

Debdutta Pal • Suman Halder

Data Structure & Algorithm

with **C**



Alpha Science

Data Structures and Algorithms with C

Data Structures and Algorithms with C

**Debdutta Pal
Suman Halder**



**Alpha Science International Ltd.
Oxford, U.K.**

Data Structures and Algorithms with C

352 pages

Debdutta Pal

Suman Halder

Department of Computer Science and Engineering
Calcutta Institute of Engineering and Management
Kolkata

Copyright © 2018

ALPHA SCIENCE INTERNATIONAL LTD.

7200 The Quorum, Oxford Business Park North
Garsington Road, Oxford OX4 2JZ, U.K.

www.alphasci.com

ISBN 978-1-78332-368-5

E-ISBN 978-1-78332-427-9

Printed from the camera-ready copy provided by the Authors.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher.

*Dedicated
to
Our Parents*

Preface

This book is designed as a stepping stone for the students to enter into the world of computer science and engineering. All the students irrespective of different domain of technology if want to build carrier in the Computer Science and Engineering domain have to have enough knowledge in data structures. The Data structure is a course that helps students to accumulate knowledge about how data are stored and manipulated in the memory of any computing device. This book is written for the students who have knowledge about C and now are going to open their eyes in the domain of data structure. Hence, the prospective audience for this book consists primarily of sophomores and juniors majoring in computer science or computer engineering.

Chapter 1, in this book is designed in such a way that the students having no knowledge in data structure can boldly step forward into the zone of data structure. This chapter has a detail overview on a data structure that includes definition of data structure, the categorisation of data structure. Operation on different data structure is also included in this chapter. The chapter ends with application of data structure.

Chapter 2, can be defined as a soul of this book. This chapter includes details of the algorithm. Definition of algorithm, life cycle of algorithm. Time complexity and space complexity measurement is explained in this chapter. Students will come to know about asymptotic notation and the manipulation of asymptotic notation in this chapter. Classification of the algorithm is also a major part of this chapter.

In the chapter 3, array is defined in broad way. The chapter includes details of one dimensional (1-D) array. Multidimensional array is also explained here. Address calculations of one and two dimensional array are shown in this chapter. Representation of a sparse matrix is given here. Manipulation of string is a major part of this chapter.

Chapter 4 can be defined as a heart of this chapter. This includes classification of pointers and array of pointers. Dynamic memory allocation (DMA) is one of the major contributions in this chapter. Students will learn about different predefined functions that are used for DMA. They will also build their knowledge about need of DMA and the shortcomings of DMA. Dynamically allocated spaces are how to de-allocate that is also explained here. Creation of one dimensional and two dimensional arrays using dynamic memory allocation is explained here.

In the chapter 5, detail of another a linear data structure is described. The name of this data structure is “Link List”. This chapter initiates with linear link list and finally it concludes with a doubly link list. Creation, traversal, insertion, deletion, reverse operations on the single link list, circular link list and the doubly link list are explained in broad way. Both algorithms and programs are given for each operation to make this chapter more student approachable. This chapter is application of DMA.

Chapter 6 deals with different aspects of Stack and Queue. The first part of this chapter explains Stack definition and basic operations on Stack. It is followed by the application of Stack in POSTFIX, PREFIX expression. Implementations of Stack using link list and array are also included in this chapter. Second part of this chapter is all about Queue. Definition of the Queue and basic operations on queue are specified here. Then categorisation of Queue is also given here. Different types of Queues like circular queue, de-queue and priority queue are explained with example. Each operation on Stack and Queue is accompanied with algorithm and programs.

viii Preface

In chapter 7, recursion is explained in a comprehensive way. When students reach at this chapter they have acquired knowledge about non recursive functions. After completion of this chapter, students will come to know about recursive function. This chapter deals with the need of recursion, depth of recursion, type of recursion and the shortcomings of recursive functions. This chapter concludes with bags of programs on recursion that make students more familiar with recursion.

Chapter 8 involves a non-linear data structure. Until now the students accumulate their knowledge about the linear data structure. With this chapter they will reveal another look of data structure. This chapter focuses on Tree structure. This chapter initiates with binary tree followed by a binary search tree and end with multi way search tree. Each type of tree structure is explained with some basic operations on tree like creation, insertion of node, traversal, and deletion of nodes. Height balance tree is one of the major contributions of this chapter.

Chapter 10 focuses on another very important part of data structure that is Graph. Here, the graph is explained in detail. The chapter introduces the different terminology of graph like nodes, link, adjacent node, complete graph, path, circuit, walk, degree of vertex, etc.; it is followed by the memory representation of the graph. Then memory representation of a graph is explained in terms of the adjacency matrix, adjacency list, multi list, incident matrix. Then different graph traversal algorithm is explained. Spanning tree formation is also shown here by the means of Prim's algorithm and Kruskal algorithm. The chapter also includes some algorithm to find shortest path using Bellman-Ford algorithm, Dijkstra's algorithm, and Floyed-Warshall algorithm. The chapter concludes with different application of graph in different aspects of computer science and engineering fields. To make it easy -for student programs are also written for all the algorithm explain in this chapter. Some lemmas are written for students who are going to study design analysis and algorithm.

In chapter 10, different types of searching and sorting algorithms are explained. For each searching and sorting algorithm recurrence relation is also derived using different approaches. The algorithms are explained with the help of examples in each case. Each sorting and searching algorithm is accompanied with programs so that the students can easily understand the algorithm and run the program without any difficulties. This chapter includes some basic sorting algorithm like bubble sort, insertion sort, merge sort, quick sort and also some advance sorting technique like radix sort, heap sort and shell sort.

Chapter 11 deals with hashing technique. It explains different terminology relates to hashing like hash function, hash table. This chapter also describes different types of hashing procedure with examples. Collision resolution techniques are also described in this chapter. Here, creation of hash table, insert in a hash table and delete from a hash table is explained with algorithm and program.

Chapter 12, the concluding chapter of this book gives a glimpse on file handling. Here, different ways of storing records in a file are explained. Indexing of records and searching of records from a file are also described in this chapter.

**Debdutta Pal
Suman Halder**

Acknowledgement

I am grateful to the Almighty, whose grace and blessings empowered me at every juncture. This book would not be completed without the contribution and support that I have acquired from many people. I would like to take the opportunity to express my heartfelt thanks to these individuals.

I would like to thank my co-author Mr. Suman Halder, who encourage me to contribute in this book. I am very grateful to my colleagues Mrs. Smita Basu, Mr. MD.Sahid Ahmed and Mr, Sudipta Basak for their valuable suggestions. I wish to acknowledge my student whose feedback helps me to make this book student centric.

Narosa Publishing House deserves my sincere thanks for providing me such a platform to share my knowledge with the universe.

Last but not the least, a word of thanks goes to my parent's in law, my beloved father Dipak Barman Roy and mother Mala Barman Roy, my husband Supriyo Pal and my little daughter Ishaani . Without their love, patience and continuous encouragements I would never be able to complete this book.

Debdutta Pal

The successful completion of any task would be incomplete without giving gratitude to the people who made it possible and whose constant encouragement made these efforts with success. The acknowledgement transcends with my deep gratitude to all the people, who inspired, supported and helped me in the completion of this book.

I express my deep-felt thanks to the co-author Dr. Debdutta Pal, who continuously inspires me in the completion of this book. I am very grateful to my colleagues at Calcutta Institute of Engineering and Management, Mrs. Smita Basu, Mr. MD.Sahid Ahmed and Mr, Sudipta Basak for their valuable constructive suggestions and genuine feedback. I extend my thanks to Mr. Surjakanta Sasimal who provided me valuable materials.that helped me in writing this book.

I would like to thank the management, editorial and production unit of Narosa Publishing House, for publishing this book in time. Suggestions and queries are most welcome through publishers.

Last but not the least, a word of thanks goes to my beloved parents, my elder sister, wife Rupa and my little son Avinaba for their love, constant encouragement and blessings.

Suman Halder

Contents

<i>Preface</i>	<i>vii</i>
<i>Acknowledgments</i>	<i>ix</i>
CHAPTER 1 INTRODUCTION	1.1
Data and Information	1.1
Representation of Data	1.2
Data Type	1.3
Data Structure	1.3
Classifications of Data Structures	1.4
Application of Data Structures	1.6
Abstract Data Type	1.6
Operations Perform on Data Structure	1.7
Overview of Different Data Structures	1.7
<i>Summary</i>	1.8
<i>Exercises</i>	1.8
CHAPTER 2 ANALYSIS OF ALGORITHM	2.1
Algorithm	2.1
Types of Algorithm	2.1
Algorithm Development Life Cycle	2.2
Analysis of Algorithm	2.3
Space Complexity	2.4
Time Complexity	2.5
Asymptotic Notations	2.7
Growth functions of Algorithm	2.10
Manipulation of Asymptotic Notations	2.13
<i>Summary</i>	2.15
<i>Exercises</i>	2.15
CHAPTER 3 ARRAY AND STRING	3.1
Array	3.1
One-dimensional array	3.1

Address calculation in One-dimensional array	3.5
Multi-dimensional array	3.5
Address calculation in two-dimensional array	3.6
Operations Perform on Array	3.9
Applications of Array	3.15
Representation of Polynomials	3.15
Sparse Matrix	3.16
Strings	3.21
Array of strings	3.23
Operations Perform on Strings	3.24
<i>Summary</i>	3.31
<i>Exercises</i>	3.31

CHAPTER 4 POINTER

Pointer Declaration	4.1
Address of Operator	4.1
Indirection Operator	4.3
Null Pointer	4.4
void Pointer	4.5
Generic Functions	4.6
Dangling Pointer	4.8
Arithmetic Operation with Pointer	4.8
Pointer to Pointer	4.8
Pointers and Arrays	4.13
Array of Pointers	4.15
Pointer to an Array	4.15
Pointer to Function	4.16
Passing addresses to Function	4.17
Function returning Pointer	4.18
Dynamic Memory Allocation	4.19
Creating one-dimensional array	4.22
Creating two-dimensional array	4.23
Pointers, Arrays and Strings	4.27
<i>Summary</i>	4.28
<i>Exercises</i>	4.28

CHAPTER 5 LINKED LIST

Limitations of Array	5.1
Linked List	5.2
Singly Linked list	5.3

Operations on Singly linked list	5.3
Representation of polynomials using linked list	5.17
Circular Linked list	5.19
Operation on Circular Link List	5.20
Josephus Problem	5.25
Doubly Linked list	5.26
Operation on Doubly Link List	5.26
Circular Doubly Linked List	5.36
Disadvantages of Linked List	5.37
<i>Summary</i>	5.37
<i>Exercises</i>	5.37
CHAPTER 6 STACK AND QUEUE	
Stack	6.1
Operations on Stack	6.1
Stack Representation with Array	6.2
Stack Representation with Linked List	6.4
Processing of function calls	6.5
Evaluation of Arithmetic expressions	6.6
Queue	6.11
Operations on Queue	6.11
Queue Representation with Array	6.12
Queue Representation with Linked List	6.13
Application of Queue	6.14
Drawback of Linear Queue	6.14
Circular Queue	6.15
Circular Queue Representation with Array	6.15
Dequeue	6.16
Operation on DeQueue	6.17
Priority Queue	6.17
Representation of Priority Queue	6.18
<i>Summary</i>	6.19
<i>Exercises</i>	6.20
CHAPTER 7 RECURSION	
Recursion Essentials	7.1
Infinite Regress	7.1
Depth of Recursion	7.2
Recursion Tree	7.2
Types of Recursion	7.2

xiv Contents

Factorial	7.6
Fibonacci Sequence	7.7
GCD	7.8
Integer Power	7.8
Tower of Hanoi	7.10
Non-attacking Eight Queens	7.12
Converting Recursive function to Iterative	7.16
<i>Summary</i>	7.17
<i>Exercises</i>	7.17

CHAPTER 8 TREE

Terminology of Tree	8.1
Binary Tree	8.2
Strictly Binary Tree	8.3
Extended Binary Tree	8.3
Complete Binary Tree	8.4
Full Binary Tree	8.4
Skewed Binary Tree	8.5
Binary Expression Tree	8.5
Balanced Binary Tree	8.7
Threaded Binary Tree	8.8
Properties of Binary Tree	8.9
Representation of Binary Tree	8.10
Binary Tree Traversal	8.13
Binary Search Tree	8.18
Operations on Binary Search Tree	8.18
Heap	8.26
Operations on Heap	8.27
AVL Tree	8.27
Operations on AVL Tree	8.27
Red-Black Tree	8.39
Multi-way Search Tree	8.40
B-Tree	8.41
Operations on B-Tree	8.42
2-3 Tree	8.50
2-3-4 Tree	8.50
<i>Summary</i>	8.51
<i>Exercises</i>	8.51

CHAPTER 9 GRAPH	9.1
Terminology of Graph	9.1
Terminology of a Directed Graph	9.6
Operations on Graph	9.9
Representation of Graph	9.9
Graph Traversal	9.16
Spanning Trees and Minimum Spanning Trees	9.21
Kruskal's Algorithm	9.23
Prim's Algorithm	9.24
Shortest Paths	9.25
Dijkstra's Algorithm	9.26
Bellman-Ford Algorithm	9.27
Floyd-Warshall Algorithm	9.28
Travelling Sales Man Problem	9.30
<i>Summary</i>	9.32
<i>Exercises</i>	9.32
CHAPTER 10 SEARCHING AND SORTING	10.1
Linear Search	10.1
Binary Search	10.3
Interpolation Search	10.6
Bubble Sort	10.8
Insertion Sort	10.12
Selection Sort	10.14
Quick Sort	10.15
Merge Sort	10.20
Heap Sort	10.24
Radix Sort	10.33
Shell Sort	10.35
Time complexity of Sorting Algorithms	10.37
<i>Summary</i>	10.38
<i>Exercises</i>	10.38
CHAPTER 11 HASHING	11.1
Hash Table	11.1
Hash Function	11.1
Division Method	11.2
Mid Square method	11.2
Folding method	11.3

Collision Resolution	11.3
Linear Probing	11.4
Quadratic Probing	11.7
Double Hashing	11.9
Separate Chaining	11.10
Load Factor	11.12
<i>Summary</i>	<i>11.12</i>
<i>Exercises</i>	<i>11.12</i>

CHAPTER 12 FILE STRUCTURE

Elements of File System	12.1
Category of File Organisation	12.2
Sequential File Organisation	12.3
Heap File Organisation	12.3
Hash File Organisation	12.4
Index Sequential File Organisation	12.4
Primary Index	12.4
Secondary Index	12.6
Clustering	12.6
Multilevel Index	12.7
B tree	12.7
B+ tree	12.7
<i>Summary</i>	<i>12.10</i>
<i>Exercises</i>	<i>12.10</i>

Appendix A

A.1

Solved GATE Question Papers

SP.1

Glossary

G.1

Bibliography

B.1

Index

I.1

CHAPTER 1

INTRODUCTION

"Those who wise succeed must ask the right preliminary questions". – Aristotle

In Computer Science, one of the core fields that belongs to its foundations, with the design, analysis, and implementation of algorithms for the efficient solutions of the problem concerned. The data structure is one of the subjects that intrinsically connected with the design and implementations of efficient algorithms.

The subject Data Structure deals with the study of methods, techniques, and tools to organize or structure data in computer memory.

Now before defining Data Structure, we should know –what is data?” and –what is the difference between data and information?”

Data and Information

Data is a plural of datum, which is originally a Latin noun meaning –something given.”

The Oxford dictionary meaning of data is:

- i) Facts or statistics used for reference or analysis.
- ii) The quantities, characters or symbols operated by a computer.

For our purpose, the second meaning is more important. Therefore, we can say that:

The data represent quantities, characters, or symbols on which operations are performed by a computer, stored and recorded on either magnetic, optical, or mechanical recording media, and transmitted in the form of digital electrical signals.

Definition: Data is the basic entity or fact that is used in a calculation or manipulation process.

Data is commonly processed by some stages. Unprocessed data or raw data is a collection of numbers, characters, that may be considered as an input of a stage and processed data is the output of the stage.

There are two types of data, such as numerical and alphanumerical data. Data may be a single value or a set of values and it is to be organized in a particular fashion. This organization or structuring of data will have a profound impact on the efficiency of the program.

Most of the individuals consider that the terms "Data" and "Information" are interchangeable and mean the same thing. However, there is a distinct difference between the two words. Data are raw facts without context, whereas Information is data with context. Data are an abstraction of Information in a problem-solving system. Data requires interpretation to become an Information.

Data can be any character, text, words, number, pictures, sound, or video and if not put into context means nothing to a human or computer. For example, 10409 is a data, whereas information

KEY FEATURES

-
-  Data and Information
 -  Data Structure
 -  Data Type
 -  Abstract Data Type
 -  Classification of Data Structure
-

1.2 | Data Structures and Algorithms with C

may be 1/04/09-the date of birth of Avinaba, 10409 a zip code of somewhere or Rs. 10409 is the salary of someone.

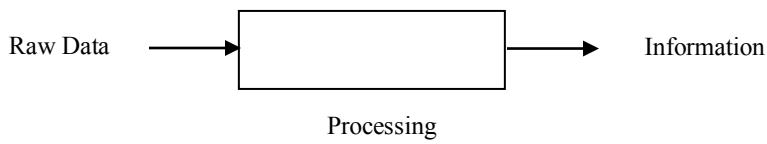


Figure 1.1: Data and Information

Representation of Data

Almost all the high-level languages, e.g. FORTRAN, COBOL, C, Pascal, C++, Java, etc. allow data to be represented in computer memory of mainly two types:

- i) Numerical Data
- ii) Alphanumeric Data

Most of the high-level languages classified numerical data into two types:

- i) Integer
- ii) Floating-point

In C language, the following Data types are used to represent numerical data.

Table 1.1: Ranges of numerical data types

Data types	Memory requirement	Ranges
char or signed char	1 byte	-128 to +127
unsigned char	1 byte	0 to 255
short signed int or short int or signed int or int	2 bytes	-32768 to +32767
long signed int or long int	4 bytes	-2147483648 to +2147483647
signed int or int (for 32 bit compiler)	4 bytes	-2147483648 to +2147483647
unsigned int	2 bytes	0 to 65,535
long unsigned int	4 bytes	0 to 4294967295
float	4 bytes	-3.4e38 to +3.4e38
double	8 bytes	-1.7e308 to +1.7e308
long double	10 bytes	-1.7e4932 to +1.7e4932

Integer

For small integer data, most of the high-level languages generally use two bytes to represent both positive and negative decimal integers. Negative integer numbers are represented by 2's complement notation. For large integer data, generally, four bytes are used to represent both positive and negative decimal integers.

Floating-Point

In the representation of small floating-point data, most of the high-level languages, e.g. C language, use four bytes for fractional decimal numbers. For large numbers, double and long double are used to accommodate in memory.

Alphanumeric data are classified into two types:

- i) Character
- ii) String

Character

The characters may be alphabets, digits, special characters and white spaces. In C language, characters are represented by char data type and one-byte memory space is used for storing the same. The ASCII format has been used in C language to represent the characters. While storing character ‘C’ in the computer, ASCII value 67 is stored in memory.

String

The string is a sequence of characters may consist of any number and any combination of characters. The characters may be alphabets, digits, special characters and white spaces. In C language, the string can be defined as an array of character terminated with a null character.

Data Type

Generally, computer programs do exist for a single purpose: how to process data. The type of data, the format of data that is going to be returned and the correctness of the processing are the primary concerns of a computer program. When a program is written, how the computer handles the data internally is usually a secondary concern.

Definition: A data type refers to the type of data that variables hold.

Now, depending on the representation of different forms of data, different data types are used. The data types are names given to a set of variables, which have common properties. A data type refers to the type of data that variables hold.

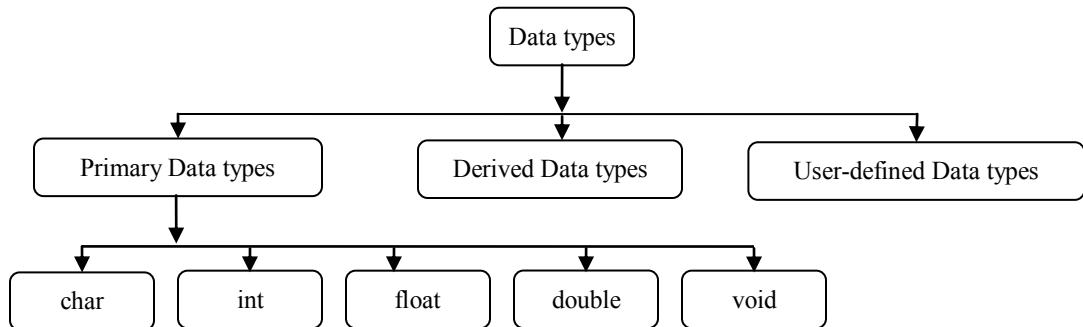


Figure 1.2: Data Types

The C language supports three classes of data types:

- i) Primary or Basic data types
- ii) Derived data types
- iii) User-defined data types

Basic or primary data types are language built-in. The C language supports five primary or basic data types. They are

- i) char
- ii) int
- iii) float
- iv) double
- v) void

Data Structure

Data Structure is the representation of the logical relationship between individual elements of data. In Computer Science, Data Structure is defined as a mathematical or logical model of organizing the data items into computer memory in such a way that they can be used efficiently. The purpose of studying

data structure is to learn how to organize data into memory so that it can be accessed quickly and conveniently.

Data Structure refers to the study of data and representation of data objects within the program; i.e., the implementation of structured relationships among different data objects.

Example: lists, stacks, queues, heaps, search trees, hash tables, etc.

Different sets of operations can be performed on different data structures. The operations that can be performed on different data elements within a data structure are accessing, traversing, inserting, deleting, modifying etc. While writing a program, a minimal data structure must be chosen that supports all the operations as per need.

Data Structure can be used for the following purpose:

- i) Organizing the data – How data items are organized in the main memory?
- ii) Accessing methods – How data items can be accessed?
- iii) Specifying the degree of associativity – How data items are interrelated?
- iv) Processing alternatives for data – How many different ways are there in which these data items can be processed?

A program is a set of instructions, which involve a computer performing some kind of computation or algorithm. Data Structure affects the design of both structural and functional aspects of a program. To implement a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore, the programs are inherited by an algorithm and its associated data structure.

Algorithm + Data Structure = Program

Programming languages provide facilities for representatives of algorithm and data. High-level Programming Language, like C, facilitates structured and modular programming by providing algorithm structures.

Classifications of Data Structure

Data Structure can be classified into two categories:

- i) Primitive data structure
- ii) Non-Primitive data structure

Primitive Data Structure

The primitive data structures are defined that can be manipulated or operated by the machine instruction. There are numerous types of data structures, generally built upon simpler primitive data types, called Primitive data structures, which are represented in computer memory.

Example: Integer, floating point, characters, pointer, boolean, etc. are some of the different primitive data structure.

In C language, the different primitive data structures are defined using the data type such as int, char, float, double etc.

Non-primitive Data Structure

The non-primitive data structures are a data structure that cannot be manipulated or operated directly by the machine instructions. These are more sophisticated data structures. Non-primitive data structures are derived from the primitive data structure. These are a group of homogeneous or heterogeneous data items.

Example: Arrays, structure, stack, queues, linked lists etc.

The Non-primitive data structures are classified into two categories:

- i) Linear data structure
- ii) Non-linear data structure

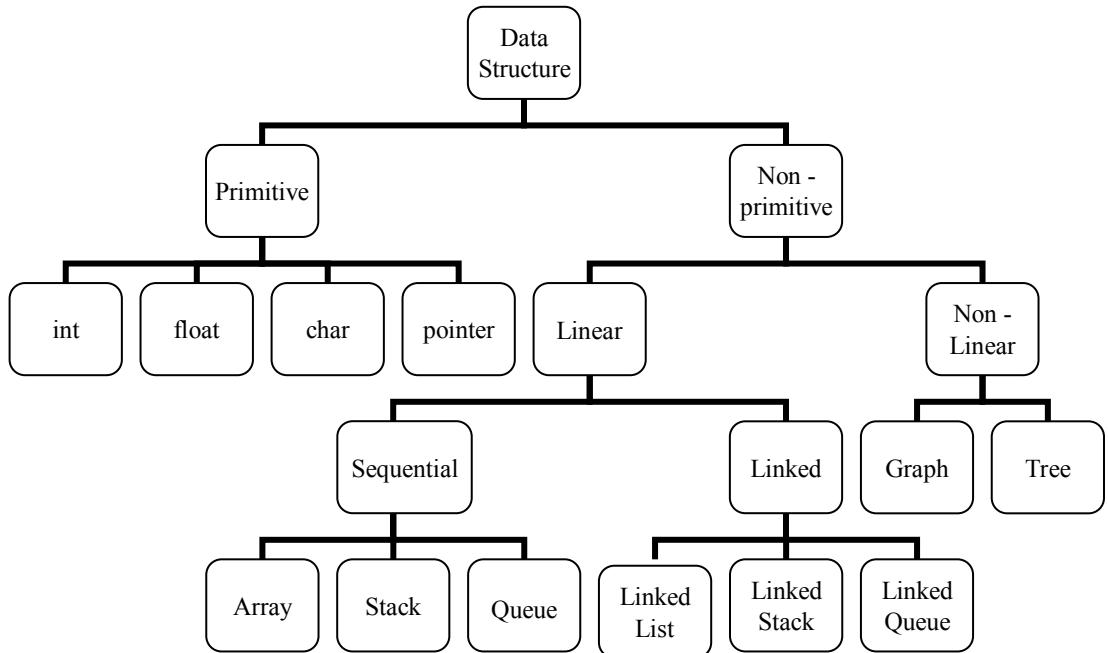


Figure 1.3: Classification of Data Structure

Linear Data Structures

The data structure is linear if every data item is related to its previous and next data items (e.g., array, linked list). In the linear data structure, data items are arranged in memory in a linear sequence and data items are accessed linearly. The traversing of the linear data structure is exactly once.

Linear data structures are two types:

- i) Sequential
- ii) Linked

Sequential Data Structures are based on arrays where objects are stored in a sequence of consecutive memory locations.

Example: Arrays, Stacks, Queues

Linked Data Structure is a data structure, which consists of a set of nodes linked together and organized with links.

Example: Linked Lists, Linked Stacks, Linked Queues

Non-linear Data Structures

A data structure is non-linear if every data item attaches to many other data items in specific ways to reflect relationships (e.g., tree). In non-linear data structures, the data elements are not in sequence, i.e., insertion and deletion are not possible in a linear manner. The traversing of the non-linear data

structure is always more than one.

Example: Graphs, Trees

Static Data Structure

The static data structure is a kind of data structure, in which once memory space is allocated it cannot extend, i.e. the memory allocation for the data structure takes place at compile-time that cannot be changed afterwards.

Example: Array

Dynamic Data Structure

Dynamic Data Structure is another kind of data structure, which can be extended or shrink during the execution, i.e., the memory allocation as well as memory de-allocation for the data structure takes place at run-time and allocates memory as required amount at any time.

Example: linked list, stack, queue, tree

Applications of Data Structure

Different Data Structure is used in real life, such as the representation of an image in the form of a bitmap, implement printer spooler so that jobs can be printed in the order of their arrival, store information about the directories and files in a system, etc. Data Structure is used in various fields of Computer Science, such as:

- Compiler Design
- Operating System
- Database Management System
- Statistical Analysis Package
- Numerical Analysis
- Graphics
- Artificial Intelligence
- Simulation

Different kinds of data structures are suitable for different kinds of applications. Some data structures are highly specialized for specific tasks. For example, databases use B-tree indexes for small percentages of data retrieval, and compilers and databases use dynamic hash tables as lookup tables, operating systems use queues for process management, I/O request handling.

Abstract Data Type

A data type refers to the type of data that variables hold. Thus, integer, real, characters are referred to as primitive data types.

Data Object represents an object having a data. The study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

Definition: An Abstract Data Type (ADT) describes the data objects, which constitute the data structure and the fundamental operations supported on them.

An ADT promotes data abstraction and focuses on what a data structure does, rather than how it implements (does).

- An ADT is a conceptual model of information structure.
- An ADT specifies the components, their structuring relationships and a list of operations that

are allowed to perform.

- It is just a specification, no design or implementation information is included.
- Specification involves the “what” the operations, not the “how”.
- ADT’s are generalizations of primitive data types.

A data structure is the **design representation** of an ADT.

- The same ADT may be represented by several data structures.
- There are many data structures corresponding to the ADT “set”.

Operations Perform on Data Structure

Data are processed by means of certain operations, which appear in the data structure. The particular data structure is chosen largely depends on the frequency of the operation that needs to be performed on the data structure. Different kinds of operations are to be performed on data structures.

Table 1.2: Operations on Data Structures

Operation	Description
Creation	Allocation of memory for the data structure, the creation of data structure may take place either during compile-time or during run-time.
Insertion	Insert a data item in the data structure.
Deletion	Delete a data item from the data structure.
Traversing	Accessing and processing each data item of the data structure exactly once.
Searching	Find the location of the key value within the data structure.
Sorting	Arranging all the data items in a data structure either in ascending or in descending order or in lexicographical order (for Strings).
Merging	Combining the data items of two different sorted lists into a single sorted list.

Overview of Different Data Structures

Different data structures can be defined as follows:

Array

An array is a collection of the same type of data items, which are stored in consecutive memory locations under a common name. In arrays, there is always a fixed relationship between the addresses of two consecutive elements as all the items of an array must be stored contiguously.

Stack

A stack is a collection of elements into which new elements may be inserted and from which elements may be deleted only at one end called the top of the stack. Since all the insertion and deletion in a stack is done from the top of the stack, the last added element will be first to be removed from the stack. That is the reason why stack is also called Last-In-First-Out (LIFO) data structure.

Queue

A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue. The element, which inserts in the queue first, will delete the queue first. In this order, a queue is called First-In-First-Out (FIFO) system.

Linked List

A linked list is an ordered collection of finite homogeneous data elements called node where the linear order is maintained by means of links or pointers. In linked list, data items may be scattered arbitrarily all over the memory. In a linked list, there is no relationship between the addresses of elements; each element of a linked list must store explicitly the address of the element, next to it.

Tree

The tree is a non-linear data structure. A Tree may be defined as a non-empty finite set of nodes, such that

- i) There is a specially designated node called the root,
- ii) The remaining nodes are partitioned into zero or more disjoint trees T_1, T_2, \dots, T_n are called the subtrees of the root R.

Graph

The graph is another non-linear data structure. A Graph G is defined as an ordered set $G = (V, E)$, consists of finite non-empty set of objects V, where $V(G) = \{v_1, v_2, v_3, \dots, v_n\}$ called vertices (or nodes or points) and another set E where $E(G) = \{e_1, e_2, e_3, \dots, e_m\}$ whose elements are called edges, that connects these vertices.

Summary

- Data is the basic entity or fact that is used in the calculation or manipulation process.
- Data are raw facts without context, whereas information is data with context.
- Data Structure is defined as a mathematical or logical model of the particular organization of data items in computer memory so that it can be used efficiently.
- An Abstract Data Type (ADT) describes the data objects, which constitute the data structure, and the fundamental operations supported on them.

Exercises

1. What are the differences between linear and non-linear data structures?
2. What are the operations can be performed on data structures?
3. What is an Abstract Data Type? What do you mean by a Dynamic Data Structure?
4. Choose the correct alternatives for the following:
 - i) Which of the following data structure is a linear data structure?

a) Trees	b) Graphs	c) Arrays	d) None of these
----------	-----------	-----------	------------------
 - ii) The operation of processing each element in the list is known as

a) Sorting	b) Merging	c) Inserting	d) Traversal
------------	------------	--------------	--------------
 - iii) Finding the location of the element with a given key in the list is known as

a) Traversal	b) Searching	c) Sorting	d) None of these
--------------	--------------	------------	------------------
 - iv) Representation of data structure in memory is known as

a) Recursion	b) Abstract data type	c) Storage structure	d) File structure
--------------	-----------------------	----------------------	-------------------
 - v) An ADT is defined to be a mathematical model of a user-defined type along with the collection of all _____ operations on that model

a) Cardinality	b) Assignment	c) Primitive	d) Structured
----------------	---------------	--------------	---------------

CHAPTER 2

ANALYSIS OF ALGORITHM

“Programming is the art of telling another human being what one wants the computer to do”. –
Donald Knuth

An algorithm is a well-defined computational procedure that transforms inputs into outputs achieving the desired input-output relationship. We must design a process to solve a specific problem before programming. This process must be formulated as a detailed sequence of basic steps, which is called an algorithm. An algorithm can perform the calculation, data processing and automated reasoning task in a finite time.

The word “algorithm” is a Persian word algorism derived from the name of the Persian author and great mathematician “Abu Abd Allh Jafar Mohammad bin Musba al Khowarzimi” (Born in 780 A.D. in Baghdad). He worked on algebra, geometry, and astronomy. He designs the first algorithm for adding numbers represented in the numerical system.

ALGORITHM

Definition: Algorithm is a finite sequence of instructions/steps, each of which is very elementary that must be followed to solve a problem.

All algorithms must satisfy the following criteria:

- **Input:** There are some inputs (zero or more quantities), which are externally supplied.
- **Output:** The result (at least one quantity) should be produced after completion of the job to the user.
- **Definiteness:** The instructions should be unique, concise, clear and unambiguous.
- **Finiteness:** The instructions should be relative in nature and should not be of an infinite type. Should be terminated after a finite number of steps.
- **Effectiveness:** Every instruction must be elementary, so that it can be carried out, in principle, by a person using only pencil and paper. Repetition of same steps should be avoided. It must be feasible.

Types of Algorithms

Algorithms that use a similar problem-solving approach can be grouped together. This classification scheme is neither exhaustive nor disjoint. The purpose of this classification is to highlight the various ways in which a problem can be solved. Using different techniques, one can develop many algorithms for the same problem. These techniques are not only applicable in Computer Science but also to other fields such as Operation Research, Electrical Engineering etc. We can formulate/develop a good algorithm by studying all the techniques.

KEY FEATURES

-  Algorithm
-  Types of Algorithm
-  Time Complexity
-  Space Complexity
-  Asymptotic Notations

Algorithm types we will consider

- Divide and conquer
- Dynamic programming
- Greedy Method
- Backtracking
- Branch and Bound
- Serial or parallel or distributed algorithm
- Deterministic or non-deterministic algorithm

Algorithm Development Life Cycle

In the life cycle of an algorithm for the development of an algorithm, the following phases are involved:

- i) Algorithm design / Design Phase
- ii) Writing Phase
- iii) Testing / Experiment Phase
- iv) Analysis of Algorithm / Analysing Phase

Algorithm Design

Algorithmic design patterns, which provide general techniques for designing algorithms, including divide-and-conquer, dynamic programming, greedy method, backtracking, and branch-and-bound. In the design phase of an algorithm, one of the algorithmic design techniques and the data structure is used.

Writing Phase

An algorithm generally is written in a modular approach. An algorithm may have the following structures or steps:

Input Step, Initialization Step, Assignment Step, Decision Step, Repetition Step and Output Step.

To make each and every step clear in the algorithm, comments are written whenever necessary.

An algorithm is written using the following basic methods:

- Sequential: Steps are performed sequentially.
- Selection/Decision: One of the several alternative actions is selected.
- Repetition: One or more than one step(s) are performed repeatedly.

The following example describes a formal representation of an algorithm that converts temperature from Celsius to Fahrenheit.

Algorithm to convert temperature from Celsius to Fahrenheit.

Algorithm: Convert(C, F)

[The variable C is used as Celsius value and F is used as Fahrenheit value]

1. Read C
2. Set $F = 1.8 * C + 32$
3. Print F
4. Return

The following algorithm calculates the summation of N natural numbers. First, we have to read the value of N.

Algorithm to calculate the sum of N natural numbers.

Algorithm: Sum (N)

[The variable N is represented the number of natural numbers.]

1. Set S = 0 and I = 1
2. Read N
3. S = S + I
4. I = I + 1
5. If I<=N Goto Step 3
6. Print S
7. Return

Searching refers to the operation of finding the location of any element in the array. The search is successful if the item is found; otherwise, it is unsuccessful. Linear search is a method where the search starts from the beginning of the array until the desired key is found or the end of the array is reached.

Algorithm of Linear search.

Algorithm: Linear_Search (A, N, Key)

[An array A with N elements and an item Key are given. This algorithm searches the location of the Key in the array A]

1. Set I = 1
2. Repeat Steps 3 while I <= N and A[I] ≠ Key
3. I = I + 1
[End of while]
4. IF I <= N then
 Print: Item found at location I
Else
 Print: Item not found
[End of IF]
5. Return

Testing Phase

After writing an algorithm, it is necessary to check whether the algorithm is correct or not. One must perform each step using paper and pencil by some required valid input, use the algorithm and get the required output in a finite time. Testing or Experiment is another step in the development of the algorithm. When any error is found, then go back to the algorithm design phase and redesign the algorithm.

Analysis of Algorithms

Algorithms are to be analysed to compute the objective criteria for different input size before actual implementation. Suppose P is a problem and A & B are two different algorithms to solve the problem P. There must have some performance measurement system to decide that which algorithm is better than the other.

Suppose M is an algorithm and n is the size of the input data. The time and space used by the

algorithm M are two main measures for the efficiency of M.

The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data.

There are some qualitative aspects of algorithms, e.g., simplicity, readability, modularity, modifiability, well documented, etc. However, based on these, there can only be some subjective judgment about algorithms. Moreover, the objective criteria for the qualities of an algorithm are the amount of memory used and execution time.

Therefore, the performance of the algorithm can be measured on the scales of time and space. Time means, we are looking for the fastest algorithm for the problem. Space means we are looking for an algorithm that consumes or needs minimum memory space for its execution.

When performance is measured in terms of space, it is known as space complexity and when performance is measured in terms of time, it is termed as time complexity.

There are two kinds of analysis of algorithm may be possible:

- i) Apriori Analysis
- ii) Posteriori Analysis

Theoretical / Apriori Analysis

Apriori meaning "from the earlier". It is on a theoretical basis, independent of programming languages and machine structures. Here, we do analysis (space and time) of an algorithm prior to running it on a specific system. That means we can determine time and space complexity of an algorithm of getting the algorithm rather than running it on a particular system (with different processor and compiler). The stress is laid on the frequency of execution of statements in the algorithm.

Empirical / Posteriori Analysis

Posteriori means "from the latter". The actual amount of space and time taken by the algorithms are recorded during execution. It is dependent on the programming language used and machine structure. Here, we do an analysis of the algorithm only after running it on the system. It directly depends on the system and changes from system to system.

In real-life, we cannot do posteriori analysis as software is generally made for an anonymous user, which runs it on a system different from those present in the industry. Therefore, in Apriori analysis, we use asymptotic notations to determine time and space complexity as they change from computer to computer but asymptotically they are same

Space Complexity

The space complexity is the amount of space (memory) is needed for an algorithm to solve the problem. An efficient algorithm takes space as small as possible.

Space needed by a program is the sum of the following components:

1. Instruction space: Space needed to store the executable version of the program and it is fixed.
2. Data Space: Space needed to store all constants, variable values and has further following components:
 - a) Space needed by constants and simple variables. This space is fixed.
 - b) Space needed by fixed sized structured variables, such as arrays and structure.
 - c) Dynamically allocated space from memory pool called heap. This space usually varies.
3. In-build stack space: Space needed to store the information needed to resume the suspended functions. Each time a function, the following data is saved on the In-build stack.

- d) Return address i.e. from where it has to resume after completion of the called function.
- e) Values of all local variables and the values of formal parameters in the function being involved.

In general, the total space needed

1. Fixed Space Requirement: Includes space for code, space for simple variables, constants.
2. Variable Space Requirement: Includes space needed by component variables, structured variable and dynamically allocated space and the recursive stack space.

Therefore, space complexity of a program P is

$$S(P) = C + Sp(I)$$

Where C is the fixed space requirement independent of a particular problem and Sp (I) is the variable Space requirement depends on the instance characteristics I. Space complexity can express that result in big O notation. Consider the following algorithm.

Algorithm: INT SUM (A, N)

1. IF $N \neq 0$ then return $SUM(A, N-1) + A[N-1]$
2. Return 0

The above recursive function for summing a list of numbers, memory space is required for each recursion call of the function, 2 bytes of the parameter A, 2 bytes for the parameter N and 2 bytes for the return value. The total space required per recursive call 6 bytes and N number of recursive calls are required to summing the list. Therefore, we get $S_{SUM}(I) = S_{SUM}(n) = 6n$.

Hence, we can write the space complexity of the above function in big-O notation is $O(n)$.

Often, there is a time-space trade-off in a problem, where we compromise; that is, we cannot have both low computing time and low memory consumption, so depending on the algorithm we choose whether low computing time is essential or memory consumption.

Time complexity

The time complexity of an algorithm is a number of computer time needs in execution to complete. It allows comparing the algorithms to check which one is efficient.

There are different factors affecting the execution time, e.g. programmer skills, compiler options, hardware characteristics (instruction set, clock speed), the algorithm used, input size, etc. Generally, an estimate of the growth rate of execution time with input size is sufficient for comparison.

The rules for computing running time

1. Sequence: Add the time of the individual statements. The maximum is the one that counts.
2. Alternative structures: Time for testing the condition plus the maximum time taken by any of the alternative paths.
3. Loops: Execution time of a loop is at most the execution time of the statements of the body (including the condition tests) multiplied by the number of iterations.
4. Nested loops: Analyze them inside out.
5. Subprograms: Analyze them as separate algorithms and substitute the time wherever necessary.
6. Recursive Subprograms: Generally, the running time can be expressed as a recurrence relation. The solution of the recurrence relation yields the expression for the growth rate of execution time.

Now, Time taken by a program P means,

$T(P) = \text{compile time} + \text{run time}$

While measuring the time complexity of an algorithm, we concentrate on the frequency count of all key statements (important statement).

Let us consider three algorithms.

Algorithm

```
a = a + 1
```

In this algorithm, the frequency count of key statement $a = a + 1$, is one.

Algorithm

```
For I = 1 to n
    a = a + 1
[End of loop]
```

In this algorithm, the frequency count of key statement $a = a + 1$, is n .

Algorithm

```
For I = 1 to n
    For J = 1 to n
        a = a + 1
    [End of loop]
[End loop]
```

In this algorithm, the frequency count of key statement $a = a + 1$, is n^2 .

In the first algorithm the statement “ $a = a + 1$ ” is executed only once, in the second algorithm, “ $a = a + 1$ ” is executed n times and in the third algorithm, “ $a = a + 1$ ” is executed n^2 times.

If an algorithm performs $f(n)$ basic operations when the size of input is n , then its total running time will be $t \times f(n)$, where t is the time required to perform a basic operation.

Note: Different operations consume different times to execute, however for simplicity, we are assuming that each basic operation consumes same time to execute, which is say t in the above case.

There are three types of time complexities, which can be found in the analysis of an algorithm:

1. Best case time complexity
2. Average case time complexity
3. Worst case time complexity

Best-case time complexity

The best-case time complexity of an algorithm is a measure of the minimum time that the algorithm will require. For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

Worst-case time complexity

The worst-case time complexity of an algorithm is a measure of the maximum time that the algorithm will require. A worst-case estimate is normally computed because it provides an upper bound for all inputs including the extreme slowest case also. For example, the worst case for a simple linear search on a list occurs when the desired element is found at the last position of the list or not on the list.

Average-case time complexity

The average-case time complexity of an algorithm is a measure of average time of all instances taken by an algorithm. Average case analysis does not provide the upper bound and sometimes it is difficult

to compute.

Average-case time complexity and worst-case time complexity are the most used in algorithm analysis. Best-case time complexity is rarely found but is does not have any uses.

ASYMPTOTIC NOTATIONS

There are some notations commonly used in performance analysis to characterize the complexity of an algorithm. Big Oh (O), little oh (o), omega (Ω), theta (Θ) and little omega (ω) are related notations to describe different kinds of bounds on asymptotic growth rates.

Big O notation introduced in 1894 by Paul Bachmann, popularized in Computer Science by Donald Knuth, who re-introduced the related Omega and Theta notations.

Big-O (“big oh”) Notation (Upper Bound)

The big- O notation is the formal method of expressing the upper bound of an algorithm's running time. It is a measure of the longest amount of time it could possibly take for the algorithm to complete. This notation allows us to compare a function to a set of functions that are bounded by some other function, allowing us to describe the performance of a particular algorithm.

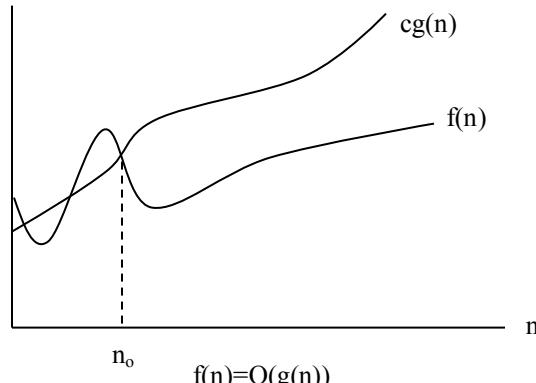


Figure 2.1: Big-O Notation

If $f(n)$ and $g(n)$ are functions defined for positive integers, then we write $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that $|f(n)| \leq c|g(n)|$, for all $n \geq n_0$.

In general, $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n, \text{ where } n \geq n_0\}$.

That is the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$. This notation gives an upper bound for a function to within a constant factor.

Example:

Suppose $f(n) = 3n + 2$

Now we can write $f(n) = 3n + 2 \leq 4n$, for all $n \geq 2$, here $g(n) = n$, $c = 4$, $n_0 = 2$

Hence, we can write $f(n) = O(g(n)) = O(n)$

Example:

Suppose $f(n) = 10n^2 + 4n + 2$

Now we can write $f(n) = 10n^2 + 4n + 2 \leq 11n^2$, for all $n \geq 5$, here $g(n) = n^2$, $c = 11$, $n_0 = 5$

Hence, we can write $f(n) = O(g(n)) = O(n^2)$

Example:

Suppose $f(n) = 2n^3 + n^2 + 2n$

Now we can write $f(n) = 2n^3 + n^2 + 2n \leq 3n^3$, for all $n \geq 2$, here $g(n) = n^3$, $c = 3$, $n_0 = 2$
Hence, we can write $f(n) = O(g(n)) = O(n^3)$

Big-Ω (Omega) Notation (Lower Bound)

Big-Ω notation is the formal method of expressing the lower bound on an algorithm's running time. It describes the best that can happen for a given data size. If $f(n)$ and $g(n)$ are functions defined for positive integers, then we write $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that $|f(n)| \geq c|g(n)|$, for all $n \geq n_0$.

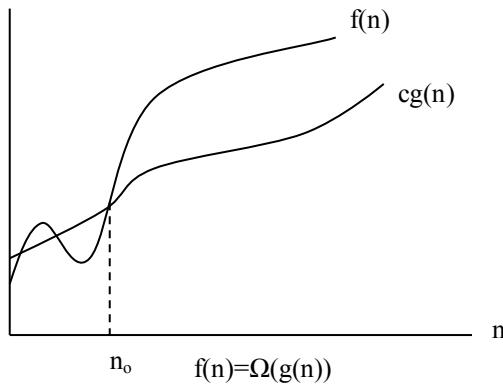


Figure 2.2: Omega Notation

In general, $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n, \text{ where } n \geq n_0\}$.

That is the right of n_0 , the value of the function $f(n)$ always lies on or above $cg(n)$. This notation gives a lower bound for a function to within a constant factor.

Example:

Suppose $f(n) = 3n + 2$

Now we can write $f(n) = 3n + 2 \geq 3n$, for all $n \geq 1$, here $g(n) = n$, $c = 3$, $n_0 = 1$

Hence, we can write $f(n) = \Omega(g(n)) = \Omega(n)$

Example:

Suppose $f(n) = 10n^2 + 4n + 2$

Now we can write $f(n) = 10n^2 + 4n + 2 \geq 10n^2$, for all $n \geq 1$, here $g(n) = n^2$, $c = 10$, $n_0 = 5$

Hence, we can write $f(n) = \Omega(g(n)) = \Omega(n^2)$

Example:

Suppose $f(n) = 2n^3 + n^2 + 2n$

Now we can write $f(n) = 2n^3 + n^2 + 2n \geq 2n^3$, for all $n \geq 1$, here $c = 2$, $n_0 = 1$

Hence, we can write $f(n) = \Omega(g(n)) = \Omega(n^3)$

Θ (Theta) -Notation (Tight Bound)

If $f(n)$ & $g(n)$ are functions defined for positive integers, then we say $f(n) = \Theta(g(n))$ if there exists positive constants n_0, c_1 and c_2 such that

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|, \text{ for all } n \geq n_0$$

In general, $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, \text{ where } n \geq n_0\}$.

That is the right of n_0 the value of the function $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. This notation bounds a function to within constant factors. This is saying that the function, $f(n)$ is bounded from both the top and bottom by the same function, $g(n)$.

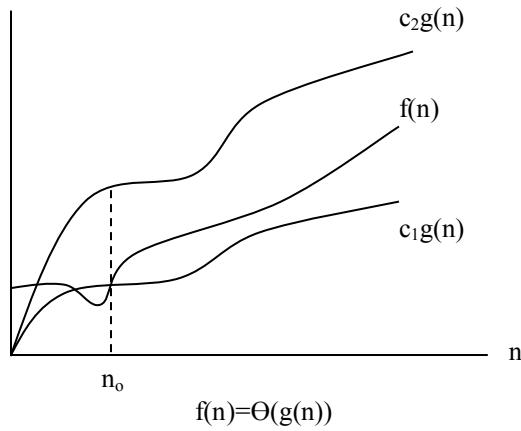


Figure 2.3: Theta Notation

Example:

Suppose $f(n) = 3n + 2$

Now we can write $3n \leq 3n + 2 \leq 4n$, for all $n \geq 2$, here $g(n) = n$, $c_1 = 3$, $c_2 = 4$, $n_0 = 1$

Hence, we can write $f(n) = \Theta(g(n)) = \Theta(n)$

Example:

Suppose $f(n) = 10n^2 + 4n + 2$

Now we can write $10n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$, for all $n \geq 5$

Here $g(n) = n^2$, $c_1 = 10$, $c_2 = 11$, $n_0 = 5$

Hence, we can write $f(n) = \Theta(g(n)) = \Theta(n^2)$

Example:

Suppose $f(n) = 2n^3 + n^2 + 2n$

Now we can write $2n^3 \leq 2n^3 + n^2 + 2n \leq 3n^3$ for all $n \geq 2$, here, $c_1 = 2$, $c_2 = 3$, $n_0 = 2$

Hence, we can write $f(n) = \Theta(g(n)) = \Theta(n^3)$

o (“little oh”) - Notation

For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is little o of $g(n)$ if and only if $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as " $f(n) = o(g(n))$ ".

2.10 | Data Structures and Algorithms with C

This represents a loose bounding version of Big O; $g(n)$ bounds from the top, but it does not bound the bottom. Little-o notation represents a loose bounding version of Big O, $o(g(n))$ bounds from the top, but it does not bound the bottom.

If $f(n) & g(n)$ are functions defined for positive integers, then we say $f(n) = o(g(n))$ iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

Example:

Suppose $f(n) = 2n^3 + 3n^2 + n + 10$

Now we can write,

$$\lim_{n \rightarrow \infty} \frac{(2n^3 + 3n^2 + n + 10)}{n^4} = 0$$

i.e. here $g(n) = n^4$

Hence, we can write $f(n) = o(g(n)) = o(n^4)$

Another definition of little oh (o) is that,

If $f(n) & g(n)$ are functions defined for positive integers, then we say $f(n) = o(g(n))$ iff
 $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$

ω (little omega)-Notation

Little- ω notation represents a loose bounding version of Big- Ω ; $g(n)$ is a loose lower boundary of the function $f(n)$; it bounds from the bottom, but not from the top. For non-negative functions, $f(n)$ and $g(n)$, $f(n)$ is little omega of $g(n)$ if and only if $f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as " $f(n) = \omega(g(n))$ ".

We can write $f(n) = \omega(g(n))$ iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Example:

Suppose $f(n) = 4n^3 + 2n + 3$

Now we can write,

$$\lim_{n \rightarrow \infty} \frac{4n^3 + 2n + 3}{n^2} = \infty$$

Here $g(n) = n^2$

Hence we can write $f(n) = \omega(g(n)) = \omega(n^2)$

Growth Functions of Algorithm

The big-O notation is used to classify algorithms by how they respond in their processing time or working space requirements to changes in the input size. Order of growth in algorithm means how the time or space for computation increases with input size, i.e. how the growth function grows with the input size. Now, based on growth functions, the algorithm can be categorized as follows:

1. Constant time algorithms [O(1)]:

This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

Example: Array access, Push and pop operations for a stack, enqueue and dequeue operations for a queue, other atomic operations.

2. Logarithmic time algorithms [O(log n)]:

Logarithm time algorithms mean that $T(n)$ is upper bounded by logarithmic of the input size, i.e. $T(n) = \log n$.

Example: Binary search in a sorted list of n elements, Insert and Find operations for a binary search tree with n nodes, Insert and Remove operations for a heap with n nodes, etc.

3. Linear time algorithms [O(n)]:

Linear time algorithm means that for large enough input sizes the running time increases linearly with the size of the input.

Example: Array traversal

4. Linearithmic time algorithm [O(nlog n)]:

An algorithm is said to run in linearithmic time when $T(n) = O(n \log n)$. Thus, a linearithmic term grows faster than a linear term, but slower than any polynomial in n with degree greater than 1.

Example: Merge sort, Heap sort

5. Polynomial time algorithms [O(n^k), for $k > 1$]:

- i) Quadratic time ($O(n^2)$) algorithm: The number of operations is proportional to the size of the task squared.

Example: Slow sorting algorithms, for example, selection sort of n elements.

- ii) Cubic time ($O(n^3)$) algorithm: The number of operations is proportional to the size of the task cubed.

Example: Matrix multiplication

6. Exponential time algorithms [O(k^n), for $k > 1$]:

Exponential time algorithm means that $T(n)$ is upper bounded by exponential expression of input size, i.e. $T(n) = k^n$

Example: Towers of Hanoi, implementation of recursive Fibonacci, generating all permutations of n symbols.

7. Factorial time algorithms [O($n!$)]:

Polynomial time algorithm means that $T(n)$ is upper bounded by the factorial of the input size, i.e. $T(n) = n!$.

Example: Non-attacking n queen problem, traveling salesman problem by brute-force search

The best time in the above list is obviously constant time, and the worst is exponential time, which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n . Polynomial growth (linear, quadratic, cubic, etc.) is considered manageable as compared to

exponential growth.

Table 2.1: Values for different growth functions

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648

Order of asymptotic behavior of the functions from the above list, we can compare among time complexities:

$$O(1) < O(\log n) < O(n) < O(n^k) < O(k^n) < O(n!)$$

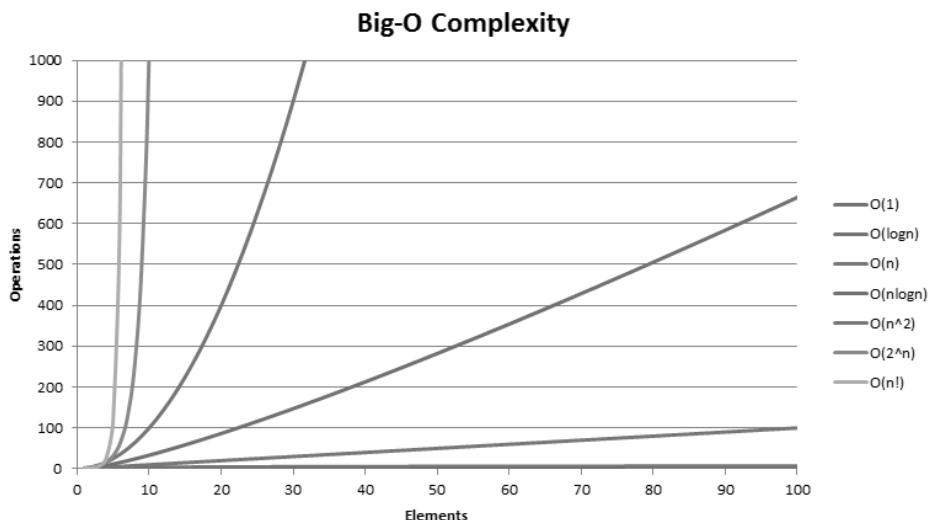


Figure 2.4: Big-O Complexity

Some important points:

- Factorial time algorithms are worst
- Factorial and exponential time algorithms are impractical
- Even polynomial time algorithms of greater than $O(n^3)$ are impractical
- It may be possible to reduce the time complexity significantly by changing the algorithm design strategy
- For logarithmic time algorithms, if the input is to be read, the total solution may become linear time.

Many algorithms such as advanced sorting algorithms - quicksort, merge sort, heap sort are of growth function $O(n \log n)$, known as linearithmic time; comparisons between $O(n \log n)$ and other growth functions

$$O(n) < O(n \log n) < O(n^2)$$

Manipulation of Asymptotic Notations

A number of rules to manipulate asymptotic notations are followed:

Rule 1: $O(c f(n)) = O(f(n))$

Rule 2: $O(O(f(n))) = O(f(n))$

Rule 3: $O(f(n)g(n)) = f(n)O(g(n))$

Rule 4: $O(f(n)O(g(n))) = O(f(n)g(n))$

Rule 5: $O(f(n) + g(n)) = O(\max(f(n), g(n)))$

Rule 6: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. [Transitivity Rule]

Rule 7: $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$. [Symmetry Rule]

Suppose, in an algorithm consists of two independent parts- A_1 and A_2 . These parts may be considered as a sub-algorithm or a single instruction. Let us assume that A_1 and A_2 take time t_1 and t_2 respectively for computation.

Now, whenever these two parts are in a sequential structure that means first A_1 is computed, and then A_2 is computed, denoted by $A_1 : A_2$, the total computation time is $t_1 + t_2$. According to maximum rule, the computation time denoted by $\max(t_1, t_2)$.

Example:

Suppose $t_1 = O(n^2)$ and $t_2 = O(n)$, calculate the total computation time for $A_1 : A_2$

$$\begin{aligned} \text{Answer: For } A_1 : A_2, \text{ computation time} &= t_1 + t_2 \\ &= \max(t_1, t_2) \\ &= \max(O(n^2), O(n)) \\ &= O(n^2) \end{aligned}$$

Again reconsider A_1 and A_2 are two independent parts of an algorithm, with t_1 and t_2 as computation times respectively. Now, whenever these two parts are in a decision structure, that means A_1 is computed only when the given condition is true, otherwise, A_2 is computed when the condition is false, denoted by A_1 / A_2 , the total computation time is t_1 / t_2 . According to maximum rule, the computation time denoted by $\max(t_1, t_2)$.

Example:

Suppose $t_1 = O(n^2)$ and $t_2 = O(n)$, calculate the total computation time for A_1 / A_2

$$\begin{aligned} \text{Answer: For } A_1 / A_2, \text{ computation time} &= t_1 / t_2 \\ &= \max(t_1, t_2) \\ &= \max(O(n^2), O(n)) \\ &= O(n^2) \end{aligned}$$

Example:

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

Here, the statement `sum++` executes $3n$ times.

Answer: The complexity is $O(n)$

Example:

```
sum = 0;
for (i=0; i<n*n; i++)
    sum++;
```

Here, the statement `sum++` executes n^2 times.

Answer: The complexity is $O(n^2)$

Example:

```
sum = 0;
for (i = 0; i < n; i++) {
    if (is_even(i)) {
        for (j = 0; j < n; j++)
            sum++;
    }
    else
        sum = sum + n;
}
```

Here, inside the loop: if “true” clause executed for half the values of n , that is $O(n)$ and if “false” clause executed for the other half of n , $O(1)$, so innermost loop is $O(n)$. The outer loop is $O(n)$.

Answer: The complexity is $O(n^2)$

Example:

```
for (i=1; i<=n; i++) {
    for (j=1; j<=2*i; j++) {
        k=j;
        while (k>=0)
            k=k-1;
    }
}
```

At first, consider the innermost while loop: it is executed $k = j, j - 1, j - 2 \dots 0$. Time consumed inside the while loop is constant. Let $I()$ be the time consumed in the while loop. Thus,

$$I(j) = \sum_{k=0}^j 1 + 1$$

Consider the middle for loop, its running time is determined by i . Let $M(i)$ be the time consumed in the for loop:

$$\begin{aligned} M(i) &= \sum_{j=1}^{2i} I(j) \\ &= \sum_{j=1}^{2i} (j+1) \\ &= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \end{aligned}$$

$$\begin{aligned}
 &= \frac{2i(2i+1)}{2} + 2i \\
 &= 2i^2 + 3i
 \end{aligned}$$

Finally, the out-most for loop, it is executed $i = 1, 2, 3, \dots, n$. Let $T(n)$ be running time of the entire algorithm:

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n M(i) \\
 &= \sum_{i=1}^n (2i^2 + 3i) \\
 &= \sum_{i=1}^n 2i^2 + \sum_{i=1}^n 3i \\
 &= 2 \frac{2n^3 + 3n^2 + n}{6} + 3 \frac{n(n+1)}{2} \\
 &= \frac{4n^3 + 15n^2 + 11n}{6} \\
 &= O(n^3)
 \end{aligned}$$

Answer: The complexity is $O(n^3)$

Example:

$T(n) = 4n^2 + 3n\log n$, express $T(n)$ in Big-O notation

$T_a = 4n^2 = O(n^2)$ and

$T_b = 3n\log n = O(n\log n)$

$$\begin{aligned}
 \text{Now, } T(n) &= O(\max(T_a, T_b)) \\
 &= O(\max(O(n^2), O(n\log n))) \\
 &= O(n^2) [\text{because } n > \log n, \text{ hence } n^2 > n\log n]
 \end{aligned}$$

Summary

- The algorithm is a finite sequence of instructions/steps, each of which is very elementary that must be followed to solve a problem.
- The space complexity of a program is the amount of memory it needed to run to completion.
- The time complexity of a program is the amount of computer time needs to run to completion.
- The asymptotic notations commonly used in performance analysis to characterize the complexity of an algorithm.
- The big-O notation is the formal method of expressing the upper bound of an algorithm's running time.

Exercises

- Show that the function $f(n)$ defined by
 $f(1) = 1$
 $f(n) = f(n-1) + \frac{1}{n}$ for $n > 1$
has a time complexity $O(\log n)$.
- Define Big-O notation, Θ notation, Ω notations.

2.16 | Data Structures and Algorithms with C

3. The designer of an algorithm needs to balance between space complexity and time complexity.”- Comment on the validity of the statement in the context of recursion algorithms.
4. Choose the correct alternatives for the following:
- i) Which of the following is the best time for an algorithm?
a) $O(n)$ b) $O(\log n)$ c) $O(2n)$ d) $O(n \log n)$
 - ii) Which of the following algorithm should execute the slowest for large value of N
a) $O(n)$ b) $O(n^2)$ c) $O(\log n)$ d) None of these
 - iii) Let us consider a function $f(n) = 1000n\log n + 500 n^4 + 0.52^n$. We can say that $f(n)$ is
a) $O(n^4)$ b) $O(n \log n)$ c) $O(2^n)$ d) None of the above
 - iv) An algorithm is made up of two independent time complexities $f(n)$ and $g(n)$. Then the complexities of the algorithm is in the order of
a) $f(n) \times g(n)$ b) $\text{Max}(f(n), g(n))$ c) $\text{Min}(f(n), g(n))$ d) $f(n) + g(n)$
 - v) Two main measures of the efficiency of an algorithm are
a) complexity and capacity b) Processor and memory
c) Time and space d) Data and space
 - vi) Which of the following shows the correct relationship between some of the more common computing time for the algorithm?
a) $O(\log n) < O(n) < O(n*\log n) < O(2^n) < O(n^2)$
b) $O(n) < O(\log n) < O(n*\log n) < O(2^n) < O(n)$
c) $O(n) < O(\log n) < O(n*\log n) < O(n^2) < O(2^n)$
d) $O(\log n) < O(n) < O(n*\log n) < O(n^2) < O(2^n)$

CHAPTER 3

ARRAY AND STRING

"Computers make it easier to do a lot of things, but most of the things they make it easier to do don't need to be done." -Andy Rooney

Many applications require the processing of multiple data items, that have common characteristics. For example, let say we have to find the average marks obtained in a subject by students of a class. In this case, we have a couple of choices. Either we can store all the values in different variables or we can create an array that will hold all these values in contiguous memory location. If they are stored in different variables and the number of such variables is quite large then it is quite unmanageable as these variables are scattered throughout the memory. In such situations, it is often convenient to place the data items into an array, where they will all share the same name.

KEY FEATURES

-  One dimensional array
-  Processing an array
-  Two-dimensional array
-  Representation of 2D array
-  Sparse Matrix
-  String

ARRAY

Definition: An array is a collection of the same type of data items, which are stored in consecutive memory locations under a common name.

An array variable must be followed by square bracket enclosing size specification of the array. The size signifies the number of elements of the array. That number should be a positive integer number (greater than 0).

One-dimensional Array

A one-dimensional array is one in which only one subscript is needed to specify a particular element of the array.

Declaration of One-Dimensional Array

The one-dimensional array can be declared as follows:

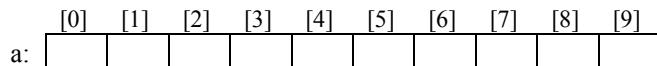
```
data_type array_name [size];
```

where array_name is the name of an array and size is the number of elements of data type data_type and size must be an integer constant specified within a pair of square brackets.

Example:

```
int a[10];
```

The above declaration states an array, named "a", consisting of ten elements, each of type int. Simply speaking, an array is a variable that can hold more than one value. (Arrays in programming are similar to vectors or matrices in mathematics.) We represent the array in above with a picture like this:

**Figure 3.1:** Array of size 10

Sometimes symbolic constant can be used to define array size rather than an integer number.

Example:

```
#define N 10
int a[N];
```

In C, arrays are zero-based, that means the index of an array starts from zero. The ten elements of a 10-element array are numbered from 0 to 9. The subscript, which specifies a single element of an array, is simply an integer expression in square brackets. The first element of the array is $a[0]$, the second element is $a[1]$, etc.

Example:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Notice that the subscripted array references (i.e. expressions such as $a[0]$ and $a[1]$) can appear on either side of the assignment operator.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression; it is common to loop over all elements of an array:

Example:

```
int i;
for(i = 0; i < 10; i++)
    a[i] = 0;
```

This loop sets 0 to all ten elements of the array $a[0]$ to $a[9]$.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them automatically. In particular, you can neither set all elements of an array at once, nor assign one array to another; both of the assignments are illegal.

Example:

```
int a[10];
a = 0; /* illegal */
and
int b[10];
b = a; /* illegal */
```

To set all of the elements of an array to some value, you must do so one by one, as shown in the loop example above. To copy the contents of one array to another, you must again do so one by one.

Example:

```
int b[10];
for(i = 0; i < 10; i++)
    b[i] = a[i];
```

Remember that for an array declared

```
int a[10];
there is no element a [10]; the end element is a [9].
```

Arrays are not limited to type int; you can have arrays of char or float or double or any other type.

Example:

```
char name[30];
float x[10];
double d[100];
```

There are some important points about arrays. These are as follows:

- Arrays are always stored in consecutive memory locations.
- An array can store multiple values that can be referenced by a single name.
- An array name is actually an address of the first element of the array.
- An array can be either an integer, character or floating-point data type can be initialized only during declaration.
- In C, array index starts from zero.
- There is no **bound checking** concept for arrays in C.

Size of One-Dimensional Array

The one-dimensional array can be declared as:

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

The total size of the array in bytes can be calculated by using the following expression:

```
sizeof(data_type)*size
```

where data_type is the data type of the array and size specifies the total number of elements of the array.

Another way to calculate the total size of the array in bytes is simpler; that uses the following expression

```
sizeof(array_name)
```

where array_name is the name of the array.

Initialization of One-Dimensional Array

Although it is not possible to assign values to all elements of an array at once using an assignment expression, it is possible to initialize some or all elements of an array when the array is defined.

The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array. The individual array elements are as follows:

```
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 5
a[6] = 6
a[7] = 7
```

```
a[8] = 8
a[9] = 9
```

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 may be omitted, and the compiler will infer the dimension from the number of initializers. For example,

```
int b[] = {10, 11, 12, 13, 14};
```

would declare, define, and initialize an array b of 5 elements (i.e. just as if you'd typed int b[5]). Only the dimension is omitted; the brackets [] remain to indicate that b is an array.

Reading array elements

As an array may contain more than one element, so to read all the elements, a single call to the scanf function will not serve our purpose.

Example:

If we have an array declared as

```
int a[10];
```

to read all the elements we can use the following statements:

```
for(i = 0; i < 10; i++)
    scanf ("%d", &a[i]);
```

When the value of i is 0 then scanf() will read the value of a[0] element, if i is 1 it will read the value of a[1] and so on.

Writing array elements

Similarly, to print all the array elements of an array we can use the printf statement within a loop. Considering the above array declaration int a [10]. We can use the following statements:

Example:

```
for(i = 0; i < 10; i++)
    printf ("%d\t", a[i]);
```

Program 3.1: Find out the minimum and maximum of n given numbers.

```
#include<stdio.h>
main()
{
    int a[10], max, min, i, n;
    printf("\nEnter the number of elements:");
    scanf ("%d",&n);
    printf("\nEnter %d elements:",n);
    for( i = 0 ; i < n; i++ )
        scanf("%d", &a[i]);
    /* Initialization */
```

```

max = a[0];
min = a[0];
/* Finding Maximum and Minimum element */
for(i = 0; i < n; i++) {
    if(a[i] > max) max = a[i];
    if(a[i] < min) min = a[i];
}
printf ("\n The maximum element is: %d", max);
printf ("\n The minimum element is: %d", min);
}

```

Output:

```

Enter the number of elements: 5
Enter 5 elements: 3 2 9 5 4
The maximum element is: 9
The minimum element is: 2

```

Address Calculation in One Dimensional Array

The address of a particular element in a one-dimensional array is given by the relation:

$$\text{Address of element } a[i] = B + W \times i$$

Where B is the base address of the array and W is the size of each element in the array.

Example:

Consider the following array declaration,

```
int a[10];
```

Let the size of the elements stored in the above array be 4 bytes each. If the base address of the array is 1000, then the address of a [5] will be $1000 + (4 \times 5) = 1020$.

Multi-Dimensional Arrays

Multi-dimensional array is nothing but Array of Arrays. In fact, a two-dimensional array is an array of one-dimensional array. A three-dimensional array is an array of two-dimensional arrays. Similarly, any d dimensional array is an array of (d-1) dimensional arrays.

Generally, when we talk about multi-dimensional arrays, we generally talk of two-dimensional arrays. Since more than two-dimensional is rarely needed.

There is a need to store and manipulate two-dimensional data structure such as matrices and tables.

A two-dimensional array is one in which two subscript specifications are needed to specify a particular element of the array.

Here the array has two subscripts. One subscript denotes the row and the other the column.

Declaration of Two-Dimensional Array

The declaration of two-dimensional arrays is as follows:

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

Example:

```
int m[2][3];
```

Here m is declared as a matrix having 2 rows (numbered from 0 to 1) and 3 columns (numbered 0 through 2). The first element of the matrix is m [0][0] and the last row last column is m[1][2].

The two-dimensional array m[2][3] is shown in figure 3.2. The first element of this array denoted by m [0][0] and the second element is denoted by m [0][1] and so on.

	[0]	[1]	[2]
[0]	m[0][0]	m[0][1]	m[0][2]
[1]	m[1][0]	m[1][1]	m[1][2]

Figure 3.2: Two-dimensional array m [2][3]

Address Calculation in two-dimensional Array

A two-dimensional array can be implemented in a programming language in two ways:

- Row-major Implementation
- Column-Major Implementation

Note: In C language a two Dimensional Array is implemented only by row major order.

Row-major Implementation

In a row-major implementation, memory allocation is done row by row; i.e. all the elements of the first row is stored then the second row is stored and so on.

	[0]	[1]	[2]
[0]	A[0][0]	A[0][1]	A[0][2]
[1]	A[1][0]	A[1][1]	A[1][2]
[2]	A[2][0]	A[2][1]	A[2][2]

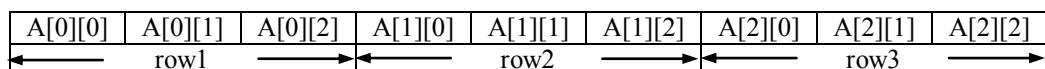


Figure 3.3: Two- dimensional array and row-major representation

The two-dimensional array can be implanted in row major order like the above figure 3.3.

Address of element $A[i][j] = B + W(n(i - L_r) + (j - L_c))$

Where B is the base address of the array

W is the size of each array element

L_r is the lower bound of row

L_c is the lower bound of column

U_r is the upper bound of row.

U_c is the upper bound of column

n is the number of columns , $n = (U_c - L_c) + 1$

Column-major Implementation

In a column-major implementation, memory allocation is done column by column; i.e. all the elements of the first column are stored then the second column is stored and so on.

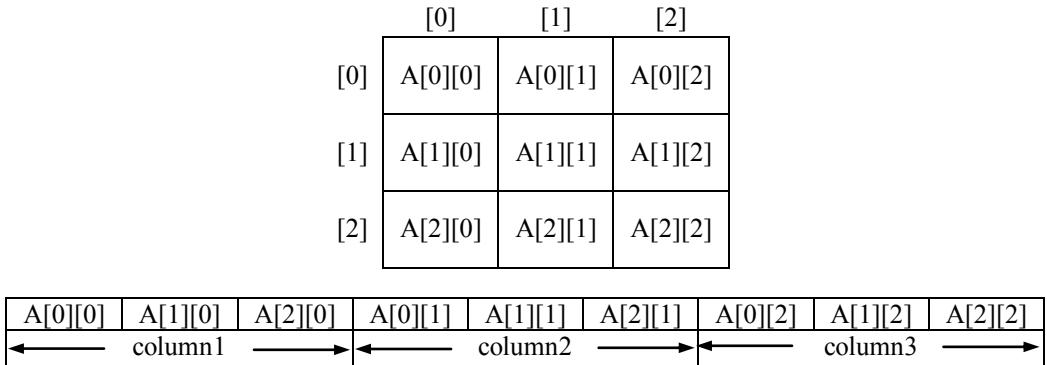


Figure 3.4: Two-dimensional array and column-major representation

The two-dimensional array can be implanted in column-major order like the above figure 3.4.

Address of element $A[i][j] = B + W((i - L_r) + m(j - L_c))$

Where B is the base address of the array

W is the size of each array element

L_r is the lower bound of row

L_c is the lower bound of column

U_r is the upper bound of row.

U_c is the upper bound of column

m is the number of rows, $m = (U_r - L_r) + 1$

Example:

Let the size of the elements stored in an 8×3 matrix be 4 bytes each. If the base address of the matrix is 3500, then find the address of $A[5][2]$ for both row major and column major cases.

Answer: As the lower bounds of row and column are not given, they are assumed as 0. [As in C lower bounds are always 0]

So, $L_r = 0$

$L_c = 0$

Number of rows $m = 8$

Number of columns $n = 3$

Size of each element $W = 4$

Base address $B = 3500$

So, in row major order address of element $A[5][2]$

$$\begin{aligned}
 &= 3500 + 4 \times ((5 - 0) \times 3 + (2 - 0)) \\
 &= 3568
 \end{aligned}$$

And in column major order address of element $A[5][2]$

$$\begin{aligned}
 &= 3500 + 4 \times ((5 - 0) + (2 - 0) \times 8) \\
 &= 3584
 \end{aligned}$$

Initialization of two-dimensional Arrays

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Example:

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

Initializes all the elements of first row to 1 and the second row to 2. The individual array elements are as follow:

```
matrix [0][0] = 1    matrix [0][1] = 1    matrix[0][2] = 1  
matrix [1][0] = 2    matrix [1][1] = 2    matrix[1][2] = 2
```

The initialization is done row by row. The above statement can be equivalently written as

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

When a two dimensional array is completely initialized with all the elements then there is no explicit need of specifying the first dimensional. That is the statement

Example:

```
int matrix[][3] = {{1, 1, 1}, {2, 2, 2}}; /* valid */
```

The compiler will automatically supply the first dimension. However, we have to specify the second dimension.

So the statement

```
int matrix[2][] = {{1, 1, 1}, {2, 2, 2}}; /* invalid */
```

is invalid , also the statement

```
int matrix[][]={{1, 1, 1}, {2, 2, 2}}; /* invalid */
```

is invalid as the second dimension is missing.

If some of the elements are missing, they are automatically initialized to zero by the compiler. So consider the following case

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

Here the element matrix[0][2] and matrix[1][2] are initialized to zero.

Now consider the following case

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

The first four elements are initialized to 1, 1, 2 and 2 respectively and the remaining elements are initialized to 0 by default by the compiler. Therefore, the matrix elements are

```
matrix[0][0] = 1  matrix[0][1] = 1  matrix[0][2] = 2  
matrix[1][0] = 2  matrix[1][1] = 0  matrix[1][2] = 0
```

Reading and Writing a two-dimensional array

We have already discussed how to read and write a one-dimensional array. Since there is only one subscript specification, it requires only one running variable; hence, we can use a single loop.

However, in the case of the two-dimensional array, there are two subscripts. So it requires two running variables, one for a row and other for a column. Therefore, we have to use nested loops. Now consider the following example.

Example:

```
int m[3][3];
/* Reading */
for(i = 0; i < 3; i++)
    for(j = 0; j < 3; j++)
        scanf("%d", &m[i][j]);
```

Therefore, the above code will read all the elements of the two-dimensional array m row by row.

Similarly, the following code can be used to print all the elements of the two-dimensional array m in matrix form.

Example:

```
/* Writing */
for(i = 0; i < 3; i++) {
    for(j = 0; j < 3; j++)
        printf("%d ", m[i][j]);
    printf ("\n ");
```

By surrounding the elements of each row by braces. C allows arrays of three or more dimensions. The compiler determines the maximum number of dimensions. The general form of a multidimensional array declaration is:

```
data_type array_name[s1][s2][s3]...[si]...[sn];
```

where s_i is the size of the i^{th} dimensional. Some examples are:

Example:

```
int survey[3][5][12];
float table [5][4][5][3];
```

Here `survey` is a three-dimensional array and `table` is a four-dimensional array.

Operations Perform on Arrays

Arrays are most important and commonly used data structures. They are also used to implement many other data structures. Almost all types of operations are performed on Arrays.

Table 3.1: Operations on Arrays

Operation	Description
Traversing	Accessing and processing each data item of the array exactly once.
Insertion	Insert a data item into the array.
Deletion	Delete a data item from the array.
Merging	Combining the data items of two different sorted arrays into a single sorted array.
Searching	Find the location of the key value within the array.
Sorting	Arranging all the array elements either in ascending or in descending order.

Traversing a Linear Array

The method of accessing and processing each element of the array exactly once is known as traversing. In an array, traversal begins from the first element in the array and ends at the last element of the array.

Algorithm to traverse a linear array**Algorithm: Traverse (A, N)**

[A is the name of the array and N is the number of elements of the array]

1. Set I=0
2. Repeat steps 3 and 4 while $I < N$
3. Visit $A[I]$
4. Set $I=I+1$
[End of loop]
5. Return

Inserting an Element into a Linear Array

On array, insertion operation can be performed in different ways. Insert an element in an array can be done in the following ways:

- To a specific position into the unsorted array
- By value of the inserted element in the sorted array

Inserting an Element to a Specific Position

Now, to insert a key element to a specific position in a linear array, at first, it needed to move each element from the back up one position until getting the specified position. Then insert a key element into a given position. Finally, increment array size by one.

Example:

Insert 50 to the 2nd position of the array with 5 elements.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	84	21	14	77					

a) Initial Array

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	84	21	14	77	77				

b) Move from 4th position to 5th position

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	84	21	14	14	77				

c) Move from 3rd position to 4th position

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	84	21	21	14	77				

d) Move from 2nd position to 3rd position

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	84	50	21	14	77				

e) Insert 50 to the 2nd position**Figure 3.5: (a-e) Insertion in the array to a specific position****Algorithm to insert an element to a specific position into a linear array****Algorithm: Insert (A, N, P, KEY)**

[A is an array of N elements; P is the position of the inserted item, KEY is the inserted item]

1. Set $I=N-1$

```

2. Repeat steps 3 and 4 while I>=P
3.     Set A[I+1]=A[I]
4.     Set I=I-1
    [End of loop]
5. Set A[P]=KEY
6. Set N=N+1
7. Return

```

Inserting an Element in a sorted Array

While inserting an element into a sorted linear array, at first, it needs to find the correct position to insert the key element. When the position cannot be found, then insert at the end. Otherwise, move each element from the back up one position until getting to position into the insert. Then insert a key element into required position. Finally, the array size is incremented by one.

Example:

Insert 50 into a sorted array with 5 elements.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
14	21	40	77	84					

a) Initial Array

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
14	21	40	77	84	84				

b) Move from 4th position to 5th position

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
14	21	40	77	77	84				

c) Move from 3rd position to 4th position

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
14	21	40	50	77	84				

d) Insert in 3rd position since 50 is greater than 2nd element

Figure 3.6: (a-d) Insertion in the sorted Array

Algorithm to insert an element in the sorted Array

Algorithm: Insert (A, N, KEY)

[A is the sorted array of N elements and KEY is the inserted item]

```

1. Set I = N - 1
2. Repeat steps 3 and 4 while A[I]>KEY
3.     Set A[I+1]=A[I]
4.     Set I=I-1
    [End of loop]
5. Set A[I+1]=KEY
6. Set N=N+1
7. Return

```

Deleting an element from a Linear Array

In the array, delete operation can be performed in different ways. Delete an element in an array can be done in the following ways:

- From a specific position of the deleted element
- By value of the deleted element

Deleting an element from a specific position

To delete a key element from a specific position of a linear array, at first, it needed to move each element from one after the given position forward one position until end of the array. Finally, the array size is decrement by one.

Example:

Delete an element from 3rd position (i.e. 21) of the array with 6 elements.

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]	a) Initial Array
40 84 50 21 14 77	
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]	b) Move from 4 th position to 3 rd position
40 84 50 14 14 77	c) Move from 5 th position to 4 th position

Figure 3.7: (a-c) Delete element from a specific position

Algorithm to delete an element from a specific position

Algorithm: Delete (A, N, P, KEY)

[A is an of N number of elements, P is the position of the deleted item; KEY is the deleted item returns by reference]

1. Set I = P
2. KEY = A[P]
3. Repeat steps 4 and 5 while I < N - 1
4. Set A[I] = A[I+1]
5. Set I = I + 1
 [End of loop]
6. Set N = N - 1
7. Return

Deleting an element by value

When deleting operation performed on the array by the value of the element, then it needed to search through the array to find its position.

Now there are two ways to find out the position of the deleted element. In the case of the sorted array, we can use binary search to find the position; otherwise if the array is unsorted then perform a linear search to find the position.

The following algorithm, deleting a key element from a specific position of an unsorted array. At first, it needed to search the array to find its position using linear searching. After getting the position, move each element from one after that position forward one position until end of the array. Finally, the array size is decremented by one.

The following algorithm, deleting an element of the linear array.

Algorithm to delete an element using by value

Algorithm: Delete (A, N, KEY)

[A is the name of the array, N is the number of elements of the array, P is the position of the deleted item, KEY is the deleted item]

1. Set I = 0
2. Repeat while A[I] ≠ KEY AND I < N
3. Set I = I + 1
 [End of loop]
4. Repeat steps 5 and 6 while I < N-1
5. Set A[I] = A[I+1]
6. Set I = I + 1
 [End of loop]
7. Set N = N - 1
8. Return

Merging

Merging is an operation of combining more than one sorted array together so that resulting array remains sorted. The merge algorithm used in the merge-sort algorithm, a comparison-based sorting algorithm.

Merging of two sorted arrays, one can be done in linear time and space. Let, two sorted arrays, A of m elements and B of n elements merge to form a sorted array C of total m + n elements.

The first element of array A is compared with the first element of array B. If the first element of array A is smaller than the first element of array B, the element from array A is moved to the new array C. The subscript of the array A is now increased since the first element is now set and we move on.

Otherwise, the element from array B should be smaller; it is moved to the new array C. The subscript of array B is increased. This process of comparing the elements in the two arrays continues until either array A or array B is empty. When one array is empty, any elements remaining in the other (non-empty) array are appended to the end of array C and the merge is complete.

Example:

Merge two sorted arrays A of 6 elements and B of 5 elements to form a sorted array C of 11 elements

	[0]	[1]	[2]	[3]	[4]	[5]		[0]	[1]	[2]	[3]	[4]
A:	3	6	9	12	14	17		2	5	8	11	15

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2										

(a) Since 3 > 2, hence copy 2 and increment indices of B and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3									

(b) Since 3 < 5, hence copy 3 and increment indices of A and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5								

(c) Since 6 > 5, hence copy 5 and increment indices of B and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6							

(d) Since $6 < 8$, hence copy 6 and increment indices of A and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8						

(e) Since $9 > 8$, hence copy 8 and increment indices of B and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9					

(f) Since $9 < 11$, hence copy 9 and increment indices of A and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9	11				

(g) Since $12 > 11$, hence copy 11 and increment indices of B and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9	11	12			

(h) Since $12 < 15$, hence copy 12 and increment indices of A and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9	11	12	14		

(i) Since $14 < 15$, hence copy 14 and increment indices of A and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9	11	12	14	15	

(j) Since $17 > 15$, hence copy 15 and increment indices of B and C

C:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	2	3	5	6	8	9	11	12	14	15	17

(k) Since the B array is empty, hence remaining element of A (i.e. 17) copy to C

Figure 3.8 (a-k): Merging two sorted arrays

Algorithm to merge two sorted arrays

Algorithm: MERGE (A, B, M, N, C)

[A and B are two sorted arrays, M is the number of elements of A, N is the number of elements of B and C is the resultant array]

1. Set I=0, J=0, K=0
2. Repeat while I<M and J<N
 - If A[I]<B[J] then
 - a) Set C[K]=A[I]
 - b) Set I=I+1
 - c) Set K=K+1
 - ELSE
 - a) Set C[K]=B[J]
 - b) Set J=J+1

```

    c) Set K=K+1
    [End of If]
[End of Loop]
3. Repeat while I<M
    a) Set C[K]=A[I]
    b) Set I=I+1
    c) Set K=K+1
[End of Loop]
4. Repeat while J<N
    a) Set C[K]=B[J]
    b) Set J=J+1
    c) Set K=K+1
[End of Loop]
5. Return

```

Time Complexity

In traversing operation, it required to access all the elements of the array exactly once. Therefore, the time complexity is O (n).

During insertion operation, a best case occurs when insert element at the last position of the array, then no array element movement is required. Therefore, the time complexity of the insertion operation in the best case is O (1), since not depends on the size of the array. The worst case occurs when insert element at the first position of the array then all the array elements required moving one position; n number of movements are required. Therefore, the time complexity of the insertion operation in the worst case is O (n), since depends on the size of the array. On an average, about half the elements of the array need to be moved. Therefore, the time complexity of the insertion operation in average case is O (n).

Similarly, in deletion operation, a best case occurs when deletes an element from the last position of the array and worst case occurs when deleting an element from the first position of the array. Therefore, the time complexity of deletion operation in the best case is O (1), in the worst case and average case is O (n).

The time complexity of this merging algorithm is O (m + n), where m + n is the number of output elements or the number of source elements combined.

Applications of Array

Arrays are used to implement mathematical elements like vectors, matrices (including sparse matrices), polynomials and different kinds of tables. Many databases consist of one-dimensional arrays whose elements are recorded. Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks and strings.

