

CS3353 C PROGRAMMING AND DATA STRUCTURES**L T P C****3 0 0 3****COURSE OBJECTIVES:**

- To introduce the basics of C programming language.
- To learn the concepts of advanced features of C.
- To understand the concepts of ADTs and linear data structures.
- To know the concepts of non-linear data structure and hashing.
- To familiarize the concepts of sorting and searching techniques.

UNIT I C PROGRAMMING FUNDAMENTALS (8+1 SKILL) 9

Data Types – Variables – Operations – Expressions and Statements – Conditional Statements – Functions – Recursive Functions – Arrays – Single and Multi-Dimensional Arrays.

UNIT II C PROGRAMMING - ADVANCED FEATURES (8+1 SKILL) 9

Structures – Union – Enumerated Data Types – Pointers: Pointers to Variables, Arrays and Functions – File Handling – Preprocessor Directives.

UNIT III LINEAR DATA STRUCTURES (8+1 SKILL) 9

Abstract Data Types (ADTs) – List ADT – Array-Based Implementation – Linked List – Doubly- Linked Lists – Circular Linked List – Stack ADT – Implementation of Stack – Applications – Queue ADT – Priority Queues – Queue Implementation – Applications.

UNIT IV NON-LINEAR DATA STRUCTURES (8+1 SKILL) 9

Trees – Binary Trees – Tree Traversals – Expression Trees – Binary Search Tree – Hashing - Hash Functions – Separate Chaining – Open Addressing – Linear Probing– Quadratic Probing – Double Hashing – Rehashing.

UNIT V SORTING AND SEARCHING TECHNIQUES

(8+1 SKILL)

9

Insertion Sort – Quick Sort – Heap Sort – Merge Sort – Linear Search – Binary Search.

TOTAL 45 PERIODS

SKILL DEVELOPMENT ACTIVITIES

(Group Seminar/Mini Project/Assignment/Content Preparation / Quiz/ Surprise Test / Solving GATE questions/ etc) 5

COURSE OUTCOMES:

CO1: Develop C programs for any real world/technical application.

CO2: Apply advanced features of C in solving problems.

CO3: Write functions to implement linear and non-linear data structure operations.

CO4: Suggest and use appropriate linear/non-linear data structure operations for solving a given problem.

CO5: Appropriately use sort and search algorithms for a given application.

CO6: Apply appropriate hash functions that result in a collision free scenario for data storage and retrieval.

TEXT BOOKS:

1. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, Second Edition, Pearson Education, 1997.
2. Reema Thareja, “Programming in C”, Second Edition, Oxford University Press, 2016.

REFERENCES:

1. Brian W. Kernighan, Rob Pike, “The Practice of Programming”, Pearson Education, 1999.
2. Paul J. Deitel, Harvey Deitel, “C How to Program”, Seventh Edition, Pearson Education, 2013.

3. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, “Data Structures and Algorithms”, Pearson Education, 1983.
4. Ellis Horowitz, Sartaj Sahni and Susan Anderson, “Fundamentals of Data Structures”, Galgotia, 2008.

List of Open Source Software/ Learning website:

<https://www.coursera.org/specializations/data-structures-algorithms>

<https://nptel.ac.in/courses/112107243> <https://nptel.ac.in/courses/112105598>

STUCOR APP

CONTENTS

UNIT-I

C PROGRAMMING FUNDAMENTALS

1.1	Introduction to Programming Paradigms	1.1
	1.1.1 Imperative Programming Paradigm	1.1
	1.1.1.1 Procedural Programming Paradigm	1.2
	1.1.1.2 Object Oriented Programming	1.3
	1.1.1.3 Parallel Processing Approach	1.7
	1.1.2 Declarative Programming Paradigm	1.7
	1.1.2.1 Logic Programming Paradigms	1.7
	1.1.2.2 Functional Programming Paradigms	1.7
	1.1.2.3 Database/Data Driven Programming Approach	1.8
	1.1.3 Characteristics of a Good Programming Language	1.8
1.2	History of C Programming	1.9
	1.2.1 Features of C Programming	1.10
1.3	Applications of C Language	1.10
1.4	Structure of C Program	1.11
1.5	Lexical Elements of C	1.14
	1.5.1 Character Set	1.15
	1.5.2 Delimiters	1.17
	1.5.3 Keywords	1.17
	1.5.4 Identifiers	1.18
	1.5.5 Data Types	1.18
	1.5.6 Constants	1.25
	1.5.7 Variables and Declaration	1.27
1.6	Operators in C	1.29
	1.6.1 Types of Operators	1.29
	1.6.1.1 Arithmetic Operators	1.30
	1.6.1.2 Relational Operators	1.32
	1.6.1.3 Logical Operators	1.32
	1.6.1.4 Assignment Operator	1.34
	1.6.1.5 Increment and Decrement Operators (Unary Operators)	1.35

1.6.1.6	Conditional Operator (Or) Ternary Operator	1.37
1.6.1.7	Bitwise Operators	1.38
1.6.1.8	The Special Operator	1.39
1.7	Expressions And Statements	1.41
1.7.1	Expressions	1.41
1.7.2	Statements	1.41
1.8	Conditional Statements	1.42
1.8.1	Conditional Branching Statement	1.43
1.8.1.1	Selection Statement	1.43
1.8.1.2	Looping Statements	1.58
1.9	Functions	1.64
1.9.1	Function Prototype	1.65
1.9.1.1	User Defined Function	1.65
1.9.1.2	Elements of User Defined Functions	1.66
1.10	Recursive Functions	1.78
1.11	Arrays	1.81
1.11.1	Declaration of an Array	1.82
1.11.2	Initialization of Arrays	1.83
1.12	One Dimensional Array	1.83
1.12.1	Declaration of Single Dimensional Array	1.83
1.12.2	Initialization of Single Dimensional Array	1.84
1.13	Multi-Dimensional Array	1.87
1.13.1	Two Dimensional Array	1.87
1.13.2	Three-Dimensional Arrays	1.91
	Review Questions	1.94

UNIT -II

PROGRAMMING - ADVANCED FEATURES

2.1	Introduction	2.1
2.2	Structure	2.1
2.2.1	Three Main Aspects of Working with Structure	2.1
2.2.1.1	Defining Structure	2.2
2.2.1.2	Initializing Structure Elements	2.3

2.2.1.3	Declaring Structure Objects	2.3
2.2.2	Operations on Structures	2.4
2.2.2.1	Aggregate Operations	2.4
2.2.2.2	Segregate Operations	2.11
2.3	Union	2.12
2.4	Pointers	2.14
2.4.1	Pointers to Variables	2.14
2.4.2	Pointer Operators	2.18
2.4.3	Arrays and Pointers	2.20
2.4.3.1	Pointers with Multi-Dimensional Array	2.23
2.4.4	Functions Pointers	2.24
2.5	Enumerated Data Types	2.25
2.5.1	Enumerated Type Declaration to Create a Variable	2.26
2.5.2	Implementing enum Using C Program	2.26
2.6	File Handling	2.27
2.6.1	Why We Need File	2.27
2.6.2	File Operations	2.28
2.6.2.1	Types of Files	2.28
2.7	Preprocessor Directives	2.41
	Review Questions	2.44

UNIT - III

LINEAR DATA STRUCTURES

3.1	Abstract Data Types (ADTS)	3.1
3.1.1	Abstract Data Type Model	3.1
3.2	List ADT	3.3
3.2.1	Operations on the List Data Structure	3.4
3.3	Array-Based Implementation	3.4
3.3.1	Properties of Array	3.4
3.3.2	Representation of an Array	3.5
3.3.3	Memory Allocation of an Array	3.5
3.3.4	Access an Element from the Array	3.6
3.3.5	Basic Operations of an Array	3.6

3.3.6	Complexity of Array Operations	3.7
3.3.7	Limitations of Array	3.7
3.3.8	Advantages of Array	3.7
3.3.9	Disadvantages of Array	3.7
3.4	Linked List	3.8
3.4.1	Representation of a Linked List	3.8
3.4.2	Why Use Linked List Over Array	3.8
3.4.3	Declare a Linked List	3.9
3.4.4	Types of Linked List	3.9
3.4.4.1	Singly-Linked List	3.9
3.4.4.2	Doubly Linked List	3.10
3.4.4.3	Circular Singly Linked List	3.10
3.4.4.4	Circular Doubly Linked List	3.10
3.4.5	Advantages of Linked List	3.10
3.4.6	Disadvantages of Linked List	3.11
3.4.7	Applications of Linked List	3.11
3.4.8	Operations Performed on Linked List	3.11
3.4.9	Complexity of Linked List	3.12
3.5	Doubly Linked List	3.12
3.5.1	Memory Representation of a Doubly Linked List	3.13
3.5.2	Operations on Doubly Linked List	3.14
3.5.2.1	Insertion at Beginning	3.15
3.5.2.2	Insertion at End	3.16
3.5.2.3	Insertion After Specified Node	3.18
3.5.2.4	Deletion at Beginning	3.19
3.5.2.5	Deletion at the End	3.20
3.5.2.6	Deletion of the Node Having Given Data	3.21
3.5.2.7	Searching for a Specific Node	3.22
3.5.2.8	Traversing in Doubly Linked List	3.23
3.5.3	Advantages of Doubly Linked Lists	3.23
3.5.4	Disadvantages of Doubly Linked Lists	3.24
3.6	Circular Linked List	3.24
3.6.1	Memory Representation of Circular Linked List	3.24

