **structure**. **Structure** variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in the following two ways.

- a) Declaring Structure variables individually

```
struct Student
{
    char name[10];
    int age;
    int semester;
} ;      struct Student S1 , S2; //declaring variables of Student
```

- b) Declaring Structure Variables with Structure definition

```
struct Student
{
    char name[20];
    int age;
    int semester;
} S1, S2;           //declaring variables of Student
```

Here S1 and S2 are variables of structure Student.

1.7.3. Accessing Structure Members

In the Structure, each data members can be accessed and assigned values in the variable using a particular method. In order to assign a value to a structure member, the member name must be associated with the structure variable using a dot “.” operator also called **period or member access operator**.

```
struct Library
{
    char Bookname[15];
    char Authorname[15];
    int volume;
```

```

float price;
} L1, L2;
L1.price = 200.50; //L1 is variable of Library type and the price is a member of Library

```

We can also use scanf() to give values to structure members through the terminal.

```

scanf(" %s ", L1.Bookname);
scanf(" %d ", &L1.price);

```

1.7.4. Structure Initialization

Similar to any other data type, structure variable can also be initialized at compile time.

```

struct Library
{
char Bookname[30];
char Authorname[30];
int volume;
float price; };
struct Library L1 = { "Data Structure," "Vinod Kumar Yadav," 1, 275.50 };
//initialization

```

or

We can initialize value using member access operator.

```

struct Library L1;
L1.Book name = "Data Structure"; //initialization of each member separately
L1.Author name = "Vinod Kumar Yadav";
L1.volume = 1;
L1.price = 275.50;

```

Program 1.7: Structure Initialization

```

#include<stdio.h>
#include<conio.h>
struct Library {
char Bookname[50];
char Authorname[50];
int volume;
float price; } l;
void main()
{
clrscr();

```

```

printf("Enter information:\n");
printf("Enter Book Name: ");
scanf("%s," l.Bookname);
printf("\nEnter Author Name:");
scanf("%s,"l.Authorname);
printf("\nEnter volume number: ");
scanf("%d," &l.volume);
printf("\nEnter Price of Book: ");
scanf("%f," &l.price);
printf("\nDisplaying Information:\n");
printf("Book Name: ");
puts(l.Bookname);
printf("Author name: %s\n,"l.Authorname);
printf("Volume: %d\n," l.volume);
printf("Price: %f \n," l.price);
getch();  }

```

The output of the program is given in Figure 1.18.

```

Enter information:
Enter Book Name: java
Enter Author Name:balaguru
Enter volume number: 2
Enter Price of Book: 345.5
Displaying Information:
Book Name: java
Author name: balaguru
Volume: 2
Price: 345.500000

```

Figure 1.18: Output of Structure Initialization.

1.7.5. Pointers to Structures

In the structure we can define pointers to structures in the same way as we define pointer to any other variable. By using pointer to structure the members can be accessed with \rightarrow .

Syntax: struct structure_name * variable_name;

Example: struct Books *struct_pointer;
 struct_pointer = &Book1;

Just now, we can store the address of a structure variable in the above defined pointer variable. To get the address of a structure variable we place the '&' operator before the structure's name as follows. To access the

members of a structure using a pointer to that structure, you must use the → operator as follows:

```
struct_pointer→ title;
```

Program 1.8: For structure pointer

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;    };
void printBook(struct Books *book); /* function declaration */
void main()  {
    clrscr();
    struct Books Book1;      /* Declare Book1 of type Book */
    struct Books Book2;      /* Declare Book2 of type Book */
    strcpy(Book1.title, "C Programming");
    strcpy(Book1.author, "Yashwant Kanitkar");
    strcpy(Book1.subject, "C Programming Tutorial");
    Book1.book_id = 649;
    strcpy(Book2.title, "Data Structure");
    strcpy(Book2.author, "Vinod Kumar Yadav");
    strcpy(Book2.subject, "Data Structure Tutorial");
    Book2.book_id = 651;
    /* display Book1 information by passing address of Book1 */
    printBook(&Book1);
    /* display Book2 information by passing address of Book2 */
    printBook(&Book2);
    getch();    }
void printBook(struct Books *book)      {
    printf("Book title : %s\n," book->title);
    printf("Book author : %s\n," book->author);
    printf("Book subject : %s\n," book->subject);
    printf("Book book_id : %d\n," book->book_id);    }
```

The output of the program is given in Figure 1.19.

```
Book title : C Programming
Book author : Yashwant Kanitkar
Book subject : C Programming Tutorial
Book book_id : 649
Book title : Data Structure
Book author : Vinod Kumar Yadav
Book subject : Data Structure Tutorial
Book book_id : 651
```

Figure 1.19: Output structure pointer.

We are using the simple structure the members of structure accessed with the dot operator and when the access structure by the pointer variable then use → operator.

1.7.6. Array of Structure

In the structure, we can also declare an array of structure. Each element of the array represents a structure variable. **Example:** struct Staff st[5]; The below program define an array st of size 5 elements. Each element of array st is of type employee.

Program 1.9: Array of Structure

```
#include<stdio.h>
#include<conio.h>
struct Staff {
    clrscr();
    char sname[20];
    int salary; };
    struct Staff st[5];
    int i,j;
    void getdata()
    {
        for(i = 0;i<3;i++) {
            printf("\nEnter %dst Staff record,"I + 1);
            printf("\n Staff name\t");
            scanf("%os,"st[i].sname);
            printf("\nEnter Staff salary\t");
            scanf("%d,"&st[i].salary); }
        printf("\nDisplaying Staff record");
        for(i = 0;i<3;i++) {
            printf("\nStaff name is %s,"st[i].sname);
            printf("\nSalary is %d,"st[i].salary); }
```

```

}
void main()
{
clrscr ();
getdata();
getch();
}

```

The output of the program is given in Figure 1.20.

```

Enter 1st Staff record
Staff name      Aman
Enter Staff salary      18000
Enter 2st Staff record
Staff name      Seema
Enter Staff salary      15000
Enter 3st Staff record
Staff name      Vijay
Enter Staff salary      20000
Displaying Staff record
Staff name is Aman
Salary is 18000
Staff name is Seema
Salary is 15000
Staff name is Vijay
Salary is 20000_

```

Figure 1.20: Output of an Array of Structure.

1.8. DYNAMIC MEMORY ALLOCATION IN DATA STRUCTURE

Before learning Dynamic memory allocation, we understand the difference between static memory allocation and dynamic memory allocation.

S.N.	Static Memory Allocation	Dynamic Memory Allocation
1.	Memory is allocated at compile time.	Memory is allocated at runtime.

2.	Memory can't be increased while executing the program.	Memory can be increased while executing the program.
3.	Used an array.	Used in the linked list.

Dynamic memory allocation allows your program to get more memory space while running, or to free it if it's not essential. Dynamic memory allocation function allows you to manually handle the memory space in your program. In the C programming language is feasible by four functions that using **stdlib.h** header file.

- **malloc()**: Allocates the requested size of memory and returns a pointer first byte of memory.
- **calloc()**: Allocates multiple blocks of requested memory space for array elements, initializes to zero and then returns a pointer to memory.
- **realloc()**: Reallocate earlier allocated space by malloc() or calloc() functions.
- **free()**: free the earlier allocated space.

1.8.1. Define Dynamic Memory Allocation Function

1. **malloc() Function** : malloc stands for “memory allocation.” The size can be obtained of any data by **size of()** operator. The malloc () function keep a block of memory a particular size and return a pointer of type void which can be typecast into a pointer of any form. It returns NULL if memory is not sufficient for allocation.

Syntax: Pointer = (datatype_cast*) malloc(byte-size);

Example: ptr = (int*) malloc(50 * size of(int));

This statement will allocate either 100 or 200 according to the size of int 2 or 4 bytes respectively and the pointer points to the address of the first byte of memory.

2. **free() function**: allocated memory that is created by either calloc() or malloc() function doesn't get freed on its own. You must use free() function to release the space.

Syntax: free(pointer_variable)

Example: free(ptr); This statement frees the space allocated in the memory pointed by ptr.

Program 1.10: Use of malloc() and free()

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main()
{ clrscr();
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int)); //memory allocated using malloc
    printf("\nallocated memory is: %u",ptr);
    if(ptr == NULL)
    {     printf("Error! memory not allocated.");
        exit(0);     }
    printf("\nEnter elements of array:\n ");
    for(i = 0; i < n; ++i)     {
        scanf("%d",ptr + i);
        sum += *(ptr + i);     }
    printf("Sum = %d", sum);
    free(ptr);
    printf("\nfree of allocated memory is: %u",ptr);
    getch(); }
```

The output of the program is given in Figure 1.21.

```
Enter number of elements: 4
allocated memory is: 1982
Enter elements of array:
20
40
30
10
Sum = 100
free of allocated memory is: 1982_
```

Figure 1.21: Output of malloc and free function.

3. **calloc(function):** calloc stands for “contiguous allocation.” The only difference between malloc() and calloc() function is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets

all bytes to zero. Syntax: **Pointer = (data_typecast*) calloc(n, data_size);**

Example: `ptr = (float*) calloc(20, sizeof(float));`

Above code allocates contiguous space in memory for an array that has 20 float elements each elements of size that use 4 bytes.

Program 1.11: Using C calloc() and free()

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main()
{ clrscr();
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d," &n);
    ptr = (int*) calloc(n, sizeof(int));
    printf("\nMemory allocated is:%u,"ptr);
    if(ptr == NULL)
    {     printf("Error! memory not allocated.");
        exit(0); }
    printf("\nEnter elements of array: ");
    for(i = 0; i < n; ++i)      {
        scanf("%d," ptr + i);
        sum += *(ptr + i);      }
    printf("Sum = %d," sum);
    free(ptr);
    printf("\nFree memory is:%u,"ptr);
    getch();
}
```

The output of the program is given in Figure 1.22.

```
Enter number of elements: 4
Memory allocated is:1966
Enter elements of array: 2
4
8
10
Sum = 24
Free memory is:1966_
```

Figure 1.22: Output of calloc and free function.

4. **realloc()** function: If the earlier allocated memory is insufficient or more memory required, you can change the earlier allocated memory size using realloc() function.

Syntax: pointer = realloc(pointer_variable, newsize);

Example: ptr = realloc(ptr, 4); // Here, ptr is reallocated with size of 4.

Program 1.12: Using realloc()

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main()
{
clrscr();
int *ptr, i , n1, n2;
printf("Enter size of array: ");
scanf("%d," &n1);
ptr = (int*) malloc(n1 * sizeof(int));
printf("\nAddress of allocated memory:\n");
for(i = 0; i < n1; ++i) {
printf("%u\n,"ptr + i); }
printf("\nEnter new size of array: ");
scanf("%d," &n2);
ptr = (int *) realloc(ptr, n2);
printf("\nAddress of reallocated memory:\n");
for(i = 0; i < n2; ++i) {
printf("%u\n," ptr + i); }
getch();
}
```

The output of the program is given in Figure 1.23.

```
Enter size of array: 4
Address of allocated memory:
1952
1954
1956
1958
Enter new size of array: 6
Address of reallocated memory:
1952
1954
1956
1958
1960
1962
```

Figure 1.23: Output of realloc Function.

2

CHAPTER

ARRAY

CONTENTS

2.1. Introduction.....	36
2.2. Application of Array	36
2.3. Definition of Array	37
2.4. Representation of Array.....	37
2.5. Ordered List.....	56
2.6. Sparse Matrices.....	57
2.7. Garbage Collection.....	60

2.1. INTRODUCTION

In computer programming, assume that we want to store ten integer numbers. If we use programming's simple variable and data type concepts, then we need ten variables of the int data type. It was simply because we had to store just ten integer numbers. Now let's assume we have to store 500 integer numbers. Are we going to use 500 variables? To handle such condition, the programming languages provide a concept to solve this situation of many variables declaration problem using **ARRAY**. An array is a data structure, that can store a fixed-size collection of elements of the same data type and it is often more useful an array as a collection of variables of the same data type.

An array is one of the simplest linear data structures. The array is a non-primitive or users define data structures. An array holds a sequence of data elements, generally of the same size or the same data type. Individual elements are accessed by index using a consecutive range of integers, as opposed to an associative array. The primary (primitive) data types, namely char, int, float, double. Although these data types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used to handle limited amounts of data. In several applications, we need to deal a huge volume of data in terms of reading, processing, and printing. To process such large amounts of data, we need a powerful data type that would be providing facilities for efficient storing, accessing and manipulation of data items. C supports a derived data type that is known as Array. That can be used for such applications.

2.2. APPLICATION OF ARRAY

There are many applications where we use the concept of Array:

- The array is used to store the number of elements belonging to similar or same data type.
- The array used for maintaining multiple variable names using a single name.
- The array can be used for Sorting Elements belonging to similar or same data type.
- The array can be used for Perform Matrix Operation.
- The array can be used in CPU Scheduling Operation.

2.3. DEFINITION OF ARRAY

The array is a linear data structure; it is a collection of all the elements with similar data types. The array is the collection of a finite number of homogeneous data elements such that the elements of the array are referenced respectively by an index set consisting of n consecutive numbers and stored respectively in successive memory locations. The arrays can be represented as one dimensional, two dimensional or multidimensional.

Advantages of sequential organization of the data structure:

- Elements can be stored and retrieved very efficiently in sequentially in the sequential organization with the help of index or memory location.
- All the elements are stored at the continuous memory location. Hence searching of an element from the sequential organization is easy.

Disadvantages of sequential organization of data structure:

- Insertion and deletion of elements becomes complicated due to sequential nature.
- Memory fragmentation occurs if we remove the elements randomly.
- For storing the data large continuous free block of memory is required.

2.4. REPRESENTATION OF ARRAY

Syntax: Data Type Name_of_Array_Variable [length];

Example: int x [25]; floaty [15];

Here ‘x’ and ‘y’ is the name of the array, x is store the integer type data, y is stored the float type data, within the square bracket size or length of the array is given. This array is of integer type all the type elements are of integer type in array ‘x.’ The number of elements is called the **length** or **size** of the array. Let, x is a linear array the address of the first element of A is denoted by **Base (x)** and called the **Base address** of x.

2.4.1. Array Initialization

We can initialize an array in C language, the syntax looks like this:

Syntax: Data_Type Name_of_Array_Variable [length] = {data1, data2....., data n};

Example: int a[5] = {10, 11, 22, 33, 44};

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array. If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0.

Example: int a[10] = {0, 1, 2, 3, 4, 5, 6}; would initialize a[7], a[8], and a[9] to 0. When an array definition includes an initializer, the array dimension maybe not there, and the compiler will infer the dimension from the number of initializers.

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

Would declare, define, and initialize an array A of 5 elements (i.e., just as if you'd typed int A[5]). Only the dimension is not there; the brackets [] remain to indicate that A is, in fact, an array. In the case of arrays of char, the initializer may be a string constant:

char st1[7] = "Hello,"; char st2[10] = "India,";

2.4.2. One-Dimensional Arrays

A list of elements can be store given variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array. The subscript can begin with number 0. That is A [0] is allowed.

Example: if we want to represent a set of five numbers, say (2, 20, 13, 15, 55, 65) by an array variable A, then we may declare as:

int A[6];

	A [0] = 2
	A [1] = 20
	A [2] = 13
	A [3] = 15
	A [4] = 55
	A [5] = 65

Figure 2.1: One-dimensional Array.

Program 2.1: A C program for Array initialization and accessing

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[6];
clrscr();
printf(" Enter the element which want to store\n");
for (int i = 0; i < 6; i++)
{
scanf("%d",&a[i]);
}
printf("Print the stored element in array\n");
for (i = 0;i<6;i++)
{
printf("%d\n",a[i]);
}
getch();
}
```

The output of the program is given in Figure 2.2.

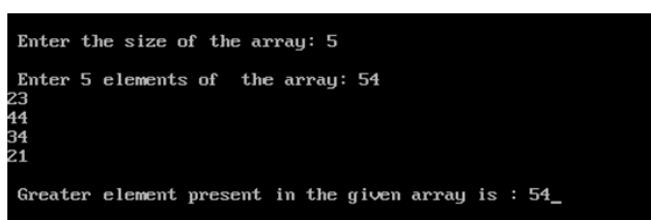
```
Enter the element which want to store
2
20
13
15
55
65
Print the stored element in array
2
20
13
15
55
65
```

Figure 2.2: Array initialization and accessing.

Program 2.2: Find out the Greater Number

```
#include <stdio.h>
#include<conio.h>
void main()
{
clrscr();
int a[20], size, i, large;
printf("\n Enter the size of the array: ");
scanf("%d," &size);
printf("\n Enter elements of the array: %d , " size);
for (i = 0; i < size; i++)
scanf("%d," &a[i]);
large = a[0];
for (i = 1; i < size; i++)
{
if (large< a[i])
larg = a[i];
}
printf("\n Greater element present in the given array is : %d," large);
getch();
}
```

The output of the program is given in Figure 2.3.



```
Enter the size of the array: 5
Enter 5 elements of the array: 54
23
44
34
21
Greater element present in the given array is : 54_
```

Figure 2.3: Output of Program Greater Number.

Write a Program in C puts even & odd elements of an array and finds the odd and elements of the array.

Program 2.3:

```
#include <stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    int ARR[10], Odd[10], Even[10];
    int i, j = 0, k = 0, n;
    printf("Enter the size of array \n");
    scanf("%d," &n);
    printf("Enter the elements of the array \n");
    for (i = 0; i < n; i++) {
        scanf("%d," &ARR[i]);
        for (j = 0; j < n; j++)
        {
            if (ARR[i] % 2 == 0)
            {
                Even[j] = ARR[i];
                j++;
            }
            else
            {
                Odd[k] = ARR[i];
                k++;
            }
        }
    }
    printf("The elements of Odd number are \n");
    for (i = 0; i < j; i++)
```

```

{
printf("%d\n," Odd[i]);
}
printf("The elements of Even number are \n");
for (i = 0; i < k; i++)
{
printf("%d\n," Even[i]);
}
getch();
}

```

The output of the program is given in Figure 2.4.

```

Enter the size of array
6
Enter the elements of the array
11
12
13
14
15
16
The elements of Odd number are
11
13
15
The elements of Even number are
12
14
16

```

Figure 2.4: Output of Program 2.3.

2.4.3. Two Dimensional Arrays

Two dimensional arrays are called **matrices** in mathematics and **tables** in business application. Hence, two-dimensional arrays are also called a matrix array which is arranged in rows and columns. Two-dimensional arrays are declared as follows:

Syntax: Data_Type Array_name [row_size][column_size];

Example: int A[3][4];

Here 'A' is the name of the 2-d array; A is store the integer type data, within the square bracket size or length of array is given 3 for row and 4 for column.

	Column 0	Column 1	Column 2	Column 3
Row 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Row 1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
Row 2	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Figure 2.5: Array positions.

2-D Array Initialization: int a[3][4] = {10, 11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 120};

Or

```
int a[3][4] = {
{10, 11, 22, 33}, /* initializers for row indexed by 0 */
{44, 55, 66, 77}, /* initializers for row indexed by 1 */
{88, 89, 110, 120} /* initializers for row indexed by 2 */};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array. A two-dimensional m x n array A is a collection of m * n data elements such that each element specified by a pair of integer I and J, called subscript such that $1 \leq I \leq m$ and $1 \leq J \leq n$.

There is a standard way of representing a two-dimensional m x n. The element of array A[I, J] appears in row I and column J. one such type two-dimensional array with dimension 3 row and 4 column is shown in figure 2.6. A [3, 4], where I = 3 is rows, and J = 4 is columns.

	0	1	2	3
Rows				
0	A[0,0]	A[0,1]	A[0,2]	A[0,3]
1	A[1,0]	A[1,1]	A[1,2]	A[1,3]
2	A[2,0]	A[2,1]	A[2,2]	A[2,3]

Figure 2.6: 3 x 4 array two-dimensional.

The length of the two-dimensional array can be calculated as follows:

$$\text{Length} = \text{Upper bound} - \text{Lower bound} + 1$$

Example: An array A [4] [5] can be represented as, A [0..3, 0..4] an array has four rows (0, 1, 2 and 3) and five columns (0, 1, 2, 3, 4).

Thus Length of the row will be calculated: **Row = Upper bound – Lower bound + 1**

$$= 3 - 0 + 1 = 4$$

Length of column will be calculated: **Column = Upper bound – Lower bound + 1**

$$= 4 - 0 + 1 = 5$$

Row Major Representation: If the elements of the 2-d array are stored in the row-wise method then it is called row major representation. It means that the store completes the first row and then complete second row is stored and so on.

Example: if we want to store elements 15, 25, 35, 45, 55, 65, 75, 85, 95, 105, 115, 125 then the elements will be filled up by row wise method as follows (consider A [3][4]).

	0	1	2	3
0	15	25	35	45
1	55	65	75	85
2	95	105	115	125

Figure 2.7: Row major store in an array.

Address of elements in Row-major: For an array, A, base (A) is the address of the first element of the array. That is, if declared by:

```
int A [m][n];
```

Where m and n are the ranges of the row and column, base (A) is the address of A [0, 0]. To calculate the address of a random element A [I][J].

$$\text{A}[I, J] = \text{Base}(A) + (I * n + J) * \text{data size}$$

Example: an array A [3][4] is stored as in Figure 2.7. The base address is 1000, m = 3, n = 4 and size = 2. Then the address of A [1][2] is computed as where I = 1 and J = 2.

$$A[1][2] = 1000 + (1 * 4 + 2) * 2 = 1012$$

Column-major representation: If the elements of the 2-d array are stored in column wise way then it is called column major representation. It

means that the complete first column is stored and then the complete second column is stored and so on.

Example: if we want to store elements 15, 25, 35, 45, 55, 65, 75, 85, 95, 105, 115, 125 then the elements will be filled up by column wise manner as follows (A [3][4]).

	0	1	2	3
0	15	45	75	105
1	25	55	85	115
2	35	65	95	125

Figure 2.8: Column major store in an array.

Address of elements in column-major: For an array, A, base (A) is the address of the first element of the array. That is if declared by, int A [m][n];

Where m and n are the ranges of the row and column, base (A) is the address of A [0][0]. To calculate the address of a random element A [I][J].

$$\text{Address of element } A[I][J] = \text{base address} + (m * J + I) * \text{data size}$$

Example: A [3][4] is stored as in Figure 2.8. The base address is 1000. Here m = 3, n = 4 and size = 2. Then the address of A [1][2] is computed as where I = 1 and J = 2.

$$A[1][2] = 1000 + (3*2+1) * 2 = 1014$$

Exercise 2.1: Consider integer array int A [4][4] declared. If the base address is 100, find the address of the element A [2][3] with row major and column major of array.

Solution:

Row major representation

Given that, base address = 100, size is integer = 2 byte, m = 4, n = 4, I = 2, J = 3.

$$\text{Then } A[I][J] = \text{Base (A)} + (I * n + J) * \text{size}$$

$$A[2][3] = 100 + (2 * 4 + 3) * 2 = 122$$

Column major representation

Given that, base address = 100, size is integer = 2 byte, m = 4, n = 4, I = 2, J = 3.

$$\text{Then } A[2][3] = \text{base address} + (m * j + i) * \text{size}$$

$$= 100 + (4 * 3 + 2) * 2 = 128$$

Exercise 2.2: Find the address of B [10][10] when a array is stored as (a) Row major (b) Column major. If each of an array B [20][50] require 4 bytes of storage, base address of data is 1000.

Solution: Row major representation

Given that, base address = 1000, size is integer = 4 byte, m = 20, n = 50, I = 10, J = 10.

Then B [I][J] = Base (B) + (I * n + J) * size

$$B[10][10] = 1000 + (10 * 50 + 10) * 4$$

$$= 1000 + 2040 = 3040$$

Column major representation

Given that, base address = 1000, size is integer = 4 byte, m = 20, n = 50, I = 10, J = 10.

Then A [I][J] = Base address + (m * J + I) * size

$$= 1000 + (20 * 10 + 10) * 4$$

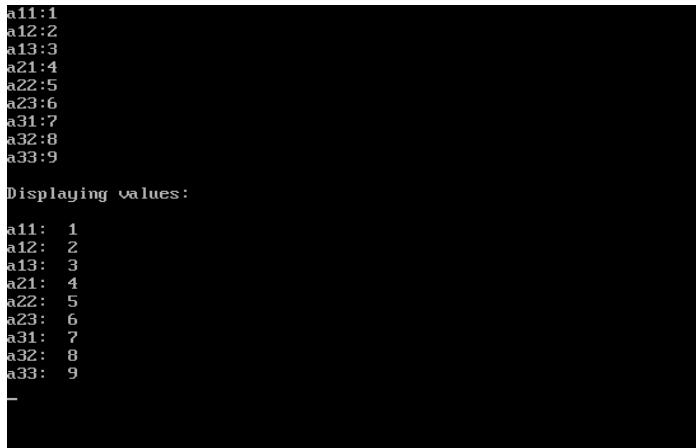
$$= 1000 + 840 = 1840$$

Program 2.4: write a program in C language for Two Dimensional Arrays to store and display values.

```
#include <stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    int a[3][3],i,j;
    for(i = 0; i < 3; ++i)
    {
        for(j = 0; j < 3; ++j)
        {
            printf("a%d%d:", i + 1, j+1);
            scanf("%d", &a[i][j]);
        }
    }
}
```

```
printf("nDisplaying values: nn");
for (i = 0; i < 3; ++i)
{
    for(j = 0; j < 3; ++j)
    {
        printf("a%d%d: %d\n", i + 1, j+1, a[i][j]);
    }
}
getch();
}
```

The output of the program is given in Figure 2.9.



```
a11:1
a12:2
a13:3
a21:4
a22:5
a23:6
a31:7
a32:8
a33:9

Displaying values:

a11: 1
a12: 2
a13: 3
a21: 4
a22: 5
a23: 6
a31: 7
a32: 8
a33: 9

-
```

Figure 2.9: Output of Program 2.4.

Program 2.5: Using Two-Dimensional Array Performed the Addition.

```
#include <stdio.h>
#include<conio.h>
void main()  {
clrscr();
int a[2][2], b[2][2], c[2][2];
```

```
int i, j;
printf("Enter elements of 1st matrix\n");
for(i = 0; i<2; ++i)
    for(j = 0; j<2; ++j) {
        printf("Enter a%d%d: , " I + 1, j+1);
        scanf("%d," &a[i][j]);
    }
printf("Enter elements of 2nd matrix\n");
for(i = 0; i<2; ++i)
    for(j = 0; j<2; ++j) {
        printf("Enter b%d%d: , " I + 1, j+1);
        scanf("%d," &b[i][j]);
    }
    for(i = 0; i<2; ++i)
        for(j = 0; j<2; ++j) {
            c[i][j] = a[i][j] + b[i][j];
        }
    printf("\nSum Of Matrix:\n");
    for(i = 0; i<2; ++i)
        for(j = 0; j<2; ++j) {
            printf("%d\t," c[i][j]);
            if(j == 1)
                printf("\n");
        }
    getch();
}
```

The output of the program is given in Figure 2.10.

```
Enter elements of 1st matrix
Enter a11: 1
Enter a12: 2
Enter a21: 3
Enter a22: 4
Enter elements of 2nd matrix
Enter b11: 5
Enter b12: 6
Enter b21: 7
Enter b22: 8
Sum Of Matrix:
6      8
10     12
```

Figure 2.10: Output of Program 2.5.

2.4.4. Operations on Array

These are the following operations can be performed on arrays:

- Traversing
- Searching
- Insertion
- Deletion
- Update
- Sorting
- Merging

1. **Traversing:** It is used to access each data item exactly once so that it can be processed.

Example: Consider a given linear array A as below:

An	0	1	2	3	4
[index]					
A[value]	15	25	35	45	55

Here we will start from the beginning and will go to the last element and during this process, we will access the value of each element exactly once as below:

- A [0] = 15
- A [1] = 25
- A [2] = 35
- A [3] = 45
- A [4] = 55

Program 2.1 is the example of traversing operation.

2. **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.

Example: Consider a given linear array A as below:

An	0	1	2	3	4
[index]					
A[value]	15	25	35	45	55

Suppose we find out the location of the data item to be searched is 35. We will start from the beginning and will compare 35 with each element. This process will continue until the element is found or array is finished. Here:

- Step 1. Compare 35 with 15 is not equal so go to next element.
- Step 2. Compare 35 with 25 is not equal so go to next element.
- Step 3. Compare 35 with 35 is equal, so 35 is found in location 2.

Program 2.6: For Searching (Linear Search)

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int arr[50],k,i,size,pos = 0;
printf("Enter size for SEQUENTIAL SEARCH\n");
scanf("%d",&size);
printf("Enter %d elements\n",size);
for(i = 0;i<size;i++)
{
scanf("%d",&arr[i]);
}
printf("Enter a number to be search\n");
scanf("%d",&k);
for(i = 0;i<size;i++)
{
if(arr[i] == k)
{
pos = i;
break;
}
}
if(pos!= 0)
printf("%d is found in the list, at position %d\n",k,pos);
else
printf("%d is not in the list\n",k);
getch();
}
```

The output of the program is given in Figure 2.11.

```
Enter size for SEQUENTIAL SEARCH
5
Enter 5 elements
45
56
55
34
67
Enter a number to be search
34
34 is found in the list, at position 3
```

Figure 2.11: Output of Searching.

3. **Insertion:** It is used to add element in a given position if the array already created. Insert operation is to insert one or more data elements into an array. On the requirement, a new element can be added at the beginning, end, or any given index of the array.

Algorithm for Insertion Operation

Consider an Array A with the size of N elements and K is a position of the array such that $K \leq N$. Following is the algorithm where ITEM is inserted into the K-th position of A:

- Step 1. Start
- Step 2. Initialize size N of Linear Array into a new variable I, set $I = N$
- Step 3. One Increment size of Array Set $N = N + 1$
- Step 4. Repeat steps 5 and 6 while $I \geq K$
- Step 5. Set $A[I + 1] = A[I]$
- Step 6. Set $I = I - 1$
- Step 7. Set $A[K] = ITEM$
- Step 8. Stop.

Program 2.7: For Insertion

```
#include <stdio.h>
#include <conio.h>
void main()  {
clrscr();
```

```

int A[] = {11,33,52,73,80};
int item = 10, k = 2, n = 5;
int i = 0, j = n;
printf("The original array elements are :\n");
for(i = 0; i<n; i++) {
    printf("A[%d] = %d \n," i, A[i]);
}
n = n + 1;
while(j >= k) {
    A[j+1] = A[j];
    j = j - 1;
}
A[k] = item;
printf("The array elements after insertion :\n");
for(i = 0; i<n; i++) {
    printf("A[%d] = %d \n," i, A[i]);
}
getch();
}

```

The output of the program is given in Figure 2.12.

```

The original array elements are :
A[0] = 11
A[1] = 33
A[2] = 52
A[3] = 73
A[4] = 80
The array elements after insertion :
LA[0] = 11
LA[1] = 33
LA[2] = 10
LA[3] = 52
LA[4] = 73
LA[5] = 80
-

```

Figure 2.12: Output of Program Insertion.

4. **Deletion:** It is used to delete an element in a given position if the array already created (collection of data items). Deletion means for removing an existing element from the array and re-organizing all elements of an array.

Algorithm for Deletion Operation

Consider a Array A with size of N elements and K is a position of array such that $K \leq N$. Following is the algorithm where ITEM is delete into the K-th position of A:

- Step 1. Start
- Step 2. Initialize a new variable I, set $I = K$
- Step 3. Repeat steps 4 and 5 while $I < K$
- Step 4. Set $A[I-1] = A[I]$
- Step 5. Set $I = I + 1$
- Step 6. Set $N = N - 1$
- Step 7. Stop

Program 2.8: For Deletion

```
#include <stdio.h>
#include<conio.h>
void main()
{
clrscr();
int A[] = {21,23,45,57,68};
int k = 2, n = 5;
int i, j;
printf("The original array elements are :\n");
for(i = 0; i<n; i++) {
printf("A[%d] = %d \n," i, A[i]);
}
j = k;
while(j < n) {
A[j-1] = A[j];
j = j + 1;
}
n = n - 1;
printf("The array elements after deletion :\n");
```

```

for(i = 0; i<n; i++)
{
printf("A[%d] = %d \n," i, A[i]);
}
getch();
}

```

The output of the program is given in Figure 2.13.

```

The original array elements are :
LA[0] = 21
LA[1] = 23
LA[2] = 45
LA[3] = 57
LA[4] = 68
The array elements after deletion :
A[0] = 21
A[1] = 45
A[2] = 57
A[3] = 68

```

Figure 2.13: Output of Program Deletion.

5. **Update:** It is used to update element in a given position if the array already created (collection of data items). Update operation means to renew an existing element from the array at a given position.

Algorithm for Deletion Operation

Consider an Array A with size of N elements and K is a position of array such that $K \leq N$. Following is the algorithm where ITEM is update into the K-th position of A:

- Step 1. Start
- Step 2. Set $A[K-1] = \text{item}$
- Step 3. Stop

Program 2.9: For Deletion

```

#include <stdio.h>
#include<conio.h>
void main()
{

```

```

clrscr();
int A[] = {13,34,55,76,87};
int k = 4, n = 5, item = 110;
int i, j;
printf("The original array elements are :\n");
for(i = 0; i<n; i++) {
    printf("A[%d] = %d \n," i, A[i]);
}
A[k-1] = item;
printf("The array elements after updation :\n");
for(i = 0; i<n; i++) {
    printf("A[%d] = %d \n," i, A[i]);
}
getch();
}

```

The output of the program is given in Figure 2.14.

```

The original array elements are :
A[0] = 13
A[1] = 34
A[2] = 55
A[3] = 76
A[4] = 87
The array elements after updation :
A[0] = 13
A[1] = 34
A[2] = 55
A[3] = 110
A[4] = 87

```

Figure 2.14: Output of Program Deletion.

6. **Sorting:** It is used to arrange the data items in some order i.e., in ascending or descending order in case of numerical data or alphanumeric data.

Example: Consider a given linear array A as below:

An	0	1	2	3	4
[index]					
A[value]	25	15	45	05	52

After arranging the elements in increasing order, the array will be:

An	0	1	2	3	4
[index]					
A[value]	05	15	25	45	52

7. **Merging:** It is used to combine the data items of two sorted files into a single file in the sorted form. We have sorted linear array A as below:

An	0	1	2	3	4
[index]					
A[value]	05	15	25	45	52

Second sorted linear array B as below:

An	0	1	2	3	4
[index]					
A[value]	10	20	30	40	50

After merging merged array C is as below:

An	0	1	2	3	4	5	6	7	8	9
[index]										
A[value]	05	10	15	20	25	30	40	45	50	52

2.5. ORDERED LIST

An ordered list is nothing but a set of elements. Such a list sometimes called as a linear list. More abstractly, we can say that an ordered list is either empty or it can be written as $(a_1, a_2, a_3, a_4, \dots, a_n)$ where the a_i are atoms from set S.

Example: 1. List of one digit numbers $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$

2. Days in a week (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)

Operation on the Ordered list: Following operations can be possible on an ordered list.

1. Display of list.
2. Searching a particular element from the list.
3. Insertion of any element in the list.
4. Deletion of any element from the list.

5. Read the list from left-to-right or right-to-left.

2.5.1. Polynomials

One classic example of an ordered list is a polynomial. A polynomial is the sum of a term consists of variable, coefficient, and exponent.

Various operations which can be performed on the polynomial are:

1. Addition of two polynomials.
2. Multiplication of two polynomials.
3. Evaluation of polynomials.

Representation of array polynomial using a single dimensional array:

For representing a single variable polynomial one can make use of the one-dimensional array. In a single dimensional array, the index of an array will act as the exponent and coefficient can be stored in that particular index, which can be represented as follows: For example, $3x^4 + 5x^3 + 7x^2 + 10x - 19$. This polynomial can be stored in a single dimensional array.

0	-19	
1	10	
2	7	
3	5	The coefficient of the polynomial
4	3	
5		
6		

The exponent of respective coefficient

Figure 2.15: Polynomial representation.

2.6. SPARSE MATRICES

Matrices play a very important role in solving many interesting problems in various scientific and engineering applications. It is, therefore, necessary for us to design efficient representation or matrices. Normally matrices are represented in a two-dimensional array. In a matrix, if there are m rows and n columns, then the space required to store the numbers will be $m \times n \times s$ where s is the number of bytes required to store the value. Suppose, there are 10 rows and 10 columns and we have to store the integer values, then the space complexity will be bytes. **10 x 10 x 2 = 200 bytes**

Here 2 bytes are required to store an integer value and time complexity will be $O(n^2)$ because the operations that are carried out on matrices need to scan the matrices one row at a time and individual columns in that row result in use of two nested loops.

Definition: A sparse matrix is that matrix which has a very few non-zero elements, as compared to the size $m \times n$ of the matrix. Or matrices with a relatively high proportion of zero entries are called **sparse matrices** or **sparse array**.

For example: if the matrix is of size 100×100 and only 10 elements are non-zero. Then for accessing these 10 elements, one has to make 10,000 times scan. Also, only 10 spaces will be with non-zero elements remaining spaces of the matrix will be filled with zero only. We have to allocate the memory of $100 \times 100 \times 2 = 20,000$.

2.6.1. Representation of Sparse Matrix

In the sense that basically the sparse matrix means very few non-zero elements having in it. Rest of the spaces is having the values zero which are basically useless values or simply empty values. Consider the matrix

A =	0	0	6	0	9	0	0	
	2	0	0	7	8	0	4	
	10	0	0	0	0	0	0	
	0	0	12	0	0	0	0	
	0	0	0	0	0	0	0	
	0	0	0	3	0	0	5	

Figure 2.16: Sparse matrix.

Program 2.10: A C program to determine if a given Matrix is a Sparse Matrix or not.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
```

```
int array[10][10];
int i, j, m, n;
int count = 0;
printf("Enter the order of the matix \n");
printf("Row :");
scanf(" %d," &m);
printf("Column :");
scanf("%d," &n);
printf("Enter the co-efficients of the matix \n");
for (i = 0; i < m; ++i) {
    for (j = 0; j < n; ++j) {
        scanf("%d," &array[i][j]);
        if (array[i][j] == 0)
            ++count;
    }
}
if (count > ((m * n) / 2)) {
    printf("The given matrix is sparse matrix \n");
} else
    printf("The given matrix is not a sparse matrix \n");
printf("There are %d number of zeros," count);
getch(); }
```

The output of the program is given in Figure 2.17.

```
Enter the order of the matix
Row :3
Column :3
Enter the co-efficients of the matix
1
0
1
0
0
0
2
0
0
The given matrix is sparse matrix
There are 6 number of zeros_
```

Figure 2.17: Output of Sparse Matrix or not.

2.6.2. Application of Sparse Matrices

Sparse matrices can be useful for computing, large-scale applications that dense matrices cannot handle. One such application involves solving partial differential equations by using the finite element method. The finite element method is one method of solving partial differential equations (PDEs). Compared with the other numeric method, the finite difference method, the finite element method can handle geometrically complicated domains straightforwardly. The finite element technique also has the flexibility to solve problems that very quickly. Using the finite element technique to solve PDEs involves the following steps.

- Separate the problem domain into discrete elements.
- Formulate the PDE into an equivalent variation problem.
- Create a finite-dimensional subspace and find out its basis on the discrete elements.

2.7. GARBAGE COLLECTION

When some object is created and is not used for a long time, then such an object is called **garbage**. Garbage collection is the method of detecting and reclaiming free nodes or objects. In this method, object no longer in use remain allocated and undetected until all available storage has been allocated but are no longer in use recovered. Garbage collection is carried out in two phases. In a first phase called the marking phase, all nodes that are accessible from an external pointer marked. The second phase, called the collection phase, involves proceeding sequentially through memory and freeing all nodes that have not been marked. The second phase is trivial when all nodes are of fixed size.

The memory allocation for the objects is done from the heap. The garbage collection is a technique in which all such garbage is collected and recycled. The garbage collector cleans up the heap so that the memory occupied by unused objects can be freed and can be used allocating for the new objects. The garbage collection algorithm works in two steps, first **Mark**: In the marking process all the live objects are located and marked them as non-collected objects or all nodes that are accessible from an external pointer are marked. Second, **Collection or Sweep**: The collection phase involves proceeding sequentially through memory and freeing all nodes that have not been marked. In this step, all the unmarked objects are swept from the heap and the space that has been allocated by these objects can be used

for allocating new objects. But the drawback of this mark and collection algorithm is that multiple fragments of memory get created hence the technique called compaction is used.

Compaction: The process of moving all used (marked) objects to one end of memory and all the available memory to the other end is called compaction. In this technique, the allocated space is moved in the heap and free space is moved up to form a contiguous block of free space in the heap. The garbage collection is also called as **automatic memory management**.

Advantages

- The manual memory management done by the programmer (after malloc use of free or delete at the end of the function) is time-consuming and error-prone. Hence this automatic memory management is done.
- Reusability of the memory can be achieved with the help of garbage collection.

Disadvantage

- The execution of the program is paused or stopped during the process of garbage collection.

Thus we have learned a dynamic data structure represented by the linear organization.

3

CHAPTER

RECURSION

CONTENTS

3.1. Introduction	64
3.2. Recursion	64
3.3. Tower of Hanoi	72
3.4. Backtracking	74

3.1. INTRODUCTION

Generally, a recursion is a block of programming code or function. That allows the programmer to express their operations for executing again and again. In C, this takes the form of a function that calls itself. A recursive function is to imagine them as a process being performed where one of the instructions is to “repeat the procedure.” Recursion works very similarly to a loop because it repeats the same code. It must be possible for the “procedure” sometimes to be completed without the recursive call.

3.2. RECURSION

Recursion is a programming method in which the group of code (function) calls itself again and again for a particular input. Recursion is a procedure of doing the same task again and again for some specific input. **Recursion is,**

- an approach of thinking about problems.
- a method for solving a particular problem.
- related to the mathematical induction concept.

A method is **recursive** if it can call itself; either directly or indirectly.

Directly call: void fun ()

```

    {
        fun () ...
    }
```

Indirectly call: void fun ()

```

    {
        get() .. }
        void get()  {
            ... fun() ... }
```

Any program a Recursion is said to **direct** if a function calls itself. It is said **indirect** if there is a sequence of more than one function call eventually calls the first function, such as function fun() calls function get (), which in turn calls fun().

3.2.1. Principles of Recursive Function

Two important conditions must be satisfied by any recursive function: First, each time a function calls itself; it must be closer, in some sense to a solution. Second, there must be a decision criterion for stopping the process or computation.

For designing the good recursive program, we must make certain assumptions such as:

- Best Case: Best case is the terminating condition for the problem while designing any recursive function.
- If conditions: if a condition in the recursive algorithm defines the terminating condition.
- When a recursive program is subjected to execution, the recursive function calls will not be executed immediately.
- The initial parameter input value pushed onto the stack.
- Each time a function is called a new set of local variable and formal parameters are again pushed onto the stack and execution starts from the beginning of the function using changed new value. This process is repeated until a base condition is reached.
- Once a base condition or stopping condition is reached the recursive function calls pop elements from the stack and returns a result of the previous values of the function.
- A sequence of returns that the solution to the original problem is obtained.

3.2.2. Applications of Recursive Functions

Various mathematical operations that can be defined by recursive function these are:

- Factorial
- Fibonacci
- GCD (greatest common denominator)
- Fourier Transform
- Prime Number

The concept of Factorial Function: One of the simplest examples of a recursive definition is that the factorial functions. For example, if 7 factorial has to be calculated, It will be = $7*6*5*4*3*2*1 = 5040$, which is defined for positive integers N by the equation,

$$N! = N \times (N - 1) \times (N - 2) \times \dots \times 2 \times 1$$

```

factorial (N)
if (N = 0) then print 1.
else
```

$$N * \text{factorial}(N - 1)$$

Written a recursive function which matches this definition:

```
fact (int n)      // function of factorial
{
    if (n == 0) return 1;
    else
        return n*fact(N - 1); }
```

Note how this function calls itself to evaluate the next term. Eventually, it will reach the termination condition and exit. We can trace this computation in the same way that we trace any sequence of function calls.

fact(7)

fact(6)

fact(5)

fact(4)

fact(3)

fact(2)

fact(1)

return 1

return $2 * 1 = 2$

return $3 * 2 = 6$

return $4 * 6 = 24$

return $5 * 24 = 120$

return $6 * 120 = 720$

return $7 * 720 = 5040$

Above fact() implementation exhibits the two main mechanism that are required for every recursive function.

- The base case returns a value without making any subsequent recursive calls. It performs for one or more particular input values in which the function can be evaluated without recursion. For fact(), the base case is $N = 1$.
- The decreasing step is the inner part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. In addition, the sequence of parameter values must converge to the base case. For fact (), the reduction step is $N * \text{fact}(N - 1)$ and N decreases by one for each call, so the sequence of parameter values converges to the base case of $N = 1$.

Program 3.1: A Factorial Program Using Iteration Method

```
#include<stdio.h>
#include<conio.h>
void main() {
    clrscr();
    int n, i;
    unsigned long factorial = 1;
    printf("Enter an integer: ");
    scanf("%d",&n); // display error when the user enters a negative number
    if (n < 0)
        printf(" Factorial of a negative number doesn't exist.");
    else {
        for(i = 1; i<= n; ++i)
        {
            factorial = factorial * i
        }
        printf("Factorial of %d = %llu," n, factorial);
    }
    getch();
}
```

The output of the program is given in Figure 3.1.

```
Enter an integer: 7
Factorial of 7 = 5040_
```

Figure 3.1: Output of a Factorial number Using Iteration Method.

Program 3.2: A Factorial Program Using Recursion Method

```
#include<stdio.h>
#include<conio.h>
int factorial(int);
void main()
{
    clrscr();
}
```

```

int num;
int result;
printf("Enter a number to find it's Factorial: ");
scanf("%d," &num);
if (num < 0)  {
    printf("Factorial of negative number not possible\n");
} else  {
    result = factorial(num);
    printf("The Factorial of %d is %d.\n", num, result);
    getch();
}
int factorial(int num)  {
    if (num == 0 || num == 1)  {
        return 1;
    } else  {
        return(num * factorial(num - 1));
    }
}

```

The output of the program is given in Figure 3.2.

```

Enter a number to find it's Factorial: 7
The Factorial of 7 is 5040.

```

Figure 3.2: Output of a Factorial number Using Recursion Method.

The concept of Fibonacci Function: One more example of a recursive function is the calculation of Fibonacci numbers. The Fibonacci series is the sequence of integers. Each number in this sequence is the sum of two preceding elements. The series can be formed in this way:

0	1	2	3	4	5	6	7	8	9
0	1	1	2	3	5	8	13	21	34

0^{th} element + 1^{st} element = $0 + 1 = 1$, 1^{st} element + 2^{nd} element = $1 + 1 = 2$, 2^{nd} element + 3^{rd} element = $1 + 2 = 3$ so on.

We can define the recursive definition of the Fibonacci sequence by the recursive function

```
function fibo(int n) {  
    if ((n == 0) || (n == 1)) return 1;  
    else  
        return fibo(N - 1) + fibo(n-2); }
```

Program 3.3: A Fibonacci program

```
#include<stdio.h>  
#include<conio.h>  
  
int fibona(int);  
void main() {  
    clrscr();  
    int num;  
    int result;  
    printf("Enter the nth number in fibonacci series: ");  
    scanf("%d," &num);  
    if (num < 0) {  
        printf("Fibonacci of negative number is not possible.\n"); }  
    else {  
        result = fibona(num);  
        printf("The %d number in fibonacci series is %d\n," num, result); }  
        getch(); }  
  
int fibona(int num) {  
    if (num == 0) {  
        return 0; }  
    else if (num == 1) {  
        return 1; }  
    else {  
        return(fibona(num - 1) + fibona(num - 2)); }  
}
```

The output of the program is given in Figure 3.3.

```
Enter number of terms in Fibonacci series: 8
Fibonacci series till 8 terms
0 1 1 2 3 5 8 13 _
```

Figure 3.3: Output of Fibonacci Series.

Program 3.4: Prime number find using Recursion

```
#include<stdio.h>
#include<conio.h>
int isPrime(int,int);
void main(){
clrscr();
int n,prime;
printf("Enter a positive number: ");
scanf("%d",&n);
prime = isPrime(n, n/2);
if(prime == 1) {
printf("%d is a prime number,"n); }
else {
printf("%d is not a prime number,"n); }
getch(); }
int isPrime(int n,int i){
if(i == 1) {
return 1; }
else{
if(n%i == 0)
return 0;
else
isPrime(n, I - 1); }
}
```

The output of the program is given in Figure 3.4.

```
Enter a positive number: 19
19 is a prime number
```

Figure 3.4: Output for Check Prime number.

3.2.3. Head and Tail Recursions

Within a program if the recursion calls happen at the last of a method, it is called a **tail recursion**. The tail recursion method is similar to a loop. The method executes all the instruction or statements before jumping into the next recursive call. If any program the recursive function calls at the beginning of a method, it is called a **head recursion**. The method saves the state before jumping into the next recursive call. Compare these two recursions:

<pre>public void tail(int a) { if(a == 1) return; else print(a); tail(a-1); }</pre>	<pre>public void head(int a) { if(a == 0) return; else head(a-1); print(a); }</pre>
-----------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------

A program with a single recursive call at the beginning of the path is used are called **head recursion**. A program with a single recursive call at the last or end of a path is using **tail recursion**.