Representation of Polynomials

A polynomial p(x) is an expression of the form $(a_nx^n + a_{n-1}x^{n-1} + \dots + a_3x^3 + a_2x^2 + a_1x + a_0)$.

Each $a_i x^i$ is a term of the polynomial, where x is a variable, a_i is respective coefficient and n is a non-negative integer. The number n is called the *degree* of the polynomial. If $a_i = 0$, then the term is zero term, otherwise it is non-zero term

An important characteristic of a polynomial is that each term in the polynomial expression consists of two parts – one is a coefficient and the other is an exponent.

Consider the following polynomials:

$$10x^5 + 15x^3 - 7x^2 - x$$

Here (10, 15, -7, -1) are coefficients and (5, 3, 2, 1) are exponents.

Representation of Polynomials using Array

All the terms of a polynomial can be represented using array in two different ways:

- i) Linear array representation
- ii) Array of structure representation

Linear array representation

In this representation, the exponent/degree of a term can be treated as an index of the linear array and the coefficient of the term stored in that index of the array.

For Example, polynomial $10x^5 + 15x^3 - 7x^2 - x$ can be represented as follows:

[0]	[1]	[2]	[3]	[4]	[5]
0	-1	-7	15	0	10

Figure 3.9: Polynomial representation using linear array

The drawback of this representation is most of the entries of the linear array may remain zero for higher order polynomials.

Array of Structure representation

In this representation, each term defined as structure variable to store degree and coefficient of the corresponding term and polynomial can be represented by an array of structures as follows:

```
typedef struct poly
{
    int deg;
    int coef;
}poly;
poly term[100];
```

Now, if we represent the polynomials using an array, then we have to define the size of the array in advance. Therefore, there is a possibility of wastage of computer memory. However, this representation is simple enough.

The polynomial can also be represented by using linked list. We have discussed polynomial representation using a linked list in Chapter 5 in this book.

SPARSE MATRIX

One of the most important developments in scientific computing is a sparse matrix. This includes the data structures to represent the matrices, the techniques for manipulating them, the algorithms used and efficient mapping of the data structures and algorithms for high performance.

Definition: A matrix is said to be Sparse Matrix if most of its elements are zero (near more than 2/3 elements are zero), having a relatively small number of non-zero elements. A matrix that is not sparse is called **Dense Matrix**.

A sparse matrix is a two-dimensional array in which most of the elements have a null value or zero. However, for sparse matrices, this invokes wastage of a lot of memory space.

For example, when we are using with matrices of size 100 X 100 and if only 1000 entries are non-zeros and remaining locations are filled with zeros. This leads to huge amounts of memory wastage. The waste memory locations are $10000 - 1000 = 9000$ i.e., 9000 memory locations filled with zeros. Hence, a huge amount of memory is wasted.

The operations like transpose, addition, multiplication also take a lot of processing time if we store null values of a matrix in the array. To avoid such circumstances different techniques are used such as linked list. In simple words, sparse matrices are matrices that allow special techniques to take advantage of a large number of null elements and the structure.

Consider the following is an example of a sparse matrix:

$$\begin{bmatrix} 20 & 0 & 0 & 15 & 0 & -10 \\ 0 & 10 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 30 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.10: A Sparse Matrix

Sparse Matrix can be classified into two groups:

- i) Triangular Matrices and
- ii) Band Matrices

Triangular Matrices

Triangular matrices have the same number of rows, as they have columns, that they are square matrices. In the triangular matrix, both main and lower diagonals are filled with non-zero values or main diagonal and upper diagonals are filled with non-zero values.

$$\begin{bmatrix} 2 & 5 & 1 & 4 \\ 3 & 6 & 9 & 0 \\ 10 & 8 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix} \quad \text{(i)}$$

$$\begin{bmatrix} 3 & 6 & 10 & 5 \\ 0 & 9 & 2 & 4 \\ 0 & 0 & 7 & 1 \\ 0 & 0 & 0 & 8 \end{bmatrix} \quad \text{(ii)}$$

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 6 & 9 & 0 & 0 \\ 10 & 4 & 7 & 0 \\ 5 & 1 & 2 & 8 \end{bmatrix} \quad \text{(iii)}$$

$$\begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 9 & 8 \\ 0 & 8 & 10 & 3 \\ 7 & 1 & 6 & 2 \end{bmatrix} \quad \text{(iv)}$$

Figure 3.11: Examples of Triangular Matrices (i) Upper left triangular matrix, (ii) Upper right triangular matrix, (iii) Lower left triangular matrix, (iv) Lower right triangular matrix.

Triangular Matrices can also classify into four types:

- i) Upper Left Triangular,
- ii) Upper Right Triangular,
- iii) Lower Left Triangular and
- iv) Lower Right Triangular

A matrix A is an upper triangular matrix if its non-zero elements are found only in the upper triangle of the matrix, including the main diagonal.

A matrix A is a lower triangular matrix if its non-zero elements are found only in the lower triangle of the matrix, including the main diagonal.

Representation of Triangular Matrix

Storage optimization is one of the main objectives of the data structure. Optimizing storage is done by storing only non-zero elements. For example, a triangular matrix can be represented using a one-dimensional array, where only non-zero elements are stored.

$$\begin{bmatrix} A_{00} & 0 & 0 & 0 \\ A_{10} & A_{11} & 0 & 0 \\ A_{20} & A_{21} & A_{22} & 0 \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Figure 3.12: Elements of lower left triangular matrix

In the above Lower Left Triangular Matrix, $A_{ij} \neq 0$, for $i \geq j$, where i represent row and j represent column. In general, the position of the A_{ij} may be found by observing that there are $(i - 1)$ rows before it, which occupies a total of $1 + 2 + 3 + \dots + i = i * (i + 1) / 2$ elements.

In addition, in the same row there are j number of elements before it. Thus, this displacement of the A_{ij} , is $i * (i + 1) / 2 + j$ from the start of the array.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A_{00}	A_{10}	A_{11}	A_{20}	A_{21}	A_{22}	A_{30}	A_{31}	A_{32}	A_{33}

Figure 3.13: Representation of lower left triangular matrix into linear array

For example, A_{32} can be put into the 8th location of the one-dimensional array.

Therefore, the storage required for the original matrix is of n^2 , whereas considering the one-dimensional array, actual it required $n * (n + 1) / 2$ locations.

In the same way, storage optimization is done for the other triangular matrices and for the symmetric matrices.

Band Matrix

An important special type of sparse matrices is a band matrix, defined as follows:

Let a matrix A of order $n \times n$. If all matrix elements are zero outside a diagonally bordered band whose range is determined by constants α and β , where

$A_{ij} = 0$ when $\alpha < i - j$ or $\beta < j - i$: where $\alpha, \beta \geq 0$ and $i, j = 1$ to n , then α and β are called the lower and upper bandwidth respectively. The bandwidth of the matrix is the maximum of α and β .

A matrix is called a band matrix or banded matrix if its bandwidth is reasonably small.

Band Matrices can also classify into four types:

- i) Diagonal
- ii) Tri-diagonal
- iii) Penta-diagonal
- iv) $\alpha\beta$ Band

A diagonal matrix is a square matrix, if all the off-diagonal entries of A is zero, that is a band matrix with $\alpha = \beta = 0$ is a diagonal matrix.

A tridiagonal matrix is a matrix that has non-zero elements only on the main diagonal, the first diagonal below this and the first diagonal above the main diagonal, that is a band matrix with $\alpha = \beta = 1$ is a tri-diagonal matrix.

A penta-diagonal matrix is a matrix that has non-zero elements only on the main diagonal, the first two diagonals below this and the first two diagonals above the main diagonal, that is a band matrix with $\alpha = \beta = 2$ is a penta-diagonal matrix.

A $\alpha\beta$ band matrix is a matrix that has non-zero elements only on the main diagonal, the first limited diagonal below this and the first limited diagonal above the main diagonal, that is a band matrix with $\alpha \neq \beta$ is a $\alpha\beta$ band matrix.

When one puts $\alpha = 0$, $\beta = n-1$, one obtains the definition of an upper triangular matrix; similarly, for $\alpha = n-1$, $\beta = 0$ one obtains a lower triangular matrix. When a sparse matrix stored in a two-dimensional array of order $m \times n$ then there is a large amount of wastage of computer memory.

$$\begin{array}{c}
 \left[\begin{array}{cccc} 2 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 3 \end{array} \right] \\
 \text{(i)} \\
 \left[\begin{array}{cccc} 3 & 6 & 0 & 0 \\ 10 & 9 & 2 & 0 \\ 0 & 1 & 7 & 1 \\ 0 & 0 & 4 & 8 \end{array} \right] \\
 \text{(ii)} \\
 \left[\begin{array}{cccc} 3 & 7 & 4 & 0 \\ 6 & 9 & 3 & 6 \\ 10 & 4 & 7 & 1 \\ 0 & 1 & 2 & 8 \end{array} \right] \\
 \text{(iii)} \\
 \left[\begin{array}{cccc} 2 & 8 & 0 & 0 \\ 7 & 3 & 9 & 0 \\ 4 & 8 & 1 & 3 \\ 0 & 1 & 6 & 2 \end{array} \right] \\
 \text{(iv)}
 \end{array}$$

Figure 3.14: Band Matrices (i) Diagonal matrix, (ii) Tri-diagonal matrix, (iii) Penta-diagonal, (iv) $\alpha\beta$ Band

Three-Tuple Representation

In the case where non-zero terms did not form any pattern such as a triangle or a band, then the nonzero values of a sparse matrix can be stored as a list of three-tuples of the form (I, J, Value).

In this representation of the sparse matrix will be a two-dimensional array consisting three columns and $(T+1)$ rows, where T is the number of non-zero terms of the sparse matrix.

I= Row position

J= Column position

Value= Non-zero value at position (I, J)

The first row contains the number of rows, columns and non-zero elements of the matrix. Subsequent rows contain the positions of the non-zero elements and the non-zero value. The no. of columns of this representation is always fixed at three. Hence, it is called as three-tuples.

	[0]	[1]	[2]
[0]	6	6	8
[1]	0	0	20
[2]	0	3	15
[3]	0	5	-10
[4]	1	1	10
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	90
[8]	5	2	30

Figure 3.15: Three-tuple representation of a sparse matrix

In the above, figure 3.14 displays a three-tuple representation of a sparse matrix given in figure 3.10.

Algorithm of three-tuple representation of a sparse matrix

```

Algorithm: THREE_TUPLE (A, M, N, B)
[A is a sparse matrix of order M x N stored in matrix B]
1. Set C=1
2. For I=0 to M-1 do
    For J=0 to N-1 do
        If A[I][J]≠0 then
            Set B[C][0] = I
            Set B[C][1] = J
            Set B[C][2] = A[I][J]
            Set C = C + 1
        [End of If]
    [End of Loop]
[End of Loop]
3. Set B[0][0]=M, B[0][1]=N, B[0][2] = C-1
4. Return

```

Importance of Sparse: Matrices Sparse matrices occur in many applications including solving partial differential equations (PDEs), text-document matrices used for latent semantic indexing (LSI), linear and non-linear optimization, and manipulating network and graph models.

STRINGS

A string is a sequence of characters. In C language, the end of the string is marked with a special character, the null character, which is simply the character with the ASCII value 0 (the null character has no relation except in name to the null pointer). The null or string terminating character is represented by a character escape sequence `\0`. However, remember that \ and 0, they are not two separate characters.

Any sequence or set of characters defined by double quotation symbols is a constant string. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string terminating character `\0`. Programs rarely get to see this end marker as most functions which handle strings use it or add it automatically. To do anything else with strings, we must typically call functions.

Strings are stored in memory as ASCII codes of characters that make up the string appended with `\0` (ASCII value of null). Normally, each character is stored in one byte (may depend on the compiler); successive characters are stored in successive bytes.

Character	H	E	L	L	O		I	N	D	I	A	<code>\0</code>
ASCII Code	72	101	108	108	111	32	87	111	114	108	100	0
Address	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011

Figure 3.16: String representation

The last character is the null character having the ASCII value zero.

Declaring and Initializing Strings

As discussed earlier, strings in C are represented by an array of characters. Therefore, to declare a string, we will declare a character array. The general form of declaring a string is:

```
char string_name [size];
```

The size determines the no of characters that a string can hold. When the compiler assigns a character string to a character array, then it automatically supplies a null character at the end of the string. Therefore, while choosing the length or size of the string it must be one more than the number of characters you want to accommodate in the string. If the null character is not assigned at the end of the string, then the extra characters may be displayed.

Following are the examples of string declaration:

Example:

```
char name[30];
char city[25];
```

A string is initialized only when it is declared. We can initialize a string in the following ways:

```
char city[8] = { 'K', 'o', 'l', 'k', 'a', 't', 'a', '\0' };
```

Then the string city is initializing to Kolkata.

The above initialization is perfectly valid but C offers a special way to initialize strings.

The above string can be initialized as

```
char city[8] = "Kolkata";
```

The characters of the string are enclosed within a part of double quotes. The compiler takes care of string enclosed within a pair of double quotes. The compiler takes care of storing the ASCII codes of the characters of the string in the memory and stores the null terminator at the end.

While initializing a string the dimension may be optional (like numeric arrays) and in this case if the dimension is not specified the compiler will infer the dimension from the number of initializers. For example, the following initialization is valid.

```
char city[] = "Kolkata";
```

In this case, the compiler will assume the size of the string as 8.

In addition, the size may be more than the number of characters in the initialize; in this case, the remaining memory cells will be padded with null characters. For example, consider the following:

```
char city[10] = "Kolkata";
```

Therefore, the storage in the memory will be like:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
K	o	L	k	a	t	a	\0	\0	\0

Figure 3.17: String Representation

However, keep in mind that size cannot be less than the number of characters in the initials.

For example the initialization

```
char city[7] = "Kolkata";
```

is invalid and this time compiler may produce a compile time error or there will be no string terminator (\0) as a result, if we print it we may get some undesirable result.

Reading String

In general, the function `scanf()` with `%s` format specification is used to read the character string from the terminal.

Example:

```
char city[15];
scanf("%s", city);
```

The `scanf()` statement has a drawback that it just terminates the statement as soon as it finds a whitespace character (new lines, tabs, blank spaces, etc.), suppose if we type the string Hello India then only the string Hello will be read and since there is a blank space after word Hello it will terminate the string. Therefore, in general, it is not possible to read a line of text using `scanf()`.

Example:

```
char city [15];
scanf("%[^\\n]", city);
printf("\n %s", city);
```

Note: We can use the `scanf()` without the ampersand symbol before the variable name.

In addition, the unformatted input function `gets` can be used to read a string. The general form of the function `gets` is

```
gets(string_name);
```

The function `gets` reads characters into the string from the terminal until a newline character is encountered and at last it assigns a null character at the end of the string.

Example:

```
char city[15];
gets(city);
```

Writing String

The printf statement along with format specifier %s can be used to display strings onto the screen.

Example:

```
printf("%s", string);
```

can be used to display the entire contents of the array string.

Also, printf function can be used to print a constant string like:

Example:

```
printf("Hello India");
```

We can also use the unformatted output function puts() to display a string onto the screen.

Example:

```
puts(string);
```

To display a constant string we can use puts like this:

```
puts("Hello India!");
```

We cannot manipulate strings since C does not provide any operators for the string. For instance, we cannot assign one string to another directly.

Example:

```
char String1[30], String2[30];
String1 = "Hello";           /* Invalid */
String2 = "India";           /* Invalid */
String1 = string2;          /* direct assignment not possible */
```

Are not valid. To copy the characters in one string to another string we may do so on a character-to-character basis.

Similarly, it is also not possible to compare two strings directly through the relational operator (==).

Example:

```
char String1[] = "Hello";
char String2[] = "India";
if(String1 == String2) /* direct comparison not possible */
    printf("\n Strings are same");
else
    printf("\n Strings are not same");
```

Array of strings

So far, we have been discussing the processing of only a single string, which is a one-dimensional array of characters. However, in many applications, it requires processing a set of strings. In such situation, we require a two-dimensional array of characters, which is called Array of Strings. Each

element of a one-dimensional character array is an ASCII character. However, each element of a two-dimensional array is a string.

So for example, consider the following declaration:

```
char os[5][10] = {      "Unix",
                      "Linux",
                      "Windows",
                      "Macintosh",
                      "DOS"
};
```

The above array of strings can hold maximum of five strings. In addition, each string can hold maximum of ten characters including null.

After the above declaration, we may have the following situation in memory:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
os[0]	U	N	I	x	\0					
os[1]	L	I	N	u	x	\0				
os[2]	W	I	N	d	o	w	s	\0		
os[3]	M	A	C	i	n	t	o	S	h	\0
os[4]	D	O	S	\0						

Figure 3.18: Array of strings

Here, os[0] means the string “Unix”, os[1] means the string “Linux”, os[2] means “Windows”, os[3] means “Macintosh” and os[4] means “DOS”.

Operations Perform on Strings

Different kinds of operations are to be performed on Strings.

Table 3.2: Operations on Strings

Operation	Description
Find length	Finds the length of the string
Copy	Copies one string to another string.
Concatenation	Appends one string at the end of another string.
Compare	Compares two strings to find whether they are identical or not.
Reverse	Reverse the string.
Uppercase	Converts lowercase alphabets (if any) of a string to uppercase.
Lowercase	Converts uppercase alphabets (if any) of a string to lowercase.
Sorting	Arranging all the strings in lexicographical order.
String matching	Searching a pattern within a string.

Length of a String

The length of a string returns the total number of characters of the string, which is not necessarily

equal to its storage capacity. The length of the empty string is zero.

Algorithm to find the length of a string

Algorithm: STRING_LENGTH (S)

[S is a given string]

1. Set N = 0
2. Repeat step 3 while S[N] ≠ Null do
3. N = N + 1
 [End of loop]
4. Return N

In C language, `strlen()` function counts and returns the number of characters in a string. The length does not include a null character, but spaces. The general form of `strlen()` function is

```
n = strlen(string);
```

Here n is an integer variable that receives the value of the length of the given string.

Program 3.2: Find the length of the string using `strlen()` function

```
#include<stdio.h>
#include<string.h>
main()
{
    char str[100];
    int length;
    printf("Enter the string:");
    gets(str);
    length = strlen(str);
    printf("\nNumber of characters in the string=%d",length);
}
Output:
Enter the string:Hello world
Number of characters in the string=11
```

String Copy

Copy the current value of the string into another string that has the same value as the original string. One cannot copy a string using the assignment operator.

Algorithm to copy a string to another string

Algorithm: STRING_COPY (T, S)

[S is a source string and T is a target string]

1. Set I = 0
2. Repeat step 3 while S[I] ≠ Null do
3. T[I] = S[I]
4. I = I + 1
 [End of loop]
5. Set T[I] = Null
6. Return

3.26 | Data Structures and Algorithms with C

In C language, the `strcpy()` function used to copy the string. The syntax of the function is illustrated below:

```
strcpy(string1, string2);
```

The `strcpy` function assigns the contents of `string2` to `string1`, where `string2` may be a character array variable or a string constant. However, `string1` should be big enough to hold the string that would be copied to it.

Program 3.3 : Copy the content of a string to another string using `strlen()` function

```
#include<stdio.h>
#include<string.h>
main()
{
    char string1[30];
    char string2[] = "Hello World";
    printf("\nSecond string was %s",string2);
    strcpy(string1, string2);
    printf("\n Now first strings is:%s",string1);
}
Output:
Second string was: Hello World
Now first strings is: Hello World
```

Concatenation of two Strings

When combine two strings, that is, appends the contents of one string at the end of another string. This operation is called concatenation.

Algorithm to concatenate two strings

```
Algorithm: STRING_CONCAT (T, S)
[string S appends at the end of string T]
1. Set I = 0, J = 0
2. Repeat step 3 while T[I] ≠ Null do
3.     I = I + 1
        [End of loop]
4. Repeat step 5 to 7 while S[J] ≠ Null do
5.     T[I] = S[J]
6.     I = I + 1
7.     J = J + 1
        [End of loop]
8. Set T[I] = Null
9. Return
```

In C language, The `strcat()` function joins two strings together. The general form of `strcat()` function is:

```
strcat(string1, string2);
```

where string1 and string2 are character arrays. When the function `strcat()` is executed string2 is appended to string1. The string at string2 remains unchanged.

Program 3.4: Concatenate of two strings using `strcat()` function

```
#include<stdio.h>
#include<string.h>
main()
{
    char str1[100], str2[100];
    int length;
    printf("Enter the first string:");
    gets(str1);
    printf("Enter the second string:");
    gets(str2);
    strcat(str1, str2);
    printf("\nString 1 after concatenation = %s", str1);
    printf("\nString 2 after concatenation = %s", str2);
}
```

Output:

```
Enter the first string:Taj
Enter the second string:mahal
String 1 after concatenation = Tajmahal
String 2 after concatenation = mahal
```

From the above program segment, the value of str1 becomes Tajmahal. The string at str2 remains unchanged as mahal.

Comparison of two Strings

Comparison of two strings to decide whether they represent a same string or not.

Algorithm to compare two strings

Algorithm: STRING_COMPARE (T, S)

[Compare between string S and T]

1. Set I = 0
2. Repeat step 3 while S[I] ≠ Null and S[I] = T[I] do
3. Set I = I + 1
 [End of loop]
4. If S[I] = Null and T[I] = Null Return True
5. Return False

In C you cannot directly compare the value of two strings in a condition like `if(string1==string2)`.

Most libraries, however, contain the `strcmp()` function, which compares two strings and returns a zero if two strings are identical , or a non-zero number if the strings are not the same. The syntax of `strcmp()` is given below:

```
strcmp(string1, string2);
```

where `string1` and `string2` may be string variables or string constants. The function returns a negative number if `string1` is alphabetically less than the `string2` and a positive number if `string1` is greater than `string2`. More correctly, `strcmp()` returns the numeric difference between the first two non-matching characters.

Example:

```
strcmp("Newyork", "Newyork");  
it will return zero because 2 strings are equal.
```

```
strcmp("their", "there");  
it will return -9 which is the numeric difference between ASCII 'i' and ASCII 'r'.  
strcmp("The", "the");  
it will return -32 which is the numeric difference between ASCII 'T' & ASCII 't'.
```

Program 3.5 : Compare two strings using strcmp() function

```
#include<stdio.h>  
#include<string.h>  
main()  
{  
    int r;  
    char string1[] = "The";  
    char string2[] = "the";  
    r = strcmp(string1, string2);  
    if(r == 0)  
        printf("Strings are identical");  
    else  
        printf("Strings are not identical");  
}
```

Output:
Strings are not identical

String Reversal

The reverse of a string is a string with the same characters, but in the reverse order. For example, if a given string is “abcd”, then the reverse of the string is “dcba”. Whenever a string is the reverse of itself (e.g. madam) is called a palindrome.

Algorithm to reverse a string

Algorithm: STRING_REVERSE (S)

[*S* is a given string]

1. Set N = Length (*S*)

```

2. Set I = 0 and J = N-1
3. Repeat while I < J do
4.     Set Temp = S[I]
5.     Set S[I] = S[J]
6.     Set S[J] = Temp
7.     Set I = I + 1 and J= J -1
        [End of Loop]
8. Return

```

In C language, `strrev()` function reverses the characters in a string. The general form of `strrev()` function is

```
strrev(string);
```

Program 3.6: Reverse a string using strcmp() function

```

#include<stdio.h>
#include<string.h>
main()
{
    char string[] = "HELLO";
    printf("\nThe string in reverse:%s", strrev(string));
}
Output:
The string in reverse: OLEH

```

String Matching

String matching (or string searching) is an operation to find the position of all occurrences of a pattern within a given string or text. String matching is used for different application software like text editors to find and replace all.

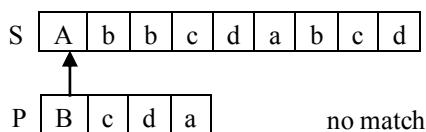
Brute Force (Naïve) String Matching

Brute Force string matching technique is a simple way to test where a pattern occurs within a string in all the possible positions. Brute Force string matching algorithm also known as Naïve algorithm. Naïve means basic.

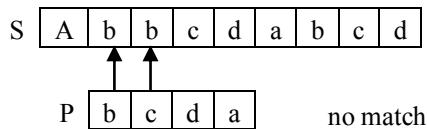
Consider m is the length of a pattern and n is the length of the searchable string or text. The algorithm can be interpreted as a sliding a pattern P over the text string S and observe for which shift all the characters in the pattern match the corresponding character in the string. Specifically, the pattern is shifting successively from I = 0 to n - m, and for each shifting compare the pattern P[1 to m] with the string S[I + 1 to I + m].

Example:

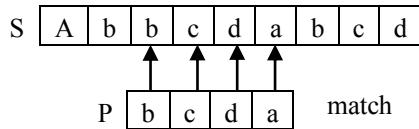
Suppose a given text string S = "ababcdabcd" and pattern string P = "bcda".



Now, T is shifted one position right



Again, T is shifted one position right



Pattern string P is a substring of text string S

Figure 3.19: String matching

Brute Force (Naïve) string matching algorithm

Algorithm: STRING_MATCHING (S, P)

[S is a text string and P is a pattern string]

1. Set N = Length (S)
2. Set M = Length (P)
3. Repeat For I = 0 to N-M do
4. Set J = 0
5. Repeat while J < M and S[I + J] = P[J] do
6. Set J = J + 1
7. If J ≥ M Return I
 [End of Loop]
8. [End of Loop]
8. Return -1

In the above algorithm, the for-loop is executed at most $n - m + 1$ times and the while loop are executed at most m times. Therefore, in the worst case, the running time complexity of this algorithm is $O((n - m + 1)m)$, which is $O(nm)$, if $n \ll m$. Consider an example where searching for a pattern like "xxxxy" in a string like "xxxxxxxxy". In the average case, the running time complexity of this algorithm is $O(m + n)$.

The advantage of this algorithm that it is a very simple technique and that does not require any pre-processing. Therefore, total running time is the same as its matching time. Although, it is a very inefficient method. Because this method takes only one position movement in each time.

In C language, the `strstr()` function finds the first occurrence of a substring in another string. On success, `strstr()` returns a pointer to the element in str1 where str2 begins (points to str2 in str1). On error (if str2 does not occur in str1), `strstr()` returns null. The general form of `strstr()` function is

```
ptr = strstr(str1, str2);
```

Program 3.7: Search a string using strstr() function

```
#include<stdio.h>
#include<string.h>
main()
{
    char string[] = "International";
    char searchstring[] = "nation";
    char substring[30];
    strcpy(substring, strstr(string, searchstring));
    printf("\nSub strings is: %s", substring);
}
Output:
Sub strings is: national
```

Summary

- An array is a collection of variables of the same data type that are stored in consecutive memory location under a common name.
- An array can be an integer, character or floating-point data type can be initialized only during declaration.
- An array name is an address of the first element of the array.
- A matrix is said to be Sparse Matrix if most of its elements are having a relatively small number of non-zero elements.
- The string is a sequence of characters.

Exercises

1. A two-dimensional array A with n rows time and m columns can be represented in either row major or column major form. Establish the address translation functions to locate any element from the one-dimensional array. The element is specified by two-dimensional parameters along with the data type.
2. Write an algorithm to calculate the address of any element A [I, J] of a two-dimensional array A [1: M, 1: N]. Assume the array is stored in column major order, B is the base address of the array and w is the size of the each element in the array.
3. Explain row major and column major implementation of a two-dimensional array.
4. Consider the array int p [10][10] and base address 2000, then calculate the address of p[[2][3] in the row and column major order.
5. How can we represent sparse matrix efficiently in the memory? Write an algorithm to find the transpose of a sparse matrix.
6. Explain with a suitable example the storage and manipulation of elements of a sparse matrix.
7. Choose the correct alternatives for the following:
 - i) In C language, arrays are stored in which representation
 - a) Column major
 - b) Row major
 - c) Layer major
 - d) None of these
 - ii) Each element of an array arr[20][50] requires 4 bytes of memory. The base address of arr is 2000. The location of arr[10][10] when the array is stored as column major order.

3.32 | Data Structures and Algorithms with C

- a) 2820 b) 2840 c) 4048 d) 4840
- iii) The expression which accesses the (i, j) th entry ($i = 0, 1 \dots m-1, j = 0, 1 \dots n-1$) of an $m \times n$ matrix (stored in column major order) is
a) $n \times (i - 1) + j$ b) $m \times (j - 1) + i$ c) $n \times (i-1) + (j - 1)$ d) $m \times (j-1) + (i - 1)$
- iv) If the address of $A[1][1]$ and $A[2][1]$ are 1000 and 1010 respectively and each element occupies 2 bytes then the array has been stored in _____ order.
a) Column major b) Row major c) Layer major d) Matrix major
- v) The largest element of an array index is called its
a) lower bound b) range c) upper bound d) None of these

CHAPTER 4

POINTER

“The duties of the Pointer were to point out, by calling their names, those in the congregation who should take note of some point made in the sermon.” —H. B. Otis

The pointers are one of the most important and powerful data structure. A pointer is a variable that is used to store an address or location of another variable. We can say that the pointer points to another variable. A pointer declaration consists of a data type, the indirection operator (*) and a variable name. The indirection operator is also called the dereference operator. Do not confuse the indirection operator with the multiplication operator, although they have the same symbol. The indirection operator is a unary operator, whereas multiplication operator is a binary operator.

The pointers are very useful in dynamic memory allocation, and used to modify variable arguments passed to a function.

Pointer Declaration

The general form of pointer declaration is:

```
data-type *pointer-variable-name;
```

The data type specifies the type of data to which the pointer points, that means it can store an address of that data type.

Example:

```
int *p;  
char *q;  
float *r;
```

In the above declaration, p is a pointer to an integer; it can store an address of integer type variables. The q is a pointer to a character, it can store an address of character type variables (it does not mean that q holds the character value rather it means that q contain the address of a character variable). Similarly r is a pointer to a floating-point, it can store an address of floating-point type variables. Remember that address is always an unsigned integer.

Address of Operator

The address-of operator (&) can return an address in memory of its operand. Do not confuse the address-of operator with the bitwise AND operator, although they have the same symbol. The address-of operator is a unary operator, whereas bitwise AND operator is a binary operator.

In the following example, display the addresses of the variables c, a and f along with their values. The addresses of the variables are dependent on compiler and operating system.

KEY FEATURES

-  Null Pointer
-  Void pointer
-  Generic Function
-  Dangling Pointer
-  Pinter to Pointer
-  Array of Pointer
-  Pointer to an Array
-  Pointer to Function
-  Dynamic memory allocation

Example:

```
#include<stdio.h>
main()
{
char c = 'C';
int a = 50;
float f = 3.45;
printf("Address of c = %u, Value of c = %c\n", &c, c);
printf("Address of a = %u, Value of a = %d\n", &a, a);
printf("Address of f = %u, Value of f = %f\n", &f, f);
}
Output:
Address of c = 65529, Value of c = C
Address of a = 65522, Value of a = 50
Address of f = 65518, Value of f = 3.450000
```

Address	Value	Variable name
	:	
65529	67	c
65522	50	a
65518	3.45	f
	:	
Memory		

Figure 4.1: Variables in memory

We can store the address of a variable in a pointer variable. It is also possible to store the address of a variable into an unsigned integer variable. That is useless, you cannot use the indirection operator as a prefix of a variable except a pointer variable.

Example:

```
int *p;
int a = 50;
p = &a;
```

Suppose, the variable a is stored at memory location 65522 and size of it four bytes. That means variable a is stored from memory location 65522 to 65525. Since its size four bytes, four memory locations are required to store the value.

In the above example, the memory location of the variable a stored into variable p. The variable p is an integer pointer that means it can store an address of integer type variables. Variable a has a value of 50. Then, after the assignment statement, pointer p will have the value 65522, which is the starting memory location of variable a and we can say pointer p points to the variable a.

Indirection Operator

The indirection operator (*) returns the value of the memory address which is stored in its operand. It

is essentially the opposite of the address of operator. This operator is also known as pointer operator.

Note: Address of operator (&) returns the address of a variable and indirection operator (*) return the value of that address.

Example:

```
int a, b, *p;
a = 50;
p = &a;
b = *p;
```

Suppose, the variable a is at memory location 65522 and a has a value of 50. Now, the memory location of variable a is assigned to the pointer variable p, hence p will have 65522. The assignment statement sets the value of the memory location 65522, which is stored into p. Now b has the value 50 because 50 are stored at location 65522. The indirection operator and address-of operators are the complements of each other.

Example:

```
#include<stdio.h>
main()
{
    int a = 50, b;
    int *p;
    p = &a;
    b = *p;

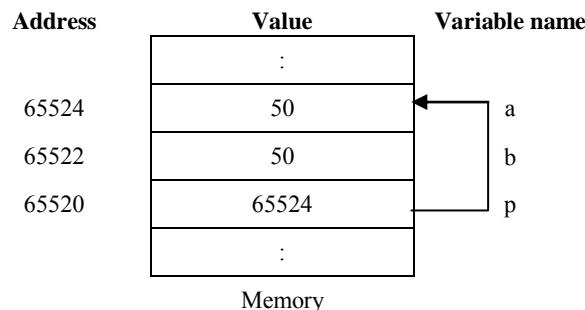
    printf("Value of a = %d\n", a);
    printf("Address of a = %u\n", &a);
    printf("Value of p = %u\n", p);
    printf("Value of *p = %d\n", *p);
    printf("Value of b = %d\n", b);
    printf("Address of b = %u\n", &b);
    printf("Address of p = %u\n", &p);
}
```

Output:

```
Value of a = 50
Address of a = 65524
Value of p = 65524
Value of *p = 50
Value of b = 50
Address of b = 65522
Address of p = 65520
```

We can get the value of b in the following manner,

$$\begin{aligned} b &= *p \text{ that means } b = * (65524) \text{ [since } p = 65524] \\ &= 50 \quad \text{[value of address 65524]} \end{aligned}$$

**Figure 4.2:** Variables in memory

Type casting of Pointers

The pointer variables have their own data type. Unlike basic data types, the pointers do not support implicit type conversion. Therefore, pointer types can be converted to other pointer types using the explicit type casting mechanism.

Example:

```
int a = 10;
float b = 12.3;
int *p;
float *q;
p = &a;
q = &b;
q = p;           /* this is invalid */
p = q;           /* this is invalid */
q = (float*)p;   /* this is valid */
p = (int*)q;     /* this is valid */
```

Note: If the compiler encountered some conversion of a pointer that caused the pointer to point to a different type then the compiler gives an error message like “*Suspicious pointer conversion*”.

Null Pointer

A null pointer is a pointer value that points to no valid location. A null pointer is a constant pointer (often represented by address zero) that is compatible with any pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any variable defined within the program.

A null pointer value is an address that is different from any valid pointer. Assigning the integer constant 0 to a pointer assigns a null pointer value to it. The mnemonic NULL (defined in the standard library header file, stdio.h) can be used for legibility.

Example:

```
int *p;
*p = 12;
```

Since p is not defined, therefore p may contain a garbage value (say, p = 65550). Now purposely or accidentally when a value assigning to *p then there are chances that modify that memory location

(e.g. 65550) which is not allocated by the program. Therefore, you should use

```
int *p = NULL;
```

Dereferencing a null pointer is meaningless, typically resulting in a run-time error. Therefore, before the use of pointer we should check it with NULL value. All pointers can be successfully tested for equality or inequality to NULL which is logically equivalent to false.

In dynamic memory allocation, when calloc() or malloc() function fails to allocate memory block then they return NULL. Therefore, after the function call, it requires checking whether the memory block is allocated or not, before the use of the memory.

Example:

```
int *p;
p = malloc(10 * sizeof(int));
if(p==NULL)
{
    printf("Insufficient Memory");
    exit(1);
}
```

The null pointer also indicates the failure of a search operation, such as in the linked-list programs.

Void Pointer

A void pointer is a pointer, which may store the address of any type of variable. That means void pointer is a pointer to anything. The void pointer is also known as a type-less pointer or generic pointer.

Note that a variable of type void cannot be declared. However, the return type of a function may be void.

```
void a;           /* this is invalid */
void *r;          /* valid */
```

The void pointers are used to store the address of any type of variable temporarily. Since the void pointer is a typeless pointer, the compiler has no information that how many bytes of data it will retrieve from the memory starting from stored address. Therefore, the indirection operator cannot be used with a void pointer.

Example:

```
int a = 10;
float b = 12.3;
int *p;
float *q;
void *r;
p = &a;
q = &b;
printf("%d", *p);
r = p;
/* this is invalid */
printf("%d", *r);
```

A void pointer may be assigned to any non-void pointer without explicit type casting operator. Non-void pointer may also be assigned to void pointer without explicit type casting.

Example:

```
r = p;
p = r;
or
r = q;
q = r;
```

where p, q, and r are declared as in the previous example.

The void pointer may be used as a lvalue or as a rvalue. Therefore, by using this concept, void pointer allows violating the basic rule of the type conversion of the pointer.

Example:

```
r = p;
q = r;
```

where p is an integer pointer and q is a floating point pointer. This implies,

```
q = p;
```

Generic Functions

The void pointer is used to write generic functions, which can accept any type of parameter. A group of functions that look the same, except the types of one or more of their arguments. A generic function allows defining a function to replace that group of functions.

Suppose we have a function that can be used to interchange the value of two integer type variables. Now we require another function to interchange the value of two floating-point type variables. For interchange the value of two long double type variables, we need one more function. Therefore, for the different data type, we require to write different function.

Example:

```
/* Interchange the value of two variables */
#include<stdio.h>
void swapi(int *a, int *b);
void swapf(float *a, float *b);
main()
{
int i = 10, j = 20;
float x = 12.3, y = 53.4;
swapi(&i, &j);
printf("i = %d, j = %d", i, j);
swapf(&x, &y);
printf("x = %f, y = %f", i, j);
}
/* Function to interchange two integer type variables*/
void swapi(int *a, int *b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
```

```

}
/* Function to interchange two floating-point type variables*/
void swapf(float *a, float *b)
{
float temp;
temp = *a;
*a = *b;
*b = temp;
}

```

Instead of a group of similar type functions, we can use a generic function. A generic function permits to define a function to replace that group of functions.

Example:

```

/*Interchange the value of two variable using generic function */
#include<stdio.h>
void swap(void *a, void *b, int n);
main()
{
int i = 10, j = 20;
float x = 12.5, y = 53.5;
swap(&i, &j, sizeof(int));
printf("i = %d, j = %d\n", i, j);
swap(&x, &y, sizeof(float));
printf("x = %f, y = %f\n", x, y);
}
/* Generic function to interchange two variables */
void swap(void *a, void *b, int n)
{
char *p, *q, temp;
p = (char*)a;
q = (char*)b;
while(n>0)
{
temp = *p;
*p = *q;
*q = temp;
p++;
q++;
n--;
}
}
Output:
i = 20, j = 10
x = 53.500000, y = 12.500000

```

Note that a character is always a byte and a character pointer works as a byte pointer. The sizeof operator calculates the size of the data type in bytes. The generic function sequentially interchanges

each byte of the two variables, since every type of variable can be treated as sequence of character.

Dangling Pointer

Dangling pointers are pointers that do not point to a valid variable of the proper type. Dangling pointers are created when memory is deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. When the system reallocates the previously freed memory to another process and the original program dereferences the dangling pointer, then it may produce an unpredictable result, as the memory may now contain completely different data.

Example:

```
{
    int *cp;
{
    int c;
    cp = &c;
}
/* c is in out of scope, cp is now a dangling pointer */
}
```

A solution to the above is to assign NULL to cp immediately before the inner block is exited. Another frequent source of dangling pointers is a combination of malloc () and free () library function calls: a pointer becomes dangling when the block of memory it points to is freed. As with the previous example, one way to avoid this is to make sure to reset the pointer to null after freeing its reference.

Example:

```
#include <stdlib.h>
{
    int *cp = malloc (100);
    free ( cp );      /* cp now becomes a dangling pointer */
    cp = NULL;        /* cp is no longer dangling */
}
```

Allowable Operations with Pointer

There are only a few numbers of operations of pointer are allowed. The allowable operations of pointer as follows:

- Increment and decrement operations with pointer variables.
- Subtraction of two pointer variables.
- Addition and subtraction of integer value with pointers.
- Relational operations between two pointer variables.
- Assignment Operation.

Note: Addition of two pointer variables of any type is not accepted.

Arithmetic Operation with Pointer

Addition and subtraction of integer value with pointer variable are permitted. However, other arithmetic operations, including multiplication, division are not permitted with pointers. Non-integer values such as floats or double value addition or subtraction with pointer variables are not accepted.

The format for adding or subtracting an integer to a pointer is:

```
Pointer-variable + integer-value
Pointer-variable - integer-value
```

One pointer may be subtracted from another pointer of the same type and the result will be an integer, not a pointer.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i, *p, *q;
p = a + 1;
q = a + 3;
printf("%d", q - p);
Output:
2
```

Increment and Decrement operations with Pointer

Increment operation can be used with pointer variable that is similar to add one with pointer variable. Decrement operation can also be used with pointer variable that is similar to subtract one from pointer variable. The formats for incrementing or decrementing a pointer are:

```
Pointer-variable++
Pointer-variable--
```

There are three possible mix pointer increment and indirection are as follows:

- i) *p++
- ii) *++p
- iii) ++*p

Where p is a pointer variable.

i) *p++ returns the content at the location being pointed by p and then increment the pointer by one. The pointer p will point the next element. The decrement operator can be used with pointers, in the same manner, to move to the previous element. Note that the compiler may give some warning or error message if next memory location is not allocated by the program statically or dynamically.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", *p++);
printf("%d ", p - a);
printf("%d ", *p);
Output:
10 1 20
```

It should be noted that increment (++) and decrement (--) operators have a higher precedence than the precedence of indirection operator (*).

ii) *++p increments the pointer by one and then it returns the content at the location being pointed by p. The pointer p will point the next element. The decrement operator can be used with pointers, in the same manner, to move to the previous element. Note that the compiler may give some warning or

error message if the next memory location is not allocated by the program statically or dynamically.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", *++p);
printf("%d ", p - a);
printf("%d ", *p);
```

Output:
20 1 20

ii) $++*p$ increments the value at the location being pointed by p. The pointer p will point to the same element. The decrement operator can be used with pointers in the same manner to decrement the value at the location being pointed.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
p = a;
printf("%d ", ++*p);
printf("%d ", p - a);
printf("%d ", *p);
```

Output:
11 0 11

Relational Operation with Pointer

A pointer variable can be compared with another pointer variable of the same type using different relational operators. Table 4.1 describes allowable relational operations with pointer.

Table 4.1: Relational operators with pointer

Operators	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

A pointer variable can also be compared with a NULL pointer.

The Table 4.2 shows pointer notations and array notations. The notations are based on the following code:

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr;
ptr = arr;
```

Table 4.2: Pointer notation and array notation

Pointer Notation	Array Notation	Results
ptr	arr, &arr[0]	Address of the first element of the array.
ptr++	&arr[x++]	Move the pointer to the next element of the array.
ptr + 1	&arr[1]	Address of the second element of the array.
*ptr	arr[0]	Value of the first element of the array.
* (ptr + 1)	arr[1]	Value of the second element of the array.
* (ptr) + 1	arr[0] + 1	Add 1 and the first element of the array together.
*ptr++	arr[x++]	Move to the next element in the array after the value is used.
*ptr--	arr[x--]	Move to the previous element in the array after the value is used.
*++ ptr	arr[+x]	Move to the next element in the array before using the value.
*-- ptr	arr[--x]	Move to the previous element in the array before using the value.
(* ptr)++	arr[0]++	Increment the first element of the array by 1.
(* ptr)--	arr[0]--	Decrement the first element of the array by 1.
(* (ptr + 2))++	arr[2]++	Increment the third element of the array by 1.
(* (ptr + 2))--	arr[2]--	Decrement the third element of the array by 1.

Implicit Scaling in Pointer Addition

When several data items of the same type are placed consecutively in the memory, then a unit increment or decrement to a pointer to any of the data items, always gives the address of the next or previous items. This works independently of the data item.

Example:

```
int *p;
float *q;
```

Now, when we write `p + i` where `i` is an integer number, then the system evaluates as `p + i × sizeof(int)`.

Similarly, `q + i` systematically evaluate as `q + i × sizeof(float)`.

Therefore, we always get the address of i^{th} element independently of the data item.

Pointer to Pointer

A pointer is a variable that is used to store an address or location of another variable. That variable may also be a pointer. The pointer to pointer is a special type of pointer that is used to store an address of another pointer variable.

Example:

```
#include<stdio.h>
main()
{
```

```

int a = 50, b, c;
int *p;
int **q;
p = &a;
b = *p;
printf("Value of a = %d\n", a);
printf("Address of a = %u\n", &a);
printf("Value of p = %u\n", p);
printf("Value of *p = %d\n", *p);
printf("Value of b = %d\n", b);
q = &p;
c = **q;
printf("Value of p = %u\n", p);
printf("Address of p = %u\n", &p);
printf("Value of q = %u\n", q);
printf("Value of **q = %u\n", **q);
printf("Value of c = %d\n", c);
printf("Address of q = %u\n", &q);
}

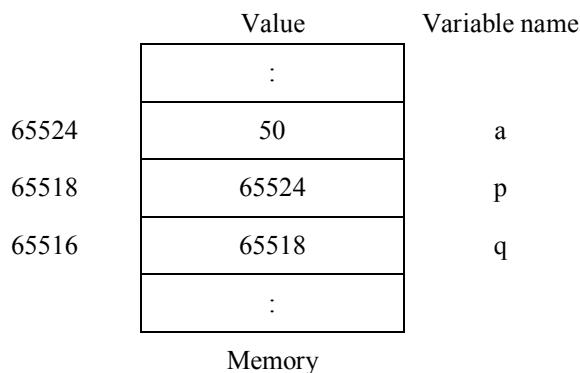
```

Output:

```

Value of a = 50
Address of a = 65524
Value of p = 65524
Value of *p = 50
Value of b = 50
Value of p = 65524
Address of p = 65518
Value of q = 65518
Value of **q = 50
Value of c = 50
Address of q = 65516

```

**Figure 4.3:** Variables in memory

Pointers and Arrays

In the C language, pointers and arrays are closely related. The name of an array is also the address of the first element of the array and points to the location in memory of the first element in the array.

Arrays are nothing but pointers. Hence, there is no index overflow or underflow checking. Arrays are also called as a static pointer. Since the name of an array is a constant pointer.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i;
a[5] = 5;           /* overflow */
a[-5] = -5;        /* underflow */
```

No index overflow or underflow error message will be given by the compiler and index overflow or underflow checking does not take place at runtime.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int *p;
a = p;             /* this is invalid */
```

Here, the pointer variable p is assigned to the base address of the array. This is not accepted because the base address of the array is a constant and a pointer is the variable.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i, *p;
p = a;
for(i=0; i<5;i++)
    printf("%d", p[i]);
```

The base address of the array is assigned to the pointer variable p. The pointer variable p can be used just like as it is an array name and the printf statement outputs the value of the each element of the array.

An alternative way doing the same thing with the following statement that uses the address of the first element of the array is assigned to the pointer variable p.

```
p = &a[0];
```

An array element a[i] can be written as *(a+i) in pointer representation and a+i signify the address of that element.

Example:

```
int a[5] = {10, 20, 30, 40, 50};
int i;
for(i=0; i<5;i++)
    printf("%d", *(a+i));
```

A two-dimensional array can be thought as a one-dimensional array and that may be represented by a pointer.

Example:

```
int a[3][4] = {{11, 12, 13, 14}, {21, 22, 23, 24}, {31, 32, 33, 34}};
```

```

int *p;
int i, j;
p = a[0];
for(i=0;i<12;i++)
    printf("%5d", p[i]);

```

The base address of the array `a[0]` is assigned to the pointer variable `p`. The pointer variable `p` can be used just like as it is a one-dimensional array and `printf` statement outputs the value of each element of the array. Figure 4.4 shows how two-dimensional array stores in memory.

4000	4002	4004	4006	4008	4010	4012	4014	4016	4018	4020	4022
11	12	13	14	21	22	23	24	31	32	33	34

Figure 4.4: Two-dimensional array stores in memory, such as one-dimensional array

Suppose a two-dimensional array is declared as

```
int a[3][4];
```

An array element `a[i][j]` of the two dimensional array can be accessed through a pointer by following different syntaxes:

```

a[i][j] = *(*(a+i)+j)
a[i][j] = *(a[i]+j)
a[i][j] = (* (a+i))[j]
a[i][j] = *((*a)+(i*4+j))

```

Both addition operations are following the rules of pointer arithmetic, but pointers to different types are involved. The inner arithmetic operation involves with a pointer to an array of size 4. That is why we need to specify the number of columns of the two-dimensional array in function definition when it is used as a formal argument. The outer arithmetic operation involves with a pointer to an integer.

Example:

```

#include<stdio.h>
main()
{
    int a[5]={3, 5, 6, 8};
    printf("%d %d ", a[3], 3[a]);
}
Output:
8 8

```

Do not get surprised! It is a correct syntax:

`3[a]` equals to `* (3+a)`

Array variable always returns the base address, `[]` does sum with array index (i.e. `*(base + index)`) and addition (+) is commutative. Therefore, `*(base + index)` is equal to `*(index + base)`.

Since `3[a]` has represented `* (3+a)` and `* (3+a)` is equal to `* (a+3)`. Therefore, `a[3]` is same as `3[a]` since `a[3]` represents `* (a+3)`.

But on the other hand `[a]3` or `[3]a` is not correct syntax and will result into syntax error, since `(a + 3)*` and `(3 + a)*` are not valid expressions.

Array of Pointers

An array of pointers is nothing more than an array of elements that contain the addresses of values in memory. The format for creating an array of pointers is similar to the format for creating any other array:

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

The array type should match the data type it points to; the indirection operator precedes the name of the array; and the number of elements (pointers) is enclosed in the braces. To assign an address to an element in an array of pointers, use the address operator, as in the following example:

Example:

```
#include<stdio.h>
main()
{
int x,y,z;
int *a[3];
int **p;
x = 10;
y = 20;
z = 30;
a[0] = &x;
a[1] = &y;
a[2] = &z;
printf("x = %d\n", *a[0]);
printf("y = %d\n", *a[1]);
printf("z = %d\n", *a[2]);
p = a;
printf("x = %d\n", *p[0]);
printf("y = %d\n", *p[1]);
printf("z = %d\n", *p[2]);
}
```

Output:

```
x = 10
y = 20
z = 30
x = 10
y = 20
z = 30
```

The address of each integer variable x, y and z is assigned to an element in the array of pointers. The format `*a[0]` indirectly refers to the value of x by using the address stored in `a[0]`.

The base address of the array is assigned to the pointer variable p. The pointer variable p can be used just like as it is an array name and the printf statement outputs the value of the each element of the array.

Pointer to an Array

A two-dimensional array name is a composite pointer that means it is a pointer to arrays, each of the same size as the number of columns. A one-dimensional array can be represented by an elementary pointer variable and a two-dimensional array can be represented by a composite pointer variable that is

a pointer to an array.

The format for creating a pointer to an array:

```
data_type (*pointer_name) [size];
```

where the data_type should match the data type it points to; the size within bracket equals to the number of columns and pointer_name is the name of the pointer.

The objective of declaring a pointer to an array is to ensure that the unit increment of the corresponding pointer always takes from beginning of one row to the beginning of next row, independent of the data type. Remember that pointer to an array is different from the array of pointer.

Example:

```
int a[3][4] = {{11, 12, 13, 14},
                {21, 22, 23, 24},
                {31, 32, 33, 34}};
int (*p)[4];      /* pointer to an array of size 4 */
int i, j;
p = a;
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
        printf("%5d", p[i][j]);
    printf("\n");
}
```

Here, the pointer to an array of size 4 is declared that should be equal to the column size of the two-dimensional array. The base address of the array is assigned to the pointer variable p. The pointer variable p can be used just like as it is an array name and the printf statement outputs the value of the each element of the array.

Pointer to Function

Pointers can point to integer, character, array, pointer as well as pointers can also point to the C function. The functions are loaded into computer memory before they are invoked by the program. Therefore, they have also addresses, from where they are loaded into the memory. When a function address is known then a pointer can point to it. The pointer can provide another way to invoke the function. Note that, the function name itself is an address of the function.

The format for creating a pointer to a function:

```
data_type (*pointer_name)();
```

where data_type same as the return type of the function and pointer_name is the name of the pointer that points to a function.

Example:

```
#include<stdio.h>
int show();
main()
{
int (*p)();
p = show;
printf("Address of Function show is %u\n", show);
```

```
/* Function show is called using pointer */
(*p)();
}
int show()
{
printf("Hello India!");
return 0;
}

Output:
Address of Function show is 4198701
Hello India!
```

Pointers can point to user-defined functions as well as they can point to library functions.

The uses of the pointer to the function are as follows:

- To write memory resident programs.
- To write viruses (i.e. Worms, Trojan horses, etc.) and vaccines (i.e. anti-virus) to remove viruses.

Passing Addresses to Function

A function can pass a value, as well as it can pass addresses to the function i.e. can pass a reference. The addresses of actual arguments in the calling function are copied to formal arguments of the called function. The addresses of the actual arguments are copied to formal arguments as a value, not a reference. In C language functions are called by value and C does not support reference data types. That means the called function uses the values of its arguments as temporary variables rather than the originals. When the temporary variables are modified in the called function, then it has no effect on the actual arguments in the calling function.

It is possible to create a function that can modify a variable in the calling function. Generally, a function can return only one value to the calling function. However, a function can return more than one value at a time to the calling function by passing addresses. The calling function passes the addresses of the variables and the called function must declare the arguments to be a pointer and access the variable indirectly through it.

Example:

```
#include<stdio.h>
void swap(int *a, int *b);
main()
{
int i = 10, j = 20;
printf("Before calling function i = %d, j = %d\n", i, j);
swap(&i, &j);
printf("After calling function i = %d, j = %d\n", i, j);
}
void swap(int *a, int *b)
{
int temp;
temp = *a;
```

```
*a = *b;
*b = temp;
}
```

Output:

```
Before calling function i = 10, j = 20
After calling function i = 20, j = 10
```

Function Returning Pointer

The functions can return an integer, a floating point or any other data type; as well as it can also return a pointer. To create a function returning a pointer should explicitly mention in the function prototype and in the function definition.

The format of function returning pointer:

```
data-type *function-name(argument-list)
```

Example:

```
#include<stdio.h>
#include<stdlib.h>
char *xstrcat(char*,char*);
main()
{
    char *dest, *source = "India!", *target = "Hello ";
    dest=xstrcat(target, source);
    printf("source = %s\n",source);
    printf("target = %s\n",target);
    printf("dest = %s\n",dest);
}
char *xstrcat(char* t,char* s)
{
    int i;
    char *temp;
    temp=(char*)malloc(25);
    for(i=0;*(t+i)!='\0';i++)
        *(temp+i)=*(t+i);
    while(*s)
    {
        *(temp+i)=*s;
        *s++;
        i++;
    }
    return(temp);
}

Output:
source = India!
target = Hello
dest = Hello India!
```

Dynamic Memory Allocation

When memory has been needed by a program, it has been set aside by declaring the desired type of variables. For variables declared in any function, space in memory is set aside to hold the value assigned to the variable. This memory is reserved until the function finishes.

Another problem may arise when declared an array in a program. The user may provide a less number of elements than the number of elements of the declared array, and then the rest of the memory space will be wasted. This leads to the inefficient use of memory. As well as the user may also enter a number of elements more than the number of elements of the declared array, then the user may get unexpected result or user may face a runtime error or system may be crushed.

This may not always be the optimal way to allocate memory. Fortunately, you can instead write your programs and obtain memory as they are running. With dynamic memory allocation, memory is not reserved or set aside at the start of the program; rather, it is allocated on an as-needed basis.

When a program is compiled, many of the memory locations needed for the program to hold variable and constant data can be determined in advance. As the compiler works through the main part of the program and each of the other functions, it can figure out what memory will be needed when the program runs. The memory can be divided into four regions as shown in the figure.

When the program is loaded, it can request the needed memory from the operating system before the program actually begins to run. The operating systems reserve the needed memory locations by stacking one variable on top of another in memory, in a tight, neat block. Because of the way this process works, this part of the memory is known as the stack. Memory reserved within the stack cannot be freed up until the program quits running.

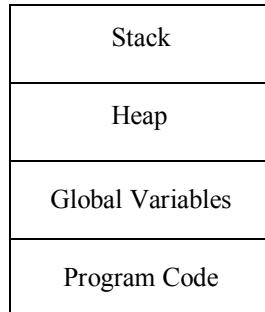


Figure 4.5: Four regions of memory

Some programs need to use a large block of memory to hold data, but they only need those blocks for a short period of time. Rather than use all that memory the entire time the program is running, such programs can temporarily allocate storage locations from another portion of memory, known as the heap. When the program is done using a particular block of heap memory, it simply tells the operating system that it is done, and the system returns that memory to the heap, where it can be freed out to other needy programs. For best utilization of memory, clearly having a heap is a good idea.

A compiled C program creates and uses four logically distinct regions of memory. The first region is the memory that actually holds the program's executable code. The next region is a memory where global variables are stored. The remaining two regions are the stack and the heap. The stack is used for a great many things while your program executes. It holds the return addresses of function calls, arguments to functions, and local variables. It will also save the current state of the CPU. The heap is a region of free memory that your program can use via C's dynamic memory allocation functions.

Although the exact physical layout of each of the four regions of memory differs among CPU types and C implementations, the diagram in Figure shows conceptually how your C programs appear in memory. The important functions involved in dynamic memory allocation are malloc(), calloc(), realloc() and free().

Table 4.3: Functions in dynamic memory allocation

Functions	Description
malloc()	The malloc() function allocates a block of size bytes from the memory heap.
calloc	calloc() allocates a block of size bytes from the memory heap and block is initialized to zero.
realloc()	realloc() function reallocates the memory block that means it attempts to reduce or increase the previously allocated block.
free()	free() function is used to de-allocate a memory block allocated by a previous call to calloc(), malloc(), or realloc().

The malloc Function

The malloc() function is a library function that uses to allocate a block of memory (size in bytes) from the memory heap. This function allows a program to allocate memory dynamically (i.e. in run time) when it is needed and in the exact amounts. The contents of the block are left unchanged.

The general form of the malloc function is

```
void *malloc(size)
```

The malloc() function returns a void pointer to the newly allocated block of memory. When there is not enough space exists for the new block or if the size argument is given zero then malloc function returns NULL.

Example:

```
int *p;
p = (int*) malloc(10 * sizeof(int));
```

In the example, the malloc() function, allocate a memory block for 10 integer elements at runtime. After the memory allocation, it returns a pointer to the memory block. It is possible to guess how much memory is required. However, because the size of an integer varies from system to system, therefore problems can arise when run the code to another platform. The sizeof operator is used to avoid problems. The sizeof operator is used to evaluate the size of each integer element in bytes. The multiplication operator calculates the total size of the memory block. The explicit type conversion is optional.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ a heap memory allocation.

The calloc Function

The calloc() function is used to allocate a block of memory (size in bytes) from the memory heap. This function allows a program to allocate memory dynamically (i.e. at run time) when it is needed and in the exact amounts. The block is cleared to zero

The general form of the calloc() function is

```
void *calloc(nitems, size)
```

Where nitems is the number of item and size represents the size of each item in bytes. The

`calloc()` function allocates a block of memory of size `nitems * size`.

Example:

```
int *p;
p = (int*) calloc(10, sizeof(int));
```

In the example, the `calloc()` function allocates a memory block for 10 integer elements at runtime. After the memory allocation, it returns a pointer to the memory block and the block is initialized to zero. The `sizeof` operator is used to evaluate the size of each integer element in bytes. The explicit type conversion is optional.

The `calloc()` function returns a void pointer to the newly allocated block of memory. When there is not enough space exists for the new block or if the argument size or nitem is zero then the `calloc` function returns NULL.

The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ a heap memory allocation.

The realloc Function

The `realloc()` function reallocates the memory block that means it attempts to reduce or increase the previously allocated block.

The general form of the `calloc()` function is

```
void *realloc(void *p , size)
```

Here, `p` is a pointer to block that is previously allocated and `size` is representing the size of the memory block.

If `size` is zero, the memory block is freed and NULL is returned. The block argument points to a memory block previously obtained by calling `malloc`, `calloc()` or `realloc()`. If the block is a NULL pointer, the `realloc ()` function works just like `malloc`. The `realloc()` function adjusts the size of the allocated block to `size`, copying the contents to a new location if necessary.

The `realloc()` function returns the address of the reallocated block, which can be different than the address of the original block. When the block cannot be reallocated then the `realloc()` function returns NULL. If the value of `size` is zero, the memory block is freed and `realloc()` returns NULL.

The free Function

The `free()` function is used to deallocate a memory block allocated by a previous call to `calloc()`, `malloc()`, or `realloc()`. The general form of the `calloc()` function is

```
void free(void *p);
```

Here, `p` is a pointer to block that is previously allocated. It returns nothing.

Example:

```
int *p;
p = malloc(10 * sizeof(int));
free(p);
```

The `free()` function releases the memory block, which is previously allocated by `malloc()` function.

Example:

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

```

main()
{
    char *s;
    /* allocate memory for string */
    s = (char *) malloc(10);
    if (s == NULL)
    {
        printf("Not enough memory to allocate memory");
        exit(1); /* terminate program if out of memory */
    }
    /* copy "Hello" into string */
    strcpy(s, "Hello");
    /* display string */
    printf("String is %s", s);
    /* free memory */
    free(s);
}

```

Creating One-dimensional Array

Sometimes we can use a pointer variable instead of an array. Suppose, a is declared as a one-dimensional array of integer and p is declared as a pointer to integer.

int a[10];

we can write instead of the array

int *p;

but p is not automatically allocated a block of a memory, therefore before the use of the pointer variable in the place of a array we need to allocate a block of memory for it.

The one-dimensional array can be created by using malloc () or calloc () function.

Example:

```

#include<stdio.h>
#include<stdlib.h>
main()
{
    int *p, n, i;
    printf("Enter the number of elements");
    scanf("%d", &n);
    p =(int*)malloc(n * sizeof(int));
    if(p==NULL)
    {
        printf("Insufficient Memory");
        exit(1);
    }
    printf("Enter the elements of the array");
    for(i=0;i<n;i++)
        scanf("%d", p+i);
    printf("Elements of the array");
}

```

```

for(i=0;i<n;i++)
    printf("%5d", *(p+i));
}

Output:

Enter the number of elements 5
Enter the elements of the array 3 5 6 7 9
Elements of the array      3      5      6      7      9

```

Creating Two-dimensional Array

There are two methods of creating two-dimensional arrays dynamically.

In the first method, the number of columns of the two-dimensional array has been known before compile time, but the number of rows will be known only at the execution time. The two-dimensional array is allocated dynamically which is represented by using a pointer to array having the same size of a number of columns of the two-dimensional array.

Example:

```

#include<stdio.h>
#include<stdlib.h>
main()
{
int (*p)[4], m, i, j;
printf("Enter number of rows: ");
scanf("%d", &m);
p = (int (*)[4])malloc(m*4*sizeof(int));
if(p==NULL)
{
    printf("Insufficient Memory");
    exit(1);
}
printf("Enter the elements of the array\n");
for(i=0;i<m;i++)
    for(j=0;j<4;j++)
        scanf("%d", &p[i][j]);
printf("Elements of the array\n");
for(i=0;i<m;i++)
{
    for(j=0;j<4;j++)
        printf("%5d", p[i][j]);
    printf("\n");
}
}

Output:

Enter numner of rows: 3
Enter the elements of the array
11 12 13 14

```

```

21 22 23 24
31 32 33 34
Elements of the array
 11   12   13   14
 21   22   23   24
 31   32   33   34

```

Note that p may now be used as an array of size 4.

In the second method, the numbers of rows, as well as a number of columns both, are unknown at compile time they will be known at the execution time. The following steps are used to create a two-dimensional array dynamically:

- i) At first, an array of pointers is to be created dynamically. This array of pointers will have the same size as the number of rows of the two-dimensional array. The i^{th} array element should point to the beginning of the i^{th} row.
- ii) Allocate memory space for the entire array dynamically. The total space may be contiguous or may not be so. However, the minimum condition is that all the elements of each row must be allocated space contiguously.
- iii) The value of each element of the pointer array that created in the first step is to be properly initialized.

Contiguous Allocation of Two-dimensional Array

In the contiguous allocation of memory space for the two-dimensional array, the whole array is allocated dynamically. The total memory space of two-dimensional array should be contiguous. An array of pointers is also to be created dynamically. This array of pointers will have the same size as the number of rows of the two-dimensional array and the i^{th} array element should point to the beginning of the i^{th} row.

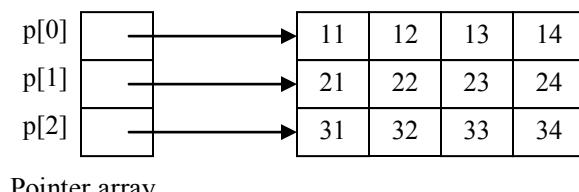


Figure 4.6: Contiguous allocation of entire 2D array

Note that the array of pointers can be represented by pointer to pointer.

Example:

```

#include<stdio.h>
#include<stdlib.h>
main()
{
int **p, *q, m, n, i, j;
printf("Enter number of rows");
scanf("%d", &m);
printf("Enter number of columns");

```

```

scanf("%d", &n);
p =(int**)malloc(m * sizeof(int *));
q =(int*)malloc(m * n * sizeof(int));
if(p==NULL || q==NULL)
{
    printf("Insufficient Memory");
    exit(1);
}
for(i=0; i<m; i++)
{
    p[i] = q;
    q += n;
}
printf("Enter the elements of the array\n");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        scanf("%d", &p[i][j]);
printf("Elements of the array\n");
for(i=0; i<m; i++)
{
    for(j=0; j<n; j++)
        printf("%5d",p[i][j]);
    printf("\n");
}
}

```

Output:

```

Enter number of rows3
Enter number of columns3
Enter the elements of the array
11 12 13
21 22 23
31 32 33
Elements of the array
11    12    13
21    22    23
31    32    33

```

Non-contiguous Allocation of Two-dimensional Array

Non-contiguous allocation of two-dimensional array is used when we need to create a large two-dimensional array (for example 1000 x 1000) and/or contiguous memory may not available. For non-contiguous allocation of memory, space for each row is to be allocated separately as the need arises. Therefore, the total space is not being contiguous. However, all the elements of each row must be allocated space contiguously.

An array of pointers is created dynamically. This array of pointers will have the same size as the number of rows of the two dimensional array and the i^{th} array element should point to the beginning of the i^{th} row.

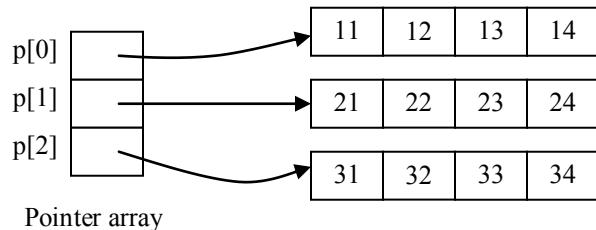


Figure 4.7: Non-contiguous allocation of 2D array, but each row must be allocated space contiguously

Example:

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int **p, m, n, i, j;
printf("Enter number of rows");
scanf("%d", &m);
printf("Enter number of columns");
scanf("%d", &n);
p =(int**) malloc(m * sizeof(int *));
for(i=0; i<m; i++)
    p[i] = (int*)malloc(n * sizeof(int));
printf("Enter the elements of the array");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        scanf("%d", &p[i][j]);
printf("Elements of the array");
for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        printf("%5d", p[i][j]);
    printf("\n");
}
}
```

Output:

```
Enter number of rows3
Enter number of columns3
Enter the elements of the array
11 12 13
21 22 23
31 32 33
Elements of the array
11    12    13
21    22    23
31    32    33
```

However, the drawbacks of dynamic memory allocation for the two-dimensional array are to be followed:

- Extra memory space is required for storage of pointer array.
- Access to the individual array element is obtained only after the first level of indirect addressing.

Pointers, Arrays and Strings

Sometimes we can use a pointer variable instead of an array. Suppose, a is declared as a one-dimensional array of integer and p is declared as a pointer to an integer.

```
int a[10];
we can write instead of the array
int *p;
```

Since p is not defined so p may contain a garbage value and p is not automatically allocated a block of a memory. Therefore, before the use of the pointer variable in the place of an array we need to allocate a memory for it.

Example:

```
p = (int *)malloc(100*sizeof(int));
This statement allocates a memory block for 100 integer values.
```

Example:

```
char *s;
scanf("%s", s);           /* invalid */
or
char *s;
s = "Hello";              /* invalid */
```

We should require allocating a block of memory before the use of pointer variable. Otherwise, it may display a runtime error like: **-Segmentation fault**.

Example:

```
char *s;
s = (char*)malloc(10);
scanf("%s", s);
```

However in the case of array, memory block is allocated at compile time.

```
char t[10];
scanf("%s", t);           /* valid */
but we cannot assign a value to a string variable
```

```
char t[10];
t = "Hello";
we should use the library function strcpy() to do this
```

Example:

```
char t[10];
strcpy(t, "Hello");
```

Program: How to find size of int data type without using sizeof operator.

```
#include<stdio.h>
main()
{
    int *ptr = 0;
    ptr++;
    printf("Size of int data type: %d",ptr);
}
```

Summary

- A pointer is a variable that can point to another variable.
- That variable may be an integer type, character type or floating-point type.
- A pointer can also point to another pointer variable then the pointer is known as pointer-to-pointer.
- The void pointer is a pointer, which may store the address of any type of variable.
- Sometimes we need to allocate memory space when a program is running. This type of memory allocation is called dynamic memory allocation.
- The malloc() and calloc() functions are used to allocate memory space at runtime.
- The realloc() function is required to modify the previously allocated memory space.
- The free() function is used to release the memory space that allocated previously call of malloc(), calloc() or realloc() function.

Exercises

1. What is a pointer?
2. What is the purpose of indirection operator? How can the indirection operator be used to access a multidimensional array element?
3. What is pointer to pointer?
4. What is a void pointer? What is a generic function?
5. What is a dangling pointer?
6. How do you get access to an element in an array by using a pointer? Explain with suitable example.
7. How pointer to function works?
8. How can a function return a pointer to its calling function? Explain with a suitable example.
9. What is dynamic memory allocation? What is the advantage of dynamic memory allocation over static memory allocation?
10. What are the differences between malloc() and calloc() function?
11. Write a program using pointers to compute the sum of all elements stored in an array.
12. Write a program using pointers to determine the length of a character string.
13. Write a function using pointers to exchange the values stored in two locations in the memory.
14. Write a program to generate an array of N elements dynamically and sort them in ascending order.
15. Write a program of matrix multiplication by using dynamic memory allocation.
16. What is the output of the following C program?
 - (i) main ()
 {
 }

```

    const int x = 5;
    int *prt;
    ptr = &x;
    *ptr = 10;
    printf("%d", x);
}
(ii) main()
{
    int a=2, *f1, *f2;
    f1 = f2 = &a;
    *f1+=*f2+=a+=2.5;
    printf("%d %d %d", a, *f1, *f2);
}
(iii) main()
{
    register int i =5;
    printf("Address of a=%u", &i);
    printf("Value of a=%d", i);
}
(iv) main()
{
    int a=10;
    void *j;
    j=&a;
    j++;
    printf("%u", j);
}
(v) main()
{
    char *p="Hello";
    printf("%c", *p);
}
(vi) main()
{
    int i=320;
    char *ptr=(char *)&i;
    printf("%d", *ptr);
}
(vii) main()
{
    static char str[]="Limericks";
    char *s;
    s=&str[6]-6;
    while(*s)
    printf("%c", *s++);
}

```

CHAPTER 5

LINKED LIST

“The whole is equal to the sum of its parts”. - Euclid

Link list, as the name suggests, is a linear list of linked elements. Like arrays, linked list represents another linear data structure. Arrays are very commonly useful data structure in most of the programming languages. Since it has several limitations and drawbacks; therefore, an alternative approach is required. These limitations can be overcome by using Linked List data structure.

Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation developed linked lists as the primary data structure in 1955–1956 for their Information Processing Language.

KEY FEATURES

-  Singly Linked List
-  Doubly Linked List
-  Circular Linked List
-  Linked Stack
-  Linked Queue
-  Polynomial Representation

Limitations of Array

The array is the most common data structure used to store a collection of homogeneous elements. In most languages, arrays are convenient to declare, and fast to access any element in a constant amount of time. The address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

In many applications, the array is not suitable as it has some drawback. The drawbacks of Array are listed below:

- The maximum size of the array needs to be predicted beforehand. One cannot change the size of the array after allocating memory, but, many applications require resizing. Most often this size is specified at compile time with a simple declaration. The size of the array can be deferred until the array is created at runtime, but after that it remains fixed. When arrays are allocating dynamically from the heap and then one can dynamically resize it with realloc(), but that requires some real programmer effort.
- Most of the space in the array is wasted when programmer allocates arrays with large size. On the other hand, when program ever needs to process more than the specify size then the code breaks.
- Storage of the array must be available contiguously. Required storage not always immediately available.
- Insertion and deletion operation may be very slow. The worst case occurs when the first element is to be deleted or inserted. Almost all the elements of the array need to be moved. On an average about half the elements of the array need to be moved. Thus, the time complexity depends on the total no of elements rather than the actual operation.
- Joining and splitting of two or more arrays is difficult.

LINKED LIST

In arrays, there is always a fixed relationship between the addresses of two consecutive elements as all the items of an array must be stored contiguously. However, note that this contiguity requirement makes expansion or contraction of the size of the array difficult. In linked list, data items may be scattered arbitrarily all over the memory, but we can achieve fast insertion and deletion in a dynamic situation.

Definition: A linked list is a linear ordered collection of finite homogeneous data elements called node, where the linear order is maintained by means of links or pointers.

A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list structure is created by the use of pointers to connect all its nodes together like the links in a chain.

In an array if the address of one element is known, addresses of all other elements become automatically known. Since, in a linked list, there is no relationship between the addresses of elements, each element of a linked list must store explicitly the address of the element next to it.

Table 5.1: Difference between Array and Linked list

Property	Array	Linked List
Storage	Storage of the array must be available contiguously.	Storage need not be contiguous.
Memory utilization	The size of the array needs to be predicted beforehand because memory allocation is done in advance.	Memory of linked list is not pre-allocated, memory is allocated whenever it is required.
Memory utilization	Not necessary for storing addresses of any element.	Extra memory space is necessary for storing addresses of the next node
Change of size	The array size is fixed, extend or shrink not possible during the execution of a program	Linked list may extend or shrink during the execution of a program
Insertion/deletion	Insertion/deletion operations are slow, half of the elements are required to move on an average	Insertion/deletion operations are performed very fast, in a constant amount of time
Searching	Linear searching, binary searching, interpolation searching are possible	Binary searching, interpolation searching not possible, only linear searching is possible
Access element	Fast access to any element in a constant amount of time.	To access any element in a linked list, traversing is required.
Joining/splitting	Joining and splitting of the two arrays is difficult.	Joining and splitting of two linked list is very easy.

Advantages of linked list

Linked lists have many advantages. Some of the very important advantages are:

- Linked list are dynamic data structures. That is, they can extend or shrink during the execution of a program.
- Storage need not be contiguous.
- Efficient memory utilization. Here memory is not pre-allocated. Memory is allocated whenever it is required.
- Insertion or deletion is easy and efficient, may be done very fast, in a constant amount of times,

independent of the size of the list.

- v) The joining of two linked lists can be done by assigning pointer of the second linked list in the last node of the first linked list.

Splitting can be done by assigning a null address in the node from where we want to split one linked list into two parts.

Types of Linked list

There are different types of linked list. We can put linked lists into the following four types:

- i) Singly linked list
- ii) Circular Linked list
- iii) Doubly Linked list
- iv) Circular Doubly linked list

Singly linked list

An element in a linked list is specially termed as a node. In a singly linked list, each node consists of two fields:

- i) DATA field that contains the actual information of the element.
- ii) LINK field, contains the address of the next node in the list.

A "DATA" field to store whatever element type the list holds for the user, and a "LINK" field, which is a pointer, used to link one node to the next node. Each node is allocated in the heap with a call to malloc() function. The node memory becomes free when it is explicitly de-allocated with a call to free() function. The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like.

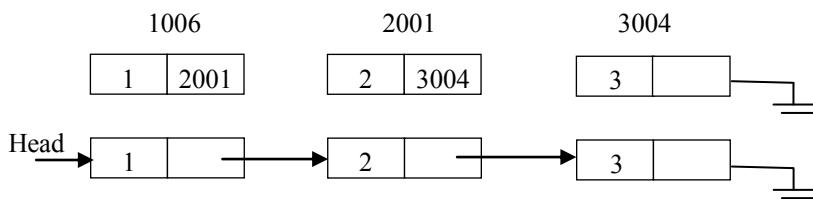


Figure 5.1: Linked list representation

Operations on Singly linked list

Operations supported by a singly linked list are as follows:

Table 5.2: Various Operation on Linked list

Operation	Description
CreateList	This operation creates a linked list.
Traverse	This operation traverse/visit all the elements of the linked list exactly once
Insertion	This operation inserts an element to the linked list
Deletion	This operation removes an element from the linked list
Searching	This operation performs linear searching for a key value in the linked list
Reverse	This operation performs the reverse of the linked list
Merging	This operation performs merging of two linked lists in a single linked list

Before going to the detail operation on singly linked list, we need two data types: Node and Node pointer.

A linked list is constructed by the nodes. These nodes are allocated in the heap. Each node contains a single data element and a pointer to the next node in the list.