3.6.2	Operations on Circular Singly Linked List	3.25
3.6.2.1	Insertion at the Beginning	3.26
3.6.2.2	Insertion at the End	3.27
3.6.2.3	Deletion at the Beginning	3.28
3.6.2.4	Deletion at the End	3.30
3.6.2.5	Searching	3.31
3.6.2.6	Searching	3.32
3.6.3	Advantages of Circular Linked Lists	3.32
3.6.4	Disadvantages of Circular Linked Lists	3.33
3.7	Stack ADT	3.33
3.7.1	Working of Stack	3.33
3.7.2	Operations on Stack	3.34
3.7.2.1	Push Operation	3.34
3.7.2.2	Pop Operation	3.35
3.7.3	Applications of Stack	3.36
3.8	Implementation of Stack	3.36
3.8.1	Array Implementation of Stack	3.37
3.8.1.1	Adding an Element onto the Stack	3.37
3.8.1.1.1	Implementation of Push Algorithm in C Language	3.37
3.8.1.2	Deletion of an Element from a Stack	3.38
3.8.1.2.1	Implementation of Pop Algorithm Using C Language	3.38
3.8.1.3	Visiting Each Element of the Stack	3.39
3.8.2	Linked List Implementation Of Stack	3.40
3.8.2.1	Adding a Node to the Stack	3.40
3.8.2.1.1	Implementation of Push in C Language Program	3.41
3.8.2.2	Deleting a Node from the Stack	3.42
3.8.2.2.1	Implementation of Pop in C Language Program	3.43
3.8.2.3	Display the Nodes (Traversing)	3.43
3.8.2.3.1	Implementation of Display in C Language Program	3.43
3.9	Applications of Stack	3.44
3.9.1	Evaluation of Arithmetic Expressions	3.45

3.9.1.1	Infix Notation	3.46
3.9.1.2	Prefix Notation	3.46
3.9.1.3	Postfix Notation	3.46
3.9.2	Balancing Symbols	3.48
3.9.3	Processing Function Calls	3.50
3.9.4	Backtracking	3.51
3.9.5	Reverse a Data	3.53
3.9.5.1	Reverse a String	3.53
3.9.5.2	Converting Decimal to Binary	3.54
3.10	Queue ADT	3.55
3.10.1	Applications of Queue	3.55
3.10.2	Basic Operations in Queue	3.56
3.10.3	Types of Queue	3.56
3.10.3.1	Simple Queue or Linear Queue	3.56
3.10.3.2	Circular Queue	3.57
3.10.3.3	Priority Queue	3.57
3.10.3.4	Deque	3.58
3.11	Priority Queues	3.58
3.11.1	Characteristics of a Priority Queue	3.59
3.11.2	Types of Priority Queue	3.60
3.11.3	Representation of Priority Queue	3.60
3.11.4	Implementation of Priority Queue	3.61
3.11.5	Analysis of Complexities Using Different Implementations	3.62
3.11.6	Heap	3.62
3.11.7	Priority Queue Operations	3.63
3.11.7.1	Inserting the Element in a Priority Queue	3.63
3.11.7.2	Removing the Minimum Element From the Priority Queue	3.65
3.11.7.3	Peeking the Element From a Priority Queue	3.66
3.11.8	Applications of Priority Queue	3.67
3.12	Queue Implementation	3.67
3.12.1	Array Implementation of Queue	3.67
3.12.1.1	Algorithm to Insert Any Element in a Queue	3.68
3.12.1.2	Implementation Using C Function	3.69

3.12.1.3	Algorithm to Delete an Element From the Queue	3.70
3.12.1.4	Implementation Using C Function	3.70
3.12.1.5	Drawback of Array Implementation	3.71
3.12.2	Linked List Implementation of Queue	3.71
3.12.2.1	Insertion on Linked Queue	3.72
3.12.2.2	Implementation Using C Function	3.73
3.13	Applications of Queue	3.74
3.13.1	Type Declarations for Queue-Array Implementation	3.74
3.13.2	Routine to Test Whether a Queue is Empty-Array Implementation	3.74
3.13.3	Routine to Make an Empty Queue-Array Implementation	3.75
3.13.4	Routines to Enqueue-Array Implementation	3.75
3.13.5	Other Applications	3.76
	Review Questions	3.77

UNIT - IV

NON-LINEAR DATA STRUCTURES

4.1	Introduction to Tress	4.1
4.1.1	Example of Tree Data Structure	4.1
4.1.2	Basic Terminologies in Tree Data Structure	4.2
4.1.3	Properties of a Tree	4.2
4.1.4	Syntax for Creating a Node	4.3
4.2	Binary Trees	4.3
4.2.1	Binary Tree Representation	4.3
4.2.2	Types of Binary Trees	4.4
4.2.3	Benefits of Binary Trees	4.4
4.3	Tree Traversal	4.6
4.3.1	Types of Tree Traversal	4.6
4.4	Expression Trees	4.10
4.4.1	Properties of an Expression Tree	4.10
4.4.2	Construction of Expression Tree	4.10
4.4.3	Example - Postfix Expression Construction	4.11
4.4.4	Implementation of Expression Tree in C Programming Language	4.12

4.4.5	Use of Expression Tree	4.15
4.5	Binary Search Tree	4.15
4.5.1	Advantages of Binary Search Tree	4.16
4.5.2	Example of Creating a Binary Search Tree	4.16
4.5.3	Operations Performed on a Binary Search Tree	4.20
4.5.3.1	Searching in Binary Search Tree	4.20
4.5.3.1.1	Steps Involved in Searching in a Binary Search Tree	4.20
4.5.3.1.2	Algorithm to Search an Element in Binary Search Tree	4.22
4.5.3.2	Deletion in Binary Search Tree	4.22
4.5.3.2.1	When the Node to be Deleted is the Leaf Node	4.22
4.5.3.2.2	When the Node to be Deleted has Only One Child	4.23
4.5.3.3	Insertion in Binary Search Tree	4.24
4.5.3.4	The Complexity of the Binary Search Tree	4.24
4.5.3.5	Implementation of Binary Search Tree	4.25
4.6	Hashing	4.29
4.6.1	Examples	4.29
4.7	Hash Function	4.30
4.7.1	Hash Table	4.30
4.7.2	How Does Hashing in Data Structure Works	4.31
4.8	Separate Chaining	4.32
4.8.1	Separate Chaining Hash Table	4.32
4.8.2	Example for Separate Chaining	4.32
4.8.3	How to Avoid Collision in Separate Chaining Method	4.33
4.8.4	Practice Problem Based on Separate Chaining	4.34
4.8.5	Advantages and Disadvantages of Separate Chaining	4.37
4.9	Open Addressing	4.37
4.10	Linear Probing	4.37
4.10.1	Solution	4.38
4.11	Quadratic Probing	4.39
4.12	Double Hashing	4.41

4.12.1	Double Hashing - Hash Function 1 or First Hash Function – Formula	4.41
4.12.2	Double Hashing - Hash Function 2 or Second Hash Function – Formula	4.42
4.12.3	Double Hashing Example - Closed Hash Table	4.42
4.13	Re-Hashing	4.44
4.13.1	Why Rehashing is Done	4.44
4.13.2	What is Load Factor in Hashmap	4.44
4.13.3	How Rehashing is Done	4.44
4.13.4	Rehashing Steps	4.45
	Review Questions	4.46

UNIT - V

SORTING AND SEARCHING TECHNIQUES

5.1	Introduction to Sorting	5.1
5.2	Insertion Sort	5.2
5.2.1	Algorithm	5.3
5.2.2	Working of Insertion Sort Algorithm	5.3
5.2.3	Analysis of Insertion Sort	5.4
5.2.4	Applications	5.4
5.3	Quick Sort	5.6
5.3.1	Algorithm for Quick Sort	5.6
5.3.2	Working of Quick Sort Algorithm	5.6
5.3.3	Quicksort Complexity	5.8
5.3.4	Applications of Quick Sort	5.8
5.4	Heap Sort	5.11
5.4.1	What is a Heap	5.11
5.4.2	Working of Heap Sort Algorithm	5.11
5.4.3	Heapsort Complexity	5.16
5.4.4	Heap Sort Applications	5.16
5.5	Merge Sort	5.18
5.5.1	Algorithm	5.19
5.5.2	Working of Merge Sort Algorithm	5.19

5.5.3	Merge Sort Complexity	5.19
5.5.4	Merge Sort Applications	5.19
5.6	Introduction to Searching	5.23
5.6.1	Searching Methods	5.24
5.7	Linear Search	5.24
5.7.1	Steps Used in the Implementation of Linear Search	5.25
5.7.2	Algorithm	5.25
5.7.3	Working of Linear Search	5.25
5.7.4	Linear Search Complexity	5.27
5.7.5	Applications of Linear Search Algorithm	5.27
5.7.6	Advantages and Disadvantages	5.27
5.8	Binary Search	5.28
5.8.1	Algorithm	5.29
5.8.2	Working of Binary Search	5.29
5.8.3	Binary Search Complexity	5.31
5.8.4	Advantages and Disadvantages	5.31
5.8.5	Linear Search Vs Binary Search	5.33
	Review Questions	5.34



C PROGRAMMING FUNDAMENTALS

Data Types – Variables – Operations – Expressions and Statements – Conditional Statements – Functions – Recursive Functions – Arrays – Single and Multi-Dimensional Arrays.

1.1 INTRODUCTION TO PROGRAMMING PARADIGMS

- Paradigm can also be termed as method to solve some problem or do some task.
- Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.
- There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfil each and every demand. The programming paradigm is divided into two broad categories.
 - Imperative programming paradigm
 - Declarative programming paradigm

1.1.1 Imperative programming paradigm

- It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The

1.2 C Programming Fundamentals

paradigm consist of several statements and after execution of all the result is stored.