3.2.4. Differences Between Iteration and Recursions

S.N.	Iteration	Recursion
1.	The iterative methods are more efficient because of better execution speed.	The recursive methods are less efficient.
2.	A recursive problem can be solved iteratively.	Not all problems can be solved as recursive.
3.	It is a process of executing a statement or a set of statements until some specified condition is specified.	It is the technique of defining anything in terms of itself.

4.	Memory utilization is less.	Memory utilization is more recursive.
5.	It is simple to implement.	It is complex to implement.
6.	The line of code is more when we use iteration.	Recursive methods bring compactness in the program.
7.	The iteration method is implemented with the help of loops (for, while, do-while) programming.	Instead of making use of loops, the code by calling the same function again and again for some condition.

3.2.5. Advantages and Disadvantages

- We can create a simple and easy version of programs using recursion.
- Always recursion will be written in the name of the recursive definition. It can be translated into programming code very easily.
- Some specific applications are meant for recursion, such as binary tree traversal; tower of Hanoi, etc., can be easily understood.

There is a number of disadvantage using recursion is:

- **It uses more memory:** when the function is called outside or called within, the function stores formal parameters local variables and returns address to confirm function are working well. It stored function variables separately.
- **It will take more time:** after matching the base condition, the function should restore the most recently saved parameters, local variables and return address. These operations spent a lot of time during pushing and pop the necessary items from the stack.
- Function execution is slower than the iterative method because of the overhead of calling functions repeatedly.

3.3. TOWER OF HANOI

The **Tower of Hanoi** (also known as the **Tower of Brahma** or **Lucas' Tower**) is a mathematical game or puzzle. This puzzle was first exposed in the West by the French mathematician Édouard Lucas in 1883. The problem of the “Towers of Hanoi,” it consists of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape as shown in Figure 3.5. The purpose of

the puzzle is to move the entire element to another rod, follow the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- Never any disk placed on top of a smaller disk.

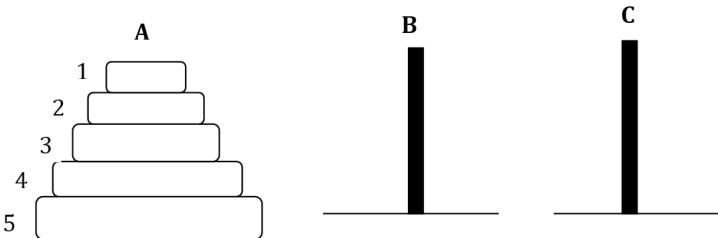


Figure 3.5: Initial Setup of Tower of Hanoi.

The solution to this problem is very simple. The solution can be stated as

1. Move top $N - 1$ disks from A to B using C as an auxiliary.
2. Move the remaining disk from A to C.
3. Move the $N - 1$ disks from B to C using A as an auxiliary.

The above solution steps are a recursive algorithm: to perform steps 1 and 3, used the same algorithm again for $n-1$. The entire procedure is a finite number of steps since at some point the algorithm will be required for $n = 1$. This step, moving a single disc from peg A to peg B, is trivial. We can convert it to:

Step 1. Move disk 1 from A to B, disk 2 from A to C and disk 1 from B to C.

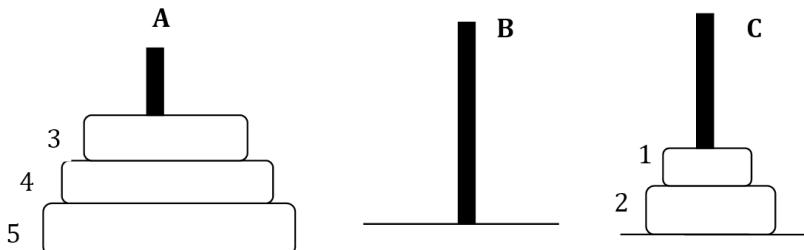


Figure 3.6: After applying first Step position of Tower.

Step 2. Move disk 3 from A to B, Move disk 1 from C to A, Move disk 2 from C to B and Move disk 1 from A to B.

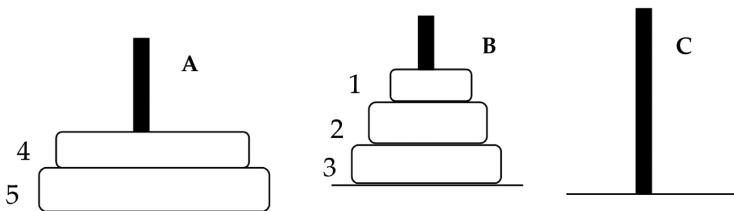


Figure 3.7: After applying second step position of tower.

Step 3. Move disk 4 from A to C, Move disk 1 from B to C, Move disk 2 from B to A. Move disk 1 from C to A and Move disk 3 from B to C.

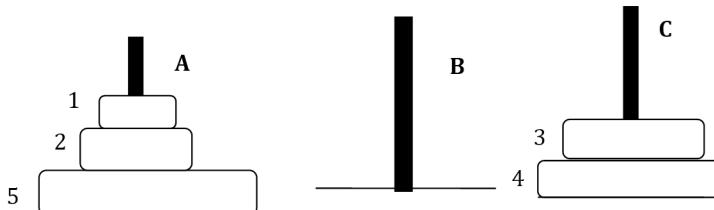


Figure 3.8: After applying third step position of tower.

Step 4. Move disk 1 from A to B, Move disk 2 from A to C and Move disk 1 from B to C.

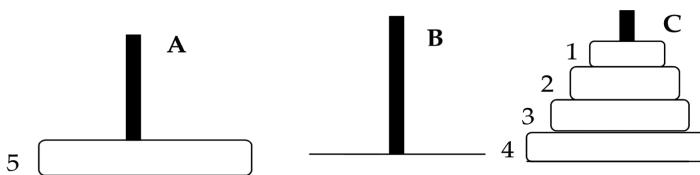


Figure 3.9: After applying fourth step position of tower.

Actually, we have moved $n - 1$ disk from peg A to C. in the same way we can move the remaining disk from A to C.

3.4. BACKTRACKING

Backtracking is a method that used to solve problems like the recursion methods. For solving any problems a great search again and again that

thoroughly tries and eliminates possibilities of the problem solution. The name backtrack was first thought up by D.H. Lehmer in the 1950s. The backtracking approach says to try each case, one after the other, if you ever get fixed, “backtrack” to the connection and try the next choice.

3.4.1. Backtracking Using Recursion

The normal situation is that when we are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated again and again until you reach a final position. If you made a good sequence of choices, your final position is a goal state if you didn't, it isn't. For example, we start from the root of a tree; the tree maybe has some good fruit and some bad fruit, though it may be that the fruit is all good or all bad. We want to get into a good fruit. At each node, beginning with the root, we choose one of its children to move to, and you keep this up until we get to a fruit. Assume that we get to a bad fruit. We can backtrack to continue the hunt for a good fruit by withdrawing your most recent choice, and trying out the next option in that set of options. If you run out of options, cancel the choice that got you here, and tries other choice at that node. If we finish up at the root with no options left to hunt, so there are no good fruits to be found on the tree. This needs an example.

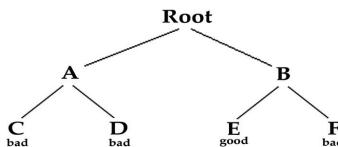


Figure 3.10: Tree Example.

- First Starting at Root, your options are A and B. You choose A.
- At position A, there are two options are C and D. You choose C.
- If C is bad. Go back to A.
- At position A, you have already tried C, and it failed. Try D.
- If D is bad. Go back to A.
- At position A, you have no options left to try. Go back to Root.
- At Root, you have already tried A. Try B.
- At position B, your options are E and F. Try E.
- E is good. Congratulations! You find a good fruit.

The backtracking algorithm is simple but important. You should understand it thoroughly. Another way of stating it is as follows: To search a tree:

- If the tree consists of a single leaf, test whether it is a goal node.
- Otherwise, search the subtrees until you find one containing a goal node, or until you have searched them all unsuccessfully.

4

CHAPTER

STACK

CONTENTS

4.1. Introduction.....	78
4.2. Definition of Stack	78
4.3. Operations on Stack	80
4.4. Disadvantages of Stack	94
4.5. Applications of Stack	94
4.6. Expressions (Polish Notation)	95
4.7. Evaluation of Postfix Expression	98
4.8. Decimal To Binary Conversion.....	99
4.9. Reversing The String.....	102

4.1. INTRODUCTION

The stack concept uses an expression evaluation method. It was first proposed by German computer scientist Mr. F.L. Bauer, who received the IEEE Computer Society Pioneer Award in 1988 for his work on the data structure for the stack. In the data structures, the stack is a linear or non-primitive data that is one of the most valuable concepts for programmers.

4.2. DEFINITION OF STACK

The stack is a linear data structure, in which performed operations like traversing, insertion and deletion of elements are restricted by following definite rules. In computer programming, a stack is a data structure that works on the principle of **Last In First Out** or (**LIFO**). This means that the last item put on the stack is the first item that can be taken out, like a physical stack of bread. The bread can be arranged one on another, when we add new bread it is always placed on the previous bread and while removing the bread the recently placed bread can be removed. It is a structured record in which perform addition (insertion) of the new data item and deletion of an already existing data item is made from only one end, called the **TOP**. Since all the insertion and deletion in a stack are made from the top of the stack; the last inserted item will be the first to be removed from the stack. When an item is added into the stack, we say that use operation **pushes** it onto the stack, and when an item is removed from a stack, we say that operation **pop** it from the stack. For example, if we have to make a stack of elements 20, 40, 60, 80, and 100. Here 20 will be the bottommost element and 100 will be the topmost element in a stack. A stack is shown in Figure 4.1.

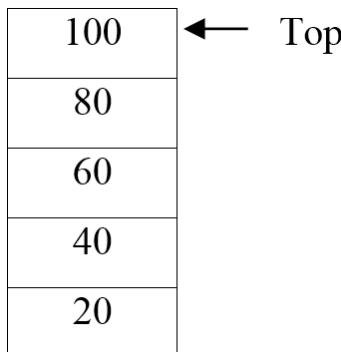


Figure 4.1: Stack Containing Items.

4.2.1. Structure of Stack

When we are implementing or create a stack in a computer programming the allocation of memory is linear. Creation of stack can be either arrays or linked lists. For that reason, we use a sequential representation for implementing a stack. We need to define an array of any maximum size. We need an integer variable top, which will keep track of the top of the stack as more and more elements are inserted into and deleted from the stack. The declarations in C are as follows:

```
# define max size 50
int stack [maxsize];
int top = -1;
```

In the above declaration stack is nothing but an array of integers. And most recent index of that array will act as a top = -1.

Stack Initially top = -1

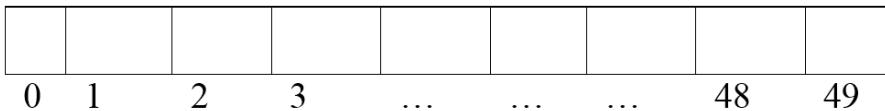


Figure 4.2: Stack using a 1-dimensional array.

The given stack of the maximum size is 50. As add or insert the number, the top position will get incremented. The elements will be placed from 0th posting in the stack. Another adding or inserting a new number in this stack the top position will get incremented again. The elements will be placed from 1st posting in the stack and so on more elements will be inserted. The stack can be used in constructing a database. For example, if we want to store marks of all students of the fourth semester we can declare a structure of stack as follows:

```
# define max size 150
typedef struct student {
    int enrollno;
    char name [30];
    float marks;
    int top = -1;
```

```

} stud;
stud st1 [maxsize];

```

The above stack will look like this database or table:

	Enroll no.	Name	Marks
149			
Top = 3 →	40	Manish	86.8
2	30	Arvind	75.5
1	20	Bhavna	78.3
0	10	Riya	85.6

st1

4.3. OPERATIONS ON STACK

The basic operations that we can perform on the stack are as follows:

- **CREATE** – This operation is used to create an empty stack.
- **PUSH** – This operation is used to perform an inserting or adding a new element in the stack; in the stack this method is called push operation. Every time a new element is inserted into the stack; TOP position will increment by one before the element is placed into the stack.
- **POP** – This operation is used to perform deleting an element from the stack; in the stack, this method is called pop operation. Every time an element is deleted into the stack; TOP position will decrement by one.
- **EMPTY** – This operation is used to perform check whether the stack is empty or not. It returns true if the stack is empty and return false otherwise.
- **TOP** – This operation is used to return the top element of the stack.
- **PEEP** – it is used to extract information stored at some location in a stack.

4.3.1. Stack Empty Operation

When we are representing a stack using array initially array is empty. At that time the top should be initialized to -1 or 0 . If we set top = -1 initially, then the stack will hold the elements from 0^{th} position and if we set top = 0 initially, then the stack will hold the element form 1^{st} position, in the stack. The stack becomes empty whenever top reaches top = -1 . We can say the stack is empty.

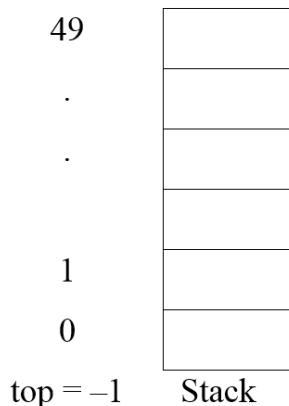


Figure 4.3: Stack empty condition.

```
int stack_empty ()
{
    if (top == -1) // stack is empty
        return 1;
    else
        return 0;
}
```

4.3.2. Stack Full Operation

In the representation of stack using arrays, the size of the array means the size of the stack. When we are going to insert or add the new elements into the stack. It is required to checks before inserting the elements whether the stack is full or not. Stack full achieved when the stack TOP position reaches to a maximum size of the array. If the stack is full cannot insert a new element into the stack, else we can insert the new element.

```

int stack_full()
{
    if (top >= size - 1) // stack is full
        return 1;
    else
        return 0;
}

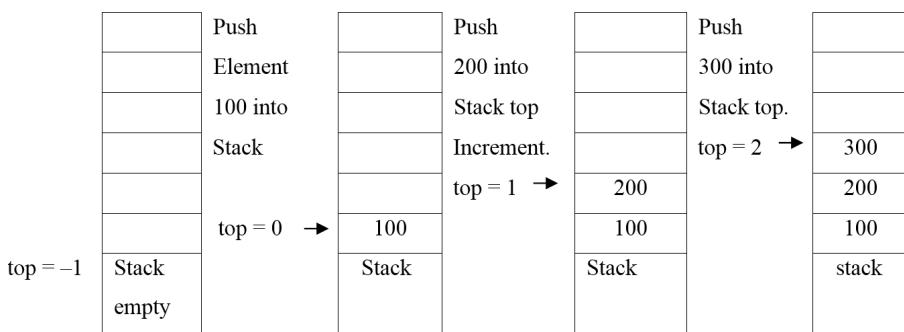
```

top = 49	60
.	50
.	40
1	30
0	20
	10
	stack

Figure 4.4: Stack full condition.

4.3.3. Stack Push Operation

This operation is used to perform an inserting or adding a new element in the stack, in the stack this method is called push operation. Every time a new element is inserted into the stack; TOP position will increment by one before the element is placed into the stack. In the Push first we have to check whether the stack is full or not, if the stack is not full then only the insertion of the element can be achieved by means of the push operation perform otherwise not perform the push operation. Push operation can be shown in Figure 4.5.

**Figure 4.5:** Performing Push Operation.

Algorithm for Push Operation of Stack

The algorithm for the push operation inserts item to the top of a stack, which is represented by S and containing size number of the item, with a pointer TOP denoting the position of the topmost item in the stack.

Step 1. Check for stack is full:

if $\text{TOP} \geq \text{Size} - 1$

Output “Stack is full” and exit

Step 2. Increment the pointer value by one:

$\text{TOP} = \text{TOP} + 1$

Step 3. Perform insertion operation:

$S[\text{TOP}] = \text{item}$

Step 4. Exit

4.3.4. Stack POP Operation

This operation is used to perform deleting an element from the stack; on the stack, this method is called pop operation. Every time an element is deleted into the stack; TOP position will decrement by one. In the selection of pop operation, first, it invokes the function ‘stack_empty’ to find out whether the stack is empty or not. If it is empty, then the function generates an error as stack underflow, If not then pop function return the element which is at the top of the stack. The value at the top is holding in some variable as an item and it then decrements the value of the top position.

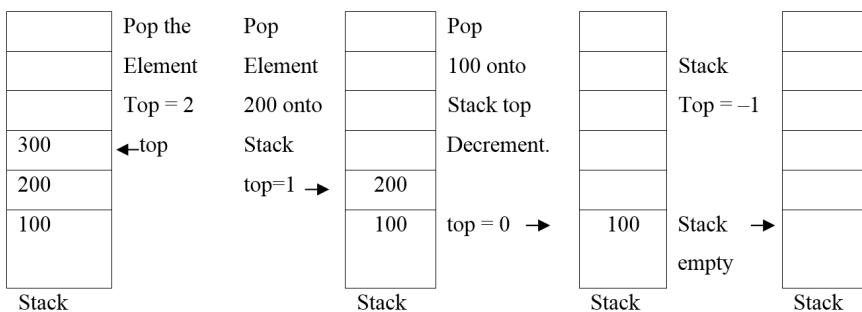


Figure 4.6: Performing Pop Operation.

Algorithm for Pop Operation of Stack

The algorithm for pop operation deletes an item to the top of a stack, which is represented by S and containing size number of the item, with a pointer TOP denoting the position of the topmost item in the stack.

Step 1. Check for stack underflow or empty:

if $\text{TOP} = -1$

and exit
 Output “Stack is Empty”

Step 2. Perform deletion operation:

item = S [TOP]

Step 3. Decrement the pointer value by one:

TOP = TOP – 1

Step 4. Exit.

Program Stack 4.1: Stack Operations Using Array in C.

```
#include<stdio.h>
#include<conio.h>
int stack[100],choice,n,top,x,i;
void push();
void pop();
void display();
void main()
{
clrscr();
top = -1;
printf("\n Enter the size of STACK[MAX = 100]:");
scanf("%d",&n);
printf("\t STACK OPERATIONS USING ARRAY");
printf("\n\t-----");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do {
printf("\n Enter the Choice:");
scanf("%d",&choice);
switch(choice) {
case 1: {
push();
break; }}
```

```
case 2: {  
pop();  
break; }  
case 3: {  
display();  
break; }  
case 4: {  
printf("\n\t EXIT POINT ");  
break; }  
default: {  
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)"); }  
getch(); } }  
while(choice!=4); }  
void push() {  
if(top>= N-1) {  
printf("\n\tSTACK is full");  
getch(); }  
else {  
printf(" Enter a value to be pushed:");  
scanf("%d,"&x);  
top++;  
stack[top] = x; } }  
void pop() {  
if(top<=-1) {  
printf("\n\t Stack is empty"); }  
else {  
printf("\n\t The popped elements is %d,"stack[top]);  
top--; } }  
void display() {  
if(top>= 0) {  
printf("\n The elements in STACK \n");  
}
```

```

for(i = top; i>= 0; i--)
printf("\n%d,"stack[i]);
printf("\n Press Next Choice"); }
else {
printf("\n The STACK is empty"); }
}

```

The output of the program is given in Figure 4.7.

```

Enter the size of STACK[MAX=100]:10
STACK OPERATIONS USING ARRAY
-----
1 .PUSH
2 .POP
3 .DISPLAY
4 .EXIT
Enter the Choice:1
Enter a value to be pushed:12
Enter the Choice:1
Enter a value to be pushed:24
Enter the Choice:3
The elements in STACK
24
12
Press Next Choice
Enter the Choice:4

```

Figure 4.7: Stack Operation.

Example 4.1: Using stack write a program to check string is palindrome or not.

Program: #include <stdio.h>

```

#include <stdlib.h>
#include <string.h>
#include<conio.h>
#define MAX 50
int top = -1, front = 0;
int stack[MAX];

```

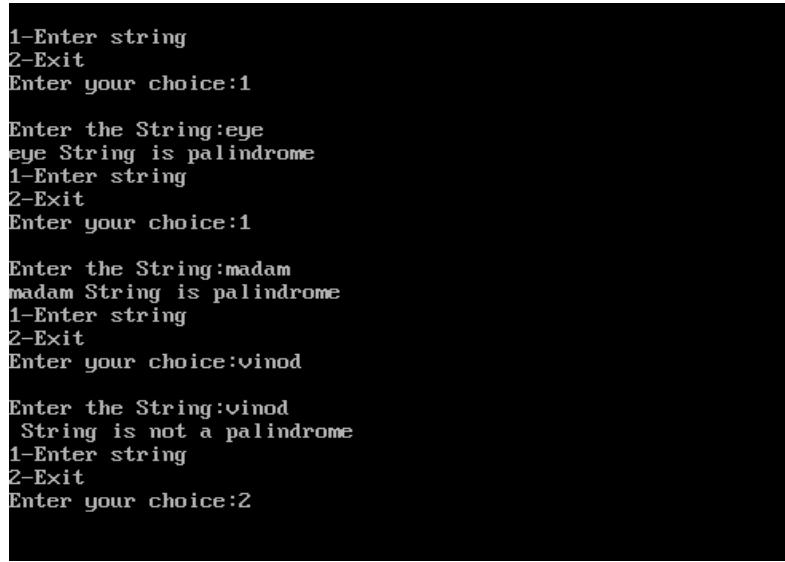
```
void push(char);
void pop();
void main()
{
    clrscr();
    int i, choice;
    char s[MAX], b;
    while (1) {
        printf("\n1-Enter string \n2-Exit");
        printf("\nEnter your choice:");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter the String:");
                scanf("%s", s);
                for (i = 0;s[i] != '\0';i++) {
                    b = s[i];
                    push(b);
                }
                for (i = 0;i < (strlen(s) / 2);i++) {
                    if (stack[top] == stack[front]) {
                        pop();
                        front++;
                    } else {
                        printf("%s \n String is not a palindrome," s);
                        break;
                    }
                }
                if ((strlen(s)/2) == front)
                    printf("%s String is palindrome," s);
                front = 0;
                top = -1;
                break;
            case 2:
```

```
exit(0);
default:
printf("enter correct choice\n");
}
}
}

void push(char a) {
    top++;
    stack[top] = a;
}

void pop()
{
    top--;
}
```

The output of the program is given in Figure 4.8.



A terminal window displaying the execution of a C program. The program asks for user input to check if a string is a palindrome. It provides two choices: 1 for entering a string and 2 for exiting. The user enters '1' for choice, then enters strings 'eye', 'madam', and 'vinod'. For 'eye' and 'madam', the program outputs that the string is a palindrome. For 'vinod', it outputs that the string is not a palindrome. The user exits the program by selecting choice 2.

```
1-Enter string
2-Exit
Enter your choice:1

Enter the String:eye
eye String is palindrome
1-Enter string
2-Exit
Enter your choice:1

Enter the String:madam
madam String is palindrome
1-Enter string
2-Exit
Enter your choice:vinod

Enter the String:vinod
String is not a palindrome
1-Enter string
2-Exit
Enter your choice:2
```

Figure 4.8: Output Example 4.1.

Program 4.2: Stack operation using Linked List

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;
int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();
int count = 0;
void main()
{
    clrscr();
    int no, ch, e;
    printf("\n Stack Operations using Linked List:");
    printf("\n 1 – Push");
    printf("\n 2 – Pop");
    printf("\n 3 – Top");
    printf("\n 4 – Empty");
    printf("\n 5 – Dipslay");
    printf("\n 6 – Stack Count");
    printf("\n 7 – Destroy stack");
    printf("\n 8 -Exit");
```

```
create();
while (1)
{
    printf("\n Enter choice : ");
    scanf("%d," &ch);
    switch (ch)
    {
        case 1:
            printf("Enter data : ");
            scanf("%d," &no);
            push(no);
            break;
        case 2:
            pop();
            break;
        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {
                e = topelement();
                printf("\n Top element : %d," e);
            }
            break;
        case 4:
            empty();
            break;
        case 5:
            display();
            break;
        case 6:
```

```
stack_count();
    break;
case 7:
destroy();
    break;
case 8:
    exit(0);
default :
printf(" Wrong choice, Please enter correct choice ");
break;
}
}
}

/* Create empty stack */

void create()
{
top = NULL;
}

/* Count stack elements */

void stack_count()
{
printf("\n No. of elements in stack : %d," count);
}

/* Push data into stack */

void push(int data)
{
if (top == NULL)
{
top = (struct node *)malloc(1*sizeof(struct node));
top->ptr = NULL;
top->info = data;
```

```
}

else

{

    temp = (struct node *)malloc(1*sizeof(struct node));
    temp->ptr = top;
    temp->info = data;
    top = temp;

}

count++;

}

/* Display stack elements */

void display()

{

    top1 = top;
    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }

    while (top1 != NULL)
    {
        printf("%d ,", top1->info);
        top1 = top1->ptr; } }

/* Pop Operation on stack */

void pop()

{

    top1 = top;
    if (top1 == NULL) {
        printf("\n Error : Trying to pop from empty stack");
        return; }

    else
```

```

    top1 = top1->ptr;
    printf("\n Popped value : %d," top->info);
    free(top);
    top = top1;
    count--;
}
/* Return top element */

int topelement() {
    return(top->info);
}
/* Check if stack is empty
or not */

void empty() {
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements," count);
}
/* Destroy entire stack */

void destroy()
{
    top1 = top;
    while (top1 != NULL) {
        top1 = top->ptr;
        free(top);
        top = top1;
        top1 = top1->ptr;
    }
    free(top1);
    top = NULL;
    printf("\n All stack elements destroyed");
    count = 0;
}

```

The output of the program is given in Figure 4.9.

```
Stack Operations using Linked List:  
1 - Push  
2 - Pop  
3 - Top  
4 - Empty  
5 - Display  
6 - Stack Count  
7 - Destroy stack  
8 -Exit  
Enter choice : 1  
Enter data : 20  
  
Enter choice : 1  
Enter data : 50  
  
Enter choice : 6  
  
No. of elements in stack : 2  
Enter choice : 5  
50 20  
Enter choice : 8_
```

Figure 4.9: Output of stack operation using linked list.

4.4. DISADVANTAGES OF STACK

- The insertion and deletion of element can perform by only one end.
- The element being inserted first has to wait for the longest time to get popped off.
- Only the element at the top can be deleted at a time.

4.5. APPLICATIONS OF STACK

Various applications of the stack are:

- Expression conversion
- Expression evaluation
- Parsing well-formed parenthesis
- Decimal to binary conversion
- Reversing a string
- Storing function calls
- Recursion

- Stack machine
- Memory Management
- String Palindrome

4.6. EXPRESSIONS (POLISH NOTATION)

The technique of representing or writing the operators of an expression either before their operands or after them is called the **polish notation**. The expression is a combination of string operands and operators. Operands are some numeric values or variable and operators. Operators are two types: Unary operators and Binary operators. Unary operators are ‘+’ and ‘-’, Binary operators are ‘+,’ ‘-,’ ‘*,’ ‘/’ and exponential. In general, there are three types of expression representation:

1. **Infix Expression:** The polish notation when the operator is present between two operands then the expression is called infix expression.

Infix expression = operand1 operator operand2

For example: 1. $(X+Y)$ 2. $(X+Y) * (X-Z)$

2. **Prefix Expression:** The polish notation when the operator is present before their operands, then the expression is called prefix expression.

Prefix expression = operator operand1 operand2

For example: 1. $(+XY)$ 2. $* +XY - XZ$

3. **Postfix Expression:** The polish notation when the operator is present after their operands, then the expression is called postfix expression.

Postfix expression = operand1 operand2 operator

For example, 1. $(XY+)$ 2. $XY + XZ - *$

4.6.1. Conversion from Infix Expression to Postfix Expression

Algorithm: Step 1. Read the infix expression from left to right one character at a time.

Step 2. If the input character read as an **operand**, then place it in the postfix expression.

Step 3. If the input character or symbol is **operator** then

- a) Check if the priority of operator in the stack is greater than the priority of incoming operator then pops that operator from the stack and place it in the postfix expression. Repeat step 3(a) until we get the operator in the stack which has greater priority than the incoming operator.
- b) Otherwise, push the operator being read, onto the stack.
- c) If we read the input operator as ')' then pop all the operators until we get '(' and append popped operators to postfix expression. Finally, pop '('.

Step 4. Finally pop the remaining contents, from the stack until stack becomes empty append them to postfix expression.

Step 5. Print the postfix expression as an answer.

Without using this algorithm convert of the given expression into postfix expression.

Infix Notation: $A - B - (C * D - F / G) * E$

$$\begin{aligned}
 &= A - B - (C * D - FG/) * E \\
 &= A - B - (C D^* - FG/) * E \\
 &= A - B - (C D^* FG/-) * E \\
 &= A - B - C D^* FG/- E^* \\
 &= A - B C D^* FG/- E^* -
 \end{aligned}$$

$= A B C D^* F G / - E ^* - -$ // postfix expression of given infix expression.

Using the given algorithm converts the given infix expression to postfix expression.

Input character read	Stack	Postfix
A	Empty	A
-	-	A
B	-	AB
-	--	AB
(--(AB
C	--(ABC
*	--(*)	ABC
D	--(*)	ABCD

-	-- (-	ABCD*
F	-- (-	ABCD*F
/	-- (- /	ABCD*F
G	-- (- /	ABCD*FG
)	--	ABCD*FG/-
*	-- *	ABCD*FG/-
E	-- *	ABCD*FG/-E
)	Empty	ABCD*FG/- * --

4.6.2. Conversion from Infix Expression to Prefix Expression

Algorithm: Step 1. First, reverse the given infix expression.

Step 2. Start reading this reversed expression from left to right one character at a time.

Step 3. If input symbol being read as operand then place it in prefix expression.

Step 4. If input symbol read is operand then:

- a) Check if the priority of operator in the stack is greater than the priority of incoming (or input) operator from the stack and place it in prefix expression. Repeat step 4(a) until we get the operator in the stack which has greater priority than the incoming operator.
- b) Otherwise, push the operator being read.
- c) If we read '(' as input symbol then pop all the operators until we get ')' and append popped operation to prefix expression.

Step 5. Finally, pop the remaining contents and append them to prefix expression.

Step 6. Reverse the obtained prefix expression and print it as result.

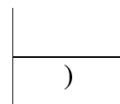
Example: Convert the infix expression $(a + b) * (c - d)$ into equivalent prefix form.

Step 1. $(a + b) * (c - d)$ must be reversed first.

So we get $) d - c - (*) b + a$ (Now we will read each character from left to right one at a time.

) d - c - (*) b + a (

↑



operator) is read push it onto the stack

Input character read	Stack	Prefix
))	
D)	D
-) -	D
C) -	Dc
(pop all content until get)	dc -
*	*	dc -

)	*)	dc -
B	*)	dc - b
+	*) +	dc - b
A	*) +	dc - <u>ba</u>
(pop all content until get)	dc - <u>ba</u> + *

Now reverse the prefix expression. It will * + ab - cd. Print it as a result.

4.7. EVALUATION OF POSTFIX EXPRESSION

Algorithm: 1. Read the postfix expression from left to right one character at a time.

2. If we read operand then push it onto the stack.

3. If we read the operator then pop two operands call first popped operand as OP2 and second popped operand as OP1. Perform arithmetic operation if the operator is

- + Then result = OP1 + OP2
- Then result = OP1 - OP2

- * Then result = OP1 * OP2
- / Then result = OP1 / OP2
- ↑ Then result = OP1 ↑ OP2 so on.
- 4. Push the result onto the stack.
- 5. Repeat steps 1–4 till the postfix expression is not over.

For example: The postfix expression is: 3 2 ↑ 5 * 3 2 * 3 - / 5 +

Input symbol	OP1	OP2	Result	Stack
3	3			3
2		2		3,2
↑	3	2	9	9
5				9,5
*	9	5	45	45
3				45,3
2				45,3,2
*	3	2	6	45,6
3				45,6,3
-	6	3	3	45,3
/	45	3	15	15
5				15,5
+	15	5	20	20

The result of this postfix expression is 20.

4.8. DECIMAL TO BINARY CONVERSION

Let us take some decimal number as 8. Now its binary equivalent can be obtained by using the stack. What we can do is that just go on dividing that number by 2, whatever is the remainder stores it onto the stack. Finally, pop the element from the stack and print it.

Divided by	Number	Remainder	Stack
2	8	0	1
2	4	0	0
2	2	0	0
2	1	1	0

Now stack is

Program 4.3: Decimal to Binary Conversion using Stack

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
int stack[SIZE];
int top = -1;
int empty();
int full();
void push(int x);
void pop();
void display();
void main()
{
clrscr();
int n,remainder;
printf("Enter a decimal number:");
scanf("%d",&n);
while(n!=0){
remainder = n%2;
n = n/2;
push(remainder); }
printf("Binary Number is:");
display();
getch(); }
int empty(){
```

```
if(top == -1){  
    return 1; }  
else {  
    return 0; } }  
int full(){  
if(top == SIZE){  
    return 1; }  
else {  
    return 0; } }  
void push(int x){  
if(full()){  
    printf("Stack overflow!!\n"); }  
else {  
    top++;  
    stack[top] = x; } }  
void pop(){  
if(empty()){  
    printf("Stack empty"); }  
else {  
    top--; } }  
void display(){  
int i;  
for(i = top;i>= 0;i--){  
    printf("%d , "stack[i]); } }
```

The output of the program is given Figure 4.10.

```
Enter a decimal number:26  
Binary Number is:1 1 0 1 0
```

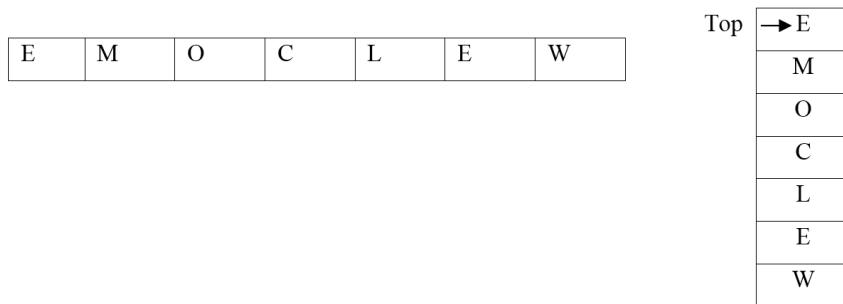
Figure 4.10: Output of Decimal to Binary.

4.9. REVERSING THE STRING

The simple mechanism is to push all the characters of a string onto the stack and then pop all the characters from the stack and print them. For example, if the input string is

W	E	L	C	O	M	E	\0
---	---	---	---	---	---	---	----

Then push all the characters onto the stack till ‘\0’ is encountered. Now if we pop each character from the stack and print it we get, reversed string.



Program 4.4: Reverse a String using Stack

```
#include <stdio.h>
#include <string.h>
#define MAX 20
#include<conio.h>

int top = -1;
int item;
char stack_string[MAX];
void pushChar(char item);
char popChar(void);
int isEmpty(void);
int isFull(void);
void main()
{
    clrscr();
    char str[MAX];
```

```
int i;
printf("Input a string: ");
scanf("%[^\\n]s,"str); //read string with spaces
for(i = 0;i<strlen(str);i++)
pushChar(str[i]);
for(i = 0;i<strlen(str);i++)
str[i] = popChar();
printf("Reversed String is: %s\\n,"str);
getch();
}
void pushChar(char item) {
if(isFull())
printf("\nStack is FULL !!!\\n");
return;
}
top = top+1;
stack_string[top] = item;
char popChar() {
ifisEmpty()
printf("\nStack is EMPTY!!!\\n");
return 0;
}
item = stack_string[top];
top = top-1;
return item;
}
int isEmpty()
{
if(top == -1)
return 1;
else
return 0;
}
int isFull()
```

```
{  
if(top == MAX-1)  
return 1;  
else  
return 0;  
}
```

The output of the program is given in Figure 4.11.

```
Input a string: vinod kumar  
Reversed String is: ramuk doniv
```

Figure 4.11: Output Reverse a String.

5

CHAPTER

QUEUE

CONTENTS

5.1. Introduction.....	106
5.2. Definition And Structure of Queue.....	106
5.3. Operations on Queue.....	106
5.4. Circular Queue.....	114
5.5. D-Queue (Double Ended Queue)	119
5.6. Priority Queues.....	124
5.7. Application of Queue	128

5.1. INTRODUCTION

In computer programming, a queue is a data structure that works on the principle of **first in, first out (FIFO)** concept. We describe another type of linear data structure is Queue, definition, algorithms, operations like insertion, deletion, etc., circular queue, D-queue, priority queue and applications of Queue, with the help of programs.

5.2. DEFINITION AND STRUCTURE OF QUEUE

A queue is a linear data structure in which insertion or addition of elements is made only at one end of the list and other end of the queue you can remove or delete the elements. The queue can be properly defined as an ordered collection of elements that has two ends named as **front** and **rear**. Form the **front** end, we can delete or remove the elements and from the **rear** end, one can insert the elements.

For example: In a Bank, when we go for the withdraw money from own account they provide a token number; basically, it makes a list of the numbers or names of customers. Adding each new token number at the front of the list and crossing the top token number. The list as a customer is called the structure of a queue. The word ‘queue’ is also used in lots of everyday examples. Following Figure 5.1 represents the queue of the few elements.

100	300	350	210	550	110	610
Front			Rear			

Figure 5.1: Queue structure.

A queue is a linear list where additions and deletions may take place at either end of the list, but never in the middle.

5.3. OPERATIONS ON QUEUE

The various operations on the queue are:

- **CREATE** – This operation is used to create an empty queue.
- **OVERFLOW**: Check whether a Queue is full or Queue overflow.
- **REAR**: Insertion or addition of the element into the Queue (at the rear).

- **UNDERFLOW:** Check whether Queue is empty or Queue underflow.
- **FRONT:** Remove or deletion of the element into the Queue (at the front).
- **TRAVERSE:** Display (print) of the Queue elements.

5.3.1. Static or Array Implementation of Queue

If the queue is implemented static means using arrays, we must be definite about the exact number of elements to be stored in the queue. A queue has two pointers in **front** and **rear**, pointing to the front and rear elements of the queue, respectively.

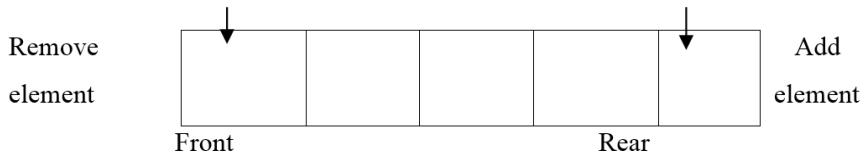


Figure 5.2: Array Implementation of Queue.

In this case, the beginning of the array will become the front of the queue and the last location of the array will act as the rear of the queue. The total number of elements present in the queue is calculated by,

$$\text{Total element in Queue} = (\text{Rear} + 1) - \text{Front}$$

We have to calculate the total elements present in the queue as shown in figure 5.3 (b). Here the front position is 1 and rear position is 4.

$$\text{Total element in Queue} = (4 + 1) - 1 = 5 = 4$$

Consider a queue that max size = 10, there only 10 elements can store in the queue. At present as shown in fig. 5.3 (a) it stores 5 elements. when we remove an element from Queue, we get the resulting Queue as shown in fig. 5.3 (b) and when we insert an element in Queue we get the resulting Queue as shown in fig 5.3 (c). When an element is removed from the Queue the value of the front pointer is increased by 1. **Front = Front +1**. Similarly, when an element is added to Queue the value of the rear pointer is increased by **Rear = Rear +1**. If **Rear < Front** then there will be no element in the queue, so the queue is empty.

0	1	2	3	4	5	6	7	8	9
111	113	115	117	119					

Front Rear

Figure 5.3 (a): Queue in Memory.

0	1	2	3	4	5	6	7	8	9
	113	115	117	119					

Front Rear

Figure 5.3 (b): Queue after deleting the first element.

0	1	2	3	4	5	6	7	8	9
	113	115	117	119	121				

Front Rear

Figure 5.3 (c): Queue after inserting an element.

5.3.2. Algorithms for Operation on Queue

Let queue be an array of size MAXSIZE, then the insertion and deletion algorithms are: **Algorithm for insertion in a Queue**

a) If $\text{Rear} >= \text{MAXSIZE}$

Output “overflow” and return

else

Set $\text{Rear} = \text{Rear} + 1$

b) $\text{Queue}[\text{rear}] = \text{item}$ // insert an item

c) If $\text{Front} = -1$ // set the front pointer

Then $\text{Front} = 0$

d) Return

1. Algorithm for deletion in a Queue

a) If ($\text{front} < 0$)

Output “underflow” and return

b) $\text{Item} = \text{Queue}[\text{front}]$ // remove an item

c) If ($\text{Front} = \text{rear}$) // set the front pointer

Then $\text{Front} = 0$

$\text{Rear} = -1$

Else

Front = front +1

d) Return

Program 5.1: Queue operation implemented using array in C.

```
#include<stdio.h>
#include<conio.h>
int queue[100],choice,n;
int rear = -1;
int front = -1;
void insert();
void remove();
void display();
void main()
{
    clrscr();
    printf("\n Enter the size of Queue[MAX = 100]:");
    scanf("%d",&n);
    printf("\n Operation on Queue using Array");
    printf("\n 1.Insert element to queue");
    printf("\n 2.Delete element from queue ");
    printf("\n 3.Display all elements of queue");
    printf("\n 4.Exit ");
    do
    {
        printf("\nEnter your choice :");
        scanf("%d",&choice);
        switch (choice)
        {
            case 1: {
                insert();
                break; }
            case 2:{ remove();
                break; }
```

```
case 3: {
    display();
    break; }

case 4: {
    printf("\n\t Exit point");
    break; }

default: {
    printf("\n\t Wrong choice a valid choice (1/2/3/4)");
}

getch(); }

}

while (choice != 4); }

void insert()
{
    int add_item;
    if (rear == N - 1) {
        printf("\n Queue is full ");
        getch(); }

    else {
        if (front == -1)
            front = 0;
        printf("Inset the element in queue : ");
        scanf("%d," &add_item);
        rear = rear + 1;
        queue[rear] = add_item; }

    void remove()
    {
        if (front == -1 || front > rear) {
            printf("Queue Underflow \n");
            return ; }

        else {
            printf("Element deleted from queue is : %d\n," queue[front]); }
```

```

        front = front + 1; } }

void display()
{
    int i;
    if (front == -1)
        printf("Queue is empty \n");
    else {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d , " queue[i]);
        printf("\n");
    }
}

```

The output of the program is given in Figure 5.4.

```

Enter the size of Queue[MAX=100]:10
Operation on Queue using Array
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Exit
Enter your choice :1
Inset the element in queue : 66
Enter your choice :1
Inset the element in queue : 55
Enter your choice :1
Inset the element in queue : 99
Enter your choice :3
Queue is :
66 55 99
Enter your choice :2
Element deleted from queue is : 66
Enter your choice :4

```

Figure 5.4: Output of Queue Operations.

Program 5.2: Queue operation implemented using Linked list in C.

```

#include<stdio.h>
#include<conio.h>
struct Node

```

```
{  
    int data;  
    struct Node *next;  
}*front = NULL,*rear = NULL;  
void insert(int);  
void delete();  
void display();  
void main()  
{  
    int choice, value;  
    clrscr();  
    printf("\n:: Queue Implementation using Linked List ::\n");  
    while(1){  
        printf("\n***** MENU *****\n");  
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");  
        printf("Enter your choice: ");  
        scanf("%d",&choice);  
        switch(choice){  
            case 1: printf("Enter the value to be insert: ");  
                scanf("%d," &value);  
                insert(value);  
                break;  
            case 2: delete(); break;  
            case 3: display(); break;  
            case 4: exit(0);  
            default: printf("\nWrong selection!!! Please try again!!!\n");  
        }  
    }  
}  
void insert(int value)
```

```
{  
    struct Node *newNode;  
    newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = value;  
    newNode -> next = NULL;  
    if(front == NULL)  
        front = rear = newNode;  
    else {  
        rear -> next = newNode;  
        rear = newNode;    }  
    printf("\nInsertion is Success!!!\n");  
}  
void delete()  
{  
    if(front == NULL)  
        printf("\nQueue is Empty!!!\n");  
    else{  
        struct Node *temp = front;  
        front = front -> next;  
        printf("\nDeleted element: %d\n," temp->data);  
        free(temp);  
    }  
}  
void display()  
{  
    if(front == NULL)  
        printf("\nQueue is Empty!!!\n");  
    else  
{  
        struct Node *temp = front;  
        while(temp->next != NULL){
```

```

        printf("%d--->,"temp->data);
        temp = temp -> next;  }
printf("%d--->NULL\n,"temp->data);
}
}

```

The output of the program is given in Figure 5.5.

```

:: Queue Implementation using Linked List ::

***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 88
Insertion is Success!!!
Enter your choice: 1
Enter the value to be insert: 44
Insertion is Success!!!
Enter your choice: 1
Enter the value to be insert: 22
Insertion is Success!!!
Enter your choice: 3
88--->44--->22--->NULL
Enter your choice: 2

Deleted element: 88
Enter your choice: 3
44--->22--->NULL
Enter your choice: 4_

```

Figure 5.5: Output of Queue operation using Linked List.

5.4. CIRCULAR QUEUE

As we have seen previous example of linear Queue, the elements get deleted logically when we perform the delete operation. This can be shown by following Figure 5.6.

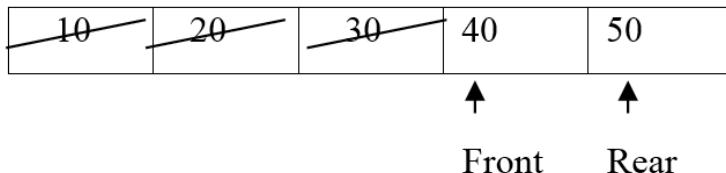


Figure 5.6: Linear Queue.

When we have deleted the elements 10, 20 and 30 means simply the front pointer is shifted ahead. The end will consider a queue from front to rear always. And now if we try to insert any more elements, then it won't be possible as it is going to give "queue full" message. Although there is a space of elements 10, 20 and 30 (these are deleted elements), we cannot utilize them because the queue is nothing but a linear array. Hence there is a concept called a circular queue. The main advantage of the circular queue is we can utilize the space of the queue fully.

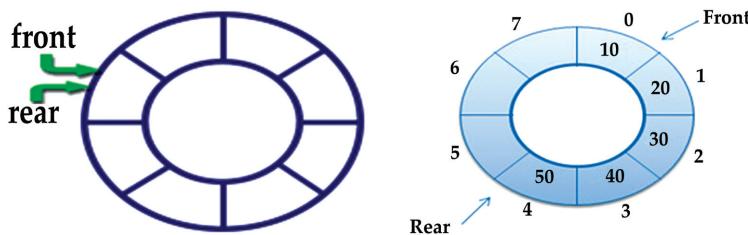


Figure 5.7: Circular Queue.

A circular queue has a front and rear to keep the track of the elements to be deleted and inserted. The following assumption is made –

- A Front will always be pointing to the first element.
- In front = rear, the queue is empty.
- When a new element is inserted into the queue the rear is incremented by one.

$$\text{Rear} = \text{Rear} + 1$$

- When an element is deleted from the queue the front is incremented by one.

$$\text{Front} = \text{Front} + 1$$

Insertion in the circular queue will be the same as with linear queue, but it is required to keep track the front and rear with some extra logic. If a new element is to be inserted in the queue, the position of the element to insert will be calculated by relation.

$$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$$

If add an element 30 into the queue. The rear is calculated as follows –

$$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$$

$$= (2 + 1) \% 5 = 3$$

The delete method for a circular queue also requires some modification as compared to linear queue. The position of the front will be calculated by the relation

$$\text{Front} = (\text{Front} + 1) \% \text{MAXSIZE}$$

5.4.1. Algorithms for operation on Circular Queue

Let queue be the array of size MAXSIZE, then the insertion and deletion algorithms are as follows:

1. Algorithm for insertion in a Circular Queue

a) If ($\text{front} == ((\text{Rear} + 1) \% \text{MAXSIZE})$)

Output “overflow” and exit

else

take the value

b) If ($\text{front} == -1$)

Set $\text{front} = \text{rear} = 0;$

$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$

c) Queue [$\text{rear}] = \text{value}$

End if

d) exit

2. Algorithm for deletion in a Circular Queue

a) If ($\text{front} == -1$)

Output “underflow” and return

b) Item = Queue [front] // remove an item

c) If ($\text{Front} == \text{rear}$) // set the front pointer

Then $\text{Front} = -1$

$\text{Rear} = -1$

Else

$\text{Front} = (\text{Front} + 1) \% \text{MAXSIZE}$

d) Exit

Program 5.3: Circular Queue operation implemented using C.