```
struct node
{
    int DATA;
    struct node *LINK;
};
```

The Node structure of a linked list is shown here. Where DATA contains value and LINK holds the address of the next node.



Figure 5.2: Symbolic representation of a node

The LINK field of a node in a linked list points to the next node of list. LINK pointer is termed as self-referential pointer as it points to the address of a node of the same type.

Create a Singly Linked List

The following algorithm creates a node and appends it at the end of the existing list. HEAD is a pointer which holds the address of the HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node and Temp is a temporary pointer.

Algorithm to create a singly Linked List

Algorithm: CREATE (HEAD, ITEM)

1. [Create NEW node]
 - a) Allocate memory for NEW node.
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = NULL
2. [Whether List is empty, head is the content of HEADER]
 - If HEAD = NULL then Set HEAD = NEW
3. Else
 - a) Set Temp = HEAD
 - b) While Temp→LINK ≠ NULL do
 - Set Temp = Temp→LINK

[End of while]
 - c) Set Temp→LINK = NEW

[End of IF]
4. Return

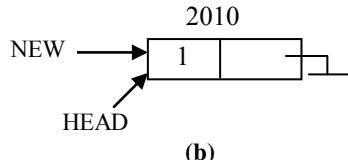
At first creates a NEW node from the heap using dynamic memory allocation and checks whether it is NULL or not. If the NEW node is NULL, then memory is not available for creating the linked list. Otherwise, stores the value of ITEM to the DATA part of the NEW node and stores NULL to the LINK part of the NEW node.

The following diagrams explain the creation of a singly linked list.
At first HEAD is assigned with NULL value



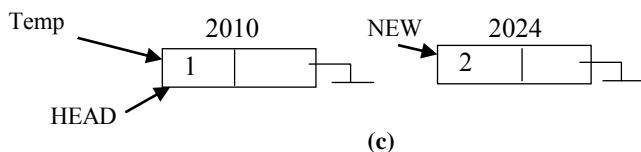
(a)

After that a new node is created and the address of this node is assigned to HEAD,

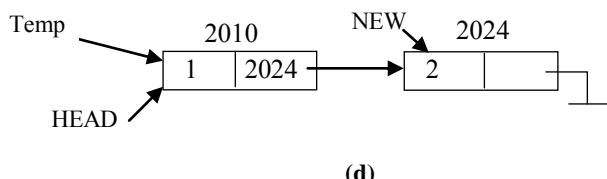


(b)

In the next step another node is created and linked to HEAD node.



(c)



(d)

Figure 5.3 (a-d): Create a Singly Linked List

Traversing / Display a Linked List

This algorithm traverses a linked list and prints the data part of each node of the linked list. The HEAD is a pointer which points to the starting node of the linked list and Temp is a temporary pointer to traverse the list.

Algorithm to traverse a Linked List

Algorithm: TRAVERSE (HEAD)

1. If HEAD = NULL then
 - i) Print: "The linked list is empty"
 - ii) Return
2. Temp = HEAD
3. Repeat while Temp ≠ NULL
 - i) Print: Temp→DATA
 - ii) Set Temp = Temp→LINK

[End of Loop]
4. Return

Insertion in Singly Linked List

Insertion operation in a singly linked list can be done in different ways using position.

- Insertion at beginning.
- Insertion in the middle.
- Insertion at end.

Insertion of a node at first position of a singly linked list

In the following algorithm insertion of node at beginning position is described. The HEAD is a pointer which points to the starting node of the linked list. NEW points to the new node.

Algorithm to insert a node at the beginning

Algorithm:ADD_BEG (HEAD, ITEM)

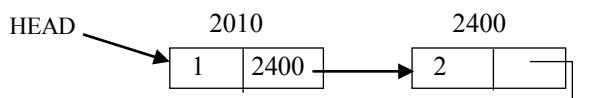
1. [Create the new node]
 - a) Allocate memory for NEW node.
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = HEAD
2. [Make the HEADER to point to the NEW node]

Set HEAD = NEW
3. Return

At first creates a NEW node from the heap using dynamic memory allocation and checks whether it is NULL or not. If the NEW node is NULL, then memory is not available. Otherwise, stores the value of ITEM of the DATA part of the NEW node.

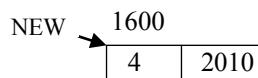
The following diagrams explain the insertion operation at the beginning of a singly linked list.

At first HEAD points to the first node of the list containing two nodes.



(a)

A new node is created and pointed by pointer NEW. LINK field of new node contains the address of head node.



(b)

Now, HEAD pointer points to the new node

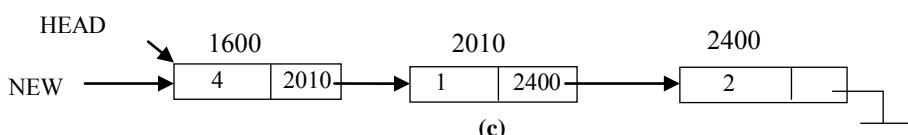


Figure 5.4 (a-c): Node insertion in a single linked list at beginning

Insertion of a node before a specified node of a singly linked list

This algorithm creates a node and inserts it before the node pointer P. The HEAD is a pointer that points to the first node of the linked list and ITEM is the value of the new node. NEW is a pointer that holds the address of the new node. Temp and PTemp are two temporary pointers to traverse the list.

Algorithm to insert a node before a given node pointer

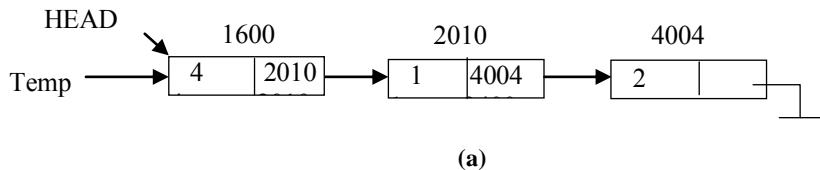
Algorithm: ADD BEFORE (HEAD, ITEM, P)

1. Set Temp = HEAD [to make temp to point to the first node]
2. Repeat step 3 while Temp ≠ P
3. a) Set PTemp = Temp
- b) Set Temp = Temp→LINK
- c) If Temp = NULL then
 - i) Print: "Not Found"
 - ii) Return
 [End of loop]
4. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = Temp
 - e) Set PTemp→LINK = NEW
5. Return

At the beginning traversing the linked list from HEAD to node pointer P to get the location of the previous node. Then, create a NEW node from the heap using dynamic memory allocation and checks whether is NULL or not. If the NEW node is NULL, then memory is not available. Otherwise, stores the value of ITEM of the DATA part of the NEW node.

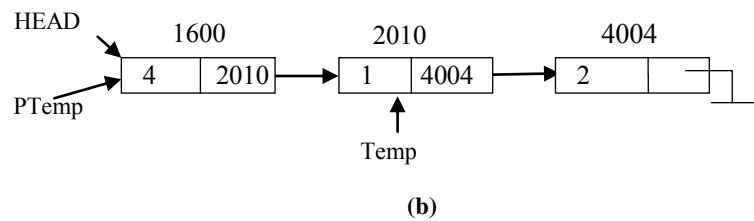
The following diagrams explain the insertion operation before a node pointer of a singly linked list.

At first, Temp is assigned to HEAD.



(a)

In the next step, Temp is moved to the next node and PTemp is assigned to the address of the previous node



(b)

After that a NEW node is created and linked with previous and next node.

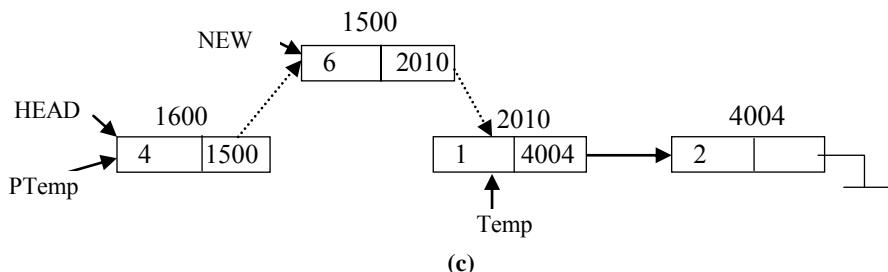


Figure 5.5 (a-c): Node insertion before any position in a linked list

Insertion of a node after a node of a singly linked list

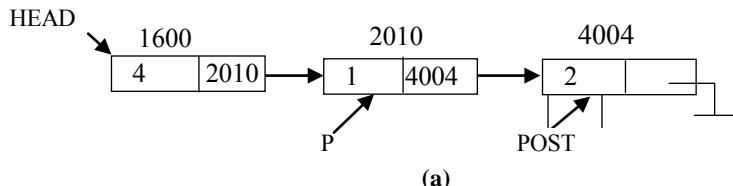
In the following algorithm, it is described how to insert a node after a specific node pointer P in a linked list. The HEAD is a pointer which points to the first node of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. POST is a temporary pointer that points to the next node of P.

Algorithm to insert a node after a given node pointer

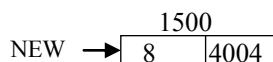
Algorithm: ADD_AFTER (HEAD, ITEM, P)

1. Set **POST = P→LINK**
2. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = POST
 - e) Set P→LINK = NEW
3. Return

At the beginning, set POST pointer by the link part of P pointer. Now, POST points to the next node of P. Then creates a NEW node and stores the value of ITEM of the DATA part of the NEW node. The following diagrams explain the insertion operation after a node pointer of a singly linked list. In the figure 5.6a, POST holds the address of the node after a specific node P.



After that a NEW node is created, ITEM is placed in DATA field and POST is assigned to the LINK field which is shown in figure 5.6b.



(b)

At last, NEW node is assigned to LINK of node P

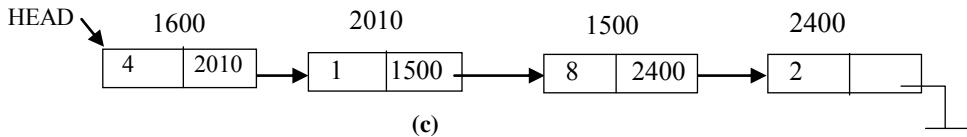


Figure 5.6 (a, b, c): Node insertion after any position in a linked list

Insertion of a node at the end of a singly linked list

The following algorithm describes how the new node is inserted at the end of a singly linked list. The HEAD is a pointer which points to the first node of the linked list and NEW is a pointer which holds the address of the new node. ITEM is the value of the new node. Temp holds the address of header node.

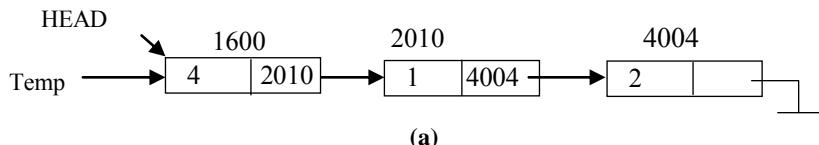
Algorithm to insert a node at end

Algorithm: ADD-END (HEAD, ITEM)

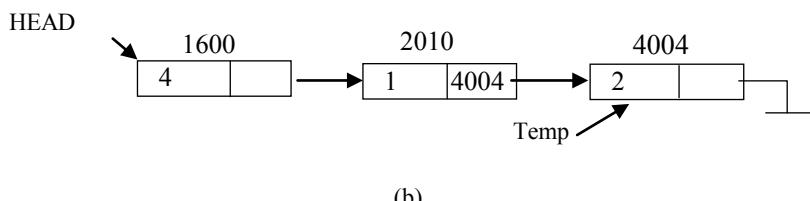
1. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = NULL
2. Set Temp = HEAD [to make Temp to point to the first node]
3. Repeat while Temp→LINK ≠ NULL
 - Set Temp = Temp→LINK
 - [End of loop]
4. Set Temp→LINK = NEW
5. Return

At the beginning creates a NEW node and stores the value of ITEM of the DATA part of the NEW node. Then, traversing the linked list from HEAD to the last node to get the location of the last node. Then The following diagrams explain the insertion operation at the end of a singly linked list.

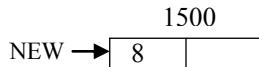
At first Temp is assigned to the address of the HEAD node of the linked list



Now, Temp moves to the end of the linked list.

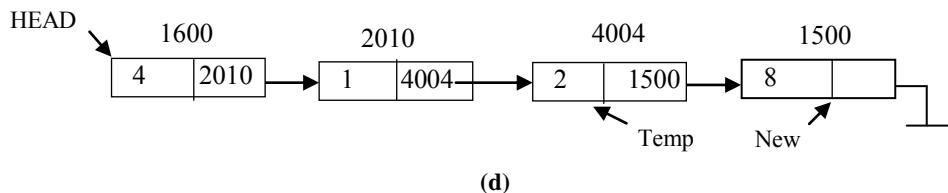


A NEW node is created ITEM is placed in DATA field and NULL is assigned to LINK field.



(c)

At last, NEW is assigned to the LINK field of the last node of the linked list.



(d)

Figure 5.7 (a, b, c, d): Node insertion at last position in a linked list

Deletion from a singly linked list

Deletion operation in a singly linked list can be done in different ways using position.

- Deletion from beginning.
- Deletion in the middle.
- Deletion from end.

Deleting a node from the beginning

In the following algorithm deletion of head node of a linked list is described. Temp is a temporary pointer holds the address of a header node (HEAD). ITEM variable is used to store the value of the deleted node.

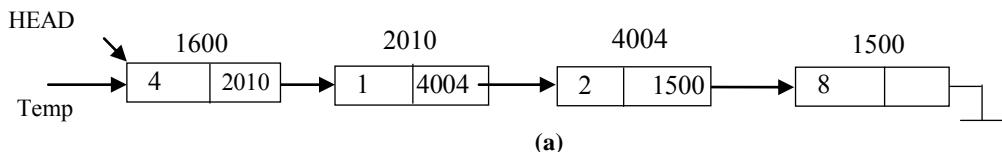
Algorithm to delete a node from the beginning

Algorithm: DELETE_BEG (HEAD, ITEM)

1. [Check for empty list]
IF HEAD = NULL then Print: "The linked list is empty" and Return
2. Set Temp = HEAD [To make Temp to point the first node]
3. Set HEAD = Temp→LINK
4. Set ITEM = Temp→DATA
5. Set Temp→LINK = NULL
6. Deallocate memory for Temp Node
7. Return

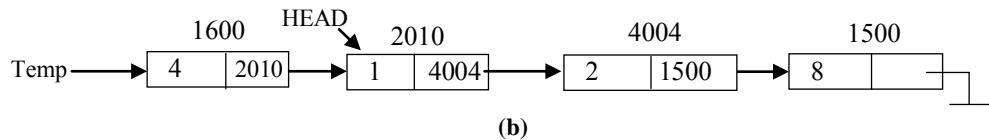
At the beginning checks whether the linked list is empty or not. Temp pointer points to the first node of the linked list, then HEAD moves to the next node and stores the DATA part of the Temp node to ITEM. The following diagrams explain the deletion operation from a singly linked list.

In the first step Temp is assigned to the address of the HEAD.



(a)

In the next step HEAD moves to the next node.



Finally, the first node of the linked list is deleted which is pointed by Temp.

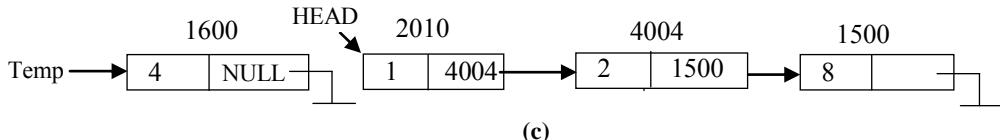


Figure 5.8 (a, b, c): Node deletion from first position of a linked list

Deletion from a singly linked list from end position

Deletion of a node from the end of a singly linked list is described in the following algorithm. Temp is a temporary pointer points to HEAD node. Temp pointer is used to traverse the linked list to keep HEAD pointer in its position. PTemp is an another temporary pointer that holds the address of the previous node of the node to be deleted.

Algorithm to delete a node from the end

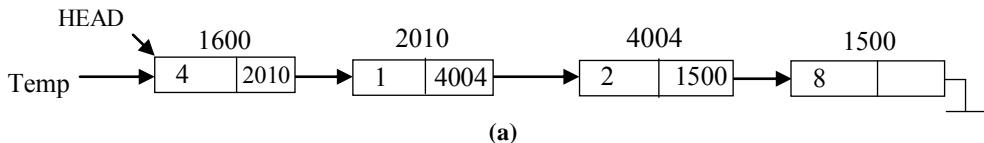
Algorithm: DELETE_END (HEAD, ITEM)

1. [Check for empty list]
IF HEAD = NULL then Print: "The linked list is empty" and Return
2. Set Temp = HEAD
3. Repeat while Temp→LINK ≠ NULL
 - a) PTemp = Temp
 - b) Set Temp = Temp→LINK
 [End of loop]
4. Set ITEM = Temp→DATA
5. Set PTemp→LINK = NULL
6. Deallocate memory for Temp Node
7. Return

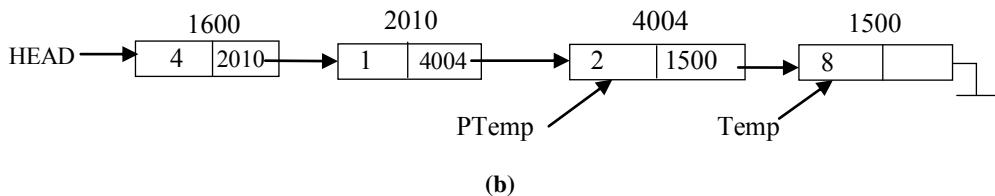
At the beginning checks whether the linked list is empty or not. Then, traversing the linked list from HEAD to the last node to get the location of the last node and the second last node. Stores the DATA part of the Temp node to ITEM.

The following diagrams explain the deletion operation from a singly linked list.

At first Temp is assigned with HEAD.



Temp is moved to the end of the list PTemp holds the address of the previous node.



At last, PTemp LINK field is assigned to NULL and Temp is deleted.

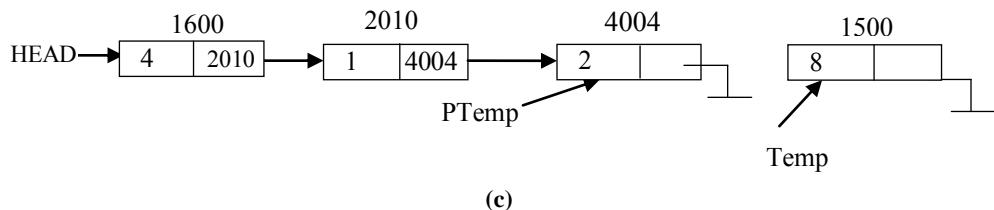


Figure 5.9 (a, b, c): Node deletion from end position of a linked list

Deletion from a singly linked list after any intermediate node

In the following algorithm the deletion of a node from a specific position is described. P holds the address of the previous node of Temp node which has to be deleted.

Algorithm to delete a node after a given node pointer

```
Algorithm: DELETE_AFTER (ITEM, P)
1. Set Temp = P→LINK
2. Set P→LINK = Temp→LINK
3. Set ITEM = Temp→DATA
4. Set Temp→LINK = NULL
5. Deallocate memory for Temp Node
6. Return
```

At the beginning stores the DATA part of the Temp node to ITEM. P points to the previous node of the Temp node that has to be deleted. LINK part of P node points to the next node of Temp node. LINK of Temp is set to NULL. Then, Temp is de-allocated.

Deletion from a singly linked list with a given ITEM

This algorithm finds and deletes a node whose value is taken as ITEM. Temp is a pointer which holds the address of HEADER of the linked list. Temp and old are two temporary pointers to traverse the list.

Algorithm to delete a node by value

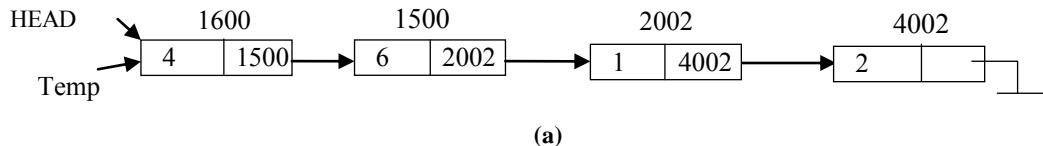
```
Algorithm: DELETE_ITEM (HEAD, ITEM)
1. [Make temp to point the first node]
   Set Temp = HEAD
2. [Check for empty list]
```

```

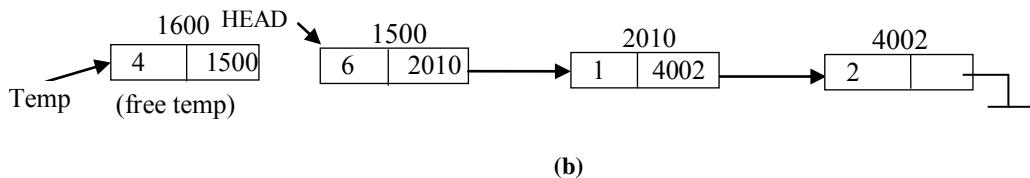
IF HEAD = NULL then
    a) Print: "The linked list is empty"
    b) Return
3. Repeat Step 4 to 5 until Temp is NULL
4. If Temp→DATA = ITEM then
    a) If Temp = HEAD then // node to be deleted is the first node
        Set HEAD = Temp→LINK
    Else
        Set PTemp→LINK = Temp→LINK
    b) Deallocate memory for Temp Node// de allocate node
    c) Return
5. Else
    Set PTemp = Temp and Temp = Temp→LINK
    [End of loop]
6. Print: "Element not found"
7. Return

```

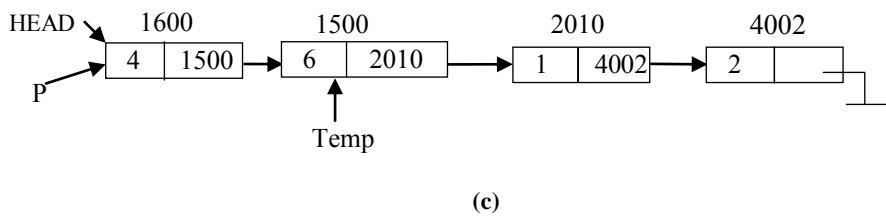
At first, Temp points to the first node of the list which is shown in the following diagram.



Then it is checked whether the list is empty or not, if not empty then the first node is deleted as shown in the figure 5.10b.



To delete any other node Temp is assigned to that node and P holds the address of the previous node.



Node of the node to be deleted. Then, LINK of P is assigned with LINK of Temp. After that Temp is deleted.

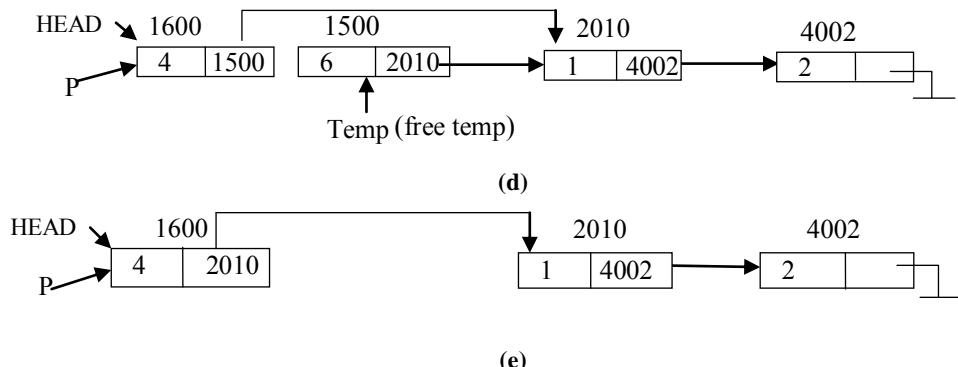


Figure 5.10 (a-e): Node deletion at any position in a linked list

Searching a singly linked list

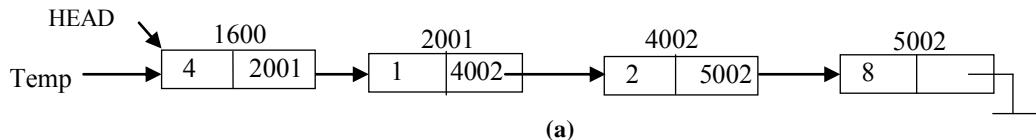
This algorithm finds the location of a node in a linked list whose value is ITEM. Temp is a pointer which points to the starting node of the linked list. *_Temp* is a temporary pointer to traverse the list. *_LOC* is the variable to store the location of the search item.

Algorithm to search Linked List by given value

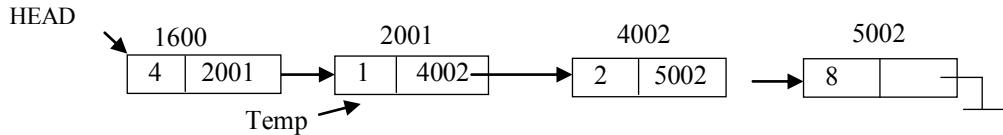
Algorithm: SEARCH(HEAD, ITEM, LOC)

1. Set Temp = HEAD, LOC=NULL
 2. If Temp = NULL then
 - i) "The linked list is empty"
 - ii) Return
 3. Repeat step 4 while Temp ≠ NULL
 4. If Temp→DATA = ITEM then
 - i) Print: "Element found"
 - ii) Set LOC = Temp
 - iii) Return
 - Else
 - Temp = Temp→LINK
- [End of If]
- [End of Loop]
5. Print: "Element not found"
 6. Return

At first Temp is assigned to the address of HEAD

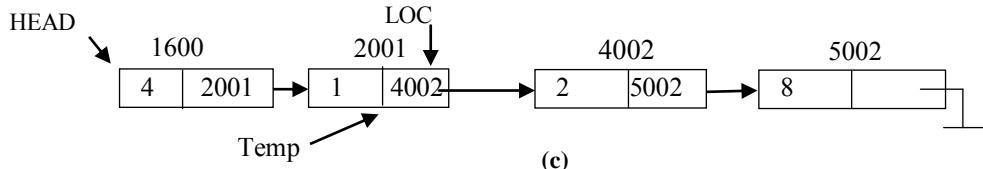


After that Temp→DATA is compared with ITEM=1 and Temp moves accordingly



(b)

When the ITEM is found LOC points to the index of that node in the list.



(c)

Figure 5.11(a, b, c): Searching of an ITEM in linked list

Reverse of a Linked List

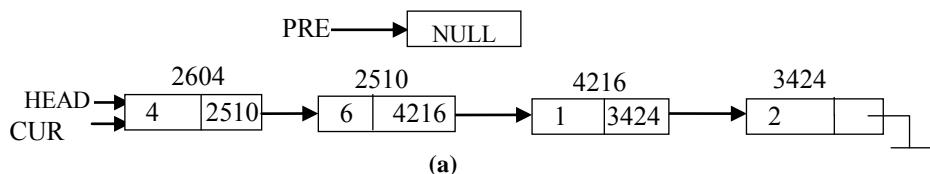
Reversing of linked list means that the last node becomes the first node and first becomes the last. HEAD is a pointer which holds the address of HEADER of the linked list. PRE, POST and CUR are temporary pointers.

Algorithm to reverse Linked List

Algorithm: REVERSE (HEAD)

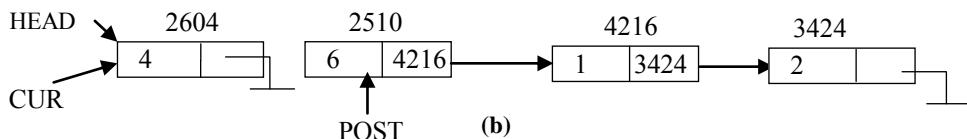
1. Set PRE = NULL and CUR = HEAD
2. Repeat step 3 to 6 while CUR ≠ NULL
3. Set POST = CUR → LINK
4. Set CUR → LINK = PRE
5. Set PRE = CUR
6. Set CUR = POST
[End of Loop]
7. HEAD = PRE
8. Return

In the first step NULL is assigned to PRE pointer and CUR pointer is assigned to the HEAD node of the linked list.



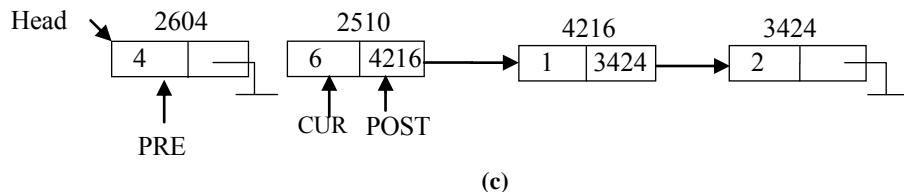
(a)

After first iteration POST points to the next node of the current node



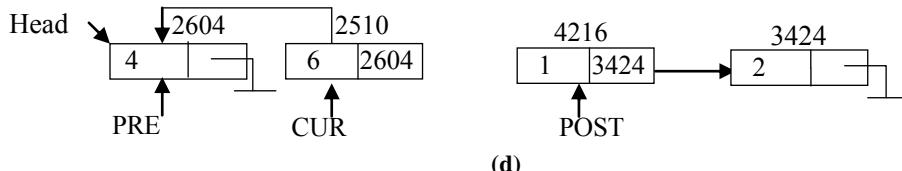
(b)

In the next step CUR is assigned to PRE and POST is assigned to CUR. LINK field of the PRE node is assigned to NULL.



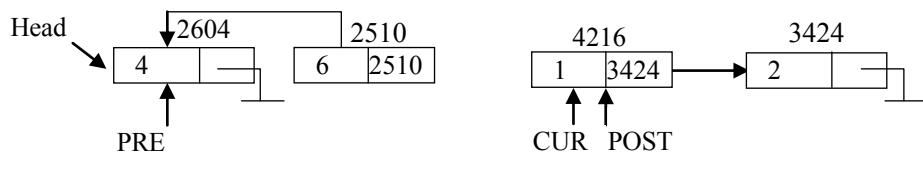
(c)

After a second iteration LINK field of CUR node is linked with PRE and POST moves to the next node.



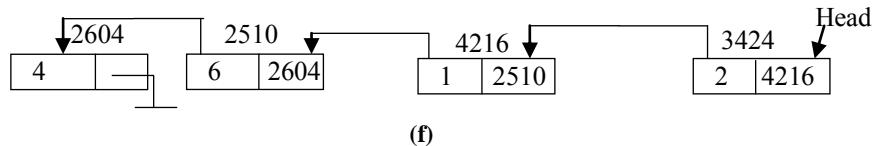
(d)

After that CUR is assigned with POST.



(e)

After 4th iteration the single linked list look like the following figure 5.11f



(f)

Figure 5.12 (a-f): Reverse of a linked list

Reverse Display of a Linked List

In this algorithm the linked list is displayed in reverse order. HEAD points to the first node of the list. P is used to forward the loop to the end of the list. Then R is set to the last node of the list. After that R is used for displaying the list in reverse order.

Algorithm to reverse display the Linked List

Algorithm: REVERSE_DISPLAY (HEAD)

1. Set R = NULL and Q = HEAD
2. Repeat step 3 to 7 while Q ≠ R
3. Set P = Q
4. Repeat step 5 while P → LINK ≠ R
5. Set P = P → LINK

```

        [End of Loop]
6.    Print: P→INFO
7.    Set R =P
    [End of Loop]
8. Return

```

Convert Array to Singly Linked List

We can convert a singly linked list using an existing array. The following algorithm creates a node, stores an array element and appends it at the beginning of the existing list.

This algorithm creates a node and inserts it at the beginning of the list. HEAD is a pointer which holds the address of the HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node.

Algorithm to convert Linked-List from an existing Array

Algorithm: CONVERT (A, N, HEAD)

```

1. Set HEAD = NULL
2. Repeat steps 3 to 8 while N ≠ -1
3.     Allocated memory for NEW node
4.     IF NEW = NULL then Print: "Memory not Available" and Return
5.     Set NEW→DATA = A[N-1]
6.     Set NEW→LINK = HEAD
7.     Set HEAD = NEW
8.     Set N = N - 1
    [End of loop]
9. Return

```

Representation of polynomials using linked list

In this representation, the polynomial can also be represented by linking list and a node in the linked list represents a term in the polynomial.

At first, we have to store exponent/degree and coefficient of the polynomial $f(x)$. For that, we have to define a structure as follows:

```

typedef struct poly
{
    int exp;
    int coef;
    struct poly *LINK;
} term;
poly *head;

```

We will always store terms of the polynomial in descending order of degree.

$4x^6 + 2x^4 + 3x + 1$ would be represented by linked list like:

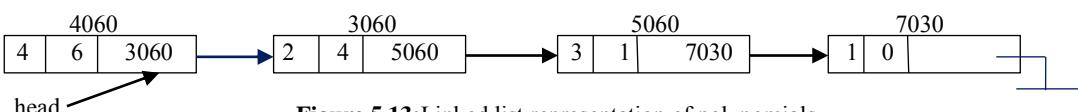


Figure 5.13: Linked list representation of polynomials

The addition of two Polynomial using Linked List

In the following algorithm addition of two polynomials are described. Two linked lists consisting of two polynomials are headed by HEAD1 and HEAD2 respectively. Insert function is used to insert a node in a new linked list which is headed by R.

Algorithm to addition of two polynomials

Algorithm: POLYADD (HEAD1, HEAD2)

1. Set P=HEAD1, Q=HEAD2, R=NULL
2. Repeat while P=NULL and Q ≠NULL
 - i) If P→EXP > Q→EXP then
 - Call INSERT(R, P→EXP, P→COEF)
 - Set P=P→LINK
 - ii) Else If P→EXP < Q→EXP then
 - Call INSERT(R, Q→EXP, Q→COEF)
 - Set Q=Q→LINK
 - iii) Else If P→COEF + Q→COEF ≠ 0
 - Call INSERT(R, P→EXP, P→COEF + Q→COEF)
 - Set P=P→LINK
 - Set Q=Q→LINK
 - [End of If]
- [End of Loop]
3. Repeat while P=NULL
 - i) Call INSERT(R, P→EXP, P→COEF)
 - ii) Set P=P→LINK
- [End of Loop]
4. Repeat while Q=NULL
 - i) Call INSERT(R, Q→EXP, Q→COEF)
 - ii) Set Q=Q→LINK
- [End of Loop]
5. Return

Function: INSERT(R, EXP, COEF)

1. Allocate memory for NEW node
2. If NEW=NULL then
 - i) Print: Out of Memory
 - ii) Return
3. Set NEW→EXP=EXP, NEW→COEF=COEF, NEW→LINK=NULL
4. If R=NULL then
 - HEAD3=R=NEW
- Else
 - Set R→LINK=NEW and R=NEW
- [End of If]
5. Return

Complexity of a singly linked list

The cost to insert or delete an element into a known location in the singly linked list is $O(1)$. Whereas for a linear array the insert or delete operation cost is $O(n)$. This is because, in the case of linked list insert or delete operation do not involve any data movement just by performing pointer exchange among nodes the insertion or deletion take place. In case of array both the operations involve data replacement by n places.

Table 5.3: Comparison of Array and Linked list

Operations	Array	Linked List
Insert/delete at the beginning	$O(n)$	$O(1)$
Insert/delete at the end	$O(1)$	$O(1)$
Insert/delete in the middle	$O(n)$	$O(1)$
Indexing	$O(1)$	$O(n)$
Wastage space	$O(1)$	$O(n)$

Circular Linked List

A circular linked list is the variation of singly linked list in which the last node points to the first node of the list. The circular linked lists have neither a beginning nor an end.

In a circular linked list, there are two methods to know if a node is the first node or not.

- i) Either an external pointer, list, points the first node or
- ii) A header node is placed as the first node of the circular list.

The header node can be separated from the others by either having a sentinel value as the info part or having a dedicated flag variable to specify if the node is a header node or not.

A linked list can be implemented either in two ways:

- i) A single pointer is used to point to the header node of the linked list and the last node of the list points to the header node of the list.
- ii) Two different pointers can be used to point to the first and last node of the circular linked list respectively.

In a linear linked list once we traverse the list, then it is difficult to return back to the first node of the list. A circular linked list provides a solution to this problem. In a circular linked list the last node points to the first node of the linked list so, there is no difficulty to return back to the first node when the list is traversed.

A circular linked list can easily be formed from a linear linked list where the last node of linear linked list points to the first node of the linked list.

The circular linked list can be represented by three nodes as follows

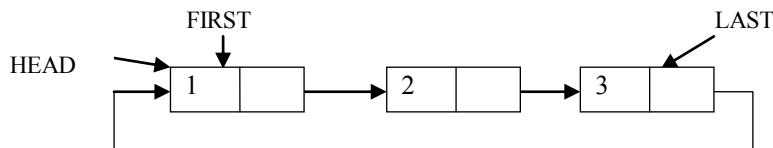


Figure 5.14: Circular linked list

Here, HEAD points to the first node of the list and last node points to the first node forming a circular list.

Operations on Circular linked list

Operations supported by a singly linked list are as follows:

Table 5.4: Operation of Circular Linked list

Operation	Description
CreateList	This operation creates a linked list.
Traverse	This operation traverse/visit all the elements of the linked list exactly once.
Insertion	This operation inserts an element to the linked list.
Deletion	This operation removes an element from the linked list.
Searching	This operation performs linear searching for a key value in the linked list.
Reverse	This operation performs the reverse of the linked list.
Merging	This operation performs merging of two linked lists in a single linked list.

Create a Circular linked list

This algorithm creates a node and appends it after the last node (which points to the first node) of the existing list. HEAD is a pointer which holds the address of the FIRST pointer of the linked list and _DATA is the value of the new node. _NEW is a pointer which holds the address of the new node and _temp is a temporary pointer.

Algorithm to create a Circular Linked List

Algorithm: CIRCULAR_ADD (HEAD, ITEM)

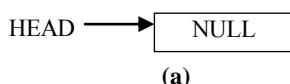
1. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LINK = NEW
2. [Check whether List is empty or not]


```
If HEAD = NULL then
          Set HEAD = NEW
```
3. Else
 - a) Set Temp = HEAD
 - b) Repeat while Temp→LINK ≠ HEAD do

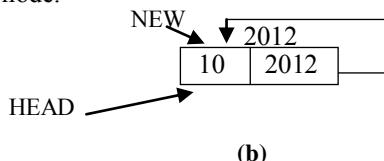

```
    Set Temp = Temp→LINK
        [End of loop]
```
 - c) Set Temp→LINK = NEW
4. [Make a new node to point to the first node]


```
Set NEW→LINK=HEAD
```
5. Return

Primarily HEAD is assigned with NULL when there is no node in a circular linked list indicates first node of the linked list. Initially it is pointing to NULL.

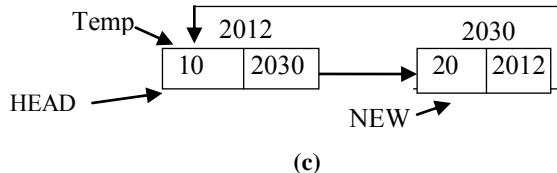


After that a NEW node is created the address is assigned with HEAD. The LINK field is assigned to the address of the HEAD node.



(b)

In the next step another node is created and linked with first node. The LINK field of the NEW node is linked with first node.



(c)

Figure 5.15 (a, b, c): Circular linked list

Add a New Node at The Beginning of the Circular Linked list

This algorithm creates a node and inserts it at the beginning of the list. HEAD is a pointer, which holds the address of the first node of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node and Temp and ln are temporary pointers.

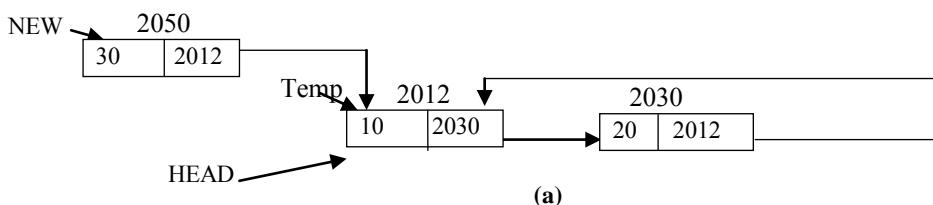
Algorithm to insert a node at the beginning

Algorithm: CIRADD_BEG (HEAD, ITEM)

1. [Create the new node]
 - a) Allocate memory for NEW node.
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA= ITEM
 - d) Set NEW→LINK= HEAD
2. a) Set Temp=HEAD
 - b) Repeat while Temp→LINK ≠ HEAD do
 - Set Temp = Temp→LINK

[End of loop]
3. [Make FIRST to point to the new node]
 - Set HEAD = NEW // to make new node the first node
 - Set Temp→LINK=HEAD
4. Return

A NEW node is created and linked with the HEAD node of Circular linked list.



(a)

At last, the address of the NEW node is assigned to the LINK field of the last node of the linked list.

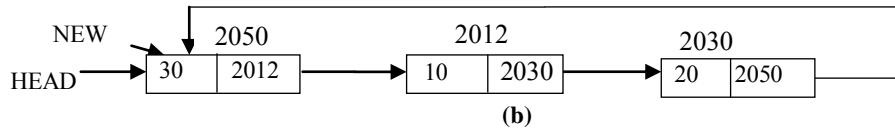


Figure 5.16(a, b): Insertion in a Circular linked list before a node

Add the New Node after a specific location of the Circular Linked list

This algorithm creates a node and inserts it after the node at location loc. HEAD is a pointer which points to the first node and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. _Temp and _PTemp are two temporary pointers to traverse the list.

Algorithm to insert a node after given node pointer

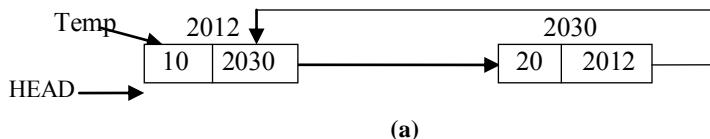
Algorithm: CIRADD_AFTER (HEAD, ITEM, LOC)

1. [Create the new node]
 - a) Allocate memory for new node.
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - c) Set NEW→LINK= Temp
2. [Make Temp point to the first node]

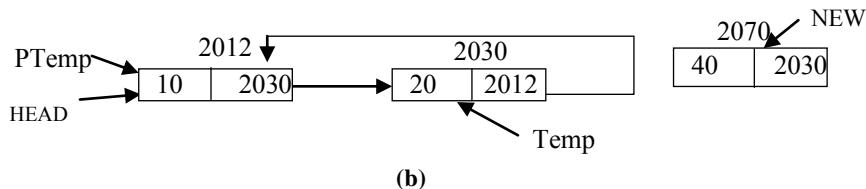
Set Temp = HEAD
3. Repeat step 3 for i=1 to LOC
3. a) Set PTemp = Temp and Temp = Temp→LINK
 - b) If Temp = HEAD then

Print: "There are less than loc+1 elements" and Return
4. Set PTemp→LINK = NEW
5. Return

In the first step Temp points to the first node of the circular linked list.



In the next step Temp is moved until loc is found. Temp points to the node at loc position. PTemp holds the address of the previous node. A NEW node is created whose LINK field is assigned with the address of the HEAD node.



At last the NEW node is inserted at the position before loc. For this example loc is considered to be 2. That means the NEW node is inserted at 2nd position of the circular linked list.

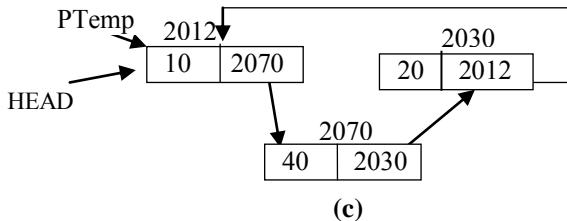


Figure 5.17 (a, b, c): Insertion in a Circular linked list

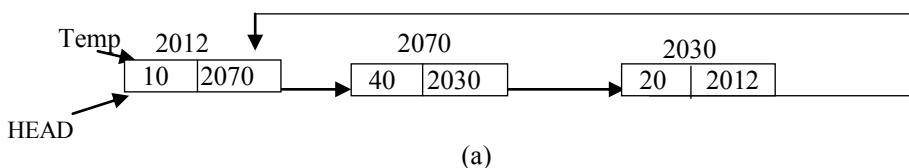
Delete a Node from The Beginning of the Circular Linked list

This algorithm deletes a node from the beginning of a circular linked list. HEAD is a pointer which holds the address of the FIRST pointer of the linked list. Temp is a temporary pointer. ITEM is the variable to store the value of the deleted item.

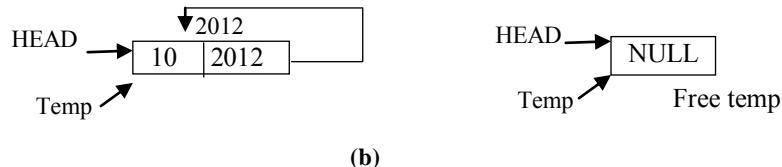
Algorithm to delete a node from the beginning**ALGORITHM: CIRDEL (HEAD)**

1. [Make Temp to point the first node]
Set Temp = HEAD
2. [Check for empty list]
If HEAD = NULL then
 Print: "The linked list is empty"
3. Else
 [Linked list contains single element]
 a) If HEAD = Temp→LINK then
 Set HEAD = NULL
 Else
 P=HEAD
 Repeat while P→LINK ≠ HEAD do
 Set P = P→LINK
 [End of loop]
 b) HEAD=HEAD→LINK
 c) P→LINK=HEAD
4. Set ITEM = Temp→DATA
5. Deallocate memory for Temp
6. Return

Firstly, Temp is assigned to the HEAD node of the circular linked list.



In the succeeding step it is checked whether the list is empty or have only one node, if the list has only one node then HEAD points to NULL and the node is deleted. This phenomena are depicted in the following figure 5.16b



If the list contains more than one node, then Temp is assigned to HEAD and HEAD moves to the next node. Then Temp is deleted. The LINK field of the last node contains the address of the new first node.

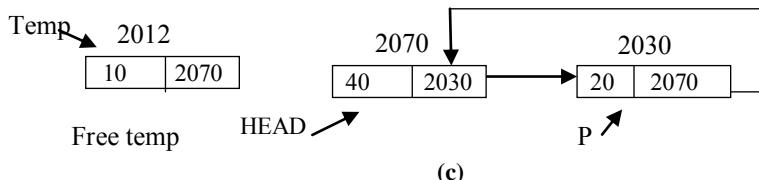


Figure 5.18 (a, b, c): Deletion from a Circular linked list

Traverse a circular Linked list

This algorithm traverses a circular linked list and prints the data part of each node of the linked list. f is a pointer which points to the starting node of the linked list . Temp’ is a temporary pointer to traverse the list.

Algorithm to display a Circular Linked List

ALGORITHM: DISPLAY (HEAD)

1. [Make Temp to point to the first node]
Set Temp = HEAD
2. [Check for empty list]
If HEAD = NULL then
 Print: "The linked list is empty" and Return
3. Print: Temp→DATA
 Set Temp = Temp→LINK
4. Repeat While Temp! = HEAD
 Print: Temp→DATA
 Set Temp = Temp→LINK
5. Return

Complexity of a circular linked list

The cost to add or delete an element from a known location in the circular linked list is O(1).

Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS is given a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example, can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.

- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Josephus Problem

The Josephus problem consists of a group of soldiers surrounded by an overwhelming enemy force. There is only a single horse available for escape. To determine which soldier is to escape, they form a circle & a number n is picked from a hat. A name is also picked from the hat. They begin to count clockwise around the circle, beginning with the soldier, whose name is picked. One soldier is removed from the circle on which count reaches n & the count starts again with next soldier, removing another soldier each time the count reaches n & the last remaining soldier is the one to take the horse.

Our problem is to find the order in which soldier will escape. The input to our program is the number n and a list of names. The names are according to the clockwise ordering of circle beginning with the soldier from whom the count is to start. The program prints the names in the order in which they are eliminated & the name of the soldier who escapes. The data structure used is, a circular linked list in which each node represents one soldier as it is possible to reach any node from any other node by moving around the circle.

The structure of the node in the circular linked list consist of name and link part

```
struct node
{
    char name[30];
    struct node *link;
}
```

Algorithm to solve Josephus problem

ALGORITHM: JOSEPHUS (HEAD)

1. Read N and Name
2. Set count= number of soldiers
Set Temp=HEAD
3. Repeat until Name=Temp->name
 Temp=Temp->LINK
 [End of loop]
4. Repeat until Count=1
 - a) Set C=1
 - b) Repeat until C= N
 - i) Set Prev=Temp
 - ii) Set Temp=Temp->LINK
 - iii) Set C=C+1
 [End of loop]
 - c) Prev->LINK=Temp->LINK
 Print Temp->DATA "soldier remove"
 - d) Set Count=Count-1
 [End of loop]
5. Print Prev->name "soldier escape"
6. Return

Doubly Linked List

A double linked list is defined as a collection of nodes in which each node has three parts: data, llink, rlink. Data contains the data value for the node, llink contains the address of node before it and rlink contains the address of node after it.

A generic doubly linked list node can be designed as:

```
struct node
{
    int data;
    struct node* rlink;
    struct node* llink;
};
```

Each node contains three parts:

- i) An information field contains *data*.
- ii) A pointer field *rlink*, which contains the location of the next node.
- iii) A pointer field *llink*, which contains the location of the preceding node.

The following figure, explain how the doubly linked list looks like:

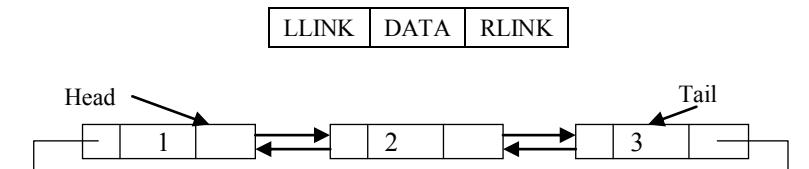


Figure 5.19: Structure of a doubly linked list

In the figure Head and Tail pointer points to the header node and tailor node of the list respectively.

Algorithm for Creation of Doubly Linked list

This algorithm creates a node and appends it at the end of existing list. Temp is a pointer which holds the address of the HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node and *_Temp* is a temporary pointer.

Algorithm to create Doubly Linked List

Algorithm: DLL_CREATE (HEAD, ITEM)

1. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LLINK = NULL
 - e) Set NEW→RLINK = NULL
2. [Whether List is empty, head is the content of HEADER]

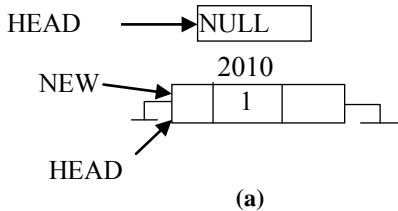
If HEAD = NULL then Set HEAD = NEW
3. Else
 - a) Set Temp = HEAD
 - b) While Temp→RLINK ≠ NULL do

```

        Set Temp = Temp→RLINK
        [End of while]
    c) Set Temp→RLINK = NEW
    d) Set NEW→LLINK = Temp
    [End of IF]
4. Return

```

Initially HEAD is assigned to NULL. Then a NEW node is created and HEAD points to that node.



After creation of HEAD node, next node is created and linked with first node. HEAD node is pointed by Temp. rlink of HEAD node holds the address of NEW node and llink of NEW node holds the address of HEAD node.

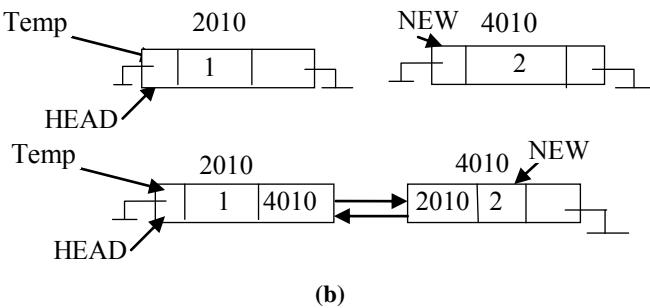


Figure 5.20 (a, b): Creation of a doubly linked list

Addition at the Beginning of the Doubly Linked list

This algorithm creates a node and inserts it at the beginning of the list. `_head` is a pointer which holds the address of the HEAD of the linked list and `ITEM` is the value of the new node. `NEW` is a pointer which holds the address of the new node.

Algorithm to insert a node at the beginning

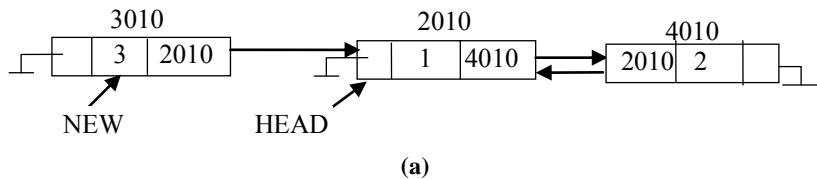
Algorithm: DLL_ADD_BEG (HEAD, ITEM)

1. [Create the new node]
 - a) Allocate memory for NEW node.
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→RLINK = HEAD
 - e) Set NEW→LLINK = NULL
2. Set HEAD→LLINK = NEW
3. [Make the HEADER to point to the NEW node]

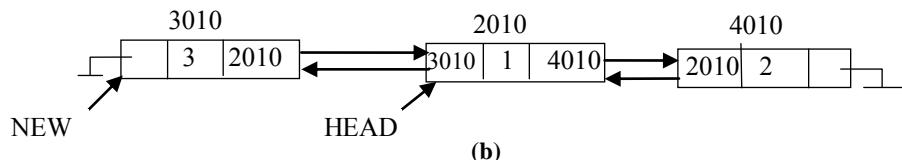
```
Set HEAD = NEW
```

4. Return

Initially a NEW node is created where rlink of NEW node contains the address of the HEAD node.



After that llink of HEAD node is assigned to the address of NEW node.



At last the HEAD pointer moves to the NEW node which is depicted in the following figure 5.19c

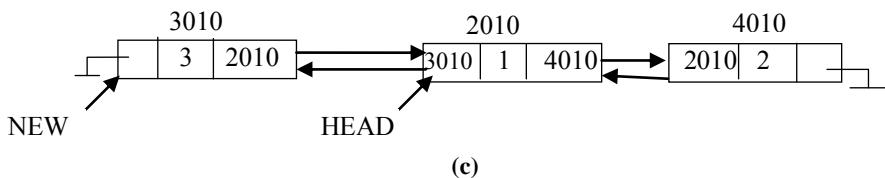


Figure 5.21 (a, b, c): Addition of node at the beginning of doubly linked list

Addition at the Beginning of the Doubly Linked list

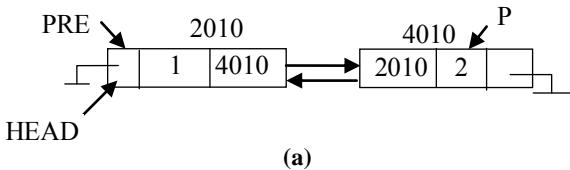
Following algorithm describes the addition of a node before any specific node in a doubly linked list. PRE holds the address of the previous node of P before which NEW node to be inserted.

Algorithm to insert a node before a node pointer

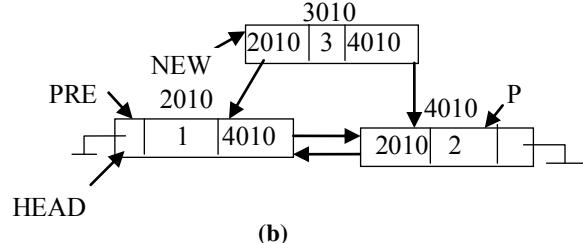
Algorithm: DLL_ADD BEFORE (HEAD, ITEM, P)

1. Set PRE = P→LLINK
2. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: Overflow and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→LLINK = PRE
 - e) Set NEW→RLINK = P
3. Set PRE→RLINK = NEW
4. Set P→LLINK = NEW
5. Return

Initially PRE holds the address of the previous node where NEW node to be inserted and P holds the address of the next node.



A NEW node is created whose rlink holds the address of the P node and llink holds the address of the PRE node.



At last, rlink of PRE is linked with NEW node and llink of P is linked with NEW node.

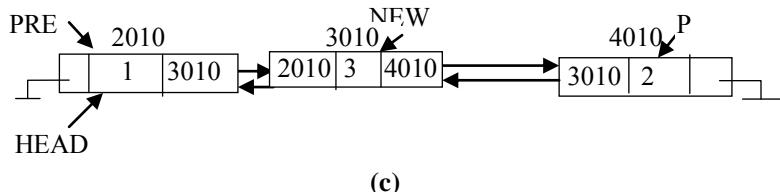


Figure 5.22 (a, b, c): Insertion in a doubly linked list

Addition after any position of the Doubly Linked list

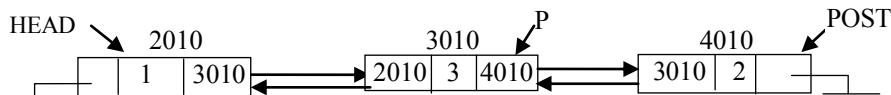
This algorithm creates a node and inserts it after the node at location P. HEAD is a pointer which points to the first node, i.e. it holds the content of HEADER of the linked list and ITEM is the value of the new node. NEW is a pointer which holds the address of the new node. POST holds the address of the next node where a new node to be inserted.

Algorithm to insert a node after a given node pointer

Algorithm: DLL_ADD_AFTER (HEAD, ITEM, P)

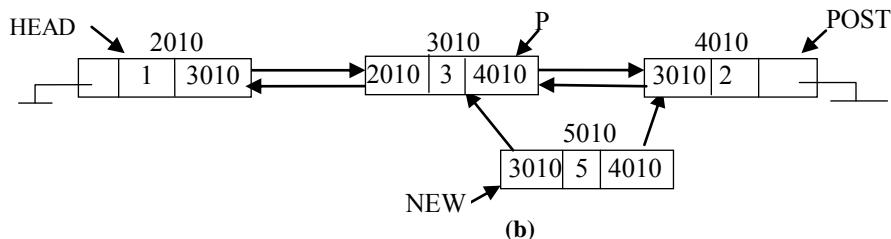
1. Set POST = P→RLINK
2. [Create the new node]
 - a) Allocate memory for NEW node
 - b) IF NEW = NULL then Print: "Memory not Available" and Return
 - c) Set NEW→DATA = ITEM
 - d) Set NEW→RLINK = POST
 - e) Set NEW→LLINK = P
3. Set P→RLINK = NEW
4. Set POST→LLINK = NEW
5. Return

At first POST is assigned with the RLINK of the node P after which a new node to be inserted.



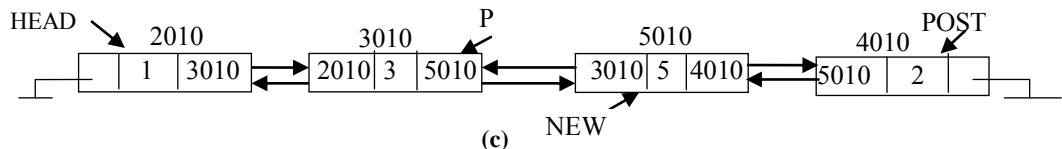
(a)

After that a NEW is created where ITEM is placed in DATA field and RLINK holds the address of the POST where LLINK contains the address of the P node.



(b)

At last, RLINK of node P is linked with NEW node and LLINK link of POST node is linked with NEW node



(c)

Figure 5.23 (a, b, c): Addition of a node after any position in a doubly linked list

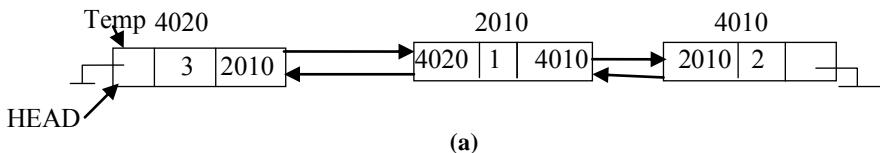
Addition at last position of the Doubly Linked list

In the following algorithm addition of node at the end of the doubly linked list is described. Here Temp points to the HEAD node. Then Temp moves to the end of the linked list. At last NEW node is linked with RLINK of Temp node.

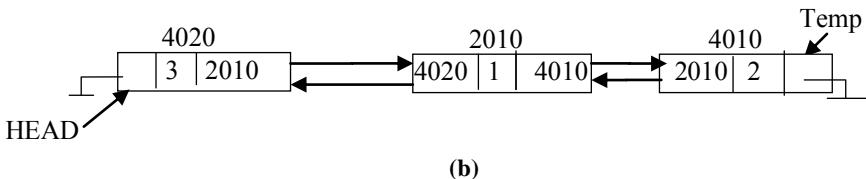
Algorithm to insert a node at the end

```
Algorithm: ADD_END (HEAD, ITEM)
1. [Make temp to point to the first node]
   Set Temp = HEAD
2. Repeat while Temp→RLINK ≠ NULL
   Set Temp = Temp →RLINK
   [End of loop]
3. [Create the new node]
   a) Allocate memory for NEW node
   b) IF NEW = NULL then Print: Overflow and Return
   c) Set NEW→DATA = ITEM
   d) Set NEW→RLINK = NULL
   e) Set NEW→LLINK = Temp
4. Set Temp→RLINK = NEW
5. Return
```

Firstly, Temp is assigned to the HEAD node.



Now, move Temp to the end of the doubly linked list.



At last a NEW node is created and linked with Temp

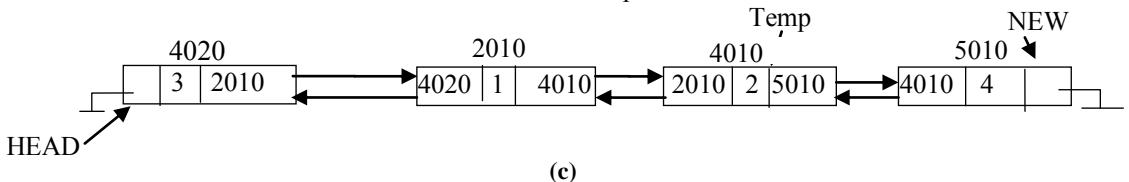


Figure 5.24 (a, b, c): Insertion of a new node at the end of a doubly linked list

Traverse a Doubly Linked list in Forward Direction

This algorithm traverses a linked list and prints the DATA part of each node of the linked list in the forward direction (from the first node to last). HEAD is a pointer which points to the starting node of the linked list. Temp is a temporary pointer to traverse the list.

Algorithm to traverse in forward direction

Algorithm: DLL_FTRAVERSE (HEAD)

1. [Check for empty list]
 - i) If HEAD = NULL then
 - i) Print: "The linked list is empty"
 - ii) Return
 2. Set Temp = HEAD
 3. Repeat while Temp ≠ NULL
 - i) Print: Temp→DATA
 - ii) Set Temp = Temp→RLINK

[End Of Loop]
 4. Return

Traverse a Doubly Linked list in Backward Direction

This algorithm traverses a linked list and prints the data part of each node of the linked list in backward direction (from the last node to first). HEAD is a pointer which points to the starting node of the linked list. Temp is a temporary pointer to traverse the list. At first Temp moves in forward direction using RLINK and then Temp is used for backward traverse of the doubly linked list using LLINK.

Algorithm to traverse in backward direction

```
Algorithm: DLL_BTRAVERSE (HEAD)
1. [Check for empty list]
   If HEAD = NULL then
      i) Print: "The linked list is empty"
      ii) Return
2. [Make temp to point to the first node]
   Set Temp = HEAD
3. Repeat while Temp→RLINK ≠ NULL
   Set Temp = Temp →RLINK
   [End of loop]
4. Repeat while Temp ≠ NULL
   i) Print: Temp→DATA
   ii) Set Temp = Temp →LLINK
   [End of loop]
5. Return
```

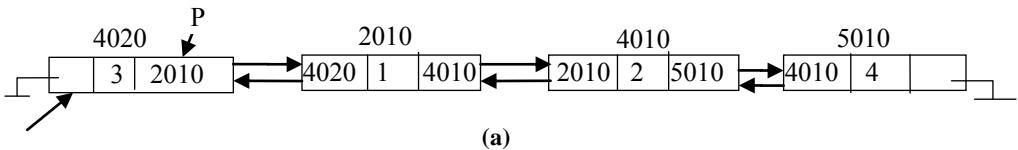
Search an Item in a doubly linked list

This algorithm describes searching of an ITEM in a doubly linked list. P is a pointer that assigned the Head node address of the doubly linked list. P moves from left to right until the ITEM is found or the list end.

Algorithm to search an item in doubly linked list

```
Algorithm: DLL_SEARCH (HEAD, ITEM, LOC)
1. Set P = HEAD, LOC=NULL
2. [Check for empty list]
   If P = NULL then
      i) "The linked list is empty"
      ii) Return
3. Repeat step 4 while P ≠ NULL
4.   If P→DATA = ITEM then
      i) Print: "Element found"
      ii) Set LOC = P
      iii) Return
   Else
      P = P→RLINK
   [End of If]
   [End of Loop]
5. Print: "Element not found"
6. Return
```

In the first step P is assigned with the address of the first node of the doubly linked list.



HEAD

Let us consider the ITEM to be searched is 2 which is at 3rd position. P moves until ITEM is found.

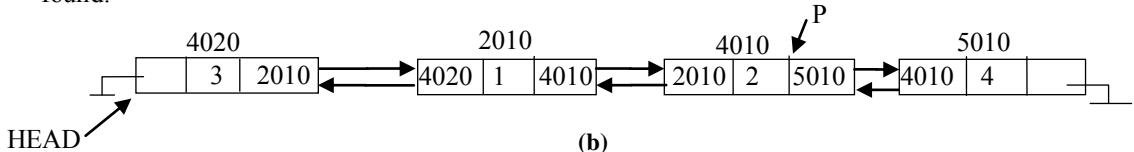


Figure 5.25 (a, b): Searching of an ITEM in doubly linked list

Delete a Node from the Beginning of a doubly linked list

In the following algorithm the process of deleting a node from the beginning of a doubly linked list is described. HEAD denotes the first node of the list. In the figure 5.26b this phenomena are depicted.

Algorithm to delete a node from the beginning

Algorithm: DLL_DELETE_BEG (HEAD)

1. [Check for empty list]
 - a) IF HEAD = NULL then
 - a) Print: "The linked list is empty"
 - b) Return
 2. [Make temp to point the first node]
 - Set Temp = HEAD
 3. Set HEAD = Temp→RLINK
 4. Set HEAD→LLINK = NULL
 5. Set ITEM = Temp→DATA
 6. Set Temp→RLINK = NULL
 7. Set Temp→LLINK = NULL
 8. Deallocate memory for Temp Node
 9. Return

Delete a Node from the End of a doubly linked list

The following algorithm describes the procedure of removing a node from the end of a doubly linked list. Here, HEAD denotes the first node of the list. Temp is used to move the pointer from the first node to last node. PTemp is used to hold the address of the preceding node of the node to be deleted. After reaching at last position the Temp is de-allocated. Figure 5.24c depicts this phenomenon pictorially.

Algorithm to delete a node from the end

Algorithm: DLL_DELETE_END (HEAD, ITEM)

1. [Check for empty list]

```

    IF HEAD = NULL then
        a) Print: "The linked list is empty"
        b) Return
    2. Repeat step while Temp→RLINK ≠ NULL
        a) Set PTemp = Temp
        b) Set Temp = Temp→RLINK
    [End of loop]
    3. Set ITEM = Temp→DATA
    4. Set Temp→LLINK = NULL
    5. Set PTemp→RLINK = NULL
    6. Deallocate memory for Temp Node
    7. Return

```

Delete a Node after a specified position (P) of a doubly linked list

Here, the algorithm specifies the procedure of deleting a node which is after a node P. Temp holds the address of the next node of the node P and POST holds the address of the next node of the node Temp which is to be deleted. Figure 5.24d describes this process.

Algorithm to delete a node after a given node pointer

Algorithm: **DLL_DELETE_AFTER (HEAD, ITEM, P)**

1. Set Temp = P→RLINK
2. Set POST = Temp→RLINK
3. Set P→RLINK = POST
4. IF POST ≠ NULL then Set POST→LLINK = P
5. Set ITEM = Temp→DATA
6. Set Temp→RLINK = NULL
7. Set Temp→LLINK = NULL
8. Deallocate memory for Temp Node
9. Return

Delete a Node before a specified position (P) of a doubly linked list

Here, the algorithm specifies the procedure of deleting a node which is before a node P. Temp holds the address of the preceding node of the node P and PRE holds the address of the preceding node of the node Temp which is to be deleted. Figure 5.24e describes this process.

Algorithm to delete a node before given node pointer

Algorithm: **DLL_DELETE_BEFORE (HEAD, ITEM, P)**

1. Set Temp = P→LLINK
2. Set PRE= Temp→LLINK
3. IF PRE ≠ NULL then Set PRE→RLINK = P
4. Set P→LLINK = PRE
5. Set ITEM = Temp→DATA
6. Set Temp→RLINK = N ULL
7. Set Temp→LLINK = NULL

8. Deallocate memory for Temp Node
9. Return

Delete a Node from the doubly linked list having a specific DATA value

This algorithm finds and deletes a node whose value is NO. Temp is a pointer which holds the address of HEADER of the linked list.

Algorithm to delete a node by value

ALGORITHM: DDEL (HEAD, NO)

1. Set Temp = HEAD //to make temp to point the first node
2. [Check for empty list]
 - If HEAD = NULL then

Print: "The linked list is empty" and Return
3. Repeat Step 4 to 5 until temp is NULL
 - a) If Temp→DATA = NO then
 - i) If TEMP = HEAD then// node to be deleted is the first node

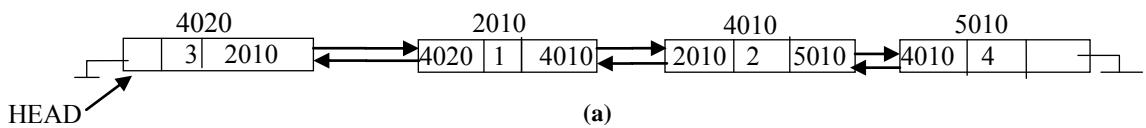
Set HEAD = Temp→RLINK
Set Temp→RLINK→LLINK = NULL
 - Else if Temp→RLINK = NULL //Check for last node

Set P=Temp→LLINK
Set POST= Temp→RLINK
Set P→RLINK=NULL
 - Else

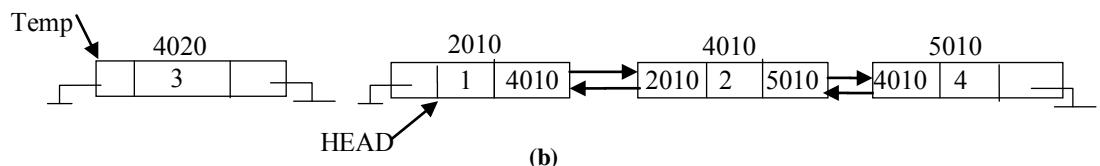
Set POST→LLINK = Temp→LLINK
Set P→RLINK=Temp→RLINK
 - ii) Deallocate memory for Temp Node // de allocate node
 - iii) Return
 - Else

Set Temp = Temp→RLINK
4. Print: "Element not found"
5. Return

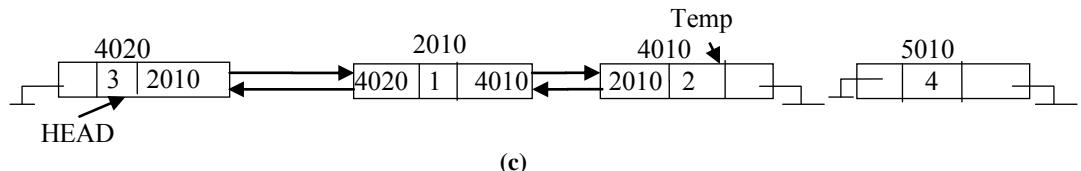
Following figure 5.26a denotes a doubly linked list where HEAD points to the first node of the list.



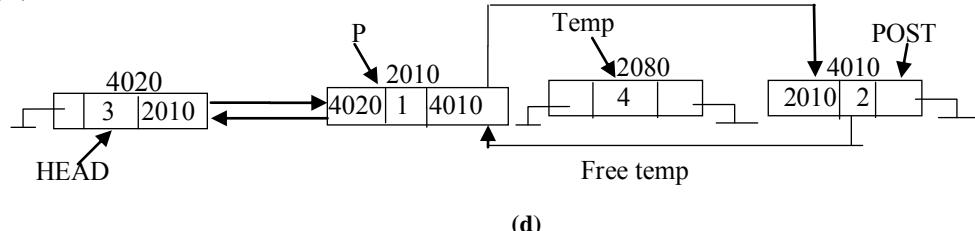
If the first node to be deleted, then how the list will like is presented in the following figure 5.26b.



In the following figure 5.26c deletion of a node from last position of doubly linked list is described.



In the following figure 5.26d, the deletion of a node Temp which is after a specific node P is shown.



In the succeeding figure it is pictorially presents that how a node to be deleted which is before a specific node of a doubly linked list.

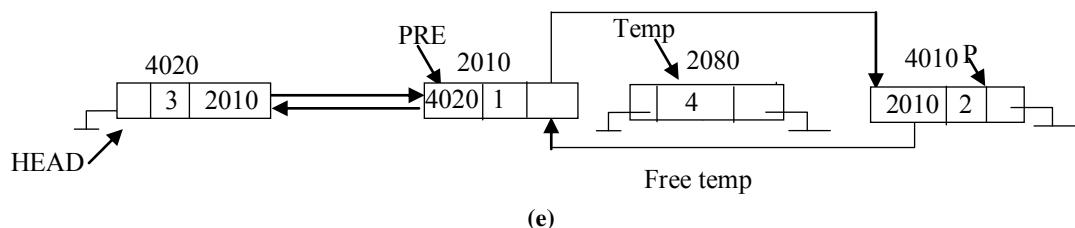


Figure 5.26 (a-e): Node deletion at any position in a doubly linked list

Time Complexity of Doubly Linked list

The time complexity for insertion and deletion from a doubly linked list is $O(1)$, as there is no movement of nodes just by exchanging some pointers the a new node can be inserted or an existing node can be deleted. Whereas for traversal, the complexity is $O(n)$.

Circular Doubly Linked List:

The advantages of both double linked list and circular linked list are incorporated into another list structure that is called circular doubly linked list and it is known to be the best of its kind.

The following is a schematic representation of a circular doubly linked list.

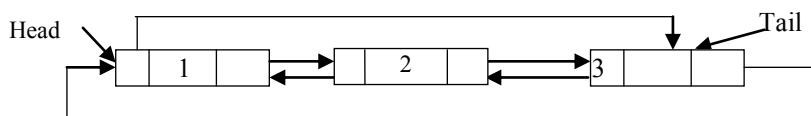


Figure 5.27: Circular Doubly linked list

As per above shown illustrations, following are the important points to be considered.

- Last Link's rlink points to the first node of the list in both cases of singly as well as doubly linked list.
- First Link's llink points to the last node of the list in case of doubly linked list.

Disadvantages of Linked List

- Linked list consumes more memory space for storing addresses of the next node.
- Searching will be slow as only linear searching will be possible.
- Direct access to any node not possible, therefore, if a node required to access it is required to traverse from the first node onwards until the desired node is found.

Therefore, arrays are suitable when a large number of searching operations are necessary, but insertion/deletion is very small in number. Whereas linked-list are suitable, when there will be frequent insertion/deletion, but very few search operations are required.

Summary

- A linked list is a linear ordered collection of finite homogeneous data elements called node.
- The linked list overcomes the demerits of array in terms of memory allocation.
- Linked list can be classified into 4 types single linked list, circular linked list, doubly linked list and circular doubly linked list.
- Polynomial addition can be done using linked list.
- Addition, Deletion and Insertion of a node in a linked list is faster than array.

Exercises

1. What are the advantages and disadvantages of linked list data structure over an array?
2. Write an algorithm to insert a data X after a specific data item Y in the linked list.
3. Write a function to delete the n^{th} node of a singly linked list. The error conditions are to be handled properly.
4. Write an algorithm to delete any node from the singly liked list where the key value of the node is known.
5. Write an algorithm to delete a node from a doubly linked list.
6. Write an algorithm to delete all nodes having greater than a given value, from a given singly linked list.
7. Write a function to reverse the direction of all links of a singly linked list.
8. Write an algorithm to add two polynomials using linked list.
9. How the polynomial $4x^3 - 10x^2 + 3$ can be represented using linked list?
10. Write a C program to create a doubly linked list in ascending order of information.
11. Write a C program to join two sorted linked list so that the third list is in sorted order.
12. Write a C program to show the information of a doubly linked list in reverse order.
13. Write a C program to INSERT and DELETE node in a circularly doubly linked list. Display the elements of the list.
14. Write C program to insert an element in a sorted doubly linked list, so that the list remain in sorted order after insertion.
15. Write a program in C language to find the predecessor of a node in linked list.
16. Choose the correct alternatives for the following:

- i) The situation when in a linked list START = NULL is
 - a) Underflow b) Overflow c) Saturated d) None
- ii) A linked list follows
 - a) random access mechanism b) sequential access mechanism
 - c) no access mechanism d) none of these.
- iii) A linear collection of data elements where the linear node is given by means of pointer is called
 - a) Linked list b) Node list c) Primitive list d) None of these.
- iv) In linked list representation a node contains at least
 - a) node address field, data field b) node number field, data field
 - c) next address field, information field d) none of these.
- v) Inserting a new node after a given node in a doubly linked list requires
 - a) four pointer exchanges b) two pointer exchanges
 - c) one pointer exchanges d) n pointer exchange.
- vi) The n^{th} node in a singly linked list can be accessed via
 - a) The head node b) The tail node c) $(n-1)^{\text{th}}$ node d) None
- vii) Linear order in linked list is provided through
 - a) Index number b) The implied position of the node c) Pointer d) None
- viii) Null pointer is used to tell
 - a) End of a linked list b) Empty pointer field of a structure
 - c) The linked list is empty d) All the above
- ix) In linked list the successive elements
 - a) Must occupy contiguous space in memory
 - b) Need not occupy contiguous space in memory
 - c) Must not occupy contiguous space in memory
 - d) None of the above
- x) Searching in a linked list requires linked list be created
 - a) In sorted order b) In any order c) Without underflow condition d) None of the above
- xi) Deletion of a node in linked list involves keeping track of the address of the node
 - a) Which immediately follows the node that is to be deleted
 - b) Which immediately precedes the node that is to be deleted
 - c) The node to be deleted
 - d) None of the above
- xii) Header in a linked list is a special node at the
 - a) End of the linked list b) At middle of a linked list
 - c) Beginning of the linked list d) None of the above
- xiii) Header linked list in which last node points to the header node is called
 - a) Grounded header list b) Circular header list c) General header list d) None of the above
- xiv) Representing polynomial using linked list requires each node having
 - a) Two fields b) Three fields c) More than three fields d) None of the above
- xv) Inserting a node after a given node in a doubly linked list requires
 - a) One pointer change b) Two pointer change c) Four pointer change d) None of the above

CHAPTER 6

STACK AND QUEUE

“The art of programming is the art of organizing complexity”. - W. W. Dijkstra

In this chapter, we discuss some elementary data structures like stack and queue along with their properties, different operations that are performed on them, algorithms of their different operations and applications of them.

In 1946, Alan M. Turing first proposed the stack, in the computer design. In 1955, Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea of the stack and filed a patent in 1957. The Australian Charles Leonard Hamblin in the first half of 1957 developed the same concept, independently.

KEY FEATURES

-  Stack
-  Evaluation of Expression
-  Queue
-  Circular Queue
-  Dequeue
-  Priority Queue

STACK

A stack is one of the most important and useful non-primitive linear data structure in computer science. Real-life examples of the stack are a stack of books, a stack of plates, a stack of cards, a stack of coins, etc.

Definition: A stack is a sequential collection of elements into which new elements are inserted and from which, elements are deleted only at one end.

As all the insertion and deletion in a stack is done from the top of the stack, the lastly added element will be first to be removed from the stack. That is the reason why stack is also called **Last-In-First-Out (LIFO)** data structure. Note that the most frequently accessed element in the stack is the top most elemental, whereas the least accessible element is the bottom of the stack.

In the stack, the top variable is used to point the top of the stack. The following tasks are performed by the top variable:

- To keep track, how many cells are used,
- Whether the stack is full or empty
- Insert new element in the stack
- Delete elements from the stack

Operations on Stack

The stack is an abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, we can implement a stack using either array or linked list. The stack includes a finite sequence of the same type of items with the different operations described in table 6.1.

Table 6.1: Operations on Stack

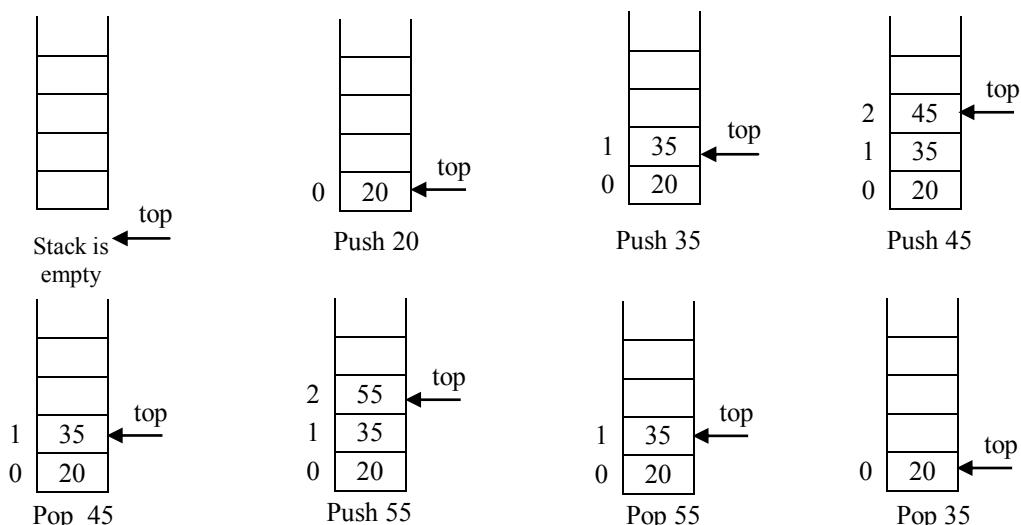
Operation	Description
Creation	This operation creates a stack and initializes the stack.
Isempty	This operation checks whether the stack is empty or not.
Isfull	This operation checks whether the stack is full or not.
Insertion (Push)	This operation inserts an item only at the top of the stack when the stack is not full.
Deletion (Pop)	This operation deletes an item only from the top of the stack when the stack is not empty.
Peek	This operation returns the value of the top of the stack without removing the element from the stack.

The insertion (or addition) operation is referred as push, and the deletion (or remove) operation as pop. A stack is said to be empty if the stack contains no elements. At this point, the top of the stack is present at the bottom of the stack. In addition, it is full when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point, the top pointer is at the highest location of the stack. If the pop operation in the stack when it is in the empty state then underflow occurs. Similarly, if the push operation is done in a full stack, then stack overflow happens.

Stack as an ADT promotes data abstraction and focuses on what operations it does rather than how it does (implements). Therefore, the stack can be implemented with an array or with the linked list.

Stack representation with Array

The array can be used to implement a stack of fixed size, therefore only fixed a number of data items can be pushed and popped. Consider the following figure 6.1, the stack of size 4; therefore, maximum only 4 data items can be inserted. The top index always keeps track of the last inserted element of the stack, which is the top of the stack. Initially, the top is initialized by -1 (for the zero-based array) when there are no items in the stack, i.e. stack is empty.

**Figure 6.1:** Stack Operations

When a new item is inserted in the stack, the top is incremented by 1 before the item is stored in the stack. Once all the four items are inserted then pop is 3, as shown in the figure. Now, if we try to insert next item, it leads to an overflow condition. It indicates that the stack is full and we cannot insert the new item. If the size of the array is MAXSIZE then when the top equals to MAXSIZE -1 then the stack is full.

When an item is deleted from the stack, the top is decremented by 1, after the item is removed from the queue. Now, if top = -1 then if we try to delete an item, it results in underflow condition. It indicates that the stack is empty and we cannot delete an item.

Therefore, it is required to check these conditions whenever push and pop operations take place.

Algorithm to insert (push) onto the stack

Algorithm: PUSH (STACK, ITEM)

[*STACK* is an array of *MAXSIZE* and *ITEM* is an item to be pushed onto stack]

1. [Check for stack overflow]
 - a) If TOP = MAXSIZE - 1 then
 - a) Print: Overflow
 - b) Return
 2. [Increase top by 1]
 - Set TOP = TOP + 1
 3. [Insert item in new top position]
 - Set STACK[TOP] = ITEM
 4. Return

Algorithm to deletes (pop) the top element from the stack.

Algorithm: POP (STACK, ITEM)

[*STACK* is an array and *ITEM* is an item to be popped from stack]

1. [Check for stack underflow]
 - If TOP = -1 then
 - a) Print: Underflow
 - b) Return
 2. [Assign top element to item]
 - Set ITEM = STACK[TOP]
 3. [Decrease top by 1]
 - Set TOP = TOP - 1
 4. Return

The stack can be represented using the following structure:

```
struct STACK
{
    int a[MAXSIZE];
    int top;
};
```

Stack Representation with Linked List

Another way to represent stack is by using the singly linked list, which is also known as **Linked Stack**. A linked list is a dynamic data structure and each element of a linked list is a node that contains a value and a link to its neighbor. The link is a pointer to another node that contains a value and another pointer to another node and so on.

The linked list header acts as the top of the Stack. All push or pop operations are taking place at the front of the linked list. Each operation always changes the header of the linked list. When the stack is empty then HEAD is null. If the stack has at least one node, the first node is the top of the stack. In the push operation, it needs to add a new node to the front of the list. The pop operation removes the first node of the linked list when the stack is not empty.

Algorithm of the push operation using linked list

Algorithm: PUSH (HEAD, ITEM)

[*HEAD is a pointer to the first node and ITEM is an item to be pushed onto stack*]

1. [Create the new node]
Allocate memory for NEW node
2. If NEW = NULL then
 - i) Print: Out of memory
 - ii) Return
3. Set NEW→DATA = ITEM
4. Set NEW→LINK = HEAD
5. Set HEAD = NEW
6. Return

Algorithm of pop operation using linked list

Algorithm: POP (HEAD, ITEM)

[*HEAD is a pointer to the first node and ITEM is an item to be popped from stack*]

1. [Whether List is empty]
If HEAD = NULL then
 - i) Print: Stack is underflow
 - ii) Return
2. ITEM = HEAD→DATA
3. Set P = HEAD
4. HEAD = HEAD→LINK
5. Set P→LINK = NULL
6. De-allocate memory for node P
7. Return

Comparisons of stack representation using linked list over array

- The array is fixed size, therefore, a number of elements will be limited in the stack. Since linked list is dynamic and can be changed easily, so the number of elements can be changed.

- The pointers in linked list consume additional memory compared to an array.
- In array and linked list push, pop operations can be done in O(1).

Applications of Stack

The stack is a very useful data structure. Most of the modern computers and microprocessor provide an inbuilt hardware stack. Even there are stack-oriented computer architectures.

1. A very important application of stack is to implement recursive function call and processing of function calls such as passing arguments.
2. Evaluation of Arithmetic expressions.
3. To simulate recursion.
4. The scope rule and block structure can also be implemented using the stack. Stacks are used in the development of Compilers, System programs, Operating systems and in many elegant application algorithms.
5. Stack is used to implement different algorithms, Depth first search, Quicksort, Mergesort etc.

Processing of Function calls

One of the most important applications of the stack is the processing of subprogram calls and their termination.

The program must remember the place where the call was made: so that it can return back after the subprogram is complete.

Suppose we have three subprograms called A(), B() and C() and one main program and main() invokes A(). A() invokes B() and B() in turn invokes C(). Then B will not have finished its work until C() has finished and returned. Similarly main() is the first to start work, but it is the last to be finished.

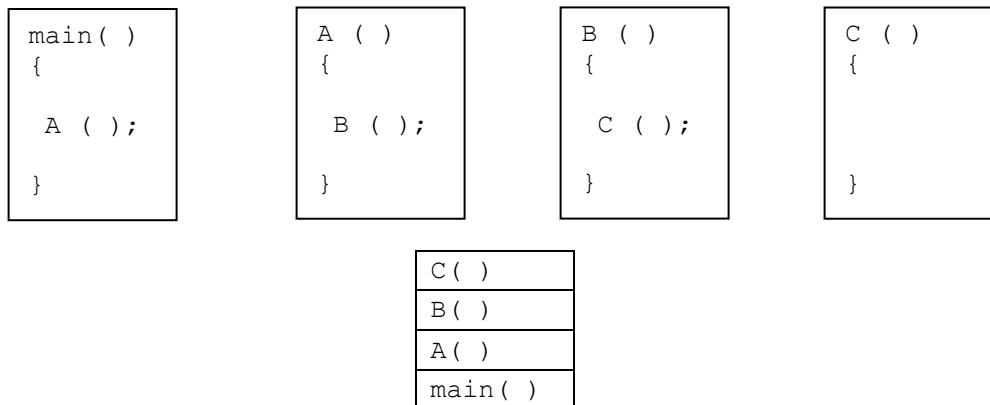


Figure 6.2: Function call processing

These function calls are pushed into the stack according to the order in which they were called along with their parameters and processed according to the order in which they are popped from the stack. The stack, which is used in function-call, is generally implemented in physical memory and in the reverse order. That means the stack top locates the last address.

Evaluation of Arithmetic Expressions

An expression is defined as a number of operands or data items combined with several operators. An Arithmetic expression consists of arithmetic operators and operands.

There are three types of notations in an arithmetic expression.

- i) Infix Notation
- ii) Prefix Notation
- iii) Postfix Notation

Infix Notation

Most usually, in arithmetic expressions, the binary operator appears between its two operands. This is called infix notation. The general form of the infix notation is:

Op1 operator Op2 where Op1 and Op2 are two operands

Example:

a + b

Prefix Notation

In prefix notation, binary operators appear before its two operands. This notation is also known as **Polish notation**. In prefix notation, the operations that are to be performed is absolutely determined by the positions of the operators and operands in the expression. Therefore, parentheses are never used when writing expressions in prefix notation. The general form of the prefix notation is:

operator Op1 Op2 where Op1 and Op2 are two operands

Example:

+ a b

Postfix Notation

In postfix notation, binary operators appear after its two operands. This notation is also known as **Reverse Polish notation**. In postfix notation, the operations are to be performed is absolutely determined by the positions of the operators and operands in the expression. Therefore, parentheses are never used when writing expressions in postfix notation. The general form of the postfix notation is:

Op1 Op2 operator where Op1 and Op2 are two operands

Example:

a b +

We are familiar with the conventional infix notation. However, postfix notation is most suitable for a computer to calculate any expression and it is the universally accepted notation for designing Arithmetic Logic Unit (ALU) of CPU. Therefore, it is necessary for us to study the postfix notation. Postfix expression is the way the computer looks towards any arithmetic expression.

Disadvantages of infix expression

- The infix expression is evaluated by traversing in more than one passes.

- The infix expression does not exclusively define the order in which the operators are to be evaluated. In infix expression, the operators are evaluated on basis of operator precedence convention.
- In infix notation, one can increase the operator precedence by using parentheses. Therefore, parentheses can change the order of evaluation in an infix expression.
- While storing the fully parenthesized expression is wasteful, since the parentheses in the expression need to be stored to evaluate the expression properly.

Advantages of postfix expression

- The postfix expression is evaluated by traversing only one pass. Therefore, evaluation of postfix expression is faster than infix expression.
- Postfix expression is already arranged according to the operator precedence. Therefore, one never needs to look at the precedence of the operator during evaluation
- In postfix expression, parentheses are never used.

The Computer usually evaluates an expression written in infix notation in two steps:

- Converts the infix expression to equivalent postfix expression
- Evaluates the postfix expression in a single pass

Converting infix expression to postfix form

The order of evaluation can be fixed by assigning a priority to each operator. The operators within parentheses having the highest priority will be evaluated first. When an expression has two operators with same priority then the expression is evaluated according to its associativity (left to right or right to left) order. In table 6.2, the priorities of different operators are specified.

Table 6.2: Priority of Operators in the order of evaluation

Operator	Description	Priority
+ -	Unary operator	5
^	Power operator	4
* / %	Multiplication, Division, Remainder	3
+ -	Addition, Subtraction	2
< <= > >= !=	Relational operators	1

There is an algorithm to convert an infix expression to the equivalent postfix expression. A stack is used here to store operators rather than operand. The purpose of the stack is to reverse the order of the operators in the expression.

Algorithm converts an infix expression to the equivalent postfix expression.

Algorithm: POSTFIX (Q, P)

[Q is a given infix expression and P is a postfix expression]

- Push "(" onto stack & add ")" to the end of Q.
- Scan Q from left to right and repeat step 3 to 6
for each element (symbol) of Q while the stack is not empty
- If the element is an operand then add it to P.

6. 8 | Data Structures and Algorithms with C

4. If the element is left parenthesis "(" then push it onto the stack.
 5. If the element is an operator then:
 - a) Repeatedly pop from stack (until the element on top of the stack has higher or same precedence than the operator currently scanned) and add it to P.
 - b) Add the operator to stack.
 6. If the element is a right parenthesis ")" then:
 - a) Repeatedly pop from stack and add to P each operator until a left parenthesis "(" is found
 - b) Pop the left parenthesis from the stack.
- [End of Loop]
7. Return

Example:

Find the postfix expression of the following infix expression:

$$Q = A + (B * C - (D / E \uparrow F) * G) * H$$

Serial Number	Symbol Scanned	Stack	Postfix Expression (P)
Initial		(
1	A	(A
2	+	(+	A
3	((+ (A
4	B	(+ (A B
5	*	(+ (*	A B
6	C	(+ (*	A B C
7	-	(+ (-	A B C *
8	((+ (- (A B C *
9	D	(+ (- (A B C * D
10	/	(+ (- (/	A B C * D
11	E	(+ (- (/	A B C * D E
12	\uparrow	(+ (- (/ \uparrow	A B C * D E
13	F	(+ (- (/ \uparrow	A B C * D E F
14)	(+ (-	A B C * D E F \uparrow /
15	*	(+ (- *	A B C * D E F \uparrow /
16	G	(+ (- *	A B C * D E F \uparrow / G
17)	(+	A B C * D E F \uparrow / G * -
18	*	(+ *	A B C * D E F \uparrow / G * -
19	H	(+ *	A B C * D E F \uparrow / G * - H
20)	STACK EMPTY	A B C * D E F \uparrow / G * - H * +

Postfix expression A B C * D E F \uparrow / G * - H * +

Example:

Find the postfix expression of the following infix expression:

$$Q = (a + b * x) / (!a - d). s - c \wedge y$$

Serial Number	Symbol Scanned	STACK	Postfix Expression
Initial		(
1	(((
2	A	((A	
3	+	((A+	
4	B	((A+)	a b
5	*	((A+*)	a b
6	X	((A+*)X	a b x
7)	(a b x * +
8	/	(/	a b x * +
9	((/(a b x * +
10	A	(/(A	a b x * + a
11	!	(/(A!	a b x * + a
12	-	(/(A-	a b x * + a !
13	D	(/(A-D	a b x * + a ! d
14)	(/	a b x * + a ! d -
15	.	(.	a b x * + a ! d - /
16	S	(.	a b x * + a ! d - / s
17	-	(-	a b x * + a ! d - / s .
18	C	(-	a b x * + a ! d - / s . c
19	\wedge	(- \wedge	a b x * + a ! d - / s . c
20	Y	(- \wedge Y	a b x * + a ! d - / s . c y
21)	stack empty	a b x * + a ! d - / s . c y \wedge -

The equivalent postfix expression of the given infix expression is a b x * + a ! d - / s . c y \wedge -

The time complexity of evaluation algorithm is O (n) where n is a number of characters in input expression.

Evaluation of a Postfix Expression

Algorithm finds the value of an arithmetic expression P written in postfix notation.

Algorithm: EVALUATION (P)

[P is a postfix expression]

1. Add a right parenthesis ")" at the end of P.
2. Read P from left to right and repeat step 3 and 4 for each element of P until the ")" is found.
3. If an operand is found, put it onto the stack.

4. If an operator # is found then
 - a) Pop the two top elements of the stack,
Where A is the top element and B is the next to top element
 - b) Evaluate R = B # A
 - c) Push R onto the stack

[End of If]

[End of Loop]
5. Set Result equals to the top element on stack
6. Return

Example:

Find the value of following postfix expression:

5 3 2 * 8 + *

Serial Number	Symbol Scanned	Stack	Output
1	5	5	
2	3	5 3	
3	2	5 3 2	
4	*	5 6	
5	8	5 6 8	
6	+	5 14	
7	*	70	
			70

The time complexity of evaluation algorithm is O(n) where n is a number of characters in input expression.

Converting infix expression to prefix form

There is an algorithm to convert an infix expression to the equivalent prefix expression. A stack is used here to store operators rather than operand.

Algorithm converts an infix expression to the equivalent prefix expression.

Algorithm: PREFIX (A, B)

[A is a given infix expression and B is a prefix expression]

1. Push ")" onto STACK, and add "(" to end of the A
2. Scan A from right to left and repeat step 3 to 6
for each element(symbol) of A while the STACK is not empty
3. If the element is an operand then add it to B
4. If the element is a right parenthesis then push it onto STACK
5. If the element is an operator then:
 - a) Repeatedly pop from STACK and add to B, each operator (on the top of STACK) which has same or higher precedence than the operator.
 - b) Add operator to STACK
6. If the element is a left parenthesis then

- | |
|---|
| a) Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is found) |
| b) Pop the left parenthesis from stack. |
| [End of loop] |
| 7. Return |

We can also convert an infix expression to prefix expression in three steps. In the first step reverse the infix expression and convert left parenthesis to right parenthesis and vice versa. Then in the second step, the modified infix expression is converted into postfix expression which algorithm is discussed above. Finally, the postfix expression is reversed to get the equivalent prefix expression.

QUEUE

The queue is also another useful non-primitive linear data structure in computer science. A real-life example of the queue is line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

Definition: A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

The element to enter the queue first will be deleted from the queue first. That is why a queue is called **First-In-First-Out (FIFO)** system.

The concept of the queue can be understood by our real life problems. For example, a customer comes and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of the queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service center.

Operations on Queue

The queue is an abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, the queue can be implemented with an array or with the help of the linked list. The queue includes a finite sequence of the same type of items with the different operations described in table 6.3.

Table 6.3: Operations on Queue

Operation	Description
Creation	This operation creates a queue and initialization is done here.
isempty	This operation checks whether the queue is empty or not.
isfull	This operation checks whether the queue is full or not.
Insertion (Enqueue)	This operation inserts an item only at the rear of the queue when the queue is not full.
Deletion (Dequeue)	This operation deletes an item only from the front of the queue when the queue is not empty.
Peek	This operation returns the value of the front of the queue without removing the element from the queue.

Table 6.4: Difference between stack and queue

Stack	Queue
In the stack, items are inserted and deleted at the one end of the list	In the queue, items are inserted at one end (called rear) and deleted at another end (called the front)
Stack is Last-in-First-out system	The queue is the First-in-First-out system

Queue Representation with Array

The array can be used to implement a queue of fixed size, therefore only fixed a number of data items can be inserted and deleted. Consider the following example, the queue of size 5, therefore, maximum only 5 data items can be inserted. The front index always keeps track of the last deleted item from the queue and rear index always keep track of the last inserted item in the queue. Initially, front and rear both are initialized by -1 (for the zero-based array) when there are no items in the queue, i.e. the queue is empty.

When a new item is inserted in the queue, the rear is incremented by 1 before the item is stored in the queue. Once all the five items are inserted, then rear is 4, as shown in the figure 6.4. Now, if we try to insert next item, it leads to an overflow condition. It indicates that the queue is full and we cannot insert the new item.

Algorithm to insert an item to rear of a queue by using an array

Algorithm: ENQUEUE (Q, ITEM)

[Q is an array represent queue and ITEM is deleted item]

1. [check overflow]
 - a) If Rear = MAX - 1 then
 - a) Print: Queue is Full
 - b) Return
2. Set Rear = Rear + 1
3. Q[Rear] = ITEM
4. Return

When an item is deleted from the queue, the front is incremented by 1, before the item is removed from the queue. Now, if front = rear then if we try to delete an item, it results in underflow condition. It indicates that the queue is empty and we cannot delete an item. Whenever the queue is found empty, then to reuse the empty slots at the front of the queue we can reset the front and rear by -1.

Algorithm to delete from the front of a queue by using an array

Algorithm: DEQUEUE (Q, ITEM)

[Q is an array represent queue and ITEM is inserted item]

1. [Check underflow]
 - a) If Rear = Front then
 - a) Print: Queue is Empty
 - b) Return
2. Set Front = Front+1
3. ITEM = Q[Front]
4. If Rear = Front then