Advantage

1. Very simple to implement
2. It contains loops, variables etc.

Disadvantage

1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

Examples of **Imperative** programming paradigm: C, FORTAN, Basic

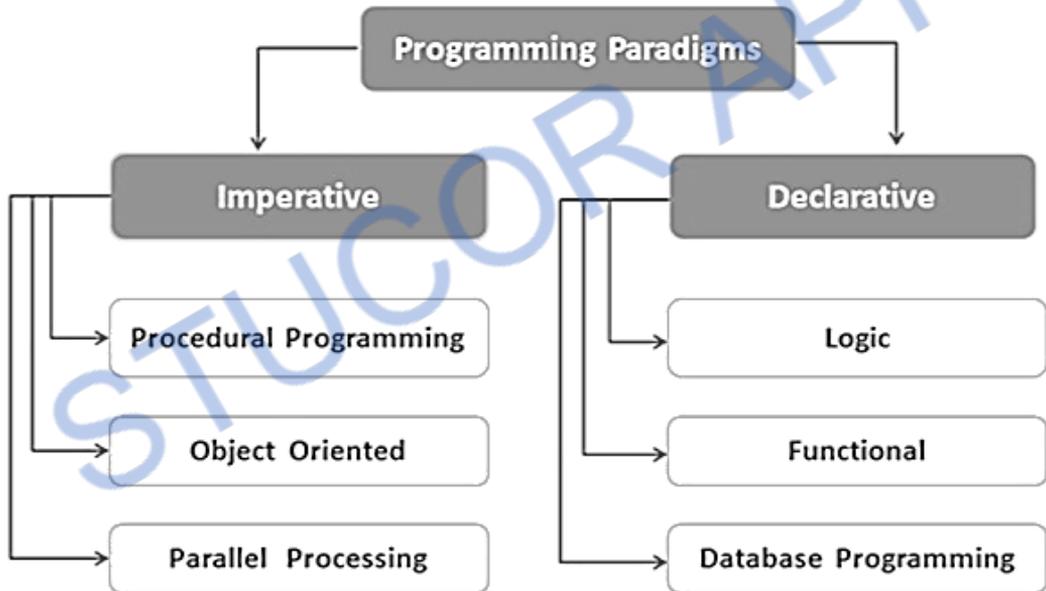


Figure 1.1 Types Programming Paradigms

- Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

1.1.1.1 Procedural programming paradigm

- This programming has a single program that is divided into small piece called procedure (also known as functions, routines, subroutines). These procedures are combined into one single location with the help of return statements.

- From the main controlling procedure, a procedure call is used to invoke the required procedure. After the sequence is processed, the flow of control continues from where the call was made.
- The main program coordinates calls to procedures and hands over appropriate data as parameters. The data is processed by the procedures and once the program has finished, the resulting data is displayed.
- Example: C, C++, JAVA, Pascal.

1.1.1.2 Object Oriented Programming

- It is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.
- In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.
- One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify. The basic concepts of OOP are as follows:
 1. Objects
 2. Classes
 3. Data abstraction and encapsulation
 4. Inheritance
 5. Polymorphism
 6. Dynamic binding
 7. Message passing.

Objects:

Objects are the basic run-time entities in an object oriented system. They may represent a person, place or a bank a/c or a table of data that the program has to handle. When a program is executed the objects interact by sending messages to one another. They can interact without knowing the details of each other's data or code. Thus an object is considered to be a partitioned area of computer memory that stores data and set of functions that can access the data.

Classes:

The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are variables of the type class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type.

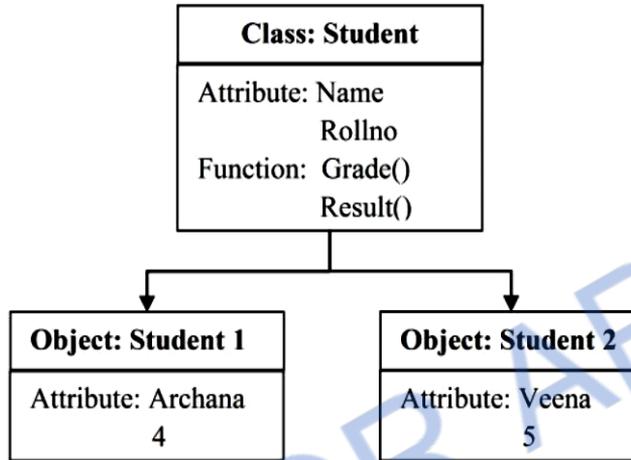


Fig. 1.2 Class and Objects

Data abstraction and encapsulation

Encapsulation:

The wrapping up of data and functions in to a single unit (that unit is called a class) is known as encapsulation. The data is not accessible to the outside world (other functions which are not the members of that class) and only those functions which are wrapped in the class can access it. This insulation of data from the direct access by the program is called data hiding or information hiding.

Abstraction:

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes. The attributes are sometimes called data members because they hold information. The functions that operate on these data are called member functions.

Inheritance:

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. In OOP, the

concept of inheritance provides the idea of reusability. This means that we can include additional features to an existing class without modifying it. It is important because it supports the concept of classification. C++ supports different types of inheritance such as single inheritance, multiple inheritance, multilevel inheritance and hierarchical inheritance.

To understand the concept of inheritance, let us consider an example of vehicles as shown in Fig.1.3. Here the class car is a subclass of automatic vehicle, which is again a subclass of the class vehicle. This implies that the class car has all the characteristics of automatic vehicles which in turn has all the properties of vehicles. However, car has some unique features which differentiate it from other subclasses. For example, it has four wheels and five gears, while scooter has two wheels and four gears.

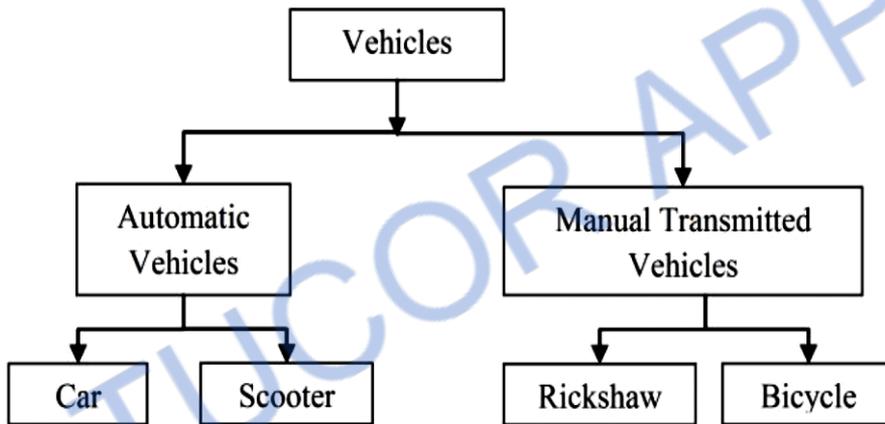


Fig. 1.3 Inheritance

Polymorphism:

Polymorphism, a Greek term means the ability to take more than one form. An operation may exhibit different behaviour in different instances. The behaviour depends up on the types of data used in the operation. The concepts of polymorphism are Operator overloading and Function overloading. For two numbers, the operator + will give the sum. If the operands are strings, then the operation would produce a third string by concatenation. Thus the process of making an operator to exhibit different behaviours in different instances is known as operator overloading. Similarly, we can use a single function to perform different tasks which is known as function overloading. A single function can be used to handle different number and types of arguments.

Binding:

Linking of procedure call to the corresponding code in response to the call.

Dynamic Binding:

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism concept.

Message Passing:

OOP consists of a set of objects that communicate with each other by sending and receiving information. A message for an object is a request for execution of a procedure (function) and therefore will invoke a function in the receiving object that generates the desired result.

Benefits of OOP

- Through inheritance we can eliminate redundant code and extend the use of existing classes.
- We can build secure programs by the principle of data hiding.
- It is easy to partition the work in a project based on objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Communication with external systems are much simpler by means of message passing techniques.
- Software complexity can be easily managed.

Applications of OOP

OOP can be applied in the following areas:

- Real time systems
- Simulation and modeling
- Object oriented data bases
- Hypertext, hypermedia and experttext
- Artificial Intelligence and expert systems.
- Neural networks and parallel programming
- Office automation systems
- CIM / CAM / CAD systems

Example: Simula, JAVA, Python, VB.NET, Ruby.

1.1.1.3 Parallel processing approach

- Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system possess many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer.
- Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

1.1.2 Declarative programming paradigm

- It is divided as Logic, Functional, and Database. In computer science the declarative programming is a style of building programs that expresses logic of computation without talking about its control flow.
- It often considers programs as theories of some logic. It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing.
- It just declare the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms.

1.1.2.1 Logic programming paradigms

- It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc.
- In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning.
- In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

1.1.2.2 Functional programming paradigms

- The functional programming paradigms has its roots in mathematics and it is language independent. The key principal of this paradigms is the execution of series of mathematical functions.

1.8 C Programming Fundamentals

- The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.
- The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like Perl, java script mostly uses this paradigm.

1.1.2.3 Database/Data driven programming approach

- This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps.
- A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application.
- For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

1.1.3 Characteristics of a Good Programming Language

- A programming language must be simple, easy to learn and use, have good readability and human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for development, debugging, testing, and maintenance of a program must be provided by a programming language.
- A programming language should provide single environment known as Integrated Development Environment (IDE).
- A programming language must be consistent in terms of syntax and semantics.

1.2 HISTORY OF C PROGRAMMING

- C is a general purpose structured programming language. C was developed by Dennis Ritchie at AT & T Bell laboratories in 1972. It is an outgrowth of an earlier language called BCPL & B. It was named as C to present it as the successor of B language which was developed earlier by Ken Thompson in 1970 at AT & T Bell laboratories. The various stages in evolution of C language is given in Fig 1.4.