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

```
#include<stdlib.h>
#include<conio.h>
#define max 3
int q[10],front = 0,rear = -1;
void main()  {
    int ch;
    void insert();
    void remove();
    void display();
    clrscr();
    printf("\nCircular Queue operations\n");
    printf("1.insert\n2.delete\n3.display\n4.exit\n");
    while(1)
    {   printf("Enter your choice:");
        scanf("%d",&ch);
        switch(ch) {
            case 1: insert();
                break;
            case 2: remove();
                break;
            case 3:display();
                break;
            case 4:exit(0);
            default: printf("Invalid option\n");  }  }
void insert()  {
    int x;
    if((front == 0 && rear == max-1)||(front>0 && rear == front-1))
        printf("Queue is overflow\n");
    else {  printf("Enter element to be insert:");
        scanf("%d",&x);
        if(rear == max-1 && front>0)  {
```

```
    rear = 0;
    q[rear] = x;    }
else  {
    if((front == 0&&rear == -1)||(rear != front-1))
        q[++rear] = x; } } }

void remove() {
int a;
if((front == 0)&&(rear == -1)) {
    printf("Queue is underflow\n");
    getch();
    exit(0); }

if(front == rear) {
    a = q[front];
    rear = -1;
    front = 0; }

else
    if(front == max-1) {
        a = q[front];
        front = 0; }

    else a = q[front++];
    printf("Deleted element is:%d\n,"a); }

void display() {
int i,j;
if(front == 0&&rear == -1) {
    printf("Queue is underflow\n");
    getch();
    exit(0); }

if(front>rear) {
    for(i = 0;i<= rear;i++)
        printf("\t%d,"q[i]);
    for(j = front;j<= max-1;j++)
        printf("\t%d,"q[j]); }
```

```

        printf("\t%d,"q[j]);
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]);    }
else   {
    for(i = front;i<= rear;i++)    {
        printf("\t%d,"q[i]);    }
    printf("\nrear is at %d\n",q[rear]);
    printf("\nfront is at %d\n",q[front]);    }
    printf("\n");
}
}

```

The output of the program is given in Figure 5.8.

```

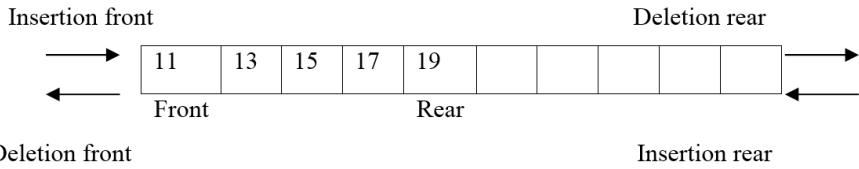
Circular Queue operations
1.insert
2.delete
3.display
4.exit
Enter your choice:1
Enter element to be insert:20
Enter your choice:1
Enter element to be insert:30
Enter your choice:1
Enter element to be insert:40
Enter your choice:1
Queue is overflow
Enter your choice:2
Deleted element is:20
Enter your choice:1
Enter element to be insert:60
Enter your choice:1
Queue is overflow
Enter your choice:2
Deleted element is:30
Enter your choice:1
Enter element to be insert:80
Enter your choice:

```

Figure 5.8: Output of Circular Queue Operations.

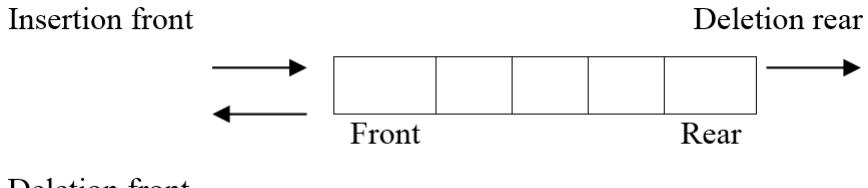
5.5 D-QUEUE (DOUBLE ENDED QUEUE)

In linear queue, insertion of elements we use one end called rear and for the deletion of elements, we use other end called as a front. But in the double-ended queue or D-queue insertion and deletion operations are performed from both the ends. That means it is possible to insert the elements by rear as well as by the front. Similarly, it is possible to delete the elements from the front as well as from the rear.

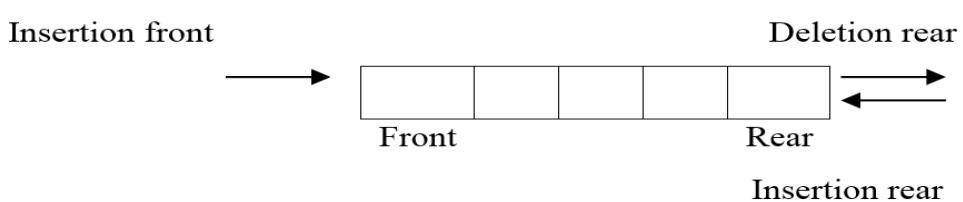
**Figure 5.9:** D-Queue.

There exist two variations of D-Queue:

- 1. Input restricted D-Queue:** Input-restricted D-queue allows insertion of an element at only one end, but it allows the deletion of an element at both of the ends.

**Figure 5.10 (a):** Input-restricted D-queue.

- 2. Output-restricted D-queue:** Output-restricted D-queue allows deletion of the element at only one end, but it allows the insertion of an element at both the ends.

**Figure 5.10 (b):** Output-restricted D-queue.

In D-queue, if the element is inserted from the “front-end” then “front” is decreased by 1 if it is inserted at the “rear-end” then rear is increased by 1. If the element is deleted from the front-end, then the front is increased by 1, if the element is deleted from the rear-end, then the rear is decreased by 1 when a front is equal to rear before deletion then front and rear are both set to NULL to indicate that the queue is empty.

Operation:

- Create (): the D-queue is created by declaring the data structure for it.
- Insert_rear (): This is used for inserting the element from the rear end.
- Delete_front (): This is used for deleting the element from the front end.
- Insert_front (): This is used for inserting the element from the front end.
- Delete_rear (): This is used for deleting the element from the rear end.
- Display (): The elements of the queue can display from front to rear.

Program 5.4: D-Queue operation implemented using C.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int max = 5;
int deque[5];
int front = -1, rear = -1;
void display();
void insertfront();
void insertrear();
void deletefront();
void deletrear();
int choice,item;
void main()  {
    clrscr();
    while(1)  {
        printf("\n 1.Insert Front. 2. Insert Rear. 3.Delete front 4.Delete Rear 5.
Display 6.Exit");
        printf("\n Enter Your Choice:");
    }
}
```

```
scanf("%d,"&choice);
switch(choice) {
    case 1:
        insertfront();
        getch();
        break;
    case 2:
        insertrear();
        getch();
        break;
    case 3:
        deletefront();
        getch();
        break;
    case 4:
        deleterear();
        getch();
        break;
    case 5:
        display();
        getch();
        break;
    case 6:
        exit(0);
    default:
        printf("\n Wrong Choice Try Again ");
        getch();
        break; } } }
```

```
void insertfront() {
    if(front == 0) {
        printf("\n Queue is FULL"); }
```

```
else
    front = front-1;
    printf("Enter a number:");
    scanf("%d",&item);
    deque[front] = item; }

void insertrear()
{
    if(rear == max) {
        printf("\n Queue is FULL");
    }
    else
        rear = rear+1;
    printf(" Enter a number:");
    scanf("%d",&item);
    deque[rear] = item; }

void deletefront()
{
    if(front == max) {
        printf("\n Queue is EMPTY");
    }
    else
        item = deque[front];
        front = front+1;
    printf("\n No. deleted is %d",item); }

void deleterear()
{
    if(rear == 0) {
        printf("\n Queue is EMPTY");
    }
    else
        item = deque[rear];
        rear = rear-1;
    printf("\n No. deleted is %d",item); }

void display()
{
    int i;
```

```

printf("\n The Queue data is:");
for(i = front;i<= rear;i++)
{
    printf("%d \n , deque[i]);  }
}

```

The output of the program is given in Figure 5.11.

```

1. Insert Front. 2. Insert Rear. 3.Delete front 4.Delete Rear 5. Display 6.Exit
Enter Your Choice:5

The Queue data is:54
44
34
24
5
64

1. Insert Front. 2. Insert Rear. 3.Delete front 4.Delete Rear 5. Display 6.Exit
Enter Your Choice:3

No. deleted is 54
1. Insert Front. 2. Insert Rear. 3.Delete front 4.Delete Rear 5. Display 6.Exit
Enter Your Choice:5

The Queue data is:44
34
24
5
64

1. Insert Front. 2. Insert Rear. 3.Delete front 4.Delete Rear 5. Display 6.Exit
Enter Your Choice:

```

Figure 5.11: Output of D-Queue Operations.

5.6. PRIORITY QUEUES

The priority queue is a data structure having a collection of elements which are associated with specific ordering. There are two types of priority queues-

- **Ascending priority queue** – it is a collection of items in which the items can be inserted arbitrarily but the only smallest element can be removed.
- **Descending priority queue** – it is a collection of items in which the items can be inserted arbitrarily, but the only largest element can be removed. In a priority queue, the elements are arranged in any order an out of which only the smallest or largest element allowed to delete each time.

For Priority Queue: Various operations that can be performed on priority queue are-

Operations:

- Create () – The queue is created by declaring the data structure for it.
- Insert () – The element can be inserted in the queue.
- Delete () – if the priority queue is ascending priority queue then the only smallest element is deleted each time.
- Display () – The elements of the queue are displayed from front to rear.

Application of Priority Queue

- In network communication, to manage limited bandwidth for transmission the priority queue is used.
- In simulation modeling, to manage the discrete events the priority queue is used.

Program 5.5: Priority Queue implementation in C

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
#define size 5
int queue[5][2] = {0};
int top = -1;
int bottom;
void push(int value, int pr)  {
    int i,j,k;
    if(top < size-1)      {
        if(queue[top][1] > pr)  {
            for(i = 0;i<top;i++)  {
                if(queue[i][1] > pr)  {
                    break; } }
            for(j = top;j>= i;j--)  {
                queue[j+1][0] = queue[j][0];
                queue[j+1][1] = queue[j][1]; }
            top++;
        }
    }
}
```

```
queue[i][0] = value;
queue[i][1] = pr;    }
else    {
top++;
queue[top][0] = value;
queue[top][1] = pr;    }    }
else    {
printf("queue overflow \n");    }    }
void pop() {
int i;
if(queue[0][0] == 0)    {
printf("\n The queue is empty \n");    }
else    {
printf("After , dequeue the following value is erased \n %d \n," queue[0]
[0]);
for(i = 0;i<top;i++)    {
queue[i][0] = queue[I + 1][0];
queue[i][1] = queue[I + 1][1];    }
queue[top][0] = 0;
queue[top][1] = 0;
top--;    }    }
void display()
{ int i,j;
printf("Element\tPriority \n");
for(i = size - 1;i>= 0;i--)  {
for(j = 0;j<2;j++)    {
printf(" %d\t,"queue[i][j]);    }
```

```
printf("\n");      }  }
int main()      {
clrscr();
int i,j, ch = 0 ,value = 0,pr = 0;
while(1)      {
printf("Please Enter the choice. \n");
printf("1. for Enqueue 2. for Dequeue 3. for display 4. for exit: \n");
scanf("%d,"&ch);
switch(ch)      {
case 1:
printf("\n Please Enter the number to be inserted: ");
scanf("%d," &value);
printf("\n Please Enter the priority: ");
scanf("%d," &pr);
push(value,pr);
break;
case 2:
pop();
break;
case 3:
display();
break;
case 4:
exit(0);
default:
printf("You entered wrong choice\n");  }  }
}
```

The output of the program is given in Figure 5.12.

```

1. for Enqueue 2. for Dequeue 3. for display 4. for exit:
1

Please Enter the number to be inserted: 100

Please Enter the priority: 2
Please Enter the choice.
1. for Enqueue 2. for Dequeue 3. for display 4. for exit:
1

Please Enter the number to be inserted: 50

Please Enter the priority: 1
Please Enter the choice.
1. for Enqueue 2. for Dequeue 3. for display 4. for exit:
3
Element Priority
0      0
0      0
0      0
100    2
50     1
Please Enter the choice.
1. for Enqueue 2. for Dequeue 3. for display 4. for exit:

```

Figure 5.12: Output of Priority Queue.

5.7. APPLICATION OF QUEUE

Typical uses of queues are in the simulations and operating systems.

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Computer systems must often provide a “holding area” for messages between two processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.
- Destination queues – any queue that the sending application sends messages to or that the receiving application reads messages from.
- Administration queues – queues used for acknowledgment messages returned by Message Queuing or connector applications.
- Response queues – queues used by receiving applications to return response messages to the sending application.
- Report queues – queues used to store report messages returned by Message Queuing.

Our software queues have counterparts in real-world queues. We wait in queues to buy pizza, to enter movie theaters, to drive on a turnpike, and to ride on a roller coaster. Another important application of the queue data structure is to help us simulate and analyze such real-world queues.

6

CHAPTER

LINKED LIST

CONTENTS

6.1. Introduction.....	130
6.2. Definition And Structure of Linked List.....	130
6.3. Characteristics of Linked List.....	134
6.4. Types of Linked List.....	135
6.5 Polynomial Representation of Linked List.....	158

6.1. INTRODUCTION

The concept of the Linked list was given in 1955–56 by Allen Newell, Cliff Shaw and Herbert Simon at RAND Corporation as the primary data structure for their Information Processing Language. IPL was used by the authors to develop several early artificial intelligence programs, including the Logic Theory Machine, General Problem Solver, and a computer chess program. Fixed or Static memory allocation is done by arrays. In arrays, the elements are stored one after the other. The elements can be accessed sequentially as well as randomly when we use arrays. But there are some drawbacks or restrictions of arrays. **First**, once the elements are stored serially, it becomes very difficult to insert the element in between or to delete the middle elements. This is because, if we insert some element in between then we have to shift down the adjacent elements. Similarly, if we delete some element from an array, then a vacant space gets created in the array. And we do not desire such vacant space in between for the arrays. The ultimate result is that the use of the array makes the overall representation time and space inefficient. **Second**, before the use of the array in the program, we require to determine the array size earlier. There are some chances that earlier decided the size of the array might be larger than a requirement. Similarly, it might be possible that earlier define the size of the array may be less than they require one. This results in either wastage of memory or shortage of memory. So that another data structure concept has come up that is known as a linked list.

6.2. DEFINITION AND STRUCTURE OF LINKED LIST

A linear list is an ordered set consisting of a variable number of elements to which addition and deletion can be made. The first element of a list is called **the head** and the last element is called the **tail** of the list. The next element to the head of the list is called its **Successor**. The previous element to the tail of the list is called its **Predecessor**.

Definition: A linked list is a linear data structure. It consists of a set of nodes where each node has two fields: an information or data and link or next address field. The ‘data’ field stores the actual part of the information, which may be an integer, character, string and ‘link’ field is used to point to the address of the next node. The entire linked list is accessed from an external pointer that’s pointing to the very first node in the list. Basically,

'link' field is nothing but address only. The last node contains NULL that indicates the end of the Link list.

Data or Information	Link or Next
------------------------	-----------------

Figure 6.1: Structure of a Node.

Example: Link list of integer 120,140, 160, 180 is,

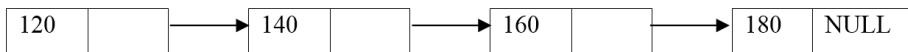


Figure 6.2: Representation of link list.

6.2.1. Representation of Linked List

'C' structure: struct node {
 int data; /*data field */
 struct node * next; /* link field */
} L;

- Declare a structure in which two members are data member and next pointer member
- The **data** member can be a character or integer or really kind of data depending upon the type of the information that the linked list is having.
- The 'next' member is essential to a pointer type. And the pointer should be the structured type. Because the 'next' field holds the address of the next node. And each node is basically a structure consisting of '**data**' and '**next**'.

Program 6.1: Implementation of Linked List

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node {
    int data;
```

```
struct node *next;
}*start = NULL;
void creat()  {
char ch;
do  {
    struct node *new_node,*current;
    new_node = (struct node *)malloc(sizeof(struct node));
    printf("Enter the data of Linkedlist : ");
    scanf("%d",&new_node->data);
    new_node->next = NULL;
    if(start == NULL)  {
        start = new_node;
        current = new_node; }
    else  {
        current->next = new_node;
        current = new_node; }
    printf("Do you want to creat another (y/n):\n ");
    ch = getche();
}while(ch!= 'n'); }
void display()  {
struct node *new_node;
printf("\nThe Linked List :");
new_node = start;
while(new_node!= NULL)  {
    printf("%d-->,"new_node->data);
    new_node = new_node->next; }
printf("NULL"); }
void main()  {
clrscr();
creat();}
```

```
display();
getch(); }
```

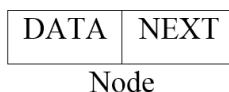
The output of the program is given in Figure 6.3.

```
Enter the data of Linkedlist : 100
Do you want to creat another (y/n):
yEnter the data of Linkedlist : 200
Do you want to creat another (y/n):
yEnter the data of Linkedlist : 300
Do you want to creat another (y/n):
yEnter the data of Linkedlist : 400
Do you want to creat another (y/n):
n
The Linked List :100--->200--->300--->400--->NULL
```

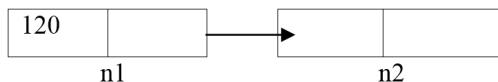
Figure 6.3: Output of Linked List.

Explanation of the logically representation of linked list:

Step 1: We have declared the nodes n1, n2, n3, n4 of structure type. The structure is for singly linked list. So every node will look like this.

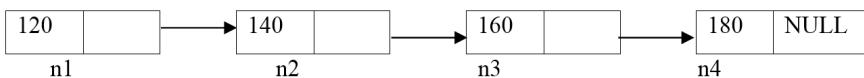


Step 2: We start filling the data in each node at data field and assigning the next pointer to the next node.



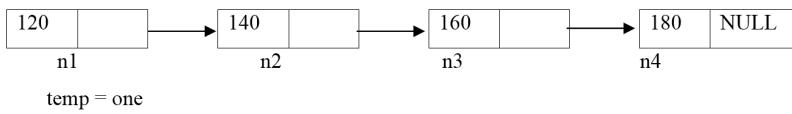
```
n1. data = 120;      n1.next = &n2;
```

Here ‘&’ is a symbol of address. So the above figure can be interpreted as the next pointer of n1 is pointing to the node n2. Then we will start filling the data in each node and will again set the next pointer to next node. Continue this we will get

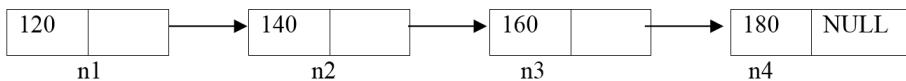


Step 3: To terminate the link list we will set, n4.next = NULL

```
one = &n1; temp = one;
```



Step 4: Now to print the data in a linked list we will use – `printf ("%d," temp → data);`



6.3. CHARACTERISTICS OF LINKED LIST

Linked Lists have the following properties:

- The size and contents of linked lists may be variances at runtime, depending on the implementations.
- Random access over linked lists may be possible, depending on the implementation.
- In modern programming languages, equality of lists is normally defined in terms of structural equality of the corresponding entries, except that if the lists are typed, then the list types may also be relevant.
- In a list, there is a linear order (called followed by or next) defined on the elements, every element is followed by one other element, and no two elements are followed by the same element.

6.3.1. Operations on the Linked List

Following are some of the basic operations that may be performed on Linked lists:

- Create a linked list
- Check for an empty linked list
- Search for an element in a list
- Search for a predecessor or a successor of an element of a list
- Delete an element at a specified location of a list
- Add an element at a specified location of a list
- Retrieve an element from a list
- Update an element of a list
- Print a list
- Delete a list

6.3.2. Advantages of Linked List

- A linked list is a dynamic data structure – it means that they can grow or shrink size during the execution of the program.
- Efficient memory utilization – Memory is allocated whenever it is required and is deallocated when it is no longer needed.
- Insertion and deletion operations are easier and efficient at any location.
- Many complex applications can be easily carried out with linked lists.

6.3.3. Disadvantages of Linked List

- The linked organization does not support random or direct access.
- If the numbers of fields are more, then more memory space is needed.
- Each data field should be supported by a link field to point to the next node.

6.4. TYPES OF LINKED LIST

There are various types of a linked list such as:

- Singly Linked List
- Singly Circular Linked List
- Doubly Linear Linked List
- Doubly Circular Linked List

6.4.1. Singly-Linked List

The simplest type of linked list is a singly-linked list which has one link per node. This link points to the next node in the list, or to a null value or the empty list if it is the final node.



Figure 6.4: A singly-linked list containing three integer values.

Various operations on the singly linked list are:

1. Creation of a Linked List

Initially, one variable flag is taken whose value is initialized to TRUE. The purpose of the flag is for making a check on the creation of the first node. That means if the flag is TRUE, then we have to create a head node or the first node of the linked list. After creation of the first node, we will reset the flag consider we have entered the element value 220 initially then

Step 1: New = get_node (); // memory gets allocated for new node

New → data = value; // value 20 will be put in data field of New

Data	Next
220	NULL
New	

Step 2: if (flag == TRUE) {

head = new;

temp = head; /* this node as temp because head's address will be preserved in

‘head’ and we can change ‘temp’ node as per requirement */

flag = FALSE; }

Data	Next
220	NULL
New/head /temp	

Step 3: If a head node of the linked list is created we can further create a linked list by attaching the subsequent nodes. Suppose we want to insert a node with value 20 then Gets created after invoking get_node () ;

220	NULL
head/temp	

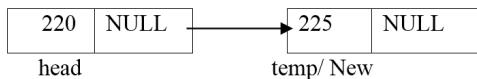
225	NULL
New	

temp → next = New;

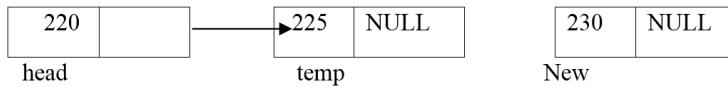
head/temp

New temp = new; // now

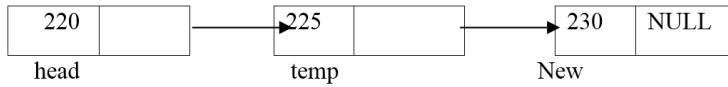
temp is moved ahead



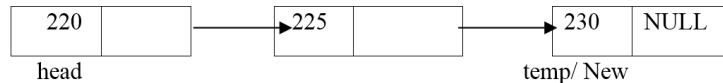
Step 4: If the user wants to enter more elements then let say for value 30 the scenario will be Gets created after invoking get_node () ;



temp → next = New;



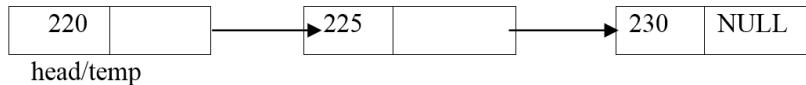
temp = new; // now temp is moved ahead



This is a final linked list.

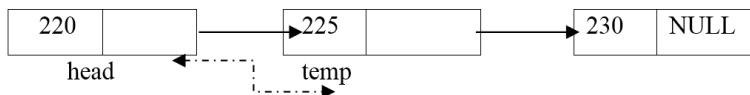
2. Display of Linked List

We are passing the address of the head node to the display routine and calling head as the ‘temp’ node. If the linked list is not created then head = temp node will be NULL. Therefore the message “the list is empty” will be displayed. If we have created some linked list like this:



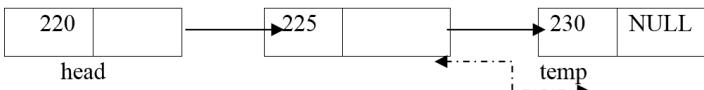
set temp = temp → next

temp → data i.e., 225 will be displayed as temp! = NULL temp



set temp = temp → next

temp → data i.e., 225 will be displayed as temp = NULL temp we will come out of the loop.



As a result display on the console.

220 → 225 → 230 → NULL.

3. Insertion of Any Element in the Linked List

There are three possible cases when we want to insert an element in the linked list-

- Insertion of a node as a head node.
- Insertion of a node as the last node.
- Insertion of a node after some node.

Case 1: Insertion of a node as a head node: if there is a node in the linked list then the value of a head is NULL. At that time if we want to insert 18 then

```

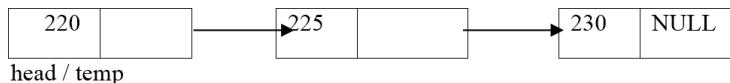
scanf ("%d," &New → data)
if (head == NULL)
    head = New;

```

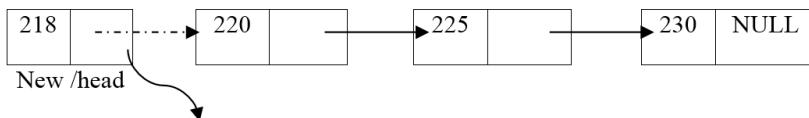
Data	Next
218	NULL

head / New

Otherwise suppose linked list is already created like this



If we want to insert this node as a head node then

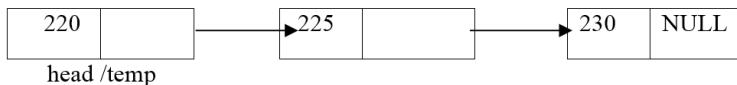


Node is attached to the linked list as a head node

New → next = temp

head = New

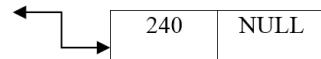
Now we will **insert a node at the end** case 2: To attach a node at the end of linked list assume that we have already created a linked list like this,

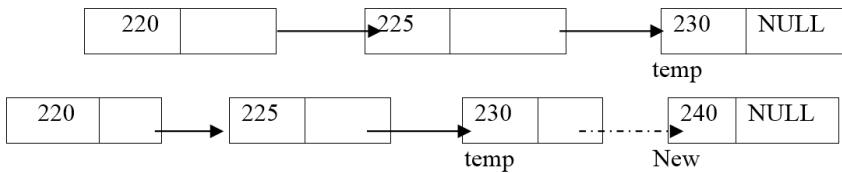


If we insert this node at last node then

while (temp → next! = NULL)

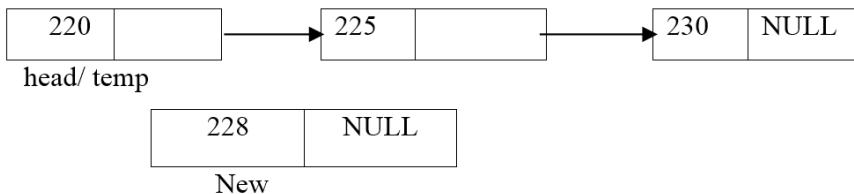
temp = temp → next; // traversing linked list



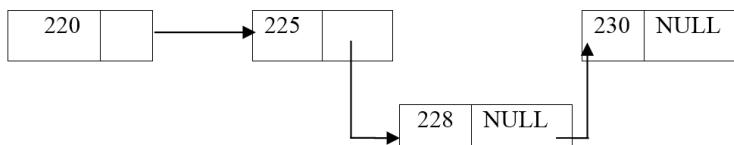


`temp → next = New ; New → next = NULL ;`

Now we will **insert a node after a node** case 3: Suppose we want to insert node 228 after containing 225 then



Then,



If (`temp → data = key`) {

`New → next = temp → next;`

`temp → next = New;`

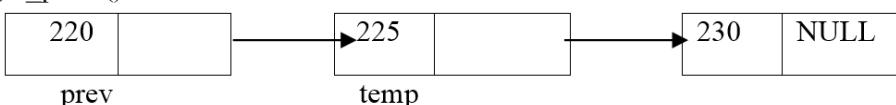
`return }`

4. Deletion of Any Element in the Linked List

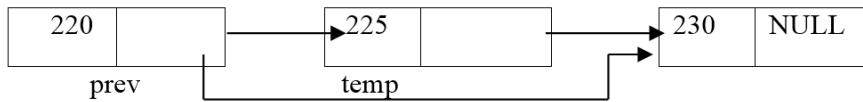
Suppose we have,



Suppose we want to delete node 225. Then we will search the node containing 225, using search (*`head, key`) routine. Mark the node to be deleted as **a temp**. Then we will obtain the previous node of **temp** using `get_prev ()` function.



then, $\text{prev} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$



Now we will **free** the **temp** node using the **free** function. Then linked list will be –

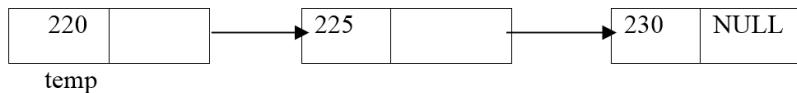


This can be done using the following statements

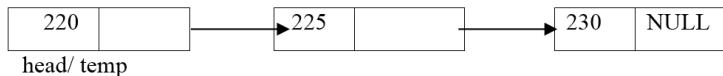
```
*head = temp → next;
free (temp);
```

5. Searching for Any Element in the Linked List

Consider that we have created a linked list as



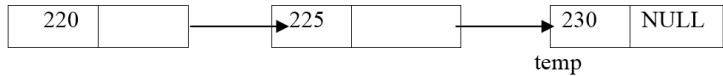
Suppose key = 30, we want a node containing value 30 then compare $\text{temp} \rightarrow \text{data}$ and key value. If there is no match then we will mark the next node as a temp.



Is $\text{temp} \rightarrow \text{data}$? Key → NO



Is $\text{temp} \rightarrow \text{data}$? Key → Yes



Hence print the message “the element is present in the list.”

Program 6.2: To perform various operations such as creation, insertion, deletion, search and display on the single link list.

Program: #include<stdio.h>

```
#include<stdlib.h>
#include<conio.h>
typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;
struct Node
{
    int e;
    Position next;
};
void Insert(int x, List l, Position p)
{
    Position TmpCell;
    TmpCell = (struct Node*) malloc(sizeof(struct Node));
    if(TmpCell == NULL)
        printf("Memory out of space\n");
    else {
        TmpCell->e = x;
        TmpCell->next = p->next;
        p->next = TmpCell; }
    int isLast(Position p) {
        return (p->next == NULL); }
Position FindPrevious(int x, List l) {
    Position p = l;
    while(p->next != NULL && p->next->e != x)
        p = p->next;
    return p; }
void Delete(int x, List l)
{
    Position p, TmpCell;
    p = FindPrevious(x, l);
```

```
if(!isLast(p))    {
    TmpCell = p->next;
    p->next = TmpCell->next;
    free(TmpCell);  }

else
    printf("Element does not exist!!!\n");  }

void Display(List l)
{
    printf("The list element are :: ");
    Position p = l->next;
    while(p != NULL)  {
        printf("%d -> , " p->e);
        p = p->next;  }
}

void Merge(List l, List l1)
{
    int i, n, x, j;
    Position p;
    printf("Enter the number of elements to be merged :: ");
    scanf("%d," &n);
    for(i = 1; i <= n; i++)  {
        p = l1;
        scanf("%d," &x);
        for(j = 1; j < i; j++)
            p = p->next;
        Insert(x, l1, p);  }

    printf("The new List :: ");
    Display(l1);
    printf("The merged List ::");
    p = l;
    while(p->next != NULL)
{
```

```
p = p->next; }
p->next = l1->next;
Display(l);
}

int main()
{
clrscr();
int x, pos, ch, i;
List l, l1;
l = (struct Node *) malloc(sizeof(struct Node));
l->next = NULL;
List p = l;
do {
printf("\n1. INSERT\t 2. DELETE\t 3. MERGE\t 4. PRINT\t 5. QUIT\
nEnter the choice :: ");
scanf("%d," &ch);
switch(ch) {
    case 1:
        p = l;
        printf("Enter the element to be inserted :: ");
        scanf("%d,"&x);
        printf("Enter the position of the element :: ");
        scanf("%d,"&pos);
        for(i = 1; i < pos; i++) {
            p = p->next; }
        Insert(x,l,p);
        break;
    case 2:
        p = l;
        printf("Enter the element to be deleted :: ");
        scanf("%d,"&x);}
```

```

        Delete(x,p);
        break;
    case 3:
        l1 = (struct Node *) malloc(sizeof(struct Node));
        l1->next = NULL;
        Merge(l, l1);
        break;
    case 4:
        Display(l);
        break;
}
}

while(ch<5);
return 0;
getch();
}

```

The output of the program is given in Figure 6.5.

```

1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 2
Enter the element to be deleted :: 10
Element does not exist!!!

1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 25
Enter the position of the element :: 1

1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 50
Enter the position of the element :: 2

1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 4
The list element are :: 25 -> 50 ->
1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: -

```

Figure 6.5: Output of Single linked list operations.

6.4.2. Array and Linked List Comparison

S.N.	Array	Linked List
1.	Any element can be accessed randomly with the help of index of the array.	Any element can be accessed sequential access only.
2.	Only logical deletion of data is possible.	Physically the data can be deleted.
3.	Insertion and deletion of data are difficult.	Insertion and deletion of data is easy.
4.	The memory allocation is static.	The memory allocation is dynamic.

6.4.3. Circular Linked List (CLL)

The circular linked list (CLL) is similar to a singly linked list except that the last node's next pointer points to the first node. The singly linked list, the last node of such lists contains the null pointer. We can improve this by replacing the null pointer in the last node of a list with the address of its first node. Such a list is called a **circularly linked list** or a **circular list**. The circular linked list is as shown below-



When we are traversing a circular list, we must be careful as there is a possibility to get into an infinite loop, if we are not able to detect the end of the list. To do that we must look at the starting node, we can keep an external pointer at the starting node and look for this external pointer as a stop sign. An alternative method is to place the header node of the first node of a circular list. This header node may contain a special value in its info field that cannot be the valid contents of a list in the context of the problem. If a circular list is empty, then the external pointer will point to null.

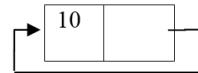
1. Creation of circular linked list:

First, we will allocate memory for **New** node using a function **get_node ()**. There is one variable **flag** whose purpose is to check whether the first node is created or not. That means the flag is 1 (set) then the first node is not created. Therefore, after the creation of the first node, we have to reset the flag (set 0).

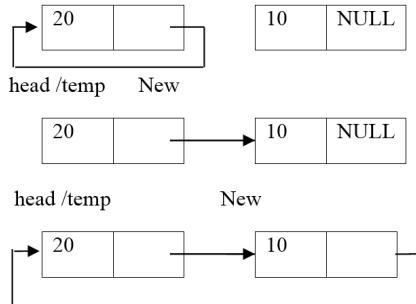
Initially, variable **head** indicates starting node. Suppose we have taken the element '10' the flag = 1, head = New;

flag = 1, head = New;

New → next = head; flag = 0;



Now as flag = 0, we can further create the nodes and attach them as follows. When we have taken element '10'



temp = head;

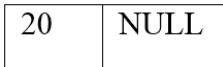
temp → next = head;

temp → next = New;

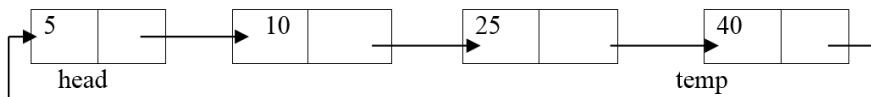
New → next = head;

2. **Insertion of a node in circular linked list:** Inserting a new node in the circular linked list, there are 3 cases:

Case 1. If we want to insert a New node as a head node then,



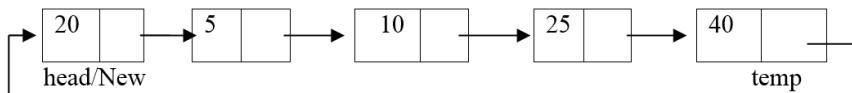
Noe then



temp → next = New;

New → next = head;

head = New;



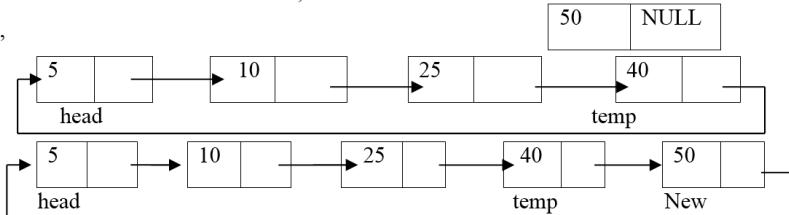
Case 2. If you want to insert a New node at last node on the circular linked list given

A New node as the last node then,

New

Then,

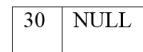
Then,



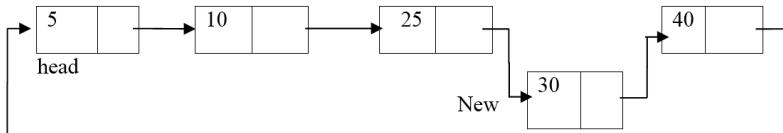
Case 3. If we want to insert an element 30 after node 25 then

A New node 30 then,

New Then,



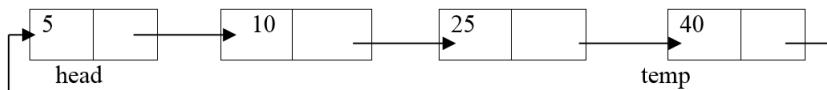
as key = 30 and temp → data = 30



New → next = temp → next; temp → next = New;

3. Deletion of any node in the Circular link list

Suppose we have created a linked list as then,



if we want to delete temp → data node 5 then

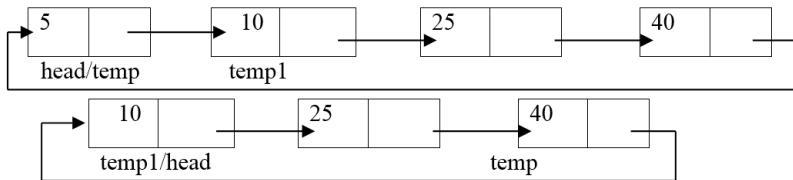
temp = temp1 → New;

while (temp → next ! head)

temp = temp → next;

temp → next = temp1;

head = temp1;



Program 6.3 Circular Linked List Operation in ‘C’

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

typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

struct Node
{
    int e;
    Position next;
};

void Insert(int x, List l, Position p)  {
    Position TmpCell;
    TmpCell = (struct Node*) malloc(sizeof(struct Node));
    if(TmpCell == NULL)
        printf("Memory out of space\n");
    else
        TmpCell->e = x;
    TmpCell->next = p->next;
    p->next = TmpCell;
}

int isLast(Position p, List l)  {
    return (p->next == l);
}

Position FindPrevious(int x, List l)  {
    Position p = l;
    while(p->next != l && p->next->e != x)
        p = p->next;
}
```

```
return p;    }
Position Find(int x, List l)  {
    Position p = l->next;
    while(p != l && p->e != x)
        p = p->next;
    return p;    }
void Delete(int x, List l)  {
    Position p, TmpCell;
    p = FindPrevious(x, l);
    if(!isLast(p, l))  {
        TmpCell = p->next;
        p->next = TmpCell->next;
        free(TmpCell);    }
    else
        printf("Element does not exist!!!\n");
    void Display(List l)  {
        printf("The list element are :: ");
        Position p = l->next;
        while(p != l)  {
            printf("%d -> , " p->e);
            p = p->next;    }    }
    void main()  {
        clrscr()
        int x, pos, ch, i;
        List l, l1;
        l = (struct Node *) malloc(sizeof(struct Node));
        l->next = l;
        List p = l;
        printf("\nCIRCULAR LINKED LIST IMPLEMENTATION");
        do  {
            printf("\n1. INSERT\t2. DELETE\t3. FIND\t4. PRINT\t5. QUIT\n\nEnter
```

```
the choice :: “);
scanf(“%d,” &ch);
switch(ch)  {
case 1:
p = l;
printf(“Enter the element to be inserted :: “);
scanf(“%d,”&x);
printf(“Enter the position of the element :: “);
scanf(“%d,”&pos);
for(i = 1; i < pos; i++)  {
p = p->next;
}
Insert(x,l,p);
break;
case 2:
p = l;
printf(“Enter the element to be deleted :: “);
scanf(“%d,”&x);
Delete(x,p);
break;
case 3:
p = l;
printf(“Enter the element to be searched :: “);
scanf(“%d,”&x);
p = Find(x,p);
if(p == l)
printf(“Element does not exist!!!\n”);
else
printf(“Element exist!!!\n”);
break;
case 4:
Display(l);
```

```

break; } }
while(ch<5);
getch();
}

```

The output of the program is given in Figure 6.6.

```

CIRCULAR LINKED LIST IMPLEMENTATION
1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 45
Enter the position of the element :: 1

1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 55
Enter the position of the element :: 0

1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 4
The list element are :: 55 -> 45 ->
1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 2
Enter the element to be deleted :: 55

1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: -

```

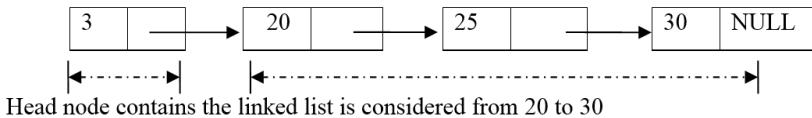
Figure 6.6: Output of Circular Linked List Operations.

Advantages of circular list over singly linked list: In circular linked list the next pointer of last node points to the head node. Hence we can move from last node to the head node of the list very efficiently. Hence accessing of any node is much faster than singly linked list.

6.4.4. Concept of Header Node

A header node is a linked list which always contains a special node, called the header node. The head node is a node which resides at the beginning of the linked list. Sometimes such extra node needs to be kept at the front of the list. But it may contain some useful information about linked list such as total number of nodes in the list, address of last or some specific unique information. The following are two types of header list:

- A **grounded header list** is a header list where the last node contains the null pointer.
- A **circular head list** is a header list where the list node points back to the header node. For example:



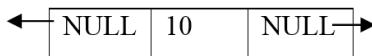
Total number of node in the list

6.4.5. Doubly Linked List (DLL)

A doubly linked list is one in which nodes are linked together by multiple numbers of links. Each node in a doubly linked list contains two pointers, one link field is the previous pointer and the other linked field is that next pointer. Thus, each node in a doubly linked list contains three fields- an info field that contains the data in the node, and prev and next fields. The doubly linked list can traverse in both the directions, forward as well as backward. It may be either linear or circular and may not contain a header node as shown in Figure 6.8.



Figure 6.7: Doubly Linked List.

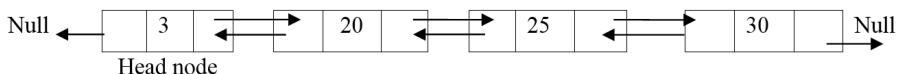


Single node

‘C’ Structure of doubly linked list:

```
typedef struct node
{
    int data;
    struct node *prev;
    struct node *next;
}dnode;
```

The linked representation of a doubly linked list is



Various operations that can be performed on the doubly linked list are:

- Insertion of a node in the doubly linked list
- Deletion of any node from a linked list

1. Insertion of a node in doubly linked list:

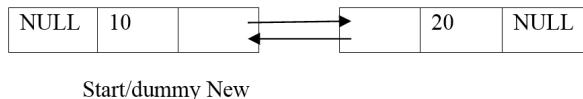
Step 1: a new node as initially a flag is taken to check whether it is the first node in a variable first, as first = 0; as soon as the very first node gets created we reset the first. First = 1;



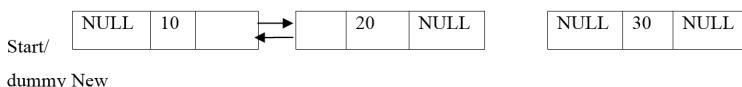
Step 2: For further addition of the nodes the New node is created.



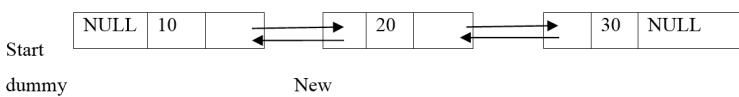
dummy → next = New; New → prev = dummy;



Step 3: For further addition of the nodes the New node is created.

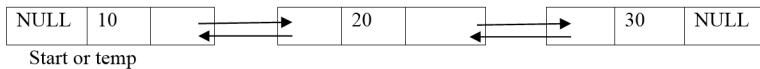


When dummy → next! = NULL; dummy = dummy → next;
then attach new node in the linked list.

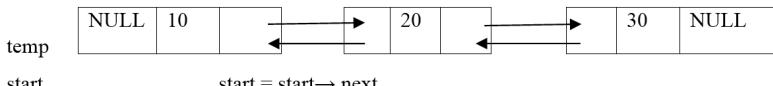


Deletion of any node from linked list:

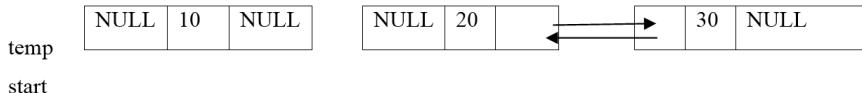
Step 1: we assume the linked list as



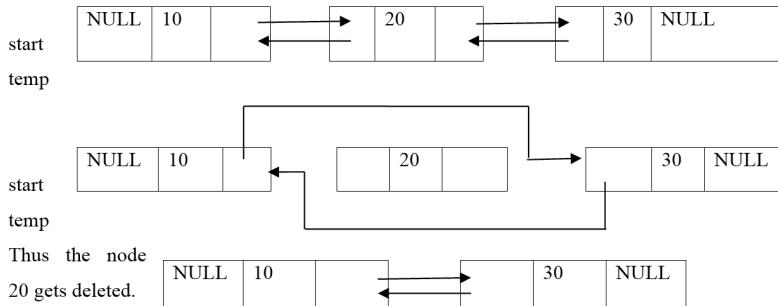
If the very first node has to be deleted, then



start → prev = NULL; temp → next = NULL



Step 2: if we want to delete any node other than the first node then, we want to delete the node than 20 calls it as temp node.



Program 6.4: Doubly Link List Operations

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

typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

struct Node
{
    int e;
    Position next;
};

void Insert(int x, List l, Position p)  {
    Position TmpCell;
    TmpCell = (struct Node*) malloc(sizeof(struct Node));
    if(TmpCell == NULL)
        printf("Memory out of space\n");
    else  {
        TmpCell->e = x;
```

```
TmpCell->next = p->next;
p->next = TmpCell; } }

int isLast(Position p) {
    return (p->next == NULL); }

Position FindPrevious(int x, List l) {
    Position p = l;
    while(p->next != NULL && p->next->e != x)
        p = p->next;
    return p; }

void Delete(int x, List l) {
    Position p, TmpCell;
    p = FindPrevious(x, l);
    if(!isLast(p)) {
        TmpCell = p->next;
        p->next = TmpCell->next;
        free(TmpCell);
    }
    else
        printf("Element does not exist!!!\n");
}

void Display(List l) {
    printf("The list element are :: ");
    Position p = l->next;
    while(p != NULL) {
        printf("%d -> ", p->e);
        p = p->next; } }

void Merge(List l, List l1) {
    int i, n, x, j;
    Position p;
    printf("Enter the number of elements to be merged :: ");
    scanf("%d",&n);
    for(i = 1; i <= n; i++) {
        p = l1;
```

```
scanf("%d," &x);
for(j = 1; j < i; j++)
p = p->next;
Insert(x, l1, p);  }
printf("The new List :: ");
Display(l1);
printf("The merged List ::");
p = l;
while(p->next != NULL)  {
p = p->next;  }
p->next = l1->next;
Display(l);  }
int main()  {
clrscr();
int x, pos, ch, i;
List l, l1;
l = (struct Node *) malloc(sizeof(struct Node));
l->next = NULL;
List p = l;
do  {
printf("\n1. INSERT\t 2. DELETE\t 3. MERGE\t 4. PRINT\t 5. QUIT\t");
nEnter the choice :: ");
scanf("%d," &ch);
switch(ch)  {
case 1:
p = l;
printf("Enter the element you want to inserted :: ");
scanf("%d," &x);
```

```
printf("Enter the position of inserted element :: ");
scanf("%d",&pos);
for(i = 1; i < pos; i++) {
    p = p->next;
}
Insert(x,l,p);
break;
case 2:
    p = l;
    printf("Enter the element to be deleted :: ");
    scanf("%d",&x);
    Delete(x,p);
    break;
case 3:
    l1 = (struct Node *) malloc(sizeof(struct Node));
    l1->next = NULL;
    Merge(l, l1);
    break;
case 4:
    Display(l);
    break;
}
while(ch<5);
return 0;
getch();
}
```

The output of the program is given in Figure 6.8.