```

Set Front = Rear = -1
5. Return

```

Therefore, it is required to check overflow and underflow conditions whenever insertions and deletions operations take place.

[0]	[1]	[2]	[3]	[4]

Front = Rear = -1
(a) Initially empty

[0]	[1]	[2]	[3]	[4]
15	20	5	30	25

Front = -1, Rear = 4
(d) Insert 10, Overflow

[0]	[1]	[2]	[3]	[4]
15	20	5		

Front = -1, Rear = 2
(b) Insert 15, 20, 5

[0]	[1]	[2]	[3]	[4]
			30	25

Front = 2, Rear = 4
(e) Delete 15, 20, 5

[0]	[1]	[2]	[3]	[4]
15	20	5	30	25

Front = -1, Rear = 4
(c) Insert 30, 25

[0]	[1]	[2]	[3]	[4]
			30	25

Front = 2, Rear = 4
(f) Insert 10, Overflow

Figure 6.4: Different operations on Queue

Queue Representation with Linked List

Singly linked list can be used to represent a queue, which is also known as **Linked Queue**. In this representation, any number of data items can be inserted and deleted. The front and rear pointers always keep track of the first node and the last node in the linked list respectively. Initially, front and rear are initialized by null (i.e. front = rear = null), when there are no items in the queue, that means the queue is empty. The linked list header acts as the front of the queue. All deletion operations take place at the front of the list. All insertion operations take place at the end of the list. If the queue contains a single element then front and rear points to head/new node (i.e. front = rear = head).

When a new item is inserted in the queue, a new node is inserted at the end of the linked list, the rear points to the new node. When an item is deleted from the queue, the node from the front of the queue is deleted. Now, if front = null then if we try to delete an item, it results in underflow condition. It indicates that the queue is empty and we cannot delete an item. Whenever the queue is found empty, we can reset the front and rear by null.

Algorithm to insert an item to rear of a queue using linked list

Algorithm: ENQUEUE (FRONT, REAR, ITEM)

[FRONT points to the first node and REAR points to the last node of the linked list. ITEM is the inserted value]

1. Allocate memory for NEW node.
2. If NEW = NULL then Print: Out of memory and Return
3. Set NEW → DATA = ITEM
4. Set NEW → LINK = NULL
5. If REAR = NULL then
 - Set FRONT = REAR = NEW
- Else

```

    Set REAR → LINK = NEW
    Set REAR = NEW
    [End of If]
6. Return

```

Algorithm to delete from the front of a queue using linked list

Algorithm: DEQUEUE (FRONT, REAR, ITEM)

[FRONT points to the first node and REAR points to the last node of the linked list. ITEM is the deleted value]

1. If FRONT = NULL then Print: Stack is underflow and Return
2. Set P = FRONT
3. Set ITEM = P → DATA
4. Set FRONT = P → LINK
5. If FRONT = NULL then Set REAR = NULL
6. Set P → LINK = NULL
7. De-allocate memory for node P
8. Return

Comparisons of queer representation using linked list over the array

- The array is fixed size, therefore, a number of elements will be limited in the queue. Since linked list is dynamic and can be changed easily, so the number of elements can be changed.
- The pointers in linked list consume additional memory compared to an array.
- In array implementation, sometimes dequeue operation not possible, although there are free slots. This drawback can be overcome in linked list representation.
- In array and linked list enqueue and dequeue operations can be done in O (1).

Application of Queue

- A major application of the queue is in simulation [see Kruse for example].
- In operating systems, queues are used for process management, I/O request handling, etc.
- **Examples:** Print queue of DOS, Message queue of Unix IPC.
- Queues are also used in some elegant algorithms like graph algorithms (breadth first search), radix sort etc.
- Different types of customer service software are designed using a queue for proper service to the customer.

Example: Railway ticket reservation system

Drawbacks of Linear Queue

The linear queue, when represented using an array, suffers from drawbacks. Once the queue is full, even though few elements are deleted from the front end and some free slots are created, it is not possible to insert new elements, as the rear has already reached the queue's rear most position. Consider the figure, the queue of size 5 and the front is 2, rear is 4. Now, we are not able to insert new data item into the queue, although there are free slots (first and second location) in the front of the queue, because of rigid rule followed by linear queue (insertion can be done at the rear end of the queue). It is also known as a boundary case problem.

This drawback can be overcome in two different ways. The first solution is by left shifting all elements after every deletion. However, this is not suitable since after every deletion, the entire elements required shifting left and front and rear should be readjusted according to that.

The second solution is by implementing a circular queue and it is a suitable method to overcome the above drawback.

CIRCULAR QUEUE

A circular queue (also known as a circular buffer) is a linear data structure that uses a single, fixed-size buffer as if it were connected end-to-end. A circular queue is just one of the efficient ways to implement a queue. It also follows First-in-First-out (FIFO) principle.

Circular Queue Representation with Array

In array representation, the queue is considered as circular queue when the positions 0 and MAX-1 are adjacent. It means when rear (or front) reaches MAX-1 position then increment in rear (or front) causes rear (or front) to reach the first position that is 0.

One solution of the problem is if not all the elements of the array can be used to accommodate queue elements; in particular, an array of size n can accommodate a maximum of $n - 1$ elements and one of the slot always remains unused.

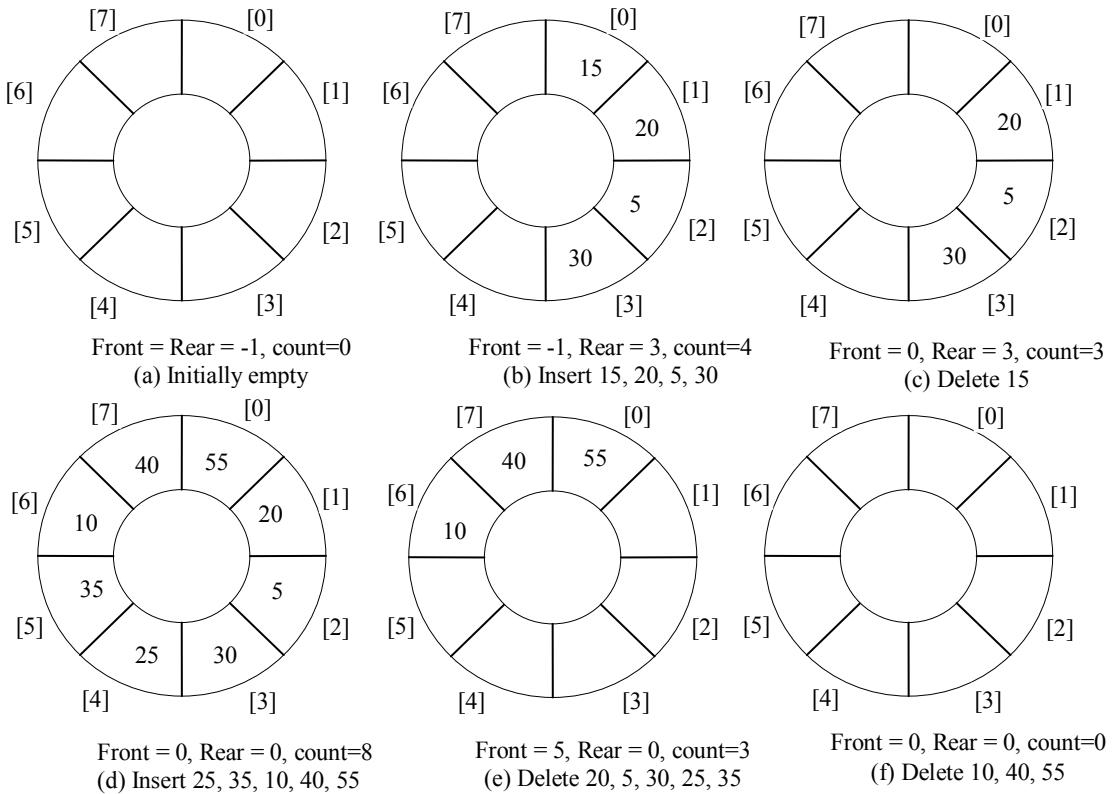


Figure 6.5: Different operations on Circular Queue

Consider the circular queue with $n = 8$ in figure 6.5.

Another solution is that if an extra variable counter is used for the identification of the empty queue and full queue. At first, the variable counter initialized by zero. When an element is inserted into the circular queue, the counter is increased by one and when an element is removed from the circular queue, the counter is decreased by one. Now, if the value of the counter is zero that means, a circular queue is empty and if the value of the counter equals to n (the size of the array) then the circular queue is full.

Note: Circular Queue's circularity is only logical. There cannot be a physical circularity in main memory.

Algorithm to insert an item to the rear of a circular queue.

Algorithm: ENQUEUE (Q, ITEM)

[Q is an array represent circular queue and $ITEM$ is inserted item]

1. [Check for overflow]
 - a) If Count = MAX then
 - a) Print: Queue is Full
 - b) Return
 2. Set Rear = (Rear + 1) mod MAX
 3. Set Q[Rear] = ITEM
 4. Set Count = Count + 1
 5. Return

Algorithm to delete from the front of a circular queue

Algorithm: DEQUEUE (Q, ITEM)

[Q is an array represent queue and $ITEM$ is deleted item]

1. [Check for underflow]
 - a) If Count = 0 then
 - a) Print: Queue is Empty
 - b) Return
 2. Set Front = (Front + 1) mod MAX
 3. ITEM = Q[Front]
 4. Set Count = Count - 1
 5. Return

DEQUEUE

A Double-ended Queue (Dequeue often abbreviated to Deque) is a linear list that generalizes a queue, for which elements can be inserted or deleted from either the front end or from the rear end but not in the middle. It is also often called a head-tail linked list.

There are two variations of a Dequeue

- i) **Input-restricted Dequeue:** Input-restricted dequeue is dequeue, where insertions can be made at only one end of the list but deletions can be made from both ends of the list.
- ii) **Output-restricted Dequeue:** Output-restricted dequeue is dequeue, where deletions can be

made from only one end of the list but insertions can be made at both ends of the list.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of dequeue and can be implemented using dequeue.

Note: Deque is sometimes written dequeue, but this use is generally deprecated in technical writing because dequeue is also a verb meaning, "to remove from a queue".

Operations on Dequeue

The operations are supported by Dequeue as follows:

Table 6.5: Operations on Dequeue

Operation	Description
Insertion at Rear	This operation inserts the element into dequeue at rear end.
Insertion at Front	This operation inserts the element into dequeue at front end.
Deletion from Rear	This operation removes an element from dequeue from rear end.
Deletion from Front	This operation removes an element from dequeue from front end.

PRIORITY QUEUE

A Priority Queue is a collection of elements such that each element has been assigned a priority and the elements are arranged on the basis of priority. The order in which elements are deleted and processed comes from the following rules:

- i) The element having a higher priority is processed before any elements of lower priority.
- ii) The two elements that have the same priority are processed according to the order in which they are inserted into the priority queue.

There are two types of priority queue:

- i) **Ascending priority queue:** In this type of the priority queue, the elements can be inserted arbitrarily, but only the element with the smallest priority can be removed.
- ii) **Descending priority queue:** In this type of the priority queue, the elements can be inserted arbitrarily, but only the element with the highest priority can be removed.

Here we discuss the operations and algorithms for descending priority queue.

Abstract Data Type

A priority queue is an abstract concept. A priority queue can be implemented with an array, a linked list or a heap. Priority Queue supports the operations as follows:

Table 6.6: Operations on Priority Queue

Operation	Description
Insert	This operation inserts the element into the priority queue with an associated priority
Peek	This operation returns the element that has the highest priority.
Extract-Maximum or Remove	This operation removes the element from the priority queue that has the highest priority.

Representation of Priority Queue

The priority queue can be represented using a (unsorted or sorted) linked list or array. For an unsorted array or for a linked list, insertion operation is done at the end or the head of the linked list, or at the end of the array, therefore, it runs in $O(1)$. In the peek operation, to return the highest-priority element, it required searching (linear search) the highest-priority element in the entire array or the linked list, therefore it runs in $O(n)$. To remove the highest-priority element, it required searching (linear search) the highest-priority element in the entire array or the linked list. In an array, we also have to shift array contents to fill the gap. Therefore, it runs in $O(n)$.

However, a linked list or unsorted array are great for inserting elements, but not good at searching high-priority elements (or removing them). An alternative is to sort the array (or List or linked list) according to the priority, with the highest priority at the end of the array. For a sorted array or for a linked list, in insertion operation at first it needs to search for the correct position using binary search in an array ($O(\log n)$) and linear search in a linked list ($O(n)$). Once the correct position is found, insert the element thereby keeping the entries in the array or linked list in order (as in insertion sort). In an array, this involves shifting all elements to the right of the inserted element over by one position. Therefore, in either case, the cost is $O(n)$. In peek operation, to return the highest-priority element, it required returning the last element of the array or linked list as the highest-priority element is always at the end. Therefore, it runs in $O(1)$. To remove the highest-priority element, it required removing the last element of the array or linked list as the highest-priority element is always at the end. In the array, by decreasing the size counter. Therefore, it takes $O(1)$.

A sorted array or linked list is fast at looking up and removing high-priority elements, but pays with linear insertion cost.

Priority Queue Representation with Heap

The priority queue can be represented efficiently using a max-heap; the representation would be as follows:

Insert Operation

Since we maintain the property of the complete binary tree, insert the element as a new leaf, as far to the left as possible, i.e. at the end of the array; increment the size of the heap. After insertion, it may violate the heap property when the newly added element has higher priority than its parent. Therefore, to restore the heap condition shift-up through the heap with that element. The running time of the insert operation on a n element heap is $O(\log n)$.

Algorithm to insert an item to Priority Queue

Algorithm: Insert (Q, N, ITEM)

[Q is an array represent priority queue, N is the number of items and $ITEM$ is an item to be inserted into priority queue]

1. Set $N = N + 1$
2. Set $I = N$ and $J = I/2$
3. Repeat step 4 and 5 while $I > 1$ and $Q[J] < ITEM$
4. Set $Q[I] = Q[J]$
5. Set $I = J$ and $J = I/2$

```

[End of loop]
6. Set Q[I] = ITEM
7. Return

```

Peek Operation

To return the highest-priority element, it required returning the root element of the heap as the highest-priority element will always be at the root of the max-heap. Therefore, it runs in $O(1)$.

Remove Operation

To remove the highest-priority element, it required removing the top element of the heap; decrement the size of the heap, and then shift-down through the heap with that item to restore the heap condition. The running time of remove operation on an n element heap is $O(\log n)$.

Algorithm to delete an item from Priority Queue

Algorithm: Remove (Q, N, ITEM)

[Q is an array represent priority queue, N is the number of items and $ITEM$ is an item to be removed from priority queue]

```

1. [check whether queue is empty]
   If N < 1 then Print: Queue Underflow and Return
2. Set ITEM = Q[1]
3. Set Q[1] = Q[N]
4. Set N = N - 1
5. Call Heapify(Q, 1)
6. Return

```

In the following table (Table 6.7) the time complexity of different operations on priority queue are given where different data structures are used. From the table, it is observed that the time complexity of Heap data structure is the best choice for implementing a priority queue.

Table 6.7: Time complexity of different operation with different representation

Operation	Priority queue representation with		
	Unsorted array or linked list	Sorted array or linked list	Heap
Insert	$O(1)$	$O(n)$	$O(\log n)$
Peek	$O(n)$	$O(1)$	$O(1)$
Remove	$O(n)$	$O(1)$	$O(\log n)$

Summary

- A stack is an ordered collection of elements into which new elements may be inserted and from which elements may be deleted only at one end called the top of the stack.
- A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

6. 20 | Data Structures and Algorithms with C

- A Double-ended Queue (Dequeue often abbreviated to Deque) is a linear list that generalizes a queue, in which elements can be inserted or deleted from either the front end or from the rear end but not in the middle.
- A Priority Queue is a collection of elements such that each element has been assigned a priority and the elements are arranged on the basis of priority.

Exercises

1. What is a queue? Write an algorithm to insert an element in such a queue.
2. Why is the queue data structure called the FIFO?
3. Define circular queue. Write an algorithm to insert an item in the circular queue.
4. What are the disadvantages of the linear queue? How can we overcome these disadvantages in case of the circular queue? Explain with an example.
5. What is priority queue? Implement the operations of the priority queue.
6. What is input restricted dequeue?
7. Write an algorithm to convert an infix expression to its corresponding postfix expression, using the stack.
8. Write the differences between stack and queue.
9. Write short notes on Dequeue- operations and applications.
10. Evaluate following expression.
 - a. $10+3-2-8/2*6-7$
 - b. $(12-(2-3)+10/2+4*2)$
11. Convert following infix expression to postfix expression:
 - a. $((a+b)/d-((e-f)+g))$
 - b. $12/3*6+6-6+8/2$
12. Convert following infix expression to prefix expression:
 - a. $((a+b)/d-((e-f)+g))$
 - b. $12/3*6+6-6+8/2$
13. Explain application of Stack.
10. Choose the correct alternative in each of the following:
 - i) Reverse Polish notation is often known as
 - a) Infix
 - b) Prefix
 - c) Postfix
 - d) none of these
 - ii) The postfix equivalent of the prefix $*+ab-cd$ is
 - a) $ab+cd-*$
 - b) $abcd+-*$
 - c) $ab+cd*-$
 - d) $ab+-cd^*$
 - iii) The following sequence of operations is performed on a stack: push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop. The sequence of popped out values are:
 - a) 2, 2, 1, 1, 2
 - b) 2, 2, 1, 2, 2
 - c) 2, 1, 2, 2, 1
 - d) 2, 1, 2, 2, 2
 - iv) The initial configuration of queue is a, b, c, d ('a' is at the front). To get the configuration d, c, b, a one needs a minimum of
 - a) 2 deletions and 3 additions
 - b) 3 deletions and 2 additions
 - c) 3 deletions and 3 additions
 - d) 3 deletions and 4 additions
 - v) A linear list that allows elements to be added or removed at either end but not in the middle is called:
 - a) Stack
 - b) Dequeue
 - c) Deque
 - d) Priority queue

- vi) If we evaluate the following post-fix expression, $2\ 3\ 5\ 7\ *\ -12\ +$, the result will be
 a) 12 b) 0 c) -12 d) 35
- vii) The evaluation of the postfix expression $3\ 5\ 7\ *\ +\ 12\ %$ is
 a) 2 b) 3 c) 0 d) 3.17
- viii) The integers 1, 2, 3, 4 are pushed into the stack in that order. They may be popped out of the stack in any valid order. The integers, which are popped out produce a permutation of the members 1, 2, 3, 4. Which of the following permutation can never produce in such a way?
 a) 1, 2, 3, 4 b) 4, 2, 3, 1 c) 4, 3, 2, 1 d) 3, 2, 4, 1
- ix) The prefix expression for the infix expression $a\ *(b+c)/e-f$ is
 a) / *a+bc-ef b) -/*+abcef c) -/*a+bcef d) none of these
- x) A stack is implemented using an array with the following declaration:
`int stack[100]; int stack_top=0;`
 To perform the POP operation, which of the following is correct?
 a) $x = stack[stack_top++]$ b) $x = stack[stack_top]$
 c) $x = stack[stack_top--]$ d) $x = stack[--stack_top]$
- xi) Translating the infix expression $P = A+(B*C->D/(E+F)*G)*H$ into postfix notation, we get,
 a) ABC*DEF/+G*-H*+ b) ABC*+DEF/-G*H—
 c) ABC*DEF/+G*-H*+ d) None of these
- xii) The number of stacks required to implement mutual recursion is
 a) 3 b) 2 c) 1 d) none of these
- xiii) Queue can be used to implement?
 a) Radix sort b) Quick sort c) Recursion d) Depth first search
- xiv) Stack is useful for implementing
 a) Radix sort b) Recursion c) Breadth first search d) Depth first search
- xv) The postfix expression for the infix expression $A+B*(C+D)/F+D^*E$ is
 a) AB+CD+*F/D+E* b) ABCD+*F/+DE*+
 c) A*B+CD/F*DE++ d) A+*BCD/F*DE++
- xvi) Stack is sometimes called a _____
 a) Push down list b) Pushdown array c) Pop down list d) Pop up array
- xvii) The prefix expression for the infix expression : $a+b*c/d$
 a) $+ab*/cd$ b) $+*ab/cd$ c) $+a*b/cd$ d) none
- xviii) Which of the following is not the operation on stack?
 a) Push b) Pop c) Peep d) Enqueue
- xix) Which of the following is related to Queue?
 a) Round Robin b) Traffic Control c) All algorithm d) None
- xx) Which of the following is not a application of Stack?
 a) Evaluation of Police notation b) Tower of Hanoi c) Stack Machine d) None

CHAPTER 7

RECURSION

“Intelligence is the ability to avoid doing work, yet getting the work done” -Linus Torvalds

There are two approaches for writing repetitive algorithms. One uses iteration/loop and the other uses recursion. Iteration is one of the categories of control structures. It allows for the processing of some action many times. Iteration is also known as looping and repetition. The math term "to iterate" means to perform the statement parts of the loop. Many problems/tasks require the use of repetitive algorithms. Recursion is defined as defining anything in terms of itself.

With most programming languages, this can be done with either:

- i) Looping control structures, specifically the for loop (an iterative approach)
- ii) The recursive calling of a function

Both approaches provide repetition, and either can be converted to the other's approach.

Definition: Recursion is a repetitive process in which a function calls itself either directly or indirectly.

In the other words, recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem.

In recursion, each time when the recursive function is called, the current state including all the local variables, formal parameters, and return address are pushed into the stack. A block of memory of contiguous locations set aside for this purpose. When the return statement is encountered, the control comes back to the previous function call, by restoring the state that resides on the stack.

Recursion Essentials

There are two key requirements to make sure that the recursion is successful:

- A recursion definition must always have certain criteria, called **base criteria** or base case for which the function stop calling itself.
- Every recursive call must simplify the computation in some way, which is known as an **inductive clause**. In other words, each time a function does call itself (directly or indirectly), it must be closer to the base criteria that terminates the recursion.

Infinite Regress

When the recursion does not terminate, we have an infinite regress. Infinite regress often occurs when:

- i) A base criterion is omitted or
- ii) When a base criterion is never reached.

KEY FEATURES

-
-  Recursion essentials
 -  Types of Recursion
 -  Recursion tree
 -  Tower of Hanoi
 -  Non-attacking 8 queens
-

A circular definition may have no base criteria and define the value of a function in terms of that value itself, rather than on other values of the function. Such a situation would lead to an infinite regress.

Note: If infinite recursion occurs in a program then you may get a runtime error message, e.g. “*Stack overflow*”.

Depth of Recursion

Suppose P is a recursive function. During the execution of a program, which contains P a level number is assigned. The original execution of P is assigned level 1 and each time P is executed because of the recursive call, its level is one more than the level of execution that had made the recursive call.

The depth of the recursive function P, with given set of arguments, refers to the maximum level number of P during its execution.

Recursion Tree

A recursion tree is a tree that is generated by tracing the execution of a recursive function. A recursion tree shows the relationship between calls to the function. Recursion tree is a pictorial representation of recursion call, which is in the forms of a tree, where at each level nodes are expanded. Each node represents how recursive function calls are generated. Descendants of a function call are further recursive calls. Calls in the tree with no descendants involve evaluation of the base case(s). Example of recursion tree is shown in figure 7.2.

Types of Recursion

The ways in which recursive functions are characterized are:

- Whether the function calls itself or not (direct or indirect/mutual recursion).
 - How many internal recursive calls are made there within the body (linear, binary, and non-linear recursion)?
 - Whether there are pending operations or not at each recursive call (tail or non-tail recursion).
- Recursion is mainly two types depending on whether a function calls itself or not.
- i) Direct Recursion
 - ii) Indirect Recursion or Mutual Recursion

Direct Recursion

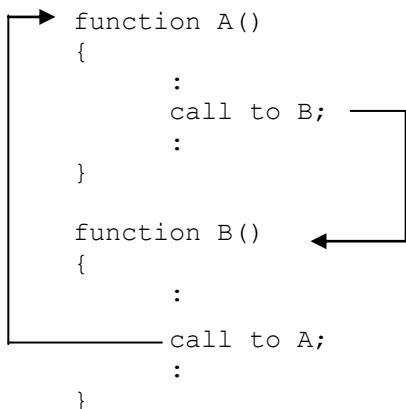
In direct recursion, a function calls itself from the body of the function. A function is directly recursive if it contains an explicit call to itself. The format of direct recursion is as follows:

```
→ function F()
{
    if base condition satisfied
        then return some value(s)
    else
    {
        :
        call F();
        :
    }
}
```

Examples: Factorial, Fibonacci, GCD, etc.

Indirect Recursion

In indirect recursion, two functions call one another directly. A function is indirectly recursive if it contains a call to another function, which ultimately calls it. Indirect recursion also is known as mutual recursion. The format of indirect recursion is as follows:



Examples: Recursive Descent Compilation, Hilbert Curves, etc.

Recursion is categorized into three types depending on the number of recursion calls:

- i) Linear Recursion
- ii) Binary Recursion
- iii) Non-Linear or Multiple Recursion

Linear Recursion

In linear recursion, a single recursion call is performed. A recursive function in which only one internal recursive call is made within the body is called linearly recursive. The format of linear recursion is as follows:

```

function L ()
{
    if base condition satisfied
        then return some value(s)
    else
    {
        :
        call L();
        :
    }
}
  
```

Examples: Factorial, GCD, Binary Search, etc.

Binary Recursion

In binary recursion, two recursion calls are performed. A function that makes two internal calls to itself is said to be binary recursive. The format of binary recursion is as follows:

```

function B (...)

{
    if base condition satisfied
        then perform actions/ return value(s)
    else
    {
        perform some action(s);
        make a call to B to solve one smaller problem ;
        make a call to B to solve the other smaller problem ;
    }
}

```

Examples: Fibonacci sequence, Quick sort, Merge sort, Binary Tree algorithms, General Divide-and-conquer algorithms.

Non-linear recursion

In non-linear recursion, more than two recursion call are performed. A function using a number of more than two internal recursive calls within the body of the function is non-linear recursive. Non-linear recursion also is known as multiple recursion. The format of non-linear recursion is as follows:

```

function N(...)

{
    for j= k to n do
    {
        perform some action(s);
        if base condition not satisfied
        then make a call to N
        else perform some action(s);
    }
}

```

Examples: Non-attacking 8-queens, Sample Generation, Combination generation, Permutation generation, etc.

Tail Recursion

A special form of recursion where the last operation of a function is a recursive call. That is a recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

Example:

```

int fact (int n)
{
    if(n==1)
        return 1;
    else
        return n * fact(n-1)
}

```

Notice that there is pending operation namely multiplication, to be performed on return from each

recursive call. Whenever there is a pending operation, the function is non-tail recursive.

The factorial function can be written in a tail-recursive way:

```
int fact_aux(int n, int result)
{
    if (n==1)
        return result;
    else
        return fact_aux(n-1, n*result)
}
int fact(n)
{
    return fact_aux(n, 1);
}
```

Here `fact_aux` is a recursive function, not `fact`.

Advantages of Recursion

- i) Recursion functions can be written easily. We can write a simple version of programs using recursion.
- ii) There are some complex problems such as Tower of Hanoi, Non-attacking Queen, etc. can be easily understood and implemented using recursion.
- iii) The recursive definitions can be translated into recursive function easily.

Disadvantages of Recursion

- i) Recursion consumes more storage space because each time a new recursive call is made; a new memory space is allocated to each automatic variable used by the recursive function.
- ii) The computer may run out of memory if base criteria are not checked.
- iii) Recursion is not efficient in terms of speed and execution time, as function calls require storing the current state of the function onto the stack, jump to execute recursion call, restoring state from the stack.

Table 7.1: Differences between iteration and recursion

Iteration	Recursion
Iteration is a process of executing a block of statements repeatedly until some specific condition.	Recursion is a technique that breaks down a problem into one or more subproblems that are similar to the original problem.
The iterative process is more efficient in terms of storage space and execution time.	Recursion is not efficient in terms of storage space and execution time.
Any recursive problem can be solved iteratively.	Not all problems have a recursive solution.
Iteration process sometimes not easy to implement. Complicated problems are difficult to solve in iteratively; e.g. tower of Hanoi problem, eight queen problem etc.	Recursive functions are easier to implement and maintain. Complicated problems are solved easily.
Iteration has four steps: initialization, condition checking, execution statements and updating.	A recursive function must have base criteria for which the function does not call itself. Each time a function does call itself (directly or indirectly); it must be closer to the base criteria.

When should we use iteration, and when use recursion? There are three factors to consider:

- i) Iterative functions are typically faster than their recursive alternatives. Therefore, if speed were an issue, you would normally use iteration.
- ii) If the stack limit is too constraining then you will prefer iteration over recursion.
- iii) Some procedures are very naturally programmed recursively, and all but unmanageable iteratively. Here, the choice is clear.

Factorial

Factorial of a number n is the product of the positive integer from 1 to n.

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

Factorial of number can be defined in recursively as follows:

$$\text{Factorial}(n) = \begin{cases} 1, & \text{when } n = 0 \text{ or } 1 \\ n \times \text{Factorial}(n-1), & \text{when } n > 1 \end{cases}$$

Function to compute the factorial of a given number using recursion

```
Function: INT FACTORIAL (N)
[N is a number]
1. [Base condition]
   If N < 2
      Then Return 1
2. Else
   [Call itself]
   Return (N*FACTORYAL (N-1))
```

The above function obtains the factorial of a given number n in a recursive manner. If the number is one then factorial is computed otherwise for one the function returns 1.

The above function fact is a recursive function because it calls itself; namely, factorial (n) is calling fact (n -1).

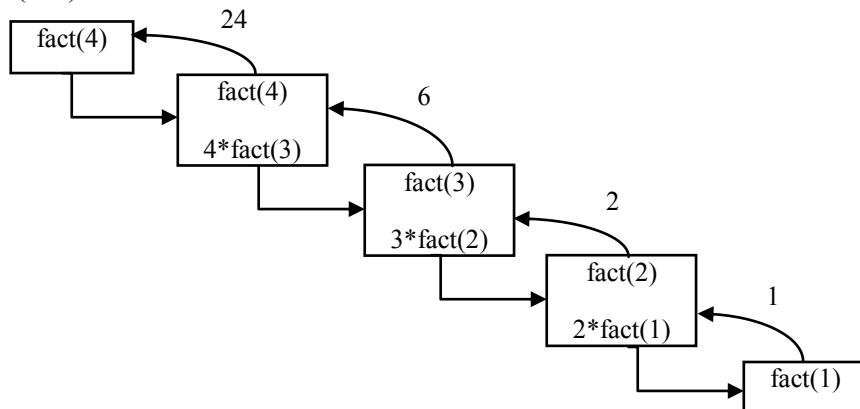


Figure 7.1: Recursive Function call processing

Therefore, if the function factorial is called by a value say 4, the function call factorial (4) invokes itself with the value 3. The function call factorial (3) invokes factorial (2), the factorial (2) invokes factorial (1). Then we have no further function call because it has reached the base condition.

Therefore, in Stack first fact (3) is pushed, then fact (2) and at last, the fact (1) is pushed. Moreover, these function calls are processed according to the sequence they are popped from the stack.

Fibonacci sequence

Fibonacci sequence is a series of positive number in a manner that the next term of the series is the addition of two previous terms:

0 1 1 2 3 5 8 13 21 34....

Fibonacci series can be defined in recursively as follows:

$$\text{Fib}(n) = \begin{cases} 0, & \text{when } n = 1 \\ 1, & \text{when } n = 2 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & \text{when } n > 2 \end{cases}$$

A function to generate a Fibonacci number using recursion

Function: INT FIB(N)

1. [Base condition]
If N = 1
Then Return 0
2. [Base condition]
If N = 2
Then Return 1
3. [Call itself]
Return (FIB (N-1) + FIB (N-2))

The above function for obtaining a Fibonacci number of a given number n in a recursive manner. If the number is greater than 2 then Fibonacci number is calculated otherwise for one or two the function returns 0 or 1 respectively.

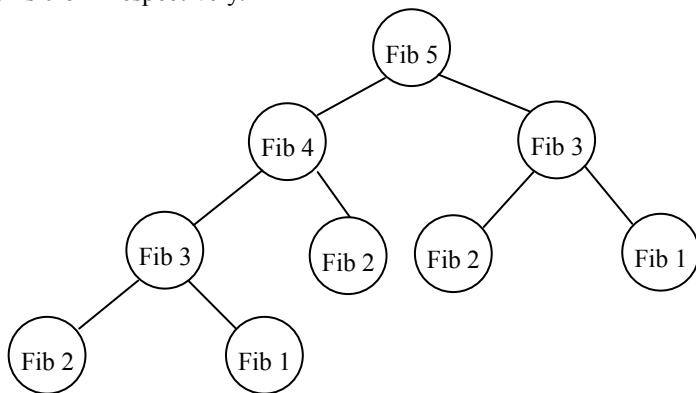


Figure 7.2: Recursion Tree for Fib (5)

The recursion tree for obtaining the Fibonacci number of a given number 5. Finally, the function adding all values of the leaf nodes.

$$\text{FIB}(5) = \text{FIB}(2) + \text{FIB}(2) + \text{FIB}(1) + \text{FIB}(2) + \text{FIB}(1) = 1 + 1 + 0 + 1 + 0 = 3$$

Therefore, the function calculated 5th term of the Fibonacci series, which is 3.

GCD

The greatest common divisor (GCD) or highest common factor (HCF) of two non-negative, not- both-zero integers a and b , denoted by $\text{GCD}(a, b)$, is defined by the greatest integer that divides both a and b evenly, that is with no remainder.

GCD can be defined by Euclid's algorithm on recursively as follows:

$$\text{GCD}(a, b) = \begin{cases} a, & \text{when } b = 0 \\ \text{GCD}(b, a \bmod b), & \text{otherwise} \end{cases}$$

where $a \bmod b$ is the remainder of the division of a by b , until b is equal to zero and final value of a is the greatest common factor of the first value a and b .

Function to find out GCD of two numbers using recursion

Function: INT GCD(A, B)

1. [Base Condition]
If $B = 0$
then Return A
2. [Call itself]
Else
Return $\text{GCD}(B, A \bmod B)$

In the above algorithm, there is no need to check whether A is greater than B or not.

For example, $A = 4$ and $B = 6$ then the $\text{GCD}(A, B)$ function is called in the following sequence:

$$\text{GCD}(4, 6) \rightarrow \text{GCD}(6, 4) \rightarrow \text{GCD}(4, 2) \rightarrow \text{GCD}(2, 0)$$

and when $A = 6$ and $B = 4$ then the $\text{GCD}(A, B)$ function is called in the following sequence:

$$\text{GCD}(6, 4) \rightarrow \text{GCD}(4, 2) \rightarrow \text{GCD}(2, 0)$$

Therefore, when B is greater than A , the $\text{GCD}(A, B)$ function is called just one more time.

This algorithm runs very fast, but division (taking remainders) is a more time-consuming operation than simple addition and subtraction.

Integer Power

Integer power problem is to find x^n where x is real and n is a positive integer, using a minimal number of multiplications. The naïve one-at-a-time algorithm required n number of repeated multiplication.

$$x^n = \underbrace{x \times x \times \dots \times x}_{n \text{ times}}$$

n times

We can call this as “ x to the power of n ” or “ x raised to the power of n ”. Here, x is the base and n is the exponent or the power.

Now, x^n can be defined recursively as follow:

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot x^{n-1}, & \text{Otherwise} \end{cases}$$

Function to find out power using recursion

```
Function: INT Power(X, N)
[X is the base and N is the exponent]
1. [Base Condition]
   If N=0 Return 1
2. [Check whether negative exponent]
   Else If N<0 then Call Power(X, N+1) / X
3. Else Call Power(X, N-1)*X
   [End of If]
4. Return
```

Power by the squaring method can minimize the number of multiplications. Power by the squaring method can be defined on recursively as follows:

$$x^n = \begin{cases} x \cdot (x^2)^{(n-1)/2}, & \text{if } n \text{ is odd} \\ (x^2)^{n/2}, & \text{if } n \text{ is even} \end{cases}$$

When n is an even number then

$$x^n = x^{2z} = (x^2)^z, \text{ where } n = 2z,$$

When n is an odd number then

$$x^n = x^{2z+1} = x \cdot (x^2)^z \text{ where } n = 2z+1.$$

Function to find out power using recursion

```
Function: INT Power(X, N)
1. If N=0 then Return 1
2. If N<0 then Call Power(1/X, -N)
3. Set P = 1
4. If N mod 2 = 1 then P = X
5. P = P * Power(X*X, N/2)
6. Return P
```

By using this method, we can minimize the multiplication operations. This algorithm uses $\lfloor \log_2 n \rfloor$ squares and at most $\lfloor \log_2 n \rfloor$ multiplications. For instance, when $n=15$, the above algorithm required 3 squares and 3 multiplications, 6 multiplications in total. However, it does not always do the fewest possible number of multiplications.

Tower of Hanoi

Tower of Hanoi problem has a historical basis in the ritual of ancient Vietnam. This Problem is a classic example of the application of recursion. The problem is stated as follows:

Suppose, there are three towers A, B, C. There are n disks of decreasing size so that no two disks are of the same size. Initially, all the disks are stacked on one tower in their decreasing order of size. Let this tower be A. Other two towers are empty. The problem is to move all the disks from one tower to other using third tower.

- i) Only one disk may be moved at a time.
- ii) Only the top disc on any tower may be moved to any other tower.
- iii) A large disk cannot be placed on a smaller one.

A solution of this problem can be stated recursive as follows:

Move n disks from tower A to C through the tower B using three steps

- i) Move first (n - 1) disks from tower A to B
- ii) Move the nth disk from tower A to C.
- iii) Move all (n - 1) disks from tower B to C.

Our objective is to find all possible moves to be performed to solve the problem.

Function to tower of Hanoi using recursion

Algorithm: Tower (A, B, C, N)

1. If N > 0 then
2. Call Tower (A, C, B, N-1)
3. Print: Move Nth disk from tower A to tower C
4. Call Tower (B, A, C, N-1)
5. Return

Example:

When a number of disks are 2, i.e. n = 2 we have the following moves.

- i) Move 1st disk from A to B
- ii) Move 2nd disk from A to C
- iii) Move 1st disk from B to C

So when a number of disks are 3, i.e. n=3 we have the following call tree:

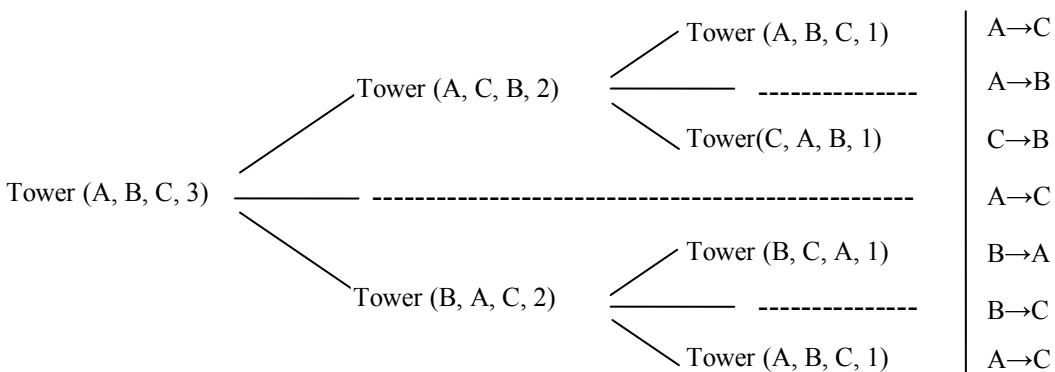
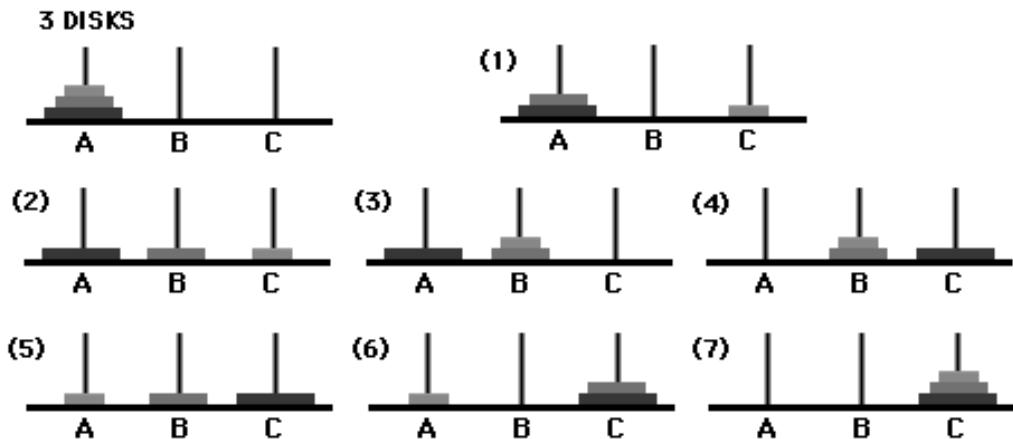


Figure 7.3: Recursion Tree for three disks

**Figure 7.4:** Disk Movements for three disks

Therefore, when $n = 3$ the moves are:

- Move 1st disk from A to C
- Move 2nd disk from A to B
- Move 1st disk from C to B
- Move 3rd disk from A to C
- Move 1st disk from B to A
- Move 2nd disk from B to C
- Move 1st disk from A to C

Table 7.2: Total number of moves required for different number of disks

Number of disks	Number of moves required
1	$2^1 - 1 = 1$
2	$2^2 - 1 = 3$
3	$2^3 - 1 = 7$
4	$2^4 - 1 = 15$
5	$2^5 - 1 = 31$

Time Complexity

The time required by the procedure move is denoted by $T(n)$, where n is the number of disks.
Now,

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ 2T(n-1)+c, & \text{when } n>1 \end{cases}$$

Where c is a constant

$$T(n) = 2T(n-1) + c \quad (\text{when } n>1)$$

$$= 2[2T(n-2) + c] + c$$

$$\begin{aligned}
&= 2^2 T(n-2) + c(1+2) \\
&= 2^3 T(n-3) + c(1 + 2 + 2^2) \\
&\vdots \\
&= 2^r T(n-r) + c(1 + 2 + 2^2 + \dots + 2^{r-1}) \\
&= 2^{n-1} T(n-n+1) + c(1 + 2 + 2^2 + \dots + 2^{n-2}) \quad [\text{when } r=n-1] \\
&= 2^{n-1} T(1) + c(1 + 2 + 2^2 + \dots + 2^{n-2}) \\
&= 2^{n-1} \cdot c + c(1 + 2 + 2^2 + \dots + 2^{n-2}) \quad [\text{Hence, } T(1)=c] \\
&= c (1 + 2 + 2^2 + \dots + 2^{n-2} + 2^{n-1}) \\
&= c(2^n - 1) \\
&= O(2^n)
\end{aligned}$$

Non-Attacking 8 Queens

The problem is to place eight queens in such a manner on an 8-by-8 chessboard that no two queens attack each other. Remember that a queen can attack another if they lie on the same row or same column or on the same diagonal in either direction. This problem can be solved by using backtracking method. However, there is more than one solution of this problem.

Backtracking is a general algorithm technique for finding all possible solution for the problem and discards partial solution as soon as possible without following them till the end. Backtracking is a depth-first traversal of the path in the graph where nodes are states of the solution and edge between two states of solution only if one state can be reached from another state. Each path may lead to a solution, taking one path at a time and as soon as if the path does not lead to the solution then go back and try with the alternate path.

In a non-attacking n queen problem, where n is the number of queens/rows/columns, it can be seen that for n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2 and n = 3. Therefore, at first, we consider the four queens problem, solve it by the backtracking technique, and then generalized it to 8-queens problems and an n-queens problem.

Now start with the empty board and then place the first queen in the first possible position at the first row, which is in the first column. The second queen 2 at the second row cannot be placed on the first or second column but can be placed on the third column that is the first acceptable position for it.

After that, it reaches a dead end, because there is no acceptable position for the third queen. Therefore, the algorithm backtracks and puts the second queen in the next possible position at 2nd row 4th column. Then the third queen can be placed at 3rd row 2nd column, which reaches to another dead end. The algorithm then backtracks all the way to 1st queen replaced to 1st row 2nd column, 2nd queen replaced to 2nd row 4th column, 3rd queen placed to 3rd row 1st column, and 4th queen placed to 4th row 3rd column, which is a solution to the problem. The state-space tree of this search is shown in Figure 7.7.

A chessboard may be represented by a two-dimensional array. It is known that each row in the array contains exactly one queen in the solution configuration. Thus, a one-dimensional array is sufficient as a suitable data structure, where the index of the array is row number and the value of the array is column number. For example, A[i] = j means the queen is placed on ith row and of the jth column.

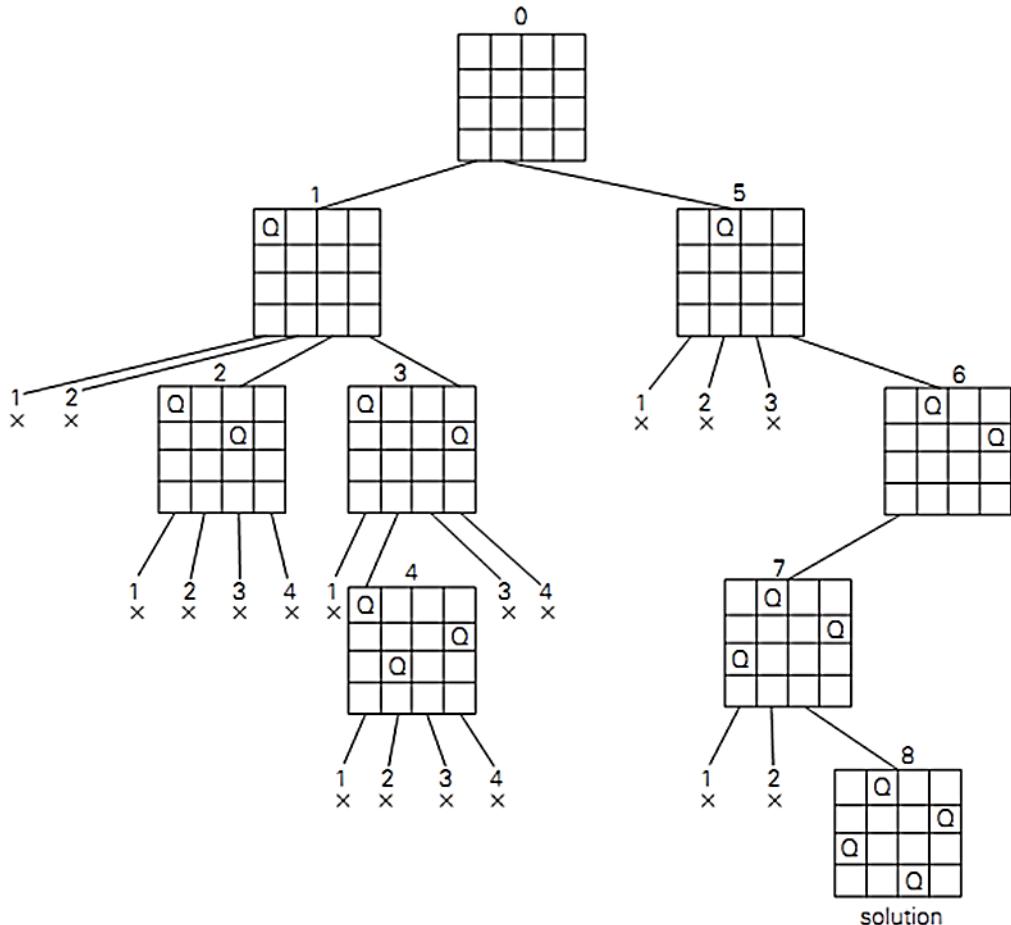


Figure 7.5: State-space tree of solving the four queens' problem by backtracking. \times denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

For a 8-queen problem, constructs a search tree by considering one row of the board at a time and eliminating most non-solution board positions at a very early stage in their construction. Now, start with the empty board and then place the first queen in the first possible position at the first row, that is first column, so set $A[1]$ to 1. The second queen at the second row cannot be placed on the first or second columns but can be placed on the third column, that is first acceptable position for it, so set $A[2]$ to 3. Now the next queen at the third row cannot be placed on 1st, 2nd, 3rd and 4th columns, however it can be placed on the 5th column; hence, set $A[3]$ to 5. Similarly, the fourth queen can be placed on the 2nd column, so $A[4]$ may be set to 2. The fifth queen can be placed on the 4th column, so $A[5]$ may be set 4. After that, it reaches a dead end, because it is impossible to place the queen in the 6th row. This situation is shown in figure 7.8.

Therefore, it is needed to revisit the position of the fifth queen in the previous row and place it in a different column. That means a backtracking is required to solve this problem.

	1	2	3	4	5	6	7	8
1	♛							
2	X	X	♛					
3	X	X	X	X	♛			
4	X	♛						
5	X	X	X	♛				
6	X	X	X	X	X	X	X	X
7								
8								

Figure 7.6: A deadlock situation of non-attacking 8-queen problem**Algorithm to solve Non-Attacking-Queen problem using recursion****Algorithm: Non-Attacking-Queen (ROW, N)**

1. Repeat steps 2 to 4 For COL = 1 to N do
2. If PLACE(ROW, COL) = True then
3. Set A[ROW] = COL
4. If ROW = N then
- Display Board Configuration
- Else
- Call Non-Attacking-Queen (ROW+1, N)
- [End of If]
- [End of If]
- [End of Loop]

5. Return

Function: PLACE(R , C)

1. Repeat steps 2 and 3 while For J=1 TO R-1 do
2. If A[J] = C then Return False
3. If ABS(A[J]-C)= R-J then Return False
- [End of Loop]

4. Return True

In the above algorithm, the initial value of ROW and N would be 1 and 8 respectively for the non-attacking 8-queen problem.

Before placing a queen in a cell, it is sufficient to check whether any of the placed queens attack it or not. The function PLACE(R, C) checks this. Note that a queen of position (R, C) may attack another queen place at cell (J, A[J]) of the chessboard if any of the following conditions is true:

i) The queens are placed on the same column. That is, $A[J]$ equals to C.

ii) The queens are placed on the either diagonal. That is, the value of $A[J] - C$ equals to $R - J$.

Suppose one queen already placed at (i, j) and another queen going to place at (m, n) cell, then they are on the same diagonal either

$$m - i = n - j$$

or

$$m - i = j - n$$

Therefore two queens lie on the same diagonal if and only if $m - i = |j - n|$.

Considering the set of combinations, we find that, there are 92 solutions ($11 \times 8 + 1 \times 4$) to the 8×8 problem. Many of these are reflections and rotations of some of the others, and there are only 12 unique solutions.

A solution configuration, shown in the figure where eight queens are placed on a chessboard in a manner that they cannot attack each other.

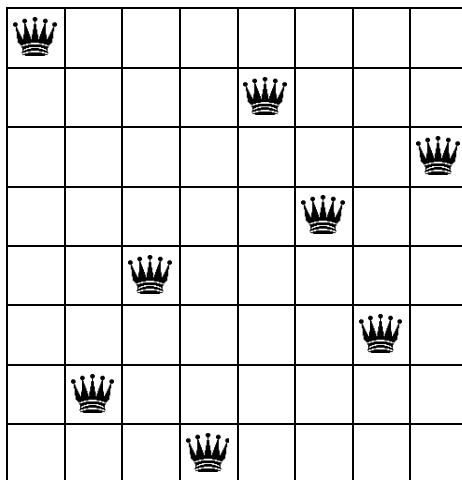


Figure 7.7: One Solution of Non-attacking 8-queen problem

Time Complexity

The time required by the procedure move is denoted by $T(n)$, where n is the number of queens/rows/columns. It can be seen that for $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$.

Now,

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ n*T(n-1) + c, & \text{when } n>1 \end{cases}$$

Where c is a constant

$$\begin{aligned} T(n) &= nT(n-1) + c \quad (\text{when } n>1) \\ &= n[(n-1)T(n-2) + c] + c \end{aligned}$$

$$\begin{aligned}
 &= n(n-1)T(n-2) + c[1 + n] \\
 &= n(n-1)(n-2)T(n-3) + c[1 + n + n(n-1)] \\
 &\vdots \\
 &= n(n-1)(n-2)(n-3)\dots 3.2.T(1) + c[1 + n + n(n-1) + \dots + n(n-1)(n-2)\dots 3] \\
 &= n(n-1)(n-2)(n-3)\dots 3.2.c + c[1 + n + n(n-1) + \dots + n(n-1)(n-2)\dots 3] \text{ [Hence } T(1) = c] \\
 &= c[1 + n + n(n-1) \dots n(n-1)(n-2)\dots 2] \\
 &= O(n!)
 \end{aligned}$$

Space complexity for this algorithm is $O(n)$. The algorithm uses an auxiliary array of length n to store just n positions.

Converting Recursive function into Iterative

The recursive function may be converted into iterative by using the following two steps:

Step 1: Converting Recursive Functions to be Tail Recursive

A non-tail recursive function can often be converted to a tail recursive function by means of an auxiliary parameter. This parameter is used to form the result.

Step 2: Converting Tail Recursive Functions to iterative

Let us assume that tail recursive functions can be expressed in the general form

```

F(x)
{
    if (P(x)) return G(x)
    return F(H(x));
}

```

That is, we established a base case based on the truth-value of the function $P(x)$ of the parameter. Given that $P(x)$ is true, the value of $F(x)$ is the value of some other function $G(x)$. Otherwise, the value of $F(x)$ is the value of the function F on some other value, $H(x)$.

Given this formulation, we can immediately write an iterative version as

```

F(x)
{
    int temp_x=x;
    while( p(x) is not true)
    {
        temp_x=x;
        x=H(temp_x);
    }
    return G(x);
}

```

Example: Factorial function

```

int fact_aux(int n, int result)
{
    if (n==1)
        return result;
    else

```

```

        return fact_aux(n-1,n*result)
    }
int fact(n)
{
    return fact_aux(n,1);
}

```

Consider the above tail recursive function

- the function F is fact_aux
- x is composed of the two parameters , n, and result
- the value of P (n , result) is the value of (n==1)
- the value of G(n , result) is the result
- the value of H(n, result) is (n-1, n*result)

Therefore the iterative version is:

```

int fact_iter(int n, int result)
{
    int temp_n;
    int temp_result;
    while(n !=1)
    {
        temp_n=n;
        temp_result=result;
        n= temp_n * temp_result;
    }
    return result;
}

```

Summary

- Recursion is a repetitive process in which a function calls itself either directly or indirectly.
- Infinite recursion occurs in a program when base criteria are omitted or base criteria are never reached, it results in a stack overflow.
- Recursion tree is a pictorial representation of recursion call, which is in the forms of a tree, where at each level nodes are expanded.
- A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.
- Recursion consumes more storage space and execution time.

Exercises

1. —The designer of an algorithm need to balance between space complexity and time completely.”— Comment on the validity of the statement in the context of recursive algorithms.
2. Are recursion routines more efficient than non-recursive routines?
3. What do you mean by recursion? Write a function to find out the GCD of two numbers using recursion technique.
4. Write down the recursive definition for generating the Fibonacci sequence.
5. Assuming Fib(x) is a recursive function; draw a recursion tree for Fib (6).

6. Write a recursive algorithm to find x^n .
7. What is tail recursion? Explain with a suitable example.
8. Write an algorithm to solve the Tower of Hanoi problem. Also, calculate the time complexity of your algorithm.
9. Write the recursive function for the Tower of Hanoi problem. Also, draw the recursion tree for any set of initial values.
10. Let, a and b denotes positive integers. Suppose a function Q is defined as follows:

$$\begin{aligned} Q(a, b) &= 0 && \text{if } a < b \\ &= Q(a-b, b) + 1 && \text{if } b \leq a \end{aligned}$$

Find the value of $Q(2, 3)$ and $Q(14, 3)$.

11. Choose the correct alternatives for the following:

- i) The Ackerman function, for all non-negative values of m and n, is recursively defined as

$$A(m, n) = \begin{cases} n + 1, & \text{if } m = 0 \\ A(m-1, 1), & \text{if } m \neq 0 \text{ but } n = 0 \\ A(m-1, A(m, n-1)), & \text{if } m \neq 0 \text{ and } n \neq 0 \end{cases}$$

Therefore, the value of $A(1, 2)$ is

- a) 4
- b) 3
- c) 5
- d) 2

- ii) Which of the following statement is not true?

- a) Each time a procedure calls itself, it must be nearer in some sense to solution
- b) Recursion functions are always fast and use less memory
- c) When last executed statement at a function in a recursive call then it is known as tail recursion
- d) Recursion uses the top down and backtracking approach for solving a problem.

CHAPTER 8

TREE

"Computers do not solve problems, they execute solutions." -Laurent Gasser

Tree is one of the most important non-linear data structure in computer algorithms. The drawbacks of linked list can be overcome by using a tree. Many real life problems can be represented and solved using trees. Trees are very flexible, versatile and powerful non-linear data structure that can be used to represent data items possessing a hierarchical relationship among the nodes of the tree.

Definition: A **Tree** may be defined as a non-empty finite set of nodes, such that,

- i) There is a specially designated node called the root,
- ii) The remaining nodes are partitioned into zero or more disjoint trees $T_1, T_2 \dots T_n$ are called the **subtrees** of the root R.

KEY FEATURES	
Binary Tree	
Binary Search Tree	
Threaded Tree	
Expression Tree	
AVL Tree	
Multi-way Search Tree	
B-tree	

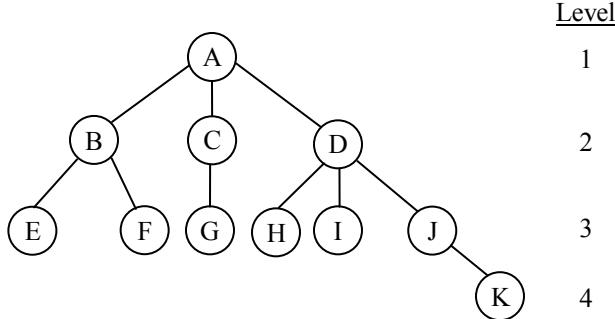


Figure 8.1: A Tree

Terminology of Tree

Node (or vertex): A node stands for the item of information with the branches to other items. Consider the tree in the above figure. This tree has 11 nodes (A, B, C, D, E, F, G, H, I, J and K).

Root: A node without any parent is called root node. In the above figure, A is the root node.

Parent node (or predecessor): Suppose N is a node in a tree with successors $s_1, s_2 \dots s_n$ then N is called the **parent** (or predecessor) of the successors. Each node in the tree, except the root, has a unique parent. In the above figure, node D is the parent of H, I and J.

Children: The successors are called **children** of N. The left and right successors of node N are called left child and right child of node N respectively. In the above figure, H, J and I are children of node D.

Siblings: The children (or the nodes) of the same parent are said to be **siblings**. In the above figure, H, I and J are siblings.

The degree of a node: The number of subtrees (or children) of a node is called its degree. In the above figure, the degree of node A and D are 3, the degree of B is 2, degree of C is 1, the degree of E, F, G, H, I and K are zero.

The degree of tree: The degree of a tree is the maximum degree of the nodes in the tree. In the above figure, degree 3 is the maximum. Therefore, the degree of the tree is 3.

Internal node (or non-terminal node): The node with at least one child is called internal nodes. In the above figure, A, B, C, D and J are internal nodes.

External nodes (or leaf node): The nodes that have degree zero are called external node or leaf or terminal nodes. In the above figure, E, F, G, H, I and K are the leaf nodes.

Level: The level of a node is defined as follows:

- i) The root of the tree is at level one.
- ii) If a node is at level L, then its children are at level L + 1.

Note: In some literature, the level of a node is defined in such a way that the root of the tree is at level zero.

In the above figure, the corresponding levels are shown; node A is at level 1, nodes B, C, and D are at level 2, nodes E, F, G, H, I and J are at level 3 and node K at level 4.

Height (or depth): The height or depth of a tree is defined to be the maximum level of any node in the tree. In the above figure, the height of the tree is 4.

Forest: A forest is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest.

Descendant: A node M in the tree is called a descendant of another node N, if M reachable from the N by repeated proceeding from parent to child.

Ancestor and Descendant: A node N is called an ancestor of node M if N is either the parent of M or the parent of some ancestor of M, i.e., there is a succession of children from N to M; the node M is called descendant of the node N. In the above figure, node D is the ancestor of H, I, J and K.

Edge: The line from a node N in the tree to a successor is called an edge.

Path and path length: A sequence of consecutive edges from the source node to the destination node is called a path. The number of edges in a path is path length. In the above figure, A→D→J is a path and the path length is 2.

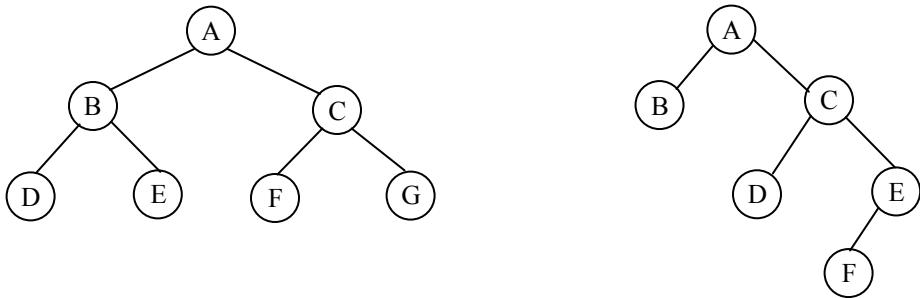
Internal path length: The sum of the levels of all the internal nodes in the tree is called internal path length.

External path length: The sum of the levels of all the external nodes in the tree is called external path length.

Branch: A path ending in a leaf node is called a branch. In the above figure, A→D→J→K is a branch.

Binary Tree

A binary tree is a finite set of nodes, which is either empty or consists of a root and two disjoint binary trees called left subtree and the right subtree. In a binary tree, the degree of any node is either zero, one or two. A binary tree is a special case of an ordered k-ary tree, where k is 2.

**Figure 8.2:** Some Binary Trees

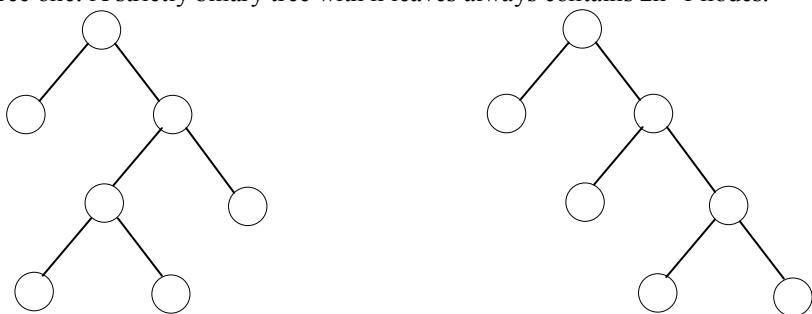
Different Types of Binary Trees

Different types of binary trees are possible. Following are common types of binary trees:

- Strictly Binary Tree
- Extended Binary Tree
- Complete Binary Tree
- Full Binary Tree
- Skewed Binary Tree
- Binary Expression Tree
- Balanced Binary Tree
- Threaded Binary Tree
- Binary Search Trees

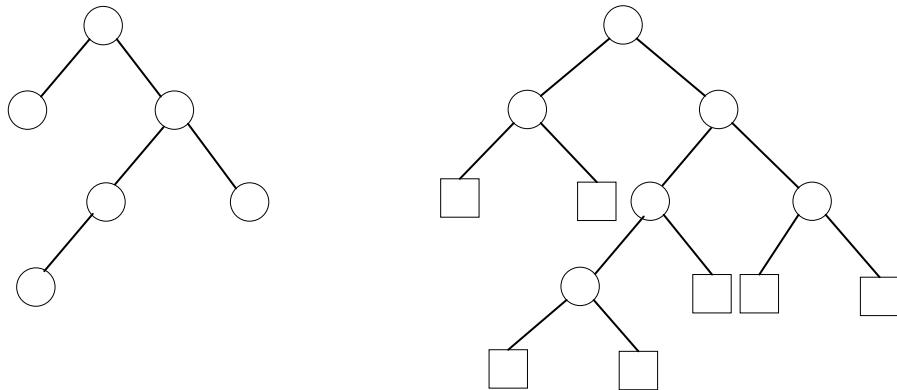
Strictly Binary Tree

A binary tree is called strictly binary tree if every non-terminal node has non-empty left and right subtree. All the non-terminal nodes must always have exactly two non-empty children. Strictly Binary tree is also known as a 2-ary tree. In the strictly binary tree, the degree of any node is either zero or two, never degree one. A strictly binary tree with n leaves always contains $2n-1$ nodes.

**Figure 8.3:** Strictly binary trees

Extended Binary Tree

A binary tree can be converted to an extended binary tree by adding special nodes to its leaf nodes and the nodes that have only one child. The extended binary tree also is known as 2-tree. The nodes of the original tree are called internal nodes and the special nodes that are added to the binary tree are called external nodes.

**Figure 8.4:** Binary tree and its corresponding extended binary tree

Now, we can define the external path length of a binary tree is the sum of all external nodes of the lengths of the paths from the root to those nodes. For example, the external path length E is:

$$E = 2 + 2 + 4 + 4 + 3 + 3 + 3 = 21$$

The internal path length is defined as the sum of all internal nodes of the lengths of the paths from the root to those nodes. For example, the internal path length I is:

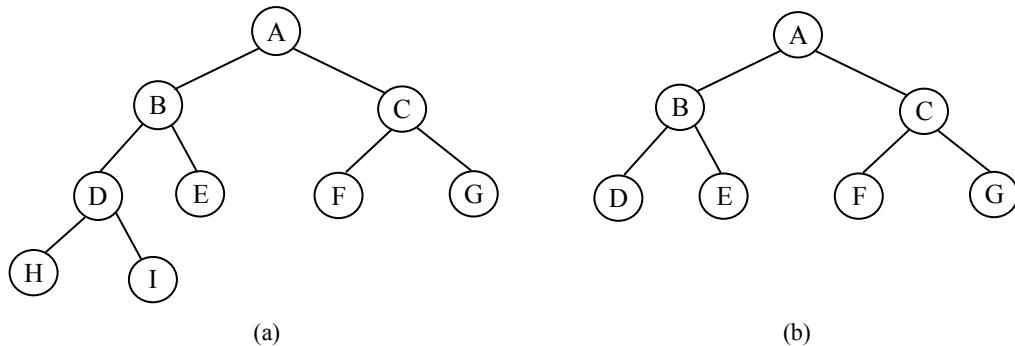
$$I = 0 + 1 + 1 + 2 + 2 + 3 = 9$$

Complete Binary Tree

A binary tree is called a complete binary tree in which all the levels are filled and the last level possibly be partially filled from left to right and some rightmost leaves may be missing. The complete binary tree is maximally space efficient. The number of internal nodes in a complete binary tree of n nodes is $\lfloor n/2 \rfloor$. Consider the figure, all the terminal nodes are at the adjacent levels. Practical example of a complete binary tree is Heap.

Full Binary Tree

A binary tree of depth $k \geq 1$, contains $2^k - 1$ nodes, is called full binary tree or perfect binary tree. Note that, a binary tree can have the maximum number of nodes $2^k - 1$. In the other word, in a full binary tree, all the internal nodes have two children and all the leaves are at the same level. Consider the figure, a full binary tree of depth 3. All full binary tree is a complete binary tree, but all complete binary trees are not full binary tree.

**Figure 8.5:** (a) Complete Binary Tree, (b) Full Binary Tree

Skewed Binary Tree

A binary tree, which is dominated solely by left child nodes or right child nodes, is called a skewed binary tree. A left-skewed binary tree has only left child nodes and a right-skewed binary tree has only right child nodes. The height of a skewed binary tree of n nodes is n . In the figure 8.6, the first is a left-skewed binary tree, skewed to left and the second is a right-skewed binary tree, skewed to right. Skewed binary trees are performed worst in all types of trees; such trees are performed similarly as linked list.

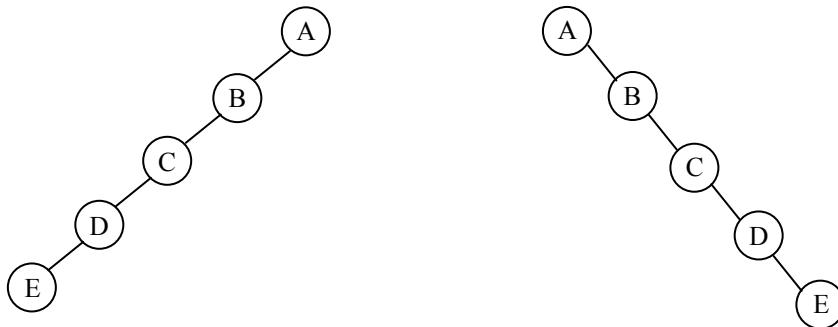


Figure 8.6: Left skewed Binary Tree and Right skewed Binary Tree

Binary Expression Tree

A binary expression tree is a strictly binary tree, which is used to represent a mathematical expression. Two common types of expressions that an expression tree can represent are algebraic and boolean.

This is not necessary that expression tree is always a binary tree. In this example, there are all the binary operators. Therefore, this tree is a binary tree. Although, it is possible to a node to have one child as in the case of a unary operator.

Applications of expression trees:

- Evaluation of expression
- Performing symbolic mathematical operations (such as differentiation) on the expression.
- Generating correct compiler code to actually compute the expression's value at execution time.

Properties of expression tree:

- Expression tree does not contain parenthesis.
- The leaf nodes contain the operands such as constants or variables.
- The non-leaf nodes contain the operators.

Inorder traversal of the expression tree produces infix form of the expression without parenthesis. When the expression tree is traversed in preorder, the prefix (polish) form of the expression is obtained. Similarly, when the expression tree is traversed in postorder then the postfix (reverse polish) form of the expression is obtained. A prefix or postfix form corresponds to exactly one expression tree.

An infix form may correspond to more than one expression tree. Therefore, it is not suitable for expression evaluation.

Consider the infix expression $A + B * C$. The expression is ambiguous because it produces more

than one expression trees with same infix form.

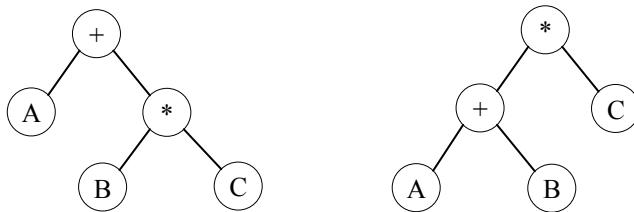


Figure 8.7: Expression Tree of infix expression $A + B * C$

Table 8.1: Different forms of expression

Expression	Prefix form	Infix form	Postfix form
$A + (B * C)$	$+ A * B C$	$A + B * C$	$A B C * +$
$(A + B) * C$	$* + A B C$	$A + B * C$	$A B + C *$

Construction of Expression Tree

An expression can be converted to its equivalent postfix expression. The following algorithm constructs an expression tree from a valid postfix expression P containing binary operators:

Algorithm to create expression tree

Algorithm: CREATE_EXPRESSION_TREE (P)

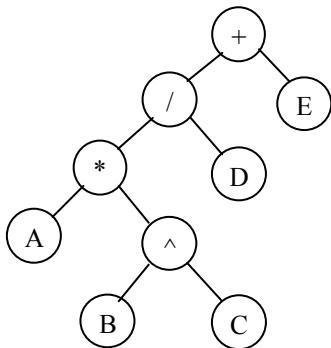
1. Repeat while not the end of the expression P
2. Read the postfix expression one symbol S at a time
3. If S is an operand then
 - i) Create a node for the operand
 - ii) Push the pointer to the created node onto a stack
4. If S is a binary operator then
 - i) Create a node for the operator
 - ii) $T1 = \text{Pop}$ from the stack a pointer to an operand
 - iii) $T2 = \text{Pop}$ from the stack a pointer to an operand
 - iv) Make $T2$ the left subtree and $T1$ the right subtree of the operator node
 - v) Push the pointer to the operator node onto the stack
- [End of while]
5. $T = \text{Pop}$ from the stack a pointer to expression tree
6. Return

Example:

Draw an expression tree from the following expression:

$A * B ^ C / D + E$

Now, at first converts the above expression to its equivalent postfix expression: $A B C ^ * D / E +$

**Figure 8.8:** Expression Tree

The preorder traversal of the above expression tree is $+ / * A ^ B C D E$, this is the prefix form of the expression. The postorder traversal of the above expression tree is $A B C ^ * D / E +$, this is the postfix form of the expression.

Evaluation of Expression Tree

A postfix expression can be converted to expression tree using the above algorithm. Now we can evaluate the expression tree, using the following algorithm:

Algorithm to evaluation of expression tree

Algorithm: EVALUATION (T)

1. If T is a leaf then
2. Return value of T's operand
3. Else
4. Operator = T.Element
5. Operand1 = EVALUATION(T.Left)
6. Operand2 = EVALUATION(T.Right)
7. Return (Operation (Operand1, Operator, Operand2))
- [End of If]
8. End

Balanced Binary Tree

There are mainly two types of balanced binary trees.

- i) Weight balanced binary tree
- ii) Height balanced binary tree

Weight Balanced Binary Tree

A weight-balanced binary tree is a binary search tree if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most one. These trees can be used to implement dynamic sets, dictionaries (maps) and sequences. The weight-balanced binary trees were introduced by Nievergelt and Reingold in 1972. It is purely functional implementations are widely used in functional programming languages.

The balance of weight-balanced binary tree is based on the sizes (number of elements) of the subtrees in each node. The size of the leaf node is zero. The size of the internal nodes is the sum of sizes of its two children, plus one ($\text{size}[n] = \text{size}[n.\text{left}] + \text{size}[n.\text{right}] + 1$). Based on the size, one defines the weight as either equal to the size, or as weight $[n] = \text{size}[n] + 1$. Now, insertion and deletion operations that modify the tree must make sure that the weight of the left and right subtrees of every node remain within some factor α of each other.

Height Balanced Binary Tree

A height-balanced binary tree has the minimum height for the leaf nodes. A binary tree is height balanced if height of the tree is $O(\log n)$ where n is number of nodes. One common height-balanced tree structure is a binary tree structure in which the left and right subtrees of every node differ in height by no more than one.

For Example, AVL tree (Adel'son-Vel'skii and E. M. Landis, 1962) maintain $O(\log n)$ height by making sure that the difference between heights of left and right subtrees is at most ± 1 . Red-Black trees (Guibas and Sedgewick, 1978) maintain $O(\log n)$ height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performed good as they provide $O(\log n)$ time for search, insert and delete operations.

Threaded Binary Tree

A threaded binary tree is a binary tree in which having a loop. In a binary tree, most of the entries in the link field will contain null elements. These null entries are replaced by special pointers, which point to nodes higher in the tree. These special pointers are called threads and binary trees with such pointers are called threaded tree.

For a n -node binary tree, there exists

- total $2n$ number of pointers or link fields
- total $n-1$ number of actual (not null) pointers or link fields
- total $n+1$ number of null pointers or link fields

For optimizing null pointers, the concept of the thread is used. There are different types of threaded binary trees are possible, inorder threaded binary trees, preorder threaded binary trees, and postorder threaded binary trees correspond to inorder, preorder and postorder traversals. Each type of threaded binary trees can be of two representations: one-way threading and two-way threading.

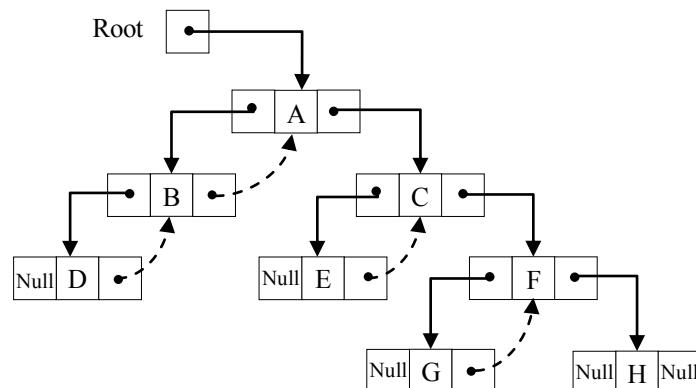


Figure 8.9: One-way inorder threaded binary tree

In the one-way inorder threaded binary tree, an only right null pointer used as a thread that will point to the next node in the inorder traversal (i.e. inorder successor). However, the right null pointer of the last node remains unused. In the two-way inorder threaded binary tree, left null pointer acts as a thread that will point to the previous node in the inorder traversal (i.e. inorder predecessor) and right null pointer acts as a thread that will point to the next node in the inorder traversal (i.e. inorder successor). However, the left null pointer of the first node and the right null pointer of the last node remains unused.

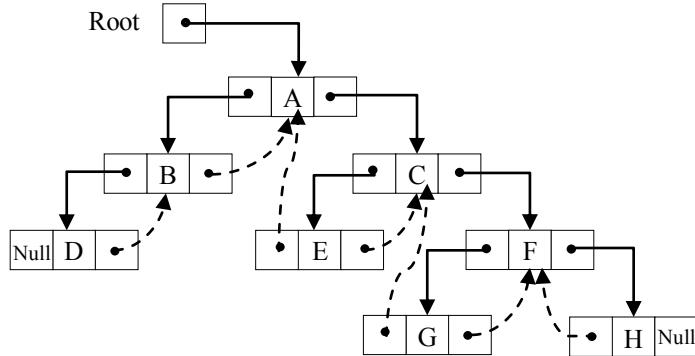


Figure 8.10: Two-way inorder threaded binary tree

The structure definition for a node of two-way inorder threaded binary tree as follows:

```

struct Node
{
    struct Node *left;
    char lthread;
    int info;
    struct Node *right;
    char rthread;
};

typedef struct Node ThreadedTreeNode;
  
```

Properties of Binary Tree

Lemma 1: A binary tree with n nodes has exactly $n - 1$ edges (same as any normal tree).

Proof: The property can be proof by induction.

Induction Base: Let $n = 1$. That is there is only one node in the tree. Therefore, a number of the edge is 0. Hence, the property is true for one node.

Induction Hypothesis: Assume the property is true for $n = k$ i.e., for k nodes there is $k - 1$ edges.

Induction Step: The number of edges for k nodes is $k - 1$ edges by the induction hypothesis. Now addition of one node (i.e. $n = k + 1$) includes one extra edge. Therefore, total number of edges are $= (k - 1) + 1 = k$. Hence proved.

Lemma 2: The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$

Proof: The property can be proof by induction on i .

Induction Base: The root is the only node on level $i=1$.

Hence the maximum number of nodes on level = 1 is $2^0 = 2^{i-1}$.

Induction Hypothesis: For all $j, i \leq j < i$, the maximum number of nodes on level j is 2^{j-1} .

Induction Step: The maximum number of nodes on level $i-1$ is 2^{i-2} , by the induction hypothesis. Since each node in a binary tree has maximum degree 2, the maximum number of nodes on level i is two times the maximum number on level $i-1$, i.e. $2 \times 2^{i-2} = 2^{i-1}$.

Lemma 3: The maximum number of nodes in a binary tree of depth k is $2^k - 1, k \geq 1$.

Proof: The maximum number of nodes on level i of a binary tree is 2^{i-1} . Therefore, the maximum number of nodes in a binary tree of depth k is,

$$\begin{aligned} &= \sum_{i=1}^k (\text{maximum number of nodes on level } i) \\ &= \sum_{i=1}^k (2^{i-1}) \\ &= 2^k - 1 \end{aligned}$$

Lemma 4: For any non-empty binary tree T , if n_0 is the number of leaves (terminal nodes) and n_2 be the number of nodes having degree 2 then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree 1 and n is the total number of nodes. Since all the nodes in T are of degree ≤ 2 we have,

$$n = n_0 + n_1 + n_2 \quad (i)$$

Now, if we count the number of branches in a binary tree, we see that every node except for the root has a branch leading into it. If B is the number of branches, then $n = B + 1$. All the branches come either from a node of degree one or from a node of degree two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = 1 + n_1 + 2n_2 \quad (ii)$$

Subtracting (ii) from (i) and rearranging terms we get

$$n_0 = n_2 + 1$$

Lemma 5: If n is the total number of nodes in a complete binary tree of height h , then

$$h = \lfloor \log_2 n \rfloor + 1.$$

Proof: From the definition of a complete binary tree of height h , it is filled up to height $h - 1$ and in the last level, it may have partially filled with nodes. Hence, we can write:

$$2^{h-1} - 1 < n \leq 2^h - 1$$

Since the maximum number of nodes at height $h-1$ is $2^{h-1} - 1$ and at height, h is $2^h - 1$.

$$\text{or we can write } 2^{h-1} \leq n < 2^h \quad (iii)$$

Taking the logarithm of (iii) we get

$$h - 1 \log_2 n < h$$

Therefore, the value of $\log_2 n$ lies between h and $h-1$. Now if we take floor value of $\log_2 n$ then it will be $h-1$.

Hence, $h = \lfloor \log_2 n \rfloor + 1$ or $h \leq \lfloor \log_2 n + 1 \rfloor$

Representation of Binary Tree

Tree is a widely used abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, we can implement a tree using either array or linked list. Binary Tree also can be representation with two different ways.

- i) Using array
- ii) Using linked list

Binary Tree Representation with Array

Binary Tree can be represented by the array. There are two different ways to represent a binary tree with array.

- Linked Representation
- Sequential Representation

Linked Representation

In Linked representation, a Binary Tree can be stored in computer memory by using three parallel arrays, DATA, LCHILD and RCHILD and a pointer variable ROOT. Now, each node N of binary tree T will correspond to a location k such that,

- i) DATA[K] contains the data at the node N
- ii) LCHILD[K] contains the location of the left child of node N
- iii) RCHILD[K] contains the location of the right child of node N

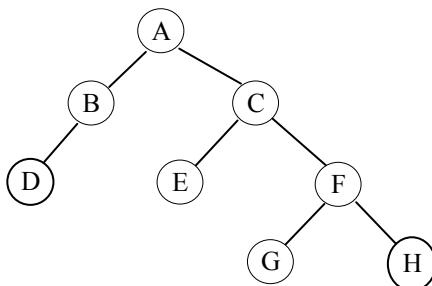


Figure 8.11: A Binary Tree

Root

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DATA	E		A		G	C		B		F		D			H
LCHILD	0		8		0	1		12		5		0			0
RCHILD	0		6		0	10		0		15		0			0

Figure 4.12: Linked Representation of above Binary Tree

Sequential Representation

There is an efficient way of maintaining or to store a Binary Tree in computer memory when the tree is complete or nearly complete. Since heap is a complete binary tree, therefore a heap can also be represented by this representation.

In the sequential representation, a Binary Tree T can be represented by using only a single linear array TREE, such that

- i) The root of T is stored in TREE[1].
- ii) When a node N stores in TREE[K], then its left child will be stored in TREE[2*K] and right child will be stored in TREE[2*K+1].

Therefore, if K is the index of current node, then parent node is stored in the FLOOR (K/2).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D		E	F							G	H

Figure 8.13: Sequential Representation of above Binary Tree

For a zero-based array,

- i) The root of T is stored in TREE[0]
- ii) When a node N stores in TREE[K], then its left child will be stored in TREE[2*K+1] and right child will be stored in TREE[2*K+2].

Therefore, if K is the index of a current node then parent node is stored in FLOOR ((K-1)/2).

For a complete binary tree, sequential representation is perfect as no space is wasted. However, it is wasteful for many other binary trees. For a skewed binary tree, less than half the array can be utilized. In the worst case, a skewed binary tree of depth k will require $2^k - 1$ memory space. In addition, insertion or deletion of a node in the middle of the tree requires movement of many nodes. These problems can be overcome using linked list representation.

Threaded Binary Tree Representation with Array

Threaded Binary Tree T may be stored in computer memory by using a linked representation. Here, the thread can be represented by a negative value of the location and ordinary pointer can be represented by the positive value of the location.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DATA	E		A		G	C		B		F		D			H
LCHILD	-3		8		-6	1		12		5		0			-10
RCHILD	-6		6		-10	10		-3		15		-8			0

Figure 8.14: Linked Representation of above Two-way Inorder Threaded Binary Tree

Binary Tree Representation with Linked List

Binary Tree can be also representation with Linked List. In this representation, each node of a binary tree consists three fields such that

- The first field contains the pointer field, which points to the left child.
- The second field contains the data.
- The third field contains the pointer field, which points to the right child.

LChild	Data	RChild
--------	------	--------

The structure definition for a node of binary tree as follows:

```
struct Node
{
    struct Node *left;
    int info;
    struct Node *right;
};

typedef struct Node BTreenode;
```

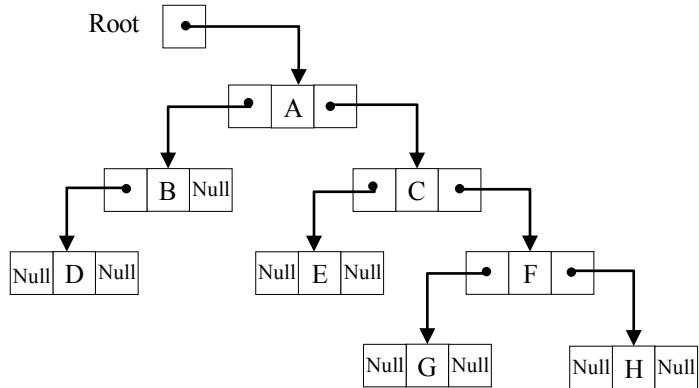


Figure 8.15: Linked Representation of Binary Tree

Binary Tree Traversal

Tree traversal is the operation of visiting each node in the tree exactly once. There are mainly two traversals are:

- i) Depth First Search (DFS)
- ii) Breadth First Search (BFS)

Depth First Search

Depth first search traversal can be implemented easily using a stack, including recursion. Starting from the root of the binary tree, there are three main steps that can be performed. These steps are moving left (L), visiting node (D) and moving right (R), then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL and RLD. Now if we traverse left before right then only three traversal remains: LDR, LRD and DLR; LDR is known as inorder, LRD is postorder and DLR is preorder, these three are only standard binary tree traversals.

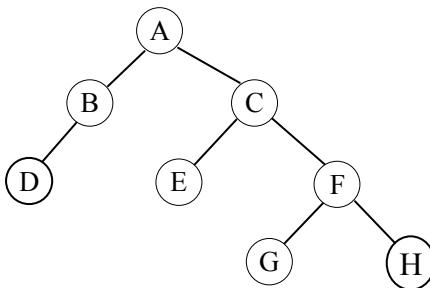


Figure 8.16: Binary Tree

Inorder Traversal

The inorder traversal of a non-empty binary tree is defined as follows, starting from root node:

- i) Traverse the left subtree of root in inorder.
- ii) Visit the root node.
- iii) Traverse the right subtree of the root in inorder.

Algorithm finds the inorder traversal of a binary tree using recursion**Algorithm: INORDER (ROOT)**

```

1. IF ROOT ≠ NULL THEN
    a) INORDER (ROOT → LCHILD)
    b) PRINT: ROOT → INFO
    c) INORDER (ROOT → RCHILD)
2. RETURN

```

The inorder of the above binary tree is:

D B A E C G F H

Algorithm finds the inorder traversal of a binary tree in a non-recursive mode.**Algorithm: INORDER (ROOT)**

```

1. P = ROOT
2. Initialize Stack
3. Repeat while Stack is not empty or P ≠ null
4.     Repeat while P ≠ null
            a) PUSH(Stack, P)
            b) P = P → LCHILD
                [End of loop]
5.     If Stack is not empty then
            a) P = POP(Stack)
            b) Print: P → Info
            c) P = P → RCHILD
                [End of loop]
6. Return

```

Preorder Traversal

The preorder traversal of a non-empty binary tree is defined as follows, starting from root node:

- i) Visit the root node.
- ii) Traverse the left subtree of root in preorder.
- iii) Traverse the right subtree of the root in preorder.

Algorithm finds the preorder traversal of a binary tree using recursion.**Algorithm: PREORDER (ROOT)**

```

1. IF ROOT ≠ NULL THEN
    a) PRINT: ROOT → INFO
    b) PREORDER (ROOT → LCHILD)
    c) PREORDER (ROOT → RCHILD)
2. RETURN

```

The preorder of the above binary tree is:

A B D C E F G H

Algorithm finds the preorder traversal of a binary tree in a non-recursive way

Algorithm: PREORDER (ROOT)

1. Initialize Stack
2. PUSH(Stack, ROOT)
3. Repeat while Stack is not empty
 - a) P = Pop(Stack)
 - b) If P ≠ null then
 - i) Print: P → Info
 - ii) Push(Stack, P → RCHILD)
 - iii) Push(Stack, P → LCHILD)
 - [End of loop]
4. Return

Postorder Traversal

The postorder traversal of a non-empty binary tree is defined as follows, starting from root node:

- i) Traverse the left subtree of root in postorder.
- ii) Traverse the right subtree of the root in postorder.
- iii) Visit the root node.

Algorithm finds the postorder traversal of a binary tree using recursion.

Algorithm: POSTORDER (ROOT)

1. IF ROOT ≠ NULL THEN
 - a) POSTORDER(ROOT → LCHILD)
 - b) POSTORDER(ROOT → RCHILD)
 - c) PRINT: ROOT → INFO
2. RETURN

The preorder of the above binary tree is:

D B E G H F C A

Algorithm finds the postorder traversal of a binary tree in a non-recursive way.

Algorithm: POSTORDER (ROOT)

1. Initialize Stack
2. P = ROOT
3. Repeat while Stack is not empty
 4. Repeat while P ≠ null
 5. Push(Stack, P)
 6. If P → RCHILD ≠ null
 7. Push(Stack, null)
 8. P = P → LCHILD
 - [End of while]
 9. Q = Pop(Stack)

```

10. If Q ≠ null
11.     PRINT: Q → INFO
12. Else
13.     Q = Pop(Stack)
14.     P = Q → RCHILD
15.     Push(Stack, Q)
    [End of If]
    [End of loop]
16. Return

```

Breadth First Search

Binary trees can also be traversed in level-order, where every node are visited on a level before the next level. This search is known as Breadth First Search (BFS). Breadth first search traversal can be implemented easily using a queue.

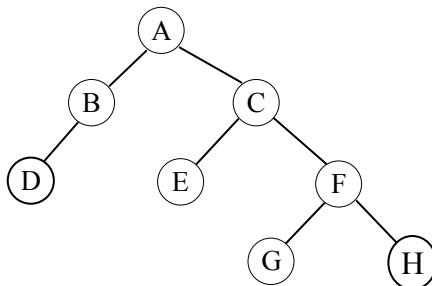


Figure 8.17: A Binary Tree

The breadth first search of the above binary tree is:

A B C D E F G H

Reconstruction Binary Tree from its Traversals

An original tree cannot be reconstructed given by its inorder or preorder or postorder traversal alone. However, a unique binary tree can be reconstructed either by inorder and preorder traversals, or by inorder and postorder traversals. However, preorder and postorder traversals give some ambiguity in the tree structure.

- The first node visited in a preorder traversal of a binary tree is the root, and then left subtree and right subtree are traversed.
- In postorder traversal of a binary tree, left subtree and right subtree are traversed then the last node visited is the root.
- In inorder traversal of a binary tree, left subtree is traversed first, then the root node is visited, finally right subtree is traversed.

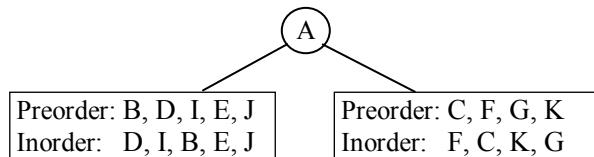
Example:

The inorder and preorder traversal sequence of nodes in a binary tree are given below:

Preorder: A, B, D, I, E, J, C, F, G, K
Inorder : D, I, B, E, J, A, F, C, K, G

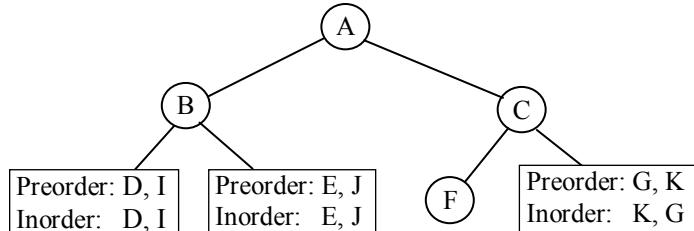
The following steps are used to reconstruct the binary tree:

In preorder traversal of a binary tree, the root node is visited first and then left subtree is traversed, and finally, the right subtree is traversed. In inorder traversal of a binary tree, left subtree is traversed first, then the root node is visited, finally, the right subtree is traversed. Therefore, in the preorder traversal, the first node A must be root. Now by searching node A in inorder traversal we can find out all elements on the left side of A are an inorder traversal of left subtree and elements on right are an inorder traversal of right subtree. Therefore, inorder and preorder traversal of left subtree and right subtree can be obtained.



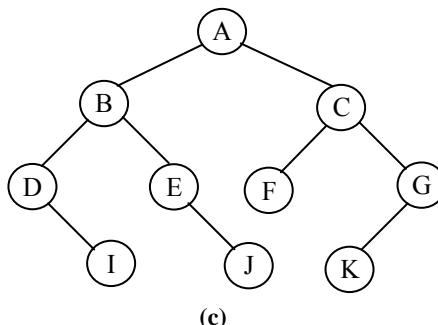
(a)

Similarly, node B is the root of the left subtree and node C is the root of the right subtree. Inorder and preorder traversal of left subtree and right subtree of B and C can be found. The node F is the obviously left child of node C.



(b)

Similarly, node D and E are the roots of the left subtree and the right subtree of node B. the node G is the right subtree of node C. In addition, node I is the right child of node D, node J is the right child of node E and node K is the left child of node G.



(c)

Figure 8.18 (a, b, c): Reconstruction of Binary Tree

Binary Search Tree

A Binary search tree (BST) is an ordered binary tree. P. F. Windley, A. D. Booth, A. J. T. Colin and T. N. Hibbard invent binary search tree, in 1960.

Definition: A Binary Search Tree, which is either empty or each node in the tree contains a key and

- i) All keys in the left subtree are less than the keys in the root node,
- ii) All keys in the right subtree are greater than the keys in the root node,
- iii) The left and right subtrees are also binary search tree.

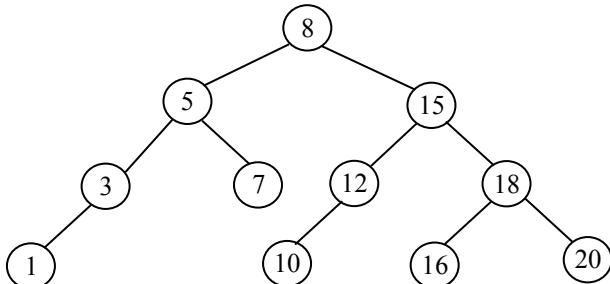


Figure 8.19: Binary Search Tree

Operations on Binary Search Tree

Operations supported by a Binary search tree are as follows:

Table 8.2: Various Operation on Binary Search Tree

Operation	Description
Traverse	This operation traversing all the nodes of binary search tree exactly once.
Insertion	This operation insert a node in the binary search tree
Deletion	This operation removes a node from the binary search tree
Searching	This operation searches a desired key value within the binary search tree.
Successor	This operation finds the successor of a given node in the binary search tree.
Predecessor	This operation finds the predecessor of given node in the binary search tree.

Binary Search Tree Traversal

The tree traversal algorithm (preorder, postorder and inorder) are the standard way of traversing a binary search tree, which is similar as traversing in a binary tree. In a binary search tree, inorder traversal always retrieves data items in increasing sorted order.

Insertion in Binary Search Tree

Suppose a new data item having a key and the tree in which the key is inserted are given as an input. Insertion operation starts from the root node. If the tree is empty then the new item inserted as the root node. Otherwise, if the tree is non-empty then compare the value of the key with the root node. If the key is less than the root node then it is inserted in the left subtree, otherwise it is inserted in the right subtree.

Example:

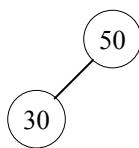
Insert the keys in the Binary Search Tree: 50, 30, 10, 90, 100, 40, 60, 20, 110, 5

Insert 50:



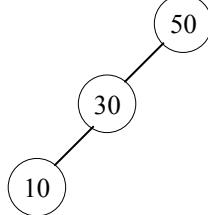
(a)

Insert 30:



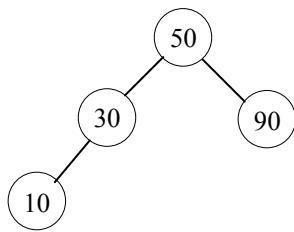
(b)

Insert 10:



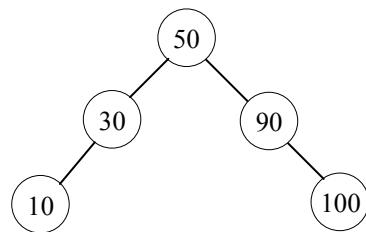
(c)

Insert 90:



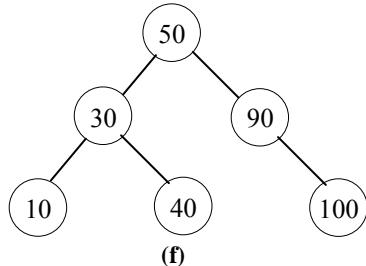
(d)

Insert 100:



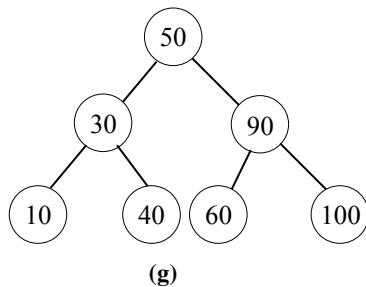
(e)

Insert 40:

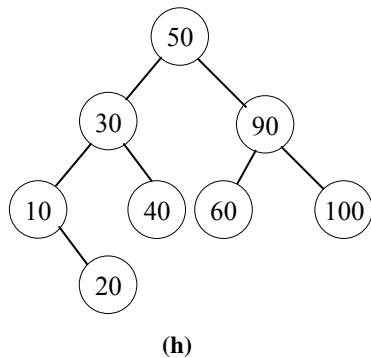


(f)

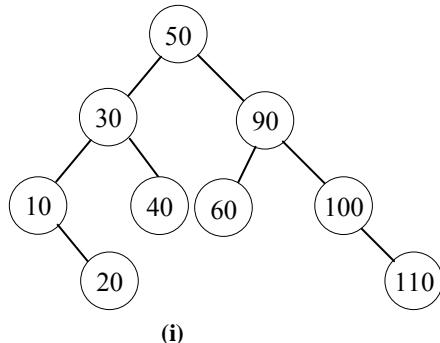
Insert 60:



Insert 20:



Insert 110:



Insert 5:

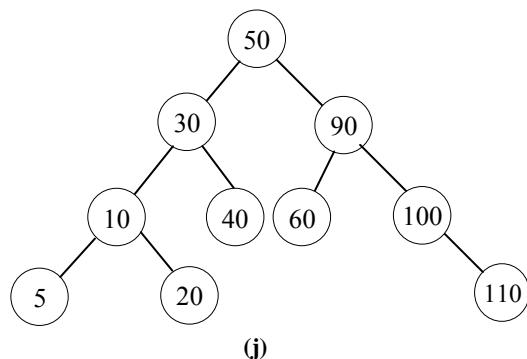


Figure 8.20(a-j): Insertion in Binary Search Tree

Algorithm to insert an item to binary search tree using recursion method

```
Algorithm: INSERT (ROOT, DATA)
1. IF ROOT = NULL THEN
    i) Allocate Memory for ROOT node.
    ii) ROOT->INFO=DATA
    iii) ROOT->LCHILD=NULL
    iv) ROOT->RCHILD=NULL
2. ELSE IF ROOT->INFO>DATA THEN
    CALL INSERT (ROOT->LCHILD, DATA)
3. ELSE IF ROOT->INFO<DATA THEN
    CALL INSERT (ROOT->RCHILD, DATA)
4. RETURN
```

Searching in Binary Search Tree

Similar to traversing, insertion operations in the binary search tree, search algorithm also utilized the recursion technique.

Suppose a key and the tree in which the key is searched for are given as an input. Now starting from the root node, check whether the value of the current node equals to the key or not. In the case, when a current node is null then the searched key value does not exist in the binary search tree. If the node has the key that is being searched for, then the search is successful.

Otherwise, the key of the current node is either smaller than or greater than the searching key value. In the first case, all the keys in the left subtree are less than the searching key value. That means do not need to search in the left subtree. Thus, it needs to search only the right subtree. Similarly, in the second case, it needs to search only the right subtree.

Algorithm to search an item from a binary search tree using recursion

```
Algorithm: BSTSearch (ROOT, DATA, P)
1. IF ROOT = NULL THEN
    i) PRINT: NOT FOUND
    ii) P = NULL
    iii) RETURN
2. IF ROOT->INFO=DATA THEN
    SET P=ROOT
3. ELSE IF ROOT->INFO>DATA THEN
    CALL BSTSearch (ROOT->LCHILD, DATA, P)
4. ELSE CALL BSTSearch (ROOT->RCHILD, DATA, P)
5. RETURN
```

Algorithm to search an item from a binary search tree using iterative methods

```
Algorithm: BSTSearch (ROOT, DATA, P)
1. P = ROOT
2. Repeat while P ≠ Null
```

```

3. If DATA = P->INFO then Return
4. Else If DATA < P->INFO then
5.     P = P->LCHILD
6.     Else P = P->RCHILD
    [End of loop]
7. Return

```

Suppose a binary search tree contains n data items. Therefore, there are $n!$ Permutations of the n items. The average depth of the $n!$ numbers of the tree is approximately $c \log_2 n$, where $c = 1.4$. The average running time $f(n)$ to search for an item in a binary search tree with n elements is proportional to $\log_2 n$, that is $f(n) = O(\log n)$.

Inorder Successor of a Node

In a binary tree, inorder successor of a node is the next node in inorder traversal of the binary tree. Inorder successor is NULL for the last node in inorder traversal. In binary search tree, inorder successor of a node with key k is a smallest key value that belongs to the tree and that is strictly greater than k . The idea for finding the successor of a given node x :

- i) If the x has a right child then its inorder successor will be the left most element in the right subtree of x (i.e. the minimum in the right subtree of x).
- ii) Otherwise, if the x doesn't have a right child then its inorder successor will be one of its ancestors, the inorder successor is the farthest node that can be reached from x by following only right branches backward.
- iii) Otherwise, if x is the right most node in the tree then its inorder successor will be NULL.

Algorithm to find the inorder successor from binary search tree

Algorithm: BST_SUCCESSOR(X)

```

1. If X → RCHILD ≠ NULL then
2.     Y = X → RCHILD
3.     Repeat while Y→LCHILD ≠ NULL
4.         Y = Y → LCHILD
5.     [End of loop]
6.     Return Y
7. Else
8.     Y = X
9.     Repeat while Parent(X) → RCHILD = X
10.        X = Parent(X)
            [End of loop]
11.    If Parent(X) ≠ NULL then Return Parent(X)
12.    Else Print: No successor
            [End of If]
            [End of If]
13. Return

```

Example:

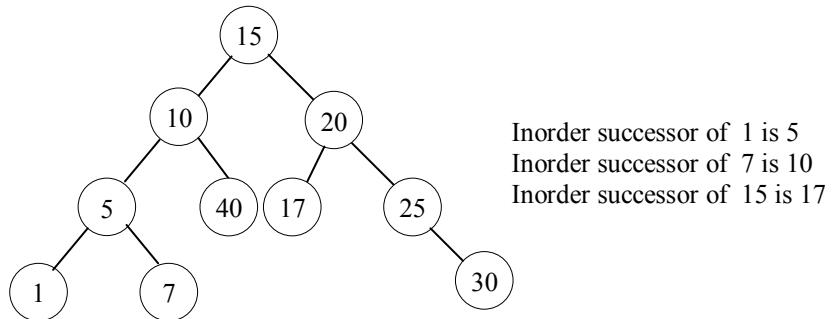


Figure 8.21: Binary Search Tree

Deletion in Binary Search Tree

Deletion operation in the binary search tree, there are three possible cases has to consider.

Case 1: When the deleting node with no children or leaf node (i.e. deleted node is a leaf node), then simply remove the node from the tree and set null to the parent's corresponding link.

Case 2: When the deleting node with one child, either left or right child (i.e. deleted node has exactly one non-empty subtree), then simply replace the node with its unique child.

Case 3: when the deleting node (P) with two children (i.e. deleted node has exactly two subtrees), then select its inorder successor node or its inorder predecessor node (R). Copy the key value from node R to node P and then recursively delete the R node until satisfying one of the first two cases. In a binary tree, inorder successor (R) of a node is only its right subtree's left-most child, as right subtree is not null (in the present case the node has two children). Now inorder successor may have zero children or only one right child, therefore it can be deleted using one of the first two cases.

Algorithm to delete an item from binary search tree

```

Algorithm: DELETE (ROOT, P, PARENT, DATA)
1. IF P→LCHILD=NULL AND P→RCHILD=NULL THEN
    i) IF PARENT→LCHILD=P THEN
        SET PARENT→LCHILD=NULL
    ii) ELSE SET PARENT→RCHILD=NULL
2. ELSE IF P→LCHILD=NULL THEN
    i) IF PARENT→LCHILD=P THEN
        SET PARENT→LCHILD=P→RCHILD
    ii) ELSE SET PARENT→RCHILD=P→RCHILD
3. ELSE IF P→RCHILD=NULL THEN
    i) IF PARENT→LCHILD=P THEN
        SET PARENT→LCHILD=P→LCHILD
    ii) ELSE SET PARENT→RCHILD=P→LCHILD
4. ELSE P→LCHILD≠NULL AND P→RCHILD≠NULL THEN
    i) SET PARENT = P
    ii) SET IN=P→RCHILD
  
```

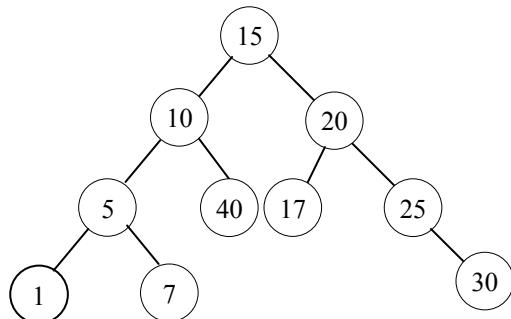
```

    iii) REPEAT WHILE IN->LCHILD ≠ NULL
        a) SET PARENT = IN
        b) SET IN=IN->LCHILD
        [END OF LOOP]
    iv) SET P->INFO=IN->INFO
    v) SET P = IN
    vi) Call DELETE(ROOT, P, PARENT, DATA)
    [END OF IF]
5. Deallocate memory for P node.
6. RETURN.

```

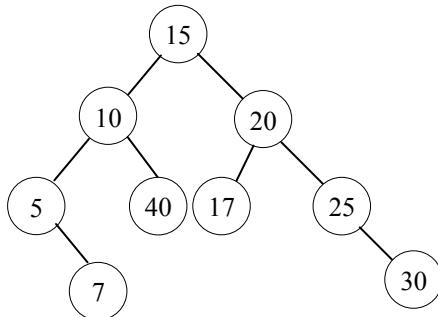
Example:

Delete the keys from the following Binary Search Tree: 1, 30, 5, 15



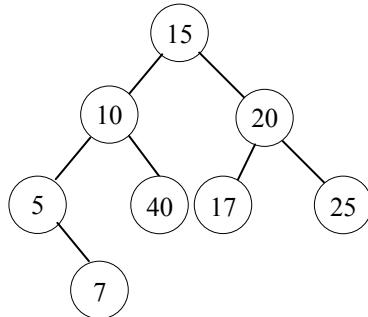
(a)

Delete 1: Here the deleted node is the left child of its parent node. Hence after setting PARENT->LCHILD=NULL we get,



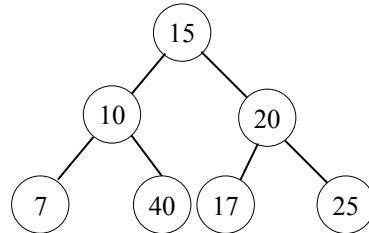
(b)

Delete 30: Here the deleted node is the right child of its parent node. Hence, after setting PARENT→RCHILD=NULL we get,



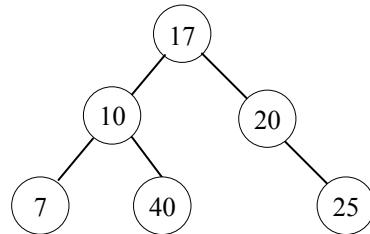
(c)

Delete 5: Here the deleted node is the left child of its parent node and it has a right subtree. Hence, after setting PARENT→LCHILD=P→RCHILD we get,



(d)

Delete 15: Here the deleted node has two subtrees. At first, find the inorder successor of the deleted node then substituting the key of the deleted node by the key of its inorder successor. Finally, delete the inorder successor.



(e)

Figure 8.22(a-e): Deletion in Binary Search Tree

On a binary search tree of height h , different operations like search, minimum, maximum, successor, predecessor, insert and delete can be made to run $O(h)$ time. On average, binary search trees with n nodes have $\log n$ height and in the worst-case, binary search trees can have n height. Therefore, different operations like search, minimum, maximum, successor, predecessor, insert and delete take $O(\log n)$ time in average-case and $O(n)$ in worst-case.

Binary search trees are a basic data structure used to construct abstract data structures such as sets, multisets, associative arrays. To sort a sequence of numbers, at first, all numbers are required to insert into a new binary search tree then traverse the tree in inorder.

Advantages of Binary Search Tree

The major advantage of binary search tree over other data structures are as follows:

- i) Sorting and search algorithm can be very efficient.
- ii) Easy to the coding of most of the operations that performed on binary search tree.

Disadvantages of Binary Search Tree

- i) The shape of the binary search tree fully depends on the sequence of insertions and deletions operations may result in skewness.
- ii) The height of the binary search tree is much higher than $\log n$ in the most of the cases, as a result, runtime may increase.
- iii) As the binary search tree is not a balanced tree, run time of most of the operations is $O(n)$ in the worst case.

HEAP

Heap is a binary tree that must satisfy the following properties:

- i) The binary tree essentially complete that means the tree completely filled all levels, the last level may be partially filled from left to right, and some rightmost leaves may be missing.
- ii) All keys in the tree, other than the root node, are greater/smaller than or equal to the key in the parent node.

There are two types of the heap:

- i) Max Heap and ii) Min Heap

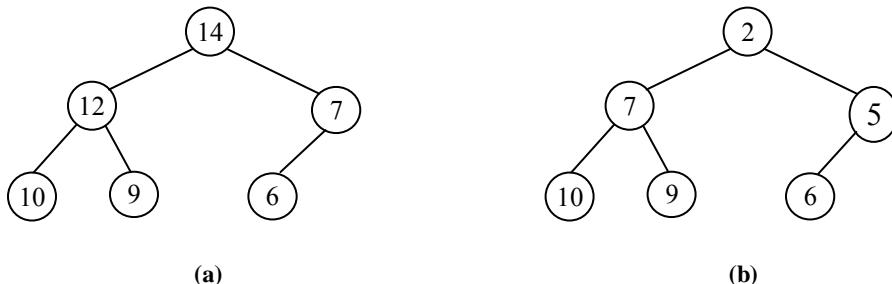


Figure 8.23: (a) Max Heap and (b) Min Heap

Max Heap

A Max Heap is defined to be a complete binary tree with the property that the key of each node is greater than or equal to the keys of its children nodes.

Min Heap

A Min Heap is also a complete binary tree with the property that the key of each node is smaller than or equal to the keys of its children nodes.

Operations on Heap

Operations supported by a Heap are as follow:

Table 8.3: Various Operation on Heap

Operation	Description
Heapify	This operation restores the heap condition. For example, if a node changed in the tree, the heap condition is not valid anymore. Then it needs to restore the condition by moving nodes up or down the tree.
Insertion	This operation inserts a node in the heap.
Deletion	This operation removes a node from the heap.
Shift-up	This operation moves a node up in the tree, as long as needed (depending on the heap condition: min-heap or max-heap).
Shift-down	This operation moves a node down in the tree.

Application of Heap

- Operating Systems- Jobs / Process scheduling
- Heap Sorting
- Graph Application
- Priority Queue

AVL Tree

A binary tree is height balanced binary tree if it is either empty or if T is a non-empty binary search tree with T_L and T_R as its left and right subtrees, if and only if

- i) T_L and T_R are height balanced and
- ii) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

It is introduced by two Russian mathematicians, **G. M. Adelson-Velskii** and **E. M. Landis**, in 1962. Hence, such trees are known as AVL trees.

The **balance factor** BF (T) of a node T in a binary tree is defined to be $h_L - h_R$ where h_L and h_R are the heights of the left and right subtrees of T. For any node in an AVL tree $BF(T) = -1, 0$ or 1 .

Operations on AVL Tree

Operations supported by an AVL tree are as follow:

Table 8.4: Various Operation on AVL Tree

Operation	Description
Traverse	This operation traversing all the nodes of AVL tree exactly once.
Insertion	This operation inserts a node in the AVL tree
Deletion	This operation removes a node from the AVL tree
Searching	This operation searches a desired key value within the AVL tree.

Insertion in an AVL Tree

When a new node is inserted to a balanced binary search tree, as a result the tree could be unbalanced. The re-balancing was carried out using four different kinds of rotations LL, RR, LR and RL.

These rotations are characterized by nearest ancestor A on the path from the inserted node B to the root node, whose balanced factor becomes ± 2 .

LL Rotation

In this rotation, new node B is inserted in the left subtree of the left subtree of A. That means B is inserted in the left subtree C_L of node C.

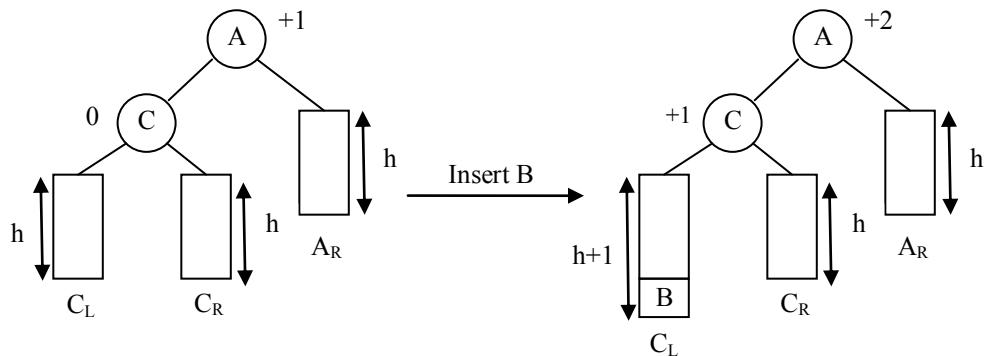


Figure 8.24 : Before and after Insertion

After insertion, the balanced factor node A and C are changed and the tree becomes unbalanced.

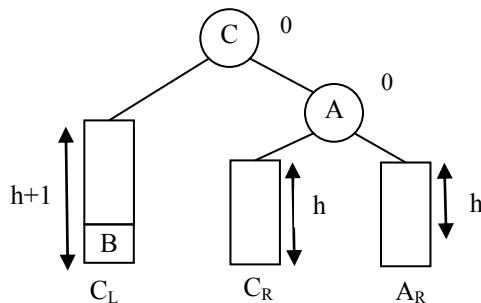


Figure 8.25: After LL Rotation

It can be easily seen in the following figure, after inserting node B in the AVL tree, we get an unbalanced AVL tree. By performing LL rotation, the resultant tree becomes balanced AVL tree.

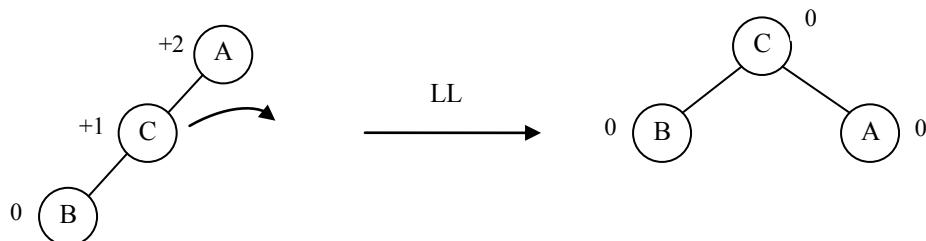


Figure 8.26: LL Rotation

RR Rotation

In this rotation, new node B is inserted in the right subtree of the right subtree of node A. That means node B inserted in the right subtree C_R of node C.

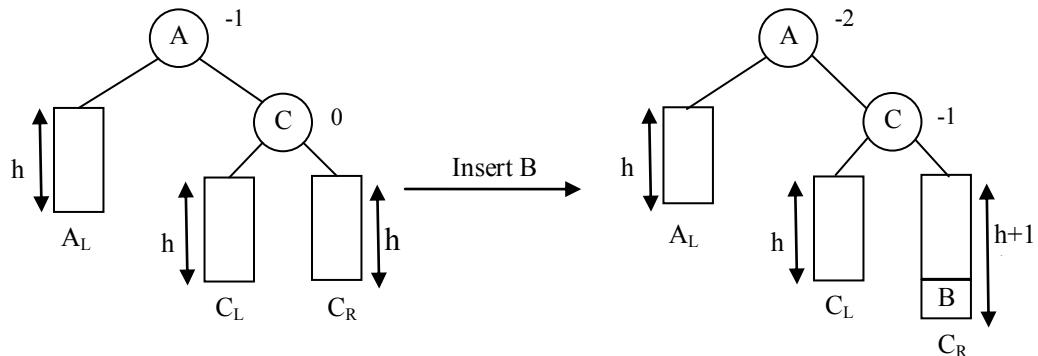


Figure 8.27: Before and after insertion

After insertion, the balanced factor node A and C are changed and the tree becomes unbalanced.

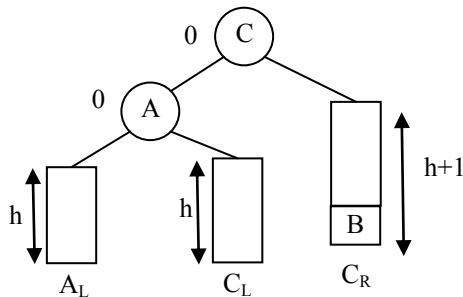


Figure 8.28: After RR rotation

It can be easily seen in the following figure, after inserting node B in the AVL tree, we get an unbalanced AVL tree. By performing RR rotation, the resultant tree becomes balanced AVL tree.

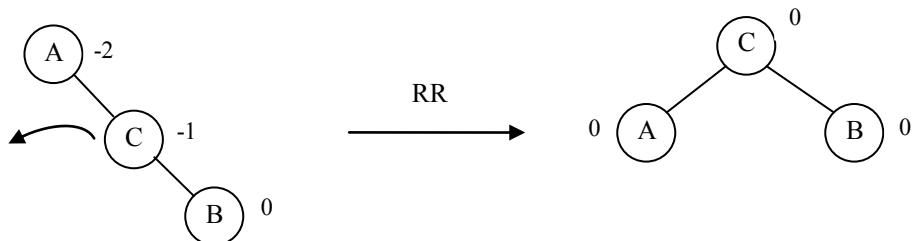


Figure 8.29: RR Rotation

LR Rotation

In this rotation, new node D is inserted in the right subtree of the left subtree of A node. That means node D inserted in the left subtree B_L of node B. LR = LL(RR).

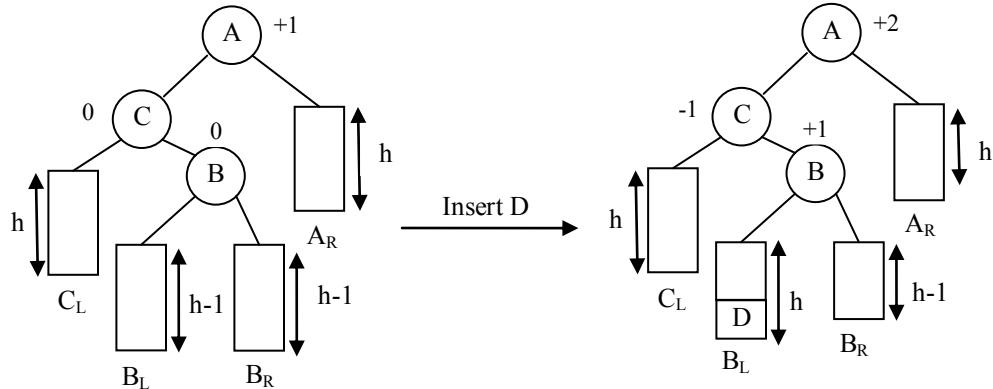


Figure 8.30: Before and after insertion

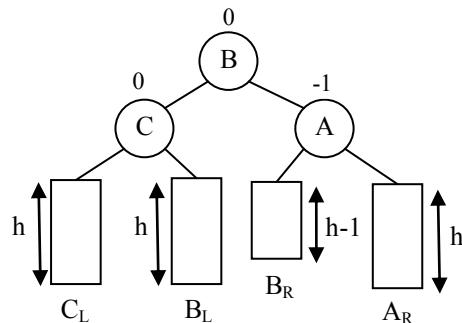


Figure 8.31: After LR rotation

It can be easily seen in the following figure, after inserting node B in the AVL tree, we get unbalanced AVL tree. By performing LR rotation, the resultant tree becomes balanced AVL tree.

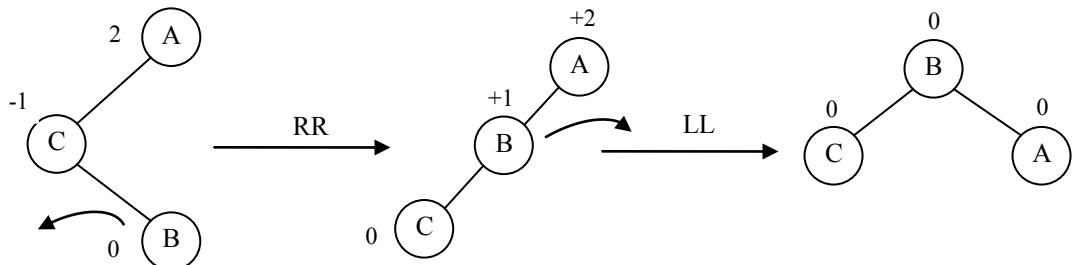


Figure 8.32: LR Rotation

RL Rotation

In this rotation, new node B is inserted in the left subtree of the right subtree of A. That means node D is inserted in the left subtree B_L of the node. RL = RR(LL).

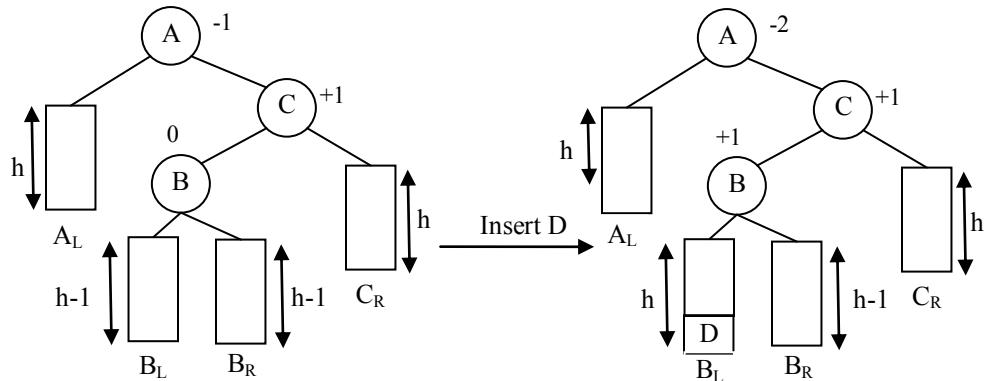


Figure 8.33: Before and after insertion

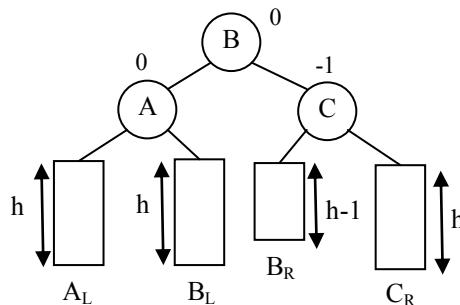


Figure 8.34: After RL rotation

It can be easily seen in the following figure, after inserting node B in the AVL tree, we get unbalanced AVL tree. By performing RL rotation, the resultant tree becomes balanced AVL tree.

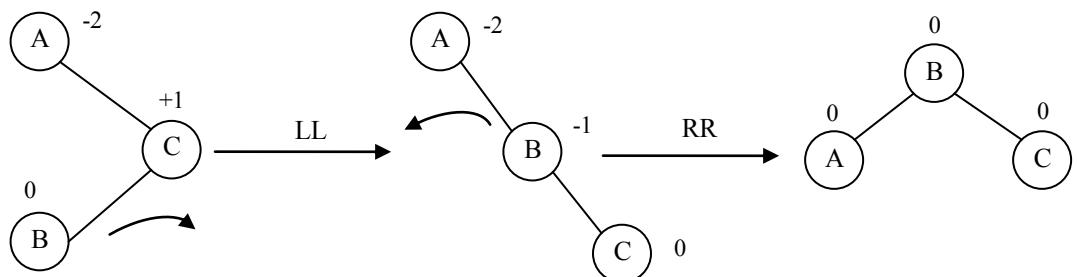


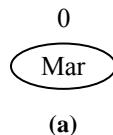
Figure 8.35: RL Rotation

Example:

Insertion of following items in AVL tree:

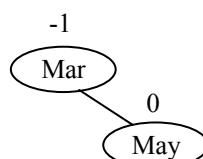
Mar, May, Nov, Aug, Apr, Jan, Dec, Jul, Feb, Jun, Oct, Sep

Insert Mar:



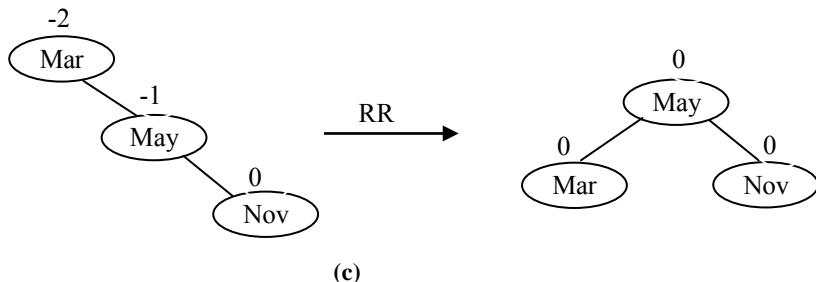
(a)

Insert May:



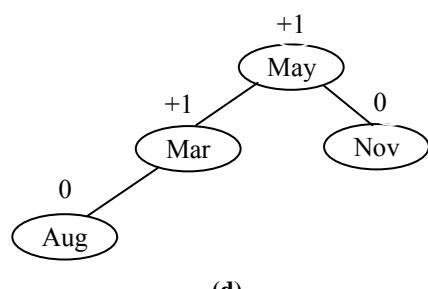
(b)

Insert Nov:



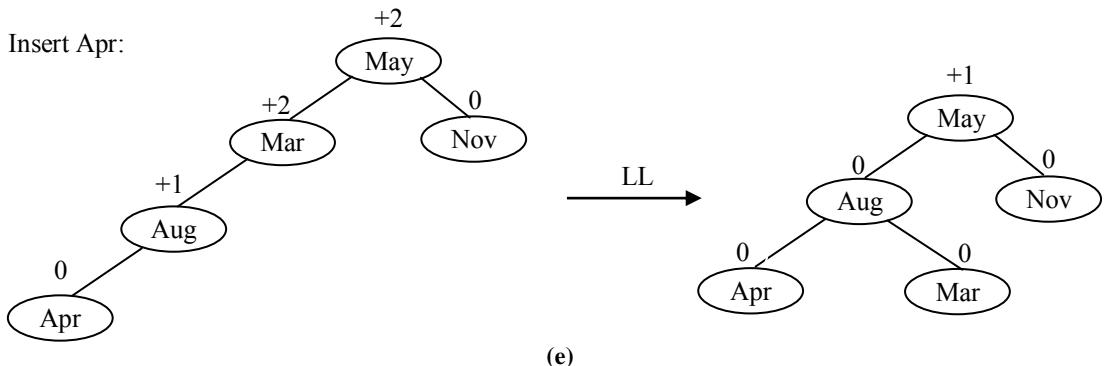
(c)

Insert Aug:



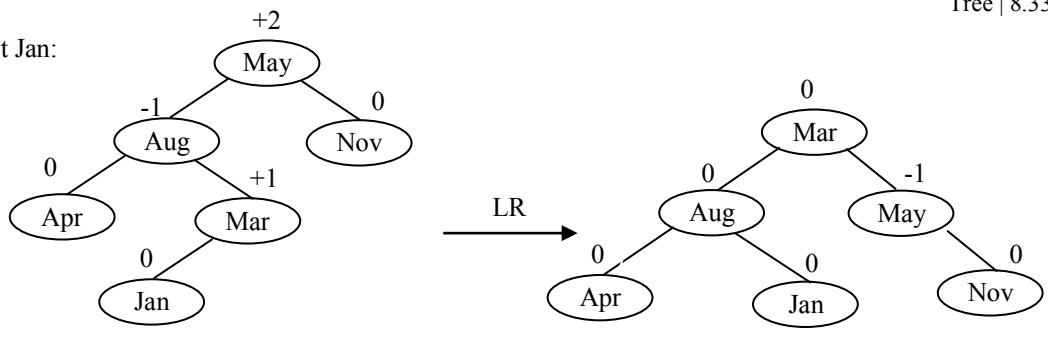
(d)

Insert Apr:

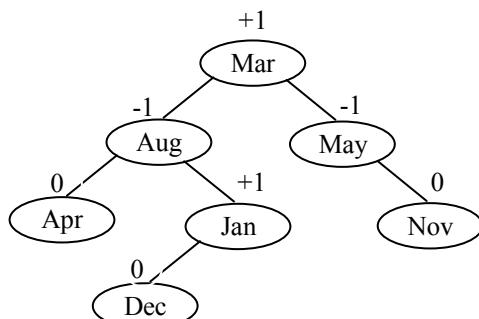


(e)

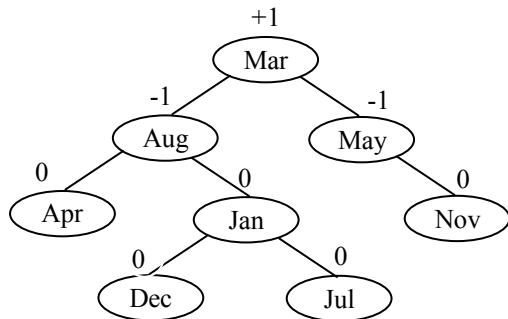
Insert Jan:



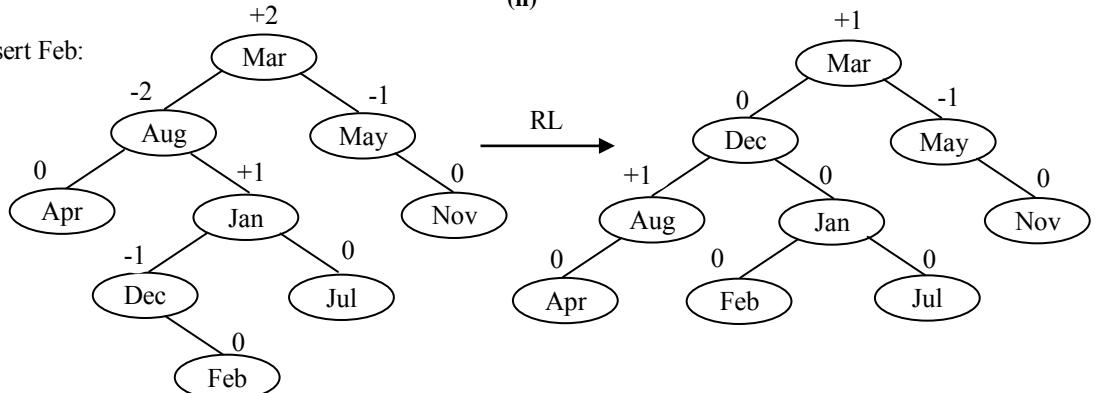
Insert Dec:



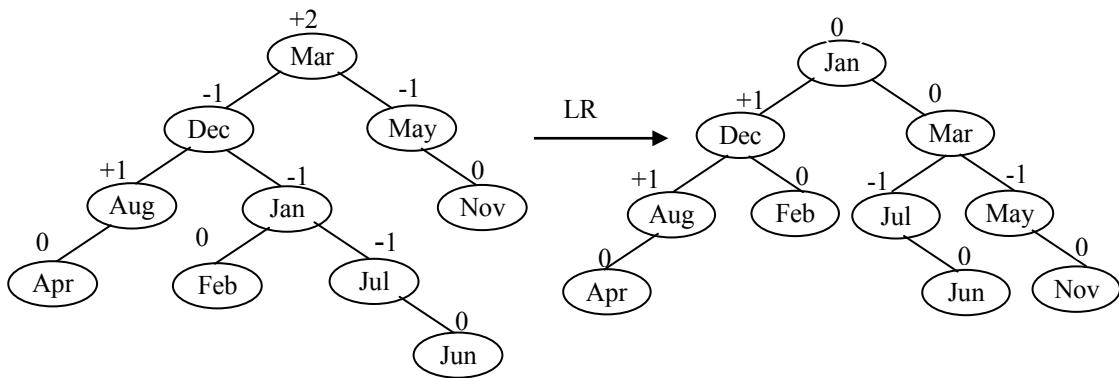
Insert Jul:



Insert Feb:

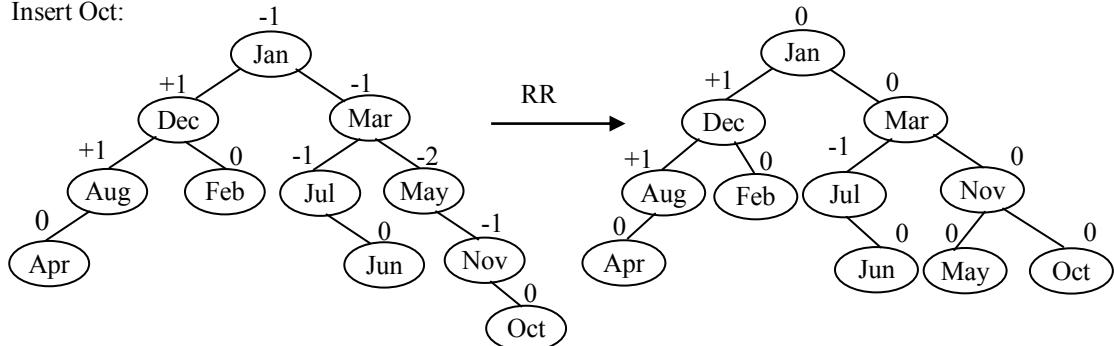


Insert Jun:



(j)

Insert Oct:



(k)

Insert Sep:

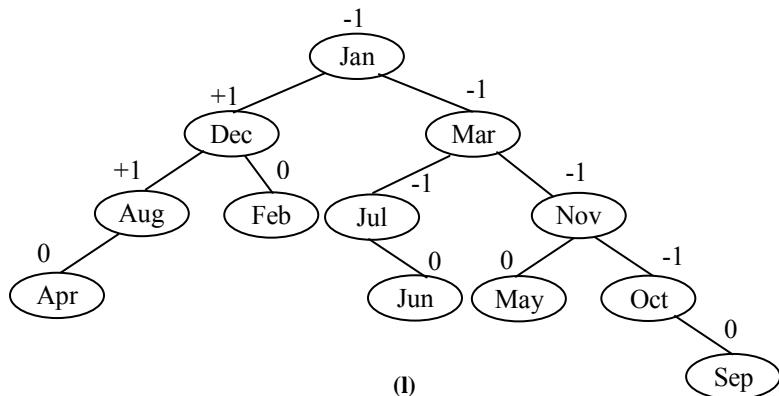


Figure 8.36 (a-l): Insertion in AVL Tree

Deletion in an AVL Tree

The deletion operation in an AVL tree is performed using the following steps:

- Search the AVL tree to find out the node to be deleted as in the Binary Search Tree
- After deleting the node, check the balance factor of each node of the tree.
- Rebalance the AVL tree if the tree is unbalanced by using AVL rotations.

Suppose X is the deleted node from the AVL tree and A be the closest ancestor node on the path from X to the root node, with a balance factor ± 2 after deletion. For balancing the AVL tree, the rotation is classified as L or R depending on whether the deletion occurred on the left or right subtree of A. Moreover, depending on the balanced factor of B, where B is the root of the left or right subtree of A, the R or L rotations is classified as R0, R1 and R-1 or L0, L1 and L-1.

R0 Rotation

Suppose right subtree of node A is A_R of height h, left and right subtree of node B are B_L and B_R of height h each. Now a node X from A_R is deleted, then the height of A_R becomes $h-1$ and A be the nearest ancestor node of X, with balance factor +2. The node B is the root of the left subtree of A with balance factor 0 and node X deleted from the right subtree of A, therefore R0 rotation is used.

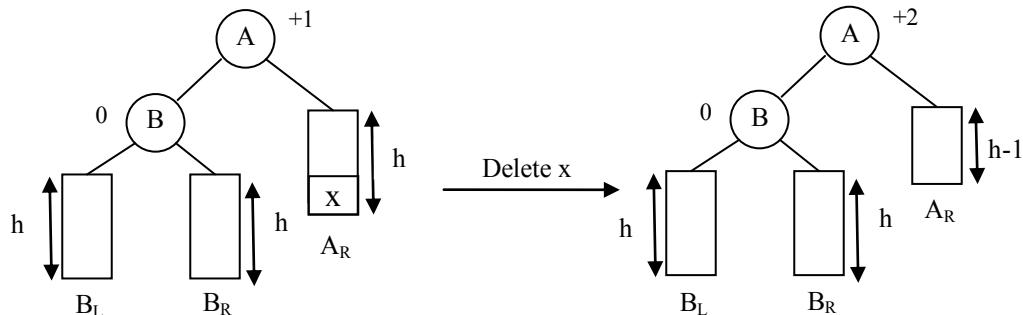


Figure 8.37: Before and after delete node X

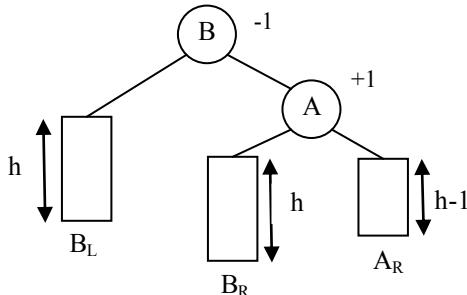
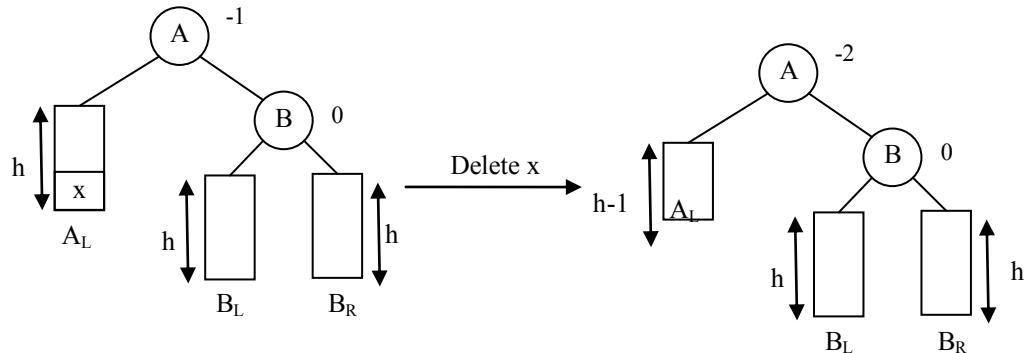
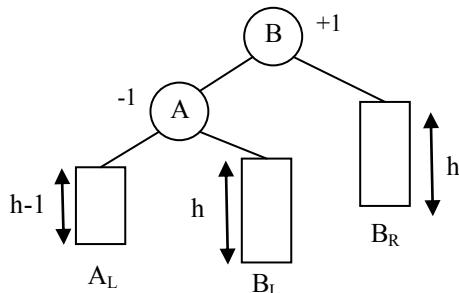


Figure 5.38: After R0 Rotation

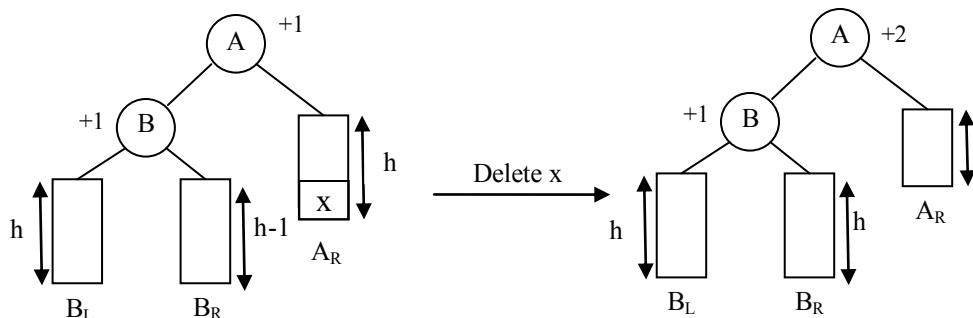
L0 Rotation

The left subtree of node A is A_L of height h, left and right subtree of node B are B_L and B_R of height h each. Now a node X from A_L is deleted, then the height of A_L becomes $h-1$ and A be the nearest ancestor node of X, with balance factor -2. The node B is the root of the right subtree of A with balance factor 0 and node X deleted from left subtree of A, therefore L0 rotation is used. L0 rotation is similar with R0 rotation.

**Figure 8.39:** Before and after delete node X**Figure 8.40:** After L0 Rotation

R1 Rotation

Suppose right subtree of node A is A_R of height h, left and right subtree of node B are B_L and B_R of height h and h-1 respectively. Now a node X from A_R is deleted, then the height of A_R becomes h-1 and A be the nearest ancestor node of X, with balance factor +2. The node B is the root of the left subtree of A with balance factor +1 and node X deleted from the right subtree of A, therefore R1 rotation is used.

**Figure 8.41:** Before and after delete node X

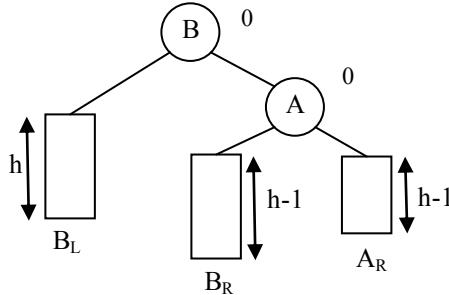


Figure 8.42: After R1 Rotation

L-1 Rotation

The left subtree of node A is A_L of height h , left and right subtree of node B are B_L and B_R of height $h-1$ and h respectively. Now a node X from A_L is deleted, then the height of A_L becomes $h-1$ and A be the nearest ancestor node of X, with balance factor -2. The node B is the root of the right subtree of A with balance factor -1 and node X deleted from left subtree of A, therefore L-1 rotation is used. L-1 rotation is similar with R1 rotation.

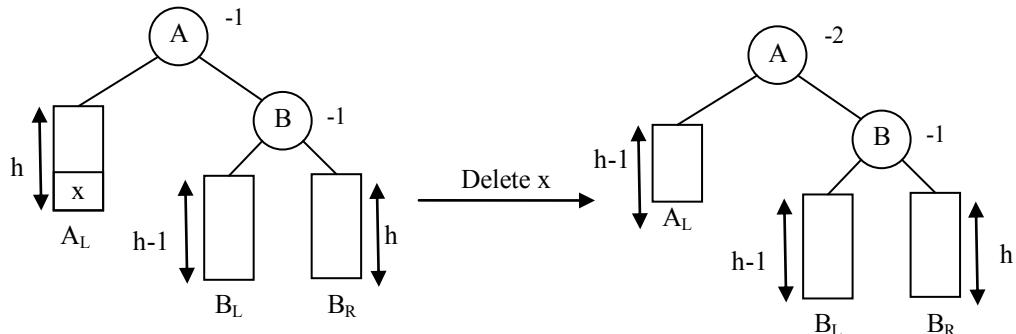


Figure 8.43: After L-1 Rotation

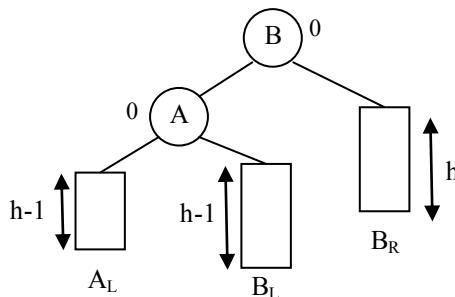


Figure 8.44: Before and after delete node X

R-1 Rotation

The right subtree of node A is A_R of height h , left subtree of node B is B_L of height $h-1$, left and right subtree of node C are C_L and C_R of height $h-1$ each. Now a node X from A_R is deleted, then the height of A_L becomes $h-1$ and A be the nearest ancestor node of X, with balanced factor +2. The node B is the

root of the right subtree of A with balance factor -1 and node X deleted from the right subtree of A, therefore R-1 rotation is used.

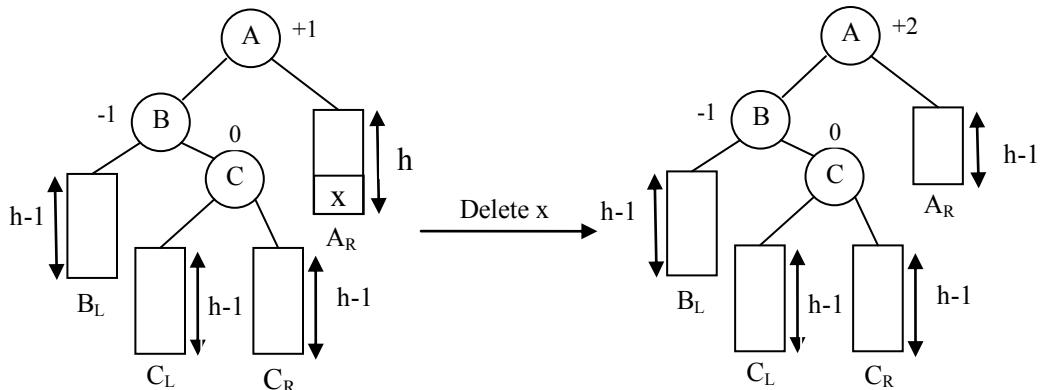


Figure 8.45: Before and after delete node X

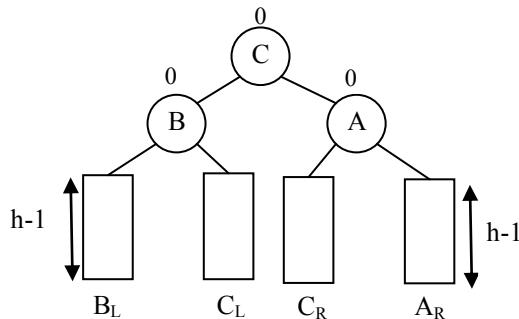


Figure 8.46: After R-1 Rotation

L1 Rotation

The left subtree of node A is A_L of height h , right subtree of node B is B_R of height $h-1$, left and right subtree of node C are C_L and C_R of height $h-1$ each. Now a node X from A_L is deleted, then the height of A_L becomes $h-1$ and A be the nearest ancestor node of X, with balance factor -2. The node B is the root of the right subtree of A with balance factor +1 and node X deleted from the left subtree of A, therefore L1 rotation is used. L1 rotation is similar with R-1 rotation.

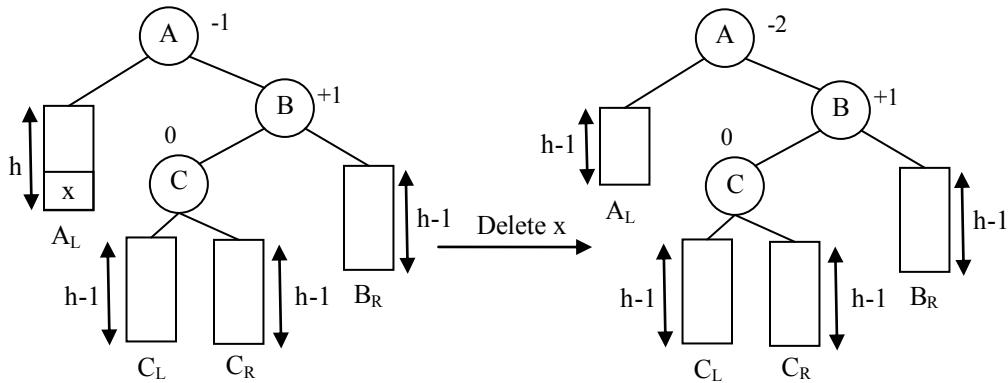


Figure 8.47: Before and after delete node X

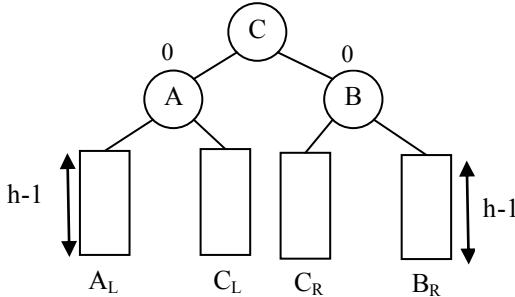


Figure 8.48: After L1 Rotation

Red-Black Tree

The red-black tree is a balanced binary search tree in which each node is colored either red or black. The red-black tree is introduced by **Guibas** and **Sedgewick**, in 1978. The tree is named after a red-black tree, as they had red and black pens to draw the trees.

In the red-black tree, ordering of the keys are the same as for binary search trees: all the keys to the left of a node are smaller, and all the keys to the right of a node are larger than the key at the node itself. The number of black nodes on every path from the root to each leaf is the same, called the black height of the tree. There are no two adjacent nodes are red. When a node is red, then both its children are black.

In insertion and deletion operation, when the tree is modified, the tree is rearranged and recolored to restore the properties. The tree is designed in such a way that this rearranging and recoloring can be performed efficiently. The searching, insertion, deletion operations are performed in $O(\log n)$ time. One extra bit per node is required to represent the colour (red or black) of the node.

Number of Binary Trees

Now, we count the binary trees having n nodes. When $n = 0$, then there is one empty binary tree and when $n = 1$ then there is only one binary tree with one node. If $n = 2$ then there are two distinct binary

trees. If $n = 3$ then there are five distinct binary trees.

Consider the nodes of a binary tree of n nodes are numbered from 1 to n . The inorder permutation defined by such a binary tree is the order in which its nodes are visited during an inorder traversal of the tree. A preorder permutation can be defined similarly.

Now, if the nodes of a binary tree are numbered such that its preorder permutation is 1, 2... n then distinct binary tree define distinct inorder permutation. For example, if the nodes of binary tree numbers 1, 2, 3 then possible permutations obtain by a stack are

1, 2, 3

1, 3, 2

2, 1, 3

2, 3, 1

3, 2, 1

It is not possible to obtain 3, 1, 2. Each of these five permutations corresponding to one of the five distinct binary trees with three nodes.

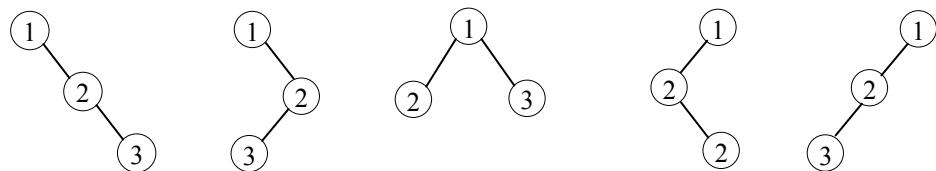


Figure 8.49: Five distinct binary trees with three nodes

It can calculate the total numbers of binary trees of n nodes by using Catalan numbers from a sequence of natural numbers

$$C_n = \sum_{i=1}^n C_{i-1} C_{n-i} = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \text{ for } n \geq 0$$

The Catalan numbers for $n = 0, 1, 2, 3, 4, 5, 6\dots$ are 1, 1, 2, 5, 14, 42, 132...

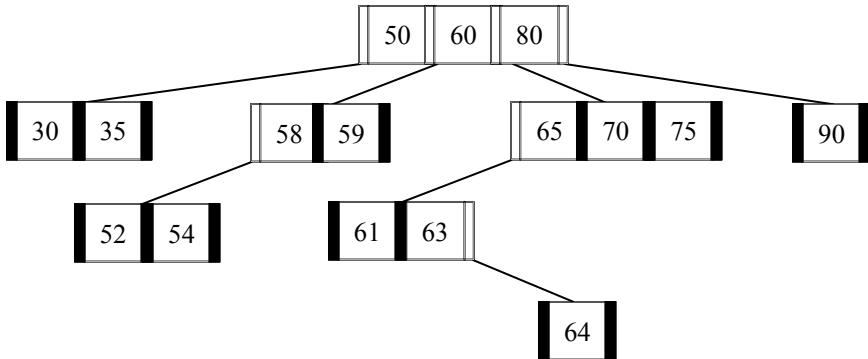
Multi-way Search Tree or M-way Search Tree

Although it is possible to use binary trees for fast searching, insertion and deletion in the internal memory (i.e. main memory), but binary trees are not appropriate for data that stored in external memory (i.e. storage devices such as disks etc.). When accessing data on a disk, an entire block (or page) of data is input at once. Therefore, it makes sure to design such tree special so that each node of the tree essentially occupies one entire block. Multi-way search tree, B-tree, B+ tree are examples of such data structures.

An m-way search tree (also known as multi-way search tree) of order m is a tree, which is either empty or for a non-empty m-way search tree has following properties:

- i) All the nodes are of maximum m degree and any node contains maximum $m-1$ values.
- ii) The nodes contain 1 to $m-1$ distinct keys K_1, K_2, \dots, K_{m-1} and keys in each node are sorted such that $K_1 < K_2 < \dots < K_{m-1}$.
- iii) A node with n values has $n+1$ subtrees such that $A_0, A_1, A_2, \dots, A_n$. The subtrees may be empty.
- iv) All the key values in the subtree A_i of a node $[K_1, K_2, \dots, K_n]$ may hold only values K in the range $K_i \leq K \leq K_{i+1}$
- v) All key values in the leftmost subtree A_0 are less than K_1 .
- vi) All key values in the rightmost subtree A_n are greater than K_n .
- vii) All the subtrees are also m-way search trees.

However, m-way search trees can become unbalance.

**Figure 8.50:** M-way search Tree of order 4

B-tree

B-tree is a good example of a data structure for external memory; it uses multi-way search tree rather than a binary tree. The B-tree optimized the system to read and write a large block of data to external memory because they allow for large amounts of data to be stored in a node. B-tree performs searching, insertion and deletion operation in $O(\log n)$ time. B-tree is commonly used in databases and file systems.

B-trees are balanced trees and an m-way search tree of order m (where $m \geq 3$). B-tree is a generalization of a binary search tree in that a node can have more than two children and only one key (where $m = 2$).

Definition: B-tree can be defined as follows:

A B-tree is either empty or for a non-empty B-tree satisfies the following properties:

- i) All the non-terminal nodes, except root node, each node has maximum m children and minimum $\lceil m/2 \rceil$ children. The non-terminal root node has maximum m children and minimum two children.
- ii) If a node has m number of children then it must have $m-1$ distinct key values K_1, K_2, \dots, K_{m-1} and all the key values of each node are in sorted order such that $K_1 < K_2 < \dots < K_{m-1}$.
- iii) All the key values in the subtree A_i of a node $[K_1, K_2, \dots, K_n]$ may hold only values K in the range $K_i \leq K \leq K_{i+1}$
- iv) Each terminal nodes must contain minimum $\lceil m/2 \rceil - 1$ keys.
- v) All the terminal nodes must be at same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced. Therefore, in B-trees do not need re-balancing as frequently as other self-balancing search trees but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of children nodes are typically fixed for a particular implementation.

For example, for a B-tree of order 4, i.e. when $m = 4$ then each internal nodes has from 2 to 4 children and from 1 to 3 keys. Another B-tree order of 7, i.e., when $m = 7$ then each internal nodes has from 4 to 7 children and from 3 to 6 keys (except the root node which may have as few as 2 children and 1 key).

The structure definition for a node of B-tree as follows:

```
struct BTreenode
{
    int nchild;
    struct *BTreenode [M];
    int key[M-1];
};
```

Operations on B-tree

Operations supported by a B-tree are as follow:

Table 8.5: Various Operation on B-tree

Operation	Description
Traverse	This operation traversing all the nodes of B-tree exactly once.
Insertion	This operation inserts a node in the B-tree.
Deletion	This operation removes a node from the B-tree.
Searching	This operation searches a desired key value within the B-tree.

Searching in B-tree

Searching in a B-tree for a key is the generalization form of the searching in Binary Search Tree. An internal node with keys $K_1 < K_2 < \dots < K_{m-1}$ can search for a searching key k in either linearly or by binary searching. If the key is found in the node then searching is over. Otherwise, determine the index i such that $K_i < k < K_{i+1}$ and recursively search the subtree A_i . Finally, search all the keys in the leaf node. If the key is not found there then searching is unsuccessful.

Insertion in a B-tree

Insertion operation into a B-tree of order m is performed by searching the appropriate leaf node where we insert the key. There are three cases for insert a key into B-tree.

Case 1: when the leaf node is not at full capacity (i.e. it has fewer than $m - 1$ keys), then simply insert it into proper position so that after insertion the keys remains sorted. Therefore, it requires to movement keys within leaf node for the newly inserted key. Since m is assumed a constant, so the constant time overhead may be ignored.

Case 2: When the node is full (i.e. it has exactly $m - 1$ keys), then check its immediate left or right siblings in the B-tree, which may have fewer than the maximum of $m - 1$ keys. Suppose Q is the right siblings has less than $m - 1$ keys and P is the node has $m - 1$ keys. Now, after inserting the new key, transfer the key k of the parent node to the Q node and make it minimum key. The maximum key transfers from P and places it in the parent node. Finally, transfer the rightmost subtree from P to become leftmost subtree of Q . At this point, the B-tree is balanced and no further rebalancing is required. This method is known as **key rotation**. Similarly, when left siblings have less than $m - 1$ keys then key rotation also can be performed.

Case 3: When the node along with its both siblings is full, then key rotation is not possible. Suppose, after inserting the new key, the node becomes overfull with k keys. Now, split this node into three parts, one with the smallest $\lceil (m - 1)/2 \rceil$ keys, a single central key and one with the largest $\lfloor (m - 1)/2 \rfloor$ keys. Then, create a new B-tree node with the smallest keys, insert the single central node

into parent node and leave the largest keys in the existing node. At this point, the parent node may be overfull, and then repeat this process recursively. This method is known as **node split**.

Example:

Insert the following keys in a B-tree of order 4

74, 72, 19, 87, 51, 10, 35, 18, 39, 60, 76, 58, 19 and 45

Insert 74:



(a)

Insert 72:



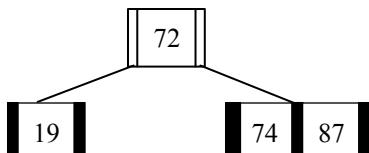
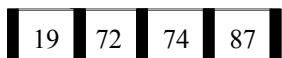
(b)

Insert 19:



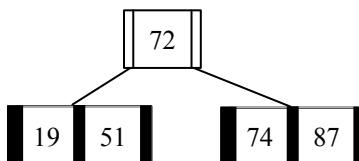
(c)

Insert 87:



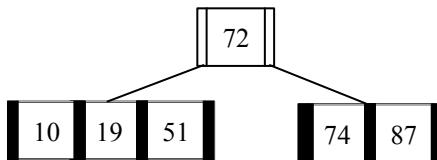
(d)

Insert 51:



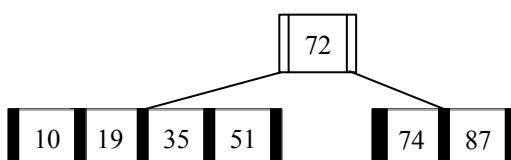
(e)

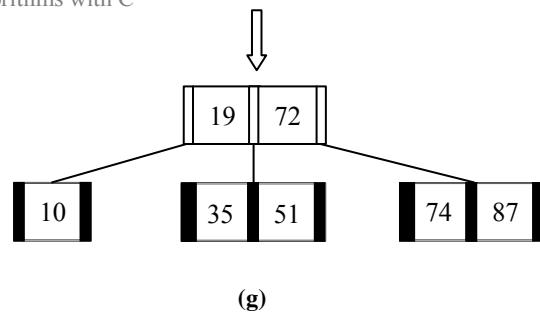
Insert 10:



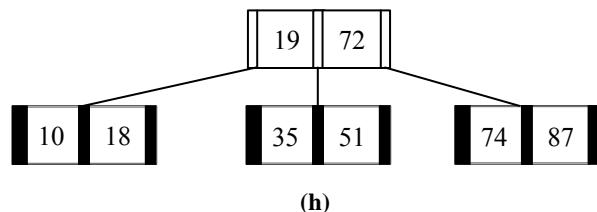
(f)

Insert 35:

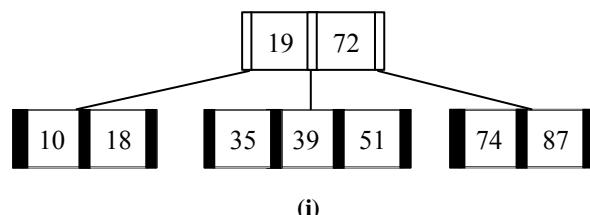




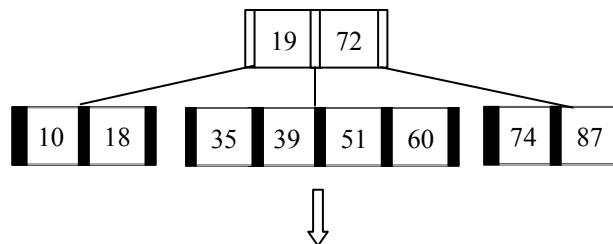
(g)

Insert 18:

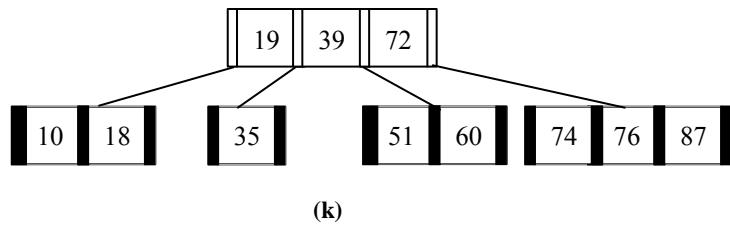
(h)

Insert 39:

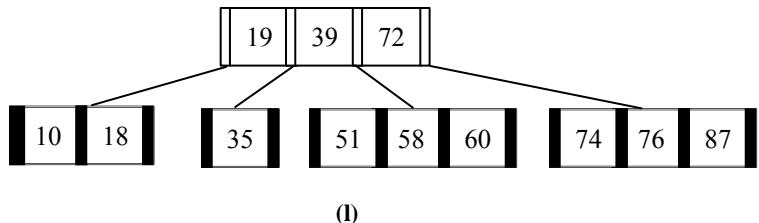
(i)

Insert 60:

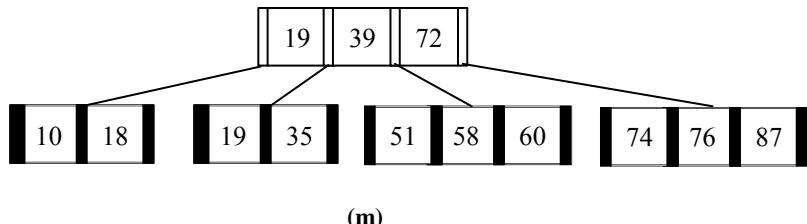
(j)

Insert 76:

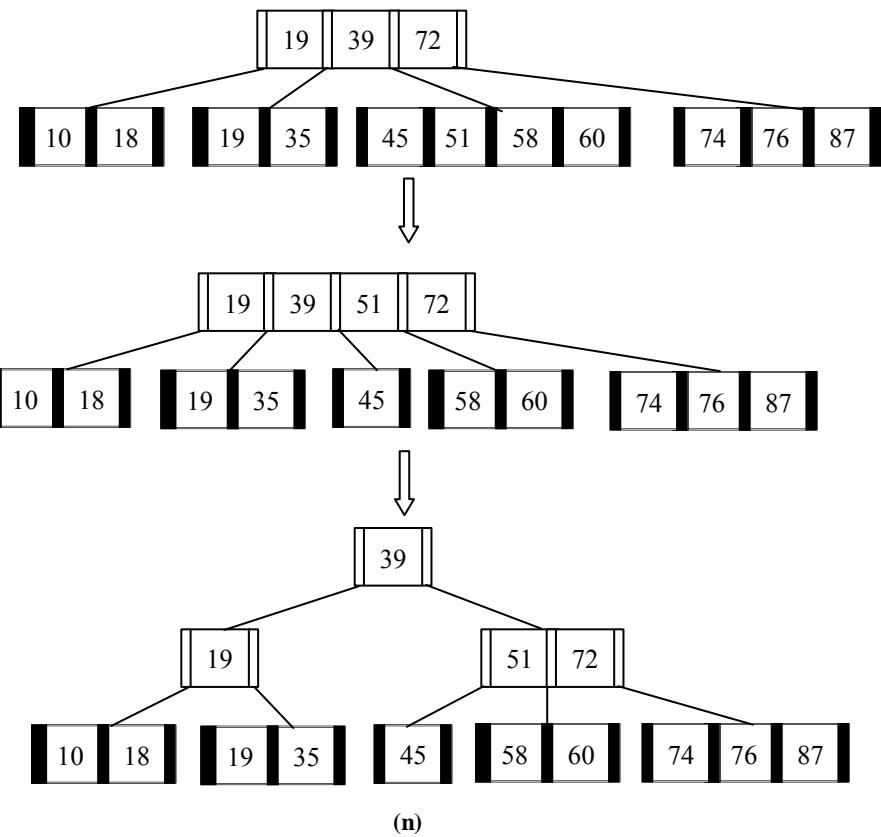
(k)

Insert 58:

(l)

Insert 19:

(m)

Insert 45:

(n)

Figure 8.51 (a-n): Insertion in B-tree

Example:

Insert the keys in a B-tree of order 3: p, q, r, d, h, m, l, s, k and n

Insert p:



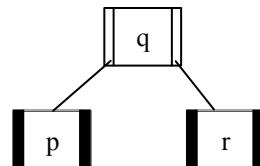
(a)

Insert q:



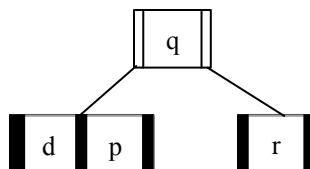
(b)

Insert r:



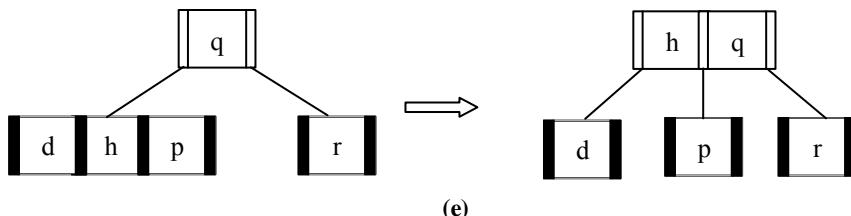
(c)

Insert d:



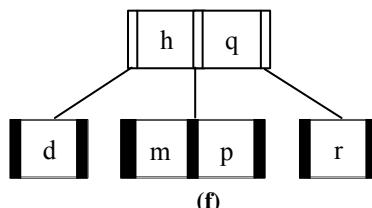
(d)

Insert h:



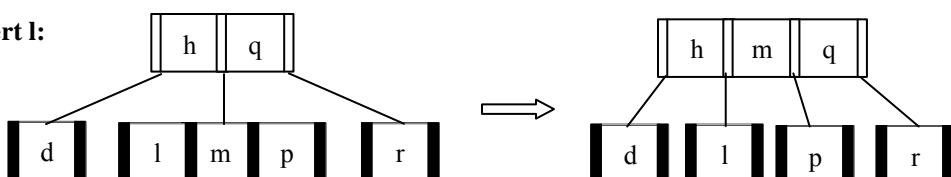
(e)

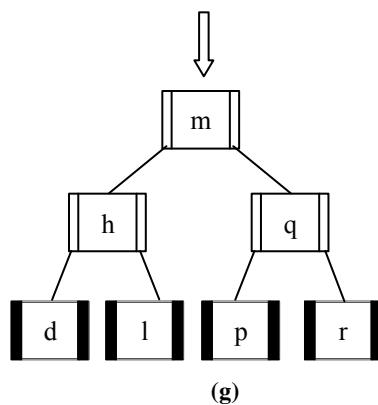
Insert m:



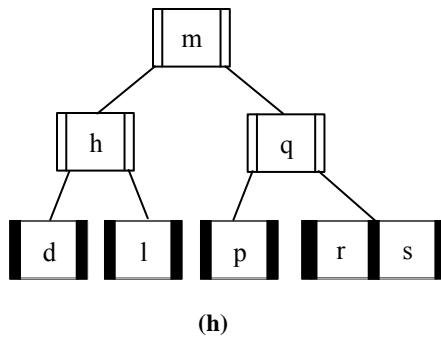
(f)

Insert l:

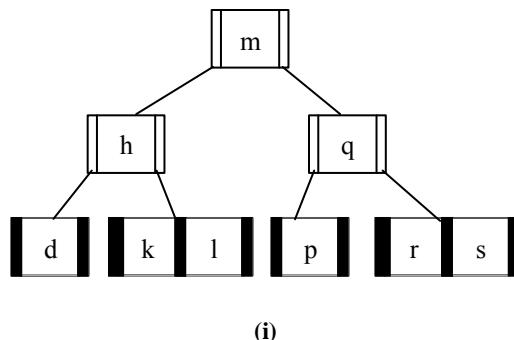




Insert s:



Insert k:



Insert n:

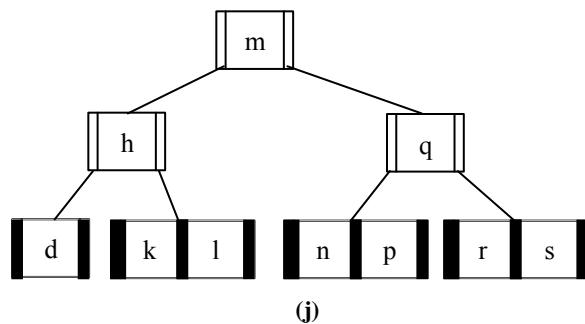


Figure 8.52 (a-j): Insertion in B-tree

Deletion in a B-tree

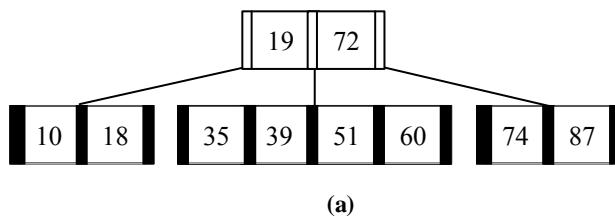
Deletion operation into a B-tree of order m is performed by searching the node where we delete the key. If the key is found then delete the key from the node. When an underflow (a leaf node has less than $\lceil m/2 \rceil - 1$ entries or an internal node has less than $\lceil m/2 \rceil$ nonempty subtrees) occurred after the key deletion, then an adjustment must be done. Now two cases need to consider, whether the key at leaf node or non-leaf node.

Case 1: When deleting a key in a non-leaf node, we must find substitute data. We use the immediate predecessor, which is the largest node in the left subtree of the entry to be deleted. In the subtree, the largest node is the rightmost subtree.

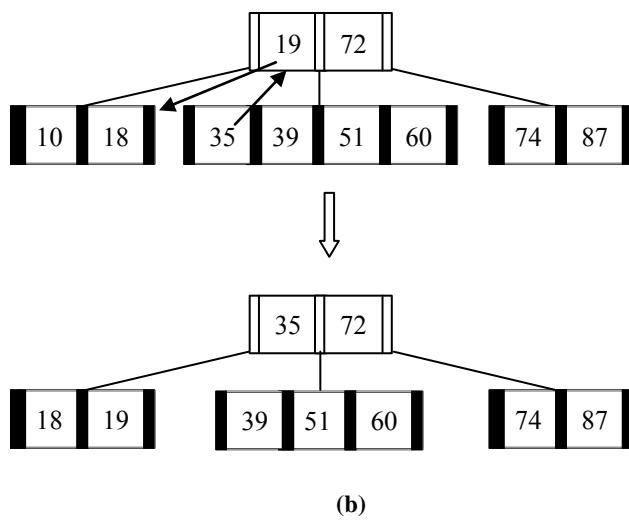
Case 2: When deleting a key in a leaf node and it contains more than the minimum number of entries, then delete the key without any adjustment. However, when a node is underflow, we need to do some adjustment, which is called reflow. Suppose one of the subtree contains underflow node, two situations need to consider: If the other subtree has more entries than the minimum number, then just move some entry from the subtree to the underflow node, which is called balance. If the other subtree only has a minimum number of entries, then we need to combine two nodes to one node together with the root entry. This is called combine.

Example:

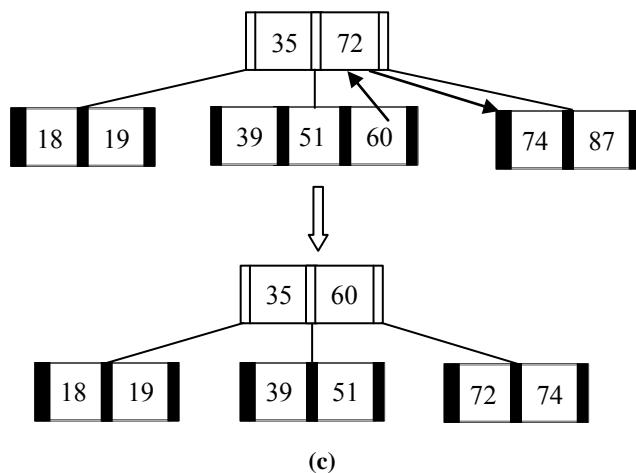
Delete the keys from the following B-tree: 10, 87, 39, 60, 72, 35.



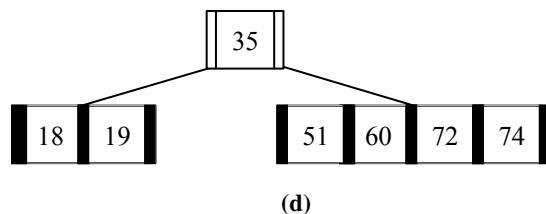
Delete 10: Borrow from right



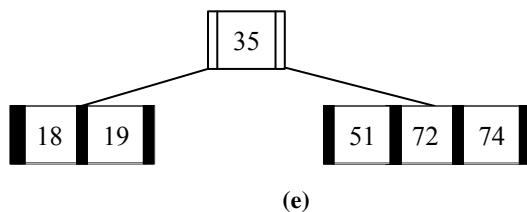
Delete 87: Borrow from right



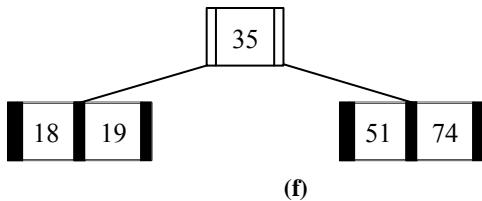
Delete 39: Combine 51, 60, 72, 74



Delete 60:



Delete 72:



Delete 35: At first copy 19 to parent node, then delete 19 from leaf node and finally combine 18, 19, 51 and 74.



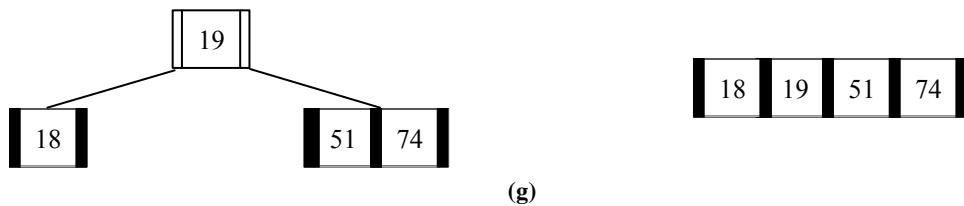


Figure 8.53 (a-g): Deletion in B-tree

2-3 Tree

The 2-3 tree is a tree, which is either empty or non-empty balance tree has following properties:

- Each non-terminal node has either two children with one data element (2-node) or three children with two data items (3-nodes).
- The terminal nodes have one or two data elements.
- All the keys values of each node $[K_1, K_2]$ are in sorted order ($K_1 < K_2$).
- The key values in the left subtree of a 3-node $[K_1, K_2]$ are less than K_1 , the key values in the right subtree are greater than K_2 and the middle subtree contains only the k values in the range $K_1 \leq k \leq K_2$.
- The key values in the left subtree of a 2-node $[K]$ are less than K , the key values in the right subtree are greater than K .
- All the terminal nodes must be at same level.
- All the subtrees are also 2-3 trees.

The 2-3 trees are B-tree of order 3, like B-tree they can perform search, insert and delete in $O(\log n)$ time.

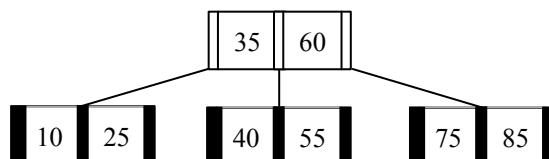


Figure 8.54: Complete 2-3 Tree

2-3-4 Tree

The 2-3-4 tree is a tree, which is either empty or non-empty balance tree has following properties:

- Each non-terminal node has two children with one data element (2-node), three children with two data items (3-nodes) or four children with three data items (4-nodes).
- The terminal nodes have one, two or three data elements.
- All the keys values of each node $[K_1, K_2, K_3]$ are in sorted order ($K_1 < K_2 < K_3$).
- All key values in the leftmost subtree A_0 are less than K_1 .
- All key values in the rightmost subtree A_3 are greater than K_3 .
- All the key values in the subtree A_i of a node $[K_1, K_2, K_3]$ may hold only values K in the range $K_i \leq K \leq K_{i+1}$
- All the terminal nodes must be at same level.
- All the subtrees are also 2-3-4 trees.

The 2-3-4 tree also called 2-4 tree. It is a specialization version of B-tree of order 4. Like B-tree, in 2-3-4 tree, searching, insertion, deletion operations are performed in $O(\log n)$ time.

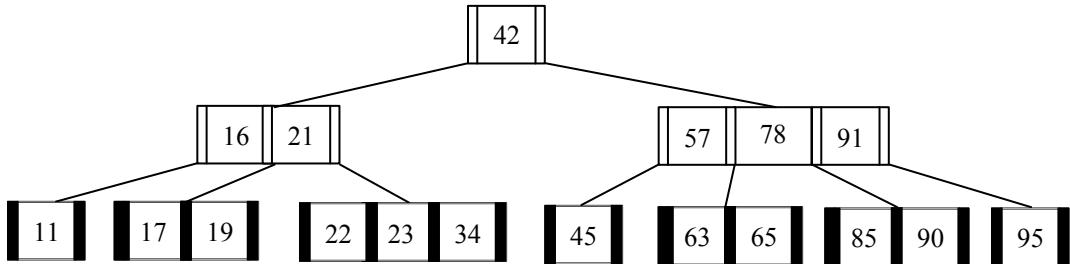


Figure 8.54: 2-3-4 Tree

Summary

- Tree is a non-linear data structure with a root node and subtrees that can be used to represent data items possessing a hierarchical relationship among the nodes of the tree.
- A binary tree is a finite set of nodes, which is either empty or consists of a root and two disjoint binary trees called left subtree and the right subtree.
- A Binary search tree is a binary tree, which is either empty or each node in the tree contains a key. Such that all keys in the left subtree of X are less than X, all keys in the right subtree of X are greater than X and the left and right subtrees are also binary search tree.
- AVL tree is a height-balanced binary tree where the heights of the two subtrees of any node differ by at most ± 1 . After insertion or deletion if they differ by more than one, then rebalancing is done to restore its property.

Exercises

1. What is a max heap?
2. Draw a minimum heap tree from the following list:
12, 11, 7, 3, 10, -5, 0, 9, 2
3. Prove that the maximum number of nodes in a binary tree of depth k is $2^k - 1$.
4. Show the following integers can be inserted in an empty binary search tree in the order they are given and draw the tree in each step.
50, 30, 10, 90, 100, 40, 60, 20, 110, 5
5. Prove that for any non-empty binary tree T, if n_0 is the number of leaves and n_1 be the number of nodes having degree 2 then $n_0 = n_1 + 1$.
6. What is a binary tree? Is it possible to represent binary trees with the help of array? If yes, then how?
7. Define AVL tree and give a sequence of AVL and non-AVL trees.
8. Insert the following keys in order given to build them into an AVL tree:
 - i) a, z, b, y, c, x, d, w, e, v
 - ii) k, m, u, t, v, p
 - iii) 12, 11, 13, 10, 9, 15, 14, 18, 7, 6, 5, 4
 - iv) 6, 3, 1, 2, 4, 5, 9, 7, 8, 10, 12

Clearly, mention different rotations used and a balanced factor of each node.

9. The inorder and preorder traversal sequence of nodes in a binary tree are given below:
- a) Inorder : E, A, C, K, F, H, D, B, G
Preorder : F, A, E, K, C, D, H, G, B
 - b) Preorder : A, B, D, I, E, J, C, F, G, K
Inorder : D, I, B, E, J, A, F, C, K, G
- Draw the actual binary tree. State briefly the logic used to construct the tree. Write its postorder traversal.
10. Construct a binary tree using the inorder and postorder traversal of the nodes given below:
- a) Inorder : D, B, F, E, A, G, C, L, J, H, K
Postorder : D, F, E, B, G, L, J, K, H, C, A
 - b) Inorder : A, B, C, D, F, G, K, L, N, P, T
Postorder : B, C, A, P, N, T, L, K, G, F, D
11. What are the differences between AVL tree and binary search tree?
12. Write an algorithm to insert a node in a given binary search tree.
13. What is binary search tree (BST)? State with an example the techniques for deletion at an element from a binary search tree.
14. Write a function to find the inorder successor of the root of a binary tree.
15. Write a non-recursive algorithm for inorder traversal of a binary tree.
16. Show the stages in the growth of 4 order B-tree when the following keys are inserted in the given order:
74, 72, 19, 87, 51, 10, 35, 18, 39, 60, 76, 58, 19, 45
17. What is a B-tree? Insert the following keys into a B-tree of order 3:
p, g, r, d, h, m, l, s, k, n
18. Construct an expression tree for the expression $E = (2x + y) * (5a - b)^3$.
19. What is a threaded binary tree? Write an algorithm for non-recursive inorder traversal of a threaded binary tree.
20. Show the steps in the creation of a height-balanced binary AVL tree using an insertion of items in the following order. Show the balancing steps required.
(March, May, November, August, April, January, December, July, February, June, October, September)
21. Write short notes on the following:
- a) Threaded binary tree
 - b) B-tree
22. What are the problems of a binary tree? Explain the improvement of performance by the use of the height-balanced tree. Explain how a height-balanced tree can be formed by inserting the following elements in the given order:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7, 11
- Show the root element that can be deleted from the above tree.
23. Choose the correct alternatives for the following:
- i) If a binary tree is threaded for inorder traversal a right NULL link of any node is replaced by the address of its
 - a) Successor
 - b) predecessor
 - c) root
 - d) own
 - ii) In a height-balanced tree, the heights of two subtrees of a node never differ by more than
 - a) 2
 - b) 0
 - c) 1
 - d) -1

- iii) Number of all possible binary trees with three nodes is
a) 3 b) 2 c) 4 d) 5

iv) Total nodes in a 2-tree (strictly binary tree) with thirty leaves is
a) 60 b) 58 c) 59 d) 57

v) Maximum possible height of an AVL tree with 7 nodes is
a) 3 b) 4 c) 5 d) 6

vi) The inorder and postorder traversal of a binary tree are DBEAFC and DEBFCA respectively. What will be the total number of nodes in the left subtree of the given tree?
a) 1 b) 4 c) 5 d) None of these

vii) The postorder traversal of some binary tree produces the sequence CDBFEA and the inorder traversal of the same produced the sequence CBDAFE. What will be the total number of nodes in the left subtree?
a) 2 b) 3 c) 4 d) 5

viii) A B-tree is
a) Always balanced b) an ordered tree c) a directed tree d) all of these

ix) A binary tree is a special type of tree
a) that is ordered b) such that no node has degree more than 2
c) for which both (a) and (b) above correct d) in which non-leaf nodes will have degree 2

x) The depth of a complete binary tree with n nodes
a) $\log(n+1) - 1$ b) $\log n$ c) $\log(n-1) + 1$ d) $\log(n) + 1$

xi) In a binary search tree, if the number of nodes of a tree is 9, then the minimum height of the tree is
a) 9 b) 5 c) 4 d) none of these

xii) Which of the following traversal techniques list the elements of a binary search tree in ascending order?
a) Preorder b) Postorder c) Inorder d) None of these

xiii) In traversing non-empty binary tree, visit the root node is made in the last in
a) Preorder b) Postorder c) Inorder d) None of these

* * * * *

CHAPTER 9

GRAPH

"Computers make it easier to do a lot of things, but most of the things they make it easier to do don't need to be done." -Andy Rooney

Graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is also used to model networks, complex data structures, scheduling, computation and a variety of other systems, where the relationship between objects in the system plays a key role.

A Graph $G = (V, E)$ consists of finite non-empty set of objects V , where $V(G) = \{v_1, v_2, v_3 \dots v_n\}$ called vertices and another set E where $E(G) = \{e_1, e_2, e_3 \dots e_m\}$ whose elements are called edges.

Definition: A Graph G is defined as an ordered set $G = (V, E)$, where V represents a set of elements called vertices (or points or vertices) and E represents a set of edges in G , that connects these vertices.

In the figure 9.1 shows a graph with five vertices, $V = \{A, B, C, D, E\}$ and six connecting edges, $E = \{(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)\}$.

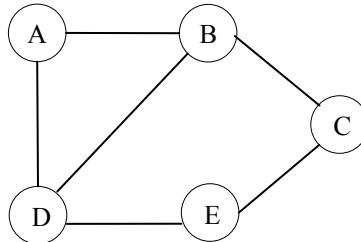


Figure 9.1: Graph with five Vertices and six Edges

Terminology of Graph

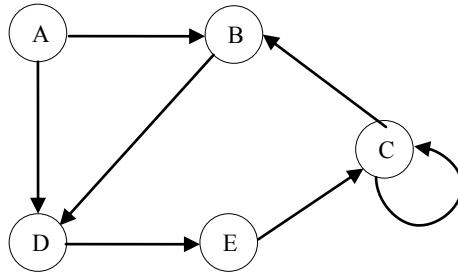
Undirected Graph: A Graph can be directed or undirected. In an undirected graph edge, do not have any direction associated with them. That is, if there is an edge between vertex A and B then the vertices can be traversed A to B, as well as B to A. Figure 9.1 shows an undirected graph.

Directed Graph: A directed graph or digraph is a graph, where the vertices are connected together and all the edges are directed from one vertex to another.

In a directed graph, edges form an ordered pair. If there is a directed edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is supposed to initiate from vertex A (initial vertex) and terminate at vertex B (terminal vertex). Figure 9.2 shows a directed graph.

KEY FEATURES

- book Categorization of Graph
- book Graph Representation
- book Graph traversals
- book Shortest Path
- book Spanning Tree
- book Application of Graph

**Figure 9.2:** Directed Graph with Five Vertices and self-loop

Adjacent vertices or neighbors: For every edge $e = (u, v)$, that connects vertices u and v , the vertices u and v are endpoints and are said to be the adjacent vertices or neighbors.

Degree: The degree of a vertex of a graph is the number of edges incident to the vertex, with self-loops counted twice. The degree of a vertex v is denoted by $\deg(v)$ or $\deg v$. If $\deg(v) = 0$, it means that v does not belong to any edge and such a vertex is known as an isolated vertex. In figure 9.2, $\deg(c) = 4$.

The maximum degree of a graph G , denoted by $\Delta(G)$ and a minimum degree of a graph, denoted by $\delta(G)$, are the maximum and minimum degree of its vertices.

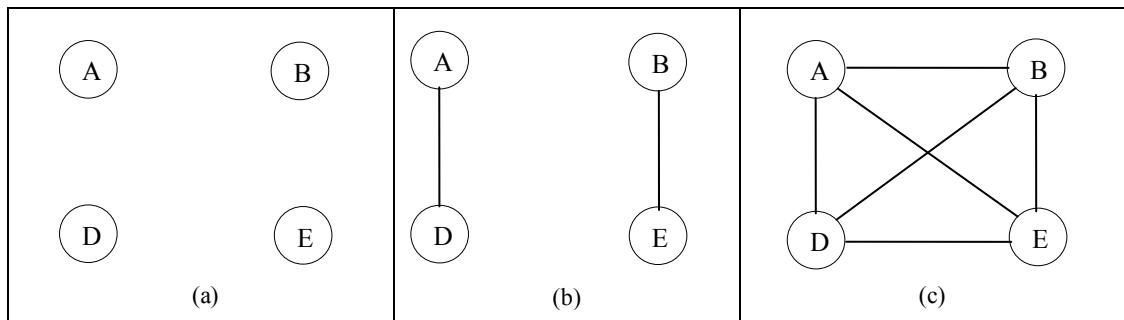
In a digraph, the number of edges coming out of a vertex is called the out-degree of that vertex. A number of edges coming in of a vertex is the in-degree of that vertex.

For a digraph $G = (V, E)$,

$\sum_{v \in V} \text{in_deg}(v) = \sum_{v \in V} \text{out_deg}(v) = |E|$, where $|E|$ means the cardinality of the set E (i.e. the number of edges).

For an undirected graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$.

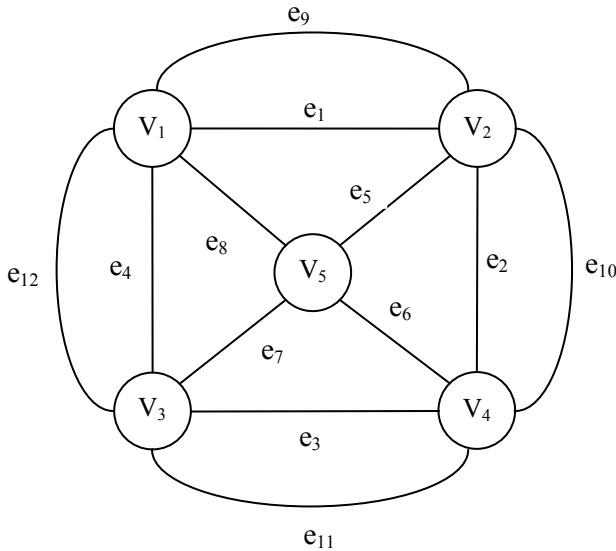
Regular graph: It is a graph, where each vertex has same no of neighbors. That is, every vertex has the same degree. A regular graph with the vertex of degree k is called a k -regular graph. Figure 9.3 shows some regular graphs.

**Figure 9.3:** Graph with Four Vertices (a) 0-regular graph, (b) 1-regular graph, (c) 3-regular graph

Walk: A walk of a graph is a finite altering sequence of vertices and edges beginning and ending with vertices. In a walk, no edge is traversed more than one.

Open Walk: An open walk is that where no edge is repeated. $V_1 e_1 V_2 e_5 V_5 e_7 V_3 e_3 V_4$ in the following graph G is an open walk as no edge is repeated.

Closed walk: A walk having same starting and end point is called closed walk. $V_1 e_1 V_2 e_2 V_4 e_{11} V_3 e_4 V_1$ is a closed walk where both starting and end vertex is V_1 .

**Figure 9.4:** Walk of Graph

Path: A path P has written as $P = \{v_0, v_1, v_2, \dots, v_n\}$, of length n from a vertex u to v is defined as a sequence of $(n+1)$ vertices. Here, $u = v_0$, $v = v_n$, and v_{i-1} is adjacent to v_i for $i = 1, 2, 3, \dots, n$.

Closed Path: A path is known as a closed path if the edge has same endpoints. In figure 9.4, V₁ V₂ V₄, V₃, V₁ is a closed path with same end point V₁.

**Figure 9.5:** (a) connected undirected graph, (b) connected directed graph

Simple/Open Path: A path is known as a simple path if all the vertices in a path are distinct. In figure 9.4, V₁ e₁ V₂ e₅ V₅ e₇ V₃ e₁₁ V₄ is an open path as no vertex is repeated.

Cycle: A path in which the first and last vertices are same and no repeated edges or vertices. In figure 9.4, V₁ e₁ V₂ e₂ V₄ e₁₁ V₃ e₁₂ V₁ form a circuit or circle. A graph is said to be **acyclic** if it contains no cycles. A directed graph that is acyclic is called **Directed Acyclic Graph (DAG)**.

Connected Graph: A graph is called connected graph, if and only if there is a simple path between

any two vertices of a graph. A connected graph without any cycle is a tree. Figure 9.5 (a) is an example of a connected graph.

Complete Graph: A graph is said to be complete if there is a path between one vertex and every other vertex in the graph. A complete graph with n number of vertices has $n(n-1)/2$ edges. In the above figure 9.5 graph G is a complete undirected graph.

Clique: In an undirected graph $G = (V, E)$, a clique is a subset of the vertex set, such that for every two vertices in this set, there is an edge that connects two vertices.

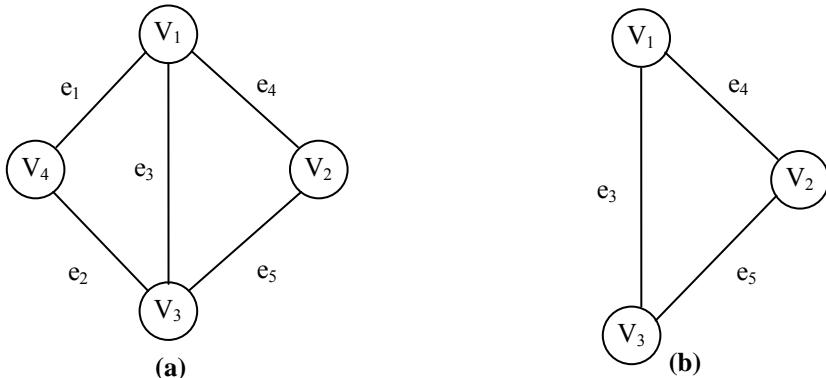


Figure 9.6: (b) Clique of Connected Undirected graph (a)

Weighted Graph: A graph is called weighted if every edge in the graph is assigned some weight data. The edge weight generally denotes by $w(e)$ is a positive value, which indicates the cost of traversing the edge in figure 9.8c.

Parallel or Multiple Edges: Distinct edges, which connect the same endpoints, are called multiple edges. That is, $e = (u, v)$ and $e' = (u, v)$ are known as multiple edges of a graph.

Loop: An edge has identical endpoints is called a loop. That is $e = (u, u)$ given in figure 9.1.

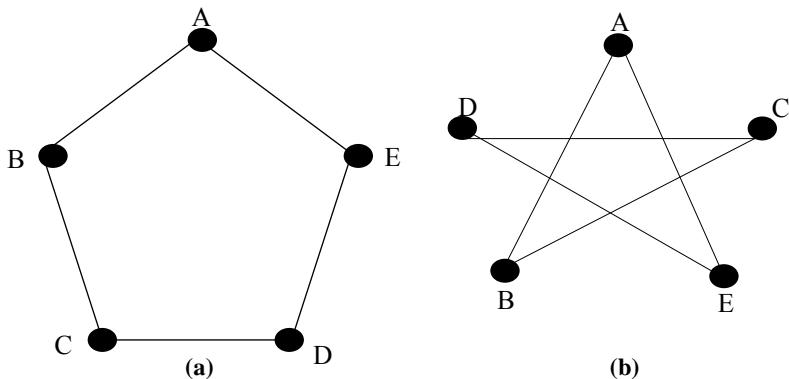


Figure 9.7: (a) and (b) are isomorphic graph

Isomorphism: Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be isomorphism graphs if there exists a one-to-one correspondence between their vertices and their edges such that the incidence relationship preserved, i.e. they contain the same number of vertices connected in the same way, the same number of edges and both have same degree sequences. In figure 9.7 example of isomorphism is given.

A directed graph G, also called as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of vertices in G. For an edge (u, v) .

- The edge begins at u and terminates at v.
- u is known as an initial vertex of e, whereas v is the terminal vertex.
- u is the predecessor of v and correspondingly v is the successor of u.
- u and v vertices are adjacent to each other.

Multigraph: A graph with multiple edges and/or loops is called multi-graph shown in figure 9.8a.

Subgraph: A subgraph H of a graph G, is a graph whose vertices are a subset of the vertex set of G and whose edges are a subset of the edge set of G. $H \subseteq G$ if $V' \subseteq V$ and $E' \subseteq E$. Figure 9.6b is an example of the subgraph.

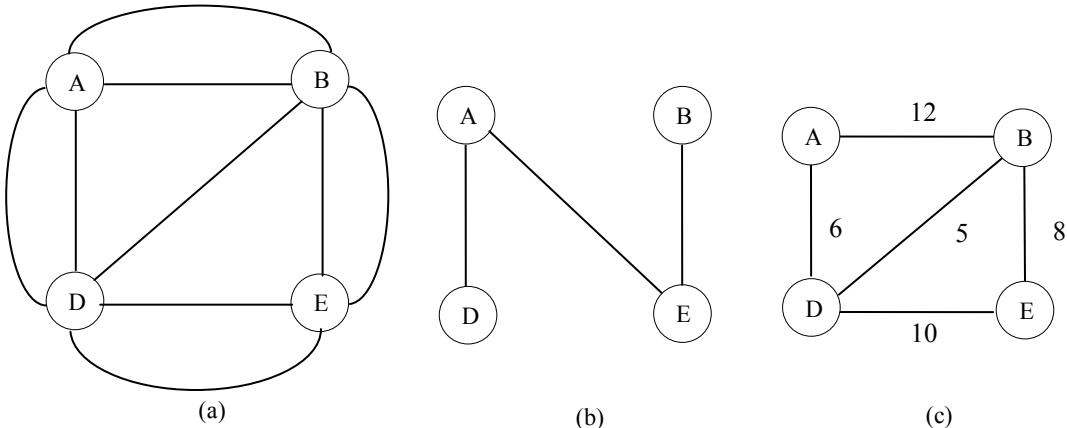


Figure 9.8: (a) Multi Graph, (b) Tree, (c) Weighted Graph

Lemma 1 (Hand Shaking Lemma): In a regular undirected graph the sum of the degree of all the vertices is twice the number of edges. That is for an undirected graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$

Proof: Let P be the proposition “In any graph, the sum of the degrees of all vertices is equal to twice the number of edges” for a graph $G = (V, E)$

$$P(n): \sum_{v \in G} \deg(|V|) = 2n \text{ where } |E|=n$$

Base case: $P(0): 2n=0|n=0$. Since there is not any edge the number of vertices must be equal to 1 or 0.

$\sum_{v \in G} \deg(|V|) = \deg(|V|) = 0$ the number of degrees is equal to 0. Thus $P(0)$ is true.

Induction step: Assuming that $P(n)$ is true for a given natural number. Let show that $P(n) \Rightarrow P(n+1)$.

$$P(n): \sum_{v \in G} \deg(|V|) = 2n$$

$$\sum_{v \in G} \deg(|V|) + 2 = 2n + 2$$

$$\sum_{v \in G} \deg(|V|) + 2 = 2(n + 1) \text{ which yields by adding 2}$$

Vertices of degree 1

$$\forall n \in N, \sum_{v \in G} \deg(|V|) = 2|E|$$

Terminology of a Directed Graph

Isolated vertex: The vertex with degree zero. Such vertex is not an end point of any edge. Vertex F in figure 9.9 is an isolated vertex.

Pendant vertex: A vertex is said to be pendant vertex if the vertex has degree one. It is also known as leaf vertex. Vertices G and H in figure 9.9 are pendant vertex.

Pendant edge: An edge is said to be pendant edge if the edge incident with the pendant vertex. The edge between G and H in figure 9.9.

Cut vertex: The vertex which when deleted would disconnect the remaining graph. If vertices B and D are deleted, then graph G become disconnected. So B and D are cut vertices in figure 9.9.

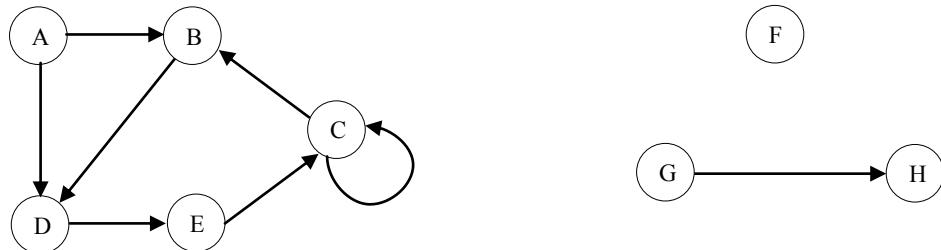


Figure 9.9: Directed Graph G

Source: The vertex with positive out-degree, but no in-degree. Vertex A and G in graph G has out-degree 2 and 1 respectively and in-degree 0. Therefore, they are source vertices in figure 9.9.

Sink: The vertex with positive in-degree, but no out-degree. Vertex H has 1 in degree and 0 out degree. Therefore, it is sink vertex in figure 9.9.

Reachability: A vertex v is said to be reachable from vertex u, if and only if there exists a path from vertex u to v. Vertex B is reachable from C but vertex G is not reachable from vertex C in figure 9.9.

Connected Graph: A graph G is a connected graph if, between every pair of vertices in G, there always exists a path in G.

Strongly connected directed graph: A digraph is said to be strongly connected, if and only if there exists a path between every pair of vertices. That is, if there is a path from vertex u to v then there must be a path from vertex v to u. Figure 9.10a is an example of a strongly connected graph.

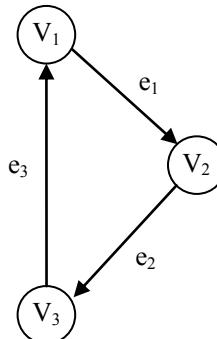


Figure 9.10a: Strongly Connected Directed Graph

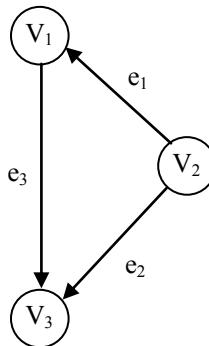


Figure 9.10b: Weakly Connected Directed Graph

Unilaterally connected graph: A digraph is said to be unilaterally connected, if there exists a path between any pair of vertices u, v in such that there is a path from u to v or a path from v to u , but not both. Figure 9.10a also is a unilaterally connected graph.

Weakly connected digraph: A digraph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any vertex from any other vertex by traversing edges in any direction. The vertices in a weakly connected digraph must have either out-degree or in-degree of at least 1. Figure 9.10b is an example of a weakly connected graph.

Lemma 2: Prove that every connected graph with n vertices has at least $n - 1$ edges.

Proof: This can be proved by induction

Base case: $n=1$

In such situation, only one vertex is there in a graph i.e. isolated vertex. There is no edge, so $n-1=0$.

Induction step: $n \geq 2$. Assume that there are $n-1$ number of vertices.

Let G is a graph with n vertices. Choose a vertex v from the set V and edge e from the set E where $e = \{v, w\}$ a unique edge.

Remove v and e from the graph G and a subgraph G' is formed where

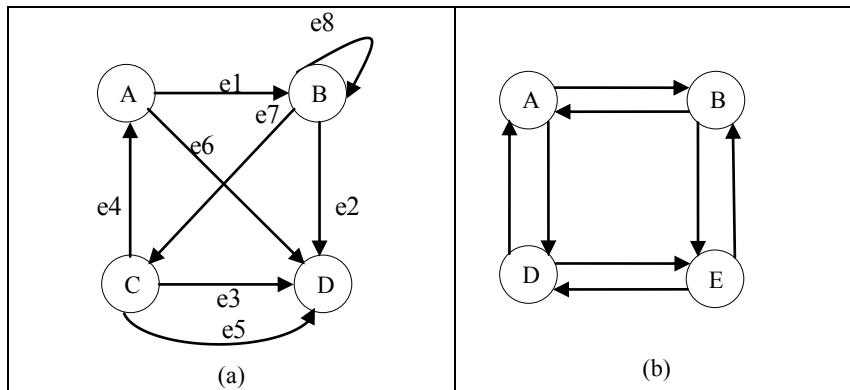
- (i) G' is connected with only one path went to/ from v .
- (ii) G' has no cycles
- (iii) So G' is a graph with $n-1$ vertices.

By the induction hypothesis G' has $n - 2$ edges.

Then G has $(n - 2) + 1 = (n - 1)$ edges.

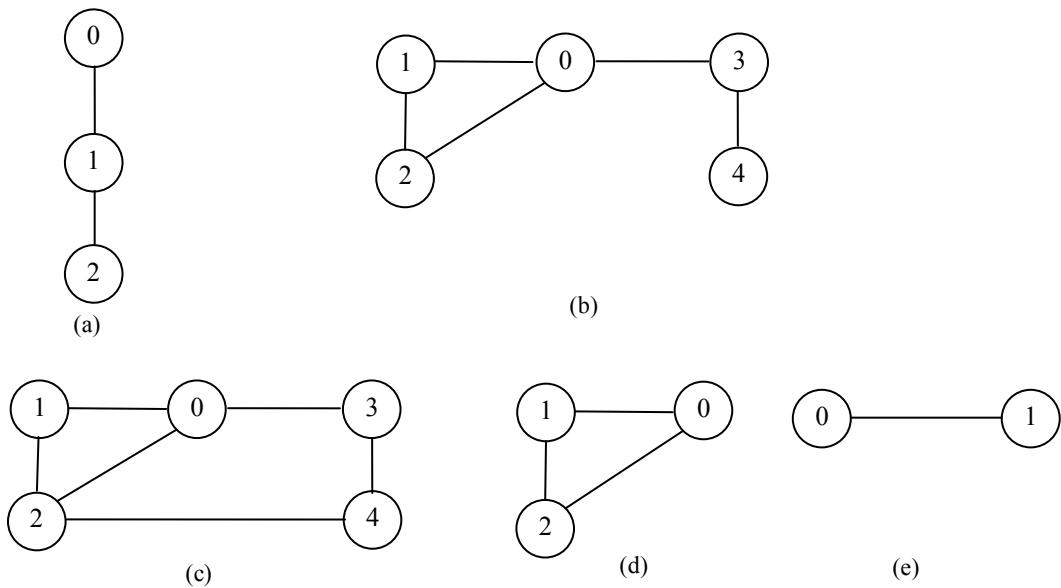
Simple directed graph: A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that is it cannot have more than one loop at a given vertex.

The graph G is a directed graph in which there are four vertices and eight edges. Note that, the edges e_3 & e_5 are parallel since both begin with C and end at D . The edge e_8 is a loop since both originates at single vertex B .

**Figure 9.11:** (a) Directed acyclic graph (b) Strongly connected directed acyclic graph

Biconnected Graphs

In graph theory, a **biconnected graph** is a connected and "non-separable" graph, meaning that if any vertex were to be removed, the graph will remain connected. Therefore, a biconnected graph has no articulation vertices.

**Figure 9.12:** (a, b) non-Bi Connected Graph, (c, d, e) Bi-Connected

Lemma 3: There are even numbers of vertices of odd degree.

Proof: From Hand Shaking Lemma, it is proved that for a graph $G = (V, E)$

$$\sum_{v \in V} \deg(v) = 2|E|$$

Partitioning the vertices into those of even degree and those of odd degree, we know

$$\sum_{v \in V} \deg(v) = \sum_{\deg(v) \text{ is even}} \deg(v) + \sum_{\deg(v) \text{ is odd}} \deg(v)$$

By the Handshaking Lemma, the value of the left-hand side of this equation equals twice the number of edges, and so is even. On the right-hand side, the first summand is even since it is a sum of even

values. Therefore, the second summand on the right-hand side must also be even. However, since it is given that the degree of the vertices must be odd, so a number of vertices should be even.

Operations on Graph

Operations supported by a graph are as follows:

Table 9.1: Different operations on graph

Operation	Description
Create	This operation representation/storing a graph in memory.
Traverse	This operation traverse/visit all the vertices of the graph exactly once.
Insertion	This operation inserts a vertex to the graph.
Deletion	This operation removes a vertex from the graph.
Searching	This operation performs searching for a key value in the graph.

Representation of Graphs

There are three common ways of representing a graph or storing a graph in computer memory. They are:

- *Sequential representation* by using an adjacency matrix.
- *Linked representation* by using adjacency lists using a linked list.
- *The adjacency multi - list, which* is an extension of linked representation.

Adjacency Matrix

Let $G(V, E)$ be a graph with n vertices where $n \geq 1$. The adjacency matrix of G is a two-dimensional $(n \times n)$ array say A , with the property that $A[i][j] = 1$, if the edge for undirected graph (V_i, V_j) is in $E(G)$ or for a directed graph an edge from V_i to V_j exist in $E(G)$. $A[i][j] = 0$, if there is no such edges in G . Adjacency matrix represents vertex to vertex relation. The adjacency matrix is also known as bit matrix or Boolean matrix.

Adjacency Matrix representation of graphs

An adjacency matrix is used to represent which vertices are adjacent to one another. By definition, two vertices are said to be adjacent, if there is an edge connecting them.

An adjacency matrix is a way of representing n vertex graph $G = (V, E)$ by a $n \times n$ matrix, a , whose entries are Boolean values.

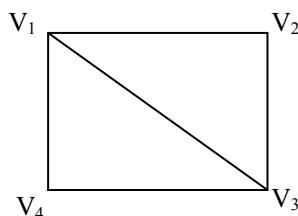
The matrix entry $a[i][j]$ is defined as

$$a[i][j] = \begin{cases} \text{true} & \text{if } (i, j) \in E \\ \text{false} & \text{otherwise} \end{cases}$$

Adjacency Matrix Representation for an Undirected Graph (G)

In an undirected graph, the adjacency matrix A of graph G will be symmetric matrix, in which $A[i][j] = A[j][i]$ for every i and j .

Example:



	V ₁	V ₂	V ₃	V ₄
V ₁	0	1	1	1
V ₂	1	0	1	0
V ₃	1	1	0	1
V ₄	1	0	1	0

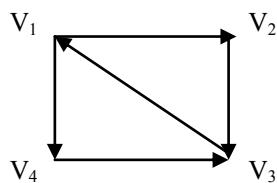
Figure 9.13: Adjacency Matrix representation of undirected graphs

Adjacency Matrix Representation for a Directed Graph (G')

The adjacency matrix of a directed graph G is defined as

$$A[i][j] = \begin{cases} 1, & \text{If } V_i \text{ is adjacent to } V_j \\ & \text{i.e. if there is an edge } (V_i, V_j) \\ 0, & \text{Otherwise} \end{cases}$$

Example:



	V ₁	V ₂	V ₃	V ₄
V ₁	0	1	0	1
V ₂	0	0	1	0
V ₃	1	0	0	0
V ₄	0	0	1	0

Figure 9.14: Adjacency Matrix representation of directed graphs G'

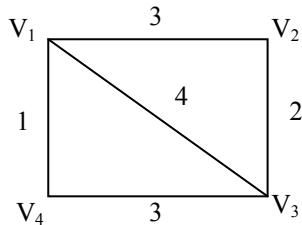
Table 9.2: Difference between Adjacency Matrix of Directed Graph (G') and Undirected Graph (G)

Feature	Directed Graph	Undirected Graph
Symmetric	May or may not be symmetric. If all the vertices are both ways connected, then only the adjacency matrix will be symmetric	Adjacency matrix is always symmetric
Degree of vertex	Row sum is out-degree of a vertex Column sum is in-degree of a vertex.	Both row sum and column sum are same for a vertex. Row sum is the degree of a vertex. No question of in-degree and out-degree.

Adjacency Matrix Representation for an Undirected Weighted Graph (G)

The adjacency matrix of a weighted graph G is defined as

$$A[i][j] = \begin{cases} w, & \text{If there is an edge } (V_i, V_j) \text{ of weight } w \\ 0, & \text{Otherwise} \end{cases}$$

Example:

	V ₁	V ₂	V ₃	V ₄
V ₁	0	3	4	1
V ₂	3	0	2	0
V ₃	4	2	0	3
V ₄	1	0	3	0

Figure 9.15: Adjacency Matrix representation of undirected weighted graph**Advantages of adjacency matrix**

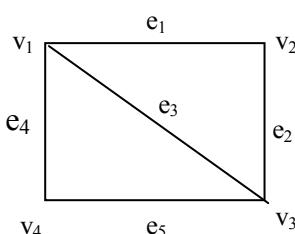
The adjacency matrix is very convenient to work with. Add or remove an edge can be done in O (1) time, the same time is required to check if there is an edge between two vertices. It is very easy to implement adjacency matrix of a graph.

Disadvantages of adjacency matrix

- i) Adjacency matrix consumes a huge volume of memory for storing big graph. For dense graph adjacency matrix is optimal, but for a sparse graph where not much edges are connected this matrix is not essential.
- ii) The complexity of scanning an adjacency matrix during the implementation of graph searching algorithm like DFS is very high O (V²), which can be reduced to O (|V|+|E|).
- iii) To analyses a graph using adjacency matrix is time-consuming, as we need to go through all the rows and columns.

Incident Matrix

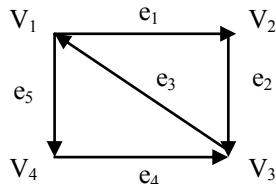
Let G (V, E) be a graph with n vertices where $n \geq 1$ and m edges where $m \geq 1$. The incident matrix of G is the two-dimensional ($n \times m$) array say A, with the property that $A[i][j] = 1$ if there is a relation between the edge e_j and vertex v_i in $E(G)$. $A[i][j] = 0$, if there is no relation between the edge e_j and vertex v_i . The relation between edge and vertex is known as “incident” relation.

Incident Matrix Representation for an Undirected Graph (G)**Example:**

	e ₁	e ₂	e ₃	e ₄	e ₅
V ₁	1	0	1	1	0
V ₂	1	1	0	0	0
V ₃	0	1	1	0	1
V ₄	0	0	0	1	1

Figure 9.16: Incident Matrix representation of undirected graphs**Incident Matrix Representation for a Directed Graph (G')**

For a directed graph, the relation between vertex and edge is represented in two ways. This is because in the case of directed graphs the edges are either outgoing edge that are coming out from a vertex or incoming edge that is the edge incident on the vertex. Depending on the relation, there are two types of incident matrix.

Example:

	e ₁	e ₂	e ₃	e ₄	e ₅
V ₁	1	0	0	0	1
V ₂	0	1	0	0	0
V ₃	0	0	1	0	0
V ₄	0	0	0	1	0

(a) Out-degree

	e ₁	e ₂	e ₃	e ₄	e ₅
V ₁	0	0	1	0	0
V ₂	1	0	0	0	0
V ₃	0	1	0	1	0
V ₄	0	0	0	0	1

(b) In degree

Figure 9.17: Incident Matrix representation of directed graphs (a) in degree & (b) out degree**Advantage**

It is easy to draw and analyze a undirected graph by the use of Incident graph.

Disadvantage

- i) Require more memory space for storing a graph.
- ii) For a directed graph in degree and out degree, the matrix should be computed during implementation.
- iii) Time-consuming.

Adjacency Lists

In this representation, the n rows of the adjacency matrix are represented as n linked list. There is one list for each vertex in G. The vertex in the list I represent the vertex that is adjacent to vertex i. Each vertex has at least two fields, one for vertex and another for the link with next vertex. The vertex field contains the index of the vertex adjacent to vertex i.

Structure definition of a vertex in Adjacency list is:

```

struct vertex
{
    int vertex;
    struct vertex *link;
};
  
```

Adjacency list representation for an Undirected Graph (G)

In the case of the undirected graph with n vertices and e edges, this representation requires n head vertices and $2e$ list vertices. The degree of any vertex in an undirected graph may be determined by just counting the number of a vertex in the adjacency list. The total number of edges in graph G may be determined in time $O(n+2e)$. If $n \ll e$ then time complexity is $O(2e) = O(e)$.

Example:

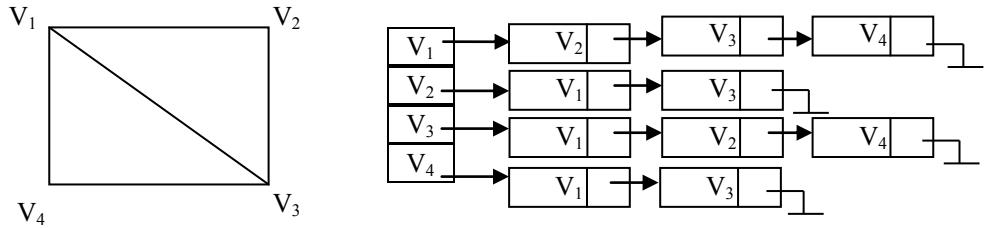


Figure 9.18: Adjacency list representation for an Undirected Graph (G)

Adjacency list representation for a Directed Graph (G')

In the case of directed graph number of head vertices n' and a number of list vertices e . The out-degree of a directed graph of any vertex may be determined by counting the number of vertices on its adjacency list. The total amount of time can, therefore, be determined $O(n + e)$. If $n \ll e$, then complexity is $O(e)$.

Example:

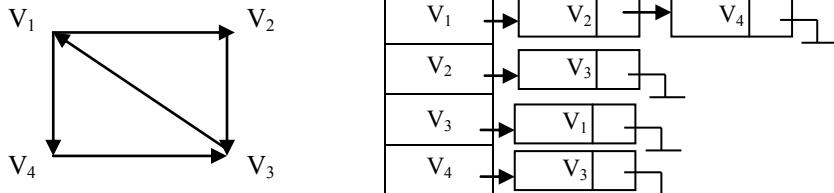


Figure 9.19: Adjacency list representation for a Directed Graph (G)

Inverse Adjacency list representation for a Directed Graph (G')

To determine the in-degree of a vertex of a directed graph another list is considered which is termed as an inverse adjacency list of a directed graph.

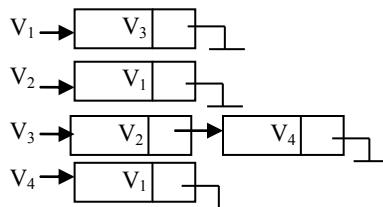


Figure 9.20: Inverse Adjacency list representation for a Directed Graph (G)

Advantages

The adjacency list allows us to store the graph in more compact form, than adjacency matrix, but the difference decreasing as a graph becomes denser. Next advantage is that adjacent list allows getting the list of adjacent vertices in $O(1)$ time, which is a big advantage for some algorithms.

Disadvantages

- i) Adding/removing an edge to/from the adjacent list is not as easy as for adjacency matrix. It

requires, on the average ($|E| / |V|$) time, which may result in cubical complexity for dense graphs to add all edges.

- ii) The adjacency list does not allow us to make an efficient implementation if a dynamic change of vertex number is required. Adding a new vertex can be done in $O(V)$, but removal results in $O(E)$ complexity.

Orthogonal List

Alternatively, one could adopt a simplified version of the list structure called orthogonal list used for sparse matrix representation. Each vertex would now have four fields and would represent one edge.

The vertex structure be would

```
struct vertex
{
    int head, tail;
    struct vertex *hp, *tp;
};
```

Example:

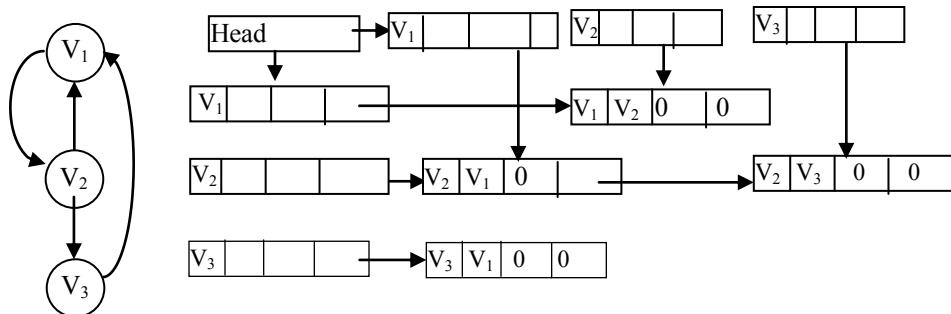


Figure 9.21: Orthogonal list representation for a Directed Graph (G)

Adjacency Multi-lists

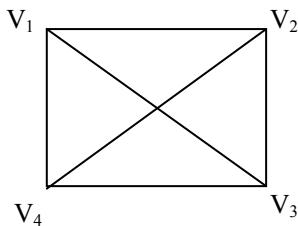
Adjacency multi-lists are an edge based, rather than vertex based in the graph representation. In the adjacency list representation of an undirected graph, each edge (v_i, v_j) is represented by two entries. One on the list for v_i and the other on the list v_j. As it is observed in some situation, it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multi-list.

For each edge, there will be exactly one vertex, but this vertex will be in two lists. The vertex structure is

```
struct vertex
{
    int m;
    int v1, v2;
    struct vertex *p1, *p2;
};
```

Mark(m)	1 st vertex(v ₁)	2 nd vertex(v ₂)	1 st path(p ₁)	2 nd path(p ₂)
---------	---	---	---------------------------------------	---------------------------------------

Example:



List of edges:

Edge (1, 2) → N1

Edge (1, 3) → N2

Edge (1, 4) → N3

Edge (2, 3) → N4

Edge (2, 4) → N5

Edge (3, 4) → N6

List of vertices\paths:

Vertex 1: N1 → N2 → N3

Vertex 2: N1 → N4 → N5

Vertex 3: N2 → N4 → N6

Vertex 4: N3 → N5 → N6

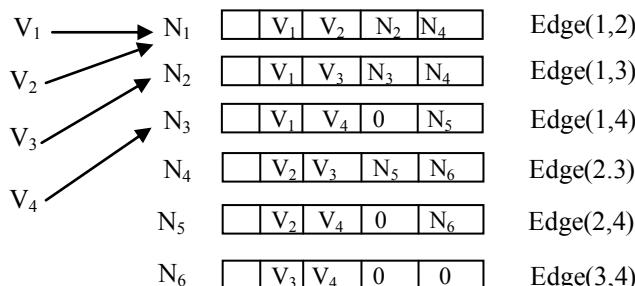


Figure 9.22: Adjacency Multi-list representation for a Directed Graph (G)

Algorithm for Adjacency Matrix Creation for A graph G

In the following algorithm, N represents a number of vertices and Adj[][][] is the adjacency matrix of the graph. Adj[i][j] is set to 1 if there is an edge between the vertices i and j else it is set to 0.

Algorithm to create an adjacency matrix of a graph

Algorithm: CREATE (G)

[G is a given graph of N vertices]

```

1. Set N = number of vertices
2. Repeat For I = 1 to N
3.   Repeat For J = 1 to N
4.     If I = J then [avoid self-loop]
        Adj[I][J] = 0
      Else
        Adj[i][j] = 1
      [End of If]
    [End of Loop]
  [End of Loop]
5. Return

```

Algorithm for Deletion of a vertex for A graph G

Following algorithm describes the procedure for deletion of a vertex from a graph. If the vertex V is to be deleted, the status of Adj[I][V] is set to 0 and the status of Adj[V][V] is set to 0.

Algorithm to delete a vertex from a graph

```

Algorithm: Delete (G)
[G is a given graph of N vertices]
1. Set N = number of vertices
2. Set V = vertex to be deleted
3. Repeat For I = 1 to N
4.   Adj[I][V]=0
    Adj[v][V]=0
  [End of Loop]
5. Return

```

Graph Traversal

Graph traversal technique is used to reach to a vertex u from a vertex v of a graph G. There are two ways of traversing the graph.

Depth First Search

In a depth first search the vertex (v) explored first when a new vertex (u) is found the exploration of vertex v is stopped and the new vertex is then explored. After an exploration of all child vertices, the parent vertex again explored.

Given an undirected graph G = (V, E) with n vertices and an array VISITED[n] initially set to zero, this algorithm visits all vertices reachable from s. G and VISITED are global.

Algorithm to traverse a graph using depth first search

```

Algorithm: DFS (G, s)
[G is a given graph of N vertices, s is the source node]
1. Set VISITED [s] = 1
2. Visit s
3. Repeat for each vertex v adjacent to s do

```

```

4. If VISITED [v] = 0 then
5.     Call DFS (v)
    [End of if]
[End of For loop]
6. Return

```

Algorithm to traverse a graph using depth first search in non-recursive/iterative process.

Algorithm: DFS (G, s)

[G is a given graph of N vertices, s is the source node]

```

1. Repeat for all vertices v of graph G do
2.     Set VISITED [v] = 0
    [End of loop]
3. Initialize Stack STK to be empty
4. Set VISITED [s] = 1
5. Call PUSH (STK, s)
6. While ISEMPTY(STK) = false do
7.     Call POP(STK, s)
8.     Visit s
9.     Repeat for all vertices v adjacent to s do
10.        If VISITED[v] = 0 then
11.            Call PUSH (STK, v)
12.            Set VISITED[v] = 1
        [End of For loop]
    [End of While loop]
13. Return

```

In case G is represented by its adjacency lists, then the vertices, w adjacent to v can be determined by following a chain of links. Since the algorithm, DFS would examine each vertex in the adjacency lists at most once and there are $2e$ list vertices, the time to complete the search is $O(e)$. If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is $O(n)$. Since at most n vertices are visited, the total time is $O(n^2)$.

Example:

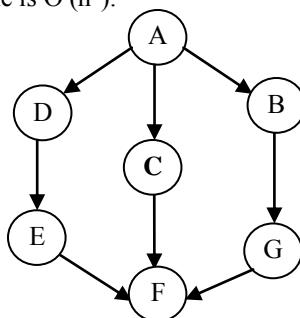


Figure 9.23: Depth First Search for a Directed Graph (G)

In the above example, the graph is traversed starting from vertex A to vertex F. In DFS recursive call is used, and recursion always uses stack data structure. So here, stack position is also given.

Table 9.2: Depth First Search

Operation	Stack	Display
PUSH A	A	-
POP A PUSH B, C, D	B,C,D	A
POP D PUSH E	B,C,E	A, D
POP E PUSH F	B,C,F	A, D, E
POP F	B,C	A, D, E, F
POP C	B	A, D, E, F, C
POP B PUSH G	G	A, D, E, F, C, B
POP G	-	A, D, E, F, C, B, G

During DFS, directed graph edges can be classified into the following:

Back edges: An edge (u, v) in which v is assumed as an ancestor of u in the tree (though it may not be in proper order). Thus, a self-loop is considered as back edge.

Forward edges: An edge (u, v) in which v is a proper descendant of u in the tree. These types of edges are known as forward edges.

Cross edges: An edge (u, v) in which u and v are not an ancestor or a descendant of one another. Such type of edges is known as cross edges.

Spanning Tree of Depth First Search

In depth-first search sequence in which vertices of a graph G are visited can form a spanning tree of that graph G . Following is the DFS spanning tree of the above-mentioned graph G .

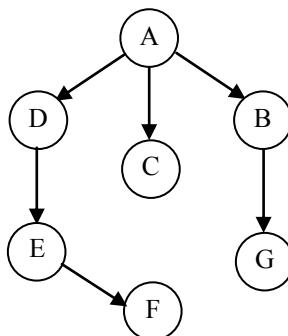


Figure 9.24: Spanning tree of Depth First Search for a Directed Graph (G)

Breadth First Search (BFS)

In breadth first search we visit a vertex (v) first and then explore all its adjacent vertices and make them visited. A vertex (v) is said to be explored when all its adjacent vertices are visited. The newly visited vertex is put into the list of unexplored vertices.

In BFS, a queue Q is used to implement unexplored vertices. A breadth first search of G is carried out beginning at vertex s. All vertices visited are marked as VISITED[i] = 1. The graph G and array VISITED are global and VISITED is initialized to zero.

Algorithm of Breadth First Search (BFS)

Algorithm: BFS (G, s)

[G is a given graph of N vertices, s is the source node]

1. Repeat for all vertices v of graph G do
2. Set VISITED [v] = 0
[End of loop]
3. Initialize queue Q to be empty
4. Set VISITED [s] = 1
5. Call ENQUE (Q, s)
6. While ISEMPTY(Q) = false do
7. Call DEQUE(Q, s)
8. visit s
9. Repeat for all vertices v adjacent to s do
10. If VISITED[v] = 0 then
11. Call ENQUE (Q, v)
12. Set VISITED[v] = 1
[End of For loop]
13. [End of While loop]
14. Return

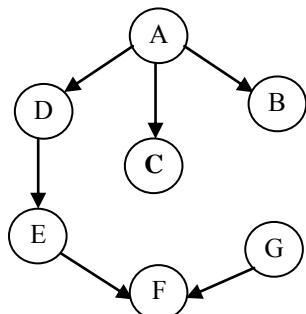
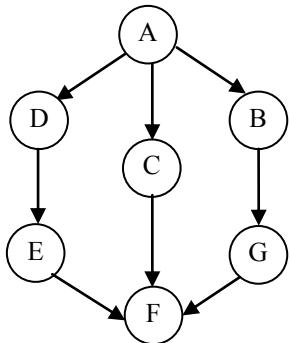
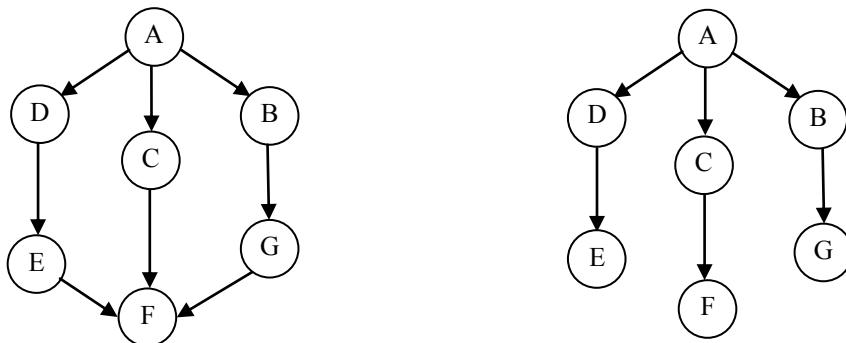


Figure 9.25: Traversal for Breadth First Search for a Directed Graph (G)

In BFS, the queue data structure is used so here in the above example, the position of the queue is given for BFS traversal.

Table 9.3: Breadth First Search

Operation	Queue(q)	Print
ENQUEUE A	A	-
DEQUEUE A	B, C, D	A
ENQUEUE B, C, D		
DEQUEUE B	C, D, G	A, B
ENQUEUE G		
DEQUEUE C	D, G, F	A, B, C
ENQUEUE F		
DEQUEUE D	G, F, E	A, B, C, D
ENQUEUE E		
DEQUEUE G	F, E	A, B, C, D, G
DEQUEUE F	E	A, B, C, D, G, F
DEQUEUE E	-	A, B, C, D, G, F, E

Spanning Tree of Breadth First Search**Figure 9.26:** Spanning tree for Breadth First Search for a Directed Graph (G)

Lemma 4: The length of the shortest path from S to V is denoted by $\delta(S, V)$. Then, on completion of BFS $\text{dist}[V] = \delta(S, V)$

Proof: The proof is based on induction taking the length of the shortest path.

Consider the shortest path from S

to V, u be the predecessor of V and BFS processes it first among all such vertices.

Thus, $\delta(S, V) = \delta(S, u) + 1$ (i)

When u is processed, then by induction we have $\text{dist}[u] = \delta(S, u)$ (ii)

Since V is a neighbour of u we set

$\text{dist}[V] = \text{dist}[u] + 1$ (iii)

thus from equation (ii)

$\text{dist}[V] = \delta(S, u) + 1$ (iv)

from equation (i) we have

$\text{dist}[V] = \delta(S, V)$

Table 9.4 Difference between Breadth First Search and Depth First Search

Feature	Breadth First Search	Depth First Search
Memory Space	More space requires	Less space requires
Backtrack	Backtracking is not required	If the wrong path is followed, then backtracking is required.
Speed	Slow searches	Fast searches
Time	We can reach goal early.	If we follow, a wrong path then more time is required to reach the goal. Else, we can reach the goal faster than BFS.
Complexity	$O(V+E)$	$O(V+E)$

Spanning Trees and Minimum Spanning Trees

A subgraph T of an undirected graph $G = (V, E)$ is a spanning tree if it is a tree that contains all the vertices of the graph G and has no cycle.

There will be more than one Spanning tree of a graph. This is shown in the following diagram.

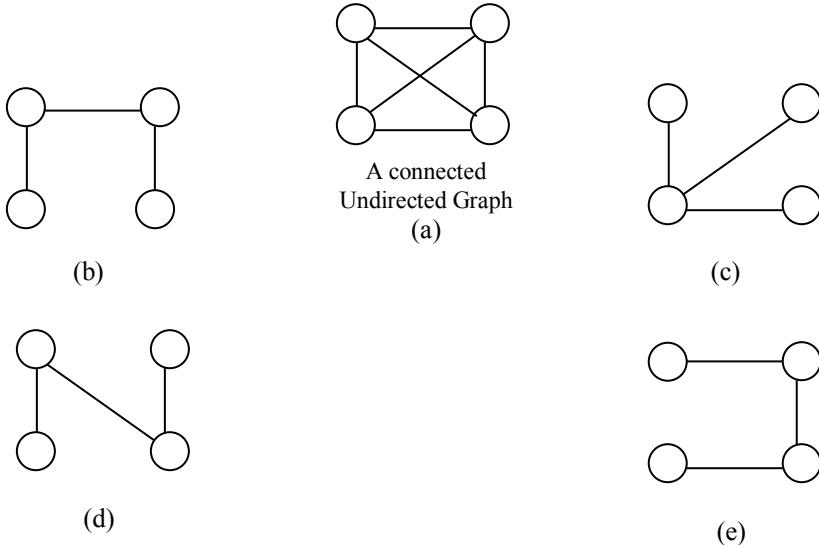


Figure 9.27: (a) An undirected graph. (b, c, d, e) different spanning tree of the graph

If the edges of a graph are weighted the graph is said to be “weighted graph”. For a weighted graph, cost of a spanning tree is measured by computing the sum of all the edges of that spanning tree.

Lemma 5: The number of spanning trees of a graph is the value of any cofactor of the Laplacian matrix of G .

Proof: Let $L = L(G)$ be the Laplacian matrix of G with eigen values $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$.

The (i, j) -cofactor of a matrix M is by definition

$$l_{xy} = (-1)^{i+j} \det M(i, j),$$

Where $M(i, j)$ is the matrix obtained from M by deleting row i and column j . Let l_{xy} be the (x, y) -cofactor of L . Note that l_{xy} does not depend on an ordering of the vertices of G . We set $N = t(G)$ and show that

$$N = l_{xy} = \det(L + 1/n^2 \cdot J) = 1/n \cdot \lambda_2 \dots \lambda_n \text{ for any } x, y \in V(G).$$

Let L_S , for $S \subset V(G)$, denote the matrix obtained from L by deleting the rows and columns indexed by S , so that $l_{xx} = \det L^{\{x\}}$. The equality $N = l_{xx}$ follows by induction on n , and for fixed $n > 1$ on the number of edges incident with x . Indeed, if $n = 1$ then $l_{xx} = 1$. Otherwise, if x has degree 0, then $l_{xx} = 0$ since $L^{\{x\}}$ has zero row sums. Now, if xy is an edge, then deleting this edge from G decreases l_{xx} by

$\det L^{\{x,y\}}$, which by induction is the number of spanning trees of G with edge xy collapsing to a point, which is the number of spanning trees containing the edge xy . This shows $N = l_{xx}$. Since the sum of the columns of L is zero, so that one column is minus the sum of the other columns, we have $l_{xx} = l_{xy}$ for any x, y .

Now, we consider the Laplacian polynomial

$$\mu(G, t) = \det(tI - L) = t^n \prod_{i=2}^n (t - \lambda_i) \text{ for graph } G \text{ then}$$

$(-1)^{n-1} \lambda_2 \dots \lambda_n$ is the coefficient of t , that is

$$d/dt(\det(tI - L)) = \sum_x \det(tI - L^{\{x\}})$$

Putting, $t = 0$ it is obtained that $\lambda_2 \dots \lambda_n = \sum_x l_{xx} = nN$

Finally, the eigenvalues of $L + 1/n^2 \cdot J$ are $1/n$ and $\lambda_2 \dots \lambda_n$, so

$$\det(L + 1/n^2 \cdot J) = 1/n (\lambda_2 \dots \lambda_n)$$

Case Study:

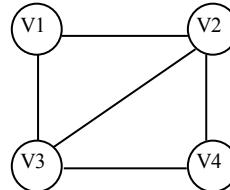


Figure 9.28: Undirected Graph G

$$\text{Degree Matrix } D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

$$\text{Adjacency Matrix } A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad \text{Laplacian Matrix } L = D - A = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$$

Compute (2, 2) cofactor of matrix $D-A$

(i, j) co-factor = $(-1)^{i+j} [\det \text{ of } (n-1) \times (n-1) \text{ matrix obtained by removing } i^{\text{th}} \text{ row and } j^{\text{th}} \text{ Column}]$

$$l_{2,2} = (-1)^{2+2} \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix} = 2 \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} - (-1) \begin{bmatrix} -1 & -1 \\ 0 & 2 \end{bmatrix} = 2(6-1) + (-2) = 10 - 2 = 8$$

A number of possible spanning tree for the above-mentioned graph G is 8.

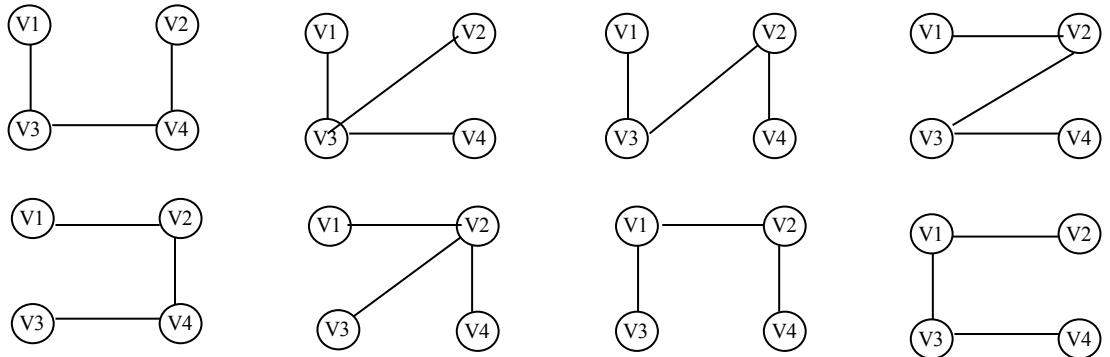


Figure 9.29: Possible Spanning Tree of Undirected Graph G

Minimal Spanning Tree

A Minimal spanning tree of a weighted graph is the spanning tree that has minimum cost. That implies the sum of the cost of all the edges is guaranteed to be a minimum of all possible spanning trees in the graph.

There are several different ways to construct a minimum spanning tree.

Kruskal's Algorithm

Let $G = (V, E)$ be a graph with $|V| = n$. T is the set of minimum cost edges which is initially empty or null. N is the number of vertices of a graph G. Assume v and w be any two adjacent vertexes of G.

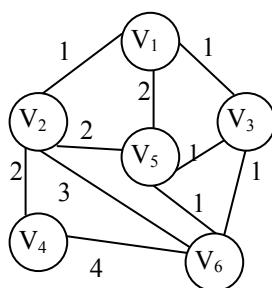
Kruskal's Algorithm for finding minimum spanning tree

Algorithm: MST (G)

[G is a given graph of N number of edges and V number of vertices]

1. $T = \text{NULL}$
2. Repeat step 3 to 5 while T contains less than $(N-1)$ edges and E not empty
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E
5. If the edge (v, w) does not create a cycle in T then
 Add (v, w) to T
 Else
 Discard (v, w)
 [End of loop]
6. If T contains fewer than $(N-1)$ edges, then there is no spanning tree.
7. Return

Example :



$$E = \{(V_1, V_2), (V_1, V_3), (V_1, V_5), (V_2, V_4), (V_2, V_5), (V_2, V_6), (V_3, V_5), (V_3, V_6)\}$$

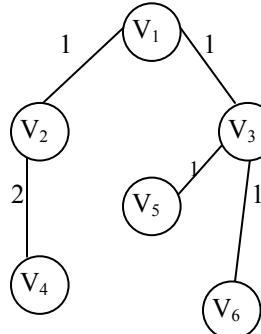


Figure 9.30: Minimal Spanning Tree of Undirected Weighted Graph G by Kruskal's Algorithm

$$T = \{(V_1, V_3), (V_1, V_2), (V_2, V_4), (V_3, V_5), (V_3, V_6)\}$$

Minimum cost=6

Complexity of Kruskal's Algorithm is $O(|E|\log|E|)$

Prim's Algorithm

Suppose $V = \{v_1, v_2, v_3, v_4, \dots, v_n\}$ of a weighted undirected graph $G = (V, E)$. The Prim's algorithm begins with a set V' initialize to you, i.e. $V' = \{u\}$. It then grows a spanning tree one edge at a time. At each step it finds the shortest edge (u, v) that connects W and $(V - V')$ and then adds v , the vertex in $(V - V')$ to V' . It repeats this step until $V' = V$.

E is the set of edges, which are to be extracted to obtain the minimum cost spanning tree.

Prim's Algorithm for finding minimum spanning tree

Algorithm: MST (G)

[G is a given graph of N number of edges and V number of vertices]

1. Initialization E' to a null set i.e. $E' = \text{NULL}$
2. Select minimum cost edge (u, v) from E
3. $V' = \{u\}$
4. Repeat step 5 to 7 until $V' = V$
5. Select lowest cost edge (u, v) such that u is in V' and v is in $(V - V')$
6. Add edge to set E' i.e. $E' = E' \cup \{(u, v)\}$

```

7. Add v to V' i.e. V' = V' ∪ {v}
[End of loop]
8: Return

```

Example:

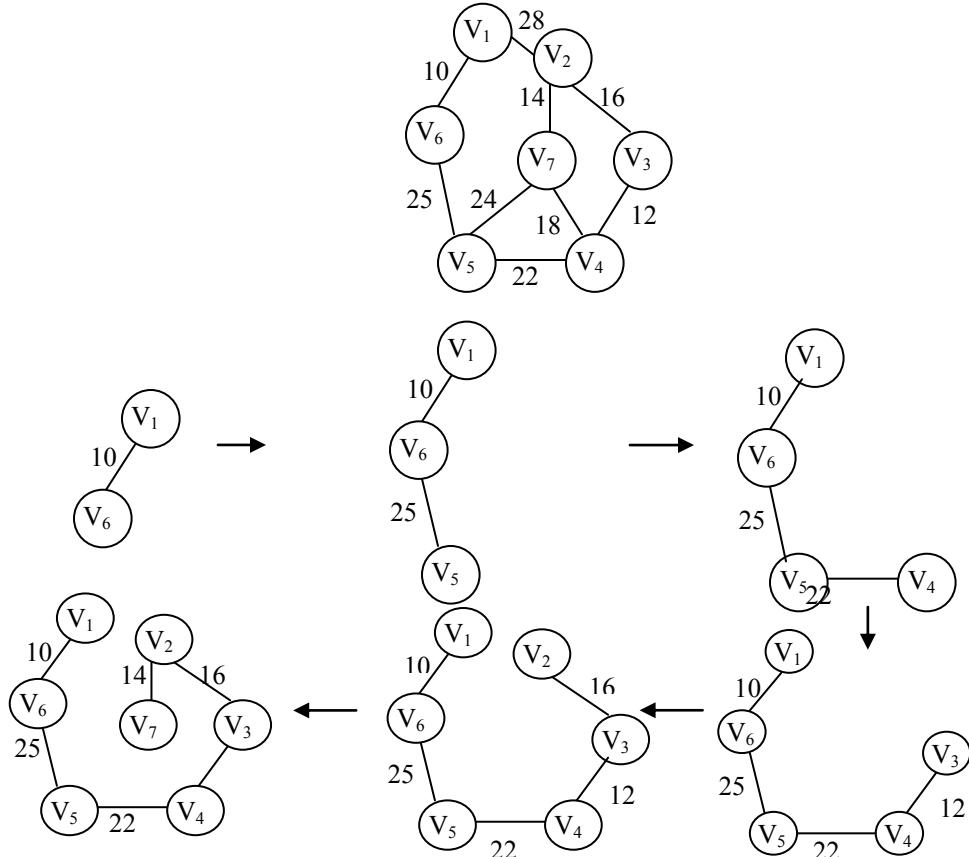


Figure 9.31: Minimal Spanning Tree of Undirected Weighted Graph G by Prim's Algorithm

Complexity of Prim's Algorithm is $O(n + E \log n)$, where n is a number of vertices and E implies a number of edges.

Shortest Paths

Shortest path problem states a way to find the path in a weighted graph connecting two given vertices u and v with the property that the sum of the weights of all the edges is minimized over all such paths.

There are a few variants of the shortest path problem:

1. Single-source shortest-path problem
2. Single-destination shortest paths problem
3. Single-pair shortest-path problem
4. All-pairs shortest paths problem

Single-source shortest-path

The starting vertex of the path is considered to be the source(S) vertex and the last vertex is the destination vertex. There are two ways for solving single source shortest path problem.

Dijkstra's Algorithm

This algorithm finds the shortest path to a vertex to the rest of the vertices in a graph. First, it explores the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. The algorithm both directed and undirected graphs. The one constraint for this algorithm is that all the edges must be non-negative edges.

$G = (V, E)$ is a weighted connected graph,

W = weight matrix

S = set of visited vertices from source vertex to destination vertex, initially it is empty.

s = source vertex, $\text{dist}[v]$ = distance of vertex v from s

Dijkstra's Algorithm for single source shortest path

Algorithm: Shortest_path(G, s)

[G is a given graph of N number of edges and V number of vertices, s is the source vertex]

1. Initialize $S = \{s\}$
2. Initialize $\text{dist}[s] = 0$
3. Repeat for all $v \in V - \{s\}$
 - $\text{dist}[v] = \alpha$
 - [End of loop]
4. Repeat while $S \neq V$
5. Find a vertex $w \in V - S$ such that $\text{dist}[w]$ is a minimum distance
6. $S = S \cup \{w\}$
7. Repeat for all $v \in V - S$
 - $\text{dist}[v] = \min(\text{dist}[v], \text{dist}[w] + W(w, v))$
 - [End of loop]
 - [End of loop]
9. Return

Example:

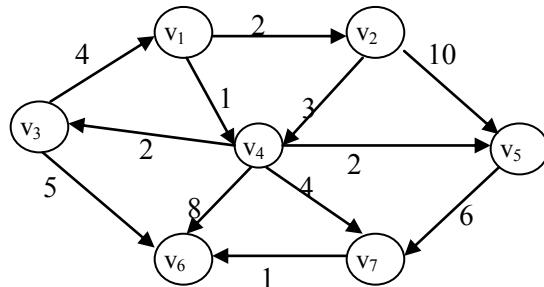


Figure 9.32: Shortest Path Using Dijkstra Algorithm

Table 9.5: Shortest Path By Dijkstra

Iteration	S	W	dist[w]	dist[v2]	dist[v3]	dist[v4]	dist[v5]	dist[v6]	dist[v7]
	{ v1 }	-	-	∞	∞	∞	∞	∞	∞
	{ v1 }	-	-	2	∞	1	∞	∞	∞
1	{ v1, v4 }	v4	1	2	3	1	3	9	5
2	{ v1, v4, v2 }	v2	2	2	3	1	3	9	5
3	{ v1, v4, v2, v3 }	v3	3	2	3	1	3	8	5
4	{ v1, v4, v2, v3, v5 }	v5	3	2	3	1	3	8	5
5	{ v1, v4, v2, v3, v5, v7 }	v7	5	2	3	1	3	6	5
6	{ v1, v4, v2, v3, v5, v7, v6 }	v6	6	2	3	1	3	6	5

Bellman Ford Algorithm

In this algorithm authors Richard Bellman, Samuel End and Lester Ford. Jr. have considered negative weighted edge that was not focused in Dijkstra's Algorithm. The algorithm finds the shortest path repeating a vertex of a graph. $G=(V, E)$ is a directed graph with source S and weight function $W:E \rightarrow R$. This algorithm returns a Boolean value TRUE if and only if the graph contains a negative weight cycle that is reachable from the source.

Bellman Ford algorithm for single source shortest path

Algorithm: Shortest_path()

[G is a given graph of N number of edges and V number of vertices, s is the source vertex]

1. Repeat step 2 for all vertices
2. $dist[i] = cost[v, i]$
[End of loop]
3. Repeat step 4-6 for $k = 2$ to $n-1$ do
4. For each u such that $u \neq v$ and
 u has atleast one incoming edge do
5. For each (i, u) in graph do
6. if $dist[u] > dist[i] + cost[i, u]$ then
 $dist[u] = dist[i] + cost[i, u]$
7. Return

Example:

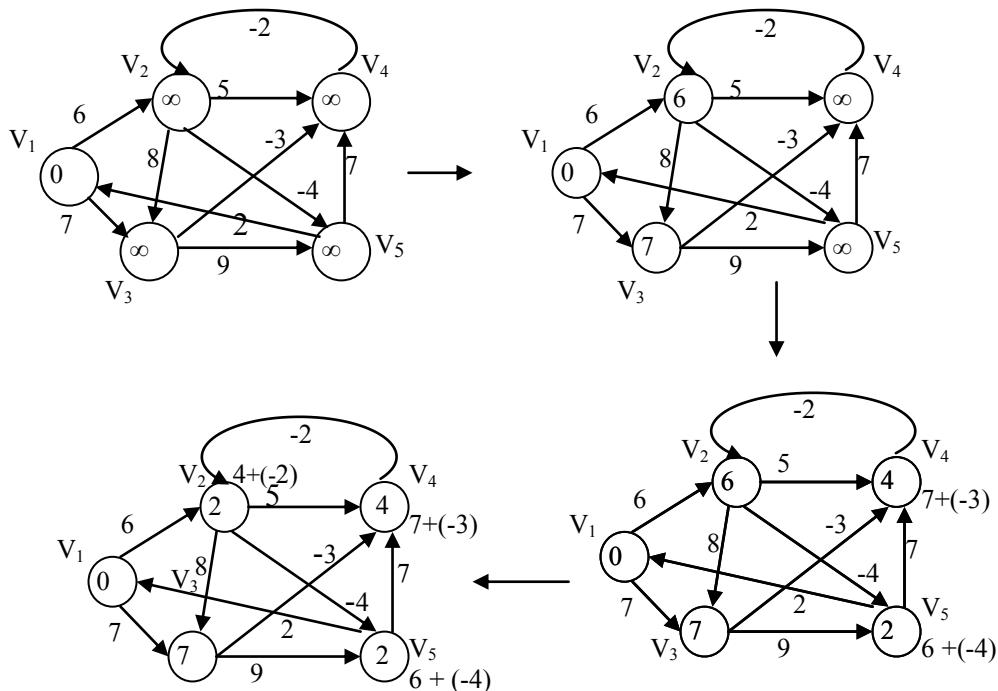


Figure 9.33: Shortest Path Using Bellman Ford Algorithm

The shortest path from source vertex v1 is $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_2$

Another shortest path from source vertex v1 is $v_1 \rightarrow v_2 \rightarrow v_5$

The complexity of this algorithm is $O(VE)$. Since initialization takes $O(V)$ and traversing each edge associated with the vertex $O(E)$. So, total complexity $O(VE)$

All Pairs Shortest Paths

Consider a weighted digraph $G = (V, E)$. Solving All Pair Shortest Path problem consists of computing the shortest distance or path in terms of cost/weight between every pair of vertices(u, v) where $u, v \in V$ in the graph G . It is assumed that the graph does not contain any negative or zero weight cycle.

Floyd-Warshall Algorithm

Robert Floyd and Stephen Warshall had invented this algorithm. In a single execution of the algorithm, the shortest path between all pairs of vertices is discovered. This algorithm is based on dynamic programming.

$\text{Cost}[1:n, 1:n]$ is the cost adjacency matrix of a graph with n vertices. $A[i, j]$ is the cost of a shortest path from vertex i to vertex j . $\text{cost}[i, j] = 0$, for $1 \leq i \leq n$.

Floyd-Warshall Algorithm for all pairs shortest path

Algorithm: `Shortest_path(A, cost)`

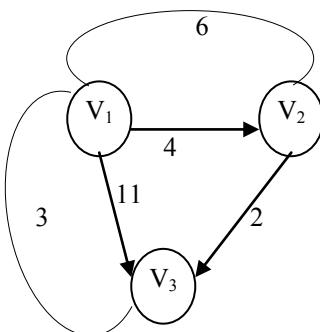
[$A[i, j]$ is the cost of a shortest path from vertex i to vertex j]