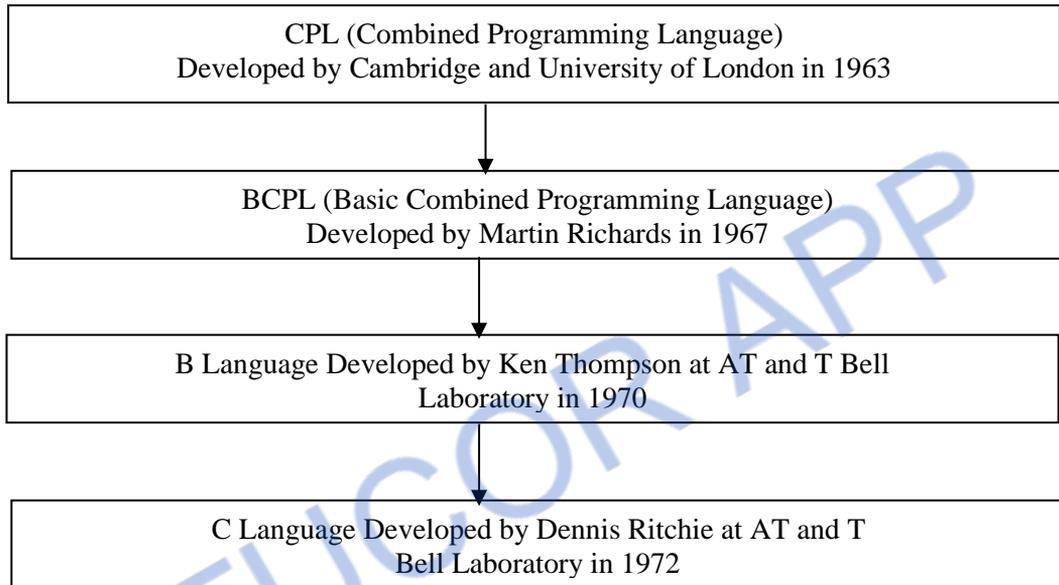


Fig. 1.4 Various Stages in Evaluation of C Languages

- C is a highly portable, which means that C program written for one computer can be run on another with little or no modification. C is well suited for structured programming; thus user has to think of a problem in terms of function modules and blocks.
- The proper collection of these modules would makes a complete program. This modular structures makes program debugging, testing and maintenance easier.
- Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can add our own functions to the C library. With the availability of a large number of functions, the programming task becomes simple.
- Its flexibility allows C to be used for system programming (For example: the UNIX operating system, the C compiler and all UNIX application software are written in C) as well as for application programming.

1.2.1 Features of C Programming

- C is the widely used language. It provides many features that are given below.
 1. Simple
 2. Machine Independent or Portable
 3. Mid-level programming language
 4. structured programming language
 5. Rich Library
 6. Memory Management
 7. Fast Speed
 8. Pointers
 9. Recursion
 10. Extensible

1.3 APPLICATIONS OF C LANGUAGE

- The applications of C are not only limited to the development of operating systems, like Windows or Linux, but also in the development of GUIs (Graphical User Interfaces) and, IDEs (Integrated Development Environments).
- Here are some striking applications offered by the C programming language:

1. Operating Systems

The first operating system to be developed using a high-level programming language was UNIX, which was designed in the C programming language. Later on, Microsoft Windows and various Android applications were scripted in C.

2. Embedded Systems

The C programming language is considered an optimum choice when it comes to scripting applications and drivers of embedded systems, as it is closely related to machine hardware.

3. GUI

GUI stands for Graphical User Interface. Adobe Photoshop, one of the most popularly used photo editors since olden times, was created with the help of C. Later on, Adobe Premiere and Illustrator were also created using C.

4. New Programming Platforms

Not only has C given birth to C++, a programming language including all the features of C in addition to the concept of object-oriented programming but, various other programming languages that are extensively used in today's world like MATLAB and Mathematica. It facilitates the faster computation of programs.

5. Google

Google file system and Google chromium browser were developed using C/C++. Not only this, the Google Open Source community has a large number of projects being handled using C/C++.

6. Mozilla Firefox and Thunderbird

Since Mozilla Firefox and Thunderbird were open-source email client projects, they were written in C/C++.

7. MySQL

MySQL, again being an open-source project, used in Database Management Systems was written in C/C++.

8. Compiler Design

One of the most popular uses of the C language was the creation of compilers. Compilers for several other programming languages were designed keeping in mind the association of C with low-level languages, making it easier to be comprehensible by the machine.

Several popular compilers were designed using C such as Bloodshed Dev-C, Clang C, MINGW, and Apple C.

9. Gaming and Animation

Since the C programming language is relatively faster than Java or Python, as it is compiler-based, it finds several applications in the gaming sector. Some of the simplest games are coded in C such as Tic-Tac-Toe, The Dino game, The Snake game and many more. Increasing advanced versions of graphics and functions, Doom3 a first-person horror shooter game was designed by id Software for Microsoft Windows using C in 2004.

1.4 STRUCTURE OF C PROGRAM

- As C is a programming language, let us go into the concepts of programming in C and it is a structured programming language. Every C program contains a number of building blocks.

1.12 C Programming Fundamentals

- These building blocks should be written in the correct order and procedure, for the C program to execute without any errors. The structure of C is given below.

```
documentation section
preprocessor section
definition section
global declaration section
main( )
{
declaration part;
executable part;
}
sub program
{
body of the subprogram;
}
```

Program 1.1

```
/* Program to find the area of a circle */
#include <stdio.h>
#include <conio.h>
#define PI 3.14
void main()
{
float area,r;
clrscr();
printf("\n Enter the radius:\n");
scanf("%f",&r);
area= PI*(r*r);
printf("\n Area of the Circle = %8.2f", area);
/* Documentation Section */
/* Preprocessor Section */
/* Definition Section */
/* main( ) function */
/* Local variable declaration */
// Executable part of the program
```

```
getch();  
}
```

Output

Enter the radius: 4.5

Area of the Circle = 63.58

Program 1.2

```
/* Program to find the sum of two numbers using function */  
/*Documentation Section */  
# include<stdio.h> /* Preprocessor Section */  
# include<conio.h>  
int a,b; /* Global Variable declaration */  
int add(int,int); /* Function declaration */  
void main() /* main( ) function */  
{  
int c; /* Local Variable declaration */  
clrscr(); // Executable part of the main() program  
printf("\n Enter the values for a and b:\n");  
scanf("%d %d",&a,&b);  
c = add(a,b);  
printf("\n Sum of %d + %d = %d", a,b,c);  
getch();  
}  
int add(int a,int b) /* Subprogram of add() function definition */  
{  
int c; /* Local Variable declaration */  
c=a+b; // Executable part of the function  
return(c);  
}
```

Output

Enter the values for a and b: 5 3

Sum of $5 + 3 = 8$

Documentation section

The documentation section is included in the comments, which contains the author name, the date of development and the program details.

Preprocessor section

The preprocessor section provides preprocessor statements which direct the compiler to link functions from the system library.

Definition section

The definition section defines all symbolic constants refer to assigning a macro of a name to a constant. The general syntax of a symbolic constant is

#define constant_name constant_value

Global declaration section

The global declaration section contains variable declarations which can be accessed anywhere within the program.

Main section

Main section is divided into two portions, the declaration part and the executable part. The declaration part used to declare any variables in the main block of the program. The executable part contains set of statements within the open and close braces. Execution of the program begins at the opening braces and ends at the closing braces.

User defined function section

The user defined function section (or) the Sub program section contains user defined functions which are called by the main function. Each user defined function contains the function name, the argument and the return value.

1.5 LEXICAL ELEMENTS OF C

- Data's can be of any kind, it may be numbers, characters and strings. These data should be processed in order to produce the information output. Programming languages are used for this processing of data into information.

- Every program consists of a sequence of steps or instruction for processing data. Each instruction must abide to certain syntax rules of grammar of the particular language. Likewise C has its own grammar. Data may be either a constant or a variable.
- In 'C' language each and every individual unit is called as Token or Lexical element. The various 'C' Tokens are
 - i) Character set
 - ii) Delimiters
 - iii) Keywords
 - iv) Identifiers
 - v) Data types
 - vi) Constants
 - vii) Variables

1.5.1 Character Set

- The set of characters used to write the program words, expressions and statements. It is the basic building block to form program elements. The set of characters used in a language is known as its *character set*. These characters can be represented in the computer.
- The C character set consists of upper and lower case alphabets, digits, special characters and white spaces. The alphabets and digits are together called the alphanumeric characters.

i) *Alphabets*

A, B, C,.....Z

a, b, c,....z

ii) *Digits*

0 1 2 3 4 5 6 7 8 9

iii) **Special Characters**

Table 1.1 List of Special Characters

,	Comma	-	Underscore
;	Semicolon	~	Tilde
:	Colon	\$	Dollar sign
#	Number sign	%	Percent sign
'	Apostrophe	&	Ampersand
“	Quotation mark	?	Question mark
!	Exclamation mark	*	Asterisk
	Vertical bar	-	Minus sign
+	Plus sign	.	Period
[Left bracket]	Right bracket
{	Left brace	}	Right brace
/	Slash	\	Backslash
^	Caret	()	Parenthesis left/right
<	Less than	=	Equal to
>	Greater than	@	At the rate

Table 1.1 List of Special Characters

iv) **White space characters**

Blank space, newline, form feed, horizontal tab, vertical tab

v) **Trigraph characters**

The *trigraph characters* are used to type certain characters that are not available on some keyboards. It consists of three characters. Two question marks followed by character.

Trigraph Characters	Translation
??=	# (pound sign)
?? ([(left bracket)
??)] (right bracket)
?? <	{ (left brace)
?? >	} (right brace)
?? !	/ backslash
?? /	vertical bar
?? ^	^ caret

Table 1.2 Trigraph Characters

1.5.2 Delimiters

- The language pattern of C uses a special kind of symbols, which are called *delimiters*. They are given in Table. 1.3.

Delimiters	Use
: Colon	Useful for label.
; Semicolon	Terminates the statement.
() Parenthesis	Used in expression and function.
[] Square Bracket	Used for array declaration.
{ } Curly Brace	Scope of the statement.
# Hash	Preprocessor directive.
, Comma	Variable separator.