```

1 - Insert at beginning 2 - Insert at end 3 - Insert at position i
4 - Delete at position i 5 - Display from beginning 6 - Display from end
7 - Search for element 8 - Sort the list 9 - Update an element 10 - Exit
Enter choice : 1

Enter value to node : 100

Enter choice : 2

Enter value to node : 200

Enter choice : 5

Linked list elements from begining : 100 200
Enter choice : 3

Enter position to be inserted : 1

Enter value to node : 50

Enter choice : 5

Linked list elements from begining : 100 50 200
Enter choice : -

```

Figure 6.8: Output of Doubly linked list Operations.

6.4.6. Difference between the Singly and Doubly Linked Lists

S.N	Singly Linked List	Doubly Linked List					
1.	The singly linked list is a collection of nodes and each node is having one data field and next link field. For example: <table border="1"><tr><td>Data</td><td>Next</td></tr></table>	Data	Next	The doubly linked list is a collection of nodes and each node is having one data field, one previous link field, and one next link field. For example: <table border="1"><tr><td>Previous</td><td>Data</td><td>Next</td></tr></table>	Previous	Data	Next
Data	Next						
Previous	Data	Next					
2.	Less efficient access to elements.	More efficient access to elements.					
3.	The elements can be accessed using the next link.	The elements can be accessed using both previous links as well as next link.					
4.	No extra field is required; hence the node takes less memory in SLL.	One field is required to store previous link hence node takes more memory in DLL.					

6.5 POLYNOMIAL REPRESENTATION OF LINKED LIST

A linked list is used to represent and manipulate polynomials. Each node contains three fields, one representing the coefficient, the second representing

the exponent and third is a pointer to the next term. The polynomial node is shown in Figure 6.9.



Figure 6.9: Structure of a polynomial node.

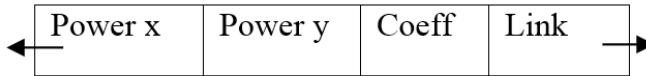
The important information about a polynomial is contained in the coefficient and exponents of x . The variable x is dropped. The variable x itself is just a placeholder variable. Thus, each node will be a structure contains a nonzero coefficient, an exponent and a pointer to the next of the polynomial. So the polynomial,

$4x^4 + 2x^3 - x^2 + 3$ is represented as a list of structures as shown in figure 6.10.



Figure 6.10: list structure of polynomial.

To represent a term a polynomial in the variables x and y , each node consists of four sequentially allocated fields that can be referred to as term. The first two fields represent the power of the variables x and y respectively. The third and fourth field represents the coefficient of the term in the polynomial and the address of the next tern in the polynomial. For example:



The polynomial $4x^4y^2 + 2x^3y - x^2 + 3$ is represented as a linked list by Figure 6.11.

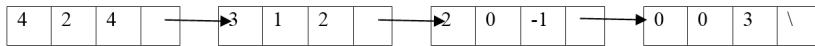


Figure 6.11: Linked representation of a polynomial two variable.

7

CHAPTER

TREE

CONTENTS

7.1. Introduction.....	162
7.2. Definition of Trees	162
7.3. Binary Tree.....	165
7.4. Binary Tree Representation.....	169
7.5. Binary Tree Traversal	172
7.6. Binary Search Tree (BST)	179
7.7. Height Balanced (AVL) Tree	196

7.1. INTRODUCTION

In earlier chapters, we have discussed some linear data structures such as arrays, stacks, queues, linked lists. Now we will learn some nonlinear data structures such as trees and graphs. Trees are basically used to represent the data items in a hierarchical manner. A tree is a non-linear data structure in which data are not arranged in a sequential manner. It is used to represent hierarchical relationships existing amongst several data items. It is one of the most important features of data structures that used computer science in many applications.

7.2. DEFINITION OF TREES

A tree ‘T’ can be defined as a finite set of one or more nodes that arranged in such a way:

- There exists a single node known as **the root** node of the tree.
- The remaining nodes of a tree divided into $n \geq 0$ disjoint subsets $T_1, T_2, T_3, \dots, T_n$, where sets belong to a tree. $T_1, T_2, T_3, \dots, T_n$ is called the sub-trees of the tree.

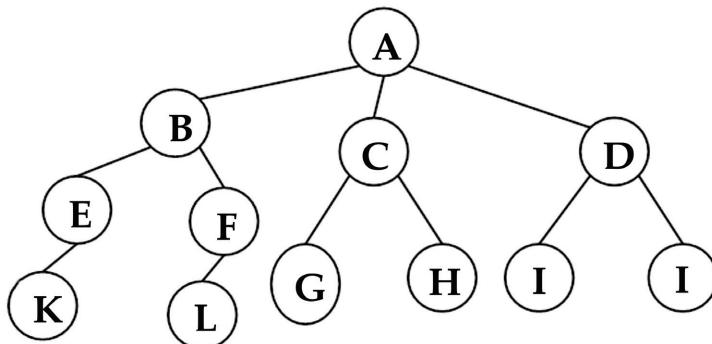


Figure 7.1: Tree.

A **forest** can be defined as a set of $n \geq 0$ disjoint trees. A forest can be generalized if we remove a root node from the given tree. One such 7.1 trees and the respective forest are shown in Figure 7.2.

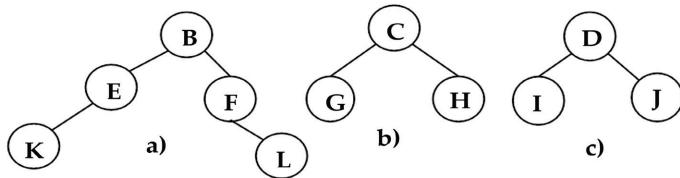


Figure 7.2: a, b and c represent a forest.

7.2.1. Basic Terminologies of Tree

Consider a tree shown in Figure 7.3 the tree has 14 nodes. Node 'A' is a **Root** node.

Left Child: In a tree, the node to the left of the root is called its left child.

Right Child: In a tree, the node to the right of the root is called its right child.

The degree of Node: The number of subtrees connected of a node is referred to its **degree** of the node. Thus the degree of a node 'A' is 3. Similarly, the degree of node 'E' is 1, and 'L' is 0.

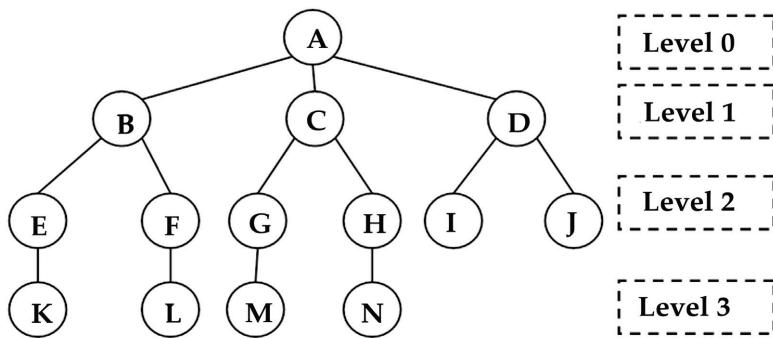


Figure 7.3: An example of Tree.

The degree of Tree: The **degree of a tree** is the maximum degree of any nodes in the tree. Figure 7.3-tree degree is 3.

Terminal Nodes: if a Nodes having degree zero are known as **Terminal nodes** or **leaf node**, and the nodes other than these nodes are known as **Non-terminal nodes**.

terminal nodes or a **Non-leaf node**. The degree of each node of figure 7.3 is given below:

NODES	DEGREES
A	3
B	2
C	2
D	2
E	1
F	1
G	1
H	1
I	0
J	0
K	0
L	0
M	0
N	0

Terminal Nodes {I, J, K, L, M, N}, **Non-Terminal Nodes** {A, B, C, D, E, F, G, H}. The node ‘A’ is the root node of a tree, and that “A” is the **parent** of nodes labeled ‘B,’ ‘C’ and ‘D.’ Nodes labeled ‘B,’ ‘C’ and ‘D’ are the **children** of node ‘A.’ Children of the same parent are called **siblings**. “A” is the parent of nodes labeled ‘B,’ ‘C’ and ‘D’ hence ‘B,’ ‘C’ and ‘D’ are siblings. The **ancestors** of a node are all the nodes along the path from the root node to that node. The ancestors of node ‘K’ are ‘E,’ ‘B’ and ‘A.’ The **descendants** of a node are all the nodes along the path from the node to terminal node. The descendants of ‘A’ are ‘B,’ ‘E’ and ‘K.’

Left Subtree: The subtree whose root is the left of some node is called the left subtree of the node. **Right Subtree:** The subtree whose root is the right child of some node is called the right subtree of that node.

Path: A **path** is referred to as a linear subset of a tree, for instance, A-B-E-K, and A-D-J are the paths. It is to be noted that there exists a unique path between a root node any other node. **Length** of the path is either calculated by the number of the intermediated nodes or the number of edges on the path. **Level** of a node is defined by the setting the root node level at zero. If any node has level ‘l’ then its children are at level $l+1$, see Figure 7.3.

The depth of the root node is zero, and the depth of any node is one plus the depth of its parent. **Height** (or sometimes depth) of a tree is the maximum level of any node in the tree.

External Nodes: a node which has no children is called external nodes or terminal nodes. **Internal Nodes:** The nodes which have one or more than one children are called internal nodes or non-terminal nodes.

7.2.2. Common Operations on Trees

- Enumerating all the items
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Removing a whole section of a tree (pruning)
- Adding a whole section to a tree (grafting)
- Finding the root of any node

7.2.3. Common Application of Trees

- Manipulate hierarchical data
- Make information easy to search
- Manipulate sorted lists of data
- Classification of data
- Useful for representing an expression.
- Useful in decision making in game theory.

7.3. BINARY TREE

Binary means maximum two, a tree is a special type of data structure in which the number of children of any node is restricted to almost two. A binary tree is a finite set of an element that is either empty or is partitioned into three disjoint subsets. The first node contains a single data called the root of the tree. The further two subtrees are binary trees, called the left and right subtrees of the original tree. The Binary tree ‘BT’ may also have zero nodes and can be defined recursively as:

- An empty tree is a binary tree.
- A distinguished node (unique node) known as the root node.

- The remaining nodes are divided into two disjoint sets ‘L’ and ‘R,’ where ‘L’ is a left sub-tree and ‘R’ is right subtree such that these are a binary tree once again.

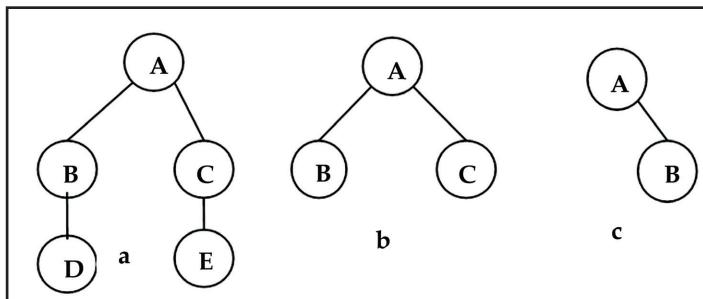


Figure 7.4: Binary tree.

7.3.1 Strictly Binary Tree

A binary tree is called a strictly binary tree if every non-leaf or terminal node in the binary tree has nonempty left and right subtree. It means that each node in a tree will have either 0 or 2 children. Figure 7.5 shows (a), (b) and (c) strictly binary tree.

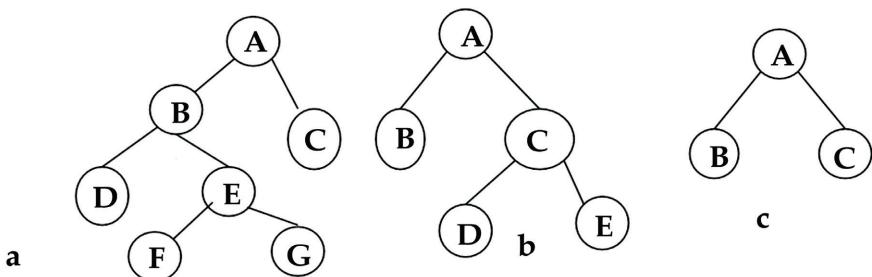


Figure 7.5: Strictly Binary Tree.

7.3.2. Almost Complete Binary Tree

A binary tree of depth d is an almost complete binary tree if- Any node at level less than $d - 1$ has two sons and any node in the tree with a right descendant at level d , node should have a left son and each left descendant of node is either a leaf at level d or has two sons. Figure 7.6 shows almost complete binary tree.

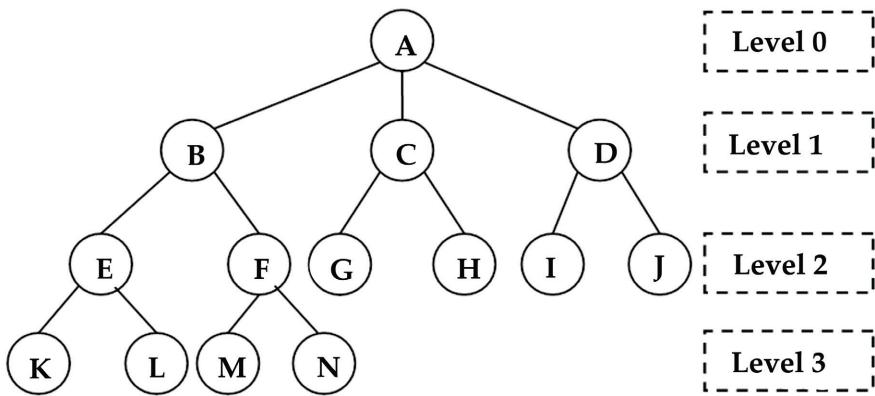


Figure 7.6: Almost Complete.

7.3.3. Complete Binary Tree

A complete binary is a tree in which all non-terminal nodes have degree 2 and all terminal nodes are at the same depth. A binary tree has n nodes and of depth, k is complete if its nodes correspond to the nodes which are numbered one to n in the full binary tree of depth k .

if there are m nodes at level l then a binary tree contains at most 2^m nodes at level $l+1$.

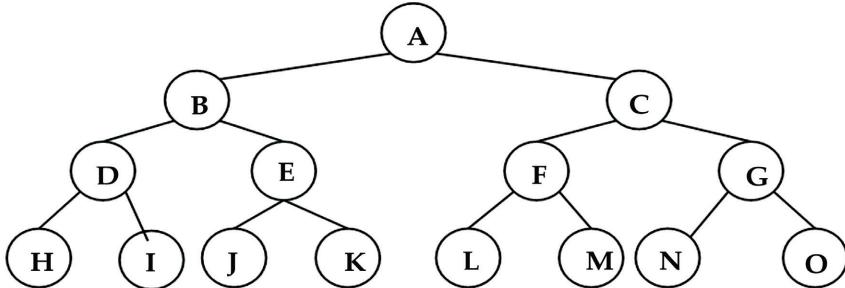


Figure 7.7: Complete Binary Tree.

7.3.4. Extended Binary Tree

A binary tree is called extended or 2-tree if each node N has either 0 or 2 children. In this case, the nodes with degree 2 are called internal and nodes

with degree 0 are called external nodes. Figure 7.8 shows an extended binary tree using circles for internal nodes and squares for external nodes.

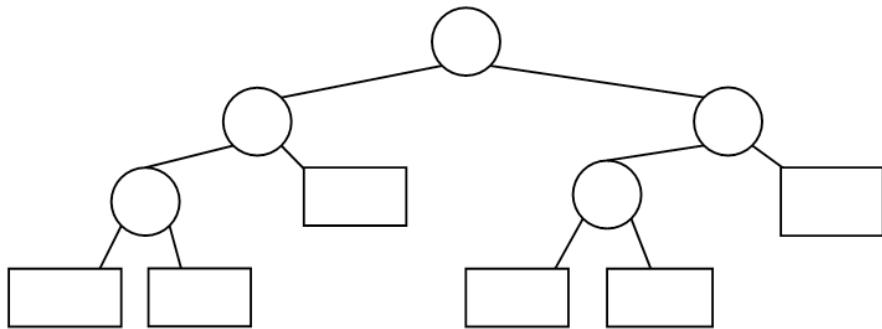


Figure 7.8: Extended Binary Tree.

7.3.5. Skewed Trees

Generally, skewed trees are two types of' **left-skewed binary tree** and **right-skewed binary tree**. A binary tree is left-skewed in which each node has an only left child, and **right-skewed binary tree** in which each node has an only right child is shown in Figure 7.9.

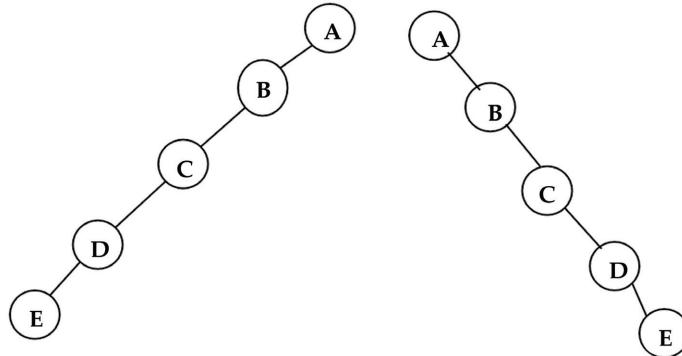


Figure 7.9: (a) Left -skewed binary tree (b) Right – skewed binary tree.

7.3.6. General Tree

A general tree is a special type of data structure in which the number of children of any node is not restricted to almost two. General tree T is a finite set of one or more nodes such that there is a special node R, that is called

the root of the tree, and remaining nodes are divided into $n \geq 0$ disjoint subtree $T_1, T_2 \dots T_n$ and whose roots R_1, R_2, \dots, R_n .

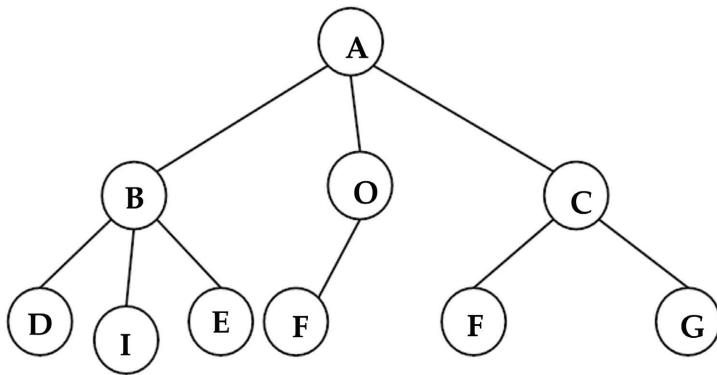


Figure 7.10: General Tree.

7.3.7. Expression Tree

Algebraic expressions involving addition, subtraction, multiplication, and division can be represented as ordered rooted trees called expression trees.

Example: The arithmetic expression $4 + 5 * 9 - 7 * 6$ can be represented as a tree.

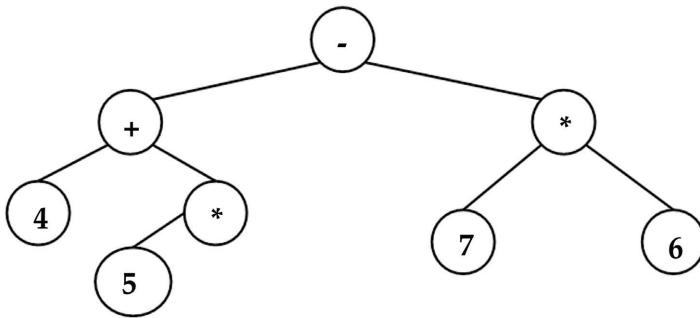


Figure 7.11: Expression Tree.

7.4. BINARY TREE REPRESENTATION

A binary tree can be represented by two popular methods that are used to maintain tree in the memory. These are sequential allocation method and using linked lists.

1. Sequential Allocation

A binary tree can be represented in a sequential manner. An array can be used to store the nodes of a binary tree. The nodes stored in an array are accessibly sequenced. In C, arrays start with index 0 to MAXSIZE – 1. Hence, the numbering of binary tree nodes starts from 0 rather than 1. Thus, the maximum number of nodes is specified by MAXSIZ. The root node is at index 0. Then, in successive memory locations, the left child and the right child are stored.

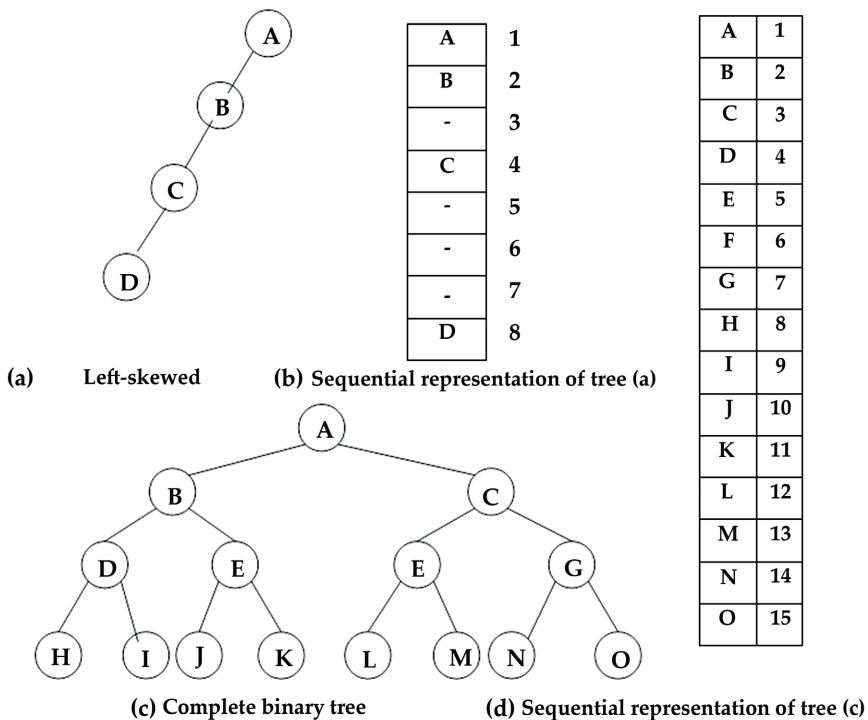


Figure 7.12: Sequential representation of a binary tree.

2. Linked List Representation

In this representation each node of a binary tree consists of three parts where: First part contains data, the second and third part contains the pointer field which points to the left child and right child.

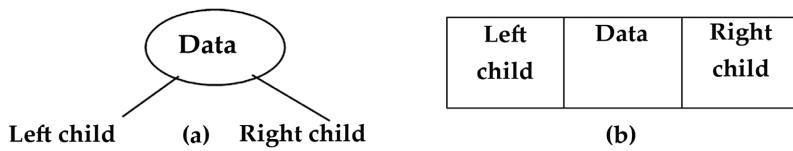


Figure 7.13: Structure of a node binary tree.

Tree nodes may be implemented as array elements or dynamically allocated variables. Each node of a tree contains data, lchild and rchild fields. Using the array implementation, we can declare

```
# define MAXSIZE 10
struct tree node
{
    int data;
    int lchildd;
    int rchild;
};

struct treenode BTNODE [MAXSIZE];
```

Consider the binary tree in Figure 7.14 (a) and 7.14 (b) are linked representation

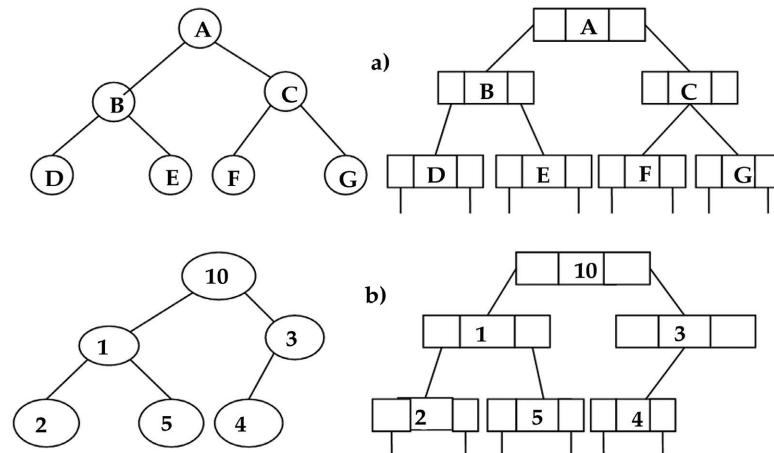


Figure 7.14: Linked List Representation.

7.5. BINARY TREE TRAVERSAL

Binary Tree traversing is the method by which traversing (processing) each node in the tree exactly once. The tree traversal is the most important operation that is performed on the tree data structures. The complete traversing of a binary tree signifies processing of nodes in some systematic manner. While traversing trees, once we start from the root, there are two ways to go, either left or right subtree. At a given node, there are three things to do in some order. To visit node itself, to traverse its left subtree and to traverse its right subtree. If root, left subtree and right subtree are designated by R, L, R respectively then the possible traversal orders can be: Right Root Left (RR'L), Right Left Root (RLR'), Left Root Right (LRR'), Left Right Root (LRR'), Root Right Left (R'RL), Root Left Right (R'LR). Here the processing of a node depends upon the nature of the application.

The three standard traversal orders are:

1. **Pre-order Traversal (R'LR):** the pre-order traversal of a binary tree is as follows:
 - First, process the root node.
 - Second, traverse the left sub-tree in pre-order.
 - Lastly, traverse the right sub-tree in pre-order.

If the tree has an empty subtree the traversal is performed by doing nothing. That means a tree having NULL sub-tree is considered to be completely traversed when it is encountered.

Algorithm Preorder: The pointer variable '**Node**' stores the address of the root node.

Step 1: Is empty?

```
If (empty [Node]) then
  Print "Empty tree" return
```

Step 2: Process the root node

```
If (Node ≠ NULL) then
  Output: (Data [Node])
```

Step 3: Traverse the left subtree

```
If (Lchild [Node] ≠ NULL) then
  Call preorder (Lchild [Node])
```

Step 4: Traverse the right subtree

```
If (Rchild [Node] ≠ NULL) then
  Call preorder (Rchild [Node])
```

Step 5: return at the point of call

Exit

Consider a binary tree and binary arithmetic expression tree in Figure 7.15 (a) and (b).

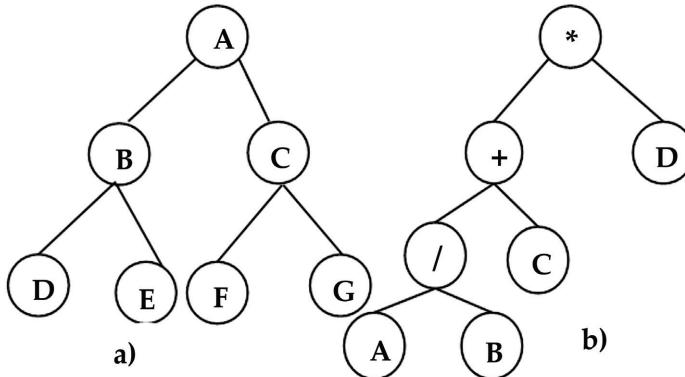


Figure 7.15: Binary Tree (a) and (b).

Pre-order traversal: Figure 7.15 (a): ABDECFCG, Figure 7.15 (b): */+/ABCD

2. **In-order Traversal (LR'R):** In the in-order traversal of a binary tree is as follows:

- First, traverse the left sub-tree in in-order.
- Second, process the root node.
- Lastly, traverse the right sub-tree in in-order.

If the tree has an empty subtree the traversal is performed by doing nothing. That means a tree having NULL, sub-tree is considered to be completely traversed when it is encountered. The algorithm for the in-order traversal in a binary tree is given below:

Algorithm In-order (Node): The pointer variable ‘**Node**’ stores the address of the root. **Step 1: Is empty?**

```

If (empty [Node]) then
  Print “Empty tree” return
  
```

Step 2: Traverse the left subtree

```

If (Lchild [Node] ≠ NULL) then
  Call in-order (Lchild [Node])
  
```

Step 3: Process the root node

If (Node \neq NULL) then

Output: (Data [Node])

Step 4: Traverse the right subtree

If (Rchild [Node] \neq NULL) then

Call in-order (Rchild [Node])

Step 5: return at the point of call

Exit

Consider binary tree and binary arithmetic expression tree shown in Figure 7.16 (a) & (b).

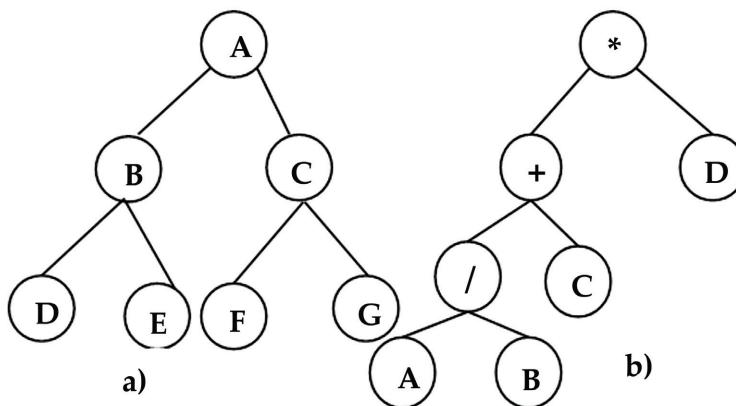


Figure 7.16: Binary Tree (a) and (b).

In-order traversal: Figure 7.16 (a): DBEAFCG, Figure 7.16 (b): A/B+C*D

3. **Post-order Traversal (LRR')**: post-order traversal of a binary tree is as follows:
 - First, traverse the left sub-tree in post-order.
 - Second, traverse the right sub-tree in post-order.
 - Lastly, process the root node.

If the tree has an empty subtree the traversal is performed by doing nothing. That means a tree having NULL sub-tree is considered to be completely traversed when it is encountered. The algorithm for the post-order traversal in a binary tree is given below:

Algorithm Post-order: The pointer variable ‘**Node**’ stores the address of the root node.

Step 1: Is empty?

If (empty [Node]) then

Print “Empty tree” return

Step 2: Traverse the left subtree

If (Lchild [Node] ≠ NULL) then

Call post-order (Lchild [Node])

Step 3: Traverse the right subtree

If (Rchild [Node] ≠ NULL) then

Call post-order (Rchild [Node])

Step 4: Process the root node

If (Node ≠ NULL) then

Output: (Data [Node])

Step 5: Return at the point of call

Exit

Consider binary tree and binary arithmetic expression tree shown in Figure 7.17 (a) & (b).

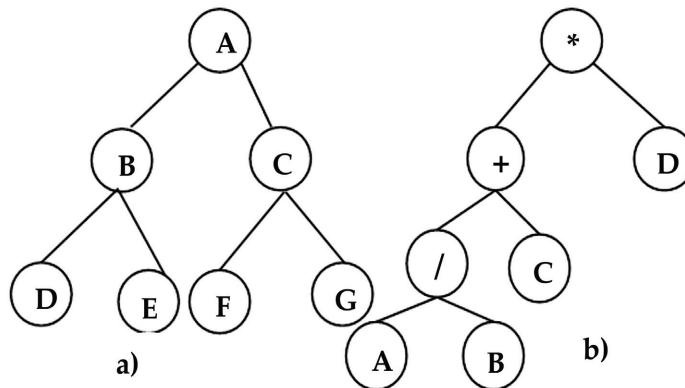


Figure 7.17: Binary Tree (a) and (b).

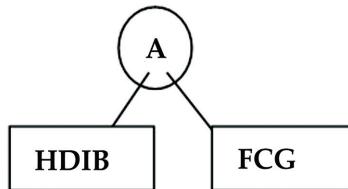
Post-order traversal: Figure 7.17 (a): DEBFGCA, Figure 7.17 (b): AB/C+D*

7.5.1. Creation of a Binary Tree using Tree Traversals

Now a question may come to our mind that is it possible to predict a tree from anyone traversal. To create the exact tree from the tree traversals we require any two traversals of the tree. Let us see the procedure of creating a binary tree from given traversals.

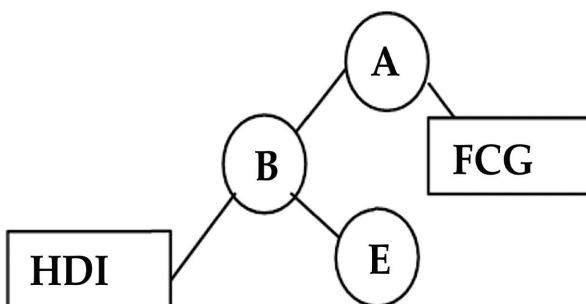
Postorder: H I D E B F G C A In order: H D I B E A F C G

Step 1: In Postorder (left, right and root) sequence the last node is the root node. In the above example, “A” is the root node. Now in order sequence locates the “A.” The left sequence to “A” indicates the left subtree and right sequence to “A” indicates the right subtree.



Step 2: In the left subtree these alphabets H, D, I, B, E observe the postorder and sequence in in order: Postorder: H I D E B In order: H D I B E

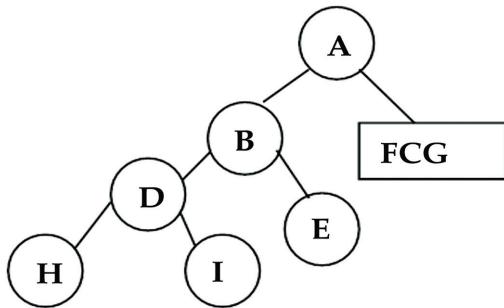
For postorder traversal root node is B, in order sequence locates the “B.” The left sequence to “B” indicates the left subtree and right sequence to “B” indicates the right subtree. Here B is the parent node; therefore pictorially tree will be, as shown in the figure.



Step 3: these alphabets H, D, I observe the postorder and sequence in order

Postorder: H I D In order: H D I

Here D is parent node; H is leftmost nodes and I is the right child of D node. So tree will be as shown in figure.



Step 4: Now we will solve for a right subtree of the root “A” with the alphabets F, C, G observe both the sequences,

Postorder: F G C

In order: F C G

C is the parent node, F is the left child and G is the right child. So finally the tree will be as shown in Figure 7.18.

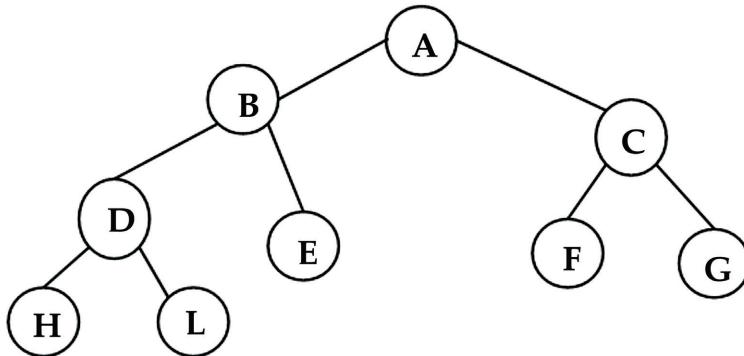


Figure 7.18: Finally Binary Tree.

7.5.2. Threaded Binary Tree

In the Linked representation of binary tree produces a large number of NULL pointers if nodes have either empty or one child. Thus for a node that does not have left child (left subtree) then its left child pointer field is set to NULL. Similarly, if a node does not contain right child (right subtree) then its right child pointer field is set to NULL. Thus it can be easily observed that in these more than that of actual pointers. The above discussion is illustrated in Figure 7.19. To solve these problems wasted of space for these NULL

pointers. The idea is to use these pointers to point some node in the tree. These NULL pointers are converted into useful linked called **threads** (optimizing NULL pointers, the concept of thread used). Thus, representation of a binary tree using these threads is called, threaded binary tree. Which node a NULL pointer should point is decided according to in-order traversal.

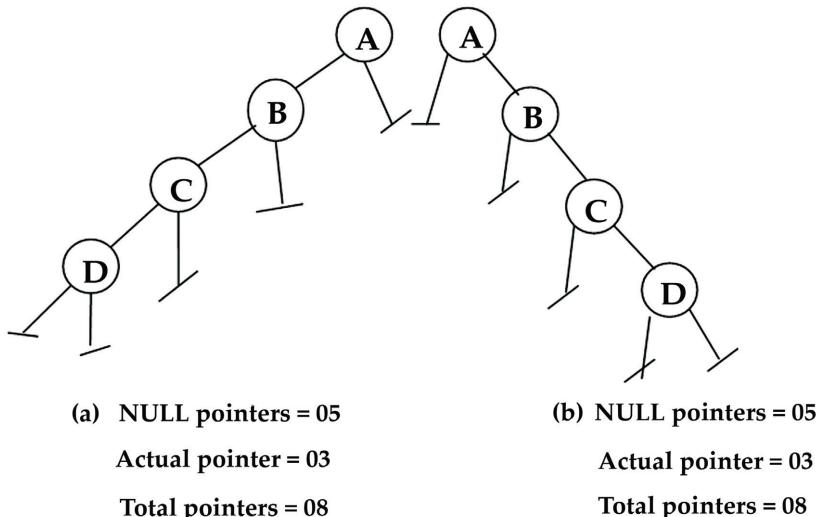


Figure 7.19: Computing pointer in a binary tree.

If the link of a node P is NULL, then this link is replaced by the address of the predecessor of P. similarly, if a right link is NULL, then this link is replaced by the address of the successor of node which would come after node P. Internally, a thread and a pointer, both are addressed. These can be distinguished by the assumption that a normal pointer is represented by positive addresses and threads are represented by negative addresses. Figure 7.20 shows a threaded binary tree where normal pointers and threads are shown by solid lines and dashed lines respectively.

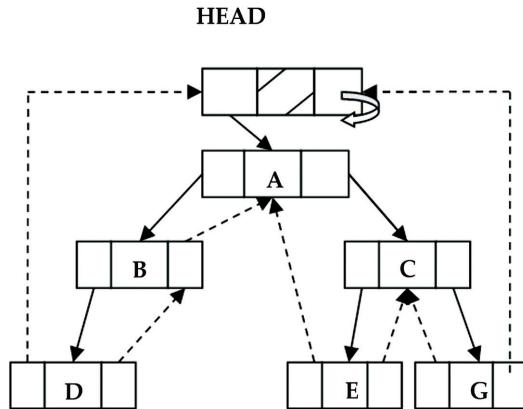


Figure 7.20: A threaded binary tree.

It is to note that by making little modification in the structure of a binary tree we can get the threaded tree structure, thereby distinguishing threads and normal pointer by adding two extra one-bit fields-child thread and child thread.

child thread = also,

rchild thread =

Advantages: 1. The in-order traversal of a threaded tree is faster than it's unthreaded.

2. threaded tree representation, it may be possible to generate the successor or predecessor of any arbitrarily selected node without having to incur the overhead of using the stack.

Disadvantages: 1. Threaded trees are unable to share common subtrees.

2. If negative addressing is not permitted in the programming language being used, two additional fields are required to distinguish between the thread and structural links.

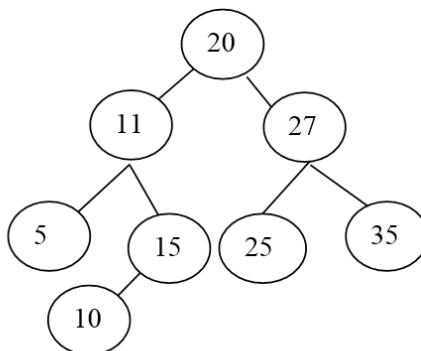
3. Insertions and deletions from a threaded tree are time-consuming since both thread and structural links must be maintained.

7.6. BINARY SEARCH TREE (BST)

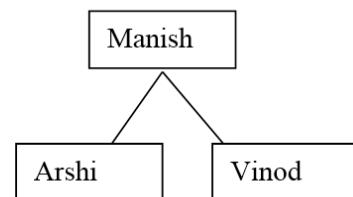
Binary search the tree is a representation of the tree in some specific manner. In a binary search tree, the data items are arranged in a certain order. The

order may be numerical, alphabetical. If the order is numerical then the left subtree of binary search tree contains those nodes that have less or equal numerical (or lexical) value than those associated with the root of the tree (or subtree). Similarly, right-subtree contains those nodes that have greater or equal numerical (or lexical) values than those associated with the root of the (or subtree). A binary search tree is a binary tree which is either empty or satisfies the following rules:

- The value of the key in the left child is less than the value of the root.
- The value of key in the right child is more than or equal to the value of the root.
- All the sub-trees of the left and right child observe by the two rules.



(a) The order is numerical BST



(b) Order is lexicographical

Figure 7.21: Binary Search Tree.

To search and find an element with an average running time $f(n) = O(\log_2 n)$. It also enables one to easily insert and delete elements as a comparison to the sequential search.

Exercise 7.1. Create the binary search tree from the following set of strings.

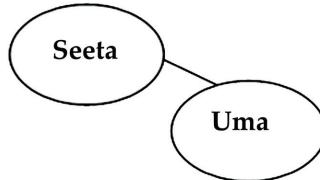
Seeta, Uma, Ram, Geeta, Tina, Arshi

Solution: we will start creating a node from left to right. We will compare these nodes based on alphabets.

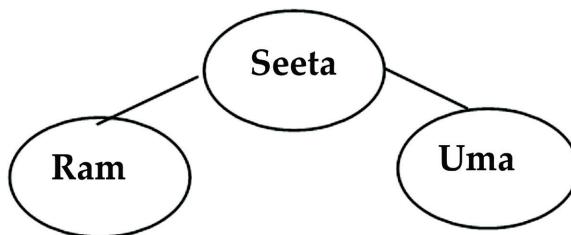
Step 1: first node Seeta,



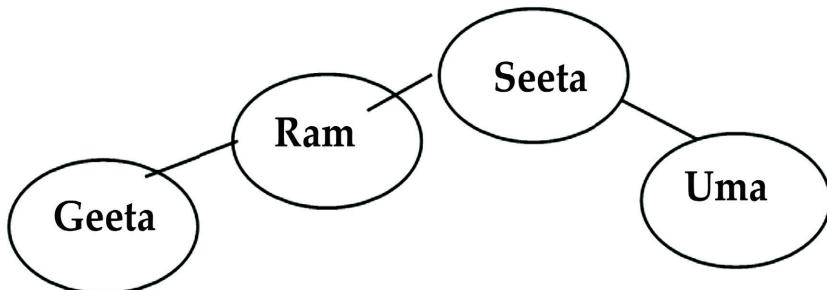
Step 2: next node Uma, as Uma > Seeta, attach Uma as a right child of Seeta.



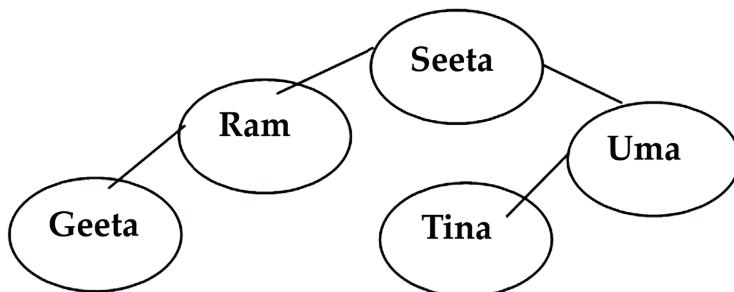
Step 3: next node Ram, as Ram < Seeta attach Ram as a left child of Seeta.



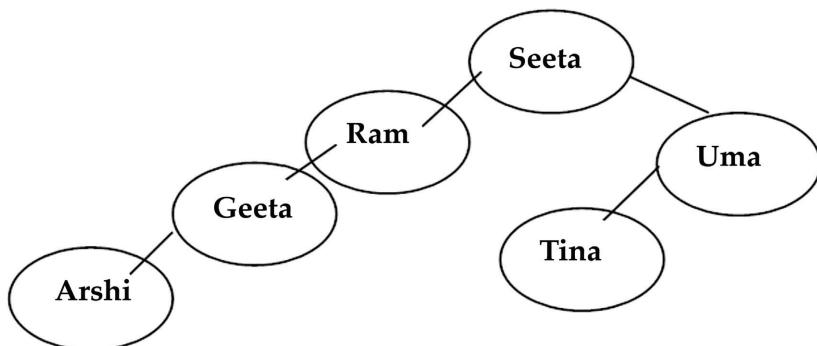
Step 4: next node Geeta, as Geeta < Ram attaches Geeta as a left child of Ram.



Step 5: next node Tina, as Tina < Uma attaché Tina as a left child of Uma.



Step 6: next node Arshi, as Arshi < Geeta attach Arshi as a left child of Geeta.



7.6.1. Operations on Binary Search Tree

1. **Searching:** The searching of the desired data item can be performed by branching into left or right-subtree until the desired data item (node) is reached. The search starts from the root node. If the tree is empty then the search is completed by doing nothing that means the search is unsuccessful. Otherwise, we compare the key 'K' of the desired data items from the key of the root. If 'K' is less than the key of the root then only the left-subtree is to be reached, as no data item in the right-subtree has the key value 'K.' if 'K' is greater than the key in the root then only the right subtree need to be searched. If 'K' equals to the key in the root then search terminates successfully. In a similar manner, the subtrees are also searched.

The algorithm of BST search: The pointer 'Root' stores the address of the root node and 'K' is the key of the desired data item to be searched.

Step 1: Checking, Is empty?

If (Root = 0), then

Print: “Empty tree”

Return 0

Step 2: if ‘K’ is equal to the value of the root node

If (Root[data] = K)

Print: “search is successful”

Return (Root[data])

Step 3: ‘K’ is less than the key value at root

If (K < Root[data])

Return (BST search (Root[lchild], K))

Step 4: ‘K’ is greater than the key value at root

If (K > Root[data])

Return (BST search (Root[rchild], K))

Exercise 7.2: Given the binary search tree, see figure 7.22 suppose we have to search a data item having key K = 15, then searching of the data item can be done by using the searching algorithm as follows.

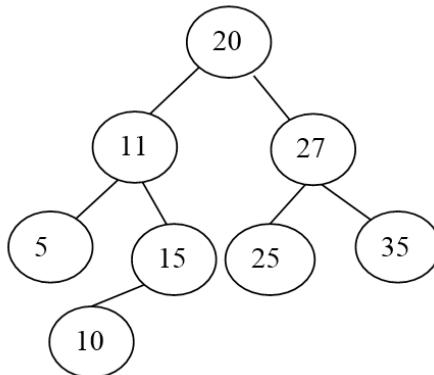


Figure 7.22: Binary Search Tree.

Solution: **Step 1:** Initially K = 15

Root[data] = 20

(K < Root[data]), so, Left subtree to be searched

Step 2: K = 15

Root[data] = 11

(K > Root[data]), so, Right subtree to be searched

Step 3: K = 15

Root[data] = 13

(K = Root[data]), so, Search is successful and it terminates.

2. **Insertion:** In binary search tree we do not allow any replica of the data items. So to insert a data item having key 'K' into a binary search tree, we must check that its key is different from those of existing data items by performing a search for the data item with the same key 'K.' If the search for 'K' is unsuccessful then the data item is inserted into a binary search tree at the point the search is terminated. While inserting the new data item having key 'K' three cases arise:

1. If the tree is empty then the new data item is inserted as the root node.
2. If the tree has only one node, root node, then depending upon the key value of the data item it is inserted in the tree.
3. If the tree is non-empty, has a number of nodes, then by comparing the value of the key the node is inserted. If 'K' is less than the root then it is inserted in left-subtree, otherwise, in right-subtree. The whole process is repeated until the appropriate place is obtained for the insertion.

The algorithm for the insertion of new data in the binary search tree is given below:

The algorithm of BST Insertion: The pointer 'Root' stores the address of the root node and 'new' points to the new node which store the 'K' is the key of the desired data item to be inserted.

Step 1: Checking, Is empty?

If (Root = NULL), then

Print: "Empty tree"

Set new [data] \leftarrow K

Set (rchild) \leftarrow NULL

Set new [lchild] \leftarrow NULL

Set Root \leftarrow new

Step 2: Inserting node ‘new’ into a tree having single node

```
If (new [data] < Root[data]) then
    Set Root [lchild] ← new
    Set Root [rchild] ← NULL
Else
    Set Root ← lchild ← NULL
    Set Root ← rchild ← new
```

Step 3: Inserting node ‘new’ into a tree having more nodes