```

1. Repeat For I = 1 to N do
2.   Repeat For J = 1 to N do
3.     A[I][J] = cost[I][J]
    [End of loop]
  [End of loop]
4. Repeat For k = 1 to N do
5.   Repeat For I = 1 to N do
6.     Repeat For J = 1 to N do
7.       A[I][J] = min(A[I][J], A[I][K] + A[K][J])
      [End of loop]
    [End of loop]
  [End of loop]
8. Return

```

Example:



	V ₁	V ₂	V ₃
V ₁	0	4	11
V ₂	6	0	2
V ₃	3	A	0

	V ₁	V ₂	V ₃
V ₁	0	4	11
V ₂	6	0	2
V ₃	3	7	0

	V ₁	V ₂	V ₃
V ₁	0	4	6
V ₂	6	0	2
V ₃	3	7	0

	V ₁	V ₂	V ₃
V ₁	0	4	6
V ₂	5	0	2
V ₃	3	7	0

Figure 9.34: All Pair Shortest Path Using Bellman Ford Algorithm

Time complexity: The time complexity of this algorithm is easy to determine because the loop is independent of the data in the matrix. The cost computing statement will execute n^3 time, so time complexity is $O(n^3)$.

Travelling Sale's Man Problem

Before describing Travelling Salesman Problem, we need to know Hamilton circuit.

Hamilton Circuit: A Hamilton circuit is a circuit that uses every vertex of a graph once.

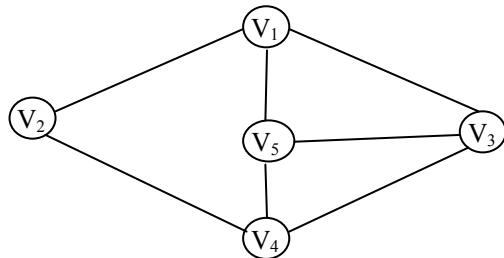


Figure 9.35: Hamilton Circuit

In this example the Hamilton circuit is

1. $V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5$
2. $V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5 \rightarrow V_3 \rightarrow V_1$
3. $V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_3 \rightarrow V_5 \rightarrow V_1$

Minimum cost Hamilton circuit: In a weighted graph, a minimum Hamilton circuit is a Hamilton circuit with the smallest possible weight.

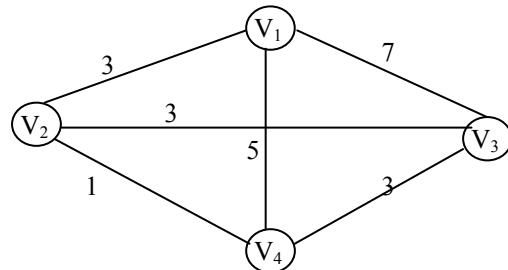


Figure 9.36: Minimal Cost Hamilton Circuit

Solution By Brute Force Algorithm

This algorithm describes the process of finding minimal cost shortest path from vertex i to vertex j.

Algorithm: Bute_force(A, cost)

[$A[i, j]$ is the cost of a shortest path from vertex i to vertex j]

1. Choose A starting Point, here it is V_1 .
2. List all possible Hamilton circuits with total cost.
3. Find weight of each circuit.
4. Find minimum Hamilton circuit with the lowest cost.
5. Return

Table 9.6: Minimal Cost Hamilton Circuit

Circuit	Weight of circuit
V ₁ → V ₂ → V ₄ → V ₃ → V ₁	14
V ₁ → V ₂ → V ₃ → V ₄ → V ₁	14
V ₁ → V ₄ → V ₂ → V ₃ → V ₁	16
V ₁ → V ₄ → V ₃ → V ₂ → V ₁	14
V ₁ → V ₃ → V ₄ → V ₂ → V ₁	14
V ₁ → V ₃ → V ₂ → V ₄ → V ₁	16

Travelling Sale's Man Problem

In this problem, the salesman needs to visit n cities in such a way that all the cities must be visited only once and at the end, he returns to the city from where he started with minimum cost.

The following assumptions are taken for traveling salesman problem:

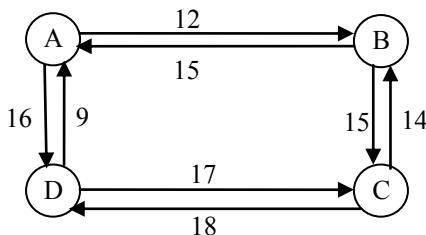
- Complete Directed Graph
- A non-empty finite set of vertices on a graph V where $V = \{1, 2, 3, \dots, n\}$
- The set of edges is in E
- The distance from i to j $C_{ij} \neq C_{ji}$
- The series of line (S) $\subseteq \{1, 2, 3, 4, \dots, n\}$
- The weight of shortest paths that start at vertex i that through all vertices in S and ends at vertex 1.

Algorithm: TSP (A, cost)

[A is the adjacency matrix of the graph G , C is the distance matrix that consist of shortest distance between vertices, S is the subset of cities]

1. Initialise distance matrix or cost matrix $C[i, j] = 0$
2. Repeat until all vertices have discovered in V
 - a. Repeat for all subset $S \subseteq \{1, 2, 3, \dots, n\}$ and containing 1
 - b. Set $C(S, 1) = \infty$
 - c. Repeat for all $j \in S$ and $j \neq 1$

$$C(S, j) = \min\{C(S - \{j\}, i) + d(i, j) \text{ for } i \in S \text{ and } i \neq j\}$$
[End of loop]
[End of loop]
[End of loop]
3. Return $\min_j C(\{1, 2, 3, \dots, n\}, j) + d(j, 1)$

**Figure 9.37:** Travelling Salesman Problem

As a number of vertices in this graph is 4. So, number of Hamilton circuit is $(n-1)! = 3! = 6$

We need to find all possible circuits to find minimum one.

Summary

- A Graph consists of finite non-empty set of vertices and finite non-empty set of edges that connects these vertices.
- A graph can be represented in memory using matrices and linked list.
- A graph can be traversed either using Breadth First Traversal or Depth First Traversal.
- A spanning tree is a subgraph of an undirected graph $G = (V, E)$ if it is a tree that contains all the vertices of the graph G.
- Minimum cost spanning tree of a weighted graph can be formed either by using Prim's or Kruskal's algorithm.

Exercises

1. Choose the correct alternatives for the following:

- Graph is a

a) linear data structure	b) non-linear data structure
c) Either (a) or (b) depending on situation	d) None of these.
- A complete directed graph of 5 vertices has _____ number of edges

a) 5	b) 10	c) 20	d) 25.
------	-------	-------	--------
- Breadth-first-search algorithm uses data structure

a) stack	b) queue	c) binary tree	d) dequeue
----------	----------	----------------	------------
- The number of edges in a complete graph with 'n' vertices is

a) $n(n - 1)$	b) $n(n - 1)/2$	c) n^2	d) $2^n - 1$.
---------------	-----------------	----------	----------------
- An adjacency matrix representation of a graph cannot contain information of

a) vertices	b) edges	c) direction of edge	d) parallel edge
-------------	----------	----------------------	------------------
- A vertex of degree one is called

a) Isolated vertex	b) Pendant vertex	c) Colored vertex	d) Null vertex.
--------------------	-------------------	-------------------	-----------------
- Any connected graph with x vertices must have at least

a) $x + 1$ edges	b) $x - 1$ edges	c) x edges	d) $x/2$ edges.
------------------	------------------	--------------	-----------------
- The adjacency matrix of an undirected graph is

a) unit matrix	b) asymmetric matrix	c) symmetric matrix	d) none of these
----------------	----------------------	---------------------	------------------
- BFS constructs

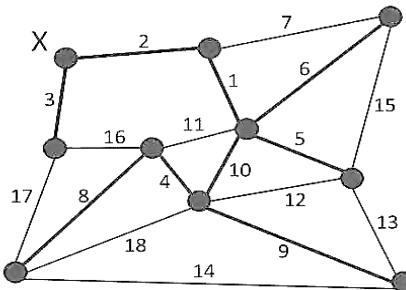
a) a minimal cost spanning tree of a graph	b) a depth-first spanning tree of a graph
c) a breadth first spanning tree of a graph	d) None of these.
- A digraph in which out degree and in degree are same is called

a) balanced	b) symmetric	c) regular	d) none of the above
-------------	--------------	------------	----------------------
- Which is not a representation of a graph

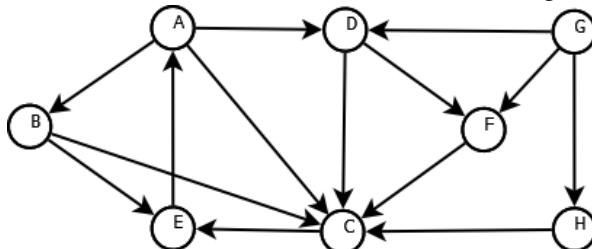
a) Adjacency matrix	b) Adjacency list	c) Edge list	d) All of the above
---------------------	-------------------	--------------	---------------------
- In adjacency matrix, parallel edges are represented by

a) Similar column	b) Similar row	c) Not representable	d) None of the above
-------------------	----------------	----------------------	----------------------
- Spanning tree consist of

- a) All vertices b) All edges c) All vertices and all edges d) None of the above
- xiv) Self-loop in adjacency matrix is represented by
 a) 1st row all 1 b) 1st column all 1 c) Diagonal all 1 d) None of the above
- xv) A graph G with n vertices bipartite if it contains:
 a) n edges b) a cycle of odd length c) no cycle of odd length d) n^2 edges
- xvi) vertex of in-degree zero in a directed graph is called
 a) Articulation point b) sink c) isolated vertex d) root vertex
- xvii) The vertex, removal of which makes a graph disconnected, is called
 a) Pendant vertex b) bridge c) articulation point
- xviii) Bellman-Ford algorithm is used for
 a) All pair shortest path b) Single source shortest path
 b) Single Destination Shortest Path d) None of the above
2. What is a minimum spanning tree? Describe Huffman's algorithm.
3. What is a complete graph? Show that the sum of the degree of all the vertices in a graph is always even.
4. State two different data structure for representing graphs.
5. Write down breadth first search algorithm for searching a graph.
6. Write down the DFS algorithm for searching a graph.
7. Write down a program to find out the shortest path using Dijkstra algorithm.
8. Compare BFS and DFS. Discuss the two different ways of representing a graph.
9. Draw the minimum cost spanning tree for the graph given below and find its cost.



10. Show the result of running BFS and DFS on the directed graph given below using vertex 3 as a source. Show the status of the data structure used at each stage :



11. A rat has entered a checkerboard maze through one corner, whose the white boxes are open

9.34 | Data Structures and Algorithms with C

and black boxes represent obstacles. Develop an algorithm by which the rat can exit the maze through the opposite corner. Clearly, explain their presentation of the maze and any specific data structure you have used for the algorithm.

CHAPTER 10

SEARCHING AND SORTING

"You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program." -Alan J. Perlis

Searching for items and sorting through items are tasks that we do everyday. We search for all occurrences of a word in a file in order to replace it with another word. We sort the items on a list into alphabetical or numerical order. Searching and sorting are also most common operations in computer programs.

SEARCHING

Searching refers to the operation of finding the location of the desired key element within some data structures. Data structures can include linked lists, arrays, search trees or hash tables. The search is successful if the item is found, and then returns its location; otherwise, the search is unsuccessful. The appropriate search algorithm often depends on the data structure being searched.

There are many different searching techniques such as

- Linear Search
- Binary Search
- Interpolation Search

KEY FEATURES

-
- book Linear Search
 - book Binary Search
 - book Interpolation Search
 - book Bubble Sort
 - book Selection Sort
 - book Insertion Sort
 - book Quick Sort
 - book Merge Sort
 - book Shell Sort
 - book Heap Sort
-

Linear Search

Linear search, or sequential search is a method for finding the position of a particular key value in a list, where the search starts from beginning of the list and checking every element, one at a time in sequence until the desired key is found or the end of the list is reached.

Algorithm of Linear Search

Algorithm: **LINEAR_SEARCH (A, N, KEY, LOC)**

[*A* is an array of *N* elements, *Key* is the searching key]

1. Set *I* = 0
2. Repeat while *I* < *N* and *A*[*I*] ≠ *KEY*
 - Set *I* = *I* + 1
 - [End of loop]
3. If *I* < *N* then Set *LOC* = *I*
ELSE *LOC* = -1
4. Return

Linear search is the simplest searching technique. Suppose A is an array with N elements from A[0] to A[N - 1]. The linear search algorithm for finding the KEY item performs as below.

At first, the algorithm compares KEY with the first element A[0] of the list. If KEY = A[0] then the search is successful and return the location. Otherwise, compares KEY with the second element A[1], third element A[2], so on until the KEY is found or the end of the list is reached. Finally, after successfully searching return the location of the KEY by reference, otherwise in the case of unsuccessful search return -1.

Now, the total $2n$ number of comparisons are required in the worst case for an unsuccessful search, but for the best case, it is only two. A **sentinel** element may be used to reduce the number of comparisons. The word sentinel comes from sentry (means guard). An extra memory location is used for sentinel element. At the beginning, put the searching key as a sentinel at the end of the list ($A[N] = \text{KEY}$), outside the valid array area. Now there is a guarantee of success, but it needs to be examined when the success occurs. When the searching key found at N^{th} position, then actually it is an unsuccessful search. Otherwise, the key is found at I^{th} position.

Algorithm of Modified Linear Search

Algorithm: **LINEAR_SEARCH (A, N, KEY, LOC)**

[*A is an array of N elements, Key is the searching key, LOC is the location of the KEY*]

1. Set $A[N] = \text{KEY}$
2. Set $I = 0$
3. Repeat while $A[I] \neq \text{KEY}$
 - Set $I = I + 1$
 - [End of loop]
4. If $I < N$ then Set $\text{LOC} = I$
 - ELSE Set $\text{LOC} = -1$
5. Return

Therefore, in the modified algorithm, the number of comparisons in worst case has become $n+1$ instead of $2n$. The element can be found in the worst case for an unsuccessful search at the N^{th} location, so array bound checking is not necessary.

Time complexity of Linear Search

The linear search runs at a worst time as it makes at most n comparisons, where n is the length of the list. In the average case, the number of comparisons calculated by the average of all possible instances:

$$f(n) = [1 + 2 + 3 + \dots + (n-1)]/n = 1/n \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2n} = \frac{n-1}{2} = O(n)$$

The best case occurs when the key is found at first position, so only one comparison is required and the worst case occurs when the key is found at the last position, hence $n+1$ comparisons are required.

Linear Searching algorithm	Best Case	Average Case	Worst Case
	$O(1)$	$O(n)$	$O(n)$

Binary Search

Binary search is an efficient searching technique that operates on ordered list. Binary search is also known as logarithmic search or bisection. Binary search or half-interval search algorithm is a method for finding the position of a particular key value within a sorted list.

Suppose A is an array, which is sorted in ascending order. The variables I and J denote the first and last location of the array respectively. Now binary searching is used for finding the KEY item within the sorted list:

$A[I], A[I + 1], A[I + 2] \dots A[J - 1], A[J]$

The Binary Searching algorithm is performed as below:

At first, the algorithm compares KEY with the middle element $A[MID]$ of the list, where MID is obtained by $MID = (I + J) / 2$. If $KEY = A[MID]$ then the search is successful and return the location. Otherwise, sub-array is obtained as follows:

- i) If $KEY < A[MID]$ then KEY item can appear only in the lower half of the array:

$A[I], A[I + 1], A[I + 2] \dots A[MID - 2], A[MID - 1]$

Therefore, we reset the value of J by using $J = MID - 1$ and start searching for lower half of the array.

- ii) If $KEY > A[MID]$ then KEY item can appear only in the upper half of the array:

$A[MID + 1], A[MID + 2], A[MID + 3] \dots A[J - 1], A[J]$

Therefore, we reset the value of I by using $I = MID + 1$ and start searching for upper half of the array.

Finally, after successfully searching return the location of the KEY by reference, otherwise in the case of unsuccessful search return -1.

Example:

We want to search for 31

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
10	14	19	26	27	31	33	35	42

(a)

First, we shall determine the middle position of the array using this formula:

$$MID = (I + J) / 2 = (0 + 8) / 2 = 4$$

It is observed that the value to be searched is more than the $A[MID] = 27$, where $MID = 4$.

As the value is more than 27 so, searching area is in the upper half of the area.

[5]	[6]	[7]	[8]
31	33	35	42

(b)

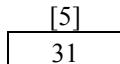
Now, modify the start index of the sub-array:

$$I = MID + 1 = 4 + 1 = 5$$

$$MID = (I + J) / 2 = (5 + 8) / 2 = 6$$

Target value is 31 which is less than the $A[MID] = 33$, so search zone will be left side of MID where,

$$J = MID - 1 = 6 - 1 = 5$$



(c)

Figure 10.1 (a-c): Binary Search

Now, $MID = (I + J) / 2 = (5 + 5) / 2 = 5$

Where the item is found at the $A[MID]$. Here, the searching procedure ends successfully.

Algorithm of Binary Search

Algorithm: BINARY_SEARCH (A, N, KEY, LOC)

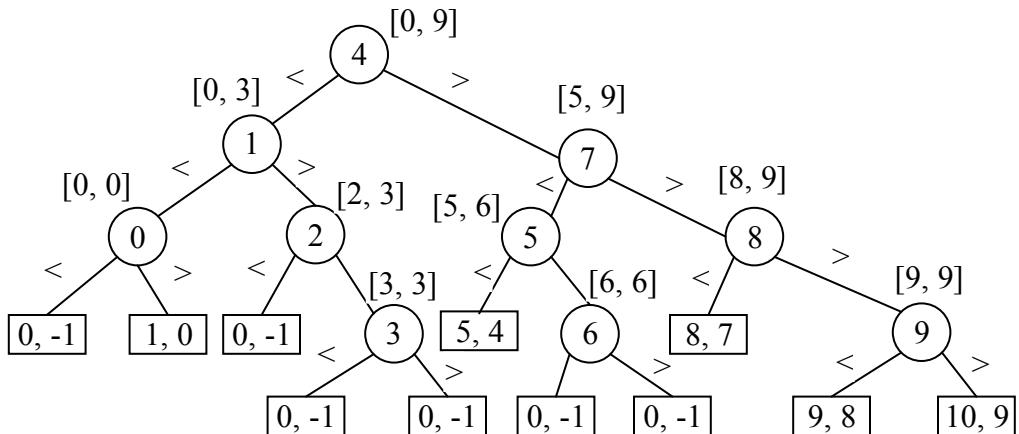
[*A* is an array of *N* elements, *Key* is the searching key, *LOC* is the location of the *KEY*]

1. Set $I = 0$, $J = N-1$
2. Repeat steps 3 and 4 while $I \leq J$
3. $MID = (I+J)/2$
4. If $KEY = A[MID]$ then Goto step 5
 Else If $KEY < A[MID]$ Set $J = MID-1$
 Else Set $I = MID+1$
 [End of If]
 [End of Loop]
5. If $A[MID] = KEY$ then Set $LOC = MID$
 Else Set $LOC = -1$
 [End of If]
6. Return

Time complexity of Binary Search

In binary search, after each iteration, we try to discard half elements of the array. Suppose, the array size is 10 and $A[0] \leq A[1] \leq A[2] \leq A[3] \leq A[4] \leq A[5] \leq A[6] \leq A[7] \leq A[8] \leq A[9]$

Initially $I = 0$ and $J = 9$, therefore $MID = (I+J)/2 = 4$

**Figure 10.2: Comparison Tree for Binary Search**

Key element equals to	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
No. of Comparisons	3	2	3	4	1	3	4	2	3	4

Average case = $29/10 \approx 3$ (approx) and worst case = 4

Worst-case time complexity: In worst case, total number of comparisons can be calculated recursively

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ T(n/2) + c, & \text{Otherwise} \end{cases}$$

Where c is a constant

Now,

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= [T(n/4) + c] + c \\ &= T(n/4) + 2c \\ &\vdots \\ &= T(n/2^r) + rc \\ &= T(n/n) + c \log_2 n && [\text{when } 2^r = n \text{ then } r = \log_2 n] \\ &= T(1) + c \log_2 n && [\text{hence } T(1)=c] \\ &= c + c \log_2 n \\ &= c[1 + \log_2 n] \\ &= O(\log n) \end{aligned}$$

The number of comparisons in $\lceil \log_2 n + 1 \rceil$ worst case is the depth or height of the binary tree. The number of comparisons in average case is approximately equals to the number of comparisons in worst case. The best case occurs when the key found at first comparison.

Binary Searching algorithm	Best Case	Average Case	Worst Case
	O(1)	O(log n)	O(log n)

Table 10.1: Differences between Linear Searching and Binary Searching

Linear Searching	Binary Searching
Linear Searching performed on sorted as well as an unsorted list of items.	Binary Searching performed on sorted list only.
Linear searching is performed with an array and linked list.	Binary searching has only performed with an array, but not on linked lists.
In average case, more comparisons are required.	Comparisons are less in the average case.
Complexity is in the average case and worst case $O(n)$	The complexity is $O(\log n)$ in the average case and worst case.

Limitation of the Binary Searching algorithm

The algorithm requires two conditions:

1. The list must be sorted
2. One must have direct access to the middle element

Binary searching cannot perform on linked lists, as in linked list we cannot get direct access to the middle element.

Interpolation Search

Another efficient searching technique that operates on ordered list is interpolation search. The list of elements must be sorted and the elements should be uniformly or near uniformly distributed on the list. Interpolation search estimates the position of the key value, taking into account the lowest and highest elements in the array and the length of the array. For example, if the key value is close to the highest element in the array, it is likely to be located near the end of the array.

Suppose we are searching for a key K in the list A.

Algorithm of Interpolation Search

Algorithm: INTERPOLATION_SEARCH(A, N, KEY, LOC)

[A is an array of N elements, Key is the searching key, LOC is the location of the KEY]

1. Set I = 0, J = N-1
2. Repeat steps 3 and 4 while I<=J
3. MID = I + (KEY-A[I]) (J-I) / (A[J]-A[I])
4. If KEY = A[MID] then Goto step 5
 Else If KEY < A[MID] Set J=MID-1
 Else Set I = MID+1
 [End of If]
 [End of Loop]
5. If A[MID] = KEY then
 Set LOC = MID
 Else Set LOC = -1
 [End of If]
6. Return

Time complexity of Interpolation Search

Interpolation search is slower than binary search for small arrays, as interpolation search requires extra computation, and the slower growth rate of its time complexity compensates for this only for large arrays.

Interpolation Searching algorithm	Best Case	Average Case	Worst Case
	O(1)	O(loglog n)	O(n)

SORTING

Sorting refers to the operation of rearranging the data elements of a list in some specified order (ascending or descending). Many applications require sorting of a list of data elements. Given a list of data elements of an array of n elements ($A_1, A_2, A_3..A_n$), which can be arranged in ascending (increasing) order or in descending (decreasing) order. Such that:

Ascending order: $A_1 < A_2 < A_3 < \dots < A_n$.

Descending order: $A_1 > A_2 > A_3 > \dots > A_n$.

In this chapter, we will discuss different sorting techniques to sort data elements of the array in ascending order.

There are many different sorting techniques such as

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort
7. Shell Sort
8. Radix Sort
9. Bucket Sort
10. Counting Sort

Internal and External Sorting

When the list of elements or records to be sorted are small enough to be accommodated in the internal (primary) memory of the computer, then it is referred to as internal sorting. Bubble sort, insertion sort, selection sort, merge sort, shell sort, quick sort, heap sort, radix sort is internal sorting techniques.

On the other hand, if the list of elements or records to be sorted in files is accommodated in external storage devices (secondary memory, such as hard disks) then the sorting is referred to as external sorting. One example of external sorting is the external merge sort algorithm.

In-Place and Stable Sorting

The sorting algorithms can be divided into two main classes: In-Place and Stable.

A sorting algorithm is in-place only if a constant amount of data elements of an input array is never stored outside the array. No additional storage is required and hence it is possible to sort a large list without the need of additional array for storage. Bubble sort, insertion sort, selection sort, Quick sort and Heap sort are in-place sorting algorithms. Merge-sort is not an in-place algorithm, as it requires extra array storage.

7	3 ₁	3 ₂	5 ₁	8	5 ₂	2	1	3 ₃
---	----------------	----------------	----------------	---	----------------	---	---	----------------

(a)

1	2	3 ₁	3 ₂	3 ₃	5 ₁	5 ₂	7	8
---	---	----------------	----------------	----------------	----------------	----------------	---	---

(b)

1	2	3 ₂	3 ₁	3 ₃	5 ₂	5 ₁	7	8
---	---	----------------	----------------	----------------	----------------	----------------	---	---

(c)

Figure 10.3: (a) Unsorted list, (b) Stable sort, (c) Unstable sort

A sorting algorithm is said to be stable if two elements that are equal (i.e. duplicate element) remain in the same relative order even after performing sorting. Suppose K₁ and K₂ are two keys such that k₁ = k₂, and POS (K₁) < POS (K₂) where POS (K_i) is the position of the keys in the unsorted list, then after sorting, POS (K₁) < POS (K₂) where POS (K_i) is the position of the keys in the sorted list.

Bubble sort and insertion sort can be implemented as stable algorithms, but selection sort cannot (without significant modifications). Merge-sort is a stable algorithm, but Quicksort and Heapsort are not stable algorithms.

Online and Offline Sorting

A sorting algorithm can start even if not all the elements are available at the beginning, is known as online sorting. Suitable when data items are arriving over a communication channel one by one. Insertion sort is an online sorting algorithm.

An offline sorting algorithm cannot start unless all the data items are present at the beginning. All the elements must be available beforehand. Bubble sort, selection sort, quick sort, merge sort are offline sorting algorithm.

Comparison based Sorting and Non-comparison based Sorting

A comparison sort examines the list of data elements only by comparing two elements with a comparison (relational) operator. In comparison-based sorting access input array elements only by using comparisons. It is a general-purpose sorting method. Bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort, shell sort, etc. are examples of comparison-based sorting algorithms. It can be proved that every comparison-based sorting algorithm must have an average or worst case running time lower bound of $\Omega(n \log n)$, that means comparison based sorting algorithms cannot perform better than $O(n \log n)$ in the average or worst case.

Other sorting algorithms are not comparison sorts. As such, they are not limited by a $\Omega(n \log n)$ lower bound. Bucket sort, counting sort, radix sort are examples of non-comparison-based sorting, they are faster than the $\Omega(n \log n)$, although they have some limitations. Non-comparison-based sorting typically require specialized versions for each data-type and applicable only for a small range of elements.

Bubble Sort

Bubble sort (also known as sinking sort) is a simple sorting algorithm that performs comparisons on each pair of adjacent elements, the first two elements in the array, then between the second and third element, then the third and fourth element, and so on and interchanging them if they are in the wrong order. The algorithm is named in the way smaller elements "bubble" to the top of the list.

Suppose A is an array with n elements from A [0] to A[N-1]. The bubble sorting algorithm performs as below.

- In the first pass, compare each pair of adjacent elements A [0] and A[1], A[1] and A[2], A[2] and A[3], until A[N-2] and A[N-1] and arrange them in proper order by interchanging them if they are in the wrong order. Therefore, in this pass, N-1 comparisons are required and the trivially largest element is placed in the N-1 position of the array at the end of pass 1.
- In the second pass, compare each adjacent element A [0] and A [1], A [1] and A [2], A [2] and A [3], until A[N-3] and A[N-2] and arrange them in proper order. Therefore, in this pass, N-2 comparisons are required and the second largest element is placed in the N-2 position of the array at the end of pass 2.
- In the third pass, compare each adjacent element A [0] and A [1], A [1] and A [2], A [2] and A [3], until A[N-4] and A[N-3] and arrange them in proper order. Therefore, in this pass, N-3 comparisons are required and the third largest element is placed in the N-3 position of the array at the end of pass 3.

- Finally, in the N-1 pass, compare A [0] and A [1] and arrange them in proper order. Therefore, in this pass one comparison is required.
- The first array element A [0] is the apparently smallest element of the array. Therefore, array A of N elements from A [0] to A [N-1] is sorted after N-1 passes.

Example:

Input Data: 14, 33, 27, 35 and 10

[0]	[1]	[2]	[3]	[4]
14	33	27	35	10

(a)

In the first pass compare each adjacent element and change their position if they are in reverse order. In this pass four comparisons and two interchanges are required: 33↔27 and 35↔10. At the end of the first pass largest element is placed in the N-1 position. So after first pass the array will be:

[0]	[1]	[2]	[3]	[4]
14	27	33	10	35

(b)

In the 2nd pass perform the same task. In this pass three comparisons and one interchange is needed: 33↔10. After 2nd pass, the array will be:

[0]	[1]	[2]	[3]	[4]
14	27	10	33	35

(c)

In the 3rd pass two comparisons and one interchange is needed 27↔10. After 3rd pass, the array will be as follows:

[0]	[1]	[2]	[3]	[4]
14	10	27	33	35

(d)

In the 4th pass one comparison and one interchange are needed 14↔10. After performing 4th pass the array will look like:

[0]	[1]	[2]	[3]	[4]
10	14	27	33	35

(e)

Figure 10.4 (a-e): Bubble Sort

It is observed that after the 4th pass the array becomes sorted. So, the desired goal is achieved here.

Algorithm of Bubble Sort

Algorithm: BUBBLE_SORT (A, N)
[A is an array of N elements]
 1. Set K = 1

```

2. Repeat steps 3 to 5 while K<N
3.   Set I = 0
4.   Repeat while I<= N - K then
      a) If A[I] > A[I+1] then
          Temp = A[I]
          A[I] = A[I+1]
          A[I+1] = Temp
          [End of If]
      b) Set I = I + 1
      [End of Loop]
5.   Set K = K + 1
      [End of Loop]
6. Return

```

Analysis / Time complexity of Bubble Sort

Best case / Worst case / Average case time complexity: In all the cases, the total number of comparisons:

$$f(n) = (n-1) + (n-2) + \dots + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Bubble Sort algorithm	Best Case	Average Case	Worst Case
	$O(n^2)$	$O(n^2)$	$O(n^2)$

Refined Bubble Sort

Bubble sort may run in $O(n)$ on sorting the data and works well on nearly sorted data. Refined bubble sort uses an extra Flag variable to indicate when no interchange takes place during a pass. When Flag=False after any pass, then the list is already sorted and there is no need to continue. This may cut down the number of passes. However, while using such Flag variable, one must initialize, reset and compare the variable during each pass.

Algorithm of Refined Bubble Sort

```

Algorithm: BUBBLE_SORT (A, N)
[A is an array of N elements]
1. Set K = 1
2. Repeat steps 3 to 5 while K<N
3.   Set I = 0, Flag = False
4.   Repeat while I<= N-K then
      a) If A[I]>A[I+1] then
          Temp = A[I]
          A[I] = A[I+1]
          A[I+1] = Temp
          Flag = True
      [End of If]

```

```

        b) Set I = I + 1
        [End of Loop]
        If Flag = False then Return
5.      Set K = K + 1
        [End of Loop]
6.  Return

```

Time complexity of Refined Bubble Sort

Best-case time complexity: The best case occurs when the array is already sorted. Total number of comparisons:

$$f(n) = n-1 = O(n)$$

Average-case / Worst-case time complexity: In both cases total number of comparisons:

$$f(n) = (n-1) + (n-2) + \dots + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Refined Bubble Sort algorithm	Best Case	Average Case	Worst Case
	$O(n)$	$O(n^2)$	$O(n^2)$

Although popular, bubble sort has poor performance on random data. This observation is justified as shown in the following figure where bubble sort is nearly three times as slow as insertion sort.

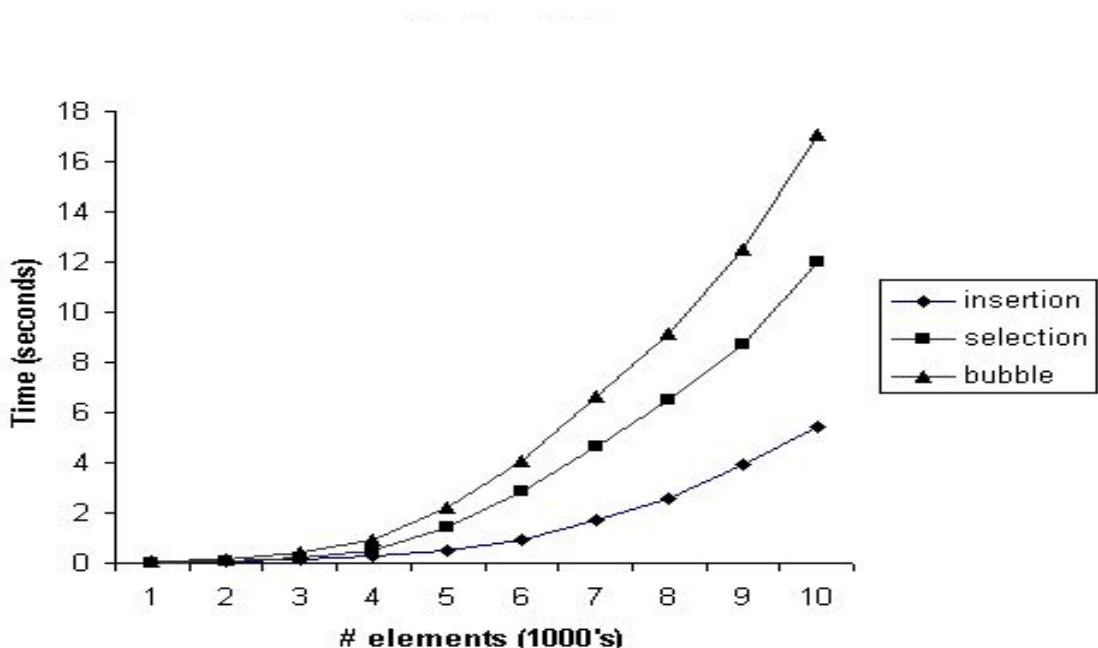


Figure 10.5: Performance graph of Insertion, Selection and Bubble sort

Insertion Sort

The intuitive idea is to insert a new element in the sorted array one by one (known as card player sort). As the insertion operation is performed in this sorting technique, hence it is named as insertion sort. This algorithm is known as on-line sorting algorithm as it can start even if not all the elements are present at the beginning.

Suppose A is an array with n elements from A[0] to A[N-1]. The insertion sort algorithm, inserts each element A[K] into the sorted sub-array A[0], A[1]],... A[K-1], so that after insertion the sub-array A[0], A[1], A[2],..A[K] remains sorted.

- The first array element A[0] by itself is trivially sorted.
- In the first pass, A[1] is inserted into its proper position so that, A[0], A[1] is sorted.
- In the second pass, A[2] is inserted into its proper position in A[0], A[1] , so that, A[0], A[1], A[2] is sorted.
- In the third pass, A[3] is inserted into its proper position in A[0], A[1], A[2] , so that, A[0], A[1], A[2], A[3] is sorted.
- In the N-1 pass, A[N-1] is inserted into its proper position in A[0], A[1], A[2], A[3], A[N-2], so that, A[0], A[1], A[2],..A[N-1] is sorted.

Algorithm of Insertion Sort

Algorithm: INSERTION_SORT (A, N)

[A is an array of N elements]

1. Repeat steps 2 to 4 For K = 1 to N
2. Set Temp = A[K] and P = K - 1
3. Repeat while Temp < A[P] and P >= 0
 - a) Set A[P+1] = A[P]
 - b) Set P = P - 1

[End of loop]
4. Set A[P+1] = Temp
- [End of loop]
5. Return

In this algorithm, two types of comparisons are required: one comparison checks for the element being searched for, and another comparison is required to check the search remains within the bounds of the list being searched. The number of comparisons can be reduced into one by using a so-called sentinel.

Generally, sentinels can be used in insert sort algorithms to make them more efficient, effectively halving the number of comparisons in a loop. We can use minimum element of the list as a sentinel. It takes three steps to sort the list:

- i) Find the minimum element of the list.
- ii) Interchange the minimum element with the 1st element of the list.
- iii) Finally, perform an efficient insertion sort on the list.

However, finding the minimum element of the list requires extra time. Alternatively, we can use a special value as a sentinel that requires extra memory space. Assume that the array of size n actually stored from 1 to N positions and initialize A [0] with $-\infty$. Here, $-\infty$ used as a sentinel element, in C language that can be represented by INT_MIN. This ensures that whatever may be the value of Temp,

always $A[0] \leq \text{Temp}$.

Temp will be found among the location 1, 2... N, but never occupy the first position of the array. The first two array elements A[0] and A[1] by themselves are trivially sorted.

Example:

Input Data: 60, 80, 20, 50, 10, 15, 95 and 90

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
1	-∞	60	80	20	50	10	15	95	90
2	-∞	60	80	20	50	10	15	95	90
3	-∞	20	60	80	50	10	15	95	90
4	-∞	20	50	60	80	10	15	95	90
5	-∞	10	20	50	60	80	15	95	90
6	-∞	10	15	20	50	60	80	95	90
7	-∞	10	15	20	50	60	80	95	90
Sorted List:	-∞	10	15	20	50	60	80	90	95

Figure 10.6: Insertion Sort

Algorithm of Efficient Insertion Sort

Algorithm: INSERTION_SORT (A, N)

[A is an array of N elements]

1. Set $A[0] = -\infty$
2. Repeat steps 3 to 5 for $K = 2$ to N
3. Set $\text{Temp} = A[K]$ and $P = K-1$
4. Repeat while $\text{Temp} < A[P]$
 - a) Set $A[P+1] = A[P]$
 - b) Set $P = P - 1$
[End of loop]
5. Set $A[P+1] = \text{Temp}$
[End of loop]
6. Return

Time complexity of Insertion Sort

Best-case time complexity: The best case occurs when the array is already sorted. Thus, only one comparison is required in each pass, which is between the newly inserted elements with the last

element of the sorted sub-list. In this case, total number of comparisons:

$$f(n) = 1 + 1 + 1 + \dots + 1 = \sum_{i=1}^{n-1} 1 = n-1 = O(n)$$

Worst-case time complexity: The worst case occurs when the array is in reverse order. Thus, we need to compare with all the elements of sorted sub-list in each pass. In this case, total number of comparisons:

$$f(n) = 1 + 2 + 3 + \dots + (n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Average-case time complexity: In average case, as on an average, half of the elements of the sorted sub-list are less than the inserted element and half of the elements are greater. Thus, we need to compare half of the size of sorted sub-list in each pass. In this case, total number of comparisons:

$$f(n) = \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \dots + \frac{n-1}{2} (n-1) = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4} = O(n^2)$$

Insertion Sort algorithm	Best Case	Average Case	Worst Case
	O(n)	O(n ²)	O(n ²)

Selection Sort

In selection sorting algorithm, first select the smallest element in the list and place it in the first position. Then select the second smallest element in the list and place it in the second position, and so on. As selection operation is performed in this sorting technique, hence it is named as selection sort.

Suppose A is an array with N elements from A[0] to A[N-1]. The selection sorting algorithm, select the Kth smallest element A [K] from the array and place into the K-1 position.

- In the pass 1, select the position POS of the smallest element in the list of N elements and then swapping A[POS] and A[0] so that, A[0] is sorted.
- In the pass 2, select the position POS of the smallest element in the sub-list of N-1 elements A[1], A[2]...A[N-1] and then swapping A[POS] and A[1] so that, A[0], A[1] is sorted.
- In the pass 3, select the position POS of the smallest element in the sub-list of N-2 elements A[2], A[3],...,A[N-1] and then swapping A[POS] and A[2] so that, A[0], A[1], A[2] is sorted.
- In the pass N-1, select the position POS of the smallest element in the sub-list of 2 elements A[N-2], A[N-1] and then swapping A[POS] and A[N-2] so that, A[0], A[1], A[2],...,A[N-2] is sorted.
- The last array element A [N-1] is the apparently largest element of the array.

Therefore, array A of N elements from A [0] to A [N-1] is sorted after N-1 passes.

Algorithm of Selection Sort

Algorithm: SELECTION_SORT (A, N)

[A is an array of N elements]

1. Repeat steps 2 and 3 for K=1 to N-1
2. Call MIN(A, K, N, LOC)
3. If A[K] ≠ A[LOC] then

```

    a) Temp = A[K]
    b) A[K] = A[LOC]
    c) A[LOC] = Temp
  [End of If]
4. Return

```

Function: MIN (A, K, N, LOC)

[A is an array of N elements, K is the starting index of sub-array, LOC is the the location the minimum element of the sub-array]

1. Set MIN = A[K], LOC = K
2. Repeat for J = K + 1 to N
 - If MIN > A[J] then
 - a) Set MIN = A[J]
 - b) Set LOC = J
- [End of If]

[End of loop]

3. Return

Time complexity of Selection Sort

Best case / Worst case / Average case time complexity: In the first pass, all the adjacent elements are compared and the largest element is placed in its proper position after (n-1) comparisons. In pass 2, second largest element is placed at after (n-2) comparisons. After (n-1) passes, total number of comparisons:

$$f(n) = (n-1) + (n-2) + \dots + 3 + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Selection Sort algorithm	Best Case	Average Case	Worst Case
	$O(n^2)$	$O(n^2)$	$O(n^2)$

Quick Sort

The Quicksort (also called partition-exchange sort) is one of the fastest sorting algorithm known and it is the method of choice in most sorting libraries. As this sorting technique has best average behaviour among all the sorting methods, hence it is named as quick sort. Quicksort is based on the divide-and-conquer strategy, which is developed by C.A.R. Hoare in 1959.

In quick sort, we select an element from the array called pivot and then partitions the array into two sub-arrays. So that, all the elements that are less than the pivot are placed in its left sub-array and all the elements that are greater than the pivot, are placed in its right sub-array. This is called the partition operation. After the partition, the pivot is in its final position. Next, we apply the same process recursively to two independent sub-array of smaller size.

Suppose A is an array with N elements from A [0] to A [N-1]. At first, select the A[0] as a pivot, placed it in POS position in such a way that it partitions the array into two sub-arrays, all the elements that are less than pivot are placed from A[0] to A[POS-1] and all the elements that are greater than pivot are placed from A[POS+1] to A[N-1]. In the next step, apply the same process recursively to two sub-arrays A[0] to A[POS-1] and A[POS+1] to A[N-1].

There are many variants of partition operation, mainly Hoare partition scheme and Lomuto partition scheme. The Hoare partition scheme is more efficient than the Lomuto partition scheme.

Example:

Input Data: 54, 26, 93, 17, 77, 31, 44, 55 and 20.

Let, 54 is the pivot element. The last element is denoted by L and the first element is denoted by F.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
54	26	93	17	77	31	44	55	20
↑ Pivot								↑ Last (L)

(a)

As the pivot element is greater than the last element, so a swapping ($54 \leftrightarrow 20$) is performed among them resulting in the following array.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
20	26	93	17	77	31	44	55	54
		↑						↑
		First (F)						Pivot

(b)

Now, read the array from left to right until an element is found to be more than pivot element. After swapping ($93 \leftrightarrow 54$), the array becomes:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
20	26	54	17	77	31	44	55	93
↑ Pivot					↑ Last (L)			

Read the array from right to left until an element is found to be less than pivot element. After swapping ($54 \leftrightarrow 44$) with the elements, the array becomes:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
20	26	44	17	77	31	54	55	93
			↑	First (d)			↑	Pivot

Now, read the array from left to right until an element is found to be more than pivot element. After swapping ($77 \leftrightarrow 54$), the array becomes

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
20	26	44	17	54	31	77	55	93
			↑	↑	Pivot	Last (L)		

Read the array from right to left until an element is found to be less than pivot element. After swapping ($54 \leftrightarrow 31$) with the element the array becomes

0	1	2	3	4	5	6	7	8
20	26	44	17	31	54	77	55	93

↑
Pivot
(f)

Figure 10.7 (a-f) : Partition operation in Quick Sort

Now, the pivot element is placed in the proper position. All the elements in left subarray are less than pivot element and all the elements in the right subarray are greater than pivot element. The same procedure is performed with both subarrays to sort the entire array.

Algorithm of Quick Sort

Algorithm: `QUICK_SORT(A, F, L)`

[*A* is an array of *N* elements, *F* is the first position and *L* is the last position of the sub-array]

1. If *F* \geq *L* then Return
2. Call `PARTITION(A, F, L, POS)`
3. Call `QUICK_SORT(A, F, POS-1)`
4. Call `QUICK_SORT(A, POS+1, L)`
5. Return

Function: `PARTITION(A, F, L, POS)`

[*A* is an array, *F* is the first position and *L* is the last position of the sub-array, *POS* is the position of the pivot]

1. Set *I* = *F*, *J* = *L*, *POS* = *F*
2. Repeat steps 3 and 4 while TRUE
 3. a) Repeat while $A[POS] \leq A[J]$ and $POS \neq J$

$$J = J - 1$$
 [End of Loop]
 - b) If $POS = J$ then Return
 - c) $A[POS] > A[J]$ then
 - i) $Temp = A[POS]$, $A[POS] = A[J]$, $A[J] = Temp$
 - ii) Set $POS = J$
4. a) Repeat while $A[POS] \geq A[I]$ and $POS \neq I$

$$I = I + 1$$
 [End of Loop]
- b) If $POS = I$ then Return
- c) $A[POS] < A[I]$ then
 - i) $Temp = A[POS]$, $A[POS] = A[I]$, $A[I] = Temp$
 - ii) Set $POS = I$
 [End of Loop]

5. Return

Time complexity of Quick Sort

Best-case time complexity: In Quick sort best case occurs when the partitions are as evenly balanced as possible: their sizes are almost equal, each partition has at most $n/2$ elements and the tree of subproblem sizes looks like the following figure.

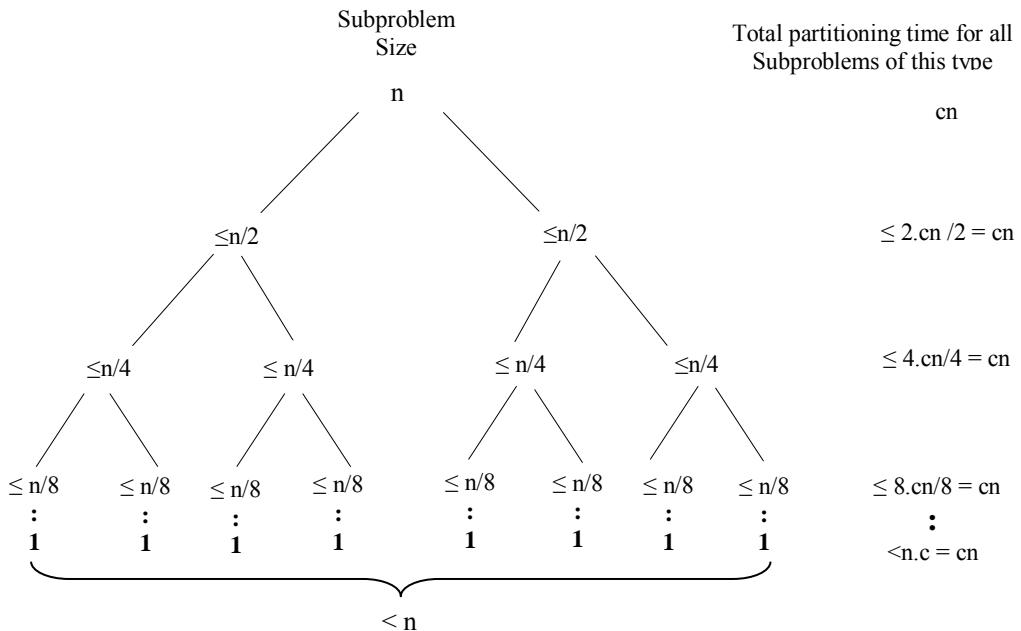


Figure 10.8: Subprogram size in Quick Sort for best case

The total number of comparisons can be calculated recursively:

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ 2T(n/2) + cn, & \text{Otherwise} \end{cases}$$

Where c is a constant

Now,

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2 \cdot [2 \cdot T(n/4) + c \cdot n/2] + cn \\ &= 2^2 T(n/4) + cn + cn \\ &= 2^2 T(n/2^2) + 2cn \\ &\vdots \\ &= 2^r T(n/2^r) + rcn \end{aligned}$$

Now, let $2^r = n$, therefore $r = \log_2 n$

$$\begin{aligned} T(n) &= nT(n/n) + cn\log_2 n \\ &= nT(1) + cn\log_2 n \\ &= cn + cn\log_2 n \quad [\text{hence } T(1) = c] \\ &= O(n\log n) \end{aligned}$$

In the best case of quick sort $\log_2 n$ passes and each pass n number of comparisons are required.

Worst-case time complexity: In quick sort worst case occurs when the array is already sorted or when the pivot chosen by the partition function is always either the smallest or the largest element in the n element subarray. Then one of the partitions will contain no elements and the other partition will contain $n-1$ elements all but the pivot. Therefore, the recursive calls will be on subarrays of sizes 0 and $n-1$ elements. The tree of subproblem sizes looks like the following figure.

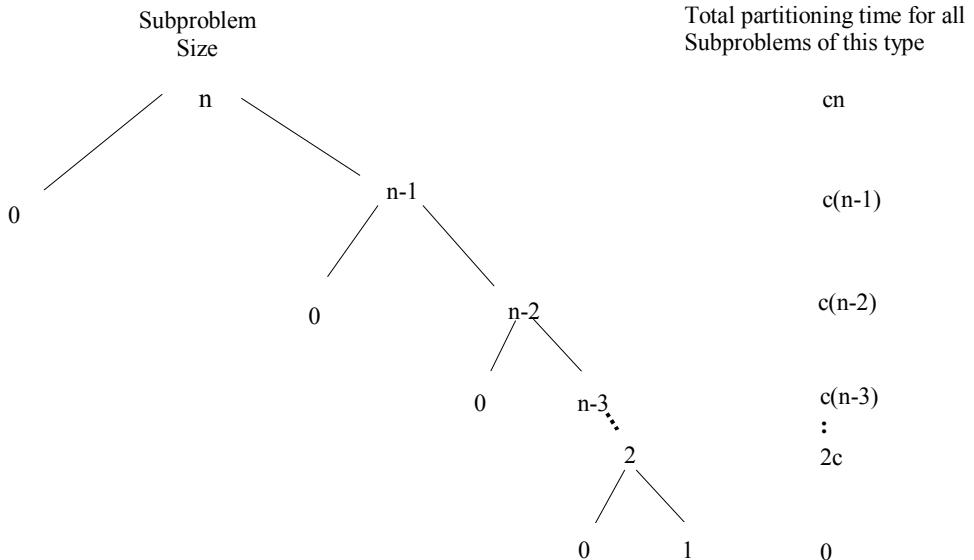


Figure 10.9: Subprogram size in Quick Sort for worst case

The original call takes cn time for some constant c , the recursive call on $n-1$ elements takes $c(n-1)$ time. The recursion call on $n-2$ elements takes $c(n-2)$ time, and so on.

In worst case, total number of comparisons can be calculated recursively:

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ T(n-1) + cn, & \text{Otherwise} \end{cases}$$

Where c is a constant

Now,

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= [T(n-2) + c(n-1)] + cn \\ &= T(n-2) + c[n + (n-1)] \\ &\vdots \\ &= T(n-r) + c[n + (n-1) + \dots + (n-r+1)] \\ &= T(n-n+1) + c[n + (n-1) + \dots + (n-n+1+1)] \quad [\text{when } r = n-1] \\ &= T(1) + c[n + (n-1) + \dots + 2] \quad [\text{hence } T(1) = c] \\ &= c + c[n + (n-1) + \dots + 2] \\ &= c[n + (n-1) + \dots + 2 + 1] \\ &= c \cdot n(n+1)/2 \\ &= O(n^2) \end{aligned}$$

In the worst case of quick sort n passes and each pass n number of comparisons are required.

Average-case time complexity: In average case, not always get evenly balanced partitions, but always get at worst a 3-to-1 split on an average. Let, each time we get one partition with $n/4$ elements and other partition with $3n/4$ elements. The tree of subproblem sizes and partitioning time looks like the following figure.

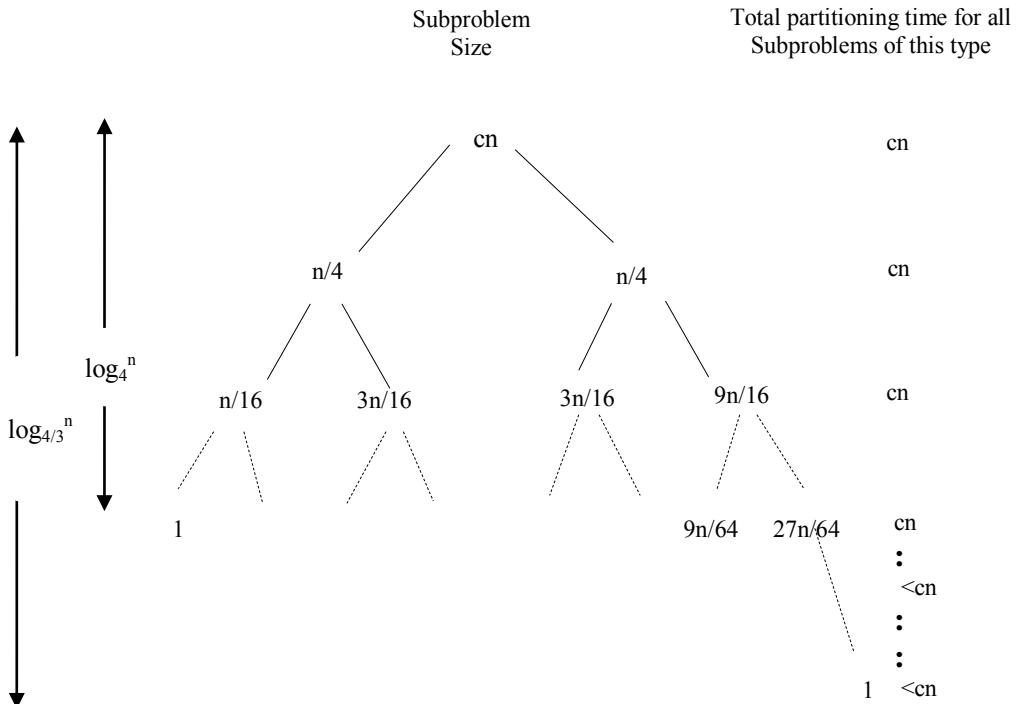


Figure 10.10: Subprogram size in Quick Sort for average case

There are maximum $\log_{4/3} n$ levels in the tree and partitioning time for every level at most cn . Therefore, if all the splits are 3-to-1 then total average running time is $cn \log_{4/3} n = \frac{cn}{\log 4/3} \log n = O(n \log n)$.

Quick Sort algorithm	Best Case	Average Case	Worst Case
	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

In the quicksort, the leftmost element of the array would often be chosen as the pivot element. However, for this reason, when the array is sorted then the time complexity becomes $O(n^2)$ in worst-case, which is a rather common use-case. The problem was easily solved by either choosing a random index for the pivot or choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. Then the time complexity will become $O(n \log n)$ in worst case.

Merge Sort

Mergesort algorithm is a classic example of recursive divide and conquer. This algorithm was invented by **John von Neumann**. As the merging operation is performed in this sorting technique, hence it is

named as merge sort. The Mergesort algorithm can be described in general terms as consisting of the following three steps:

Suppose A is an array with N elements from A[0] to A[N-1].

1. **Divide Step:** If given array A has zero or one element, then return, as it is already sorted. Otherwise, divide A into two sub-arrays, $A_1 = A[0], A[1], \dots, A[\lfloor N/2 \rfloor - 1]$ and $A_2 = A[\lfloor N/2 \rfloor], A[\lfloor N/2 \rfloor + 1], \dots, A[N-1]$, each containing about half of the elements of A.
2. **Recursion Step:** Recursively sort two sub-array A_1 and A_2 .
3. **Conquer Step:** Combine the elements back in A by merging the two sorted arrays A_1 and A_2 into a single sorted sequence.

Example:

Input Data: 8, 6, 3, 5, 2, 7, 4 and 1.

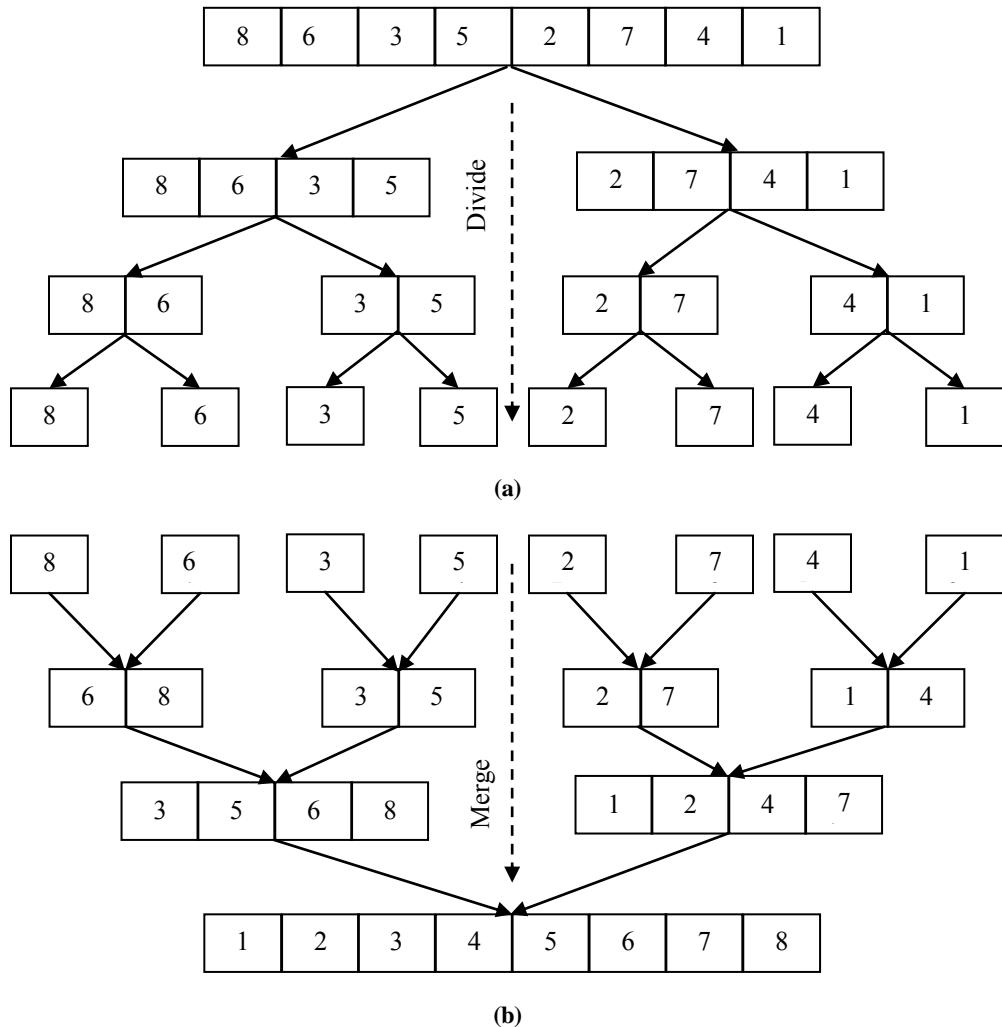


Figure 10.116: (a) Divide phase and (b) Combine phase

Algorithm of Merge Sort**Algorithm: MERGE_SORT(A, F, L)**

[A is an array of N elements, F is the first position and L is the last position of the sub-array]

1. If F < L then
 - a) Set MID = (F + L)/2
 - b) Call MERGE_SORT(A, F, MID)
 - c) Call MERGE_SORT(A, MID+1, L)
 - d) Call MERGE(A, F, L, MID)

2. Return

Function: MERGE(A, F, L, MID)

[A is an array, F is the first position, L is the last position and MID is the middle position of the sub-array]

1. Set I = F, J = MID + 1, K = F
2. Repeat while I <= MID and J <= L
 - If A[I] < A[J] then
 - a) Set C[K] = A[I]
 - b) Set I = I + 1
 - c) Set K = K + 1
 - ELSE
 - a) Set C[K] = A[J]
 - b) Set J = J + 1
 - c) Set K = K + 1
 - [End of If]
- [End of Loop]
3. Repeat while I <= MID
 - a) Set C[K] = A[I]
 - b) Set I = I + 1
 - c) Set K = K + 1
- [End of Loop]
4. Repeat while J <= L
 - a) Set C[K] = A[J]
 - b) Set J = J + 1
 - c) Set K = K + 1
- [End of If]
- [End of Loop]
5. Repeat For I = F to K
 - Set A[I] = C[I]
6. Return

Time complexity of MergeSort

Best-case / Worst-case / Average-case time complexity: In Merge-sort the partitions are always evenly balanced, their sizes are equal, each partition has $n/2$ elements. Each of the subproblems of size $n/2$

recursively sorts two subarrays of size $n/4$. There are four subproblem of size $n/4$ and merging time for each will be $cn/4$, for a total merging time of $4.cn/4 = cn$. The tree of subproblem sizes and total merging time for all subproblems looks like the following figure.

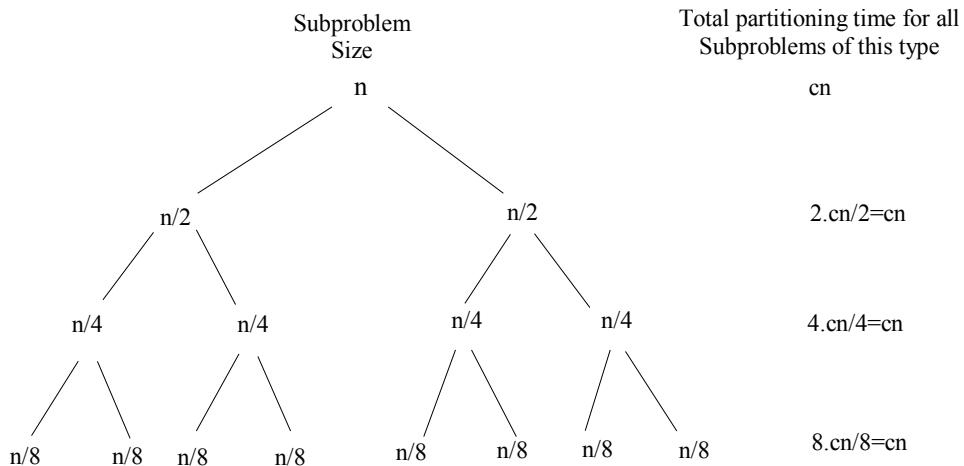


Figure 10.12: Subprogram size in Merge Sort

For all the cases total number of comparisons can be calculated recursively

$$T(n) = \begin{cases} c, & \text{when } n=1 \\ 2T(n/2) + cn, & \text{otherwise} \end{cases}$$

Where c is a constant

Now,

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2.[2.T(n/4) + cn/2] + cn \\ &= 2^2T(n/4) + cn + cn \\ &= 2^2T(n/2^2) + 2cn \\ &\vdots \\ &= 2^rT(n/2^r) + rcn \end{aligned}$$

Now, let $2^r = n$, therefore $r = \log_2 n$

$$\begin{aligned} T(n) &= nT(n/n) + cn\log_2 n \\ &= nT(1) + cn\log_2 n \\ &= cn + cn\log_2 n \quad [\text{hence } T(1)=c] \\ &= O(n\log n) \end{aligned}$$

Merge Sort algorithm	Best Case	Average Case	Worst Case
	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

Space complexity of MergeSort

The space complexity of the merge sort is $O(n + \log n)$, n for the auxiliary array and $\log n$ for the stack space.

Heap Sort

The Heapsort is more efficient and improved version of Selection sort. It also works by determining the largest element of the list, placing that at the end of the list, then continuing with the rest of the list. Heapsort is an in-place sorting algorithm, but it is not a stable sort. Heap-sort was invented by **J. W. J. Williams** in 1964.

The heapsort is based on a data structure called heap, a special kind of binary tree. Recall that, in heap, the left child of A[K] is stored at A[2*K] and right child is stored at A[2*K+1]. As this sorting technique uses heap data structure, hence it is named as heap sort.

Once the data list has been made into a max-heap, the root node is guaranteed to be the largest element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root.

The operation for creating a heap is also referred as **heapify** helps to build a heap from a list of data items. Now, heapify operation has two variations: shift-up and shift-down.

The shift-up process begins with an empty heap, then successive inserts elements in the tree and build a heap by the shift-up operation. Here, the building of heap is done using top-down approach. In shift-up operation, if the newly inserted element is greater (max-heap) or smaller (min-heap) than its parent then it needs to be interchanged with parent.

Example:

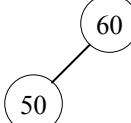
Build the heap using shift-up process: 60, 50, 30, 10, 80, 70, 20 and 40

Insert 60:



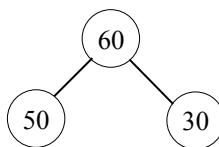
(a)

Insert 50:



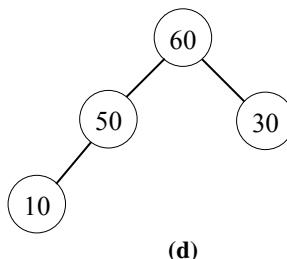
(b)

Insert 30:



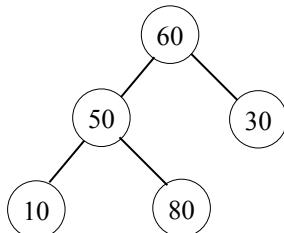
(c)

Insert 10:



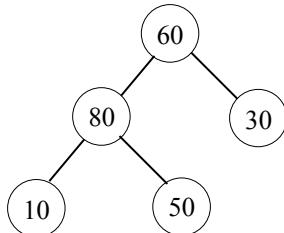
(d)

Insert 80:



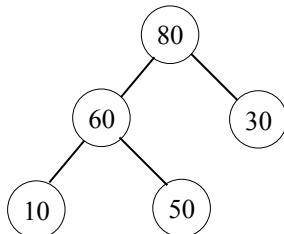
(e)

Now, since 80 is greater than its parent node 50, so interchange with its parent node.



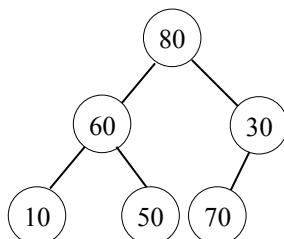
(f)

Now, again 80 is greater than its parent node 60, so interchange with its parent node.



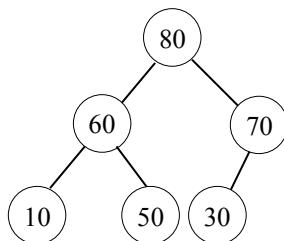
(g)

Insert 70:



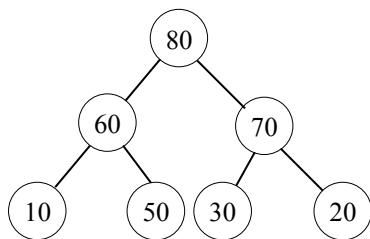
(h)

Now, since 70 is greater than its parent node 30, so interchange with its parent node.



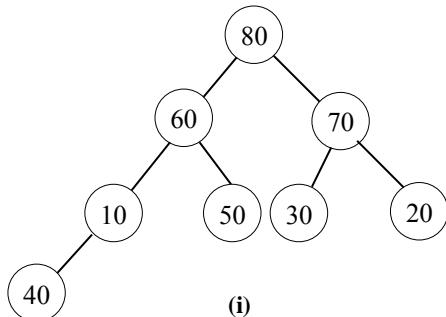
(i)

Insert 20:



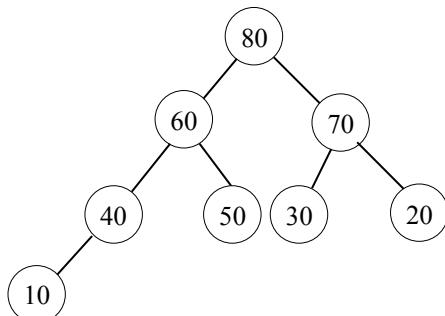
(h)

Insert 40:



(i)

Now, since 40 is greater than its parent node 10, so interchange with its parent node.



(j)

Figure 10.13 (a-j): Create Max-heap using shift-up

The shift-down process presents the entire input array as a complete binary tree, but non-heap and maintains it starts from the last non-trivial sub-heap (that is, the last parent node). Here, the building of the heap is done using a bottom-up approach and nodes are successively shifting downward to establish the heap property. The elements $A[N/2+1], A[N/2+2], \dots, A[N]$, have no children, each of these elements is a sub-heap of size 1. Now, shift-down operation is performed for each I where $I=N/2, N/2-1, \dots, 1$. Each of the two children of $A[I]$ are the root of a sub-heap, so calling shift-down makes $A[I]$ into the root of its own sub-heap. In shift-down operation, if one of the child is greater (max-heap) or smaller (min-heap) than its children, then it needs to be interchanged with that child.

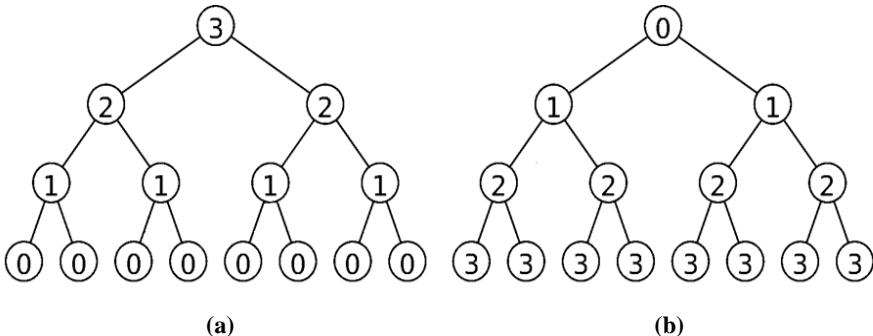


Figure 10.14: (a) Bottom-up (Shift-Down), (b) Top-Down (Shift-Up). The number in the circle indicates the maximum time of interchanges are needed when building the Heap.

This shift-down version is more efficient than shift-up where insertion operation is called N times. Note that, for $N/2$ elements we call shift-down for $N/4$ elements on a sub-heap rooted at $A[I]$ and whose height is 1. For $N/8$ elements, we call shift-down on a sub-heap whose height is two, and so on. In the figure 10.14, the number in the circle indicates the maximum time of interchanges are needed when building the heap. The shift-down version needs a less number of interchanges than shift-up version. Therefore, typically shift-down version is used to build a heap.

Now, heap sort algorithm can be divided into two parts. Suppose A is an array with N elements from $A[1]$ to $A[N]$.

- In the first step, a max-heap is built from the N data items. As a result, the root $A[1]$ is to be the largest element.
- In the second step,
 - The interchange between $A[1]$ and $A[N]$. Trivially, the largest element is placed in the N position of the array at the end of pass 1. Decreasing the heap size by one and shifting down the new first value $A[1]$ into its proper position to rebuild the heap.
 - The interchange between $A[1]$ and $A[N-1]$. The second largest element is placed in the $N-1$ position of the array at the end of pass 2. Decreasing the heap size by one and shifting down the new first value $A[1]$ into its proper position to rebuild the heap.
 - The interchange between $A[1]$ and $A[N-2]$. The third largest element is placed in the $N-2$ position of the array at the end of pass 3. Decreasing the heap size by one and shifting down the new first value $A[1]$ into its proper position to rebuild the heap.
 - Finally, the interchange between $A[1]$ and $A[2]$. Decreasing the heap size by one and trivially, the smallest element is in $A[1]$.

Therefore, the array of N elements from $A[1]$ to $A[N]$ is sorted after $N-1$ passes.

Example:

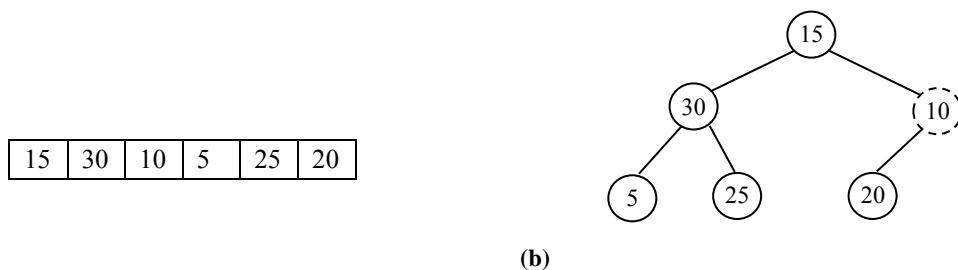
Input Data: 15, 30, 10, 5, 25 and 20.

In the first step a max-heap is built from the given data items. The list represented as a complete tree, but not ordered.



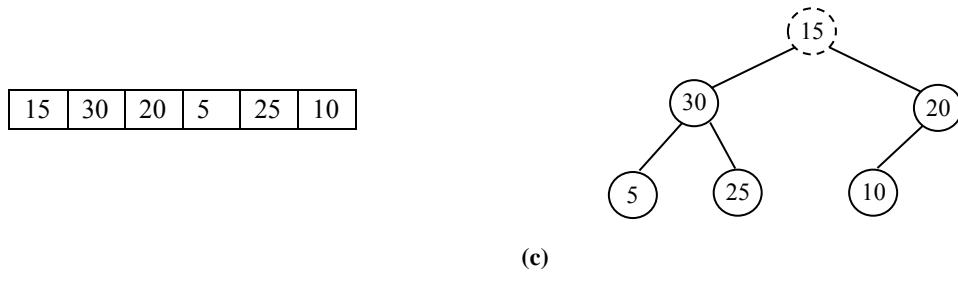
(a)

Start from the node at $N/2 = 6/2 = 3$ position, it has one greater child and has to be downwards gradually. Therefore, the 3rd element has to be interchanged with 6th element.



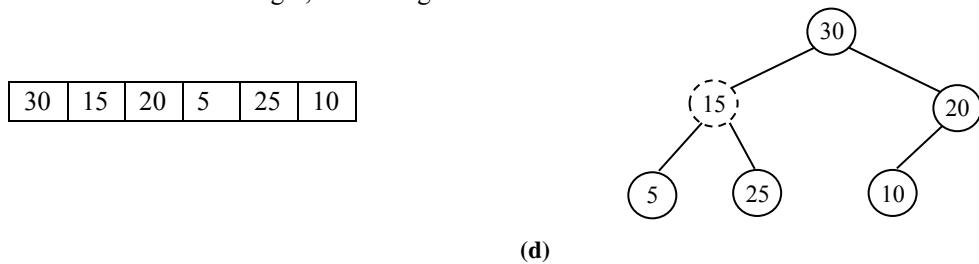
(b)

After processing 3rd element:



(c)

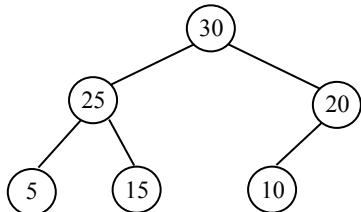
Now for the 2nd element as its children are smaller, so no downward movement is required. Next the 1st element to be processed, its left child is greater of the children. So, the 1st element has to be down to the left, interchanged with 2nd element. Now the right child of 2nd element is greater and it has to be moved down to the right, interchange with 5th element.



(d)

Now the tree is ordered and the max-heap is built.

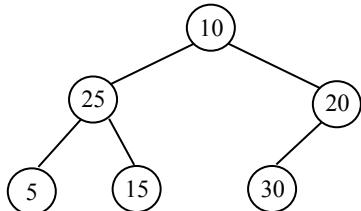
30	25	20	5	15	10
----	----	----	---	----	----



(e)

After building the max-heap, sorting can be started. The root is to be the largest element of the max-heap, interchange with the last element (6th element) of the heap.

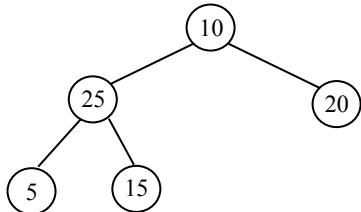
10	25	20	5	15	30
----	----	----	---	----	----



(f)

Decrease the size of the heap by deleting last element.

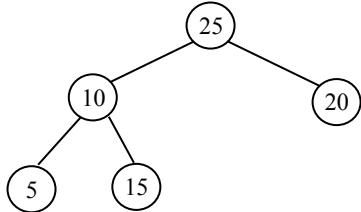
10	25	20	5	15	30
----	----	----	---	----	----



(g)

Now to rebuild the heap, the 1st element (10) will be gradually moved downwards. As its left child is greater of the children, so, the 1st element has to be down to the left, interchanged with 2nd element.

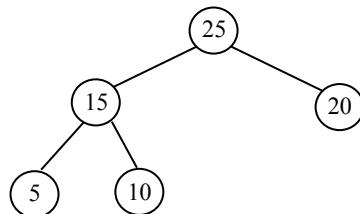
25	10	20	5	15	30
----	----	----	---	----	----



(h)

Again, its right child is greater of the children, it has to be down to the right, interchanged with 5th element.

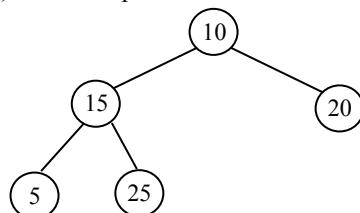
25	15	20	5	10	30
----	----	----	---	----	----



(i)

Now again the tree is ordered and the max-heap is built. The root is to be the largest element of the max-heap, interchange with the last element (5th element) of the heap.

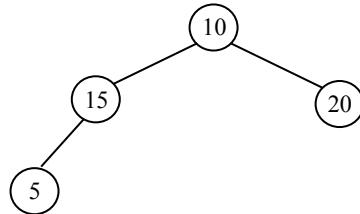
10	15	20	5	25	30
----	----	----	---	----	----



(j)

Decrease the size of the heap by deleting last element.

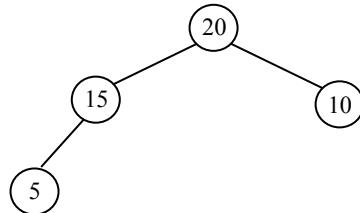
10	15	20	5	25	30
----	----	----	---	----	----



(k)

Now to rebuild the heap, the 1st element (10) will be gradually moved downwards. As its right child is greater of the children, so, the 1st element has to be down to the right, interchanged with 3rd element.

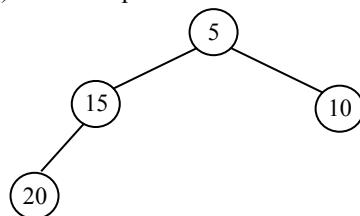
20	15	10	5	25	30
----	----	----	---	----	----



(l)

Now again the tree is ordered and the max-heap is built. The root is to be the largest element of the max-heap, interchange with the last element (4th element) of the heap.

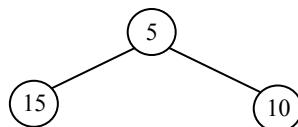
5	15	10	20	25	30
---	----	----	----	----	----



(m)

Decrease the size of the heap by deleting last element.

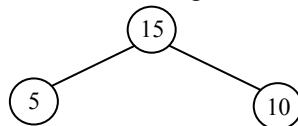
5	15	10	20	25	30
---	----	----	----	----	----



(n)

Now to rebuild the heap, the 1st element (5) will be gradually moved downwards. As its left child is greater than the children, so, the 1st element has to be down to the left, interchanged with 2nd element.

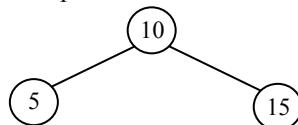
15	5	10	20	25	30
----	---	----	----	----	----



(o)

Now again the tree is ordered and the max-heap is built. The root is to be the largest element of the max-heap, interchange with the last element (3rd element) of the heap.

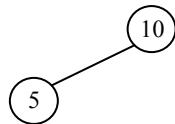
10	5	15	20	25	30
----	---	----	----	----	----



(p)

Decrease the size of the heap by deleting last element.

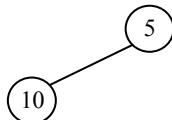
10	5	15	20	25	30
----	---	----	----	----	----



(p)

As the left child is less than 1st element, no downward movement is required, the interchange 1st element with the last element (2nd element) of the heap.

5	10	15	20	25	30
---	----	----	----	----	----



(q)

Decrease the size of the heap by deleting last element.



5	10	15	20	25	30
---	----	----	----	----	----

(r)

5 is the last element of the heap, therefore the array is now sorted.

Figure 10.15 (a-r): Heap Sort

Algorithm of Heap Sort

Algorithm: HEAP_SORT (A, N)

[A is an array of N elements]

1. Repeat For I = N/2 to 1
 Call SHIFT_DOWN(A, I, N)
 End of Loop]
2. Repeat For I = N TO 2
 a) Temp = A[I]
 b) A[I] = A[1]
 c) A[1] = Temp
 d) Call SHIFT_DOWN(A, 1, I-1)
 [End of Loop]
3. Return

Function: SHIFT_DOWN (A, K, N)

[A is an array of N elements, K is the position of the element that shifted down]

1. Set Parent = K, Child = 2* Parent and Temp = A[Parent]
2. Repeat while Child<=N
 - a) If Child < N and A[Child +1] > A[Child]
 then Set Child = Child + 1
 - b) If Temp >= A[Child] Goto Step 3
 - c) Set A[Parent] = A[Child]
 - d) Set Parent = Child and Child = 2* Parent
 [End of Loop]
3. Set A[Parent] = Temp
4. Return

In the above algorithm, efficient shift-down version is used to build a heap by shifting operations instead of interchange operation. When an element gradually moves downwards, it is stored in a temporary variable and if one child is greater than its children, then stores the child to its original

position. After that, it is compared with its descendants. Finally, when none of the descendants is greater than it then restores it to the position of the predecessor.

Time complexity of Heap Sort

The build heap operation is run once and has $O(n)$ time complexity. The shift-down version of heapify has $O(\log n)$ time complexity and is called n times. Therefore, the performance of this algorithm is $O(n + n \log n)$ which evaluates to $O(n \log n)$.

Heap Sort algorithm	Best Case	Average Case	Worst Case
	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Radix Sort

The radix sort is a sorting algorithm that sorts integers by processing individual digits. Radix sort is not limited to integers because integers can represent strings of characters (e.g., names or dates) and specially formatted floating-point numbers. As this sorting technique is based on the radix of the data items, hence it is named as radix sort. Radix sort is an example of non-comparison based sorting.

Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least significant digit and moves towards the most significant digit. MSD radix sort process from most significant digit to least significant digit.

LSD radix sort is suitable for sorting variable length integers and fixed length strings. MSD radix sort is suitable for sorting variable length strings and fixed-length integer representations. LSD radix sort illustrated in the following example.

Example:

Input Data: 347, 142, 361, 423, 538, 126, 320, 549, 365

The numbers are sorted in the first pass according to the (least significant digits) unit digits.

Input	0	1	2	3	4	5	6	7	8	9
347								347		
142			142							
361		361								
423				423						
538									538	
126						126				
320	320									
549										549
365						365				

(a)

On the second pass, the numbers are sorted according to the ten digits.

Input	0	1	2	3	4	5	6	7	8	9
320			320							
361							361			
142					142					
423			423							
365							365			
126			126							
347					347					
538				538						
549					549					

(b)

On the third and last pass, the numbers are sorted according to the (most significant digit) hundred's digits.

Input	0	1	2	3	4	5	6	7	8	9
320				320						
423					423					
126		126								
538						538				
142		142								
347				347						
549						549				
361				361						
365				365						

(c)

Figure 10.16: LSD Radix Sort

Finally sorted numbers after the third pass:

126, 142, 320, 347, 361, 365, 423, 538, 549

Algorithm of Radix Sort

```

Algorithm: RADIX_SORT(A, N)
[Q is an array of Circular Queue of size 10]
[A is an array of N elements]
1. Initialize array Q
2. Set X = maximum element of A
3. Count the digits of X and Set pass = number of digits
4. Set DIV = 1
5. Repeat For I = 1 to pass do
6.   Repeat steps For J = 0 to N-1 do
7.     Set Remainder = A[J] mod (DIV *10)
8.     Set Remainder = Remainder / DIV
9.     Call INSERT(Q[Remainder], A[J])
      [End of Loop]
10. Set DIV = DIV * 10
11. Delete the elements from the queue and place them into A
12. [End of Loop]
13 Return

```

Time complexity of Radix Sort

The lower bound of the comparison-based sorting algorithms (Quick sort, Merge sort, Heap sort etc.) is $\Omega(n \log n)$, i.e. they cannot be better than $n \log n$ to sort n keys.

Now, let there be k average digits in the input data for Radix sort, b is the base for representing numbers, for example, for the decimal system, b is 10. An LSD radix sort takes $O(k*(n+b))$ or $O(nk)$ time. Sometimes k is presented as a constant, which would make the radix sort better (for sufficiently large n) than the best comparison-based sorting algorithms. The space complexity of the radix sort is $O(n+b)$ as it needs space for the original numbers and b buckets to place the n items.

Radix Sort algorithm	Best Case	Average Case	Worst Case
	$O(nk)$	$O(nk)$	$O(nk)$

Shell Sort

Shell sort can be seen as a generalized form of insertion sort. Donald Shell published the first version of this sort in 1959. As Donald Shell developed this sorting technique, hence it is named as Shell sort. The difficulty for insertion sort is that an element moves towards its final position very slowly, one position at a time and it cannot make a long jump quickly to reach its destination.

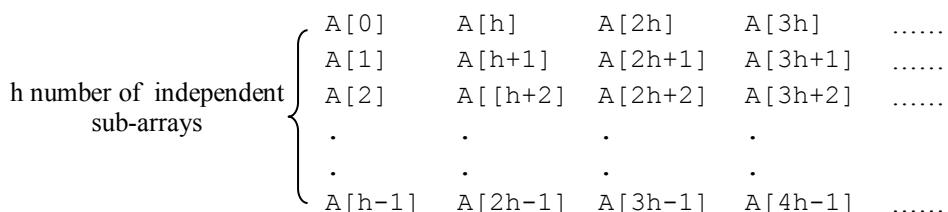


Figure 10.17: h independent sub-arrays

Sort the sub-array using insertion sorting technique. The difference between two consecutive elements of a sub-array is h. In general, an array is to be sorted for several values of h, which forms a monotone decreasing sequence and the last value of h sequence must be always one. The last pass is same as an ordinary insertion sort. More specifically, a sequence of numbers are

$$h_t > h_{t-1} > h_{t-2} > \dots > h_2 > h_1 = 1$$

to be decided and the array is to be sorted over t different passes using the method. The general idea in insertion sort, the predecessor of a [i] is always a [i-1], but in the shell sort predecessor of a [i] is a [i-h] for some value of h.

Example:

Suppose A is an array with 12 elements from A[1] to A[12]. Array elements are: 60, 80, 20, 50, 10, 15, 95, 90, 45, 70, 25 and 30

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
Input Data	60	80	20	50	10	15	95	90	45	70	25	30
After 5-sorting	15	30	20	45	10	25	80	90	50	70	60	95
After 3-sorting	15	10	20	45	30	25	70	60	50	80	90	95
After 1-sorting	10	15	20	25	30	45	50	60	70	80	90	95

Figure 10.18: Shell sort

- At the first pass, 5-sorting ($h = 5$) performs insertion sort on the separate sub-arrays (A[1], A[6], A[11]), (A[2], A[7], A[12]), (A[3], A[8]), (A[4], A[9]) and (A[5], A[10]).
- In the second pass, 3-sorting ($h = 3$) performs insertion sort on the separate sub-arrays (A[1], A[4], A[7], A[10]), (A[2], A[5], A[8], A[11]) and (A[3], A[6], A[9], A[12]).
- In the third and final pass, 1 sorting ($h = 1$), is an ordinary insertion sort of the entire array (A[1], A[2], ...A[12]).

Algorithm of Shell Sort

```

Algorithm: SHELL_SORT(A, N)
[A is an array of N elements]
1. Set GAP = LAST_GAP_SEQUENCE
2. Repeat steps 3 to 8 while GAP >= 1
3. Set I = GAP
4. Repeat steps 5 to 7 For I = GAP to N - 1
5.     Set Temp = A[I]
6.     Repeat while J >= GAP and A[J-GAP] > Temp
        a) Set A[J] = A[J-GAP]
        b) Set J = J - GAP
    