Table 1.3 Delimiters

1.5.3 Keywords

- These are certain reserved words called *keywords*, which are standard, predefined meanings in C. They must be written in lower case. There are 32 keywords available in C. The standard keywords are shown in Table 1.4.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 1.4 Keywords in C

1.5.4 Identifiers

- *Identifiers* are the names defined by the programmer for various program elements such as variables, constant and functions. An identifier consists of a sequence of letters and digits.
- The following rules are to be followed to declare an identifier.
 - i) The first character must be a letter followed by a letter or a digit.
 - ii) Special characters are not allowed except underscore.
 - iii) The length of the variable varies from one compiler to another. Generally, most of the compilers support eight characters excluding extension. However, the ANSI standard recognizes the maximum length of a variable up to 31 characters.
 - iv) The variable should not be a C keyword.
 - v) The variable names may be a combination of upper and lower characters. For example, suM and sum are not the same variable.
 - vi) The variable name should not start with a digit.

Examples

- The following names are valid identifier

count	ic123a
area	ROSE
a123	india_123
book_no	Int
_filename	s_r_s

1.5.5 Data Types

- The C language supports different types of data. Each data may be represented differently within the computer memory. Typical memory requirements for each data will determine the possible range of values for that data type.
- The varieties of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

C supports three categories of data types:

1. Primary data type
2. Derived data type
3. User defined data type.

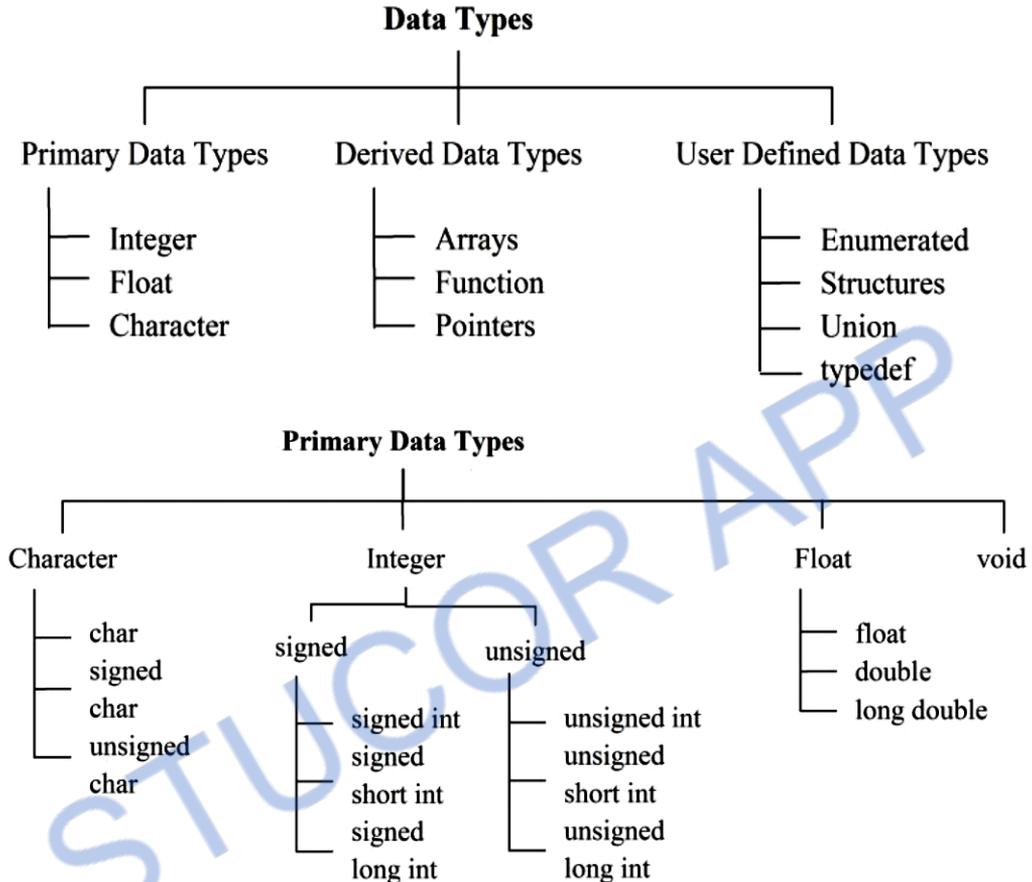


Fig. 1.5 Classification of 'C' Data types

Primary Data Types

- C compiler supports the following four Fundamental/ Primary/ Primitive/ Basic/ Built-in data types:
 - **Character: Character data type** is used to store a character. A variable of character data type allocated only one byte of memory and can store only one character. Keyword `char` is used to declare variables of type character. The range of character (`char`) data type is -128 to 127. For Example: `char ch = 'A';`
 - **Integer: Integer data type** is used to store a value of numeric type. Keyword `int` is used to declare variables of integer type. The memory size of a variable

1.20 C Programming Fundamentals

of integer data type is dependent on the operating system. For example the size of integer data type in a 32 bit computer is 4 bytes whereas size of integer data type in 16 bit computer is 2 bytes. For Example: *int* count = 10;

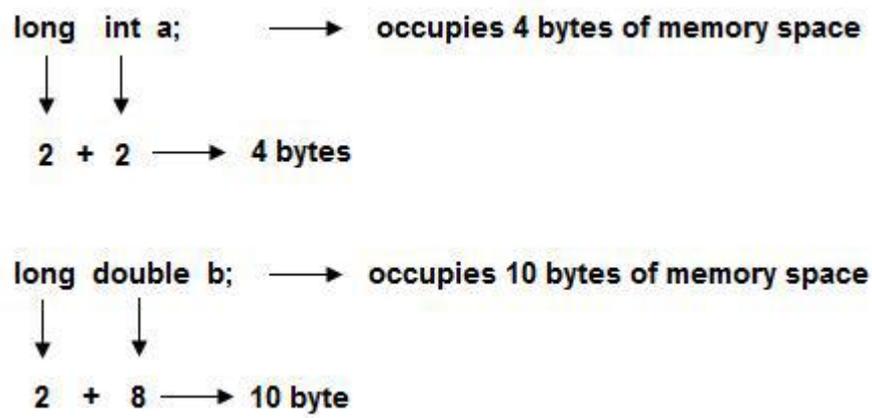
- **Float: Floating point data type** is used to store a value of decimal values. The memory size of a variable of floating point data type is dependent on the Operating System. Keyword *float* is used to declare variables of floating data type. For example the size of a floating point data type in a 16 bit computer is 4 bytes. For Example: *float* rate = 5.6;
- **Double: Double data type** is similar to floating data type except that it provides up to ten digits of precision and occupies eight bytes of memory. For Example: *double* d = 11676.2435676542;
- **Void Data Type:** *Void* is an empty data type that has no value. This can be used in functions and pointers.

Type modifiers in C

- In c language Data Type Modifiers are keywords used to change the current properties of data type. Data type modifiers are classified into the following types.
 - Long
 - Short
 - Unsigned
 - signed
- Modifiers are prefixed with basic data types to modify (either increase or decrease) the amount of storage space allocated to a variable. For example, storage space for int data type is 4 bytes for a 32 bit processor. We can increase the range by using long int which is 8 bytes. We can decrease the range by using short int which is 2 bytes.

long

- This can be used to increase the size of the current data type by 2 more bytes, which can be applied on int or double data types. For example int occupy 2 bytes of memory; if we use long with integer variable, then it occupies 4 bytes of memory.



Syntax

long a; —> by default which represent long int

short

- In general int data type occupies different memory spaces for a different operating systems; to allocated fixed memory space a short keyword can be used.

Syntax

short int a; —> occupies 2 bytes of memory space in every operating system

unsigned

- This keyword can be used to make the accepting values of a data type of positive data type.

Syntax

unsigned int a = 100; // right

unsigned int a = -100; // wrong

Signed

- This keyword accepts both negative and positive values and this is the default property or data type modifier for every data type.

int a = 10; // right

int a = -10; // right

signed int a = 10; // right

signed int a = -10; // right

Type	Size (bytes)	Range
int or signed int	2	-32,768 to 32,767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Table 1.5 Size and range of Integer type on 16-bit machine

Type	Size (bytes)	Range
float	4	3.4E - 38 to 3.4E + 38
double	8	1.7E - 308 to 1.7E + 308
long double	10	3.4E - 4932 to 1.1E + 4932

Table 1.6 Size and range of Float type on 16-bit machine

Type	Size (bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

Table 1.7 Size and range of Character type on 16-bit machine

Derived data types

- Derived data types are derived from the collection of primary data types. C supports the following derived data types.
- **Arrays**
 - Array is a collection of variables of same data type that are referenced by a common name.

Syntax: <datatype> <variable name> [Index];

Example: int a[10];

➤ Functions

- A function is a self-contained program segment (block of statements) that carries out some specific, well defined task.

Syntax for function prototype

<return datatype> function name (forma) arg, formal arg2 ... formal arg n);

Syntax for function definition

```
<return type> function name (parameter list)
parameter declarations
{
body of the function;
return (expression);
}
```

Example

```
void swap (int x, int y)
{
int z;
z = x;
x = y;
y = z;
}
```

➤ Pointers

- Pointer is a variable which stores the address of another variable.

Example

```
int *p; // declaration of pointer
int x; // declaration of variable
p=&x; // pointer variable stores the address of x variable.
x=5 ; // x variable assigned with value 5.
```

User defined data types

- The user defined data types enable a program to invent his own data types and define what values it can taken on. Thus these data types can help a programmer to reducing programming errors.
- C supports the following user defined data types.

- **Structures**

A structure is a single entity representing a collection of data items of different data types.

Example

```
struct student
{
    int roll_no;
    char fname[25];
    char branch[15];
    int marks;
} s1;
```

- **Unions**

A union is a data type in 'c' which allows overlay of more than one variable in the same memory area.