```
While (Root ≠ NULL) {
    If (Root [data] < new [data]) then {
        If (R [lchild] = NULL) then {
            Set Root [lchild] ← new
            Set Root ← NULL }
        Else
            Set Root ← Root[lchild]
        Else if (Root [rchild] = NULL) then {
            Set Root [rchild] ← new
            Set Root ← NULL }
        Else
            Set Root ← Root[rchild]}
```

Step 4: return to the point of call

Return

The insertion of the new data item into a binary search tree is performed in O (h) time where ‘h’ is the height of the tree.

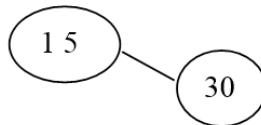
Exercise 7.3: Suppose T is an empty binary search tree, now we have to insert the following five data items into the binary search tree: 15, 30, 12, 40, 35

Solution: **Step 1:** Insertion 15 So, the node becomes the root node as the tree is empty.

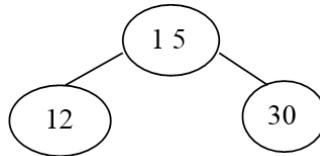


Step 2: insertion 30,

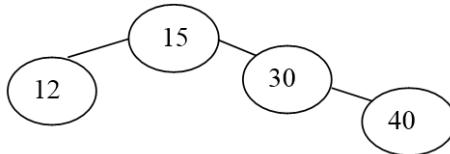
First checking with the root node $30 > 15$, So, it is inserted at the right of the root node.



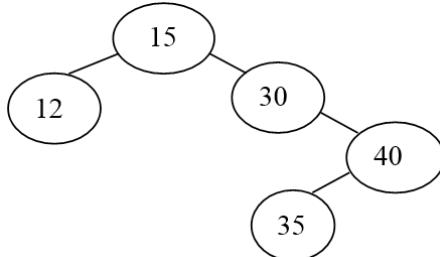
Step 3: insertion 12, Now checking with the root node $12 < 15$ So, it is inserted at the left of the root node



Step 4: insertion 40, Checking with root node $40 > 15$, So, it is inserted at the right subtree of the root node, Checking with the root node of the right subtree $40 > 30$ So, it is inserted at the right.



Step 5: Insertion 35, Checking with root node $35 > 15$, So, it is inserted at the right subtree of the root node, Checking with the root node of the right subtree $35 > 30$, So, it should be in the right subtree, but $35 < 40$ So, it is inserted in its left subtree.



3. Deletion: In a binary search tree for deleting a particular node from the binary search, it is searched first by the searching algorithm discussed previously. If the search is not successful then the algorithm terminated. Otherwise, from a binary search tree, there are

three cases which are possible for the node delete that contain the data items.

Case 1: Deletion of Leaf Node

Consider a figure 7.23, in this ‘delete,’ is the left leaf node which has to be deleted. The only task for the deletion of this node is discarding the leaf node, to set its parent left child pointer to NULL.

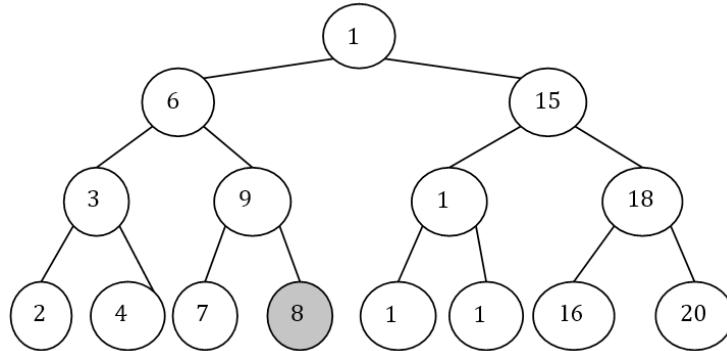


Figure 7.23: Before deletion.

Above tree, we want to delete the node having value 8 then we will set the right pointer of its parent node as NULL. That is a right pointer of the node having value 9 is set to NULL.

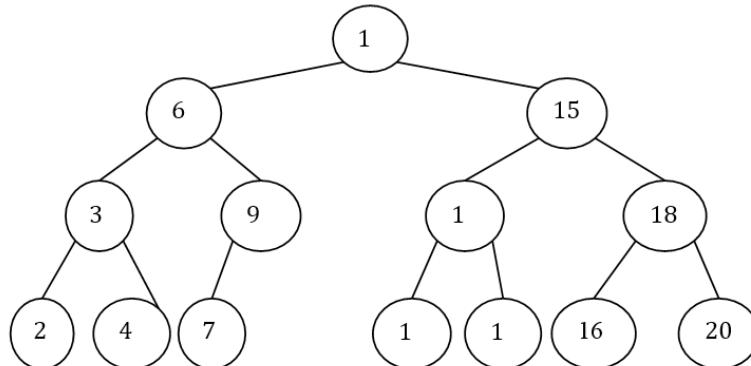


Figure 7.24: After deletion.

The algorithm of BST deletion of leaf node: The procedure deletes the node pointed by ‘del’ from the binary search tree.

Step 1: a Searching leaf node

Call to BST search (R, del)

Step 2: deletion leaf node

```
if (x (lchild) = NULL and (x (rchild) = NULL) then {  
    if (parent (lchild) = x) {  
        set parent (lchild) = NULL) }  
    else  
        set parent (lchild) = NULL)  
}
```

Step 3: free the node

Freenode (x).

Case 2: Deletion of a node having one child

Figure 7.25, in this, delete the dark point’s node which has exactly one non-empty tree.

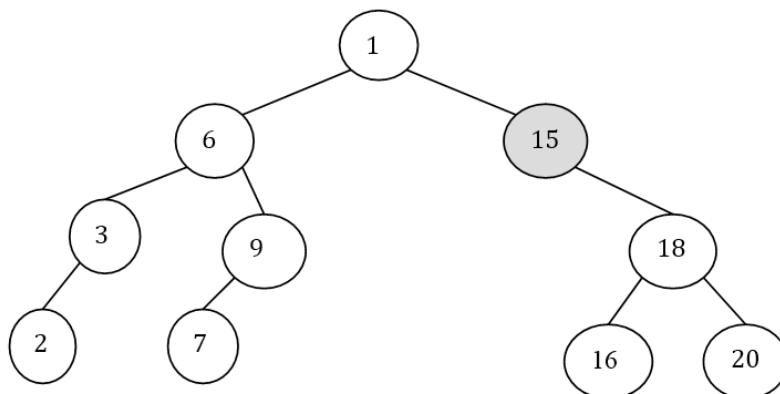


Figure 7.25: Before deletion.

If we want to delete the node 15, then we will simply copy node 18 of 15 and then set the node tree. If delete node that has a right child then right child pointer value is assigned to the right child value of its parent, but delete a node that has left child then left child pointer value is assigned to the left child value of its parent.

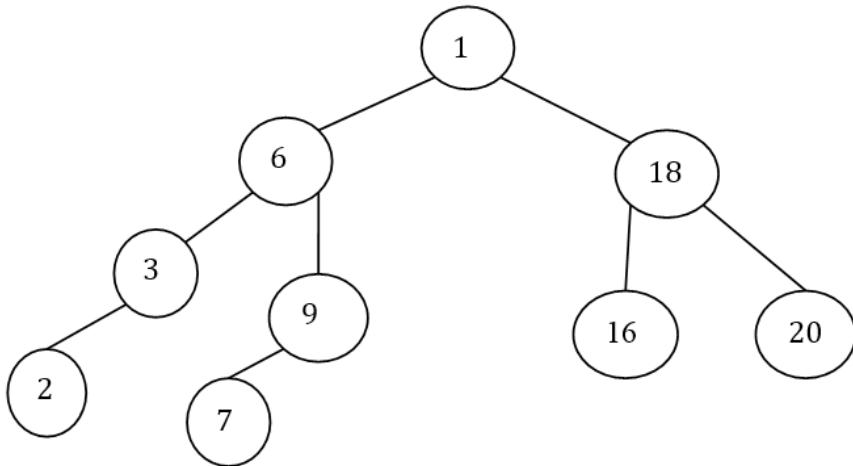


Figure 7.26: After deletion of a node from the tree.

The algorithm of BST Deletion of a node having one child: The procedure deletes the node pointed by ‘del’ from the binary search tree which has exactly one non-subtree.

Step 1: if the node pointed by ‘del’ has only right-subtree

```

if (del (lchild) = NULL) then {
  if (parent (lchild) = del) then {
    set parent (lchild)← del (rchild) }
  else
    set parent (rchild)← del (rchild) }
```

Step 2: if node pointed by ‘del’ has only left-subtree

```

if (del (rchild) = NULL) then {
  if (parent (lchild) = del) then {
    set parent (lchild)← del (lchild) }
  else
    set parent (rchild)← del (lchild) }
```

Step 3: free the node

Freenode (del).

Case 3: Deletion of a node having two children

Consider Figure 7.27, delete the dark point’s node which has exactly two non-subtrees. We want to delete node having value 6.

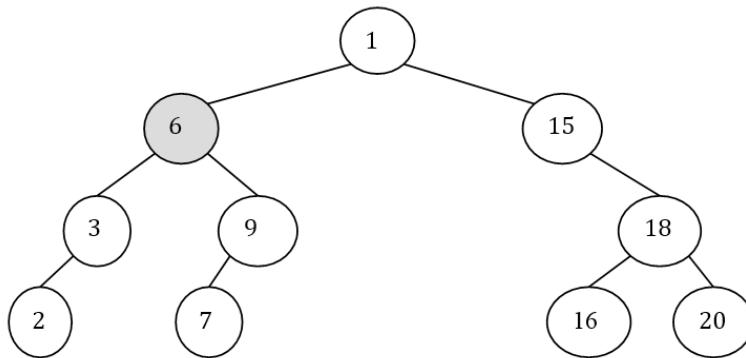


Figure 7.27: Before deletion.

We will then find out the inorder successor of node 6. The inorder successor will be simply copied at the location of node 6. That means copy 7 at the position where the value of the node is 6. Set left pointer of 9 as NULL. This completes the deletion procedure.

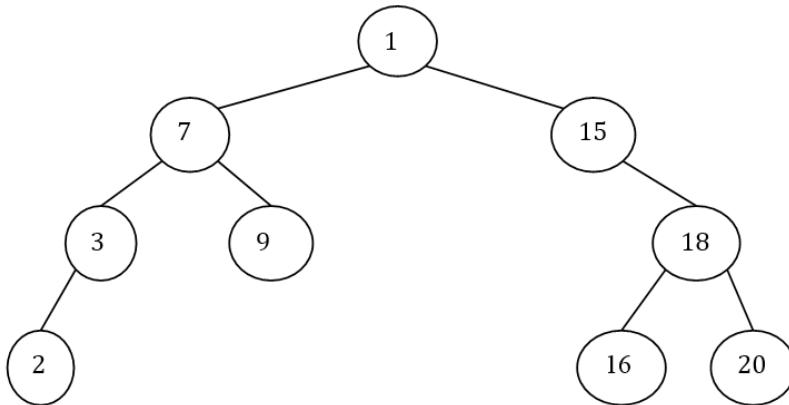


Figure 7.28: After the deletion of a node from the tree.

The algorithm of BST Deletion of a node having two children: The procedure deletes the node pointed by ‘del’ from the binary search tree which has exactly two non-subtrees.

Step 1: Initialization

Set parent \leftarrow del

Set inos \leftarrow del (rchild)

Step 2: loop, finding in order successor

```

while (inos (lchild) ≠ NULL {
    Set parent ← inos
    Set inos ← inos (lchild) }

```

Step 3: substituting in order successor to appropriate place

```

Set del (data) ← inos (data)
Set del ← inos

```

Step 4: return the node to the free storage pool

```
Freenode (del).
```

7.6.2. Red-Black Tree

Generally, a red-black tree is a kind of self-balancing binary search tree. The innovative structure was discovered in 1972 by Rudolf Bayer who called them “symmetric binary B-trees,” but acquired its modern name in a paper in 1978 by Leo J. Guibas and Robert Sedgewick. It is difficult but has good worst-case running time for its operations and it is efficient in performing: it can search, insert and deleted in $O(\log n)$ time, where n is the number of elements in the tree.

As the red-black tree is also a binary search tree, they must assure the restriction that every node contains a value greater than or equal to all nodes in its left subtree, and less than or equal to all nodes in its right subtree. Its color, which can be either Red or Black, tree node can be colored on any path from the root to leaf, red-black tree ensures that no such path is more than twice as long as any others, so that the is approximately balanced. Each node of the tree now contains the fields color, key, left, and right. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains value NIL.

Properties: A red-black tree is a binary search tree where each node has a color feature. The value of which is either red or black. We use the following properties to make any suitable red-black tree.

- Each node is colored either black or red.
- The root is black.
- Every leaf is black.
- If a node is red, then both children of each red node are black.
- Each leaf-nil node, known as the external node is colored black.

- Every simple path from any given node to its leaf nodes contains the same number of black nodes.

R- Red node B- Black node External Nodes

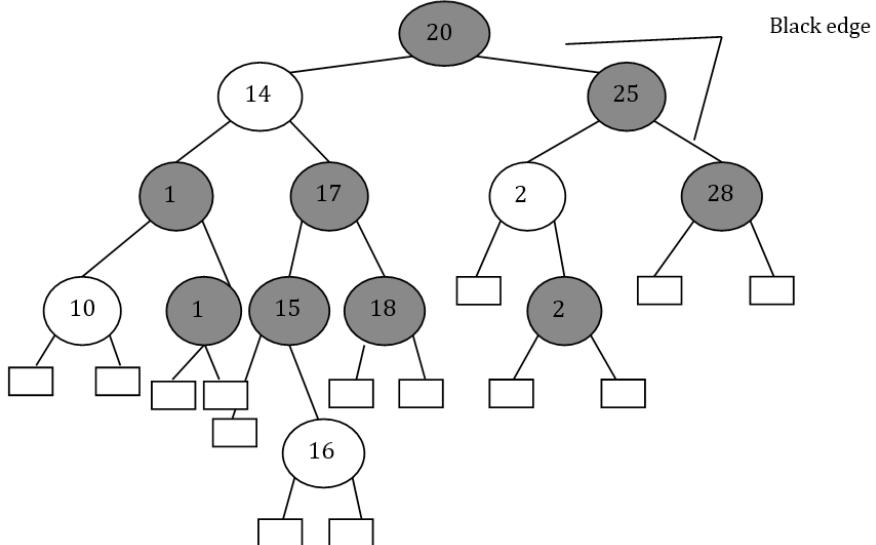


Figure 7.29: Red-Black Tree.

Insertion Operation in Red-Black Tree

The purpose of the insert operation is to insert a new node (key) K into a tree T, following red-black tree properties. A particular case is required for an empty tree. If T is empty, replace it with a single black node containing K. These ensure that the root property is fulfilled.

If T is a non-empty tree, then we perform the following steps:

- 1. Use the Binary Search Tree insert algorithm to add new node K to the tree.
- 2. Use red-black tree properties for the new node.

In step 2 inserting a new node into a non-empty tree, what we need to do

will depend on the color of node K's parent. Consider P node be K's node parent. We need to judge two cases:

Case 1: if K's parent P is black

If K's parent P is black, then the addition of K did not use in the red property being violated.

Case 2: if K's parent P is red

If K's parent P is red, then P now has a red child, which violates the red property. Note that P's parent, G, (K's grandparent) must be black (why?).

This type of condition handle by the double-red situation, we will need to consider the color of G's other child, that is, P's sibling, S.

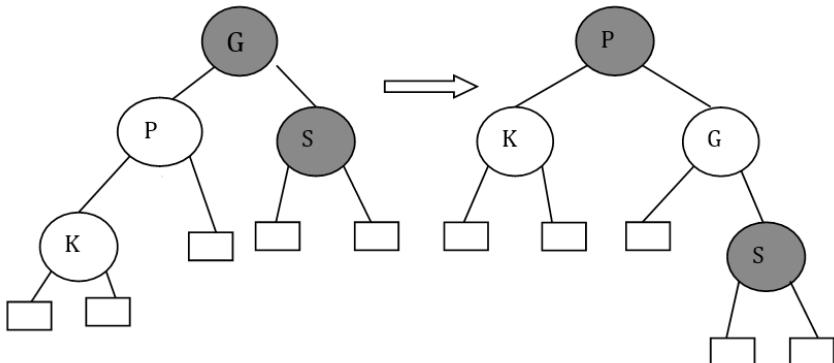
We have two cases:

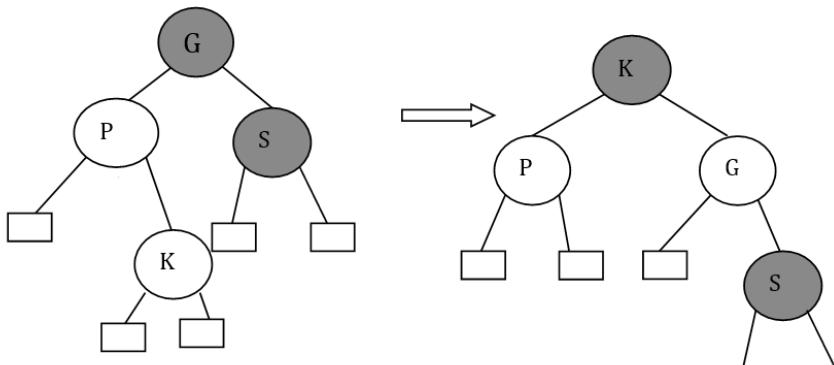
Case 2a: if P's sibling S is black or null

If P's sibling S is black or null, then we will do a triode restructuring of K (the newly added node), P (K's parent), and G (K's grandparent).

To do a restructuring, we first put K, P, and G in order; let's call this order A, B, and C. We then make B the parent of A and C, color B black, and color A and C red. We also need to make sure that any subtrees of P and S (if S is not null) end up in the appropriate place once the restructuring is done. There are four possibilities for the arranging of K, P, and G.

The way a restructuring is done for each of the possibilities is shown below.





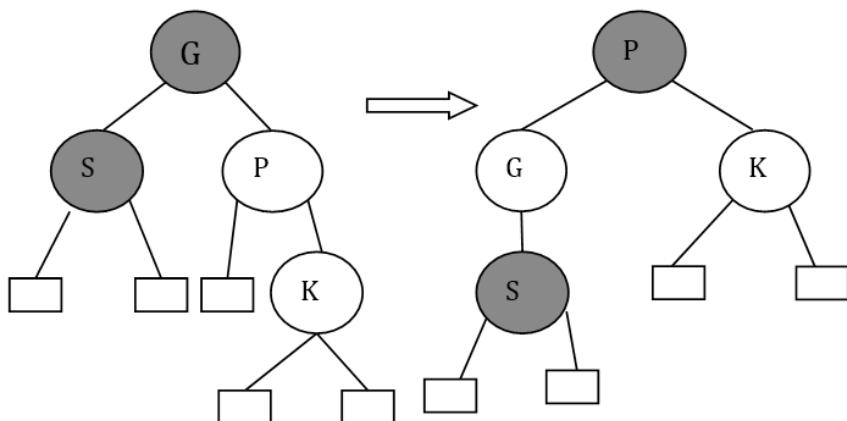
Before restructuring

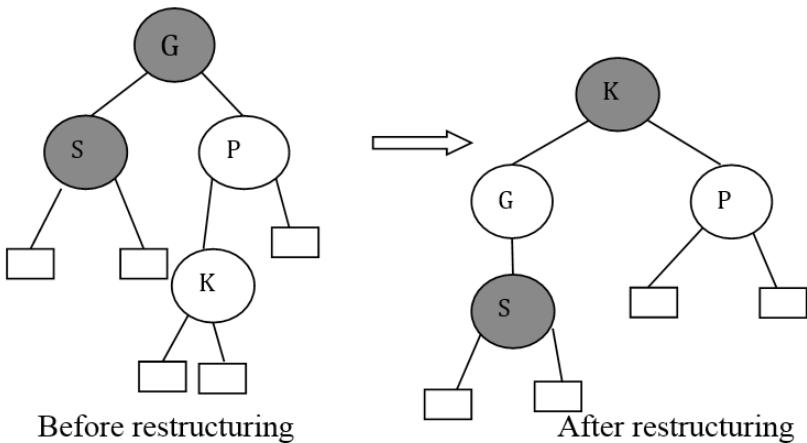
After restructuring

The first two possibilities are:

If S is null, then in the pictures above, S would be replaced with nothing (i.e., null).

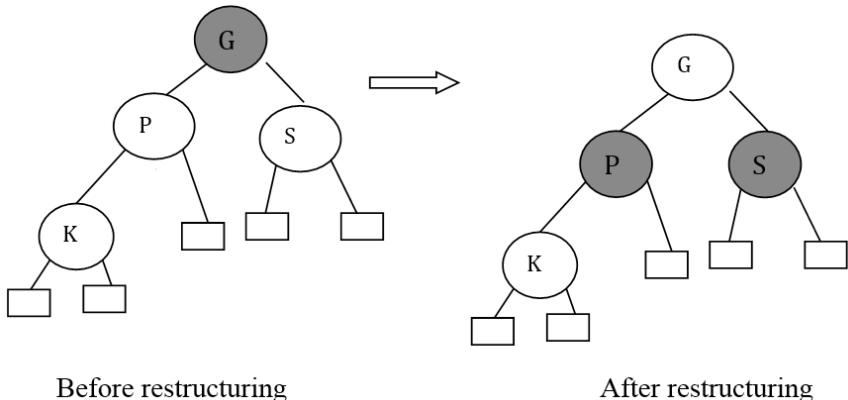
The second two possibilities are mirror-images of the first two possibilities:





Case 2b: P's sibling S is red, If P's sibling S is red, then we will do a recoloring of P, S, and G: the color of P and S is changed to black and the color of G is changed to red (unless G is the root, in which case we leave Gblack to preserve the root property).

Recoloring does not affect the black property of a tree: the number of black nodes on any path that goes through P and G is unchanged when P and G switch colors (similarly for S and G). But, the recoloring may have introduced a double-red situation between G and G's parent. If that is the case, then we recursively handle the double-red situation starting at G and G's parent (instead of K and K's parent).



7.7. HEIGHT BALANCED (AVL) TREE

Balance trees are useful data structures that are useful to store the data at the desired location or at a specific location. The time to search an element in a binary search tree is defiantly limited by the height (or depth) of the tree. Each step in the search goes down one level, so in the absolute worst case, we will have to go all the way from the root to the deepest leaf in order to find the element (X) or to find out that X is not in the tree. So we can say with certainty that search is $O(\text{Height})$. Height balanced tree solves the depth problem in searching the skewed binary tree. As compared to a simple binary tree, the balanced search trees are more efficient because the insertion or deletion of nodes in this data structure requires $O(\log n)$ time.

AVL TREE: Adelson Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to the height of subtrees. The tree can be made balanced and because of this retrieval of any node can be done in $O(\log n)$ times, where n is a total number of nodes. From the name of these scientists, the tree is called the AVL tree.

An empty tree is height balanced if T is a non-empty binary tree with T_L and T_R as its left and right subtrees. The T is height balanced if and only if

- T_L and T_R are height-balanced.
- The height of the left $h_L - h_R$ height of right ≤ 1 where h_L and h_R are the height of T_L and T_R .

Balanced Factor: The balance factor BF (T) of a node in the binary tree is defined to $h_L - h_R$ where h_L and h_R are the height of the left and right subtrees of T.

Balance Factor of Node = Height of Left – Height of Right

For any node in the AVL tree, the balanced factor i.e., BF (T) is $-1, 0, 1$.

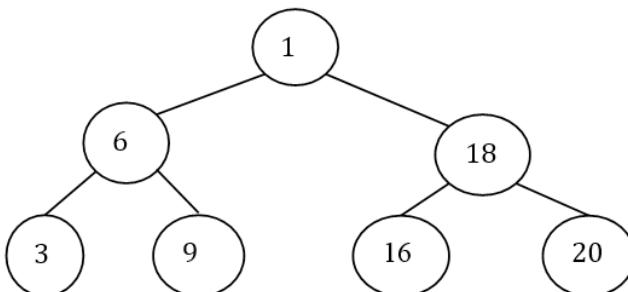


Figure 7.30: AVL tree.

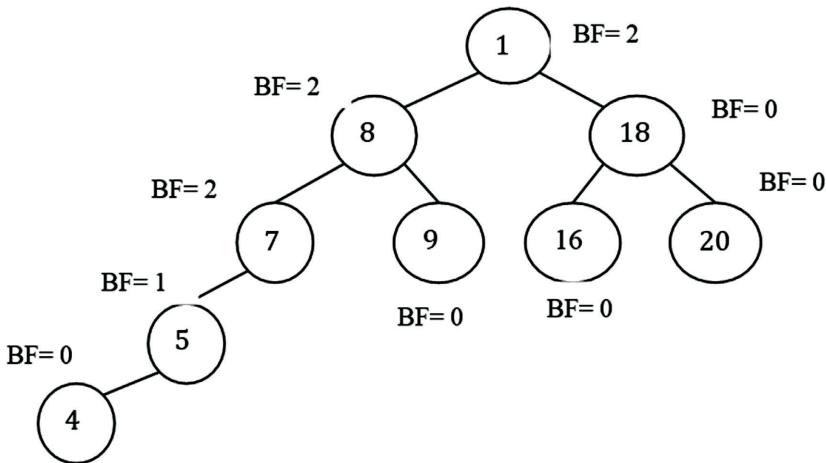


Figure 7.31: Not an AVL tree.

7.7.1. Operation on AVL Tree

The AVL tree follows the property of binary search tree. In fact, AVL trees are basically binary search trees with balance factor as $-1, 0, 1$. After insertion of any tree if the balanced factor of any node becomes other than $-1, 0, 1$ then it is said that AVL property is violated.

INSERTION: There are four different cases when rebalancing is required after insertion of a new element or node.

1. An insertion of a new node into a left subtree of the left child (LL).
2. An insertion of a new node into a right subtree of the left child (LR).
3. An insertion of a new node into a left subtree of the right child (RL).
4. An insertion of a new node into a right subtree of the right child (RR).

Some modification is done on AVL tree in order to rebalance it is called **rotations** of AVL tree. These are a classification of rotations as shown in Figure 7.32:

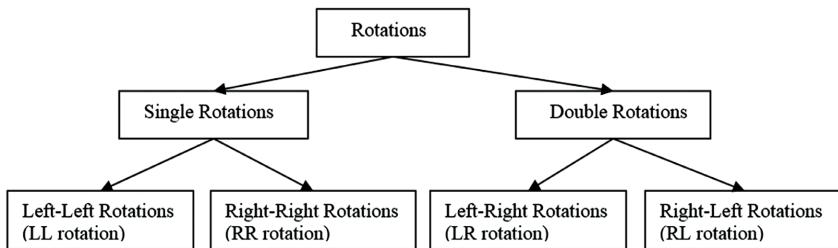


Figure 7.32: Classification of Rotation.

Insertion in an AVL search tree is a binary search tree, thus the insertion of the data item having key 'K' in AVL search tree is same as performed in the binary search tree. The insertion of the data item with key 'K' is performed at the leaf, in which three cases arise.

- If the data item with 'K' is inserted into an empty AVL search tree, then the node with key 'K' is set to be the root node. In this case, the tree is balanced.
- If the tree contains an only a single node, root node, then the insertion of the node with key 'K' depends upon the value of 'K.' If 'K' is less than the key value of the root then it is appended to the left of the root. Otherwise, the greater value 'K' appended to the right of the root. In this case, the tree is height-balanced.
- If an AVL search tree contains a number of nodes (which are height balanced), then, in this case, has to be taken from inserting data item with key 'K.' So that after insertion the tree must be height-balanced.

We have noticed that insertion may cause the unbalanced tree. So, rebalancing of the tree is performed for making it balanced. The rebalancing is accomplished by performing four kinds of rotations.

Left-Left (L-L) Rotation: Given the AVL search tree is shown in Figure 7.33 after inserting a node with value 15 the tree becomes unbalanced. So, by performing LL rotation tree become balanced.

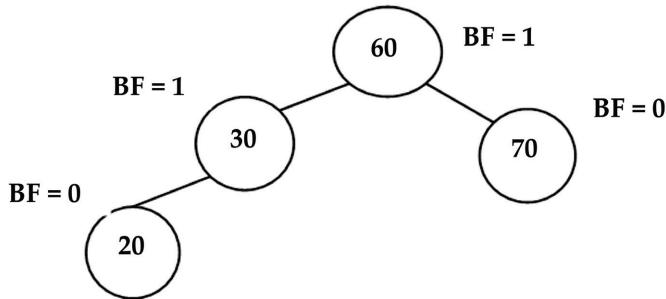
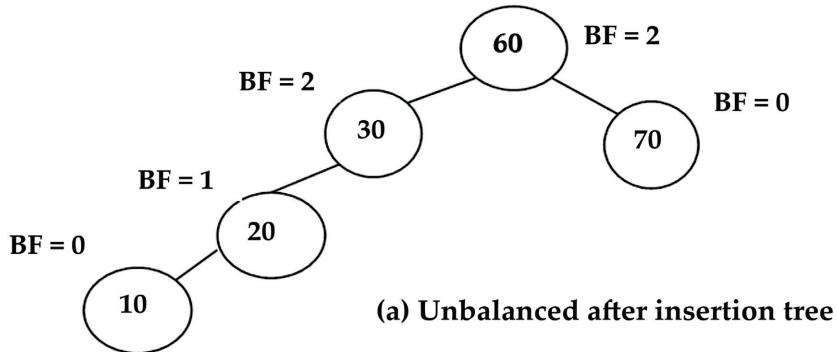


Figure 7.33: Balanced AVL search tree.

After inserting new node 10 tree, Figure 7.33 become unbalanced, so performing LL rotation tree become balanced as shown in Figure 7.34.



After performing LL rotation,

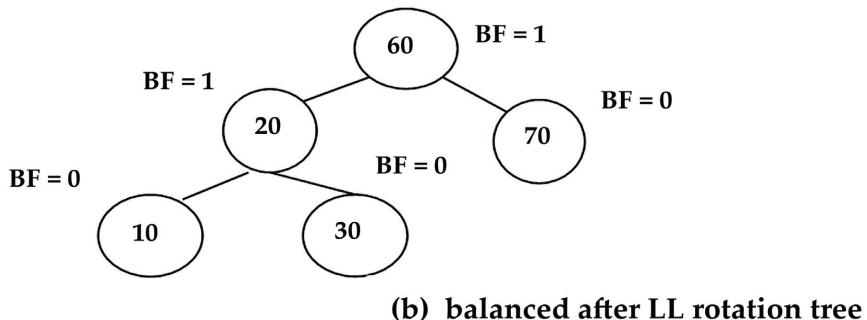


Figure 7.34: AVL search tree after performing LL rotation.

Right-Right (RR) Rotation: Given the AVL search tree is shown in figure 7.35 after inserting a node with value 75 the tree becomes unbalanced. So, by performing RR rotation tree become balanced.

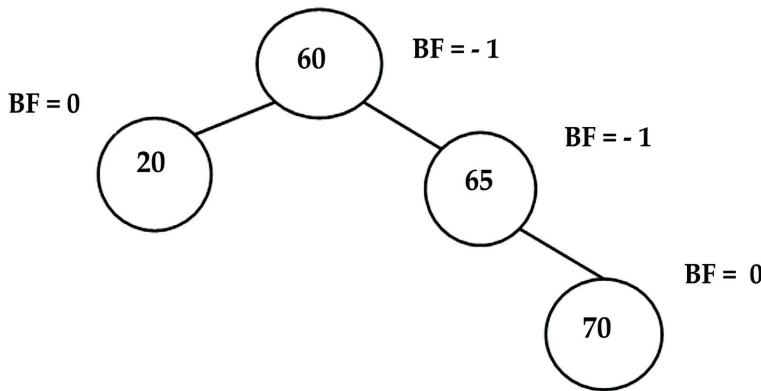
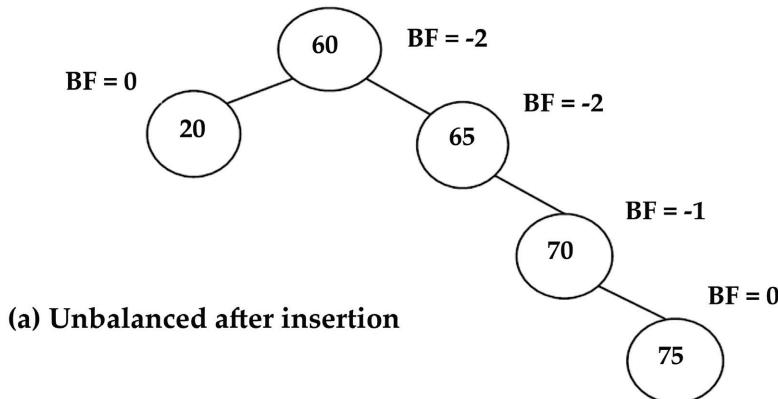


Figure 7.35: Balanced AVL search tree.

After inserting new node 75 tree, Figure 7.35 become unbalanced, so performing RR rotation tree become balanced as shown in Figure 7.36.



After performing RR rotation,

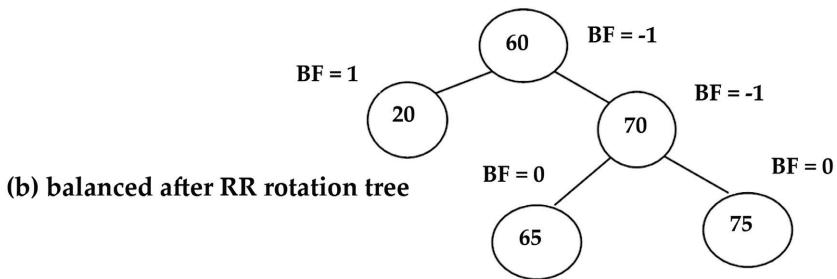


Figure 7.36: AVL search tree after performing RR rotation.

Left-Right (LR) Rotation: Given the AVL search tree is shown in Figure 7.37 after inserting a node with value 25 the tree becomes unbalanced. So, by performing LR rotation tree become balanced.

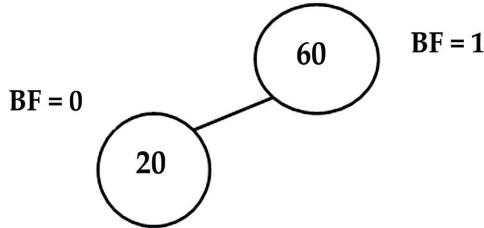
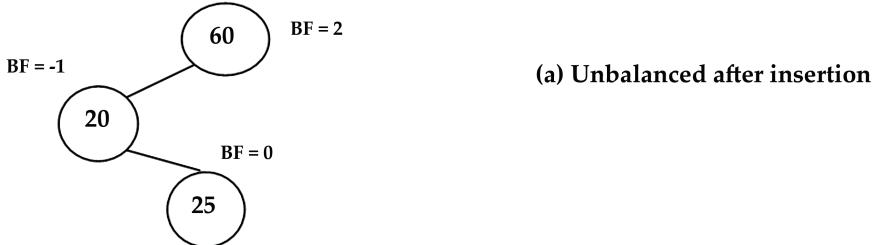


Figure 7.37: Balanced AVL search tree.

After inserting new node 25 tree, Figure 7.37 become unbalanced, so performing LR rotation tree become balanced as shown in Figure 7.38.



After performing LR rotation,

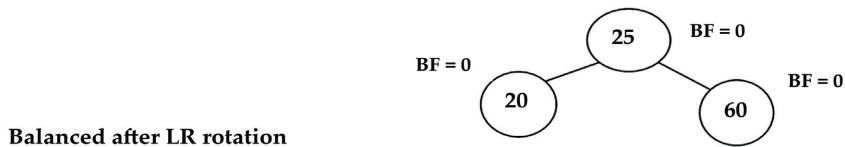


Figure 7.38: AVL search tree after performing LR rotation.

Right-Left (RL) Rotation

Given the AVL search tree is shown in Figure 7.39 after inserting a node with value 25 the tree becomes unbalanced. So, by performing RL rotation tree become balanced.

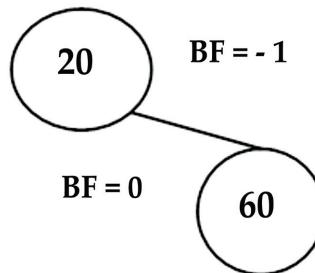
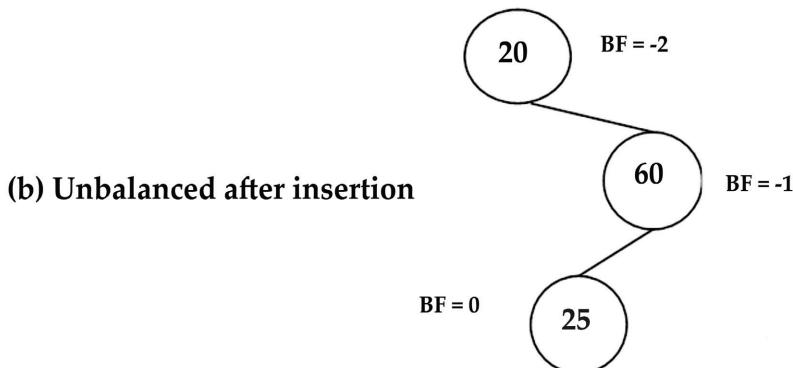
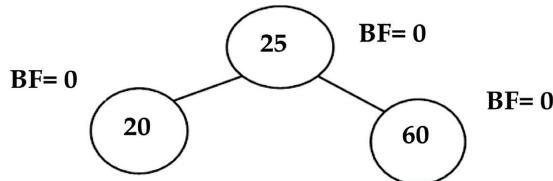


Figure 7.39: Balanced AVL search tree.

After inserting new node 25 tree, Figure 7.39 become unbalanced, so performing LR rotation tree become balanced as shown in Figure 7.40.



After performing RL rotation,

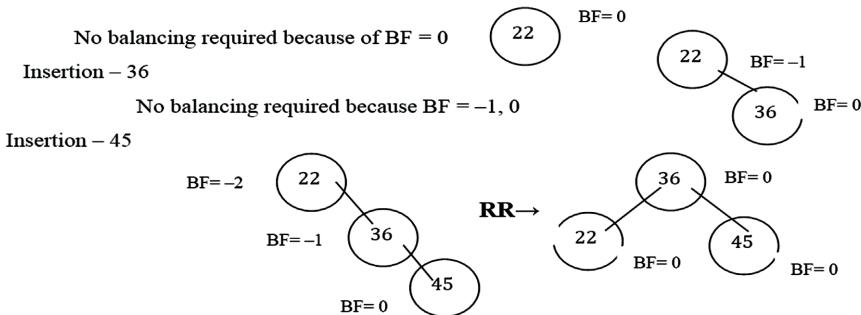


(b) balanced after LR rotation tree

Figure 7.40: AVL search tree after performing RL rotation.

Exercise 7.4: Creation of an AVL search tree is illustrated from the given set of values: 22, 36, 45, 50, 60, 57, 56, 55.

Solution: Insertion – 22

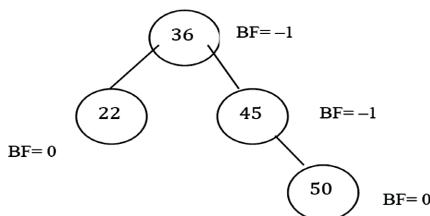


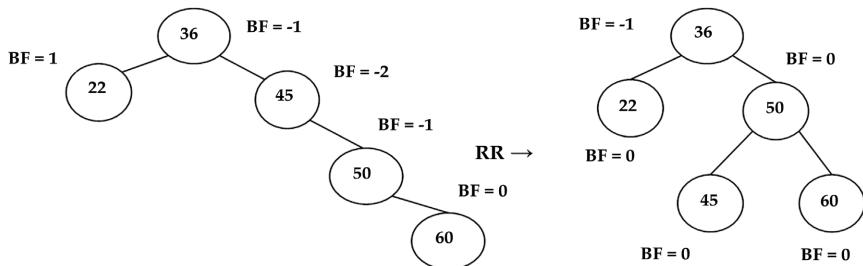
Balancing required because BF = -2, -1, 0 **Right-Right rotation** for balanced tree

Insertion – 50

No balancing required

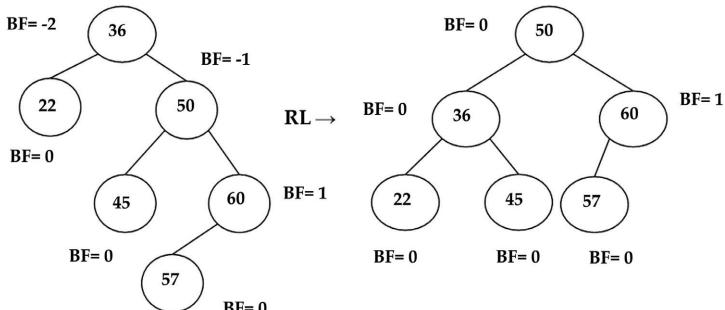
Insertion – 60





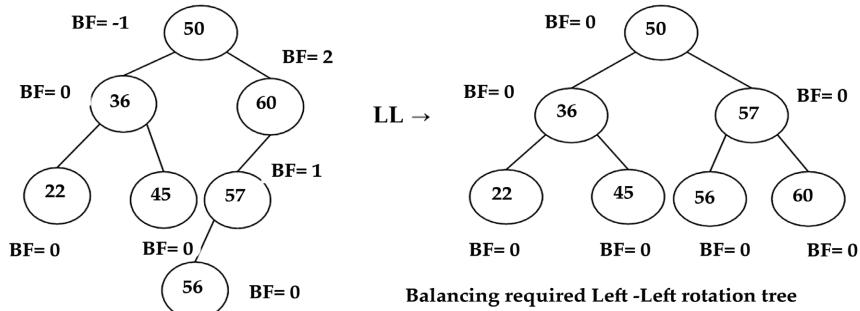
Balancing required Right - Right rotation

Insertion - 57



Balancing required Right Left rotation

Inserting - 56



Insert - 55

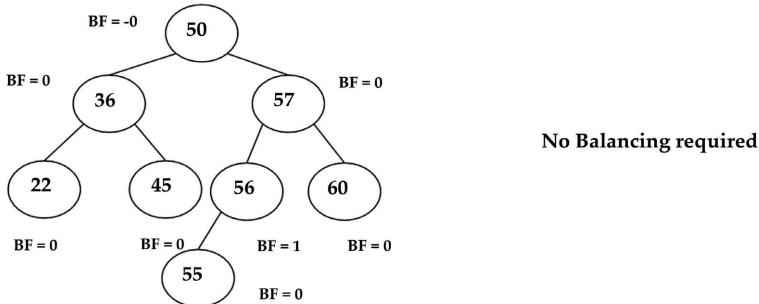


Figure 7.41: AVL tree Exercise of 7.4.

DELETION: In AVL tree deletion of any particular node from AVL tree, the tree has to be reconstructed in order to preserve AVL property and various rotations need to be applied for balancing the tree.

Algorithm for Deletion

Deletion algorithm is more complex than insertion algorithm.

1. Search the node which is to be deleted.
2. A) If the node to be deleted is a leaf node then simply make it NULL to remove.
B) If the node to be deleted is not a leaf node i.e., node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove the node.
3. Now we have to traverse back up the path towards the root, checking the balance factor of every node along the path. If we encounter unbalancing in some subtree then balance that subtree using appropriate single or double rotation.

SEARCHING: The searching of a node in an AVL tree is very simple. As AVL tree is basically a binary search tree, the algorithm used for searching a node from a binary search tree is the same one is used to search a node from AVL tree. The searching of a node takes $O(\log n)$ time to search any node.

7.7.2. Weight Balanced Tree

Weight balanced tree is a tree whose each node has an information field which contains the name of the node and number of times the node has been visited.

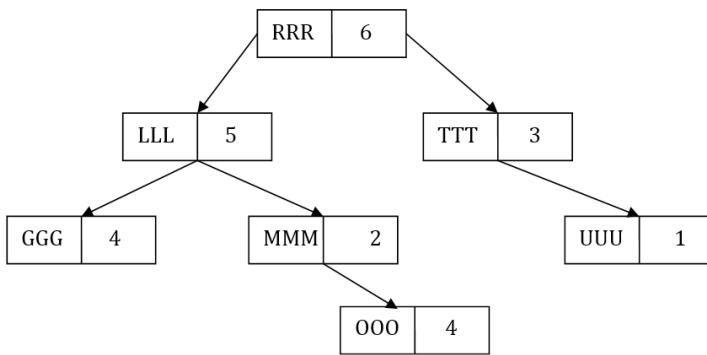


Figure 7.42: Weight Balance Tree.

For example, consider the tree given in Figure 7.42. This is a balanced tree, which is organized according to the number of accesses. The rules for putting a node in a weight balanced tree are expressed recursively as follows:

1. The first node of the tree or subtree is the node with the highest count of a number of times it has been accessed.
2. The left subtree of the tree is composed of nodes with values lexically less than the first node.
3. The right subtree of the tree is composed of nodes with value lexically higher than the first node.

7.8. B-TREES

B-Trees are balanced trees and a specialized multiway (m -way) tree is used to store the records in a disk. There is a number of subtrees to each node. The height of the tree is relatively small. That only small number of nodes must be read from the disk to retrieve an item. The goal of B-trees is to get fast accesses of the data. B-trees try to minimize the disk accesses, as disk accesses are expensive.

Multiway search tree: A multiway search tree of order m is an ordered tree where each node has at the most m children. If there are n numbers of children in a node then $(N - 1)$ is the number of keys in the node. The **B-tree** is of order ' m ' if it satisfies following conditions:

1. The root node should have at least two children.
2. Except for the root node, each node has at most m children and at least $m/2$ children.

3. The entire leaf node must be at same level. There should be no empty subtree above the level of the leaf nodes.
4. In the order of tree **m**, that means **m-1** keys are allowed.

7.8.1. Operation on B-Trees

1. **Insertion:** First search the place where the element or record must be put is done. If the node can accommodate the new record insertion is simple. The record is added to the node with an appropriate pointer so that number of points remain one more than the number of records. If the node overflows because there is an upper bound on the size of a node, splitting is required.

The node is split into three parts. The middle record is passed upward and inserted into the parent, leaving two children behind where there were once before. Splitting may propagate up the tree because the parent into which a record to be split in its child node, may overflow. Therefore, it may also split. If the root is required to be split, a new root is created with just two children, and the tree grows taller by one level.

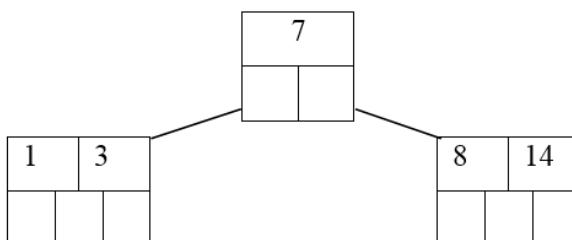
Exercise 7.5: Construct a B-Tree of order 5 following numbers.

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20.

The order 5 means at the most 4 keys are allowed. The internal node should have at least 3 non-empty children and each leaf node must contain at least 2 keys. Step 1: Insert 3, 14, 7, 1 as follows.

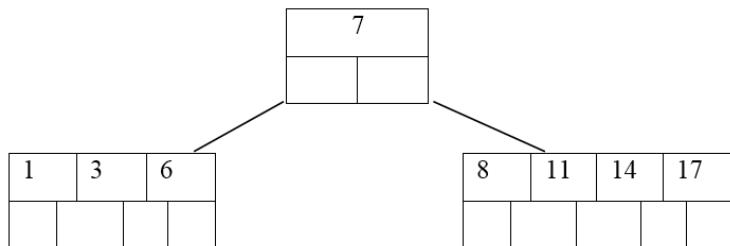
1	3	7	14

Step 2: insert next element 8 then we need to split the node 1, 3, 7, 14 at medium. Hence,

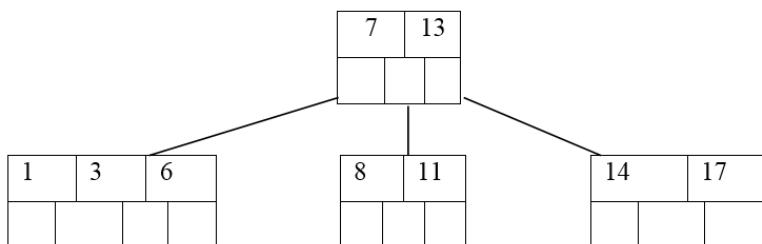


Here 1 and 3 are < 7 so these are at the left branch, node 8 and 14 > 7 so these are at right.

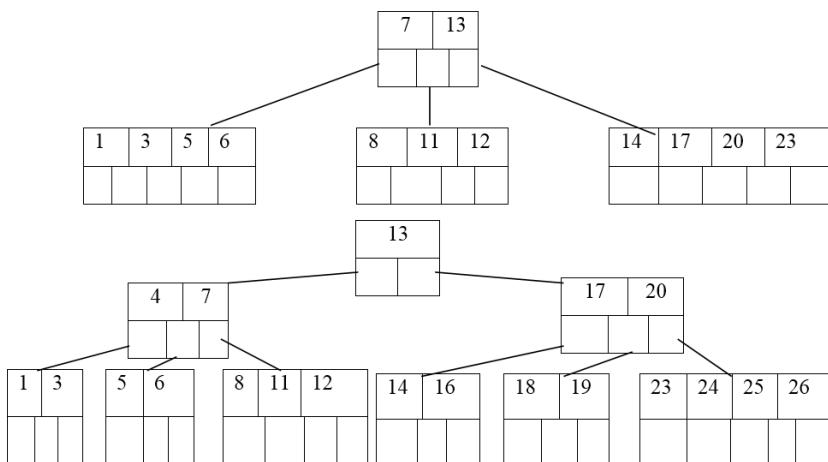
Step 3: insert 5, 11, 17 which can be easily inserted in a B-tree.



Step 4: Insert the next element 13. But if we insert 13 then the leaf node will have 5 keys which are not allowed. Hence 8, 11, 13, 14, 17 is split and medium node 13 is moved up



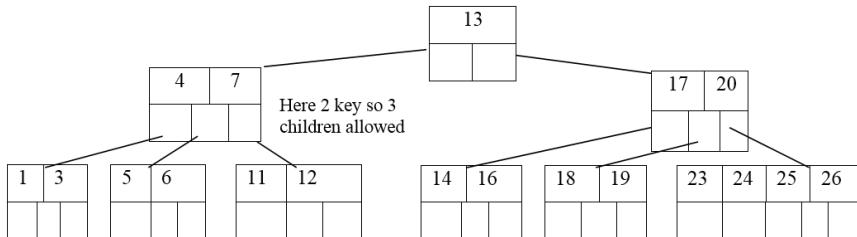
Step 5: insert 6, 23, 12, 20 without any split.



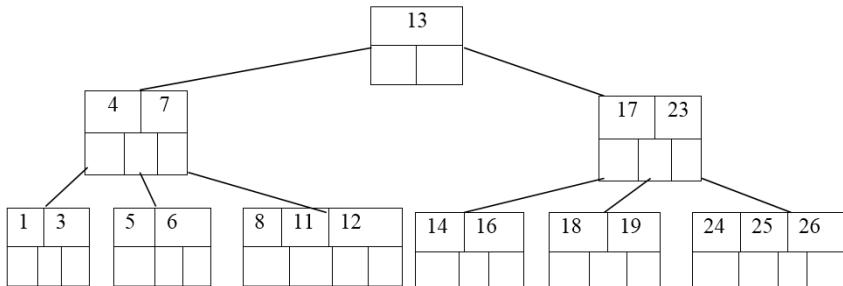
2. Deletion

As in the insertion method, the record to be deleted is first searched for. If the record is in a terminal node, deletion is simple. The record along with

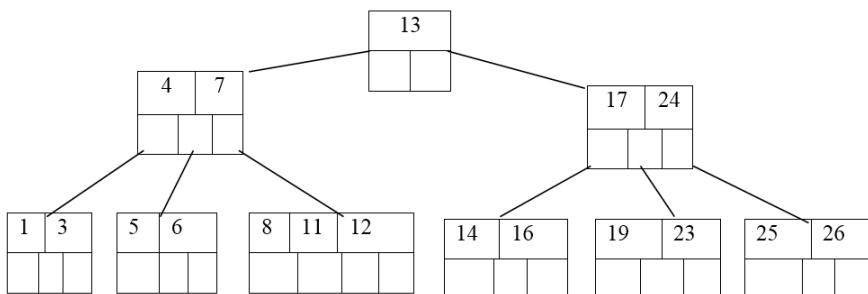
an appropriate pointer is deleted. If the record is not in a terminal node, it is replaced by a copy of its successor, which is a record with next higher value. Consider a B- Tree, If we want to delete 8 then,



Now we want to delete 20, the 20 is not in a leaf node so we will find its successor which is 23. Hence 23 will be moved up to replace 20.

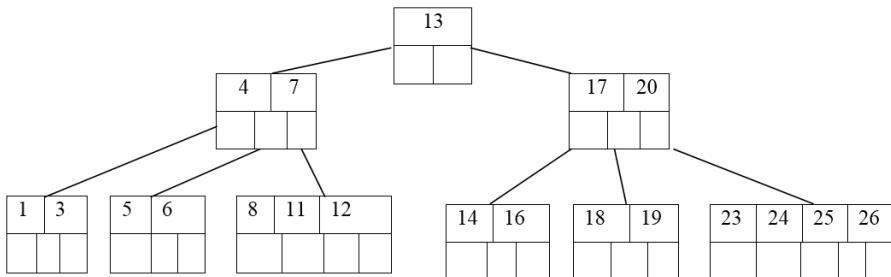


Next, we will delete 18; Deletion of 18 from the corresponding node causes the node with only one key, which is not desired in B-tree of order 5. The sibling node to the immediate right has an extra key. In such a case we can borrow a key from parent and move spare key of a sibling to up.



3. Searching

The search operation on B-tree is similar to a search on a binary search tree. Instead of choosing between a left and right child as in a binary tree, B-tree makes an m-way choice



Consider a B-tree as given below- If we want to search node 11 then

1. $11 < 13$: Hence search left node
2. $11 > 7$: Hence rightmost node
3. $11 > 8$, move in the second block
4. Node 11 is found.

The running time of search operation depends upon the height of the tree. It is $O(\log n)$.

7.9. B+ TREE

The major drawback of the B-tree is the difficulty of traversing the keys sequentially. B+ tree retains the rapid random access property of the B-tree, while also allowing rapid sequential access. In the B+ tree, all keys are maintained in leaves and keys are replicated in non-leaf nodes to define paths for locating individual records. The leaves are linked together to provide a sequential path for traversing the keys in the tree. The B+ Tree is called a balanced tree because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disk. The B+ tree is anchored by a special node called the root and bounded by leaves. It has a unique path to each leaf and all paths are equal length. It stores keys

only at leaves and store reference values in other, internal nodes. A B+ Tree of order M ($M > 3$) is an M-ary tree with the following properties:

- The data items are stored on leaves.
- The root is either a leaf or has between two and M children.
- Node: the internal node stores up to $M - 1$ keys to guide the searching, the key I represents the smallest key in subtree $I + 1$. All nodes (except the root) have between $[M/2]$ and M children.
- Leaf: a leaf has between $[L/2]$ and L data items, for some L (usually $L \ll M$, but we will assume $M = L$ in most example). All leaves are at the same depth.
- Less disk accesses due to fewer levels in the tree.
- B+ tree provides faster sequential access of data.

Searching in a B+ Tree: In a B+ tree, searching start from the root. If an internal node is reached: Search key among the keys in that node.

- If key \leq smallest key, follow the leftmost child pointer down
- If key \leq largest key, follow the rightmost child pointer down
- If $k_i \leq$ key $< k_j$, follow the child pointer between k_i and k_j
- If a leaf is reached:
 - Search key among the keys stored in that leaf
 - If found, return the corresponding record otherwise report not found.

7.9.1. Comparison between B-Tree and B+ Tree

B-Tree	B+ Tree
1. Data pointers are stored in all nodes.	1. Data pointers are stored only in leaf nodes.
2. Search can end at any node.	2. Search always ends at a leaf node.
3. No redundant keys	3. Redundant keys may exist.
4. Slow sequential access.	4. Efficient sequential access.
5. Higher trees.	5. Flatter trees.

8

CHAPTER

GRAPH

CONTENTS

8.1. Introduction	214
8.2. Definition of Graph	214
8.3. Representation of Graphs.....	221
8.4. Graph Traversal.....	224
8.5. Spanning Tree	229
8.6. Shortest Path Problem	240
8.7. Application Of Graph	243

8.1. INTRODUCTION

In the earlier chapter, we have discussed a non-linear data structure i.e., Tree. Now we have discussed another non-linear data structure, i.e., Graphs. The tree data structure the main limitation is every tree has a unique root node. If we remove this limitation, we get a more complex data structure is called graphs. In computer science the graphs are used in a wide range, there are many theorems on graphs. The study of graphs in computer science is known as graph theory. One of the first results in graph theory appeared in Leonhard Euler's paper, on seven bridges of Konigsberg, published in 1736. It's also regarded as one of the first topological results in geometry; it does not depend on any measurements. In 1945, Gustav Kirchhoff published his Kirchhoff's circuit laws for calculating the voltage and current in electric circuits. In 1852, Francis Guthrie proposed the four-color problem which asks if it is possible to color, using only four colors, any map of countries in such a way as to prevent two bordering countries from having the same color. This problem, which was only solved a century later in 1976 by Kenneth Appel and Wolfgang Haken, can be considered the birth of graph theory while trying to solve it, mathematicians invented many fundamental graph-theoretic terms and concepts.

Differences between a Tree and a Graph:

- A tree is a connected graph having no circuits, while a graph can have circuits.
- A loop may be part of the graph but the loop does not take place in the tree.
- Always tree is graph but the graph is not always a tree.

8.2. DEFINITION OF GRAPH

A graph is a non-linear data structure. A Graph is a set of elements called vertices (nodes) connected by links called edges which can be directed (assigned a direction) or undirected. A Graph $G = (V, E)$ consists of the finite non-empty set of objects V , where $V(G) = \{V_1, V_2, V_3, \dots, V_n, \dots\}$ called **vertices**, and another set E , where $E(G) = \{e_1, e_2, e_3, \dots, e_n, \dots\}$, whose elements are called **edges**. A graph may be pictorially representation, in which the vertices are represented as points and each edge as a line segment joining its end vertices.

From the Figure 8.1 we can represent: $V(G) = \{a, b, c, d, e, f\}$, $E(G) = \{(a, b), (b, a), (b, c), (c, b), (a, d), (d, a), (d, e), (e, d), (e, f), (f, e), (c, f), (f, c)\}$

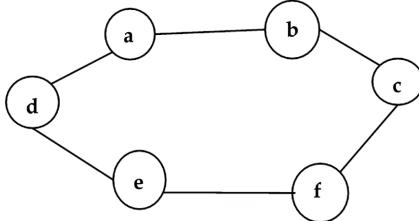


Figure 8.1: Simple Graph.

We could have written (a, e) and (e, a) means the ordering of vertices is not significant in an undirected graph.

8.2.1. Some Terminology of Graph

Directed Graph: If a graph in which every edge is identified by an ordered pair of vertices then the graph is said to be a **directed graph**.

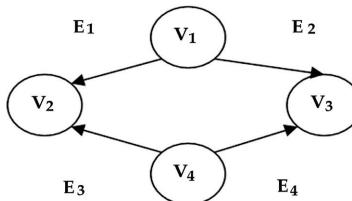


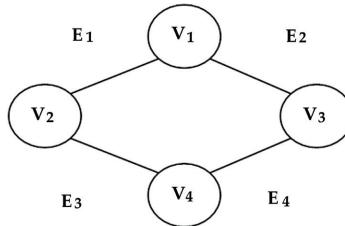
Figure 8.2: A Directed Graph.

Undirected Graph: If the edges of a graph are undirected or “two-way” then the graph is known as **an undirected graph**.

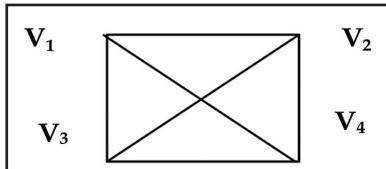
By an unordered pair of edges, we mean that the order in which the ' V_i ', ' V_j ' occur in the pair of vertices (V_i, V_j) is unrelated for describing the edge. Thus the pair (V_i, V_j) and (V_j, V_i) both represented the same edge that connected the vertices V_i and V_j . Figure 8.3 shown an undirected graph.

$$\begin{aligned} \text{Set of vertices } V &= \{V_1, V_2, V_3, V_4\}, \\ \text{Set of edges } E &= \{e_1, e_2, e_3, e_4\}. \end{aligned}$$

We can say E_1 is the set of (V_1, V_2) and of (V_2, V_1) represent the same edge.

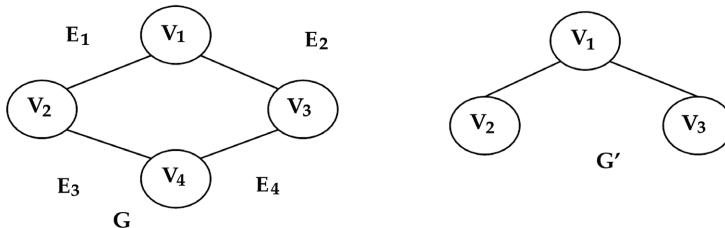
**Figure 8.3:** An Undirected Graph.

Complete graph: if each vertex of a graph is connected to each other. An undirected graph of n vertices consists of $n(N - 1)/2$ numbers of edges then it is called a complete graph. The graph shown in figure 8.4 is a completed graph.

**Figure 8.4:** A Complete Graph.

Above graph have 4 vertices and it has 6 edges. So $n = 4$ and edges $n(N - 1)/2 = 4(4 - 1)/2 = 12/2 = 6$ edges. This is called complete graph.

Subgraph: A subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' is a proper subset of the set of the edges of G .

**Figure 8.5:** A Sub Graph.

Connected Graph: An undirected graph is said to be connected if every pair of distinct vertices V_i and V_j in $V(G)$ there is a graph from V_i and V_j in G .

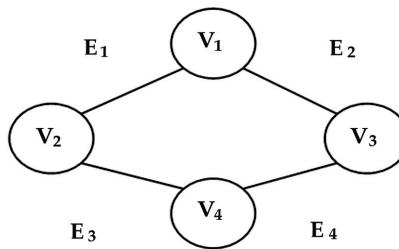


Figure 8.6: A connected graph.

Multigraph: A graph which contains a pair of nodes joined by more than one edge is called a Multigraph such edges is called **parallel edges**.

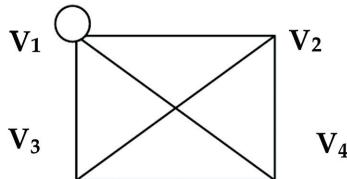


Figure 8.7: A Multigraph.

An edge having the same vertex as both its end vertices are called a **self-loop** (or a loop). The graph shown in Figure 8.7 is a **Multigraph**. A graph that has self-looped or is parallel edges called a **General graph**. A graph that does not self-loop nor are parallel edges called a **simple graph**.

Table for Compare Simple, Multigraph and General Graph

Type	Edge	Multiple Edges?	Loop?
Simple	Undirected	No	No
General	Undirected	Yes	No
Multigraph	Undirected	Yes	Yes

Degree: Degree of a vertex is always a non-negative integer. It is the total number of edges incident with 'V_i'. It is to be noted that self-loops on a given vertex are counted twice. An edge having the same vertex as both its end vertices are called a **self-loop**.

- If the degree of a vertex V is 0 then V is called an isolated vertex of then graph and

- If the degree is 1 the V is called an end vertex of pendant vertex or a leaf.

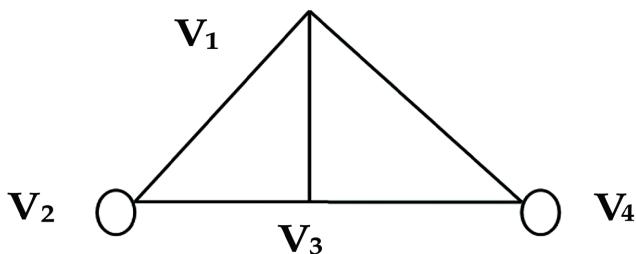


Figure 8.8: A Graph with four Vertices.

From Figure 8.8, we can calculate the degree of vertices, $d_G(V_1) = 3$, $d_G(V_2) = 4$, $d_G(V_3) = 3$, $d_G(V_4) = 4$. In a directed graph, the edges of not only incident on a vertex, but also incident out of vertex and incident into a vertex. In this case, the degree is considered as **out degree** and **in degree**. When the edge is incident out of given vertex V_i , then it is denoted as $d^+_G(V_i)$, and when it is incident into a vertex V_i then it is denoted as $d^-_G(V_i)$.

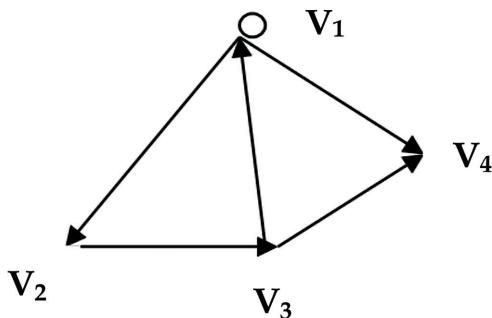


Figure 8.9: A directed graph with four Vertices.

For Figure 8.9, the degree of vertices is as follows:

$$\begin{array}{ll}
 d^+_G(V_1) = 2 & d^-_G(V_1) = 1 \\
 d^+_G(V_2) = 1 & d^-_G(V_2) = 1 \\
 d^+_G(V_3) = 2 & d^-_G(V_3) = 1 \\
 & d^+_G(V_4) = 2
 \end{array}$$

$$d^+_G(V_4) = 0 \quad d^-_G(V_4) = 2$$

As we have observed that for an undirected graph edge contributes two degrees. A graph 'G' with ' e_k ' edges and 'n' vertices V_1, V_2, \dots, V_n , the number of edges is half the sum of the degrees of all vertices. $\frac{1}{2} \sum_{i=1}^n d_i = e$

Again, it can be easily calculated that for any directed graph the sum of all in-degrees is equal to the sum of all out-degrees, each sum is equal to the number of edges of a graph G, thus $\sum_{i=1}^n d_i^+ = \sum_{i=1}^n d_i^- = e$

Null Graph: If a graph contains an empty set of edges and non-empty sets of vertices, the graph is known as a Null graph. The graph shown in Figure 8.10 is a null graph.

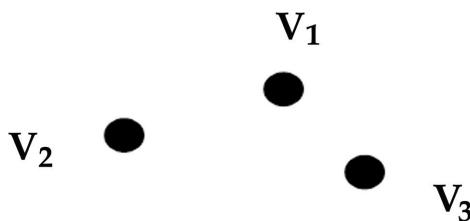


Figure 8.10: A Null Graph.

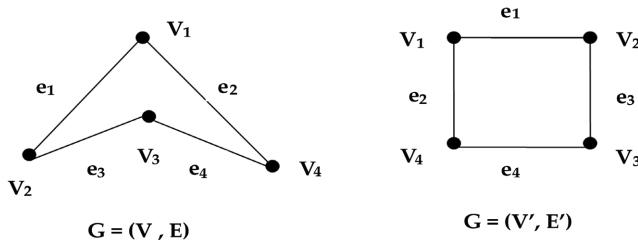
Finite and Infinite Graph

A graph has a finite number of edges as well as a finite number of vertices is called finite graph, otherwise an infinite graph.

Graph Isomorphism

Two graphs, $G = \{V, E\}$ and $G' = \{V, E\}$ are said to be **isomorphic graphs** if there exists a one-to-one correspondence between their vertices and between their edges such that the incidence relationship is preserved. Suppose that an edge ' e_k ' has end vertices ' V_i ' and ' V_j ' in G , then the corresponding edge ' e'_k ' in ' G' must be incident on the vertices ' V_i' and ' V_j' that correspond to ' V_i ' and ' V_j ', respectively.

Two Isomorphic graphs are shown in the Figure 8.11.

**Figure 8.11:** An Isomorphic Graph.**Isomorphic Properties:**

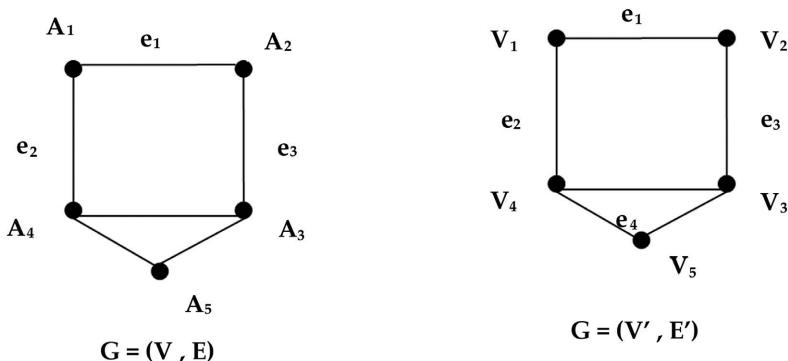
- Both the graphs G and G' have the same number of vertices.
- Both the graphs G and G' have the same number of edges.
- Both the graphs G and G' have same degree sequences.

Homeomorphic Graph

Two graphs G_1 and G_2 are said to be Homeomorphic to each other if one is obtained from the other by the merger of two edges incident on a vertex or by inspection of a vertex of degree 2, who are resulting in the creation of an edge in series.

Example:

The following graphs G_1 and G_2 are Homeomorphic:

**Figure 8.12:** A Homeomorphic Graph.

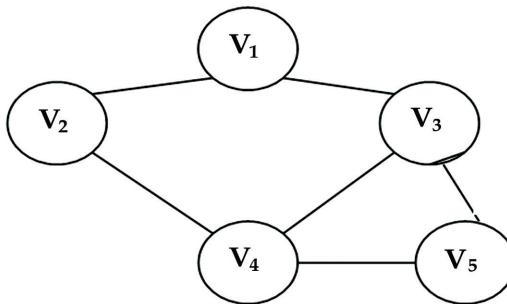
8.3. REPRESENTATION OF GRAPHS

There are two most important approaches to represent graphs:

- Adjacency Matrix Representation
- List Representation

1. **Adjacency Matrix Representation:** Consider a graph G has n vertices and the matrix M. if there is an edge present between vertices V_i and V_j then $M[i][j] = 1$ else $M[i][j] = 0$. Note that for an undirected graph if $M[i][j] = 1$ then for $M[j][i]$ is also 1. Here are some graphs are shown by an adjacency matrix.

Example: Adjacency matrix for given Figure 8.13, an undirected graph.



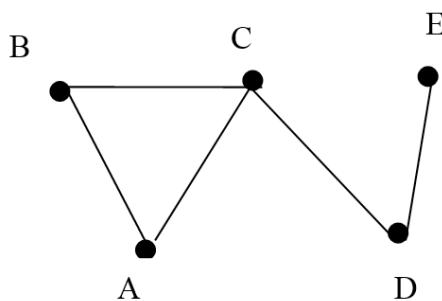
	1	2	3	4	5	
1	0	1	1	0	0	
2	1	0	0	1	0	
3	1	0	0	1	1	
4	0	1	1	0	1	
5	0	0	1	1	0	

Figure 8.13: An Undirected Graph.

Example 8.1: Draw the undirected graph G corresponding to the adjacency matrix.

	A	B	C	D	E	
A	0	1	1	0	0	
B	1	0	1	0	0	
C	1	1	0	1	0	
D	0	0	1	0	1	
E	0	0	0	0	1	

Solution: Graph (G) = $[G_{ij}]_{5 \times 5}$ matrix, so that graph G has five vertices, say A, B, C, D, E. draw the edge from A to E, where $G_{ij} = 1$, the graph is:



2. **Adjacency List Representation:** We have seen how a graph can be represented using an adjacency matrix. We have used an array data structure over there. But the problems associated with array are still there in the adjacency matrix that there should be some flexible data structure and so we go for a linked data structure for the creation of a graph. The type in which a graph is created with the linked list is called **adjacency list**. We will represent a graph using adjacency lists. This adjacency list stores information about only those edges that exist. The adjacency list contains a directory and a set of linked list. This representation is also known as **Node directory representation**.

The directory contains one entry to each node of the graph. Each entry into the directory points to a linked list that represents the nodes that connected to that node. The directory represents the nodes and linked list represent the edges.

Node_id	Next	Or	Node_id	Weight	Next
---------	------	----	---------	--------	------

Figure 8.15 represents the linked list representation of the directed graph given Figure 8.14.

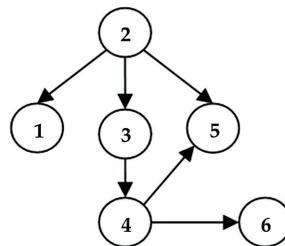


Figure 8.14: A Directed Graph.

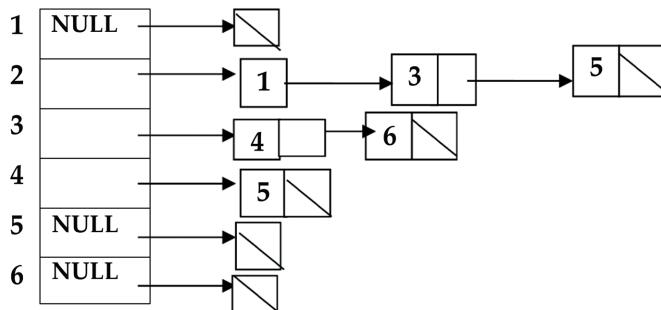


Figure 8.15: Linked list representation of Figure 8.14.

An undirected graph of order N with E edges requires N entries in the directory and $2 * E$ linked list entries. The adjacency list representation of Figure 8.16 is shown in Figure 8.17.

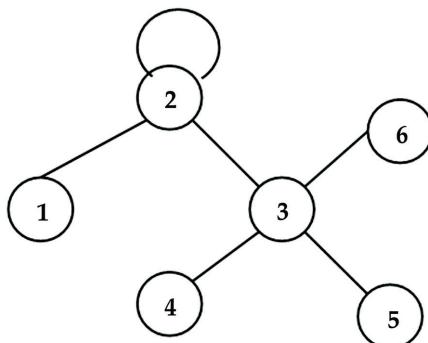
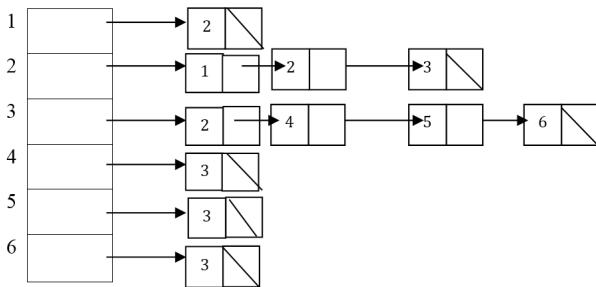


Figure 8.16: An Undirected Graph.

**Figure 8.17:** Linked list of Figure 8.16.

Properties of Adjacency List Matrix:

- Except for the self-loop the diagonal element has value zero. A self-loop at the i^{th} vertex corresponds to $a_{ii} = 1$
- An adjacency matrix of an undirected graph is symmetric, as $a_{ij} = a_{ji} = 1$.
- The non-zero elements in the matrix represent the number of edges in a graph.

8.4. GRAPH TRAVERSAL

To traverse a graph is to process every node in the graph exactly once. There are many paths leading from one node to another node, the hardest part in traversing a graph is making sure that you do not process some node twice. So, we have to process the node exactly once. Initially, all the nodes are ‘unreached.’ When the first node is encountered to mark it as ‘reached’ and process that node. So, while traversing the node is every time checked whether if it is marked ‘reached’ or not. The graph traversing continues until all the nodes are processed. If we delete the node after processing, then there will be no paths leading to that node. The general techniques for graph traversing are given below:

- Mark all nodes in the graph as **unreached**.
- Pick a starting node, mark it was **reached** and places it on the **ready** list.
- Pick a node on the **ready** list, process it. Remove it from **ready**, find all its neighbors those that are **unreached** should be marked as **reached** and added to **ready**.

- Repeat 3 until ready are empty.

The process of traversing the graph is given below: $V = \{V_1, V_2, V_3, V_4, V_5, V_6\}$ marked as ‘unreached.’ V_1 = start vertex.

Ready list = $\{V_1\}$ process V_1 , place adjacent vertices to vertices to V_1 in ready lists and delete node V_1 .

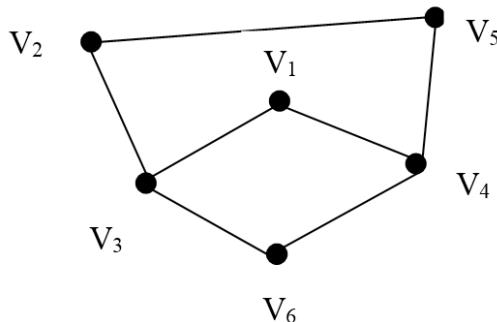


Figure 8.18: Graph with six vertices.

Ready list = $\{V_3, V_4\}$ process V_3 , place adjacent vertices to V_3, V_2, V_6 as V_1 is deleted so no path from V_3 to V_1 .

Ready list = $\{V_4, V_2, V_6\}$ process V_4 , place adjacent vertices to V_4, V_5 , as V_6 is marked as reached and V_4 is deleted.

Ready list = $\{V_2, V_6, V_5\}$ process V_2 , place adjacent vertices to V_2 as V_3 is deleted and V_5 is marked as reached and V_2 is deleted.

Ready list = $\{V_6, V_5\}$ process V_6 , place adjacent vertices to V_6 , as V_3 and V_5 are deleted, V_6 is deleted.

Ready list = $\{V_5\}$ process V_5 , as all the adjacent vertices are already deleted so finally delete V_5 .

The graph is traversed as V_1, V_3, V_4, V_2, V_6 , and V_5 . Traversing is only for the connected graph. For the unconnected graph, the whole procedure is repeated until all the vertices are marked as ‘reached’ and then processed. The graph can be traversed in two ways:

1. Depth-first search traversal (DFS)

Depth-first traversal of an undirected graph is similar to a preorder traversal of an ordered tree. The start vertex v is visited first. Let w_1, w_2, \dots, w_k be the vertices adjacent to v . then the vertex w_1 is visited next. After visiting w_1 all the vertices adjacent to w_1 is visited next. After visiting w_1 all the

vertices adjacent to w_1 have visited in a depth-first manner before returning to traverse w_2, \dots, w_k .

The algorithm for depth-first traversal of an undirected graph is given below-