```

```

    [End of loop]
7. Set A[J+GAP] = Temp
    [End of loop]
8. Set GAP = PREV_GAP
    [End of Loop]
9. Return

```

There are different gap sequences; each of them gives correct sort. However, the properties of thus obtained versions of Shell sort may be different. Some of the gap sequences are given in the following table. These are increasing in infinite sequences, whose elements less than n should be used in reverse order.

Table 10.1: Gap Sequences

Gap Sequence	General Term ($k \geq 1$)	Worst-case time complexity
1, 4, 13, 40, 121, ...	$(3^k - 1)/2$ less than $\lceil n/3 \rceil$	$O(n^{3/2})$
1, 3, 5, 9, 17, 33,	$2^k + 1$, prefixed with 1	$O(n^{3/2})$
1, 3, 7, 15, 31,	$2^k - 1$	$O(n^{3/2})$
1, 2, 3, 4, 6, 8, 9, 12...	$2^p 3^q$	$O(n \log^2 n)$

Time complexity of Shell Sort

Worst-case time complexity: The worst-case time complexity of the shell sort depends on the gap sequence (increment sequence). For the gap sequence 1, 4, 13, 40, 121,... the time complexity is $O(n^{3/2})$. It is easy for calculating the previous term of this gap sequence using integer arithmetic:

$$\text{GAP} = \text{GAP} / 3$$

For the gap sequence 1, 2, 3, 4, 6, 8, 9, 12,... the time complexity is $O(n \log^2 n)$. This is the best known performance compare to any other gap sequences, but it uses too many increments. Almost all the gap sequences gives the running time complexity $O(n^{3/2})$, some other gap sequences, time complexity is $O(n^{4/3})$.

Therefore, neither tight upper bounds on time complexity nor the best increment sequence are known.

Shell Sort algorithm	Best Case	Average Case	Worst Case
	$O(n \log^2 n)$	$O(n \log n)$	$O(n^{3/2})$

Shell sort is unstable and it has a higher cache miss ratio than quick sort. However, its low overhead (does not use the call stack), relatively simple implementation, adaptive properties, and sub-quadratic time complexity, shell-sort may be a feasible alternative to the $O(n \log n)$ sorting algorithms for some applications when the data to be sorted is not very large.

Time complexity of Sorting Algorithms

The following table covers the space complexity and time complexity in Big-O notation for different sorting algorithms commonly used in Computer Science.

Table 10.1: Space complexity and time complexity of different sorting algorithms

Algorithm	Time Complexity			Stable	Space Complexity
	Best Case	Average Case	Worst Case		Worst Case
Refined Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	$O(1)$
Shell Sort	$O(n)$	$O(n^{1.5})$ or	$O(n \log^2 n)$	No	$O(1)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	No	$O(n+k)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	Yes	$O(n+k)$

Summary

- Linear or sequential search is the simplest searching technique.
- Binary search finding the position of a particular value within the sorted list.
- A sorting algorithm is said to be stable, if two elements that are equal remain in the same relative order even after performing sorting.
- A sorting algorithm is in-place only if a constant amount of data elements of an input array is never stored outside the array.
- The lower bound of the comparison-based sorting algorithms is $\Omega(n \log n)$.
- Bubble sort is a simplest sorting technique that compares each pair of adjacent items and interchanging them if they are in the reverse order.
- Quicksort and Mergesort algorithm are based on recursive divide-and-conquer technique.
- Radix sort is an example of non-comparison based sorting.

Exercises

1. Write a Radix sort algorithm. Radix Sort the following list:
189, 205, 986, 421, 97, 192, 535, 839, 562, 674
2. Describe Radix sort with an algorithm. Show that it works in linear time.
3. Write the insertion sort algorithm.
4. What do you mean by external sorting? How does it differ from internal sorting?
5. Write an algorithm for sorting a list of numbers in ascending order using selection sort technique.
6. Compare the complexity of insertion sort and selection sort. Find the time complexity of the quicksort algorithm.
7. Compare the time complexity between Quicksort and Bubble sort algorithm.
8. What is the advantage of binary search over linear search?

9. Write down the algorithm for Merge sort and show the operation of Merge sort with a suitable sample data. Show that the running time for the Merge sort algorithm is $O(n \log_2 n)$
10. Explain with a suitable example, the principle operation of Heapsort.
11. Write short notes on the following
 - a) Radix sort
12. Choose the correct alternatives for the following:
 - i) A sort, which compares adjacent elements in a list and switches where necessary is
 - a) Insertion sort
 - b) Heapsort
 - c) Quicksort
 - d) Bubble sort
 - ii) The best-case time complexity of insertion sort is
 - a) $O(n^2)$
 - b) $O(n \log_2 n)$
 - c) $O(n^3)$
 - d) $O(n)$
 - iii) A machine needs a minimum of 100 sec. to sort 1000 names by quick sort. The minimum time needed to sort 100 names will be approximately
 - a) 72.7 sec.
 - b) 11.2 sec.
 - c) 50.2 sec.
 - d) 6.7 sec.
 - iv) What will be the time complexity for selection sort to sort an array of n elements?
 - a) $O(\log n)$
 - b) $O(n \log n)$
 - c) $O(n)$
 - d) $O(n^2)$
 - v) Priority queue can be implemented using
 - a) Array
 - b) Linked list
 - c) Heap
 - d) All of these
 - vi) In quick sort, a best desirable choice of pivot for partitioning the list will be
 - a) First element of the list
 - b) The last element of the list
 - c) Median of the list
 - d) A randomly chosen element of the list
 - vii) Stability of sorting algorithm is important for
 - a) Sorting records on the basis for analysis
 - b) Worst-case performance of sorting algorithm
 - c) Sorting alphanumeric because they are likely to be the same
 - d) None of these
 - viii) The drawbacks of the binary tree sort are remedied by the
 - a) Linear sort
 - b) Quick sort
 - c) Heap sort
 - d) Insertion sort
 - ix) Sorting is useful for
 - a) Report generation
 - b) Minimizing the storage needed
 - c) Making searching easier and efficient
 - d) Responding to queries easily
 - x) Which of the following sorting method will be the best if number of swapping done, is the measure of efficiency?
 - a) Bubble sort
 - b) Selection sort
 - c) Insertion sort
 - d) Merge sort
 - xi) The order of the binary search algorithm is
 - a) n
 - b) n^2
 - c) $n \log n$
 - d) $\log n$

CHAPTER 11

HASHING

“Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.” -Isaac Asimov

In the previous chapters, we have talked about different data storing and data accessing technique in the data structure. Irrespective of linear and non-linear data structure, the data are stored sequentially and searching of data is done in a sequential order. However, in some cases when the amount of data, then too large parallels accessing of memory space can increase the efficiency of the program. This can be achieved by Hashing.

Hashing or hash addressing is the process of mapping a large amount of data items to a smaller table with the help of a hash function. Hashing is a searching technique, which is essentially independent of the number of elements.

A hash table or Hashmap is a data structure, which uses a hash function to generate a key corresponding to the associated value. The advantage of this searching method is its efficiency to handle a large amount of data item in a given collection.

The two primary problems associated with the creation of hash tables are:

- i) The efficient hash function is designed so that it distributes the index values of inserting objects uniformly across the table.
- ii) The efficient collision resolution algorithm is designed so that it computes an alternative index for an object whose hash index corresponds to an object previously inserted into the hash table.

Hash Functions

A hash function is a function or process that map variable length data to fixed size data. The value generated by the hash function is called hash value, hash codes, hash sum or simply hashes. A hash function is used to speed up table lookup or data comparison task such as searching for an item in a database, find redundant data in a database or large files.

Definition: The fixed process to convert a key to a hash value is known as the hash function.

Hashing function or hash function maps the key with the corresponding key address or location. It provides the key to address transformation. Hash function (H) is used by a hash table to compute a set (L) of a memory address from the set K of the key. Thus, hash function denoted as: $H : K \rightarrow L$

The mapping between the key value and the location is not easy to maintain. The hash function will generally map several different keys to the same index. If the desired record is in the location given by the index, then there will not be any problem. However, if the record is not found, then either another hash function or same hash function is used.

KEY FEATURES

 Hash Functions

 Linear Probing

 Quadratic Probing

 Double Hashing

 Separate Chaining

Choice of Hash Function

The choice of the hash function is a very difficult job in hashing. It is not possible for selecting a perfect hash function, which is suitable for all different kinds of problem. There may be three possible ways to choose a hash function.

- Perfect Hash Function: There is no feasibility of this type of hash function if the data is large because practically it is not possible for huge data.
- Desirable Hash Function: For these hash function the address space should be small and collision should be kept very less or minimum.

Criteria of Hash Function

The two main criteria used in selecting a hash function are:

- Function H should be very easy and quick to compute.
- The function H should as far as possible uniformly distribute the hash address throughout the set L. So, there is a minimum number of the collision.

Typical Hash functions are as follows:

Division method

Division method is the simplest and most common way of implementing a hash function for a large volume of data. The aim of this function is to compute the index of a hash table.

The hash function H is defined by

$$H(K) = K \bmod m, \quad [\text{generates index value from 0 to } m-1]$$

or

$$H(K) = (K \bmod m) + 1, \quad [\text{generates index value from 1 to } m]$$

The number m is usually chosen to be a prime number (since this frequently minimizes the number of collisions). Here, the first function denotes the remainder when k is divided by m. The second function is used if the hash address ranges from 1 to m rather than from 0 to m-1.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100. Select M = 97 is a prime number

$$H(3205) = 3205 \bmod 97 = 4$$

$$H(7148) = 7148 \bmod 97 = 67$$

$$H(2345) = 2345 \bmod 97 = 17$$

Mid squares method

In mid square method, the key is multiplied by itself (i.e. k^2) and select a number of digits from the middle of the result. How many digits you select will depend on your table size and the size of your hash key. If the square is considered as the decimal number, the table size must be a power of 10. The mid square method squares the key value and then takes out the middle digits of the square result.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100.

K:	3205	7148	2345
K^2 :	10272025	51093904	5499025
H(K):	72	93	99

Folding method

The key K is partitioned into a number of parts, K_1, K_2, \dots, K_r , where each part except possibly the last has the same number of digits as the required address. Then the parts are added together, ignoring the last carry.

$$H(K) = K_1 + K_2 + \dots + K_r$$

Folding method can be classified into three types.

- Pure Fold method
- Fold Shifting method
- Fold Boundary

Pure Fold method: In this technique, parts of numbers are added together without any change.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100.

$$H(3205) = 32 + 05 = 37$$

$$H(7148) = 71 + 48 = 19 \quad [\text{ignoring the last carry}]$$

$$H(2345) = 23 + 45 = 68$$

Fold Shifting method: In this method the alternate parts are reversed before addition.

Example:

Suppose key is 1522756

$$H(1522756) = 01 \ 52 \ 27 \ 56 \ (\text{as the number contains 7 digits, so 0 is padded with left most digit})$$

$$H(1522756) = 10 + 52 + 72 + 56 = 36 \quad [\text{ignoring the last carry}]$$

Fold Boundary: Here, the first and last part of the number are reversed before addition.

Example:

Suppose key is 1522756

$$H(1522756) = 01 \ 52 \ 27 \ 56 \ (\text{as the number contains 7 digits, so 0 is padded with left most digit})$$

$$H(1522756) = 10 + 52 + 27 + 65 = 54 \quad [\text{ignoring the last carry}]$$

Operations on Hash Table

The hash functions used to perform different operations on hash tables.

Table 11.1 Operations on Hash table

Operation	Description
Insertion	This operation inserts a key in the Hash Table.
Deletion	This operation removes a key from the Hash Table.
Searching	This operation searches a desired key value within the Hash Table.

Collision Resolutions

Sometimes a hash function H may produce the same values rather than distinct values; it is possible that two different keys K_1 and K_2 will produce the same hash address. This situation is called **Collision**. Some methods must be used to resolve it.

There are two broad ways of collision resolution technique.

- i) Open Addressing /Closed Hashing (in array based implementation)
- ii) Separate Chaining / Open Hashing (in array of linked list implementation)

There are many ways of Open Addressing/ Close Hashing

- Linear Probing
- Quadratic Probing
- Double Hashing or Rehashing

Linear Probing

Linear probing is a collision resolution technique that used to resolve the collision by sequentially searching the hash table for a first available location. We assume that the hash table T with m location is circular so that $T(0)$ comes after $T(m - 1)$.

For insertion, a new record with key k is to be added to the hash table T, with a memory location $H(K) = h$, if an empty slot is available, otherwise try with $h+1, h+2, \dots$ etc. for first available location.

For searching a record R with key K, find out memory location $H(k) = h$, and then searches the table T by linearly searching the location, which $h+1, h+2, \dots$ until finding R or meeting an empty location, which indicates an unsuccessful search.

Insertion in a Hash Table

In the linear probing, key will be inserted into the hash table if an empty slot is available. When a collision occurs, the key will be stored in the next available slot in the hash table, assuming that the table is not already full. This is done by linear search (i.e. Linear probe) for an empty slot from the collision slot. When end of table reached during linear search, the search will wrap around to the beginning of the table and continue from there. If an empty slot is not found, then the table is full.

Example:

Suppose input keys in hash table are $\{65, 12, 27, 38, 49, 80, 10, 35, 97\}$ and hash table size is 10, where, hash key = key mod table size

	Index	Value
65 mod 10 = 5	[0]	80
12 mod 10 = 2	[1]	
27 mod 10 = 7	[2]	12
38 mod 10 = 8	[3]	
49 mod 10 = 9	[4]	
80 mod 10 = 0	[5]	65
	[6]	
	[7]	27
	[8]	38
	[9]	49

Figure 11.1a: Linear Probing without collision

Add remaining keys 10, 35, 97 to the previous hash table:
 $10 \bmod 10 = 0$, collision occurs, inserted in the next slot.
 $35 \bmod 10 = 5$, collision occur, inserted in the next slot

Index	Value
[0]	80
[1]	10
[2]	12
[3]	97
[4]	
[5]	65
[6]	35
[7]	27
[8]	38
[9]	49

Figure 11.1b: Linear Probing Showing Collision

Following algorithm describes the process of insertion of key in a hashtable using a Linear Probing collision resolution technique. The Flag array is used to indicate whether HashTable consist of key value at HashIndex. If HashTable has key at HashIndex then HashIndex of flag array is set to true else it is set to false. Initially flag array is initialised with false.

Algorithm of Insertion in Linear Probing

Algorithm: `INSERT_LINEAR_PROBING (HashTable, Key, Size)`

[HashTable is an array represents a hash table, Key is a value to be inserted as key and Size is the size of the HashTable]

1. Set HashIndex = Key mod Size
2. Repeat while HashTable[hashIndex] ≠ NULL
3. If HashTable[hashIndex] = NULL then
 - i) Set HashTable[hashIndex] = Key
 - ii) Set Flag[HashIndex] = true
 - iii) Return
- Else
 - Set HashIndex = (HashIndex + 1) mod Size
- [End of If]
- [End of while]
4. Return

Searching in Hash Table

In linear probing, searching a key in the hashtable with the slot at location H (K) = h, and continue linear search the adjacent slots in the table with $h+1, h+2, \dots$ until finding either an empty slot, which indicates an unsuccessful search or finding a slot whose stored key.

Following algorithm describes the process of searching a key in a hash table using a Linear Probing technique.

Algorithm of Searching with Linear Probing

```
Algorithm: SEARCH_PROBING (HashTable, Key, Size)
[HashTable is an array represents a hashtable, Key is a searching key and Size is the size of the HashTable]
1. Set HashIndex = Key mod Size
2. Set count=0
3. Repeat while HashTable[hashIndex] ≠ Key and count<size
    Set hashIndex = (hashIndex + 1) mod Size
    count = count+1
    [End of While]
4. If HashTable[hashIndex] = Key then
    Print "FOUND"
    Else
    Print "NOT FOUND"
    [End If]
5. Return
```

Deletion in a Hash Table

In the following algorithm, the process of deleting a key in a hashtable using a Linear Probing technique is described. Primarily searching a key in the hashtable with the slot at location $H(K) = h$, and continue linear search the adjacent slots in the table with $h+1, h+2, \dots$ until finding either an empty slot, which indicates an unsuccessful search or finding a slot whose stored key. When the key is found, then that location is set to NULL and flag at that location is set to false.

Algorithm of Deletion in Linear Probing

```
Algorithm: DELETE_PROBING (HashTable, Key, Size)
[HashTable is an array represents a hash table, Key is value to be deleted from HashTable and Size is the size of the HashTable]
1. Set HashIndex = Key mod Size
2. Set count = 0
3. Repeat while HashTable[HashIndex] ≠ Key and count<size
    Set HashIndex=(HashIndex + 1) mod Size
    count = count + 1
    [End of while]
4. If HashTable[HashIndex] = Key then
    i) Set HashTable[HashIndex] = NULL
    ii) Set Flag[HashIndex] = false
    Else
    Print "NOT FOUND"
    [End If]
5. Return
```

The main drawback of linear probing is that records tend to form primary cluster, that is a contiguous block of items and when a new key hashes in the cluster, then the cluster size increases and

the cluster need several attempts to resolve the collision. As a result increases the average search time for a record. Insertion and searching time depend on the length of the cluster.

An example of primary clustering:

Example:

Suppose input keys {89, 18, 49, 58, 69} and table size is 10. Using Linear probing keys are inserted in the index shown in the following table.

Index	Value
[0]	49
[1]	58
[2]	69
[3]	
[4]	
[5]	
[6]	
[7]	
[8]	18
[9]	89

Figure 11.2: Primary Clustering in Linear Probing

To overcome this limitation of linear probing another collision resolution technique is used which is known as Quadratic Probing.

Quadratic Probing

Suppose a record R with key K has the hash address H (K) = h. Then, instead of searching the location with address h, h+1, h+2...we search the locations by address

$$h, h + 1, h + 4, h + 9, h + 16, \dots, h + i^2, \dots$$

If the number m of location in the table T is a prime number, then the above sequence will access half of the locations in T. Quadratic probing is a more efficient algorithm in a closed hash table, since it better avoids the primary clustering problem that can occur linear probing.

With quadratic probing, there is no guarantee of finding an empty cell once the table gets more than half-full or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve the collision.

Quadratic probing leads to secondary clustering, when two keys do only have the same collision chain if their initial position is the same. As it turns out, secondary clustering prevents us from guaranteeing an insert if the table is greater than half full.

Secondary clustering is less severe in terms of performance hit than primary clustering. It is a process to keep clusters formation by using Quadratic Probing. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

Example:

Suppose input keys are {89, 18, 49, 58, 69}, hash table size is 10, where, hashkey = key mod table-size

	Index	Value
89 % 10 = 9	[0]	49
18 % 10 = 8	[1]	
49 % 10 = 9, 1 attempt needed $1^2 = 1$ spot movement is required	[2]	58
58 % 10 = 8, 2 attempt needed $2^2 = 4$ spots movement are required	[3]	69
69 % 10 = 9, 2 attempt needed $2^2 = 4$ spots movement are required	[4]	
	[5]	
	[6]	
	[7]	
	[8]	18
	[9]	89

Figure 11.3: Linear Probing with cluster

Using quadratic probing there is a probability of secondary clustering when more than half of the hash table is filled up. That means elements that hash to the same hash key will always probe the same alternative cells.

Searching a key in a Hash Table using Quadratic Probing

Following algorithm describes the process of inserting a key in a hash table using a Quadratic Probing technique.

Algorithm of Searching in Quadratic Probing

Algorithm: **INSERT_QUAD_PROB(HashTable, key, size)**

[HashTable is an array represents a hash table, Key is a searching key and Size is the size of the HashTable]

1. Set HashIndex = Key mod size
2. If HashTable[HashIndex] = NULL then
 - a) Set HashTable[HashIndex] = Key
 - b) Set flag[HashIndex] = 1
 - c) Return
- else
 - a) Set count = 0
 - b) Set index = HashIndex
 - c) Repeat while HashTable[HashIndex] ≠ NULL
 - Set count = count + 1
 - Set HashIndex = HashIndex + count * count

```

        Set HashIndex = HashIndex mod size
        [End of Loop]
d) If HashTable[HashIndex] = NULL then
    Set HashTable[HashIndex] = Key
    Set flag[HashIndex] = 1
    [End of If]
[End of IF]
3. Return

```

Double Hashing

Double hashing is a popular collision resolution technique in open addressed hash table. It uses two hash functions:

- i) Like linear probing First hash function H is used as a starting address and
- ii) Second hash function H' is used to find out the interval to skip a variable amount. Suppose a record R with key K has the hash address $H(K) = h$ and $H'(K) = h' \neq m$, then we search the location with the address
 $h, h + h', h + 2h', h + 3h', \dots$

If m is a prime number, then the above sequence will access all the location in the table T .

Second hash function in double hashing has the following couple of requirements

- It must never evaluate to 0.
- Must make sure that all cells can be probed.

A popular hash function for double hashing is $\text{Hash2(key)} = R - (\text{key \% } R)$ where R is a prime number that is smaller than the size of the table.

Example:

Suppose input keys $\{89, 18, 49, 58, 69\}$ and table size:10

Largest prime number less than 10 is 7
 $\text{Hash1(key)} = \text{key \% } 10$ and $\text{Hash2(key)} = 7 - (\text{key \% } 7)$
 $\text{Hash1}(89) = 89 \% 10 = 9$
 $\text{Hash1}(18) = 18 \% 10 = 8$
 $\text{Hash1}(49) = 49 \% 10 = 9$, collision occur
 $\text{Hash2}(49) = 7 - (49 \% 7)$
 $= 7$ position move from [9]
 $\text{Hash1}(58) = 58 \% 10 = 8$, collision occur
 $\text{Hash2}(58) = 7 - (58 \% 7)$
 $= 5$ position move from [8]
 $\text{Hash1}(69) = 69 \% 10 = 9$, collision occur
 $\text{Hash2}(69) = 7 - (69 \% 7)$
 $= 1$ position move from [9]

Index	Value
[0]	69
[1]	
[2]	
[3]	58
[4]	
[5]	
[6]	49
[7]	
[8]	18
[9]	89

Figure 11.4: Double Hashing

Advantage:

- It effectively eliminates clustering.
- It can allow the table to become nearly full.

Disadvantage

The main drawback of this method is in the implementation of deletion.

Insertion a key in a Hash Table using Double Probing

Following algorithm describes the process of inserting a key in a hash table using a Double Probing collision resolution technique.

Algorithm of Insertion in Double Probing

Algorithm: `INSERT_DOUBLE_PROB(HashTable, Key, Size)`

[*HashTable* is an array represents a hash table, *Key* is a searching key and *Size* is the size of the *HashTable*]

1. Set *HashIndex* = *Key* mod *size*
Set *Flag*[*HashIndex*] = false
2. If *HashTable*[*HashIndex*] = NULL then
 - a) Set *HashTable*[*HashIndex*] = *Key*
 - b) Set *Flag*[*HashIndex*] = true
else
 - i) Set *move* = prime - (*Key* mod prime)
 - ii) set *count* = 1
 - iii) Repeat while *HashTable*[*HashIndex*] ≠ NULL
 - a) Set *HashIndex* = *HashIndex* + *count* * *move*
 - b) Set *HashIndex* = *HashIndex* mod *size*
 - c) set *count*=*count*+1
 - [End of Loop]
 - iv) Set *HashTable*[*HashIndex*] = *Key*
 - v) Set *Flag*[*HashIndex*] = true
 - [End of IF]
3. Return

Separate Chaining

In this technique, each bucket is independent and has the same sort of list of entries with the same key value. The time for hash table operation is the time to find the bucket, in addition, the time for the list operation.

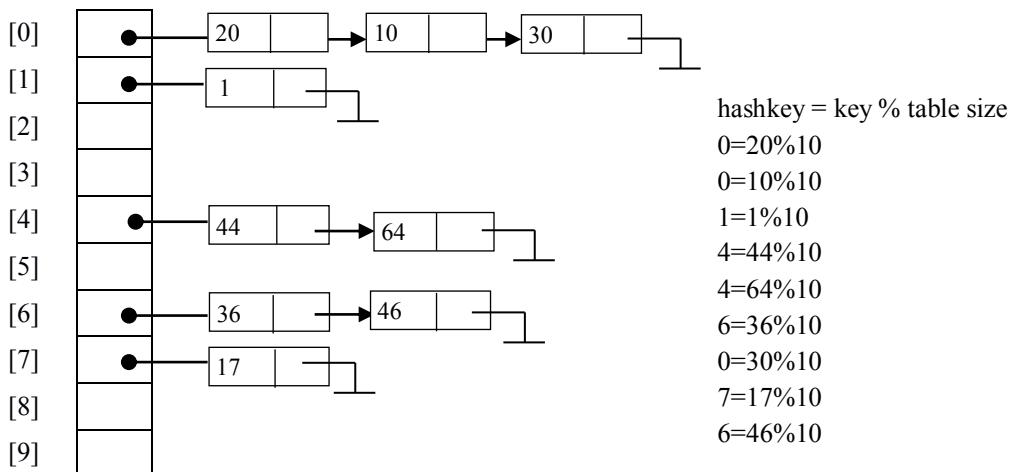
Suppose, the hash table contains *m* slots. Each slot in the hash table contains a pointer to a linked list (separate chain) and the list stores the elements hashed to that slot. Hence, this method is known as Separate Chaining. A new key can be placed anywhere within a chain, as no ordering among the keys is used. As such, the new node is inserted at the end of a chain.

The hash table containing *m* pointers can be defined as follows:

```
node *Hash[m];
```

Example:

Suppose input keys are {20, 10, 1, 44, 64, 36, 30, 17, 46} and hash function $h(K) = K \bmod 10$.

**Figure 11.5:** Separate Chaining

The main drawback to chaining is that one needs more memory spaces for the link field and there are m memory slots for the pointer array. An additional disadvantage is that traversing a linked list has poor cache representation.

- Cost is proportional to the length of the list: Cost in terms of time complexity is proportional to the length of the list.
- Worst case: If the hash function is not efficient then it generates same hash key that increases the length of one list.
- M too large: if the value of m is too large then the number of chains becomes a more and more chain will be empty.
- M too small: If the value of m is too small, that increase the length of the list.

Table 11.2: Type of Collision Resolution Technique

Feature	Linear Probing	Quadratic Probing	Double Hashing
Efficiency	Fastest among three.	Easy to implement and deploy	Make more efficient use of memory
Probe use	Use few probes.	Uses extra memory for links and it does not probe all locations in the table.	Use few probes, but take more time.
Demerit	Primary clustering is a limitation.	Secondary clustering is the limitation.	More complicated to implement.
Computing Interval	The interval between probes is fixed and it is 1.	The interval between probes increases proportionally to the hash value.	The interval between probes is computed using the second hash function.

Problems for which Hash Tables are not suitable are:

- The problem for which data ordering is required.
- Problems having multi-dimensional data.
- Prefix searching if keys are long and of variable length.
- The problem that has dynamic data.
- The problem in which data does not have any unique key.

Advantages of Hash Table

- It can efficiently handle a large volume of data.
- The speed of searching a data item increased as parallel searching can be applied here.
- Look up cost may be reduced by proper hash function, hash table size and internal data structure.

Disadvantages of Hash Table

- Hash table implementation is more difficult than search tree.
- Hash table does not allow duplicate keys.
- In some cases, the cost of a good hash function is more than that of the search tree.
- For a dynamic hash table cost of insertion and searching of the element in the hash table is more.
- If collision occurs very frequently in the hash table then the efficiency of hash table degrades.

Load Factor

The performance of hash table depends on two factors, initial capacity and load factor. Initial capacity is the number of cells in the hash table. A Load Factor (α) of a hash table is defined as the ratio of the number of elements (n) in the hash table to the table size (tsize).

$$\alpha = n/tsize$$

Load Factor is a measure which decides when exactly to increase the hash table capacity, so that get and put operation can still have O (1) complexity.

For linear probing α tends to 1 ($0 \leq \alpha \leq 1$), that implies collision probability is higher in linear probing. Load Factor tends to zero (0) implies that the proportion of unused space on hash table is increasing. However, there is no necessary reduction in search cost. The result is wasted memory.

Generally, the default load factor (0.75) indicates better performance for a hash table. The expected number of entries in the hash table and its' load factor should be considered when the initial capacity of the hash table is defined. If the initial capacity is more than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Summary

- Hashing is one of the most suitable techniques for storing and retrieving data when data volume is more than the storage volume.
- The choice of best hash function depends on the application.
- Collision resolution is one of the key feature of hashing process.

Exercises

1. What do you mean by hashing? What are the applications where you will prefer hash tables to other data structures?
2. Why do the hash functions need to be simple?

3. What do you mean by a collision? How is it handled? Discuss two collision resolution techniques and compare their performance.
4. What is the primary advantage of hashing over deterministic search algorithms?
5. Why is hashing referred as a heuristic search method?
6. Explain with a suitable example the collision resolution scheme using linear probing with open addressing.
7. Choose the correct alternatives for the following:
 - i) The ratio of items present in the hash table, to the total table size, is called
 - a) Balanced factor
 - b) Load factor
 - c) Item factor
 - d) None of these
 - ii) What causes a collision?
 - a) The program you are running crashes
 - b) There are too many hash keys in the array
 - c) Two hash keys are the same
 - d) The program is out of the memo
 - iii) What does hashing do?
 - a) Delete old files
 - b) Create new files
 - c) Improve the air quality of the room
 - d) None of the above
 - iv) How is a hash key computed?
 - a) Long division
 - b) Subtraction
 - c) Random number generation
 - d) Modulo division
 - v) What can be done to compute the hash key value of a string?
 - a) Convert them all to their ASCII values
 - b) Generate random numbers for the letters every time
 - c) a and b only
 - d) Give them each a value according to their place in the alphabet
 - vi) What is a restriction of the regular 'Direct Address Tables'?
 - a) The range of the key must be severely bounded
 - b) It takes up too much memory on the hard drive
 - c) The range of the key is unlimited.
 - d) It is far too slow
 - vii) What is the hash key value of 'Donaldson' if you assign each letter its corresponding number in the alphabet (i.e. f=6) and if you use 9 as the divisor?
 - a) 12
 - b) 3
 - c) 7
 - d) 9
 - viii) The goal of hashing is to produce a search that takes
 - a) O(1) time
 - b) O(n^2) time
 - c) O(logn) time
 - d) O(nlogn) time

CHAPTER 12

FILE STRUCTURE

"The most beautiful thing we can experience is the mysterious. It is the source of all true art and science." -Albert Einstein

In computing, a file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol.

ELEMENTS OF FILE SYSTEM

There are four building blocks of the file system.

- **Field:** Basic element of data contains a single value fixed or variable length
- **Record:** A collection of related fields that can be treated as a unit by some application program fixed or variable length.
- **File:** a collection of similar records treated as a single entity may be referenced by name access control restrictions usually apply at the file level.
- **Database:** It is a collection of related data. The relationships among elements of data are explicit. It is designed for use by a number of different applications consists of one or more types of files.

KEY FEATURES

-
- File system
 - Elements of File System
 - Software Architecture of file system
 - Objective of file management system
 - Category of file Organization
-

Needs of File Management System

File Management System is essential for managing the data in a file. An optimized file management system is required to manipulate the data in the file. It provides support to a variety of storage device type. It also provides a set of I/O interfaces routine to user processes.

File System Architecture

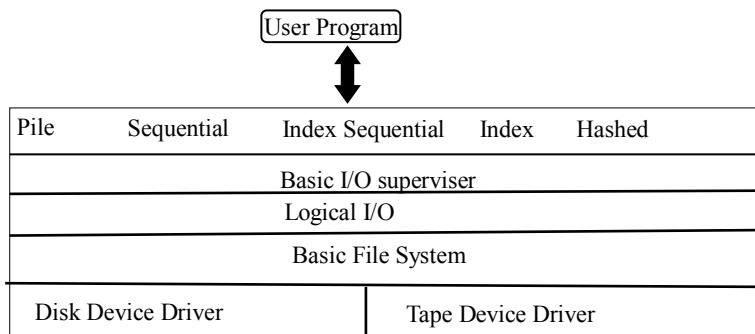


Figure 12.1: Architecture of File system

Device Driver: It is the lowest level component of the file system that directly communicates with hardware devices. It is responsible for I/O operation on a device. It provides an interface for the I/O peripherals to communicate with the operating system of the computing device.

Basic File System: This component of the file system is considered as the interface between the memory device and the environment outside. The access methodology of secondary memory is defined by this interface. The memory block size, placement of memory block in secondary memory is also described by the basic file system. Buffering in main or primary memory is also the task of this file system.

Basic I/O Supervisor: This component is responsible for all I/O initiation and termination process. It controls the scheduling disk and tape access to optimize performance. I/O buffers are assigned and secondary memory is allocated by this I/O Supervisor.

Logical I/O: It presents a logical view of devices. Details of hardware devices and error handling are made hidden from the application. It optimise the use of I/O devices and CPU. Sharing and scheduling of devices are also done by logical I/O component of the file system.

File Organization Type

The basic file operations that are performed on files are

- a) Insertion of record
- b) Deletion of records
- c) Manipulation of records
- d) Retrieval of records

File organization is defined as the activity of planning data records onto a storage medium in such a way as to facilitate ease of access to the stored data records. The effective file organization is required to reduce file access time for needed information, to store a large quantity of data records in the minimum space possible, to speed up computerized information storage and retrieval process, and to handle several sets of files using a hierarchy of storage device. Selection of the type of file organization is to be chosen depends upon several factors:

- Type of application
- The method of processing for updating files
- The size of the file
- The inquiry capabilities
- The volatility
- Response time

Category of file organization

There are many ways of organizing the records in a file. Depending on the process of records are organized the file organization is categorized into various ways. The following figure describes the taxonomy of file organization:

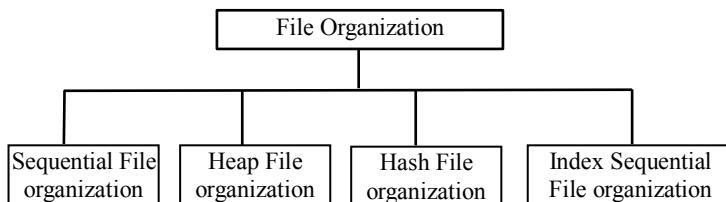


Figure 12.2: Taxonomy of File Organization

Sequential File Organization

In a sequential file organization the records are stored by maintaining an order either descending or ascending order determined by the key field. Sequentially organized files that are processed by the computer system are normally stored on storage media such as magnetic tape, punched paper tape, punched cards or magnetic disks. To access these records, the computer must read the file in sequence from the beginning. During searching of a record from a file, each record is examined sequentially until and unless the desired key field is matched. This method is suitable for taking backup for a database file where all the records should be processed at one time.

Advantages

- Simple to understand approach.
- Easy to organize, maintain and understand.
- Loading a record requires only the record key.
- Efficient and economical if the activity rate, i.e. the preparation of file records to be processed is high.
- Relatively inexpensive I/O media and devices may be used.
- Errors in file remain localized.
- Files may be relatively easy to reconstruct since a good measure of built-in backup is usually available.

Disadvantages

- Random inquiries are usually impossible to handle.
- Data redundancy is typically high since the same data may be stored in several files sequenced in different ways.
- The entire file is needed to be processed even when it is not required.
- Increase time complexity during searching for a single record.

Heap File Organization

It is the simplest file organization where the records are stored without any order. As the records are inserted they are stored in that manner. A new record is inserted in the last page of the file; if there is insufficient space on the last page, a new page is added to the file. This makes insertion very efficient. However, as the heap file does not maintain any order so the records are retrieved using a linear search. A linear search involves reading pages from the file until the required is found. This makes retrievals from heap files that have more than a few pages relatively slow unless the retrieval involves a large proportion of the records in the file. During deletion of a record from a heap file, the required page first has to be retrieved, the record marked as deleted, and the page written back to disk. The space with deleted records is not reused. Consequently, performance progressively deteriorates as deletion occurs. This means that heap files have to be periodically reorganized by the Database Administrator (DBA) to reclaim the unused space of deleted records.

Heap files are one of the best organizations for bulk loading data into a table, as records are inserted at the end of the sequence; there is no overhead of calculating what page the record should go on.

Advantage

- It is useful when we deal with the bulk of data at a time.

- The relation is only a few pages long. In this case, the time to locate any tuple is Short, even if the entire relation has been searched serially.
- When every tuple in the relation has to be retrieved (in any order) every time the relation is accessed. For example, retrieve the name of all the students.

Disadvantages

- Heap file organization is not good when we deal with selected records of a file, this will increase processing time and thus degrade the performance.

Hash File organization

In hash file organization the records are stored in the file using a hash function. A hash function is defined over a key field and that field is known as Hash Field. Hash function generates a value which is mapped to the key and accordingly the record is placed in the file. This file organization is also termed as Direct or Random file organization. These types of files are used in a direct access storage device (DASD) like a floppy disk, computer disk, optical disk, etc. in such storage devices, the records are stored in random order that allows immediate access as direct access to the individual records. Commonly some arithmetic function is applied to the hash field so that the records will be evenly distributed throughout the file.

Advantages

- Access and retrieval of records become easy and fast.
- Transaction need not be sorted and placed in sequence prior to processing.
- Accumulation of transactions into batches is not required before processing them.
- Immediate updating of several files as a result of the single transaction is possible.

Disadvantages

- These files must be stored on a direct- access storage device. Hence, relatively expensive hardware and software resources are required.
- Special precaution should be taken during manipulation of records.
- Updating of records becomes difficult.
- Less efficient than sequential file organization.

Index Sequential File Organization

In this file organization the records are organized in a sequential manner. The difference with Sequential file organization is that here an index table is maintained for easy and fast access to the record. The records of the files can be stored in a random sequence but the index table is in a sorted sequence on the key value. The file management system simply accesses the data records in the order of the index value. These, indexed sequential files provide the user sequential access, even though the file management system is accessing the data records in a physically random order.

Index file contains two fields

- i) Index field
- ii) Pointer field

Depending upon these two fields, file indexing is categorized into following classes:

Primary Index

If the primary key of the table is used as an index field then the corresponding indexing is known as

primary indexing.

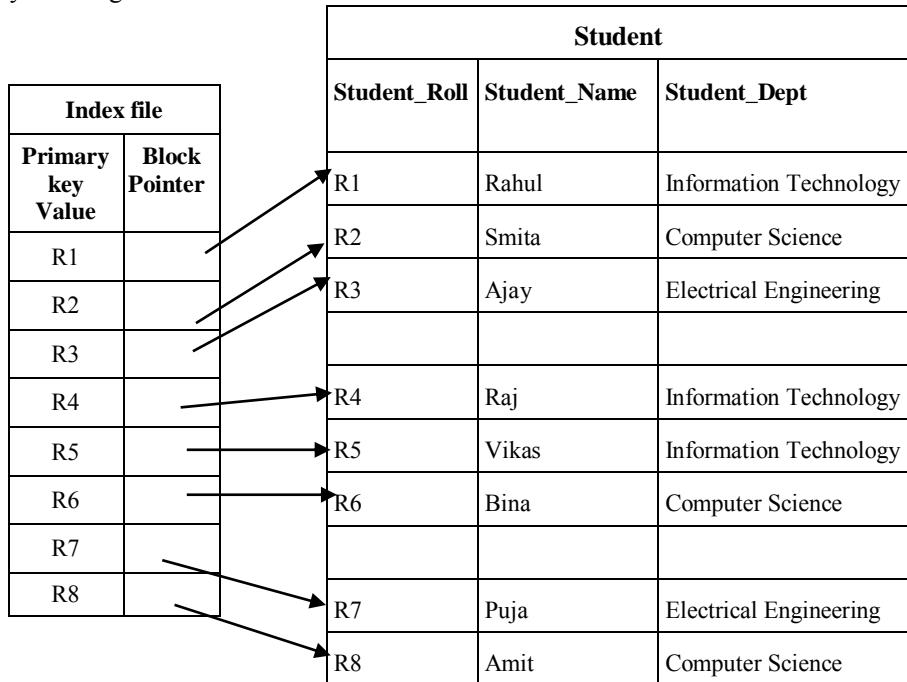


Figure 12.3: Dense Primary Index

In the figure 12.3, the dense primary index is explained. In dense primary index, all primary key values are present in the index file. Each record of relation is pointed by each block pointer.

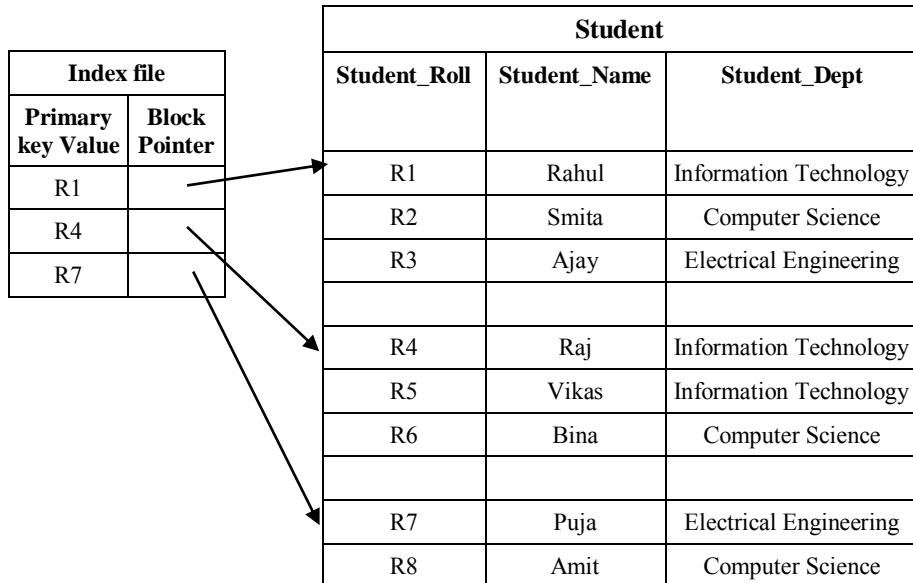


Figure 12.4: Sparse Primary Index

In the above example, primary indexing is explained. In the Student relation, Student_Roll is the primary key. It is used to identify a record in the relation uniquely. In the index file, the Student_Roll is used as a primary key value, block pointers are used to point the block that contains the record. The indexing that is used here is called Sparse Index, where each primary key is not present in the index file rather some blocks are created where the first record of each block is indexed by the block pointer of the index file.

Table 12.1: Comparison between Dense and Sparse Primary Index

Feature	Dense Primary Index	Sparse Primary Index
Indexing	Pointer field directly points to a record for a search key.	Pointer corresponds to the first record of each block.
Searching Time	Searching is fast.	Searching takes more time.
Space Allocation	Consumes greater space.	Consume less space.

Secondary Index

To assist primary key index, a secondary index is created to improve search performance. This secondary index is created to improve search performance. This secondary key is generally on a non-key attribute and may have duplicate values. Generally, a base table is associated with one primary index and several secondary indices.

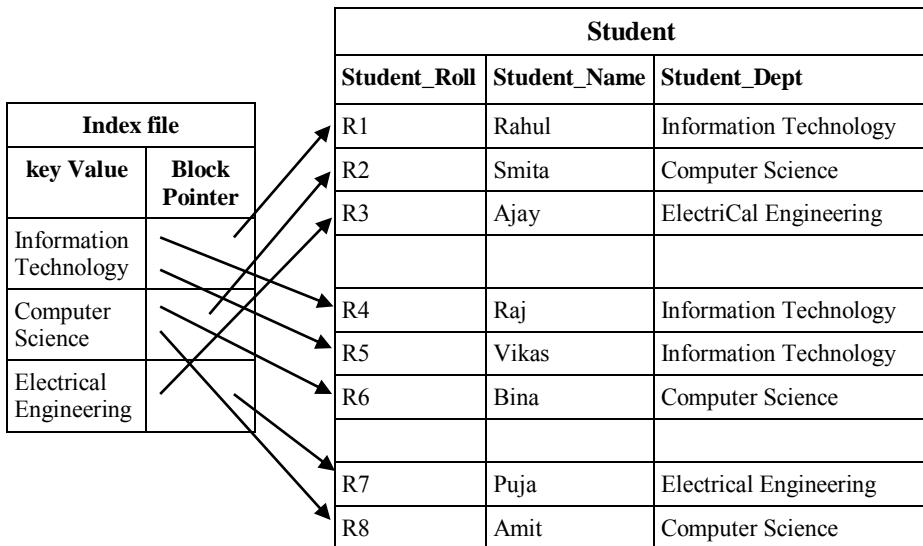


Figure 12.5: Secondary Index

In the above figure, secondary indexing is explained. Here, Student_Dept is considered as a key value, which is not the primary key. Depending on the key value the block pointer points to the corresponding records in the relation.

Clustering Index

When records of a file are physically ordered on a non-key attribute which does not have a unique value for each record, the ordering attribute is called clustering attribute. If the values of indexing attribute for all records are not unique, but ordered, al clustering index is used to facilitate the search option.

In the figure 12.6, clustering index is described. Where the primary Key value is not the primary key (Student_Roll) of the relation rather a non-key attribute, Student_Dept which has redundant value. Here the Student_Dept form cluster, where each cluster is pointed by the block pointer.

Index file	
key Value	Block Pointer
Information Technology	
Computer Science	
Electrical Engineering	

Student		
Student_Roll	Student_Name	Student_Dept
R1	Rahul	Information Technology
R4	Raj	Information Technology
R5	Vikas	Information Technology
R2	Smita	Computer Science
R6	Bina	Computer Science
R8	Amit	Computer Science
R7	Puja	Electrical Engineering
R3	Ajay	Electrical Engineering

Figure 12.6: Clustering Indexing

Multilevel Indexing

Initially, the content of index file are stored in many fashions but when the number of records increases obviously the size of index file also increases. Though the contents are stored in sorted fashion the searching time cannot be less than $\lceil \log_2 n \rceil$.

In order to reduce the searching time, tree structure will be the first choice but binary tree will not be the appropriate tree structure because it can not reduce searching time.

In order to reduce the height of the tree structure m-way tree like B-tree or B+ tree are selected where ' m ' depends on the size of the disk.

Here at first the root node block is accessed where pointers point to the inner block node. Generally, leaf node pointer points to a particular record or block where the desired record is placed.

B-tree

A B-tree of order m is an m way search tree where a node may have m children. A B-tree allows dynamic multilevel indexing. An index entry is nothing but a pair of key value and a pointer. In B-tree the leaf nodes are not connected that may increase the search time. This difficulty of B-tree is overcome in B+ tree.

B + Tree

A B+ tree is modified B-tree where searching time is decreased and searching performance is increased.

Following are the properties of B + tree:

- Every node should store $(m-1)$ key-pointer pair(k, P_b), value m should be odd.
- Each node should have my three pointers (P_T) to another node at the next level.

- Every non-terminal node except root node has at least $\lceil m/2 \rceil$ non null pointers. The root node has at least 2 non-null pointers.
- All leaf nodes are at the same level and the tree is balanced.
- If a node has $(m+1)$ children it must contain m key-pointers pairs.
- For each node key- value pair is less than next key-value pair.
- Each and every key- value pair of non-terminal node should be present at leaf level.
- All the leaf nodes must be linked with each other and form a linked structure at leaf level.

Insertion in a B+ Tree

Insertion operation into a B+ tree of order m is performed by searching the appropriate leaf node where we insert the key. There are three cases for insert a key into B+ tree.

Case 1: When the leaf node is not at full capacity (i.e. it has fewer than $m - 1$ keys), then simply insert it into proper position so that after insertion the keys remain seated. Therefore, it requires to movement keys within leaf node for the newly inserted key. Since m is assumed a constant, so the constant time overhead may be ignored.

Case 2: When the node is full (i.e. it has exactly $m - 1$ keys), then one key is moved to immediate upper level. The key at $\lceil m/2 \rceil^{\text{th}}$ position is moved upward. If m is even, then key at $(\lceil m/2 \rceil + 1)^{\text{th}}$ position can be moved upward. The rule of insertion should be same for inserting key elements at any level. When a key is moved then a copy of that key is stored at the leaf level. Intermediate nodes do not contain any duplicate values. Each leaf node makes link with succeeding leaf node to reduce searching time.

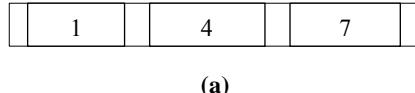
Example:

Insert the following keys in a B+ tree of order 4

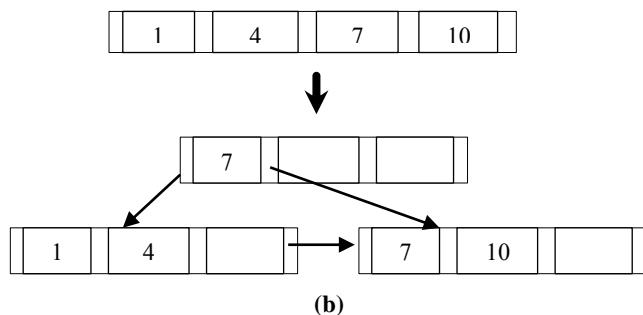
1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42

Since the order of the B + tree is 4, the node of this tree can hold maximum 4 children, which implies the key pointers for each node is 4 whereas the key value is 3.

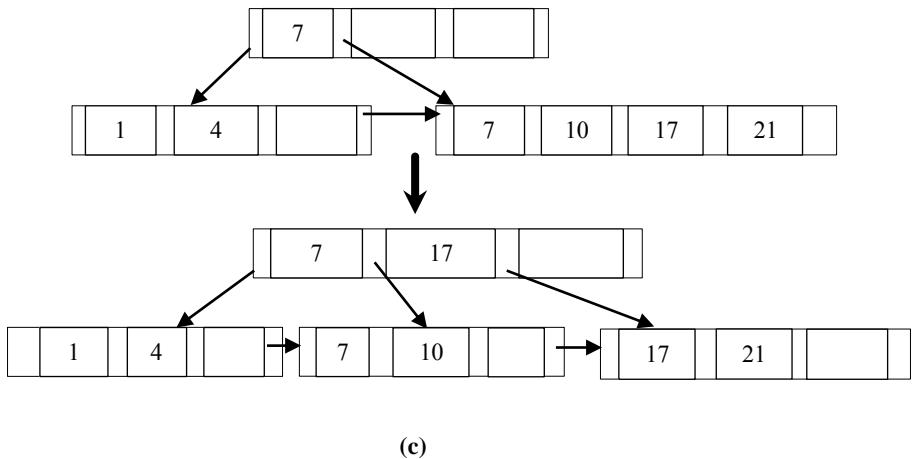
Insert 1, 4, 7



Insert 10

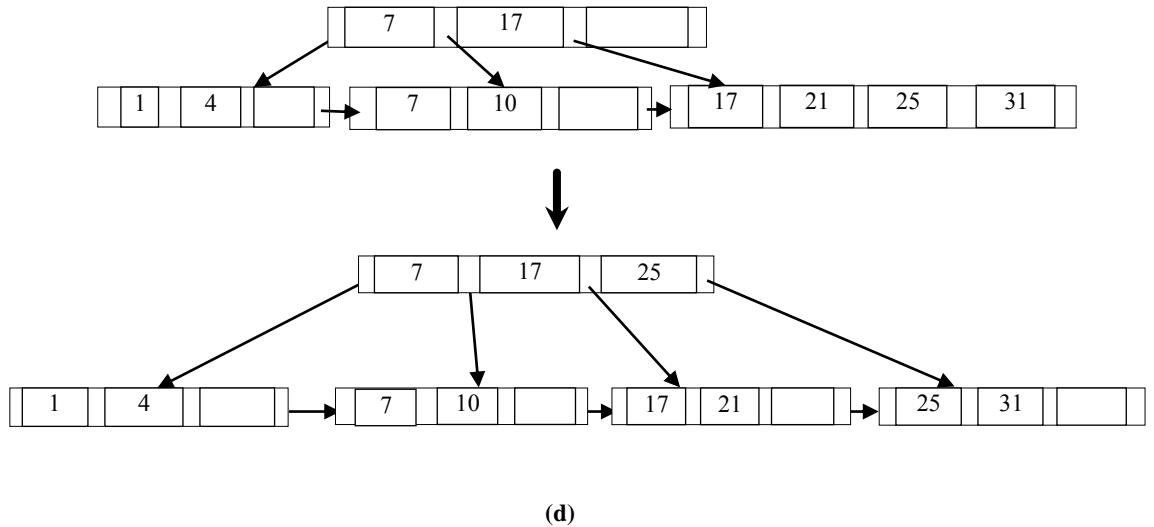


Insert 17, 21



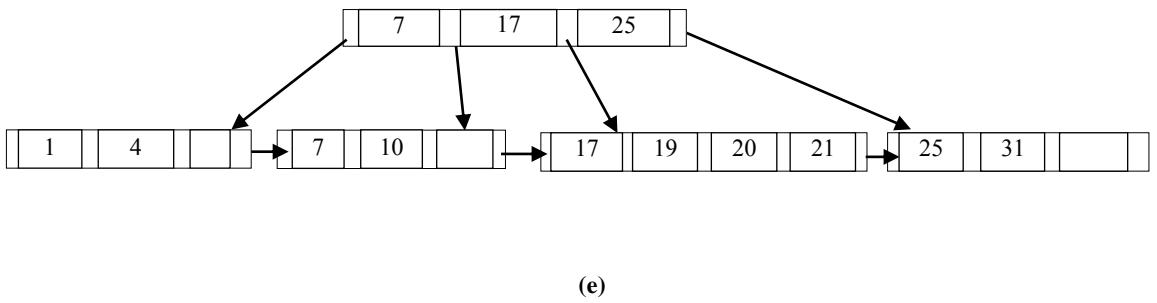
(c)

Insert 31, 25



(d)

Insert 19, 20



(e)

Insert 28, 42

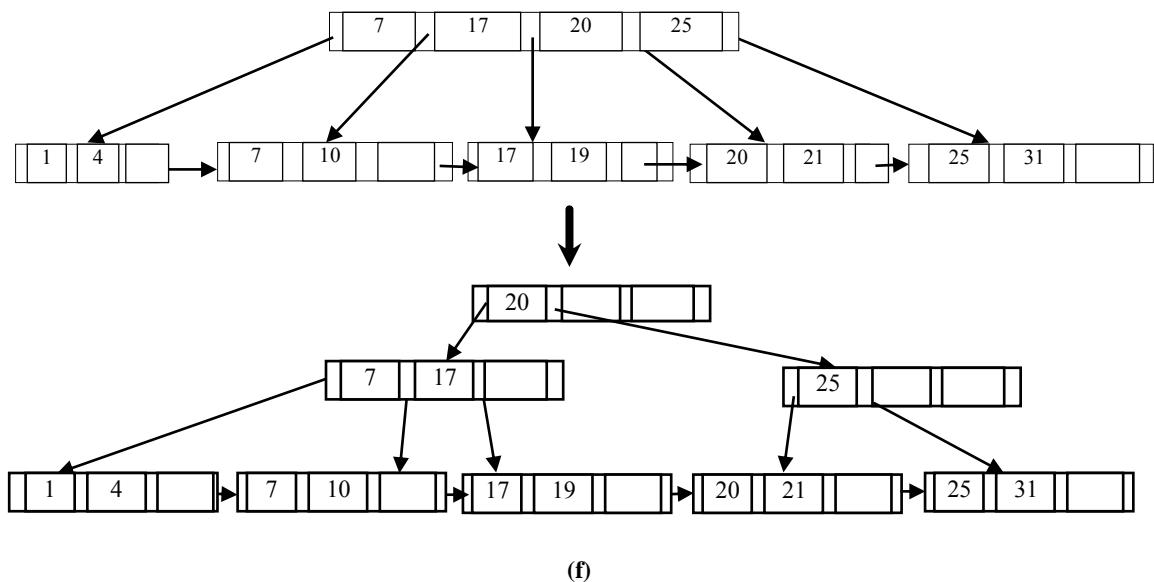


Figure 12.3 (a, b, c, d, e, f): B+ tree insertion

Summary

- The file management system is must for storing and retrieving data in an organized manner.
- Different file organization processes are used for different type of application.
- B+ tree is the most suitable indexed sequential file organization.

Exercises

1. What is the utility of file management system?
2. What is the relationship between file organization and file management system?
3. Compare different file organization method.
4. Compare and contrast among different indexed file organization.
5. Discuss file system architecture.
6. Draw a B+ tree of order 3 with the following key
January, March, December, February, April, May, July, August, November, June, September, October.
7. Choose the correct alternatives for the following:
 - i) Storage media that is operated directly from computer's central processing unit is considered as
 - a) Primary storage
 - b) Secondary storage
 - c) Tertiary storage
 - d) All of above
 - ii) Hashing technique which allocates a fixed number of buckets is classified as
 - a) Dynamic hashing
 - b) Static hashing
 - c) External hashing
 - d) Internal hashing
 - iii) Kind of allocation in which file blocks contain pointer to next blocks of file is classified as
 - a) Linked allocation
 - b) Indexed allocation
 - c) Header allocation
 - d) Contiguous allocation

- iv) Field on which equality condition is placed in hashing technique is called
 - a) Hash field
 - b) Cluster file
 - c) Spanned field
 - d) Sequential field
- v) Additional access path added into ordered file is called
 - a) Ternary index
 - b) Tertiary index
 - c) Primary index
 - d) secondary index
- vi) Type of organization in which records are inserted at end of stored file is classified as
 - a) Pile file
 - b) Linear search file
 - c) Relative file
 - d) External file
- vii) Kind of field with which record searching is done is classified as
 - a) Sorting field
 - b) Relative field
 - c) Linear field
 - d) Ordering field
- viii) File which consists of ordering fields for records is called
 - a) Sequential file
 - b) Ordered file
 - c) Spanned file
 - d) Both a and b
- ix) File which consists of ordering attribute which is non-key in nature called
 - a) Sequential file
 - b) Spanned file
 - c) Master file
 - d) Clustered file
- x) Type of allocation in which blocks of file are allocated to consecutive blocks of disks is classified as
 - a) Indexed allocation
 - b) Header allocation
 - c) Contiguous allocation
 - d) Linked allocation
- xi) The difference between linear array and a record is
 - a) An array is suitable for homogeneous data, but the data items in a record may have different data type.
 - b) In a record, there may not be a natural ordering in opposed to linear array.
 - c) A record form a hierarchical structure, but a linear array does not.
 - d) All of the above
- xii) Which one of the following is a key factor for preferring B-trees to binary search trees for indexing database relations?
 - a) Database relations have a large number of records
 - b) Database relations are sorted on the primary key
 - c) B-trees require less memory than binary search trees
 - d) Data transfer from disks is in blocks.
- xiii) B+ trees are preferred to binary trees in databases because
 - a) Disk capacities are greater than memory capacities
 - b) Disk access is much slower than memory access
 - c) Disk data transfer rates are much less than memory data transfer rates
 - d) Disks are more reliable than memory

Appendix A: Data Structure Operations

The following table covers the space complexity and time complexity in Big-O notation for different operations like access, search, insertion and deletion in different data structures commonly used in Computer Science.

Data Structure	Time Complexity								Space Complexity	
	Average Case				Worst Case					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)	O(n)	
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	
B-tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	
Hash Table		O(1)	O(1)	O(1)		O(n)	O(n)	O(n)	O(n)	

GLOSSARY

Abstract Data Type (ADT) describes the data objects, which constitute the data structure and the fundamental operations supported on them.

Algorithm a finite sequence of steps, each of which is elementary that must be followed to solve a problem.

Array a collection of the similar type of data items, which is stored in the consecutive memory locations under a common name.

AVL tree is a height-balanced binary tree if it is either empty or if T is a non-empty binary search tree with T_L and T_R as its left and right subtrees, if and only if i) T_L and T_R are height balanced and ii) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

Binary tree is a finite set of nodes, which is either empty or consists of a root and two disjoint binary trees called left subtree and the right subtree.

Complete binary tree is a binary tree in which all the levels are filled and the last level possibly is partially filled from left to right and some rightmost leaves may be missing.

Data is the basic entity or fact that is used in calculation or manipulation process.

Data type refers to the type of data that variables hold.

Double-ended Queue (Deque) is a linear list that generalizes a queue, for which elements can be inserted or deleted from either the front end or from the rear end but not in the middle. There are two variations of a Deque: Input-restricted Deque and Output-restricted Deque

Dynamic Data Structure is one kind of data structure, which can be extended during execution.

Dynamic memory allocation allocates storage locations at execution time from part of memory, known as the *heap*.

Expression tree is a binary tree, which is used to represent a mathematical expression.

Extended binary tree can be converted from a binary tree by adding new nodes to its leaf nodes and the nodes that have only one child. The extended binary tree also is known as 2-tree.

External sorting is a type of sorting, when the list of elements or records to be sorted in files are accommodated in external storage devices (secondary memory such as hard disks).

Full binary tree is a binary tree of depth $k \geq 1$, that contains the maximum number of nodes $2^k - 1$.

Graph G is defined as an Ordered set $G = (V, E)$, where V represents a set of elements called nodes (or points or vertices) and E represents a set of edges in G, that connects these vertices.

In-place sorting algorithm when a constant amount of data elements of an input array is never stored outside the array. No additional array is required.

Internal Sorting is a type of sorting, when the list of elements or records to be sorted are small enough to be accommodated in the internal (primary) memory of the computer.

Linear data structure is a type of data structure in which every data item is related (or attached) to its previous and next data item and data items are arranged in memory in a linear sequence.

Linked list is an ordered collection of finite homogeneous data elements called node where the linear order is maintained by means of links or pointers.

Non-linear data structure is a type of data structure in which every data item is attached to many other data items in specific to reflect relationships and the data items are not in sequence.

Non-primitive data structures are derived from the primitive data structure and that cannot be manipulated or operated directly by the machine instructions.

Merging is an operation that combines the data items of two or more different sorted lists into a single sorted list.

Offline sorting algorithm cannot start unless all the data items are present at the beginning. All the elements must be available beforehand.

Pointer a variable that is used to store an address or location of another variable.

Polish notation where binary operators appear before its two operands. This notation also is known as prefix notation.

Primitive data structure is defined that can be manipulated or operated by the machine instruction and generally built upon simpler primitive data types.

Priority Queue is a collection of elements such that each element have been assigned a priority and the elements are arranged based on priority.

Queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

Recursion a repetitive process in which a function calls itself either directly or indirectly.

Reverse Polish notation where binary operators appear after its two operands. This notation also is known as postfix notation.

Searching is an operation of finding the location of the desired key value within the data structure.

Skewed binary tree is a binary tree, which is dominated solely by left child nodes or right child nodes, more specifically left skewed binary tree, or right skewed binary tree.

Sparse Matrix a matrix most of its elements are zero, having a relatively small number of non-zero elements.

Sorting means arranging all the data items in a data structure either in ascending or in descending order or in lexicographical order (for Strings).

Stable Sorting is a type of sorting when two elements that are equal (i.e. duplicate element) remain in the same relative order even after performing sorting.

Stack a collection of elements into which new elements may be inserted and from which elements may be deleted only at one end called the top of the stack.

Static data structure is a kind of data structure, in which once memory space is allocated it cannot extend.

Strictly binary tree is a binary tree when every non-terminal node has non-empty left and right subtree.

String is a sequence of characters.

Tail recursion is a special form of recursion where the last operation of a function is a recursive call and there are no pending operations to be performed on return from a recursive call.

Traversing is an operation of accessing and processing each data item of the data structure exactly once.

Tree is a non-linear data structure. A Tree may be defined as a non-empty finite set of nodes, such that, i) There is a specially designated node called the root, ii) The remaining nodes are partitioned into zero or more disjoint trees $T_1, T_2 \dots T_n$ are called the subtrees of the root R.

BIBLIOGRAPHY

- [1] Seymour Lipschutz, Data Structure and Algorithms, Tata McGrawHill,
- [2] G A Vijayalakshmi PAI, Data Structures and Algorithms, Tata McGrawHill
- [3] Nitin Upadhyay, Data Structures and Algorithms using C, Katson
- [4] Samiran Chattopadhyay, Data Structures through C Language, BPB Publications

SOLVED PROBLEMS

Q1. We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?

Ans. B

Given a binary tree, the inorder traversal outputs exactly one permutation of its nodes. Since the values are distinct, the values must be inserted in sorted order in these nodes in order. SO, there is only one way.

Q2. Statement for Linked Answer Questions 54 and 55:

An undirected graph $G(V, E)$ contains $n(n > 2)$ nodes named v_1, v_2, \dots, v_n . Two nodes $v_i - v_j$ connected

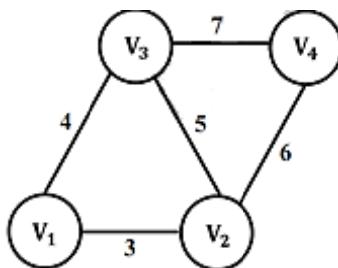


Figure G.1

Q3. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes?

- (A) $\frac{1}{12}(11n^2 - 5n)$ (B) $n^2 - n + 1$ (C) $6n - 11$ (D) $2n + 1$

Ans. One can test that for $n = 2$ and $n = 3$, the minimum spanning tree weight must be 3 and 7 respectively. We can thus infer that option B is the right answer. The optimum way of constructing the tree is as follows.

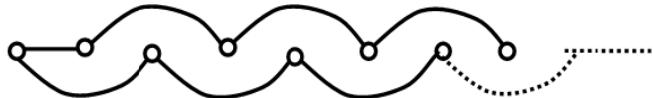


Figure G.2

A new vertex V_{n+1} added is made adjacent to the V_{n-1} in the spanning tree. The spanning tree weight increases by $(n+1) + (n-1) = 2n$

One can verify that $(n^2-n+1)+2n = (n+1)^2-(n+1)-1$

Q4. The length of the path from V_5 to V_6 in the MST of previous questions with $n=10$ is

Ans. C

Let e_{ij} denote the edge from V_i to V_j in the spanning tree.

The path from V_i to V_j is given by $e_{53}e_{31}e_{12}e_{24}e_{46}$ and its weight is

$$5 + 3 + 3 + 1 + 1 + 2 + 2 + 4 + 4 + 6 = 31$$

Q5. The worst case running time to search for an element in a balanced binary search tree with n^{2^n} elements is

- (A) $\Theta(n \log n)$ (B) $\Theta(n^2)$ (C) $\Theta(n)$ (D) $\Theta(\log n)$

Ans: (C)

Q6. Let G be a weighted graph with edge weights greater than one and G' be the graph constructed by squaring the weights of edges in G. Let T and T' be the minimum spanning trees of G and G', respectively, with total weights t and t'. Which of the following statements is TRUE?

- (A) $T' = T$ with total weight $t' = t^2$ (B) $T' = T$ with total weight $t' < t^2$
 (C) $T' \neq T$ but total weight $t' = t^2$ (D) None of the above

Ans: (D)

Q7. Suppose a circular queue of capacity $(n - 1)$ elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are

- (A) Full: $(\text{REAR}+1) \bmod n == \text{FRONT}$, empty: $\text{REAR} == \text{FRONT}$
 (B) Full: $(\text{REAR}+1) \bmod n == \text{FRONT}$, empty: $(\text{FRONT}+1) \bmod n == \text{REAR}$
 (C) Full: $\text{REAR} == \text{FRONT}$, empty: $(\text{REAR}+1) \bmod n == \text{FRONT}$
 (D) Full: $(\text{FRONT}+1) \bmod n == \text{REAR}$, empty: $\text{REAR} == \text{FRONT}$

Ans: (A)

Q8. A list of n strings, each of length n, is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is

- (A) $O(n \log n)$ (B) $O(n^2 \log n)$ (C) $O(n^2 + \log n)$ (D) $O(n^2)$

Ans: (B)

Q9. The height of a tree is defined as the number of edges on the longest path in the tree. The function shown in the pseudocode below is invoked as height (root) to compute the height of a binary tree rooted at the tree pointer root

```
int height(treeptn n)
{
if(n==NULL) return -1;
if(n->left==NULL)
    if(n->right==NULL) return 0;
    else return B1      //Box1
else {h1 = height(n->left);
      if(n->right==NULL) return(1+h1);
      else {h2 = height(n->right);
            return B2;  //Box2
          }
      }
}
```

The appropriate expression for the two boxes B1 and B2 are

- (A) B1 : $(1 + \text{height}(n->\text{right}))$, B2 : $(1 + \max(h1, h2))$
 (B) B1 : $(\text{height}(n->\text{right}))$, B2 : $(1 + \max(h1, h2))$
 (C) B1 : $\text{height}(n->\text{right})$, B2 : $\max(h1, h2)$
 (D) B1 : $(1 + \text{height}(n->\text{right}))$, B2 : $\max(h1, h2)$

Ans: (A)

Q10. What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of n vertices?

- (A) $\Theta(n^2)$ (B) $\Theta(n^2 \log n)$ (C) $\Theta(n^3)$ (D) $\Theta(n^3 \log n)$

Ans. C

Time complexity of Bellman-Ford algorithm is $O(VE)$ where V is number of vertices and E is number of edges. For a complete graph with n vertices, $V = n$, $E = O(n^2)$. So overall time complexity becomes $O(n^3)$

Q11. Which of the following statements is/are TRUE for undirected graphs?

P: Number of odd degree vertices is even.

Q. Sum of degrees of all vertices is even.

Ans. C

P is true for undirected graph as adding an edge always increases degree of two vertices by 1.Q is true: If we consider sum of degrees and subtract all even degrees, we get an even number because every edge increases the sum of degrees by 2. So total number of odd degree vertices must be even.

Q12. The line graph $L(G)$ of a simple graph G is defined as follows: There is exactly one vertex $v(e)$ in $L(G)$ for each edge e in G . · For any two edges e and e' in G , $L(G)$ has an edge between $v(e)$ and $v(e')$, if and only if e and e' are incident with the same vertex in G . Which of the following statements is/are TRUE?

- (A) The line graph of a cycle is a cycle.
 - (B) The line graph of a clique is a clique.
 - (C) The line graph of a planar graph is planar.
 - (D) The line graph of a tree is a tree.

Ans: (A)

Q13. The number of elements that can be sorted in $\Theta(\log n)$ time using heap sort is

- (A) $\Theta(1)$ (B) $\Theta(\sqrt{\log n})$ (C) $\Theta\left(\frac{\log n}{\log \log n}\right)$ (D) $\Theta(\log n)$

Ans. C

Q14. Consider the following function:

```
int unknown(int n) {
```

```
int i, j, k = 0;
```

```
for (i = n/2; i <= n; i++)  
    for (j = 2; j <= n; j = j * 2)
```

$$k = k + m$$

re

- (A) $O(n^2)$ (B) $O(n^2 \log n)$ (C) $O(n^3)$ (D) $O(n^3 \log n)$

(A) Θ(n)

Q15. Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

K is a global parallel MultiDegree(Ω) {

MULTIDIM

while (Ω is not empty and $m \geq 0$) {

Dequeue(0)

$m \equiv m - 1$

1

3

What is the worst case time complexity of a sequence of n MultiDequeue() operations on an initially empty queue?

- (A) $\Theta(n)$ (B) $\Theta(n+k)$ (C) $\Theta(nk)$ (D) $\Theta(n^2)$

Ans Δ

Q16 Consider the function func shown below:

Q18. Consider the following function:

```
{  
    int count = 0;  
    while (num)  
    {  
        count++;  
        num >= 1;  
    }  
    return (count);  
}
```

Run on IDE

The value returned by func(435) is _____.

Ans. (B)

Q17. Suppose n and p are unsigned int variables in a C program. We wish to set p to $nC3$. If n is large, which of the following statements is most likely to set p correctly?

- (A) $p = n * (n-1) * (n-2) / 6$ (B) $p = n * (n-1) / 2 * (n-2) / 3;$
 (C) $p = n * (n-1) / 3 * (n-2) / 2;$ (D) $p = n * (n-1) * (n-2) / 6.0;$

Ans. As n is large, the product $n*(n-1)*(n-2)$ will go out of the range(overflow) and it will return a value different from what is expected. So we consider a shorter product $n*(n-1)$. $n*(n-1)$ is always an even number. So the subexpression " $n * (n-1) / 2$ " in option B would always produce an integer, which means no precision loss in this subexpression. And when we consider ' $n*(n-1)/2*(n-2)$ ', it will always give a number which is a multiple of 3. So dividing it with 3 won't have any loss.

Q18. Let G be a graph with n vertices and m edges. What is the tightest upper bound on the running time of Depth First Search on G , when G is represented as an adjacency matrix?

- (A) $\Theta(n)$ (B) $\Theta(n+m)$ (C) $\Theta(n^2)$ (D) $\Theta(m^2)$

Ans. C

Q19. Consider a rooted Binary tree represented using pointers. The best upper bound on the time required to determine the number of subtrees having exactly 4 nodes $O(n \log b)$. Then the value of $a + 10b$ is

Ans. A

We can find the subtree with 4 nodes in $O(n)$ time. The following can be a simple approach.

- 1) Traverse the tree in bottom up manner and find the size of the subtree rooted with the current node
 - 2) If size becomes 4, then print the current node.

Q20. Consider the following graph

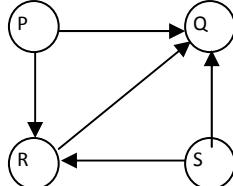


Figure G.3

Which one of the following is true?

- (A) The graph does not have any topological ordering
 (B) Both PQRS and SRPQ are topological ordering

- (C) Both PSRQ and SPRQ are topological ordering
 - (D) PSRQ is the only topological ordering

Ans. C

The graph does not contain any cycle, so there exist topological ordering.

P and S must appear before R and Q because there are edges from P to R and Q, and from S to R and Q.

Q21. Let P be a QuickSort Program to sort numbers in ascending order using the first element as a pivot. Let t_1 and t_2 be the number of comparisons made by P for the inputs $\{1, 2, 3, 4, 5\}$ and $\{4, 1, 5, 3, 2\}$ respectively. Which one of the following holds?

- (A) $t_1 = 5$ (B) $t_1 < t_2$ (C) $t_1 > t_2$ (D) $t_1 = t_2$

Ans. C

Q22. Consider a hash table with 9 slots. The hash function is $h(k) = k \bmod 9$. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are

- (A) 3, 0, and 1 (B) 3, 3, and 3 (C) 4, 0, and 1 (D) 3, 0, and 2

Ans: A

Q23. Consider an undirected graph G where self-loops are not allowed. The vertex set of G is $\{(i, j) : 1 \leq i \leq 12, 1 \leq j \leq 12\}$. There is an edge between (a, b) and (c, d) if $|a - c| \leq 1$ and $|b - d| \leq 1$. The number of edges in this graph is _____.

- The number of edges in this graph is _____.

Ans: C

Q24. A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is: 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

- (A) 0 8 7 3 2 1 5 (B) 10 8 7 2 3 1 5 (C) 10 8 7 1 2 3 5 (D) 10 8 7 5 3 2 1

Ans: A

Q25. Consider the tree arcs of a BFS traversal from a source node W in an unweighted, connected, undirected graph. The tree T formed by the tree arcs is a data structure for computing

- (A) The shortest path between every pair of vertices.
(B) The shortest path from W to every vertex in the graph.
(C) The shortest paths from W to only those nodes that are leaves of T.
(D) The longest path in the graph

Ans: B

Q26. The order in which the nodes are visited during in-order traversal is

- (A) SOPTRWUV (B) SOPTURWV (C) SOPTWUVR (D) SOPTRUWV

Ans: (A) Go through the in-order traversal of the tree

Q27. You have an array of n elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is

- (A) $O(n^2)$ (B) $O(n \log n)$ (C) $\Theta(n \log n)$ (D) $O(n^3)$

Q28. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is

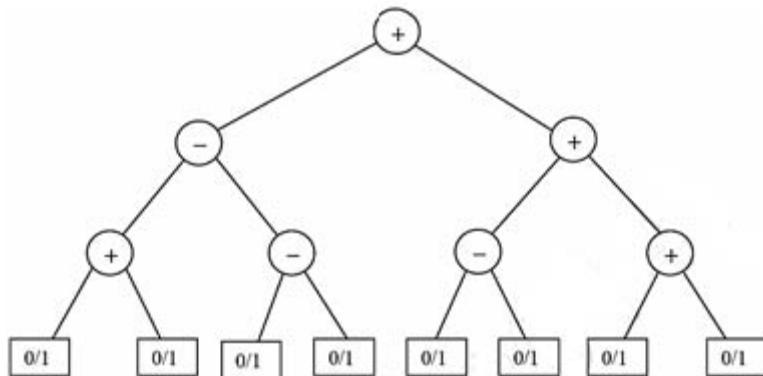


Figure G.4

(A) 4

(B) 6

(C) 8

(D) 10

Q29. A cycle on n vertices is isomorphic to its complement. The value of n is ____.

(A) 2

(B) 4

(C) 6

(D) 5

Ans.(D)5

The complement graph is also isomorphic (same number of vertices connected in same way) to given graph.

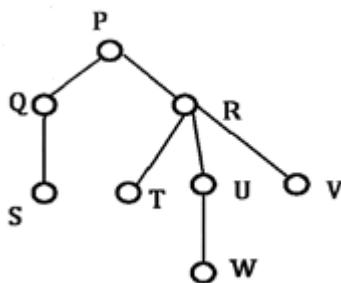


Figure G.5

Q30. Consider the pseudocode given below. The function DoSomething() takes as argument a pointer to the root of an arbitrary tree represented by the leftMostChild-rightSibling representation. Each node of the tree is of type treeNode.

```

typedef struct treeNode* treeptr;
struct treeNode
{
    treeptr leftMostChild, rightSibling;
};
int DoSomething (treeptr tree)
{
    int value=0;
    if (tree != NULL)
    {
        if (tree->leftMostChild == NULL)
            value = 1;
    }
}
  
```

```
    else
        value = DoSomething(tree->leftMostChild);
        value = value + DoSomething(tree->rightSibling);
    }
    return(value);
}
```

When the pointer to the root of a tree is passed as the argument to DoSomething, the value returned by the function corresponds to the

- (A) number of internal nodes in the tree.
(B) height of the tree.
(C) number of nodes without a right sibling in the tree.
(D) number of leaf nodes in the tree

Ans: (D)

Q31. Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)?

- 1. 3, 5, 7, 8, 15, 19, 25
 - 2. 5, 8, 9, 12, 10, 15, 25
 - 3. 2, 7, 10, 8, 14, 16, 20
 - 4. 4, 6, 7, 9, 18, 20, 25

- (A) 1 and 4 only (B) 2 and 3 only (C) 2 and 4 only (D) 2 only

Ans: A

Q32. What are the worst-case complexities of insertion and deletion of a key in a binary search tree?

- Q2. What are the worst case complexities of
(A) $\Theta(\log n)$ for both insertion and deletion
(B) $\Theta(n)$ for both insertion and deletion
(C) $\Theta(n)$ for insertion and $\Theta(\log n)$ for deletion
(D) $\Theta(\log n)$ for insertion and $\Theta(n)$ for deletion

Ans: (B)

Q33. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are

- (A) 63 and 6, respectively (B) 64 and 5, respectively
 (C) 32 and 6, respectively (D) 31 and 5, respectively

Ans: (A)

Q34. A file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index. Then that index is called

- (A) Dense (B) Sparse (C) Clustered (D) Unclustered

(A) Dens

Q35. Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is

- a value 35 is inserted into this heap. After insertion, the new heap is
 (A) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35 (B) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
 (C) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15 (D) 40, 35, 20, 10, 15, 16, 17, 8, 4, 20

▲ 50 (B)

Q36. The graph shown below has 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges: $\{(A, C), (B, C), (B, E), (E, F), (D, F)\}$. The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is

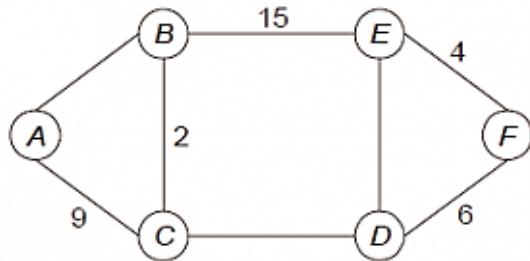


Figure G.6

Ans: (B)

Q37. With reference to the B+ tree index of order 1 shown below, the minimum number of nodes (including the root node) that must be fetched in order to satisfy the following query: "Get all records with a search key greater than or equal to 7 and less than 15" is

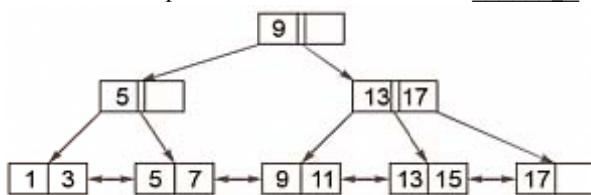


Figure G.7

Ans: (B)

Q38. A binary tree T has 20 leaves. The number of nodes in T having two children is

- (A) 18 (B) 19 (C) 17 (D) Any number between 10 and 20

Ans: (B)

Q39. Consider a complete binary tree where the left and the right subtrees of the root are max-heaps. The lower bound for the number of operations to convert the tree to a heap is

- (A) $\Omega(\log n)$ (B) $\Omega(n)$ (C) $\Omega(n \log n)$ (D) $\Omega(n^2)$

Ans: (A)

Q40. The result evaluating the postfix expression $10\ 5\ +\ 60\ 6\ /\ *\ 8$ – is

- (A) 284 (B) 213 (C) 142 (D) 71

Ans: (C). Postfix evaluation of expression

Q41. While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is

- (A) 65 (B) 67 (C) 69 (D) 83

Ans: (B)

Q42. Consider the following array of elements. {89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100}. The minimum number of interchanges needed to convert it into a max-heap is

1

ANS.(D)

Q43. Consider a binary tree T that has 200 leaf nodes. Then, the number of nodes in T that have exactly two children are _____.

Ans: (A) This can be proved using Handshaking Lemma. Refer below post to see complete proof.

Q44. Assume that a mergesort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

Ans: (B)

Q45. Consider the following recursive C function. If get(6) function is being called in main() then how many times will the get() function be invoked before returning to the main()?

```
void get (int n)
```

```
{  
    if (n < 1) return;  
    get(n-1);  
    get(n-3);  
    printf("%d", n);
```


Ans: (B)

Q46. Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes _____.

Ans: (B)

Q47. Consider B+ tree in which the search key is 12 bytes long, block size is 1024 bytes, record pointer is 10 bytes long and block pointer is 8 bytes long. The maximum number of keys that can be accommodated in each non-leaf node of the tree is

Ans: (B)

Q48. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)?

- (A) Both operations can be performed in $O(1)$ time
 - (B) At most one operation can be performed in $O(1)$ time but the worst case time for the other operation will be $\Omega(n)$
 - (C) The worst case time complexity for both operations will be $\Omega(n)$
 - (D) Worst case time complexity for both operations will be $O(\log n)$

Ans: (A)

Q49. In an adjacency list representation of an undirected simple graph $G = (V, E)$, each edge (u, v) has two adjacency list entries: $[v]$ in the adjacency list of u , and $[u]$ in the adjacency list of v . These are called twins of each other. A twin pointer is a pointer from an adjacency list entry to its twin. If $|E| = m$ and $|V| = n$, and the memory size is not a constraint, what is the time complexity of the most efficient algorithm to set the twin pointer in each entry in each adjacency list?

(A) $\Theta(n^2)$

(B) $\Theta(m+n)$

(C) $\Theta(m_2)$

(D) $\Theta(n^4)$

Ans: B

Q50. Consider the following directed graph. The number of different topological orderings of the vertices of the graph is Note: This question was asked as Numerical Answer Type.

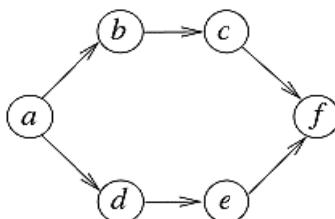


Figure G.8

(A) 1

(B) 2

(C) 4

(D) 6

Ans: (D)

Q51. What will be the output of the following C program?

```

void count(int n)
{
    static int d = 1;
    printf("%d ", n);
    printf("%d ", d);
    d++;
    if(n > 1) count(n-1);
    printf("%d ", d);
}
main()
{
    count(3);
}
  
```

Run on IDE

(A) 3 1 2 2 1 3 4 4 4

(B) 3 1 2 1 1 1 2 2 2

(C) 3 1 2 2 1 3 4

(D) 3 1 2 1 1 1 2

Ans: A

Q52. An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i-th node. Assume that the heap is implemented in an array and i refers to the i-th index of the array. If the heap tree has depth d (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element?

(A) O(1) (B) O(d) but not O(1) (C) O(2^d) but not O(d) (D) O($d2^d$) but not O(2^d).

Ans: B

Q53. Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. [This Question was originally a Fill-in-the-Blanks question]

(A) 6

(B) 7

(C) 8

(D) 9

Ans: B

Q54. Breadth First Search (BFS) is started on a binary tree beginning from the root vertex. There is a vertex t at a distance four from the root. If t is the n -th vertex in this BFS traversal, then the maximum possible value of n is [This Question was originally a Fill-in-the-blanks Question]

- (A) 15 (B) 16 (C) 31 (D) 32

Ans: C

Q55. The value printed by the following program is

```
void f(int* p, int m)
```

```

    {
        m = m + 5;
        *p = *p + m;
        return;
    }

```

1

```
}
```

{

```
int i=5, j=10;  
f(&i, j);  
printf("%d", i+j);
```

}

Run on IDE

Ans: C

Q56. N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease-key operation, a pointer is provided to the record on which the operation is to be performed. An algorithm performs the following operations on the list in this order: $\Theta(N)$ delete, $O(\log N)$ insert, $O(\log N)$ find, and $\Theta(N)$ decrease-key. What is the time complexity of all these operations put together?

- (A) $O(\log^2 N)$ (B) $O(N)$ (C) $O(N^2)$ (D) $\Theta(N^2 \log N)$

Ans: C

Q57. B+ Trees are considered BALANCED because

- (A) the lengths of the paths from the root to all leaf nodes are all equal.
(B) the lengths of the paths from the root to all leaf nodes differ from each other by at most 1.
(C) the number of children of any two non-leaf sibling nodes differ by at most 1.
(D) the number of records in any two leaf nodes differ by at most 1.

Ans: A

Q58. A complete binary min-heap is made by including each integer in [1, 1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus, the root is at depth 0. The maximum depth at which integer 9 can appear is

Ans: C

Q59. Consider the following New-order strategy for traversing a binary tree: Visit the root; Visit the right subtree using New-order Visit the left subtree using New-order The New-order traversal of the expression tree corresponding to the reverse polish expression $3\ 4\ *\ 5\ -\ 2\ ^\ 6\ 7\ *\ 1\ +$ - is given by:

- $$\begin{array}{l} (A) + - 1 \ 6 \ 7 * 2 ^ 5 - 3 \ 4 * \\ (C) - + 1 * 7 \ 6 ^ 2 - 5 * 4 \ 3 \end{array} \quad \begin{array}{l} (B) - + 1 * 6 \ 7 ^ 2 - 5 * 3 \ 4 \\ (D) 1 7 \ 6 * + 2 5 \ 4 \ 3 * - ^ \wedge \end{array}$$

Ans: C

Q60. The number of ways in which the numbers 1, 2, 3, 4, 5, 6, 7 can be inserted in an empty binary search tree, such that the resulting tree has height 6, is _____ Note: The height of a tree with a single node is 0. [This question was originally a Fill-in-the-Blanks question]

Q61. Consider the weighted undirected graph with 4 vertices, where the weight of edge $\{i, j\}$ g is given by the entry W_{ij} in the matrix W . The largest possible integer value of x , for which at least one shortest path between some pair of vertices will contain the edge with weight x is _____. Note: This question was asked as Numerical Answer Type.

$$W = \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix}$$

Figure G.9

INDEX

- 2-3 tree 8.50
2-3-4 tree 8.50
- Adjacency List 9.12
Adjacency Matrix 9.9
ADT 1.6
Algorithm 2.1
Ancestor 8.2, 8.27, 9.18
Array 3.1
Array of Pointers 4.27
Asymptotic notation 2.7
AVL tree 8.27
- B-tree 8.41,12.7
B+ tree 12.7
Backtracking 2.2, 7.12, 7.13
Balance Factor 8.27
Bellman ford Algorithm 9.27
Big O notation 2.5, 2.7
Binary search tree 2.11, 8.18
Binary Searching 5.2, 8.42, 10.3-10.6
Binary tree 7.4, 8.2
Branch 8.2
Branch-and-bound 2.2
Breadth First Search 6.14, 8.13, 8.16
Brute Force Algorithm 9.30
Bubble Sort 10.7-10.11
- Children 8.1
Circular doubly linked list 5.3, 5.36
Circular linked list 5.19, 5.25
Circular queue 5.24, 6.15
Close Hashing 11.4
Clustering Index 12.6, 12.7
Collision 11.1-11.3
Collision Resolution 11.1, 11.3
Complete Binary Tree 6.18, 8.4, 8.10, 8.26, 10.27
Cut vertex 9.6
Cycle 9.3, 9.7, 9.21, 9.33
- Dangling Pointer 4.8
Degree 8.2, 9.2
Dense Matrix 3.16
Dense Primary index 12.5, 12.6
Depth First Search 6.5, 8.13, 9.16, 9.21
Deque 6.16, 6.17
Dequeue 1.7, 2.11, 6.11
Descendant 8.2, 9.18, 10.33
Digraph 9.1, 9.2, 9.5-9.7
Dijkstra's algorithm 9.26, 9.27
Divide and Conquer 2.2, 10.20
Doubly linked list 5.3, 5.25
- Dynamic Data Structure 1.6, 5.2, 6.4
Dynamic memory allocation 4.5, 4.19, 4.27, 5.4
Dynamic programming 2.2, 9.28
- Edge 1.8, 8.2, 8.9, 9.1
Enqueue 1.7, 2.11, 6.11
Expression Tree 8.5-8.7
Extended Binary Tree 8.3
External Sorting 10.7
- FIFO 1.7, 6.11, 6.15
File 12.1
Floyd-Warshal Algorithm 9.28
Forest 8.2
Full Binary Tree 8.4
- Graph 1.6, 1.8, 9.1
Graph colouring 9.32
Greedy method 2.2
- Hash File organization 12.4
Hash Function 11.1, 11.2, 11.3
Hash table 1.4, 1.6, 3.15, 10.1, 11.1
Hashing 11.1
Heap 1.4, 2.4, 3.15, 6.18, 8.26
Heap File organization 12.3, 12.4
Heap Sort 2.11, 8.27, 10.7, 10.24, 10.33
Height Balanced binary tree 8.7, 8.8, 8.27
Height of tree 8.2, 8.8, 8.10
- Incident matrix 9.11
In-degree 9.2, 9.6, 9.7, 9.10
Index Sequential File organization 12.4
Infix notation 6.6, 6.7
Inorder Traversal 8.5, 8.9, 8.13, 8.17
In-Place Sorting 10.7, 10.24
Insertion Sort 10.7, 10.8, 10.12
Internal node 8.2-8.4, 8.8, 8.41, 8.48
Internal Sorting 10.7
Interpolation Searching 5.2, 10.6
Isolated vertex 9.2, 9.6
Isomorphism 9.4
- Josephus Problem 5.25
- Kruskal's Algorithm 9.23, 9.24
- L0 rotation 8.35
L1 rotation 8.38
L-1 Rotation 8.37
Leaf Node 6.18, 7.8, 8.2
Level of tree 7.2, 8.2

- LIFO 1.7, 6.1
- Linear Data Structure 1.5
- Linear probing 11.4, 11.12
- Linear Searching 3.12, 5.2, 5.20, 5.36, 10.1
- Linked list 5.1
- Little o notation 2.7, 2.9
- LL rotation 8.28
- Load factor 11.12
- Loop 9.2, 9.4
- LR rotation 8.30

- Merge sort 2.12, 7.4, 10.7, 10.20
- Merging 1.7, 3.9, 3.13
- Minimum Spanning tree 9.21, 9.24
- Multidimensional array 3.9
- Multigraph 9.5
- Multilevel Indexing 12.7
- Multi-way Search tree 8.40, 8.41

- Network flow 9.32
- Non-attacking 8 queens 7.4, 7.5, 7.12, 7.14
- Non-Linear Data Structure 1.5, 1.8
- Null Pointer 4.4

- Offline Sorting 10.8
- Omega (Ω) notation 2.7, 2.8
- Online Sorting 10.8
- Open Addressing 11.4
- Orthogonal list 9.14
- Out-degree 9.2, 9.6, 9.7

- Parent node 8.1, 10.25
- Path 8.2, 8.4, 9.3
- Pendant edge 9.6
- Pendant Vertex 9.6
- Pointer 1.4, 4.3
- Pointer to Pointer 4.11
- Polish notation 6.6
- Pop 2.11, 6.2
- Postfix notation 6.6
- Postorder Traversal 8.7, 8.8, 8.15, 8.16
- Precedence graph 9.32, 9.33
- Predecessor 8.1
- Prefix notation 6.6
- Preorder Traversal 8.7, 8.14, 8.16
- Prim's algorithm 9.24
- Primary clustering 11.6, 11.7, 11.11
- Primary Index 12.4-12.6
- Primitive data structure 1.4
- Priority queue 6.17, 6.18, 6.19
- Push 2.11, 6.2

- Quadratic probing 11.4, 11.7, 11.8
- Queue 1.4, 2.11, 5.24, 6.11
- Quick Sort 10.15

- R0 rotation 8.35
- R1 rotation 8.36
- R-1 Rotation 8.37
- Radix Sort 6.14, 10.7, 10.33
- Reachability 9.6
- Recursion 2.5, 6.5, 7.1
- Recursion Tree 7.2, 7.8
- Red-Black Tree 8.8, 8.39
- Regular graph 9.2
- Reverse Polish notation 6.6
- RL rotation 8.31
- Root 8.1, 10.24
- RR rotation 8.29

- Secondary clustering 11.7, 11.8, 11.11
- Secondary Index 12.6
- Selection Sort 2.11, 10.7, 10.8, 10.14, 10.15
- Sequential File organization 12.3
- Shell Sort 10.7, 10.8, 10.35-10.37
- Shortest Path 9.20, 9.25-9.29, 9.31-9.32
- Siblings 8.2, 8.42
- Sink 9.6, 9.33
- Skewed Binary Tree 8.5
- Source 9.6
- Space complexity 2.4, 2.5
- Spanning tree 9.18, 9.20-9.25
- Sparse Matrix 3.16
- Sparse Primary Index 12.6
- Stable Sorting 10.7
- Stack 1.4, 1.7, 2.4, 4.19, 6.1, 7.1
- Static Data Structure 1.6
- Strictly Binary tree 8.3, 8.5
- String 1.3, 3.21
- Strongly connected graph 9.6
- Subgraph 9.5, 9.7, 9.21

- Tail recursion 7.4
- Theta (Θ) notation 2.7, 2.9
- Threaded Binary Tree 8.8, 8.9, 8.12
- Time complexity 2.4, 2.5, 2.6
- Tower of Hanoi 7.10
- Travelling Salesman Problem 9.30
- Traversing 1.4, 1.5, 1.7
- Tree 1.4, 2.11, 8.1

- Undirected graph 9.1
- Unilaterally connected graph 9.7

- Void pointer 4.5

- Walk 9.2
- Weakly connected digraph 9.7
- Weight Balanced binary tree 8.7