Example

```
union emp
{
    char name[20];
    char eno[10];
}
union emp e1;
```

- **Enumerated data type**

A enumerated data type is a set of values represented by identifiers called *enumeration constants*. It is a user-defined data type and the general format of this data type is

enum name {number 1, number 2, ... number n};

In above format enum is a keyword, name is given by the programmer by the identifier rules. number 1, number 2, ... number n are the member of enumerated datatype.

- **Type definition**

“type definition” that allows user to define an identifier that would represent a data type using an existing data type.

Syntax:

```
typedef type identifier;
```

```
typedef <existing_data_type> <new_user_define_data_type>;
```

Example

```
typedef int number;
```

```
typedef long big_number;
```

```
typedef float decimal;
```

```
number visitors = 25;
```

```
big_number population = 12500000;
```

```
decimal radius = 3.5;
```

1.5.6 Constants

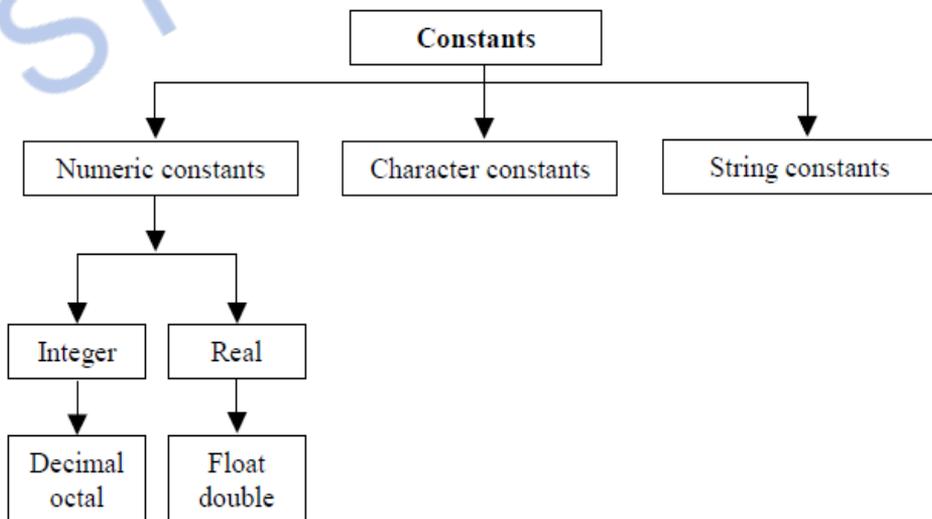


Fig: 1.6 Classification of ‘C’ Constants

1.26 C Programming Fundamentals

- *Constants* are fixed values and they remain unchanged during the execution of the program. The constants are classified as follows:

Integer constants

- It consists of a sequence of digits without any decimal point. Integer constant can be written in three different number systems: decimal, octal and hexadecimal. A *decimal integer constant* can consist of any combination of digits taken from the set 0 through 9.

1. Decimal number – 0 to 9
2. Octal number – 0 to 7
3. Hexadecimal number – 0 to 9, A, B, C, D, E, F

- **Examples**

Decimal number – 10, 145, -89, 067 etc.

Octal number – 037, 0, 057, 0456 etc.

Hexadecimal number – 0x4, 0x9C, 0xA FE etc.

- **Rules for an integer constant**

- It must have at least one digit.
- Decimal point is not allowed.
- It can be either positive or negative.
- If it is negative the sign must be preceded. For positive the sign is not necessary.
- No commas or blank spaces are allowed.
- The allowable range for integer constant is $-32,768$ to $+32,767$

Real Constant

- It is made up of a sequence of numeric digits with presence of a decimal point.
- It is to represent quantities that vary continuously such as distance, height, temperature etc.

- **Example:**

Distance=134.9;

Height=88.10;

➤ **Rules for a real constant**

- It must have one digit.
- It must have decimal point.
- It can be either positive or negative.
- If it is negative the sign must be preceded. For positive the sign is not necessary.
- No commas or blank spaces are allowed.

Character constants

Single Character Constant

- It contains a single character enclosed within a pair of single quote marks.

Example

'd', 'r', '6', '_'

String Constant

- It is a sequence of characters enclosed in double quotes.
- The characters may be letters, numbers, special characters and blank spaces
- At the end of string '\0' is automatically placed.

➤ **Example**

"hai"

"4565"

1.5.7 Variables and Declaration

- *Variables* are identifiers whose value changes during the execution of the program. Variables specify the name and type information. The compiler allocates memory for a particular variable based on the type.
- Variables can be modified using the variable name or address of the variable. The variable name must be chosen in a meaningful way. The declaration of the variable must be done before it can be used in the program.
- The general syntax of the variable declaration is given below.

datatype : var1, var2, ..., varn;

where *datatype : may be any data type*

var1, var2 : variable name separated by a comma

Examples

1. int sum, count;
2. int rollno;
3. float int_rate;
4. double avg, netsal;
5. char char;

Variable declaration with qualifiers

Examples

1. short int number;
2. unsigned int total;
3. long int ser_no;
4. long double volume;

Variable Initialization

- Assigning a relevant value to a variable for the first time in a program is known as *initialization*. Sometimes a variable may be initialized on its declaration itself. Variables can be initialized with a constant value or expression.

Syntax:

datatype variablename = expression;

(or)

datatype variablename = constant;

Example

1. int c = 10, d = c + 5;
2. float rate = 12.5;
3. char ch = 'Y';
4. int count = 0 , sum = 0;
5. float pi = 3.14;

CONSTANT AND VOLATILE VARIABLE

Constant variable

- If we want that the value of a certain variable remain the same or remain unchanged during the execution of a program, then it can be done only by declaring the variable as a constant.

- The keyword `const` is then added before the declaration. It tells the compiler that the variable is a constant. Thus, constant declared variables are protected from modification.

Example

```
const int a = 20;
```

where, **const** is a keyword, **a** is a variable name and **20** is a constant value. The compiler protects the value of 'a' from modification. The user cannot assign any value to a; by `scanf ()` statement the value can be replaced.

Volatile variable

- The *volatile variables* are those variables that are changed at any time by any other external program or the same program. The syntax is as follows.

Example

```
volatile int b;
```

where **volatile** is a keyword and **b** is a variable. If the value of a variable in the current program is to be maintained constant and desired not to be changed by any other external operation, then the declaration of the variable will be as follows;

```
volatile const b = 20;
```

1.6 OPERATORS IN C

Operator: An operator is a symbol that specifies an operation to be performed on operands. Eg: `x= a+b`; where `+` is an operator.

Operands: An operand is an entity on which an operation is to be performed. An operand can be a variable name, a constant, a function call or a macro name.

Eg. `x= a+b`; where `x`, `a`, `b` are the operands.

Expression: An expression is a sequence of operands and operators that specifies the computations of a value. An expression is made up of one or more operands. Eg. `x= a+b`.

1.6.1 Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators

1.30 C Programming Fundamentals

5. Increment and Decrement Operators
6. Conditional Operators (Ternary Operators)
7. Bitwise Operators
8. Special Operators

1.6.1.1 Arithmetic Operators

- Addition, subtraction, multiplication, division and modulo are the arithmetic operations.
- The arithmetic operators are used for numerical calculations between two Constants

Operators	Explanations	Examples
+	Addition	$2 + 2 = 4$
-	Subtraction	$3 - 2 = 1$
*	Multiplication	$5 * 4 = 20$
/	Division	$10 / 2 = 5$
%	Modular Division	$11 \% 2 = 1$

Example:

```
void main()
{
    int a=5, b=4, c;
    c=a-b;
    printf("%d", c);
}
```

- The following table show the division operator on various data types.

Operation	Result	Example
int/int	int	$2/5=0$
real/int	real	$5.0/2=2.5$
int/real	real	$5/2.0=2.5$
real/real	real	$5.0/2.0=2.5$

- Arithmetic operators can be classified as
 - **Unary arithmetic** – it requires only one operand.
Example: +a, -b
 - **Binary arithmetic** – it requires two operands.
Example: a+b, a-b, a/b, a%b
 - **Integer arithmetic** – it requires both operands to be integer type for arithmetic operation.
Example:
a=4, b=3
a+b =4+3 =7
a-b =4-3=1
 - **Floating Point arithmetic** – It requires both operands to be float type for arithmetic operation.
Example:
a=6.5, b=3.5
a+b =6.5+3.5 =10.0
a-b =6.5-3.5=3.0

Program 1.3

```
#include<stdio.h>
#include<conio.h>
void main()
{
int b,c;
int sum, sub, mul;
float div;
clrscr();
printf("enter the value of b,c:");
scanf("%d%d", &b, &c);
sum=b+c;
```

1.32 C Programming Fundamentals

```

sub=b-c;
mul=b*c;
div=b/c;
printf("\n sum=%d,sub=%d,mul=%d,div=%f",sum,sub,mul,div);
getch();
}
    
```

Output:

Enter the value of b,c: 8 4
 sum=12,sub=4,mul=32,div=2

1.6.1.2 Relational Operators

- Relational operators are used to compare two or more operands.
- Operands may be variable, constant or expression

Operators	Descriptions	Example	Return Value
>	Greater than	5>4	1
<	Less than	10<9	0
<=	Less than or equal to	10<=10	1
>=	Greater than or equal to	11>=5	1
==	Equal to	2==3	0
!=	Not equal to	3!=3	0

Syntax

AE1 *operator* AE2

where, AE- Arithmetic Expression or Variable or Value.

- These operators provide the relationship between two expressions.
- If the condition is true it returns a value 1, otherwise it returns 0.
- These operators are used in decision making process. They are generally used in conditional or control statement.

1.6.1.3 Logical Operators

- Logical Operators are used to combine the result of two or more conditions.

- The logical relationship between the two expressions is checked with logical operators.
- After checking the condition, it provides logical true (1) or false (0).