```

visited (v) = TRUE;
visited (v);
for each vertex w adjacent to v do
if not visited (w) then
traverse (w);
end;
```

For example, let a graph is shown in Figure 8.19 which is visited in depth first traversal starting from vertex A.

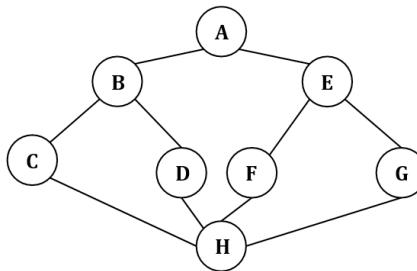


Figure 8.19: An undirected graph.

The sequence of nodes to be visited in depth-first search traversal is as: A B C H D E F G.

2. Breadth-first search traversal (BFS)

The breadth-first traversal differs from depth-first traversal in that all unvisited vertices adjacent to v are visited after visiting the starting vertex v and marking it as visited. Next, the unvisited vertices contiguous to these vertices are visited and so on until the entire graph has been traversed.

For example, the breadth-first traversal of the graph of Figure 8.20 results in visiting the nodes in the following order:

A B E C D F G H.

Breadth-first search explores the space level by level only when there are no more states to be explored at a given level does the algorithm move

on the next level. We implement BFS using lists open & closed to keep track of progress through the state space.

Algorithm for BFS:

```

begin
    open = [start];
    closed = [];
    while open ≠ [] do
        begin
            remove the leftmost state from open call it x;
            if x is a goal then return success
            else
                begin
                    generate children of x;
                    put x on closed;
                    put children on the right end of open;
                end
        end
    end
    return (failure)
end

```

For example, consider the tree shown Figure 8.20. The open and closed lists maintained by BFS are shown below:

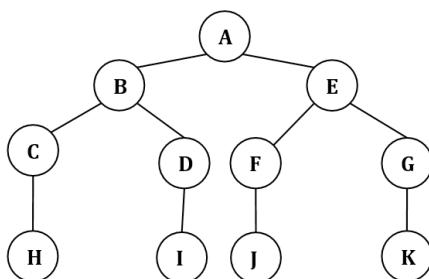


Figure 8.20: An undirected graph.

Open = [A];	Closed = []
Open = [B, E];	Closed = [A]
Open = [E,C,D];	Closed = [A,B]
Open = [C,D,F,G];	Closed = [A,B,E]
Open = [D,F,G,H];	Closed = [A,B,E,C]
Open = [F,G,H,I];	Closed = [A,B,E,C,D]
Open = [G,H,I,J];	Closed = [A,B,E,C,D,F]
Open = [H,I,J,K];	Closed = [A,B,E,C,D,F,G]
Open = [I,J,K];	Closed = [A,B,E,C,D,F,G,H]
Open = [J,K];	Closed = [A,B,E,C,D,F,G,H,I]
Open = [K];	Closed = [A,B,E,C,D,F,G,H,I,J]
Open = [];;	Closed = [A,B,E,C,D,F,G,H,I,J,K]

To understand DFS, consider the Figure 8.21. The open and closed list maintained by DFS is shown below-

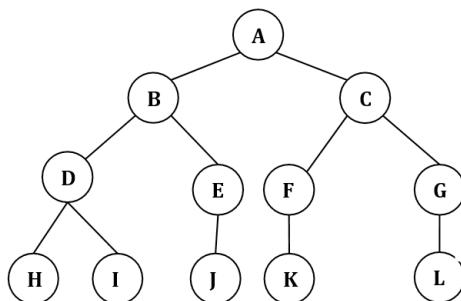


Figure 8.21: An undirected graph.

Open = [A];	Closed = []
Open = [B, C];	Closed = [A]
Open = [D,E,C];	Closed = [A,B]
Open = [H,I,E,C];	Closed = [A,B,D]
Open = [I,E,C];	Closed = [A,B,D,H]
Open = [E,C];	Closed = [A,B,D,H,I]
Open = [J,C];	Closed = [A,B,D,H,I,E]
Open = [C];	Closed = [A,B,D,H,I,E,J]

Open = [F,G];	Closed = [A,B,D,H,I,E,J,C]
Open = [K,G];	Closed = [A,B,D,H,I,E,J,C,F]
Open = [G];	Closed = [A,B,D,H,I,E,J,C,F,K]
Open = [L];	Closed = [A,B,D,H,I,E,J,C,F,K,G]
Open = [];	Closed = [A,B,D,H,I,E,J,C,F,K,G,L]

Advantages of BFS:

- BFS will not get trapped on dead end paths. This contrasts with DFS which may follow a single unfruitful path for a long time, before the path actually terminates in a state that has no successor.
- If there is a solution then BFS guarantees to find it. Furthermore, if there are multiple solutions than a minimal solution will be found.

Disadvantages of BFS:

- Full tree explored so far will have to be stored in memory.

Advantages of DFS:

- DFS requires less memory since only the nodes on the current path are stored. This contrasts with BFS where all of the trees that have so far been generated must be stored.
- By chance, DFS may find a solution without examining much of the search space at all. This contrasts with BFS in which all parts of the trees must be examined to level n before any nodes of level n + 1 are examined.

Disadvantages of DFS:

- DFS may be trapped on dead-end paths. DFS follows a single unfruitful path for a long time before the path actually terminates in a state that has no successor.
- DFS may find a long path to a solution in one part of the tree when a shorter path exists in some other unexpected part of the tree.

8.5. SPANNING TREE

Consider a graph $G = (V, E)$, if 'T' is a sub-graph of G and contains all the vertices but no cycle or circuit, then "T" is said to be a spanning tree. Here

we are using the connected graph, the reason for this straight because a tree is always connected and in the unconnected graph of ‘n’ vertices we cannot find a subgraph with ‘n’ vertices. For creating the spanning tree of a given graph, we have deleted the edge from the circuit and the resultant obtained tree should be connected. If a graph $G = (V, E)$ is any connected graph, a spanning tree in G is a subgraph T of g , which is a tree. If it follows these properties:

- T has the same vertex V of vertices as does G ,
- T is a tree, T has n -vertices and $9N - 1$) edges,
- T is a subgraph of G .

Figure 8.22 (b) illustrates the spanning tree of graph G shown in Figure 8.22(a).

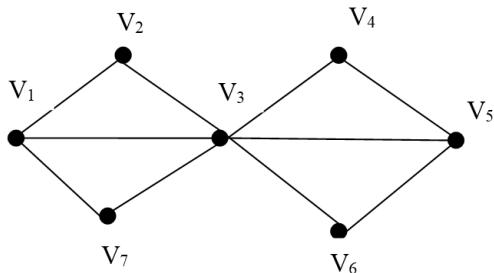


Figure 8.22(a): Undirected Graph.

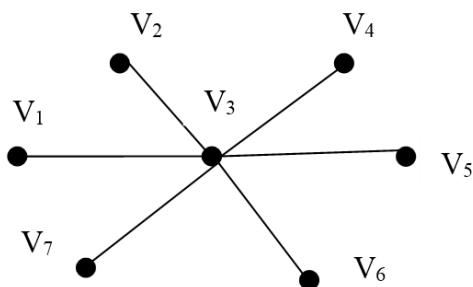


Figure 8.22(b): Spanning Tree.

8.5.1. Minimum Spanning Tree

A spanning tree in a graph G is a minimum subgraph connecting all the vertices of G . if a weighted graph is considered, then the weight of the spanning tree

“T” of graph ‘G’ can be calculated by summing all the individual weights in the spanning tree T. as we have observed that there exists several spanning trees of a graph “G” so, in the case of weighted graph, different spanning trees of “G” will have different weights. A spanning tree with the minimum weight in a weighted graph is called **minimal spanning tree or shortest spanning tree or minimum cost spanning tree**. There are several methods for finding a minimum spanning tree in a given graph.

Kruskal Algorithm – In Kruskal’s algorithm minimum weight is obtained. Firstly, the list of all edges of the graph “G” in order of decreasing weights. Then the edge with the shortest of the minimum weight is selected. Next, for each successive step select from the remaining edges another edge that has minimum weight and then, follows the condition that this edge does not make any circuit with previously selected edges. The whole process continues until all $N - 1$ edges are selected and these edges will form the desire minimal spanning tree.

Algorithm steps:

1. Initialize $T = \text{NULL}$
2. (scan $N - 1$ edges from the given set E)

Repeat through $E_i = 1, 2, 3, \dots, N - 1$, edges

Set edge = minimum (E_i)

Set temp = edge [delete edge from the set E]

3. (add temp to T if no circuit is obtained)

Repeat while E_i does not create a cycle

Set $T = \text{temp}$ [minimum weight edges]

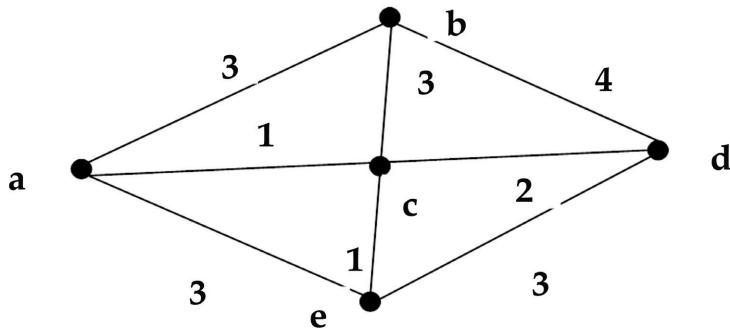
4. (no spanning tree)

If edges of T is less than $N - 1$ edges

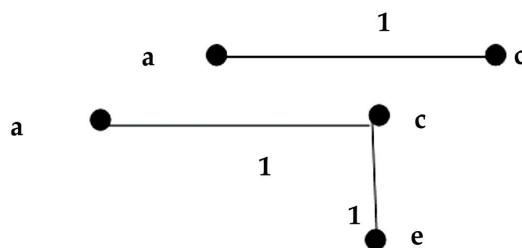
Then message = “No spanning Tree”

5. Exit

Exercise 8.2: Using Kruskal’s algorithm, find a minimum spanning tree for a given graph.

**Figure 8.23:** Undirected Graph.

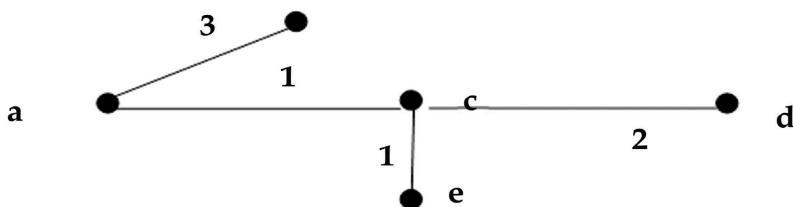
Step1. Choose the edge (a, c) it has a minimum weight.



Step2. Select the next edge (c, e).

Step3. Select the next edge (c, d).

Step4. Select the next edge (a, b).

**Figure 8.24:** Minimum spanning tree Graph of 8.23.

The minimum spanning tree of G, and minimum weight is: $1 + 1 + 2 + 3 = 7$.

Example 8.3: Consider a graph $G = (V, E, W)$ undirected connected weighted graph shown in Figure 8.25. The Kruskal algorithm on graph ‘G’ produces the minimum spanning tree shown in Figure 8.26.

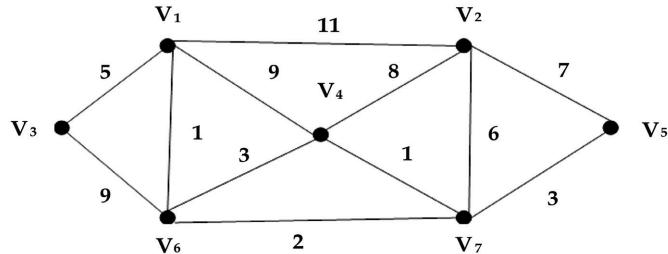
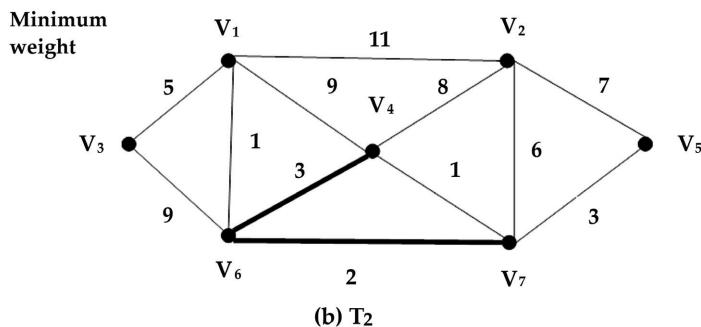
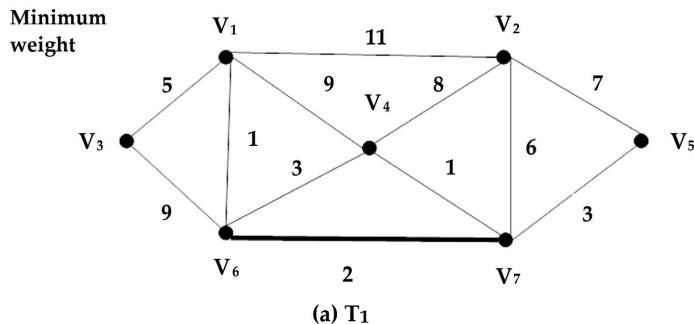
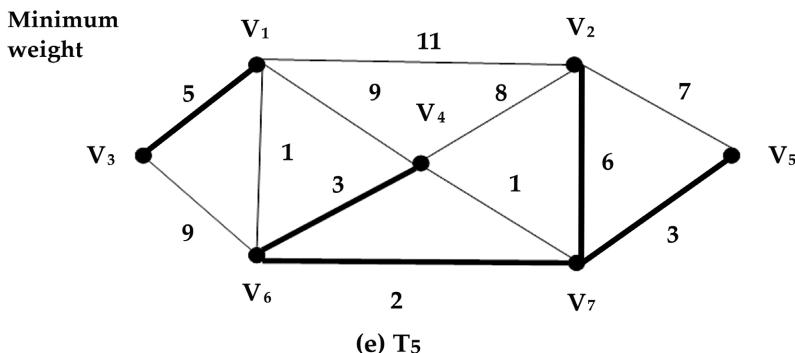
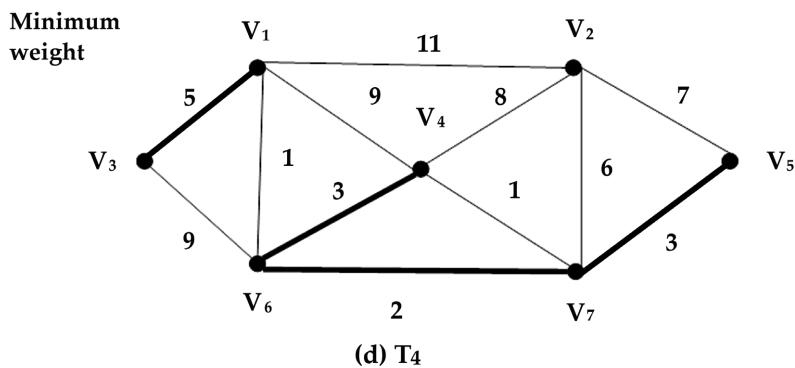
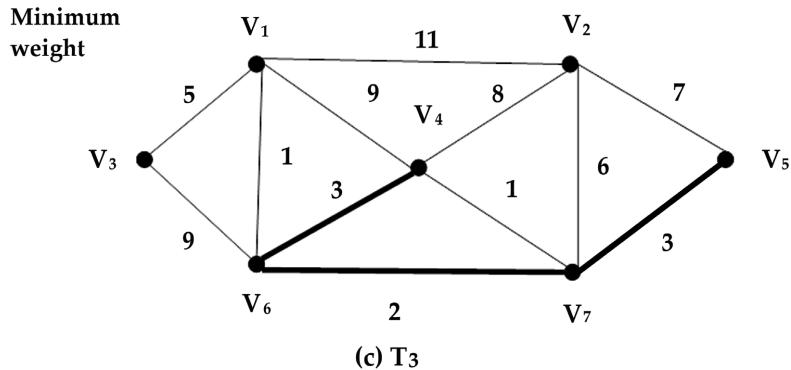
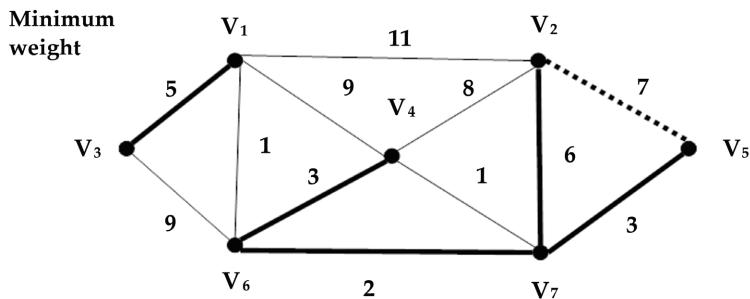


Figure 8.25: Undirected Graph G.

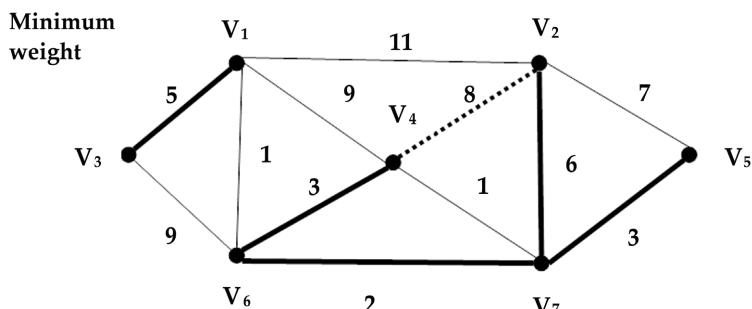
Solution: The process for obtaining the minimum spanning tree using Kruskal algorithm is pictorially shown below:



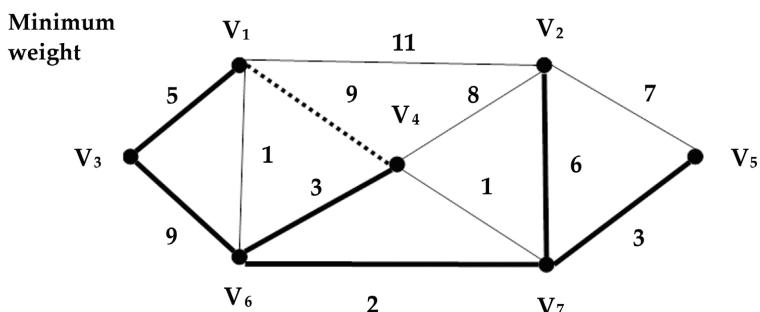




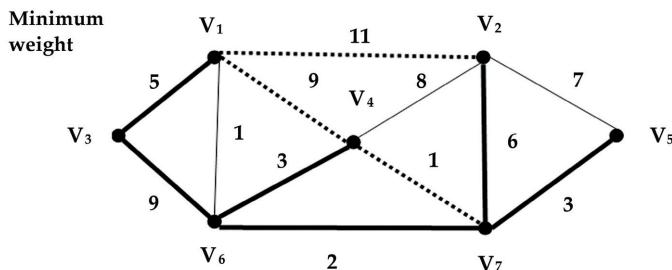
(f) T₆ weight 7 produces cycle so ignore the



(g) T₇ weight 8 produces cycle so ignore the



(h) T₈ weight 9 produces cycle so ignore the



(i) T₉ weight 10, 11 produces cycle so ignore the

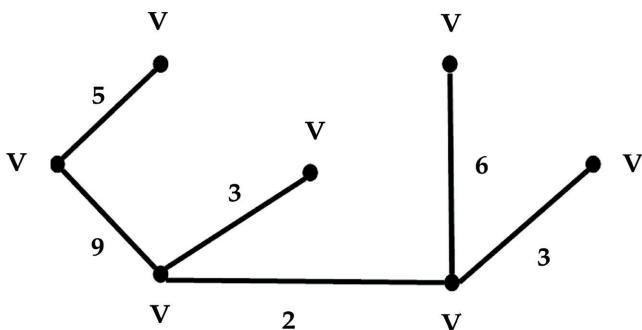


Figure 8.26: A Minimum Spanning tree of Figure 8.25.

Hence, the minimum cost of spanning tree of the given graph using Kruskal's algorithm is = 2 + 3 + 3 + 5 + 6 + 9 = 28

2. **Jarnik-Prim's Algorithm:** In Prim's algorithm the pair with the **minimum weight** is to be chosen. The adjacent to these vertices whichever is the edge having minimum weight selected. This process will be continued until all the vertices are not covered. The necessary condition, in this case, is that the **circuit should not be formed**. From Figure 8.27 we will build the minimum spanning tree. In this algorithm starting at a designated vertex chooses an edge with minimum weight and considers this edge and associated vertices as part of the desired tree. Then iterate looking for an edge minimum weight not yet selected that has one of its vertices in the tree while the other vertex is not. The process terminates when $(n - 1)$ edges have been selected from a graph of n vertices to form a minimal spanning tree

Exercise 8.4: Using Prim's algorithm, find a minimum spanning tree for a given graph.

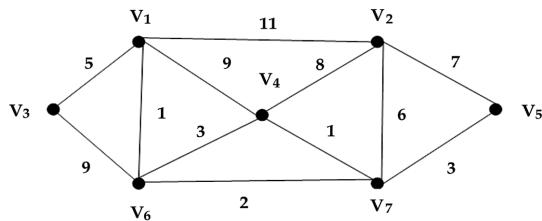


Figure 8.27: Undirected Graph G.

Solution: minimum spanning tree using Prim's algorithm is pictorially shown below:

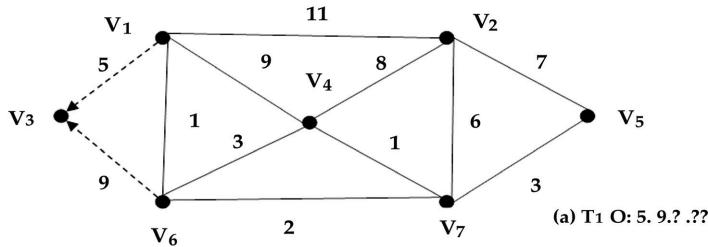
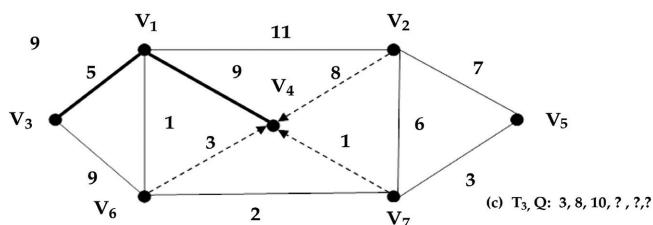
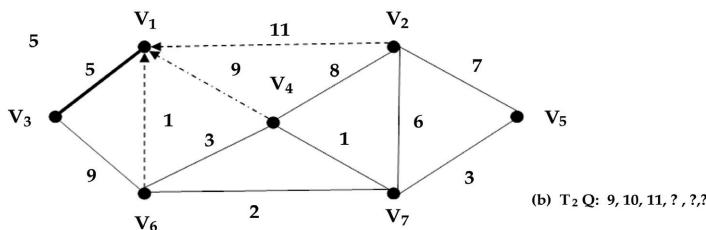
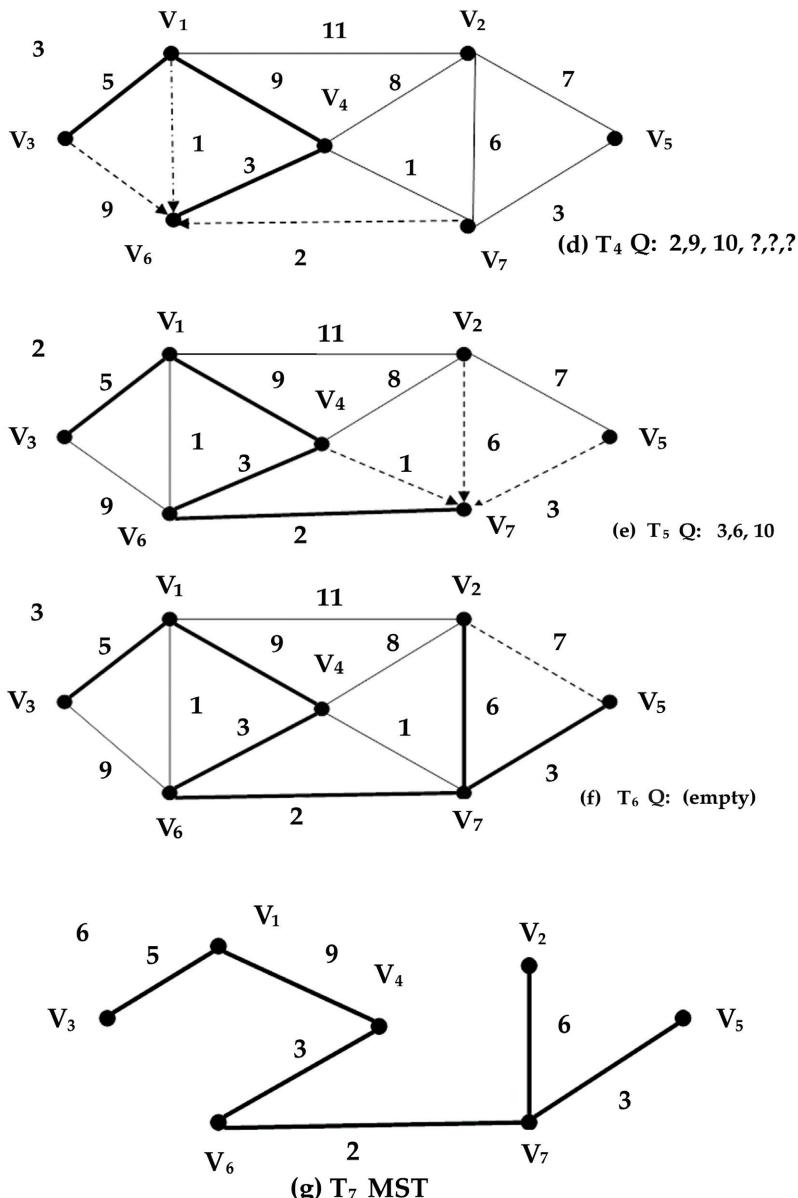


Figure 8.28: Undirected Graph G.



**Figure 8.29:** A Minimum Spanning tree of Figure 8.27.

Minimum cost of spanning tree of the given graph is $= 5 + 9 + 3 + 2 + 3 + 6 = 28$.

8.5.2. Difference between Prim's Algorithm and Kruskal's Algorithm

In **Prim's algorithm**, an arbitrary node is chosen initially as the root node. The nodes of the graph are then appended to the root one at a time until all nodes of the graph are included. The node of the graph added to the tree at each point is that node adjacent to a node of the tree by an arc of minimum weight. The arc of minimum weight becomes a tree arc connecting the new node to the tree. When all the nodes of the graph have been added to the tree, a minimum spanning tree has been constructed for the graph.

While in **Kruskal's algorithm**, the nodes of the graph are initially considered as n distinct partial trees with one node each. At each step of the algorithm, two distinct partial trees are connected into a single partial tree by an edge of the graph. When only one partial tree exists, it is a minimum spanning tree.

8.5.3. Travelling Salesman Problem

Traveling salesman problem is the problem of finding the shortest path that goes through every node exactly once, and return to the start. That problem is NP-complete, so an efficient solution is not likely to exist. Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city once and then returns to the starting city? An equivalent formulation in terms of graph theory is: given a complete weight graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost of the distance of that road) find the Hamilton cycle with the least weight. It can be shown that the requirement of returning to the starting city does not change the computational complexity of the problem. Now, consider a directed graph where edges represented the roads with their weights as distance and vertices as the cities. Then for this graph traveling salesperson problem is solved by selecting by selecting all the Hamiltonian circuits and then selecting the shortest one. The total number of the Hamiltonian circuits present in the graph having ' n ' number of vertices can be obtained by $(n - 1)!$. For finding the shortest routes various algorithms are available but none of them prove to be best.

8.6. SHORTEST PATH PROBLEM

In our daily life, everybody is facing a problem of choosing the shortest path from one location to another location. By minimum spanning tree, we are not able to obtain the shortest path between two nodes (source and destination nodes). We can obtain simply the minimum cost. But the using shortest path algorithm we can obtain the minimum distance between two nodes. A solution to the shortest path problem is sometimes called the pathing algorithm. The most important algorithms for solving this problem are:

- Dijkstra's algorithm: In this algorithm solves a single source problem if all edge weight is greater than or equal to zero. Without worsening the runtime, this algorithm can, in fact, compute the shortest paths from a given start point to all other nodes.
- Bellman-Ford algorithm: In this algorithm solves a single source problem if edge weights may be negative.
- A*algorithm: a heuristic for single source shortest paths.
- Floyd-Warshall algorithm: solves all pair's shortest paths.

There is a weighted and unweighted graph, Based on this category, let us discuss the shortest path algorithm.

Unweighted shortest path: The unweighted shortest path algorithm gives a path in the unweighted graph which is equal to a number of edges traveled from source to destination.

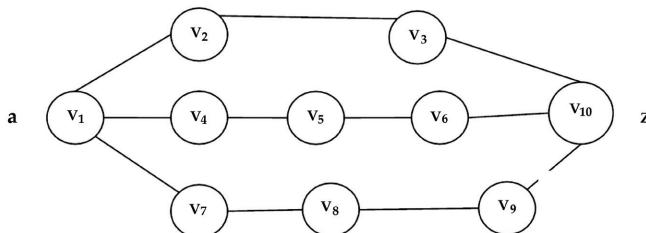


Figure 8.30: Unweighted graph.

The paths between a to z are as below:

S.N.	Path	Number of edges
1	V ₁ -V ₂ -V ₃ -V ₁₀	3
2	V ₁ -V ₄ -V ₅ -V ₆ -V ₁₀	4

3	$V_1 - V_7 - V_8 - V_9 - V_{10}$	4
---	----------------------------------	---

Out of these, the path 1 i.e., $V_1 - V_2 - V_3 - V_{10}$ is shortest only 3 edges from a to z.

Dijkstra's shortest path algorithm: The Dijkstra's shortest path algorithm suggest the shortest path from some source node to some other destination node. The source node or the node from we start measuring the distance is called the start node and the destination node is called the end node. In this algorithm, we start finding the distance from the start node and find all the paths from it to neighboring nodes. Among those whichever is the nearest node that path is selected. This process of finding the nearest node is repeated until the end node. Then whichever is the path that path is called the shortest path. Since in this algorithm all the paths are tried and then we are choosing the shortest path among them, this algorithm is solved by a greedy algorithm. One more thing is that we are having all the vertices in the shortest path and therefore the graph doesn't give the spanning tree.

Exercise 8.5: find the shortest distance between a to z for the given graph Figure 8.31.

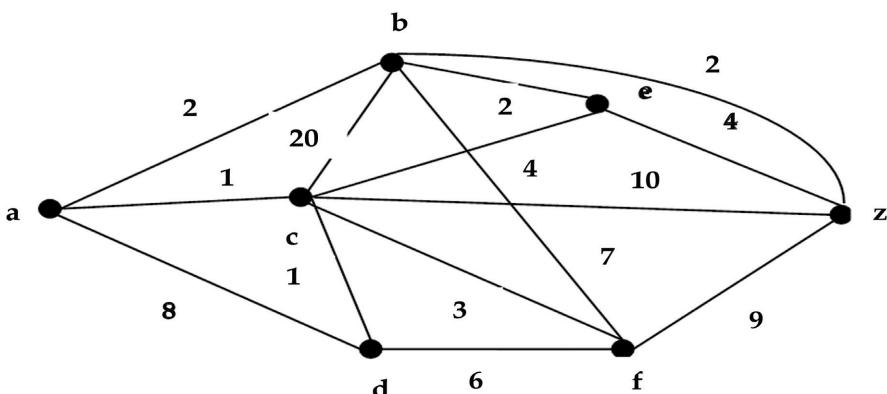


Figure 8.31: A Graph.

Solution: The shortest distance between a and z is computed for the given graph using Dijkstra's algorithm as follows: P = Set which is for nodes which have already selected

T = Remaining nodes

Step 1: $v = a, P = \{a\}, T = \{b, c, d, e, f, z\}$

distance (b) = min {old distance (b), distance (a) + w (a,b)}

dist (b) = min { ∞ , 0+22}

dist(b) = 22, dist(c) = 16, dist(d) = 8 **minimum node**

dist(e) = ∞ , dist(f) = ∞ , dist(z) = ∞ so minimum node is selected in P i.e., node d

Step 2: v = d, P = {a,d}, T = {b,c,e,f,z}

distance (b) = min {old distance (b), distance (a) + w (a,b)}

dist (b) = min {22, 8+ ∞ }

dist(b) = 22, dist(c) = min{16, 8+10} = 16

dist(e) = min{ ∞ , 8+ ∞ } = ∞

dist(f) = min{ ∞ , 8 + 6} = 14 **minimum**

dist(z) = min{ ∞ , 8+ ∞ } = ∞

Step 3: v = f, P = {a,d,f}, T = {b,c,e,z}

distance (b) = min {old distance (b), distance (a) + w (a,b)}

dist (b) = min {22, 14 + 7} = 21

dist(b) = 21

dist(c) = min{16, 14 + 3} = 16 minimum

dist(e) = min{ ∞ , 14 + ∞ } = ∞

dist(z) = min{ ∞ , 14 + 9} = 23

Step 4: v = c, P = {a,d,f,c}, T = {b,e,z}

distance (b) = min {old distance (b), distance (a) + w (a,b)}

dist (b) = min {21, 16 + 20} = 21

dist(b) = 21

dist(e) = min{ ∞ , 16 + 4} = 20 minimum

dist(z) = min{23, 16 + 10} = 23

Step 5: v = e

P = {a,d,f,c,e}, T = {b,z}

distance (b) = min {old distance (b), distance (a) + w (a,b)}

dist (b) = min {21, 16 + 20} = 21

dist(b) = 21 minimum

dist(z) = min{23, 20 + 4} = 23

Step 6: $v = b$

$$P = \{a, d, f, c, e, b\}, \quad T = \{z\}$$

$$\text{dist}(z) = \min\{23, 21 + 2\} = 23$$

Now the target vertex for finding the shortest path is z. hence the length of the shortest path from the vertex a to z is 23. The shortest path is the given graph is {a,d,f,z}

8.7. APPLICATION OF GRAPH

The graph theory is used in the computer science broadly. There are many interesting applications of graph. We will list out few applications.

- In computer networking such as Local Area Network (LAN), Wide Area Networking, internetworking.
- In the telephone cabling, graph theory is effectively used.
- In job scheduling algorithms.
- In a study of molecules in science in condensed matter physics, the three-dimensional structure of the complicated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties.
- Konigsberg bridge problem.
- Seating problem
- Problem related to electric networks
- Timetable or schedule of periods
- Utility problem

A graph can also be used to represent physical situation involving discrete objects and a relationship between them.

9

CHAPTER

SORTING

CONTENTS

9.1. Introduction.....	246
9.2. Types of Sorting	246
9.3. Basic Terms of Sorting.....	246
9.4. Sorting Techniques.....	247

9.1. INTRODUCTION

Sorting is a process that plays important role in various applications, most of them the database applications occupy a huge amount of data. Think about a payroll system for a multinational company, that have several departments and each department is having many employees. Now if we want to find the salary of a particular employee, it will very difficult for us to find each and every record of the employee data. If the data is prepared according to the employee's salary, i.e., either in ascending (increasing) or in descending (decreasing) order of the employee Id then the searching of the desired data will be an easy task. **Sorting** is nothing but an arrangement of the data in a particular order, the arrangement means based on some key the data should be arranged in ascending or descending order.

9.2. TYPES OF SORTING

Mainly sorting can be classified into two types **of Internal** sorting and **External** sorting. The Internal sorting is a sorting process in which the data resides in the main memory of the computer. For many applications it is not possible to store the entire data on the main memory for two reasons, first, the available space size of main memory is smaller than the total size of data. Secondly, the main memory is a volatile device so it will lose the data when the power is shut down. To overcome these problems the data is sorted on the secondary storage devices. The internal sorting can be performed on a small amount of data. The technique which is used to sort the data that resides in secondary storage (auxiliary storage) devices; this sorting method are called **external sorting**. This sorting can be performed on a large amount of data.

9.3. BASIC TERMS OF SORTING

Order: The sorting is a technique in which we expect the list of data to be arranged as we expect. Sorting order is nothing but the arrangement of the data in some specific style. Generally sorting order is of two types:

1. **Descending Order:** it is the sorting order in which the data or elements are arranged in form of greater to a smaller value. In other words, elements are in decreasing order.

Example: 115, 315, 415, 215, 515, 110

Above example is arranged in descending order after applying some sorting methods

515, 415, 315, 215, 115, 110

2. **Ascending Order:** It is the sorting order in which the data or elements are arranged in form of smaller to a greater value. In other words, elements are in increasing order. Example: 115, 315, 415, 215, 515, 110

Above example is arranged in ascending order after applying some sorting methods

110, 115, 215, 315, 415, 515

Efficiency: In sorting technique one of the most important issues analyses the efficiency of the sorting algorithm. We generally denote the efficiency of sorting algorithm in terms of time complexity. The time complexities are given in terms of big-O notations. Generally, time complexities for various sorting algorithms are $O(n^2)$ and $O(n\log n)$. The sorting techniques such as bubble sort, insertion sort, selection sort, shell sort has the time complexity $O(n^2)$ and the sorting techniques such as merge sort, quick sort, heap sort has time complexities as $O(n\log n)$. Efficiency also depends on a number of records or data to be sorted. Sorting efficiency is nothing but it only shows how much time that algorithm has taken to sort the elements.

Passes: Sorting the data or elements in some particular order there is a group of the arrangement of elements. The phases in which the elements are moving to obtain their proper position is called **passes**.

Example: 100, 300, 200, 500, 400. If the above example data sorted in ascending order then we use some phases: Pass 1: 100, 200, 300, 500, 400

Pass 2: 100, 200, 300, 400, 500

Above method, we can see that data is getting sorted in two passes. By applying logic as a comparison of each element with its neighboring element gives result in two passes.

9.4. SORTING TECHNIQUES

There are various sorting algorithms for sort elements such as: Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, Heap sort, Radix sort or Bucket sort.

9.4.1. Bubble Sort

Bubble sort also is known as “sorting by exchange,” in order to find the successive smallest or greatest elements; the entire method relies strongly on the exchange of the neighboring element. This approach of sorting requires “ $n - 1$ ” passes, to sort the given list of data in some proper order. Assume that if “ n ” number of elements present in an array “A.” The first pass starts with the comparison of the value of n^{th} and $(n - 1)^{\text{th}}$ position element. If the “ n^{th} ” position element is smaller than the $(n - 1)^{\text{th}}$ position element then these two elements are interchanged. The smaller value is compared with the value of the $(n - 2)^{\text{th}}$ position element. And required, the element is interchanged to place the smaller among the two in the $(n - 2)^{\text{th}}$ position. Elements which have a small value that move or “bubble up.” The entire process is to continue in this manner and the first pass ends with the comparison and possible exchange of elements A[1] and A[0]. The whole sorting method terminated with “ $(n - 1)$ ” passes, thereby resulting into a sorted list.

The algorithm of Bubble Sort:

Step 1: Read the total number of elements say n .

Step 2: Store the elements in an array.

Step 3: Set the initial element $i = 0$.

Step 4: Compare the adjacent elements.

4 (a). For ascending order, if the first element is smaller than the second element then no interchange of this element. Otherwise, interchange the element position.

4 (b). For descending order, if the first element is smaller than the second element then interchange of this element. Otherwise no interchange the element position.

Step 5: Repeat step 4 for all n elements.

Step 6: Increment the value of i by 1 and repeat step 4, 5 for $i < n$.

Step 7: Print the sorted list of elements.

Step 8: Stop.

Example: Consider 6 unsorted elements are given that elements are sort in **ascending order** 415, 515, 315, 910, 710, 310. Suppose an array “A” consists of 6 elements as:

415	515	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 1: In this pass, each element will be compared with its neighboring element one by one if the first element is smaller than the second element then no interchange of this element otherwise change the element position.

415	515	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[0] = 415 and A[1] = 515, Is $415 > 515$ is false, so no Interchange.

415	515	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[1] = 515 and A[2] = 315, Is $515 > 315$ is true so Interchange position, A[1] = 315 and A[2] = 515.

415	315	515	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[2] = 515 and A[3] = 910, Is $515 > 910$ is false so no Interchange.

415	315	515	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[3] = 910 and A[4] = 710, Is $910 > 710$ is true so Interchange. A[3] = 710 and A[4] = 910.

415	315	515	710	910	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[4] = 910 and A[5] = 310, Is $910 > 310$ is true so Interchange. A[4] = 310 and A[4] = 910.

415	315	515	710	310	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the first pass, the array will hold the elements which are sorted to some level.

Pass 2: Repeat above process again from the first element.

Compare A[0] = 415 and A[1] = 315, Is 415 > 315 is true so Interchange.
A[0] = 315 and A[1] = 415.

415	315	515	710	310	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[1] = 415 and A[2] = 515, Is 415 > 515 is false so no Interchange.

315	415	515	710	310	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[2] = 515 and A[3] = 710, Is 515 > 710 is false so no Interchange.

315	415	515	710	310	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[3] = 710 and A[4] = 310, Is 710 > 310 is true so Interchange.
A[3] = 310 and A[4] = 710.

315	415	515	310	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[4] = 710 and A[5] = 910, Is 710 > 910 is false so no Interchange.

315	415	515	310	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the second pass, the array will hold the elements which are sorted to some level.

Pass 3: Repeat above process again from the first element.

Compare $A[0] = 315$ and $A[1] = 415$, Is $315 > 415$ is false so no Interchange.

315	415	515	310	710	910
A_0	A_1	A_2	A_3	A_4	A_5

Compare $A[1] = 415$ and $A[2] = 515$, Is $415 > 515$ is false so no Interchange.

315	415	515	310	710	910
A_0	A_1	A_2	A_3	A_4	A_5

Compare $A[2] = 515$ and $A[3] = 310$, Is $515 > 310$ is true so Interchange.
 $A[2] = 310$ and $A[3] = 515$.

315	415	310	515	710	910
A_0	A_1	A_2	A_3	A_4	A_5

Compare $A[3] = 515$ and $A[4] = 710$, Is $515 > 710$ is false so no Interchange.

315	415	310	515	710	910
A_0	A_1	A_2	A_3	A_4	A_5

Compare $A[4] = 710$ and $A[5] = 910$, Is $710 > 910$ is false so no Interchange.

315	415	310	515	710	910
A_0	A_1	A_2	A_3	A_4	A_5

After the third pass, the array will hold the elements which are sorted to some level.

Pass 4: Repeat above process again from the first element.

315	415	310	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[0] = 315 and A[1] = 415, Is 315 > 415 is false so no Interchange.

315	415	310	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↔		↔			

Compare A[1] = 415 and A[2] = 310, Is 415 > 310 is true so Interchange.
A[1] = 310 and A[2] = 415.

315	310	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↔		↔			

Compare A[2] = 415 and A[3] = 515, Is 415 > 515 is false so no Interchange.

315	310	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↔		↔			

Compare A[3] = 515 and A[4] = 710, Is 515 > 710 is false so no Interchange.

315	310	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↔		↔			

Compare A[4] = 710 and A[5] = 910, Is 710 > 910 is false so no Interchange.

315	310	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the fourth pass, the array will hold the elements which are sorted to some level.

Pass 5: Repeat above process again from the first element.

315	310	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[0] = 315 and A[1] = 310, Is 315 > 310 is true so Interchange.
A[0] = 310 and A[1] = 315.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[1] = 315 and A[2] = 415, Is 315 > 415 is false so no Interchange.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[2] = 415 and A[3] = 515, Is 415 > 515 is false so no Interchange.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[3] = 515 and A[4] = 710, Is 515 > 710 is false so no Interchange.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[4] = 710 and A[5] = 910, Is 710 > 910 is false so no Interchange.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Finally, at the end of the last pass, the array will hold the entire sorted element in ascending order.

310	315	415	515	710	910
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Since the comparison positions look like bubbles, there for it is called **bubble sort**.

Example: Consider 6 unsorted elements are given that elements are sort in descending order

415, 515, 315, 910, 710, 310.

Suppose an array “A” consists of 6 elements as:

415	515	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 1:

In this pass, each element will be compared with its neighboring element one by one if the first element is smaller than the second element then interchange of this element positions otherwise no interchange the element positions.

415	515	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[0] = 415 and A[1] = 515, Is 415 < 515 is true so Interchange positions, A[0] = 515 and A[1] = 415

515	415	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[1] = 415 and A[2] = 315, Is 415 < 315 is false so no Interchange position,

515	415	315	910	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[2] = 315 and A[3] = 910, Is 315 < 910 true so Interchange positions, A[2] = 910 and A[3] = 315.

515	415	910	315	710	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[3] = 315 and A[4] = 710, Is 315 < 710 is true so Interchange.
 A[3] = 710 and A[4] = 315.

515	415	910	710	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[4] = 315 and A[5] = 310, Is 315 < 310 is false so no Interchange.

515	415	910	710	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the first pass, the array will hold the elements which are sorted to some level.

Pass 2: Repeat above process again from the first element.

515	415	910	710	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[0] = 515 and A[1] = 415, Is 515 < 415 is false so no Interchange.

515	415	910	710	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[1] = 415 and A[2] = 910, Is 415 < 910 is true so Interchange positions. A[1] = 910 and A[2] = 415

515	910	415	710	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



Compare A[2] = 415 and A[3] = 710, Is 415 < 710 is true so Interchange positions. A[2] = 710 and A[3] = 415

515	910	710	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[3] = 415 and A[4] = 315, Is 415 < 315 is false so no Interchange.

515	910	710	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[4] = 315 and A[5] = 310, Is 315 < 310 is false so no Interchange.

515	910	710	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the second pass, the array will hold the elements which are sorted to some level.

Pass 3: Repeat above process again from the first element.

515	910	710	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[0] = 515 and A[1] = 910, Is 515 < 910 is true so Interchange positions. A[0] = 910 and A[1] = 515

910	515	710	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[1] = 515 and A[2] = 710, Is 515 < 710 is true so Interchange positions. A[1] = 710 and A[2] = 515

910	710	515	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[2] = 515 and A[3] = 415, Is 515 < 415 is false so no Interchange.

910	710	515	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[3] = 415 and A[4] = 315, Is $415 < 315$ is false so no Interchange.

910	710	515	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Compare A[4] = 315 and A[5] = 310, Is $315 < 310$ is false so no Interchange.

910	710	515	415	315	310
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

After the third pass, the array will hold the elements which are sorted to some level.

Finally, at the end of the last pass, the array will hold the entire sorted element in **descending order**.

Program 9.1: Sorting the Elements by Bubble Sort Algorithm

```
Program: #include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int a[20], n, c, d, swap;
printf("Enter number of elements\n");
scanf("%d," &n);
printf("Enter %d integers\n," n);
for (c = 0; c < n; c++)
scanf("%d," &a[c]);
for (c = 0 ; c < (n - 1); c++) {
for (d = 0 ; d < n - c - 1; d++) {
```

```

if (a[d] > a[d+1]) /* For descending order use < */
{
    swap = a[d];
    a[d] = a[d+1];
    a[d+1] = swap;
}
}

printf("Sorted list in ascending order:\n");
for (c = 0 ; c < n ; c++)
printf("%d\n", a[c]);
getch();
}

```

The output of the program is given in Figure 9.1.

```

Enter number of elements
6
Enter 6 integers
25
67
87
98
45
12
Sorted list in ascending order:
12
25
45
67
87
98

```

Figure 9.1: Output of Bubble Sort.

Complexity Analysis: The complexity of any sorting method depends on number of comparisons. The number of passes may vary from 1 to $(n - 1)$, but the number of comparison required in a pass is not dependent on data. For the i^{th} pass, the number of comparisons required is $(n - 1)$. In the **Best case**, the Bubble sort performs only one pass and gives the sorted number, which gives $O(n)$ complexity. The number of comparison required is obviously $(n - 1)$. This case arises when the given list of array is sorted. In the **worst case**, performance of the Bubble sort is given by-

$$= n(n - 1) = O(n^2)$$

9.4.2. Insertion Sort

Inserting the sort method is based on the concept of inserting records into an existing file. To insert a record or data, we must find the appropriate place where the insertion is to be made. To find this place, we need to search. Once we have found the correct place, we need to move the records to make a place for the new record. In this sorting, we combine the two operations searching and sorting.

The algorithm of Insertion Sort:

Step 1: Read the total number of data or elements declare n.

Step 2: Store the elements in an array.

Step 3: Put the initial element position i = 1.

Step 4: Compare the element (which we want to sort) to the next element of the array.

If the key \leq array

Then

Move down the next array element by one.

Else

Insert the key into an array.

Step 5: Repeat step 4 for all n elements.

Step 6: Increment the value of i by 1 and repeat step 4, 5 for $i < n$.

Step 7: Print the sorted list of elements.

Step 8: Stop.

Example: Consider 6 unsorted elements are: 130, 170, 120, 150, 140, 110

Assume an array “A” consists of 6 elements as:

130	170	120	150	140	110
A_0	A_1	A_2	A_3	A_4	A_5

Pass 1:

Compare $A[1] > A[0]$ or $170 > 130$, true to the position of the elements remain same.

130	170	120	150	140	110
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 2:

Compare A[2] > A[1] or 120 > 170, false so interchange the position of elements. And A[1] > A[0] or 120 > 130, false so interchange the position of elements.

120	130	170	150	140	110
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 3:

Compare A[3] > A[2] or 150 > 170, false so interchange the position of elements. And A[2] > A[1] or 150 > 130, true to the position of the elements remain same.

120	130	150	170	140	110
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 4:

Compare A[4] > A[3] or 140 > 170, false so interchange the position of elements. And A[3] > A[2] or 140 > 150, false so interchange the position of elements. A[2] > A[1] or 140 > 130 , true to the position of the elements remain same.

120	130	140	150	170	110
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 5:

Compare A[5] > A[4] or 110 > 170, false so interchange the position of elements. And A[4] > A[3] or 110 > 150, false so interchange the position of elements. A[3] > A[2] or 110 > 140 , false so interchange the position of elements. A[2] > A[1] or 110 > 130, false so interchange the position of elements. And A[1] > A[0] or 110 > 120, false so interchange the position of elements.

110	120	130	140	150	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Finally, at the end of the last pass, the array will hold the entire sorted element like this

110	120	130	140	150	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Program 9.2: Sorting the Elements by Insertion Sort Algorithm

Program: #include<stdio.h>

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

```
void main() {
```

```
clrscr();
```

```
int n, a[20], c, d, temp;
```

```
printf("Enter number of elements\n");
```

```
scanf("%d," &n);
```

```
printf("Enter %d integers\n," n);
```

```
for (c = 0; c < n; c++) {
```

```
scanf("%d," &a[c]); }
```

```
for (c = 1 ; c <= n - 1; c++) {
```

```
    d = c;
```

```
    while (d>0 && a[d-1]> a[d]) {
```

```
        temp = a[d];
```

```
        a[d] = a[d-1];
```

```
        a[d-1] = temp;
```

```
        d--; } }
```

```
printf("Sorted list in ascending order:\n");
```

```
for (c = 0; c <= n - 1; c++)
```

```
{
```

```
    printf("%d\n," a[c]);
```

```
}
```

```
getch();
```

```
}
```

The output of the program is given in Figure 9.2.

```
Enter number of elements
7
Enter 7 integers
23
45
64
34
42
14
25
Sorted list in ascending order:
14
23
25
34
42
45
64
```

Figure 9.2: Output of Insertion Sort.

Complexity: an array of elements is nearly sorted then it is **Best case** complexity. The best case time complexity of insertion sort is $O(n)$. If an array of the element is randomly then it results in **Average case** time complexity which is $O(n^2)$. If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

9.4.3. Selection Sort

In the selection sort method, select the first element and scan starts from the first element and searches the entire array list until it finds the minimum value of this element and swaps it with the first element. The sort places the minimum value in the first place, selects the second element and searches for the second smallest element. This process continues until the complete list is sorted.

The algorithm of Selection Sort:

Step 1: Read the total number of elements declare n.

Step 2: Store the elements in an array.

Step 3: Set the initial element $i = 0$ or min.

Step 4: repeat step 9 while ($i < n$)

Step 5: $j = i + 1$

Step 6: repeat step 8 while ($j < n$)

Step 7: if $A[i] > A[j]$ then

$\text{temp} = A[i]$

$A[i] = A[j]$

$A[j] = \text{temp}$

Step 8: $j = j + 1$

Step 9: $i = i + 1$

Step 10: Print the sorted list of elements.

Step 11: Stop.

Now we consider an unsorted array and perform the selection operation using algorithm:

Example: Consider 6 unsorted elements are:

170, 145, 125, 150, 190, 120

Assume an array “A” consists of 6 elements as an Initially set array list

170	145	125	150	190	120
A_0	A_1	A_2	A_3	A_4	A_5

↑
Min

Pass 1:

170	145	125	150	190	120
A_0	A_1	A_2	A_3	A_4	A_5

↑ ↑ scan from A_1 in array find the

Min smallest element to min value

170	145	125	150	190	120
A_0	A_1	A_2	A_3	A_4	A_5

↑ ↑
i the smallest element to min value

Now swap $A[i]$ with the smallest element. Then we get the array list,

120	145	125	150	190	170
A_0	A_1	A_2	A_3	A_4	A_5

Pass 2:

120	145	125	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑ i,	↑ scan from A ₂ find the smallest				
min	element to min value				

120	145	125	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑	↑				
i	smallest element to i value				

Now swap A[i] with the smallest element. Then we get the array list,

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 3:

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑ i,	↑ scan from A ₃ find the				
min	smallest element				

As there is no smallest element than 145 we will increment pointer i.

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑					
i					

Then we get the array list,

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 4:

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑ i, ↑ scan from A ₄					

find the smallest element to i value

As there is no smallest element than 150 we will increment pointer i.

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑ i					

We get the array list,

120	125	145	150	190	170
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Pass 5:

120	145	125	150	190	
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
↑ ↑ i, smallest					

Now swap A[i] with the smallest element. Then we get the array list,

120	125	145	150	170	190
A ₀	A ₁	A ₂	A ₃	A ₄	A ₅

Finally, we get the sorted element in the array list.

Program 9.3: Sorting the Elements by Selection Sort Algorithm

Program: #include<stdio.h>

```
#include<conio.h>
void main() {
    clrscr();
    int a[20], n, c, d, p, s;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c<n; c++)
        scanf("%d", &a[c]);
    for (c = 0;c<(N - 1); c++) {
        p = c;
        for (d = c+1;d<n;d++) {
            if(a[p]>a[d])
                p = d;
        }
        if(p!= c) {
            s = a[c];
            a[c] = a[p];
            a[p] = s;
        }
    }
    printf("Sorted list in ascending order:\n");
    for (c = 0 ; c<n; c++)
        printf("%d\n", a[c]);
    getch(); }
```

The output of the program is given in Figure 9.3.

```
Enter number of elements
8
Enter 8 integers
20
98
78
88
65
32
12
33
Sorted list in ascending order:
12
20
32
33
65
78
88
98
-
```

Figure 9.3: Output of Selection Sort.

Complexity: an array of elements is nearly sorted then it is **Best case** complexity. The best case time complexity of insertion sort is **O(n)**. If an array of element is randomly then it results in **average case** time complexity which is **O(n²)**. If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is **O(n²)**.

- Advantage:**
 1. Selection sort is faster than bubble sort.
 2. If an item is in its correct final position, then it will never be moved.
 3. The selection sort has better predictability, that is, the worst case time will differ little from its best.

9.4.4. Merge Sort

The merge sort is sorting algorithms that use the divide and conquer method. Merge sort on an input array with n elements consist of three steps:

Divide: Partition array into two sublists SL1 and SL2 with n/2 elements each.

Conquer: Then sort sub list SL1 and SL2.

Combine: Merge SL1 and SL2 into a unique sorted group.

Example: The entire process of merge sort is as follows-

[15] [17] [13] [16] [12] [18] [14] [11]

Pass 1: [15 17] [13 16] [12 18] [14 11]

Pass 2: [13 15 16 17] [11 12 14 18]

Pass 3: [11 12 13 14 15 16 17 18]

Sorted element 11 12 13 14 15 16 17 18

Program 9.4: Sorting the Elements by Merge Sort

```
Program: #include<stdio.h>
#include<conio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
void main() {
clrscr();
int a[20],n,i;
```

```
printf("Enter no of elements:");
scanf("%d",&n);
printf("Enter array elements:");
for(i = 0;i<n;i++)
scanf("%d",&a[i]);
mergesort(a,0,N - 1);
printf("\nSorted array is :");
for(i = 0;i<n;i++)
printf("%d ,",a[i]);
getch(); }

void mergesort(int a[],int i,int j) {
int mid;
if(i<j) {
mid = (i+j)/2;
mergesort(a,i,mid); //left recursion
mergesort(a,mid+1,j); //right recursion
merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
} }

void merge(int a[],int i1,int j1,int i2,int j2) {
int temp[50]; //array used for merging
int i,j,k;
i = i1; //beginning of the first list
j = i2; //beginning of the second list
k = 0;
while(i<=j1 && j<=j2) //while elements in both lists
{
if(a[i]<a[j])
temp[k++] = a[i++];
else
temp[k++] = a[j++];
```

```

}

while(i<=j1)      //copy remaining elements of the first list
temp[k++] = a[i++];
while(j<=j2)      //copy remaining elements of the second list
temp[k++] = a[j++];

//Transfer elements from temp [] back to a[]
for(i = i1,j = 0;i<=j2;i++,j++)
a[i] = temp[j];  }

The output of the program is given in Figure 9.4.

```

```

Enter no of elements:8
Enter array elements:10
24
54
44
88
66
33
99
Sorted array is :10 24 33 44 54 66 88 99 -

```

Figure 9.4: Output of Merge Sort element.

When Merge Sort Apply Two Unsorted List

Merging is the process of combining two or more sorted files into a third sorted file. Let ‘A’ be a sorted list combining ‘X’ number of elements and ‘B’ be a sorted list containing ‘Y’ number of elements. Then the operation that combines the elements A and B into new sorted list C with $Z = X + Y$ number of elements is called merging. Compare the smallest elements of A and B after finding the smallest, put it into new list C, the process is repeated until either list A or B is empty. Now place the remaining elements of A (or perhaps B) in C. the new list C contain the sorted elements which are equal to the total sum of elements A and B lists.

Algorithm: Given two sorted list A and B that consists of ‘X’ and ‘Y’ number of elements respectively. These algorithms merge the two lists and produce a new sorted list C.

Variable ‘ P_a ’ and ‘ P_b ’ keep track the location of the smallest element in A and B. variable P_c refers to the location in C to be filled.

Step 1: Set $P_a = 1$; $P_b = 1$; $P_c = 1$;

Step 2: loop comparisons

Repeat while ($P_a \leq X$ and $P_b \leq Y$)

If ($A[P_a] < B[P_b]$) then

Set $C[P_c] = A[P_a]$

Set $P_c = P_c + 1$

Set $P_a = P_a + 1$

else

$C[P_c] = B[P_b]$

Set $P_c = P_c + 1$

Set $P_b = P_b + 1$

Step 3: append C list with remaining elements in A (or B)

If ($P_a > X$) then

Repeat for $i = 0, 1, 2, \dots, Y - P_b$

Set $C[P_c + i] = B[P_b + i]$

End loop

Repeat for $i = 0, 1, 2, \dots, Y - P_a$

Set $C[P_c + i] = B[P_a + i]$

End loop

Step 4: finished.

For example: consider two sorted lists A and B is as follows:

A :	10	50	100	200	250
B:	70	140	210	280	350

The process of merging and sorting illustrated below, which will produce a new sorting list C. Initially: $P_a = 1$; $P_b = 1$; $P_c = 1$;

Step 1: compare $A[P_a]$ and $B[P_b]$ or ($A[1]$ and $B[1]$), $A[P_a] < B[P_b]$, ($10 < 70$) so put 1 in $C[P_c]$

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10				

$P_a = P_a + 1$, $P_a = 2$ $P_b = 1$ $P_c = P_c + 1$, $P_c = 2$

Step 2: compare $A[P_a]$ and $B[P_b]$ or ($A[2]$ and $B[1]$), $A[P_a] < B[P_b]$, ($50 < 70$) so put 5 in $C[P_c]$

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50			

$$P_a = P_a + 1 = 3 \quad P_b = 1 \quad P_c = P_c + 1 = 3$$

Step 3: compare A[P_a] and B[P_b] or (A[3] and B[1]), A[P_a] > B[P_b], (100 > 70) so put 70 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70		

$$P_a = 3 \quad P_b = P_b + 1 = 2 \quad P_c = P_c + 1 = 4$$

Step 4: compare A[P_a] and B[P_b] or (A[3] and B[2]), A[P_a] < B[P_b], (100 < 140) so put 100 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	

$$P_a = P_a + 1 = 4 \quad P_b = 2 \quad P_c = P_c + 1 = 5$$

Step 5: compare A[P_a] and B[P_b] or (A[4] and B[2]), A[P_a] > B[P_b], (200 > 140) so put 140 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140

$$P_a = 4 \quad P_b = P_b + 1 = 3 \quad P_c = P_c + 1 = 6$$

Step 6: compare A[P_a] and B[P_b] or (A[4] and B[3]), A[P_a] < B[P_b], (200 < 210) so put 200 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140 200

$$P_a = P_a + 1 = 5 \quad P_b = 3 \quad P_c = P_c + 1 = 7$$

Step 7: compare A[P_a] and B[P_b] or (A[5] and B[3]), A[P_a] > B[P_b], (250 > 210) so put 210 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140 200 210

$$P_a = 5 \quad P_b = P_b + 1 = 4 \quad P_c = P_c + 1 = 8$$

Step 8: compare A[P_a] and B[P_b] or (A[5] and B[4]), A[P_a] < B[P_b], (250 < 280) so put 250 in C[P_c]

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140 200 210 250

$$P_a = P_a + 1 = 6 \quad P_b = 4 \quad P_c = P_c + 1 = 9$$

Step 9: Append the element of B in C, As P_a > x so, put all the remaining element

ts of B in C and increment P_b and P_c respectively by 1 until the list B is also empty.

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140 200 210 250 280

$$P_a = 6 \quad P_b = P_b + 1 = 5 \quad P_c = P_c + 1 = 10$$

A :	10	50	100	200	250
B:	70	140	210	280	350
C:	10	50	70	100	140 200 210 250 280 350

$$P_a = 6 \quad P_b = P_b + 1 = 6 \quad P_c = P_c + 1 = 11$$

Now, P_b > y, this shows that B is also empty finally we have a sorted new list C as follows:

$$C = 10, 50, 70, 100, 140, 200, 210, 250, 280, 350$$

Program 9.5: Sorting the Elements by Merge Sort for Two Unsorted List

```

Program: #include<stdio.h>
#include<conio.h>
#define MAX 20
void main() {
    clrscr();
    int a[MAX],b[MAX];
    int m,n; // to read the size of two arrays
    int i,j,temp;
    clrscr();
    printf("Enter the 1st array size(1-20) :");
    scanf("%d",&m);
    printf("Enter the 2nd array size(1-20) :");
    scanf("%d",&n);
    printf("\nEnter the 1st array elements:\n");
    for(i = 0; i< m; i++)
        scanf("%d",&a[i]);
    printf("\nEnter the 2nd array elements:\n");
    for(j = 0; j< n; j++)
        scanf("%d," &b[j]);
    for(j = 0; j< n; j++) // to store 's' elements to first list
    {
        a[i] = b[j];
        i++;
    }
    printf("\nBefore descending, the merged array is:\n");
    for(i = 0; i< m+n; i++)
        printf("%5d",a[i]);
    for(i = 0; i< m+n; i++) // to arrange the first list elements in
    Descending order
    {
        for(j = i; j< (m+n)-1; j++)
        {
            if(a[i]< a[j+1]
            {
                temp = a[i];
                a[i] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

```

```

a[i] = a[j+1];
a[j+1] = temp; } } }
printf("\nAfter descending, the merged array is:\n");
for(i = 0; i < m+n; i++)
printf("%5d,"a[i]);
getch(); }

```

The output of the program is given in Figure 9.5.

```

Enter the 1st array size(1-20) :5
Enter the 2nd array size(1-20) :6

Enter the 1st array elements:
23
45
67
87
64

Enter the 2nd array elements:
78
98
21
33
11
22

Before descending, the merged array is:
 23  45  67  87  64  78  98  21  33  11  22
After descending, the merged array is:
 98  87  78  67  64  45  33  23  22  21  11

```

Figure 9.5: Output of merge sort for two unsorted list.

Complexity Analysis: When an array of elements is nearly sorted then it is **Best case** complexity. The best case time complexity of insertion sort is **O(nlog₂n)**. If an array of element is randomly then it results in **average case** time complexity which is **O(nlog₂n)**. If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is **O(nlog₂n)**.

9.4.5. Quick Sort

Quick sort algorithm is also based on the Divide and Conquer design method. In this at every step, each element is placed in its proper position. It performs well on a longer list. The three steps of quick sort are as follows:

Divide: given list of element divide the array list into two sub list that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element. The splitting of the array into two subarrays is based on the pivot element. All the elements that are less than pivot should be in left subarray and all the elements that are more than pivot should be in the right subarray.

Conquer: recursively sort the two sub-arrays. **Combine:** Combine all the sorted elements in a group to form a list of sorted elements.

Working of Quick Sort

First selecting a random ‘pivot value’ from the array (list). Then it partitions the list into elements that are less than the pivot and greater than the pivot. The problem of sorting a given list is reduced to the problem of sorting two sublists. By scanning that last element of the list from the right to left, and checks with the element. The comparisons of elements with the first element stop when we obtain the elements smaller than the first element. Thus, in this case, the exchange of both the elements takes place. The whole procedure continues until all the elements of the list are arranged in such a way that on the left side of the element (pivot), the elements are lesser and on the right side, the elements are greater than the pivot. Thus, the list is subdivided into two lists. The working of quicksort is illustrated in Figure 9.6.

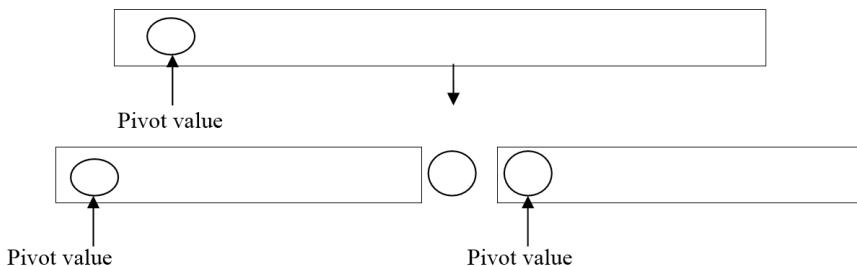
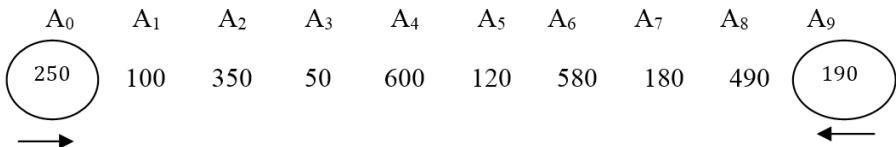


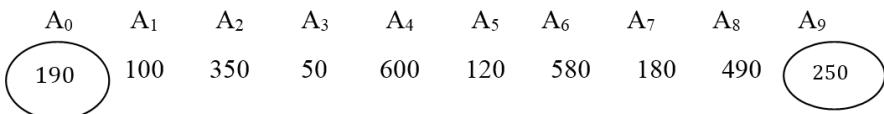
Figure 9.6: Quick Sort.

Example: Consider a list 250, 100, 350, 50, 600, 120, 580, 180, 490, 190 we have sorted the list using quick sort method.

Solution: Given

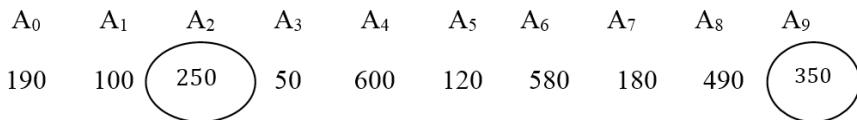


We use the first number 250. Beginning with the last number, 190, start scanning from the right to left, comparing each number with 250 and stop at the less than 250. The first number visited that has a value less than 250 is 190. Thus, interchange both of them.

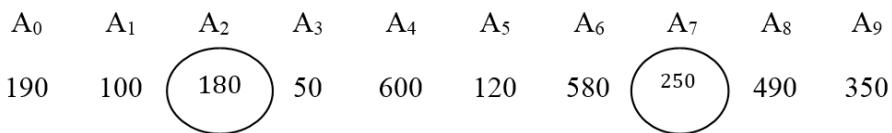


Scanning from left to right, the first number visited that has a value greater than 250 is 350.

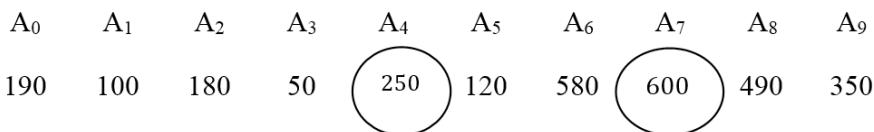
Thus, interchange both of them.



Scanning from right to left, the first number visited that has a value less than 250 is 180. Thus, interchange both of them.



Scanning from left to right, the first number visited that has a value greater than 250 is 600. Thus, interchange both of them.



Scanning from right to left, the first number visited that has a value less than 250 is 120. Thus, interchange both of them.

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
190	100	180	50	120	250	580	600	490	350

Thus 250 are correctly placed in its final position, and we get two Sublist-sublist1 and sublist2. Sublist1 has lesser value than 250 while sublist2 has greater values.

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
190	100	180	50	120	250	580	600	490	350
\longleftrightarrow					\longleftrightarrow				
Sublist1					Sublist2				

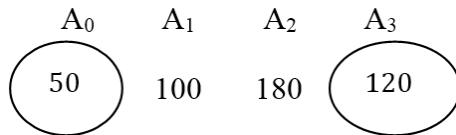
Now sorting of the first Sublist1.

A_0	A_1	A_2	A_3	A_4
190	100	180	50	120

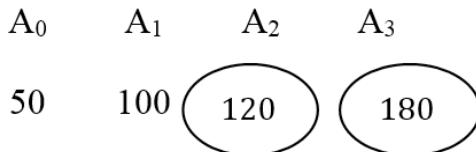
Beginning with the last number, 120, scanning from the right to left, comparing each number with 190 and stopping at the first number less than 190. The first number visited that has a value less than 190 is 120. Thus, interchange both of them.

A_0	A_1	A_2	A_3	A_4
120	100	180	50	190

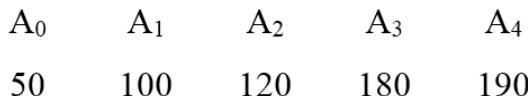
Now, 190 are correctly placed in its final position. Therefore we sort the remaining Sublist beginning with 120; we scan the list from right to left. The first number less than 120 are 50. We interchange 50 and 120 to obtain the list.



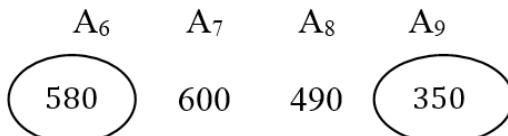
Beginning with 50 we scan the list from left to right. The first number greater than 120 is 180. We interchange 120 and 180 to obtain the list.



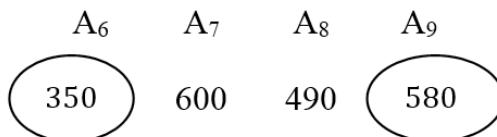
Hence the **first Sublist1** has been sorted as



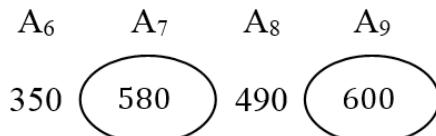
Now sorting of the first Sublist2.



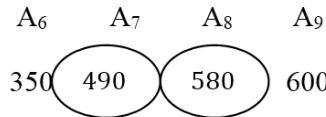
Beginning with 580 we scan the list right to left. The first number less than 580 are 350. We interchange 580 and 350 and obtain the list.



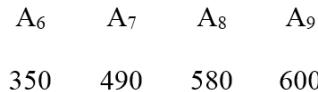
Beginning with 350 we scan the list from left to right. The first number greater than 350 are 600. We interchange 580 and 600 to obtain the list.



Beginning with 600 we scan the list right to left. The first number less than 580 are 490. We interchange 580 and 490 and obtain the list.



Hence, first **Sublist2** has been sorted as



The resulted sorted list is as follows-

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	A ₈	A ₉
50	100	120	180	190	250	350	490	580	600

Program 9.6: Sorting the Elements by Quick Sort Algorithm

```
Program: #include<stdio.h>
#include<conio.h>
void quicksort(int n[25],int first,int last){
int i, j, pivot, temp;
if(first<last){
    pivot = first;
    i = first;
    j = last;
    while(i<j){
        while(n[i]<=n[pivot]&&i<last)
            i++;
        while(n[j]>n[pivot])
            j--;
        if(i<j){
            temp = n[i];
            n[i] = n[j];
            n[j] = temp; } }
    temp = n[pivot];}
```

```

n[pivot] = n[j];
n[j] = temp;
quicksort(n,first,j-1);
quicksort(n,j+1,last); }

void main() {
clrscr();
int i, count, n[20];
printf("How many elements are u going to enter?: ");
scanf("%d",&count);
printf("Enter %d elements: ", count);
for(i = 0;i<count;i++)
scanf("%d",&n[i]);
quicksort(n,0,count-1);
printf("Order of Sorted elements: ");
for(i = 0;i<count;i++)
printf(" %d,%n[i]);
getch(); }

```

The output of the program is given in Figure 9.7.