Operators	Descriptions	Example	Return Value
&&	Logical AND	5>3 && 5<10	1
	Logical OR	8>5 8<2	1
!=	Logical NOT	8!=8	0

- **&&** - This operator is usually used in situation where two or more expressions must be true.

Syntax:

(exp1) && (exp2)

- **||** – This is used in situation, where at least one expression is true.

Syntax:

(exp1) || (exp2)

- **!** – This operator reverses the value of the expression it operates on. (i.e.,) it makes a true expression false and false expression true.

Syntax:

!(exp1)

Program 1.4

```

/* Program using Logical operators */
#include<stdio.h>
#include<conio.h>
void main( )
{
clrscr( );
printf("\n Condition : Return values ");
printf("\n 5<=8 && 4>2: %5d",5<=8 && 4>2);
printf("\n 5>=3 || 6<8: %5d",5>=3 || 6<8);

```

1.34 C Programming Fundamentals

```
printf("\n !(7==7): %5d",!(7==7));  
getch( );  
}
```

Output

Condition : Return values

$5 \leq 8 \ \&\& \ 4 > 2 : 1$

$5 >= 3 \ || \ 6 < 8 : 1$

$!(7 == 7) : 0$

1.6.1.4 Assignment Operator

- Assignment Operator are used to assign constant or a value of a variable or an expression to another variable.

Syntax

variable = expression (or) value;

Example

x=10;

x=a+b;

x=y;

Program 1.5

/ Program using Assignment and Short-hand Assignment operators */*

#include <stdio.h>

#include <conio.h>

void main()

{

int a=20,b=10,c=15,d=25,e=34,x=5;

clrscr();

printf("\n Value of a=%d",a);

printf("\n Value of b=%d",b);

a+=x;

```

b-=x;
c*=x;
d/=x;
e%=x;
printf("\n Value of a=%d",a);
printf("\n Value of b=%d",b);
printf("\n Value of c=%d",c);
printf("\n Value of d=%d",d);
printf("\n Value of e=%d",e);
getch();
}

```

Output

```

Value of a = 20
Value of b = 10
Value of a = 25
Value of b = 5
Value of c = 75
Value of d = 5
Value of e = 4

```

1.6.1.5 Increment and Decrement Operators (Unary Operators)

- The ‘++’ adds one to the variable and ‘--’ subtracts one from the variable. These operators are called unary operators.

Operator	Meaning
++X	Pre increment
--X	Pre decrement
X++	Post increment
X--	Post decrement

Pre-increment operator

- This operator increment the value of a variable first and then perform other actions.

Program 1.6

```
#include <stdio.h>
void main()
{
int a,b;
a=10;
b=++a;
printf("a=%d",a);
printf("b=%d",b);
}
output: a=11 b=11
#include <stdio.h>
void main()
{
int a,b;
a=10;
b=--a;
printf("a=%d",a);
printf("b=%d",b);
}
```

Output:

a=9 b=9

Post-increment operator

- This operator perform other actions first and then increment the value of a variable.

Program 1.7

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

```
int a,b;
a=10;
b=a++;
printf("a=%d",a);
printf("b=%d",b);
}
```

Output:

a=11 b=10

Program 1.8

```
#include <stdio.h>
void main()
{
int a,b;
a=10;
b=a--;
printf("a=%d",a);
printf("b=%d",b);
}
```

Output:

a=9 b=10

1.6.1.6 Conditional Operator (or) Ternary Operator

- Conditional operator checks the condition itself and executes the statement depending on the condition.

Syntax

condition?exp1:exp2;

Example

```
void main()
{
int a=5,b=3,big;
```

1.38 C Programming Fundamentals

```
big=a>b?a:b;
printf("big is...%d",big);
}
```

Output

big is...5

1.6.1.7 Bitwise Operators

- Bitwise operators are used to manipulate the data at bit level.
- It operates on integers only.
- It may not be applied to float or real.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

Bitwise AND (&):

- This operator is represented as '&' and operates on two operands of integer type. If both the operands bit is '1' then the result is '1'.

Bitwise OR (|):

- Bitwise OR (|) operator gives the value '1' if either of the operands bit is '1'

Bitwise Exclusive OR (^)

- Bitwise Exclusive OR(^) gives the value '1' if both operands bit are same.

a	b	a b	a&b	a^b	~
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Program 1.9

```

/* Program using One's complement operator */
#include<stdio.h>
#include<conio.h>
void main( )
{
int a;
clrscr( );
printf("\n Enter the value for a : ");
scanf("%d",&a);
printf("\n The One's complement value for a is : %d", ~a);
getch( );
}

```

Output

Enter the value for a: 3
The One's complement value for a is: -4

1.6.1.8 The Special Operator

- C language supports some of the special operators given below.

Operator	Meaning
,	Comma operators
sizeof	Size of operators
& and *	Pointer operators
. and —>	Member selection operators

a) Comma operator(,):

- The comma operator is used to separate the statement elements such as variables, constants or expression etc.,
- This operator is used to link the related expression together.
- Such expression can be evaluated from left to right and the value of right most expression is the value of combined expression.

Example:

```
val=(a=3,b=9,c=77,a+c);
```

Where,

First assigns the value 3 to a

Second assigns the value 9 to b

Third assigns the value 77 to c

Last assigns the value 80.

b) The sizeof() operator:

- The sizeof() is a unary operator that returns the length in bytes of the specified variable and it is very useful to find the bytes occupied by the specified variable in memory.

Syntax:

```
sizeof(var);
```

Example:

```
void main()  
{  
    int a;  
    printf("size of variable a is...%d", sizeof(a));  
}
```

Output:

size of variable a is.....2

c) Pointer operator:

- & : This symbol specifies the address of the variable.
- * : This symbol specifies the value of the variable.

d) Member selection operator:

- . and —>: These symbols are used to access the elements from a structure.

1.7 EXPRESSIONS AND STATEMENTS

1.7.1 Expressions

- An expression represents data item such as variables, constants and are interconnected with operators as per the syntax of the language.
- An expression is evaluated using assignment operators.

Syntax

Variable = expression;

Example: 1

*x=a*b-c;*

- In example 1, the expression evaluated from left to right. After the evaluation of the expression the final value is assigned to the variable from right to left.

Example: 2

a++;

- In example 2, the value of variable *a* is incremented by 1, i.e, this expression is equivalent to *a = a + 1*.

1.7.2 Statements

- A statement is an instruction given to the computer to perform an action. There are three different types of statements in C:
 1. Expression Statements
 2. Compound Statements
 3. Control Statements

Expression Statement

- An expression statement or simple statement consists of an expression followed by a semicolon (;).

Example

a=100;

b=20;

c=a/b;

Compound Statement

- A compound statement also called a block, consists of several individual statements enclosed within a pair of braces { }.

Example

```
{  
    a=3;  
    b=10;  
    c=a+b;  
}
```

Control Statement

- A single statement or a block of statements can be executed depending upon a condition using control statements like if, if-else, etc.

Example

```
a=10;  
if (a>5)  
{  
    b= a+10;  
}
```

1.8 CONDITIONAL STATEMENTS

- The conditional statement requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- In a conditional statement, the flow of execution may be transferred from one part to another part based on the output of the conditional test carried out. It has been further classified into selective and loop constructs. In a selective constructs, the statements are selected for execution based on the output of the conditional test given by an expression. It supports the following constructs such as if-else, if-else-if, nested-if and switch case statement. In loop constructs, the block of statements will be executed repeatedly until the condition is true else the loop will

be terminated. It supports the following constructs such as For, While and Do-while loops.

1.8.1 Conditional Branching Statement

1.8.1.1 Selection Statement

❖ Simple If statement

- The syntax for a simple if statement is

```
if (expression)
{
    block of statements;
}
```

- In this statement, if the expression is true, the block of statements are executed otherwise false and it comes out of the if condition.

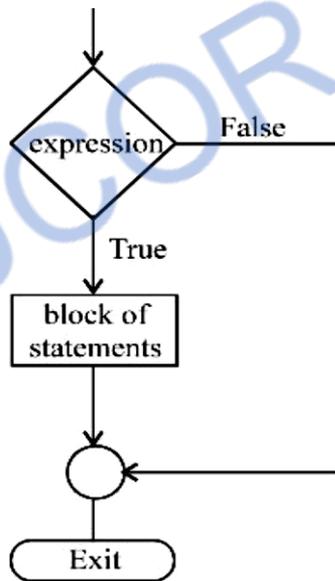


Fig. 1.7 Flowchart for an If statement

Program 1.10

```
/*Program to find the given number is divisible by 2 */
#include<stdio.h>
void main()
```

1.44 C Programming Fundamentals

```
{
int n;
printf("\n Enter the number");
scanf("%d",&n);
if(n%2==0)
{
printf("\n The given number%d is divisible by 2", n);
}
getch();
}
```

Output

Enter the number : 10

The given number 10 is divisible by 2

Program 1.11

```
/* Program to check the given numbers are equal or not */
#include<stdio.h>
#include<conio.h>
void main()
{
int m,n;
clrscr();
printf("\n Enter two numbers:");
scanf("%d %d", &m,&n);
if (m==n)
printf("\n Two numbers are equal");
getch();
}
```

Output

Enter two numbers: 10 10

Two numbers are equal.