```

How many elements are u going to enter?: 10
Enter 10 elements: 23
43
34
65
56
76
78
98
12
11
Order of Sorted elements: 11 12 23 34 43 56 65 76 78 98_

```

Figure 9.7: Output of Quick Sort element.

Complexity Analysis: When pivot is chosen such that the array gets divided at the mid then it gives the **best case** complexity. The best case time complexity of insertion sort is **O(nlog₂n)**. If an array is randomly then it results in **average case** time complexity which is **O(nlog₂n)**. The worst case for quick sort occurs when the pivot is minimum or maximum of all the elements in the list. Then it results in **worst case** time complexity which is **O(n²)**.

9.4.6. Heap Sort

The term heap can be defined as a heap of size n is a binary tree of n nodes that satisfy the two important properties regarding heap.

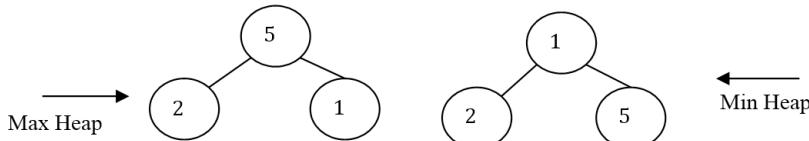
1. The heap must be either “almost complete binary tree” or “complete binary tree.”

Almost complete binary tree: The almost complete binary tree is a tree in which:

- (a) Each node has a left child whenever it has a right child.
 - (b) The leaf in a tree must be present at height h or $h - 1$. That means all the leaves are on two adjacent levels.
2. The heap must be either max heap (i.e., parent is greater than all its child nodes) or min heap (i.e., parent node is lesser than all child nodes).

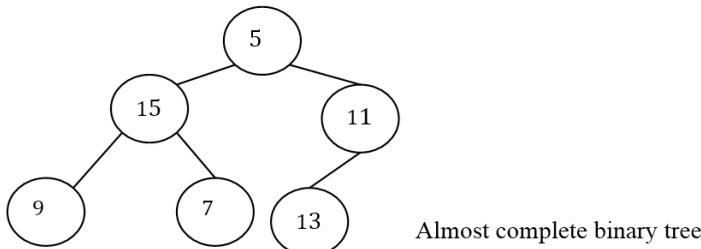
Heap sort is a sorting method discovered by J.W.J. Williams. It works in two stages, **Heap construction** and **Processing the Heap**.

Heap construction: heap is a tree data structure in which every parent node must be either greater than or lesser than its child nodes. Such heaps are called as max heap and min heap respectively.

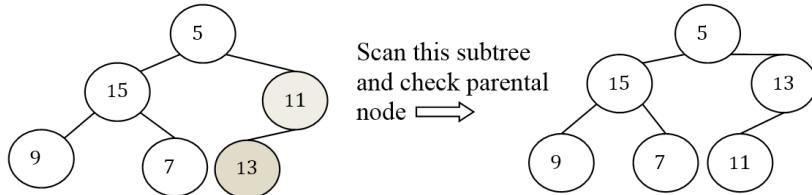


Example: Consider a list 5, 15, 11, 9, 7, 13 construct a heap.

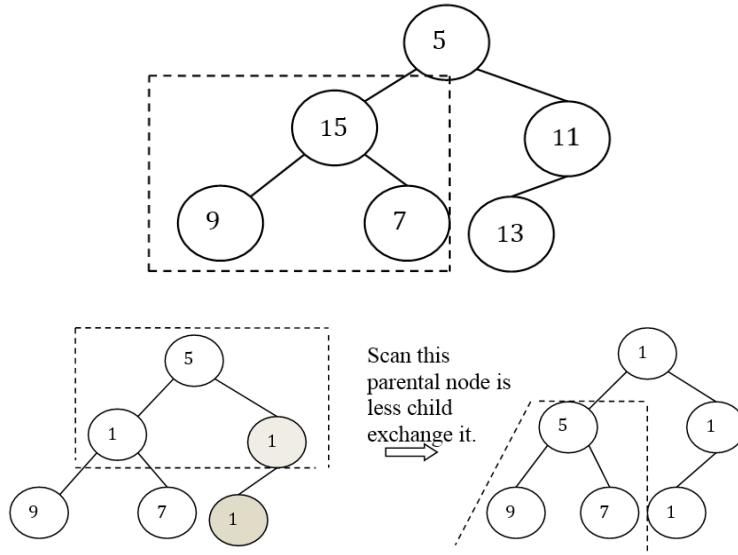
Solution: we will first create a complete binary tree or almost complete binary tree.



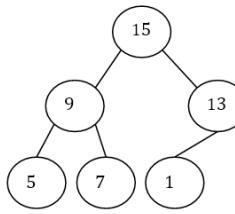
Now we will scan the tree from the bottom and check the parental property in order to build max heap.



As a parent is greater than its children no need to exchange it.



Thus heap is getting



Max Heap

Processing the Heap:

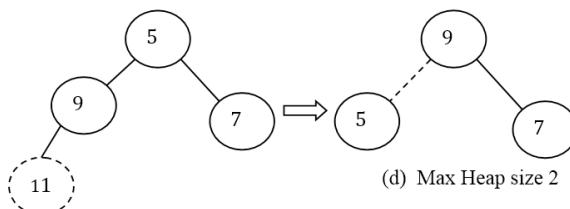
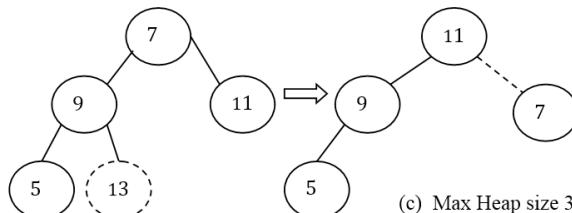
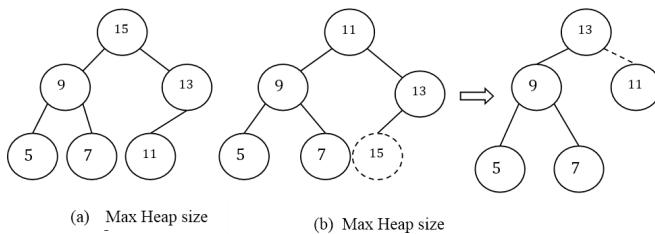
Heap may be represented as an array. The resulting heap depends on the initial Order of the unsorted list. For a different order of input list, the heap would be different. At this point, we have the heap of keys. We now have to process the heap in order to generate a sorted list of keys. This means we have traversed the heap in such a way so that the sorted keys are output. We know that the largest element is at top of the heap which is sorted in the

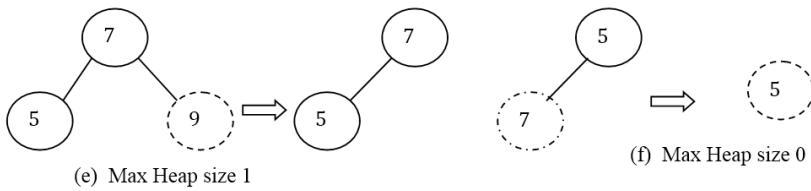
array at position heap [0]. We interchange heap [0] with the last element in the heap array heap [maxnodes], so that heap [0] is in the proper place. We then adjust the array to be a heap of size $n - 1$. Again interchange heap [0] with heap [$n - 2$], adjusting the array to be a heap of size $n - 2$ and so on. In the end, we get an array which contains the keys in the sorted order.

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
15	9	13	5	7	11

Array representation of a heap

The nodes which are moved to their final positions in the array are shown with a dashed circle, they are no longer part of the heap. A dashed line shows an edge whose two nodes have been interchanged to adjust the tree to be a heap again.





Now the sorted heap sort array is,

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
5	7	9	11	13	15

Program 9.7: Sorting the Elements by Heap Sort Algorithm (Max Heap)

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int heap[20], n, i, j, c, root, temp;
printf("\nEnter no of elements :");
scanf("%d", &n);
printf("\nEnter the number: ");
for (i = 0; i < n; i++)
scanf("%d", &heap[i]);
for (i = 1; i<n; i++)
{
c = i;
do
{
root = (c-1)/2;
if (heap[root] < heap[c]) /* to create MAX heap array */
{
temp = heap[root];
heap[root] = heap[c];
heap[c] = temp;
}
c = root;
}
while (c > 0 && heap[root] < heap[c]);
}
}
```

```
heap[c] = temp; }
c = root; }
while (c != 0); }
printf("Heap array : ");
for (i = 0; i < n; i++)
printf("%d," heap[i]);
for (j = N - 1; j >= 0; j--) {
temp = heap[0];
heap[0] = heap[j]; /* swap max element with rightmost leaf element */
heap[j] = temp;
root = 0;
do {
c = 2 * root + 1; /* left node of root element */
if((heap[c] < heap[c+1]) && c < j-1)
c++;
if (heap[root]<heap[c] && c<j) /* again rearrange to max heap array */
{
temp = heap[root];
heap[root] = heap[c];
heap[c] = temp;
}
root = c;
} while (c < j);
printf("\nThe sorted array is : ");
for (i = 0; i < n; i++)
printf("%d," heap[i]);
getch(); }
```

The output of the program is given in Figure 9.8.

```
Enter no of elements :10
Enter the number: 95
65
45
25
20
67
87
10
90
55
Heap array : 95 90 87 65 55 45 67 10 25 20
The sorted array is : 10 20 25 45 55 65 67 87 90 95_
```

Figure 9.8: Output of Heap Sort.

Complexity Analysis: **Worst case:** $O(n \log_2 n)$, **Average case:** $O(n \log_2 n)$, **Best case:** $O(n \log_2 n)$

9.4.7. Radix Sort or Bucket Sort

Radix Sort also known as Bucket Sort. Sorting can be done digit by digit and thus all the elements can be sorted. The following example illustrates this Radix sort method: Consider the unsorted array of 9 elements.

448, 243, 461, 523, 638, 228, 421, 643, 466

Step 1: in the first pass, sort the element according the units digits.

Unit digit	0	1	2	3	4	5	6	7	8	9
Elements	461,421		243,523,643			466		448,638,228		

Now sort this number

Unit Digits	Element
0	
1	421, 461
2	
3	243, 523, 643
4	
5	
6	466
7	

8	228, 448, 638
9	

Element after first pass:

421, 461, 243, 523, 643, 466, 228, 448, 638

Step 2: in the second pass, sort the element according to the tens digits.

Tens Digits	Element
0	
1	
2	421, 523, 228
3	638
4	243, 643, 448
5	
6	461, 466
7	
8	
9	

Element after second pass: 421, 523, 228, 638, 243, 643, 448, 461, 466

Step 3: in the third or final pass, sort the element according to the hundred digits.

Tens Digits	Element
0	
1	
2	228, 243
3	
4	421, 448, 461, 466
5	523
6	638, 643
7	
8	
9	

Element after third pass:

228, 243, 421, 448, 461, 466, 223, 638, 643.

Thus finally the sorted list by radix sort method will be,

228, 243, 421, 448, 461, 466, 223, 638, 643

Algorithm for radix sort:

- Read the total number of elements in the array.
- Store the unsorted elements in the array.
- Now sort the elements by digit by digit.
- Sort the elements according to the unit digit than tens digit than hundred and so on.
- Thus the elements should be sorted for up to the most significant bit.
- Store the sorted element in the array and print them.
- Stop.

Program 9.8: Sorting the Elements by Radix Sort Algorithm

Program:

```
#include<stdio.h>
#include<conio.h>
int largest(int a[], int n)
{
    int large = a[0], i;
    for(i = 1; i < n; i++){
        if(large < a[i])
            large = a[i]; }
    return large;
}
void Radix(int a[], int n)
{
    int bucket[10][10], bucket_c[10];
    int i, j, k, remainder, NOP = 0, divisor = 1, large, pass;
    large = largest(a, n);
```

```
printf("The large element %d\n",large);
while(large > 0) {
    NOP++;
    large/= 10; }

for(pass = 0; pass < NOP; pass++) {
    for(i = 0; i < 10; i++) {
        bucket_c[i] = 0; }
    for(i = 0; i < n; i++) {
        remainder = (a[i] / divisor) % 10;
        bucket[remainder][bucket_c[remainder]] = a[i];
        bucket_c[remainder] += 1; }

i = 0;
for(k = 0; k < 10; k++) {
    for(j = 0; j < bucket_c[k]; j++) {
        a[i] = bucket[k][j];
        i++; } }

divisor = divisor* 10;
for(i = 0; i < n; i++)
    printf("%d ,",a[i]);
    printf("\n"); }

}

void main()
{
    clrscr();
    int i, n, a[10];
    printf("Enter the number of elements :: ");
    scanf("%d",&n);
    printf("Enter the elements :: ");
    for(i = 0; i < n; i++) {
        scanf("%d,&a[i]); }

    Radix(a,n);
```

```
printf("The sorted elements are :: ");
for(i = 0; i < n; i++)
printf("%d ,",a[i]);
printf("\n");
getch();
}
```

The output of the program is given in Figure 9.9.

```
Enter the number of elements :: 10
Enter the elements :: 234
567
435
123
987
765
666
348
999
431
The large element 999
431 123 234 435 765 666 567 987 348 999
123 431 234 435 348 765 666 567 987 999
123 234 348 431 435 567 666 765 987 999
The sorted elements are :: 123 234 348 431 435 567 666 765 987 999
```

Figure 9.9: Output of Radix sort elements.

10

CHAPTER

SEARCHING AND HASHING

CONTENTS

10.1. Introduction	292
10.2. Searching	292
10.3. Hashing	296
10.4. Collision	300
10.5. Collision Handling Method	301
10.6. Rehashing	306
10.7. Application of Hashing	307

10.1. INTRODUCTION

Searching is a method for finding the particular or desired data element that has been stored with specific given identification is referred to as **searching**. Every day in our daily life, most the people spend their time in searching their data or keys. We are using **Key** as the identification of the data, which has to be searched. Although searching, we are asked to find a record that contains other information associated with the key. For example, given a name, we are asked to find a telephone number or given an account number, we are asked to find the balance in that account. The searching method is based completely on comparing keys. The organization of the file and the order in which the keys are inserted affect the number of keys that must be examined before getting the desired one. If the location of the record within the table depends only on the values of the key and not on the locations of their keys, we can retrieve each key in a single access. The most efficient way to achieve this is to store each record at a single offset from the base application of the table. This suggests the use of arrays. If the record keys are integers the keys themselves can serve as the index to the array. There is a one-to-one correspondence between keys and array index.

10.2. SEARCHING

Such a key is called an **internal key** or an **embedded key**. There may be a separate table of keys that includes a pointer to records, and then it will be necessary to store the records on secondary storage. This kind of searching where most of the table is kept in secondary storage is called **external searching**. Searching where the table to be searched is stored entirely in the main memory is called **internal searching**. There are two searching methods: **Linear Search** and **Binary Search**.

10.2.1. Linear or Sequential Searching

The sequential or linear search is the simplest search techniques. In this technique, we start at a beginning of a list or a table search for the desired data by examining each successive record until the desired data is found or the list is finished.

Algorithm for Linear search and Program explain in Chapter 2 in Program 2.4.

Complexity Analysis: **Worst case:** $O(n)$, **Average case:** $O(n)$, **Best case:** $O(1)$

The advantage of Linear Search:

- It is a simple and easy method.
- It is efficient for small lists.
- No sorting of items is required.

The disadvantage of Linear Search:

- It is not suitable for a large list of elements.
- It requires more comparisons.

10.2.2. Binary Searching

The sequential search algorithms have the problem; they walk over the entire list. Consider a list in ascending sorted order. It would work to search from the beginning until an item is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine which half the item is in (because the list is sorted). This effectively divides the working range in half with a single test. By repeating the procedure, the result is a highly efficient search algorithm called binary search.

Algorithm for Linear search: Binary_search (K, N,X)

Given an array K, consisting of N elements in ascending order, this algorithm searches the structure for a given element whose value is given by X. the variable LOW, MIDDLE, and HIGH denote lower, middle and upper limits respectively.

1. Initialize

LOW = 1

HIGH = N

2. Perform Search

Repeat through step 4 while

LOW ≤ HIGH

3. Obtain the index of the midpoint of the interval

MIDDLE = (LOW + HIGH) / 2

4. Compare

```

If X < K[MIDDLE]
Then HIGH = MIDDLE - 1
Else if X > K[MIDDLE]
    Then LOW = MIDDLE + 1
Else Print ("successful search")
Return (MIDDLE)
5.      Unsuccessful search
Print "unsuccessful search"
Return(0)

```

Example: Given an ordered set contains 8 data items (in an ascending order) 5 10 15 20 25 30 35 40 from the set we have to search the data item with X = 10.

Solution: initially, LOW = 1, HIGH = 8

The data items are arranged in the following manner along with their respective keys:

A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	A ₈
5	10	15	20	25	30	35	40

Step 1: MIDDLE = (LOW + HIGH) / 2

$$\text{MIDDLE} = (1 + 8) / 2 = 4$$

Now, K[MIDDLE] = 20, X ≠ K[MIDDLE] (10 ≠ 20)

X < K[MIDDLE], therefore

$$\text{HIGH} = \text{MIDDLE} - 1$$

$$\text{Now, HIGH} = 4 - 1 = 3$$

Step 2: MIDDLE = (LOW + HIGH) / 2

$$\text{MIDDLE} = (1 + 3) / 2 = 2$$

Now, K[MIDDLE] = 10

$$\mathbf{X = K[MIDDLE] = 10}$$

The search is successful as it searched for the desired data item, it requires 2 comparisons.

Program 10.1: For Binary search algorithm

Program: #include<stdio.h>

#include<conio.h>

```
void main() {  
    clrscr();  
    int c, first, last, middle, n, search, a[20];  
    printf("Enter number of elements\n");  
    scanf("%d",&n);  
    printf("Enter %d integers ascending order\n", n);  
    for (c = 0; c < n; c++)  
        scanf("%d",&a[c]);  
    printf("Enter value to find\n");  
    scanf("%d", &search);  
    first = 0;  
    last = n - 1;  
    middle = (first+last)/2;  
    while (first <= last) {  
        if (a[middle] < search)  
            first = middle + 1;  
        else if (a[middle] == search) {  
            printf("%d found at location %d.\n", search, middle+1);  
            break; }  
        else  
            last = middle - 1;  
        middle = (first + last)/2; }  
    if (first > last)  
        printf("Not found! %d is not present in the list.\n", search);  
    getch();  
}
```

The output of the program is given in Figure 10.1.

```
Enter no of elements :10
Enter the number: 95
65
45
25
20
67
87
10
90
55
Heap array : 95 90 87 65 55 45 67 10 25 20
The sorted array is : 10 20 25 45 55 65 67 87 90 95_
```

Figure 10.1: Output for Binary search.

Complexity Analysis: **Worst case:** $O(\log_2 n)$, **Average case:** $O(\log_2 n)$, **Best case:** $O(1)$

The advantage of Binary Search:

- It is more efficient than a linear search for a large number of elements in the list.
- It requires less number of comparisons.

The disadvantage of Binary Search:

- It requires the sorting of items.
- The ratio of insertion time or deletion time to search item is quite high for this method.

10.3. HASHING

The perfect relationship between the key value and the location of an element is not easy to establish or maintain. Consider if a college uses for students ID five digit number as the primary key. Now, the range of key values is from 00,000 to 99,999. It is clear that it will be impractical to set up an array to 1,00,000 elements of but each only 100 are needed. What if we keep the array size down to the size that we actually need (array of 100 elements) and just use the last two digits of the key to identifying each student? For instance, the element of student 53374 is in the student record.

Position	Key	Record
0	31600	
1	49101	
2	52302	
.	.	
.	.	
99	01899	

Definition: Hashing is an approach to convert a key into an integer within a limited range. This key to addressing conversion is known as a hashing function which maps the key space (K) into an address space (A). Thus, a hash function H produces a table address where the record may be located for the given key value (K).

The hashing function can be denoted as: **H: K → A.**

Ideally, no two keys should be converted into the same address. Unfortunately, there exists no hash function that guarantees this. This situation is called a collision. For example, the hash function in the preceding example is $h(k) = \text{key \% } 100$. The function $\text{key \% } 100$ can produce an integer between 0 and 99, depending on the value of the key.

10.3.1. Hash Table

A hash table or a hash map is a data structure that associates keys with their values. The primary operation it supports efficiently is a lookup: given a key (e.g., person's name), find the corresponding value (e.g., that person's telephone number).

It works by transforming the key using a hash function into a hash, a number that the hash table uses to locate the desired value. Using the hash key the essential data can be searched in the hash table by few or more key comparisons. The searching time is dependent upon the size of the hash table. Hashtable is a data structure that is used for storing and retrieving data very speedily. Insertion of data in the hash table is based on the key value.

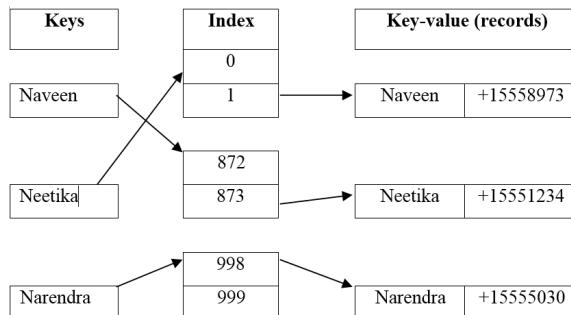


Figure 10.2: A small phone directory book as a hash table.

10.3.2. Hash Function

A hash function is a mathematical or logical function that is used to place the data in the hash table. For this reason, one can use the similar hash function to retrieve the data from the hash table. This hash function is used to implement the hash table. The integer returned by the hash function is called a **hash key**.

Example: Assume that we want to place some student records in the hash table. The record of a student is placed with the help of key: student ID. The student ID is a 7 digit number for placing the record in the hash table. To place the record, the key 7 digit number is converted into 3 digits by taking only last three digits of the key. If the key is 496700 it can be stored at 0th position. The second key is 8421002, the record of this key is placed 2nd position in the array. Hence the hash function will be: $H(\text{key}) = \text{key \% 1000}$, Where key \% 1000 is a hash function and key obtained by the hash function is called a hash key. The hash table will be:

	Student ID	Record
0	496800	
1		
2	7421002	
.	.	
.	.	
998	7886998	
999	1245999	

Hash function $H(key)$ is used to map some dictionary entries in the hash table. Each position of the hash table is called a bucket.

10.3.3. Types of Hash Function

There are different types of hash functions that are used to put the record in the hash table.

1. **Division method:** In the division method the hash function depends upon the remainder of the division. Typically the divisor is table length.

Example: If the record 64, 52, 99, 47 is to be placed in the hash table and table size is 10.

Hash Function: $H(key) = \text{record \% table size}$

$$4 = 64 \% 10$$

$$2 = 52 \% 10$$

$$9 = 99 \% 10$$

$$7 = 47 \% 10$$

Hence the record will be placed in the hash table with respect to 4, 2, 9 and 7 positions.

2. **Mid square:** In the mid square method, the hash function key is squared and middle or mid part of the result is used as the index. If the key is a string, it has to preprocess to produce a number. Consider that if we want to place a record 3111 then

$3111^2 = 9678321$ For the hash table size 1000

$H(3111) = 783$ (the middle 3 digits)

3. **Multiplicative hash function:** In the multiplicative hash function, the given record is multiplied by some constant value. The formula for computing the hash key is: $H(key) = \text{floor}(p * \text{fractional part of key} * A)$ where p is integer constant and A is a constant real number. Donald Knuth suggest using constant $A = 0.61803398987$

If key 107 and $p = 50$ then $H(key) = \text{floor}(50 * 107 * 0.61803398987)$

$$= \text{floor}(3306.4818458045) = 3306$$

At 3306 location in the hash table, the record 107 will be placed.

4. **Digit folding:** In the digit folding method, the key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.

Example: Consider a record 22365412 then it is divided into separate parts as 223 654 12 and these are added together. $H(\text{key}) = 223 + 654 + 12 = 889$. The record will be placed at location 889 in the hash table.

10.4. COLLISION

The hash function is a function that returns the key value, using this key the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at the appropriate position and due to this, we can retrieve the record directly from that location. This function needs to be designed very carefully and it should not return the same hash key address for two different records. The position in which the hash function returns the same hash key value for more than one record is called collision and two same hash keys returned for different records is called synonym. When there is no room for a new pair in the hash table such a situation is called overflow. Collision and overflow show the poor hash functions performance. Consider a hash function. $H(\text{key}) = \text{key \% 10}$ having the hash table of size 10. The record keys to be placed are 231, 54, 43, 78, 19, 36, 57 and 77. Hash Function: $H(\text{key}) = \text{record \% table size}$

$$1 = 231 \% 10,$$

$$4 = 54 \% 10,$$

$$3 = 43 \% 10,$$

$$8 = 78 \% 10,$$

$$9 = 19 \% 10,$$

$$6 = 36 \% 10,$$

$$7 = 57 \% 10,$$

$$7 = 77 \% 10.$$

In a hash table,

0	
1	231
2	
3	43
4	54
5	
6	36
7	57
8	78
9	19

Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called a **collision**. From the index 7 if we look for the next vacant position at subsequent indices 8, 9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

Features of Good Hashing Function:

- The hash function should be easy to compute.
- A number of collisions should be less while placing the record in the hash table. Ideally, no collision should occur. Such a function is called the perfect hash function.
- Hash functions should produce such a key will get distributed uniformly over an array.
- The function should depend upon every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

10.5 COLLISION HANDLING METHOD

An idea for the collision resolution strategy, if a collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique. Mainly there are two methods for detecting the collision and overflows in the hash table: **Chaining and linear probing**.

Two more difficult collision handling techniques are:

- Quadratic probing
- Double hashing

10.5.1. Chaining

In the collision handling method chaining is concepts that initiate an additional data field are called chain. A separate chain field is used to maintain for colliding data. When a collision occurs then a linked list (chain) is maintained at the home bucket. Chaining involves maintaining two tables in memory. First of all, as before, there is a table in memory which contains the records except that now has an additional field Link, which is used so that all records in table with same hash address H may be linked together to form a linked list second there is a hash address table List, which contains pointers to the linked list in table. Chaining hash tables have advantages over open-addressed hash tables in that the removal operation is simple and resizing the table can be postponed for a much longer time because performance degrades more gracefully even when every slot is used.

Example: Consider the keys to be placed in their home buckets are 14, 161, 131, 124, 19, 17, 97, 121, 44, 119 Then we will apply a hash function as $H(\text{key}) = \text{key \% D}$ Here D is the table size. The hash table will be: when $D = 10$,

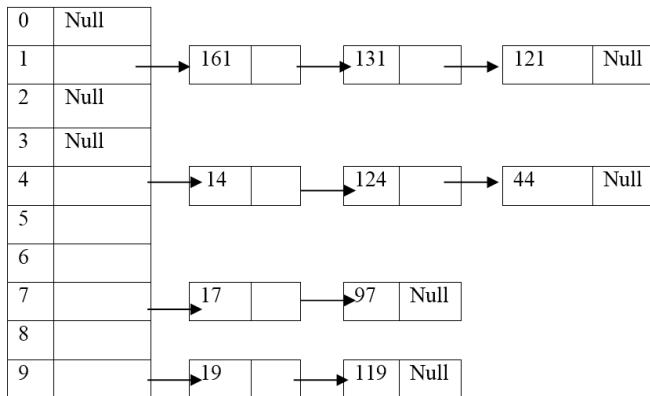


Figure 10.3: Chaining.

Above example, a chain field is maintained by colliding elements. For example, 161 have a home bucket (key) 1. In the same way, 121 and 131 keys demand for home bucket 1. The chain at index 4 and 7 is maintained.

10.5.2. Linear Probing (Open Addressing)

Linear probing is the easiest method of handling collision problem. When two records demand for the same location in the hash table then collision

problem occurs in the hash table. This collision problem can be solved by placing the second record linearly down whenever the empty location is found. When use linear probing the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we insert elements cans be inserted into the hash table.

Example: Consider the keys to be placed in their home buckets are 14, 161, 138, 127, 21, 15, 91, 121, 49, 119. We will apply a hash function; we will use division hash function. That means the keys are placed using the formula: $H(\text{key}) = \text{key \% table size}$

$$H(\text{key}) = \text{key \% 10}$$

For instance, the element 14 can be placed at $H(\text{key}) = 14 \% 10 = 4$, Index 4 will be the home bucket for 14. Continuing in this fashion we will place 1, 4, 7, and 8.

0	Null
1	161
2	Null
3	Null
4	14
5	Null
6	Null
7	127
8	138
9	Null

Table 1

0	Null
1	161
2	21
3	91
4	14
5	15
6	121
7	127
8	138
9	49

Table 2

0	119
1	161
2	21
3	91
4	14
5	15
6	121
7	127
8	138
9	49

Table 3

Now the next key to be inserted is 21. According to the hash function $H(\text{key}) = 21 \% 10$, $H(\text{key}) = 1$. But the index 1 location is already occupied by 161 i.e., collision occurs. To resolve this collision we will linearly move down and at the next empty location (Table2). Therefore 21 will be placed at index 2. 15 placed at index 5, 91 placed at index 3, 121 placed at index 6, 49 placed at index 9. Hence we will place 49 at index 9. Now the next element is 119 and its hash key 9. But the home bucket is used there is no next empty bucket in the table, to handle this problem we move back to bucket 0 and is the location over there is empty 119 will be placed at 0th index.

10.5.3. Quadratic Probing

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses formula: $H_i(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$. Here m can be a table size or any prime number.

Example: If we want to insert the following elements in the hash table with table size 10: 37, 90, 55, 42, 11, 17, 49, 87. We will fill the hash table step-by-step.

$$37 \% 10 = 7,$$

$$90 \% 10 = 0,$$

$$55 \% 10 = 5,$$

$$42 \% 10 = 2,$$

$$11 \% 10 = 1,$$

Hash Table:

0	1	2	3	4	5	6	7	8	9
90	11	42			55		37		

Now if we want to place 17 a collision will occur as $17 \% 10 = 7$ and bucket 7 has already an element 27. Hence we will apply quadratic to insert this record in the hash table. $H_i(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$. Consider $i = 0$ then $(17 + 0^2) \% 10 = 7$ and $(17 + 1^2) \% 10 = 8$ when $i = 1$. The bucket 8 is empty hence we will place the element at index 8. Now next element is 49 which will be placed at index 9.

$$\% 10 = 9$$

0	1	2	3	4	5	6	7	8	9
90	11	42			55		37	17	49

Now to place 87 we will use quadratic probing.

$$(87 + 0) \% 10 = 7$$

$$(87 + 1) \% 10 = 8 \quad \text{but already used}$$

$$(87 + 2^2) \% 10 = 1 \quad \text{already used}$$

$$(87 + 3^2) \% 10 = 6$$

0	1	2	3	4	5	6	7	8	9
90	11	42			55	87	37	17	49

It is observed that if we want to place all the necessary elements in the hash table the size of divisor (m) should be twice as large as a total number of elements.

10.5.4. Double Hashing

Double hashing is a method in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert. There are two important rules to be followed for the second function:

- It must never evaluate to zero.
- Must make sure that all cells can be probed.

The formula to be used for double hashing is $H_1(\text{key}) = \text{key mod table size}$, $H_2(\text{key}) = M - (\text{key mod } M)$, Here M is a prime number smaller than the size of the table. Let the elements 37, 90, 45, 22, 17, 49, 55 to be placed in the hash table size 10, Initially insert the elements using the formula for $H_1(\text{key})$.

Insert 37, 90, 45, 22

$$H_1(37) = 37 \% 10 = 7$$

$$H_1(90) = 90 \% 10 = 0$$

$$H_1(45) = 45 \% 10 = 5$$

$$H_1(22) = 22 \% 10 = 2$$

$$H_1(49) = 49 \% 10 = 9$$

0	1	2	3	4	5	6	7	8	9
90		22			45		37		49

Now if 17 is to be inserted then $H_1(17) = 17 \% 10 = 7$, $H_2(\text{key}) = M - (\text{key mod } M)$, Here M is a prime number smaller than the size of the table. A prime number that is smaller than the table size of 10 is 7. Hence $M = 7$, $H_2(17) = 7 - (17 \bmod 7) = 7 - 3 = 4$.

That means we have to insert the element 17 at 4 places from 37. In short we have to take 4 jumps. Therefore the 17 will be placed at index 1. Now to insert 55,

$$H_1(55) = 55 \% 10 = 5$$

$$H_2(55) = 7 - (55 \bmod 7) = 7 - 6 = 1$$

0	1	2	3	4	5	6	7	8	9
90	17	22			45	55	37		49

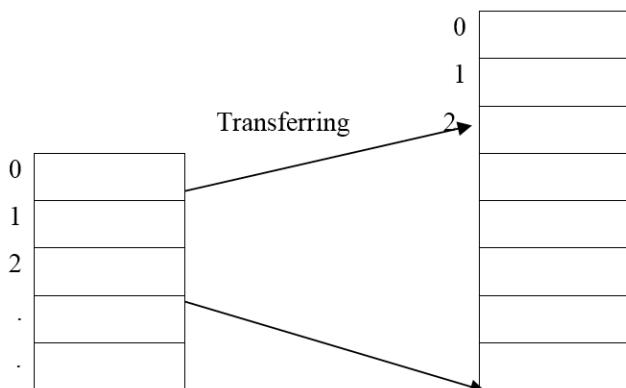
That means we have taken one jump from index 5 to place 55. Finally, this is a hash table.

Difference between Quadratic Probing and Double Hashing

- The double hashing is more complex to implement than quadratic probing. The quadratic probing is a fast technique than double hashing.
- The double hashing requires another hash function whose efficiency is same as some another hash function required when handling random collision.

10.6. REHASHING

Rehashing is a method by which the table is resized; the size of the table is doubled by creating a new table. It is preferable if the total size of the table is a prime number. There is some situation in which the rehashing is required. A) When the hash table is completely full. B) Quadratic probing when the table is filled half. C) When insertions fail due to overflow. In such situations, we have to transfer entries from the old table to the new table by re-computing their positions using suitable hash functions.



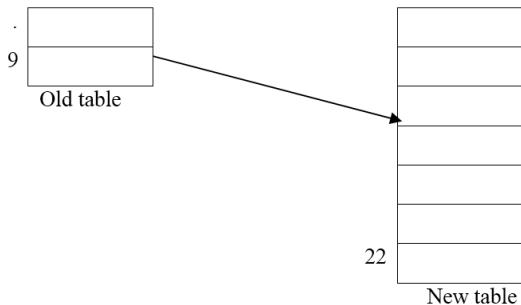


Figure 10.4: Rehashing.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. The table size is 10 and will use hash function, $H(key) = \text{key mod table size}$

$37 \% 10 = 7, 90 \% 10 = 0, 55 \% 10 = 5, 22 \% 10 = 2, 17 \% 10 = 7$ collision solved by linear probing $49 \% 10 = 9$

0	1	2	3	4	5	6	7	8	9
90		22			55		37	17	49

Hence we will rehash by doubling the table size. The old table size is 10 then we should this size for the new table, which becomes 20. But 20 is not a prime number, we will prefer to make the table size of 23. Now hash function will be

$$H(key) = \text{key mod } 23$$

$$37 \% 23 = 14, \quad 90 \% 23 = 21, \quad 55 \% 23 = 9, \quad 22 \% 23 = 22, \quad 17 \% 23 \\ = 17,$$

$$49 \% 23 = 3, \quad 87 \% 23 = 18$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
									55					37			17	87			90	22

Now the hash table is sufficiently large to accommodate new insertions.

10.7. APPLICATION OF HASHING

- Hash tables are commonly used for symbol tables, caches, and sets.
- In compiler to keep track of declared variables.

- For online spelling checking the hashing functions are used.
- Hashing helps in the game playing programs to store the moves made.
- For browser program, while caching the web pages, hashing is used.
- In computer chess, a hash table is generally used to implement the transposition table.

INDEX

A

abstract data type 3
adjacency matrix 224
adjacent elements 130
Administration queues 128
Algebraic expressions 169
algorithms 9, 10, 106, 108, 116
alphanumeric data 55
arithmetic operation 98
array 5, 6, 22, 27, 29, 30, 31, 32
artificial intelligence 130
Ascending priority queue 124
asymptotic notions 10
Average case 262, 286
Axioms 3

B

Backtracking 74, 75
Balance trees 196
bandwidth 125

base address 44, 45, 46
Bellman-Ford algorithm 240
Best case 258, 262, 267, 274, 286
big-O notations 247
Binary 165, 166, 167, 168, 172,
 173, 174, 175, 176, 177, 179,
 180, 182, 183, 192
Binary operators 95
Binary Search 292, 296
binary search tree 179, 184, 185,
 186, 191, 197, 198, 205, 210
binary tree 165, 166, 167, 168, 170,
 172, 173, 174, 176, 178, 180,
 196, 210
breadth-first traversal 226
browser program 308
B+ tree 210
B-Trees 206, 207
Bubble sort 247, 248, 258

C

caching 308
 Chaining 301, 302
 circular linked list (CLL) 145
 circular list 145, 151
 circularly linked list 145
 C language 13, 14, 16
 collision 297, 300, 301, 302, 303,
 304, 305, 306, 307
 collision handling technique 301
 collision resolution strategy 301
 Compaction 61
 Complete graph 216
 complexity 247, 258, 262, 267,
 274, 280
 Complexity Analysis 258, 274,
 280, 286
 Computer Programming 2
 condensed matter physics 243
 Connected Graph 216
 conquer method 267
 contiguous allocation 30
 copy node 188

D

Data 1, 2, 3, 4, 5, 8, 19, 24, 26
 database 79, 80
 data element 292
 Data Structures 1, 2, 4
 data type 36
 decimal number 99, 100
 degree of Node 163
 delete operation 114
 deletion 37, 53
 Deletion 49, 52, 53, 54, 55, 56
 deletion procedure 190
 Depth-first search traversal (DFS)
 225

descending order 246, 247, 248,
 254, 257, 262, 267, 274
 Descending priority queue 124
 Destination queues 128
 diagonal element 224
 digit folding method 300
 Dijkstra's algorithm 240, 241
 dimensional arrays 42
 directed graph 215, 239
 discrete events 125
 disk accesses 206, 211
 Divide and Conquer design method
 274
 dot operator 27
 double-ended queue 119
 Double hashing 301, 305
 double rotation 205
 Doubly Circular Linked List 135
 Doubly Linear Linked List 135
 D-queue 106, 119, 120
 Dynamic memory allocation 28, 29

E

edges 7, 214, 215, 216, 217, 224,
 230, 231, 236, 239, 240
 Efficiency 247
 elementary items 2
 embedded key 292
 Expression conversion 94
 Expression evaluation 94
 external nodes 165, 168
 external searching 292
 External sorting 246

F

factorial functions 65
 Fibonacci numbers 68
 Fibonacci sequence 69

Fibonacci series 68
finite element technique 60
finite number 5, 8
finite set 8
first popped operand 98
float data type 5
front pointer 107, 108, 115, 116
function 3, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 26, 29, 30,
31, 32

G

garbage 60, 61
General graph 217
graph 7, 12
Graphs 214
graph theory 214, 239, 243

H

Hamiltonian circuits 239
hardware 9
hash function 297, 298, 299, 300,
301, 302, 303, 305, 306, 307
hash key 297, 298, 299, 300, 301,
303
Hashtable 297
header node 145, 151, 152
head recursion 71
Heap construction 281
Heap Sort 281, 284, 286
home bucket 302, 303
homogeneous data elements 37

I

index set 37
Infix Expression 95, 97
Information Processing Language
130

inorder successor 190, 205
input string 102
inserting data item 198
Insertion 37, 49, 51, 52, 56
insertion sort 247, 262, 267, 274,
280

Insertion Sort 259, 261, 262

integer 2, 5, 7, 20
integer numbers 36
internal key 292
internal nodes 165, 168, 211
internal searching 292
Internal sorting 246
internetworking 243

J

Jarnik-Prim's Algorithm 236
J.W.J. Williams 281

K

Kenneth Appel 214
Key 292, 297
key value 296, 297, 300
Kruskal's algorithm 231, 236, 239

L

last in first out (LIFO) 6
left-skewed binary tree 168
Leonhard Euler's paper 214
linear array 37, 49, 55, 56
linear data structure 37, 78
linear list 56
linear probing 301, 303, 307
linear Queue 114
Linear Search 292, 293
linked lists 162, 169
Local Area Network (LAN) 243
logic 247

M

matrices 42, 57, 58, 60
matrix array 42
max heap 281, 285
maxnodes 283
member access operator 23, 24
memory 2, 6, 9, 17, 19, 20, 28, 29,
 30, 31, 32
Memory fragmentation 37
merging 268, 269, 270
Message Queuing 128
min heap 281
Multigraph 217
multinational company 246
multiway search tree 206

N

negative number 67, 68, 69
node 75, 76, 130, 131, 132, 133,
 135, 136, 137, 138, 139, 140,
 145, 146, 147, 151, 152, 153,
 154, 158, 159
nonnegative functions 12
Non-terminal nodes 164
null pointer 21

O

one-dimensional array 38, 57
operand 95, 97, 98
Operators 95
ordered list 56, 57
output 86, 88, 94, 101, 104
overflow 300, 301, 306

P

palindrome 86, 87
parallel edges 217
Parameters 15

partial differential equations (PDEs)
 60
passes 247, 248, 258
payroll system 246
pictorially representation 214
placeholder 15
placeholder variable 159
pointer 130, 131, 133, 145, 151,
 152, 159
pointer variable 19, 20, 21, 25
polish notation 95
polynomials 57, 158
pop function 83
pop operation 80, 83
Postfix Expression 95
Predecessor 130
predictability 267
Prefix Expression 95, 97
primary data structure 130
primary key 296
primitive data 5
Prim's algorithm 236, 237, 239
Priority Queue 124, 125, 128
Processing the Heap 281, 282
program 39, 40, 42, 46, 47, 48, 51,
 52, 54, 55, 58, 59, 61
programming code 64, 72
push operation 80, 82

Q

Quadratic probing 301, 304, 306
queue 106, 107, 108, 109, 110, 111,
 115, 116, 119, 120, 121, 124,
 125, 126, 128
Queue elements 107
Queue underflow 107
Quick sort algorithm 274

R

Radix Sort 286, 288
 real values 5
 rear 106, 107, 108, 109, 110, 111, 112, 113, 115, 116, 117, 118, 119, 120, 121, 123, 124, 125
 recursion 64, 66, 71, 72, 74
 recursive algorithm 65, 73
 recursive calls 66
 recursive function 64, 65, 66, 68, 69, 71
 red-black tree 191, 192
 Rehashing 306, 307
 Report queues 128
 Response queues 128
 reversed string 102
 Right-Left (RL) Rotation 202
 right-skewed binary tree 168
 root node 162, 164, 165, 170, 172, 173, 174, 175, 176, 182, 183, 184, 185, 186, 198, 206, 210
 Rudolf Bayer 191

S

Searching 49, 50, 51, 56, 291, 292, 293
 second popped operand 98
 segment 214
 Selection Sort Algorithm 265
 self-balancing binary search tree 191
 self-loop 217, 224
 sequential manner 162, 170
 sequential search algorithms 293
 simple graph 217
 simulation modeling 125
 single dimensional array 57
 single node 162, 185, 198

Singly Circular Linked List 135
 Singly Linked List 135, 158
 Software development 2
 Sorting 36, 49, 55, 245, 246, 247, 257, 261, 265, 267, 273, 279, 284, 286, 288
 sorting algorithms 247, 267
 spanning tree 229, 230, 231, 232, 233, 236, 237, 238, 239, 240
 sparse array 58
 sparse matrices 58
 special node 151
 stack 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 96, 97, 98, 99, 100, 101, 102, 103
 stack_empty 83
 Stack machine 95
 stack underflow 83
 Static memory allocation 130
 strictly binary tree 166
 String Palindrome 95
 structure elements 23
 subgraph 216, 230
 subitems 2
 subscript 38, 43
 subsets 162, 165
 subtree 164, 166, 169, 172, 173, 174, 175, 176, 177, 180, 182, 183, 184, 186, 189, 191, 197, 205, 206, 207, 211
 subtrees 76, 163, 165, 179, 182, 189, 190, 193, 196, 206
 Successor 130
 symmetric binary B-trees 191
 syntax 37

T

tables 42

tail recursion 71

Terminal Nodes 163, 164

termination condition 66

token number 106

Tower of Hanoi 72, 73

Traveling salesman problem 239

Traversing 49

tree 162, 163, 164, 165, 166, 167,
168, 169, 170, 171, 172, 173,
174, 175, 176, 177, 178, 179,
180, 182, 183, 184, 185, 186,
187, 188, 189, 190, 191, 192,
195, 196, 197, 198, 199, 200,
201, 202, 203, 205, 206, 207,
208, 209, 210, 211

Tree 7

tree data structure 281

two-dimensional array 43, 57

U

Unary operators 95

Undirected Graph 215, 216, 223,

230, 232, 233, 237

Unweighted shortest path 240

Update 49, 54

user-defined data type 22

user-defined functions 14

V

variable 2, 5, 9, 14, 16, 19, 20, 21,

23, 24, 25, 27, 29, 32

vertex 216, 217, 224, 225, 226,

230, 236, 243

vertices 7, 214, 215, 216, 217, 225,

226, 229, 230, 236, 239

W

web pages 308

Wide Area Networking 243

Wolfgang Haken 214

worst case 258, 262, 267, 274, 280