❖ If –else statement

- The syntax for the if-else statement is

```
if(expression)
{
    block of statements1;
}
else
{
    block of statements2;
}
```

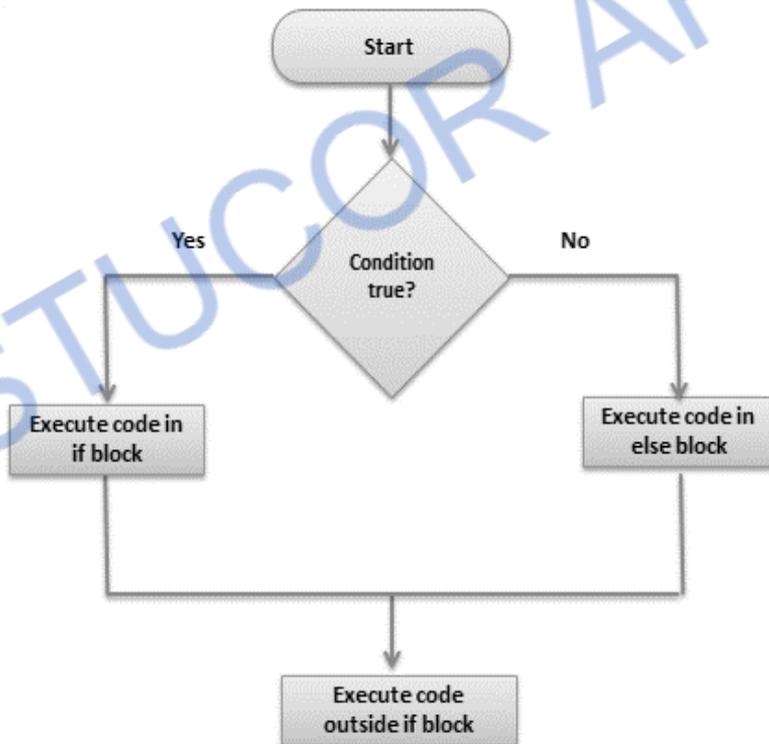


Fig. 1.8 Flowchart for the If-else statement

- In this statement, if the expression is true the block of statements1 will be executed, otherwise the block of statements2 will be executed.

Program 1.12

/* Program to find the given number is positive or negative */

```
#include<stdio.h>
void main()
{
    int n;
    printf("\n Enter the number: ");
    scanf("%d",&n);
    if(n>0)
    {
        printf("\n The given number %d is positive", n);
    }
    else
    {
        printf("\n The given number %d is negative", n);
    }
}
```

Output

```
Enter the number : 5
The given number 5 is positive.
```

Program 1.13

/* Program to find the given number is even or odd */

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

```
printf("\n Enter the number: ");
scanf("%d",&a);
if (a%2==0)
printf("\n %d is an even number",a);
else
printf("%d is an odd number",a);
getch();
}
```

Output

Enter the number: 10
10 is an even number.

❖ Conditional Expression

- The *ternary operator* is used to form a conditional expression. It uses three operands and hence it is called as a *ternary operator*. The syntax for a conditional expression is:

$\langle \text{expression-1} \rangle ? \langle \text{expression-2} \rangle : \langle \text{expression-3} \rangle ;$

- In this method if expression-1 is true then expression-2 is evaluated, otherwise expression-3 is evaluated.

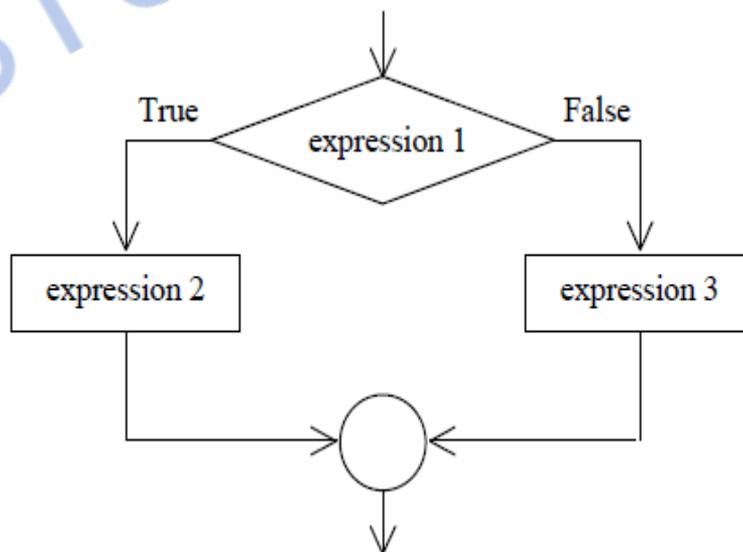


Fig. 1.9 Flowchart for a Conditional expression

Program 1.14

/* Program to find biggest of two given numbers */

```
#include<stdio.h>
void main()
{
int x,y,z;
printf("\n Enter the value of x and y:");
scanf("%d%d",&x,&y);
z = ((x>y)?x:y);
printf("The biggest value is %d",z);
getch();
}
```

Output

Enter the value of x and y: 5 10

The biggest value is 10

❖ If-else-if statement

- The syntax for the if-else-if statement is

```
if(expression1)
{
statements1;
}
else if(expression2)
{
statements2;
}
else if(expression3)
{
statements3;
}
```

```

else
{
statements4;
}

```

- In this statement, if the expression1 is true, statements1 will be executed, otherwise the expression2 is evaluated, if it is true then statements2 is executed, otherwise the expression3 is evaluated, if it is true then statements3 is executed, otherwise statements4 is executed.

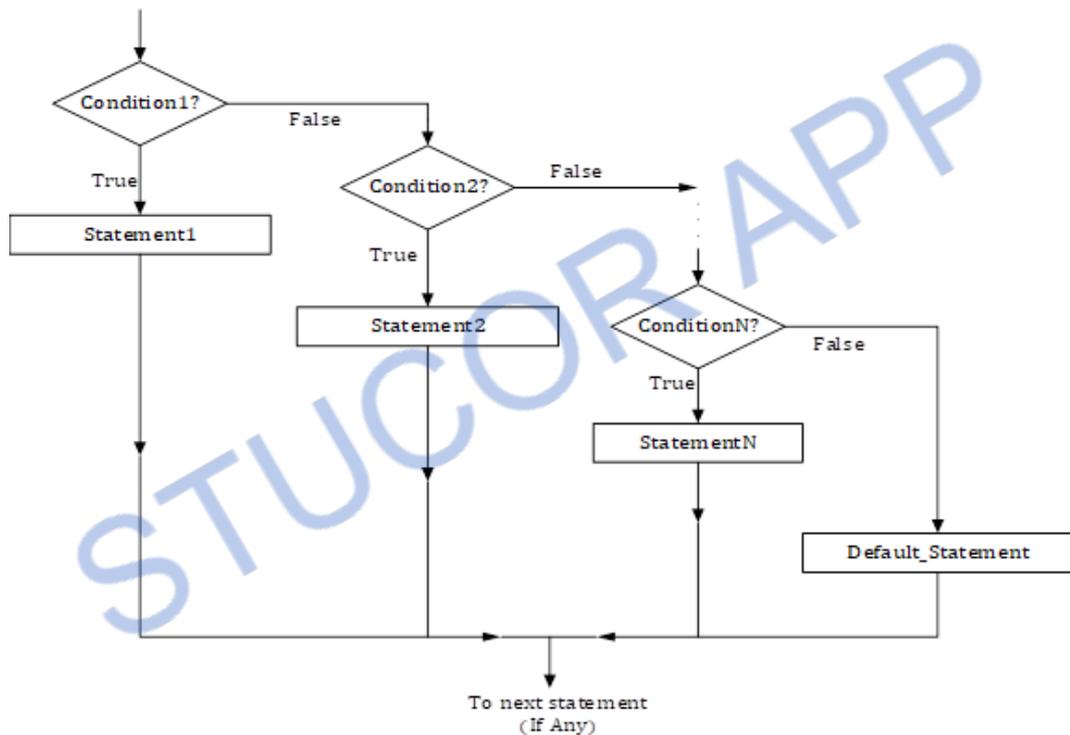


Fig. 1.10 Flowchart for the If-else-if statement

Program 1.15

/* Program to find the student's class for the given average marks using if-elseif*/

```

#include<stdio.h>
void main()
{
int Avg_Mark;

```

1.50 C Programming Fundamentals

```
printf("Enter the Average mark:")
scanf("%d",&Avg_Mark);
if(Avg_Mark>=75)
{
printf("Distinction");
}
elseif((Avg_Mark>=60) && (Avg_Mark<75))
{
printf("First Class");
}
elseif((Avg_Mark>=50) && (Avg_Mark<60))
{
printf("Second Class");
}
elseif((Avg_Mark>=45) && (Avg_Mark<50))
{
printf("Third Class");
}
else
{
printf("Fail");
}
}
```

Output

```
Enter the Average Mark : 65
First Class
```

Program 1.16

Write a program to calculate the gross salary for the conditions given below:

Basic Salary (Rs)	DA (Rs)	HRA (Rs)	Conveyance (Rs)
Basic \geq 5000	110% of basic	20% of basic	500
Basic \geq 3000 && basic $<$ 5000	100% of basic	15% of basic	400
Basic $<$ 3000	90% of basic	10% of basic	300

/* Program to Calculate the gross salary */

```

#include<stdio.h>
#include<conio.h>
void main()
{
float bs, hra, da, cv, ts;
clrscr();
printf("\n Enter Basic salary:");
scanf("%f",&bs);
if(bs>=5000)
{
hra = bs*20/100;
da = bs*110/100;
cv = 500;
}
else if(bs>=3000 && bs<5000)
{
hra = bs*15/100;
da = bs*100/100;
cv = 400;
}
else if(bs<3000)
{

```

1.52 C Programming Fundamentals

```
hra = bs*10/100;
da = bs*90/100;
cv = 300;
}
ts = bs+hra+da+cv;
printf("\nBasic salary: %5.2f",bs);
printf("\nHRA: %5.2f",hra);
printf("\nDA: %5.2f",da);
printf("\nConveyance: %5.2f",cv);
printf("\nGross Salary: %5.2f",ts);
getch();
}
```

Output

Enter basic salary: 5400

Basic salary: 5400

HRA: 1080

DA: 5940

Conveyance: 500

Gross salary: 12920

❖ Nested if statement

- The syntax for the nested if statement is

```
if(expression1)
{
statements1;
}
else
{
if(expression2)
{
```

```

statements2;
}
else
{
statements3;
}
}

```

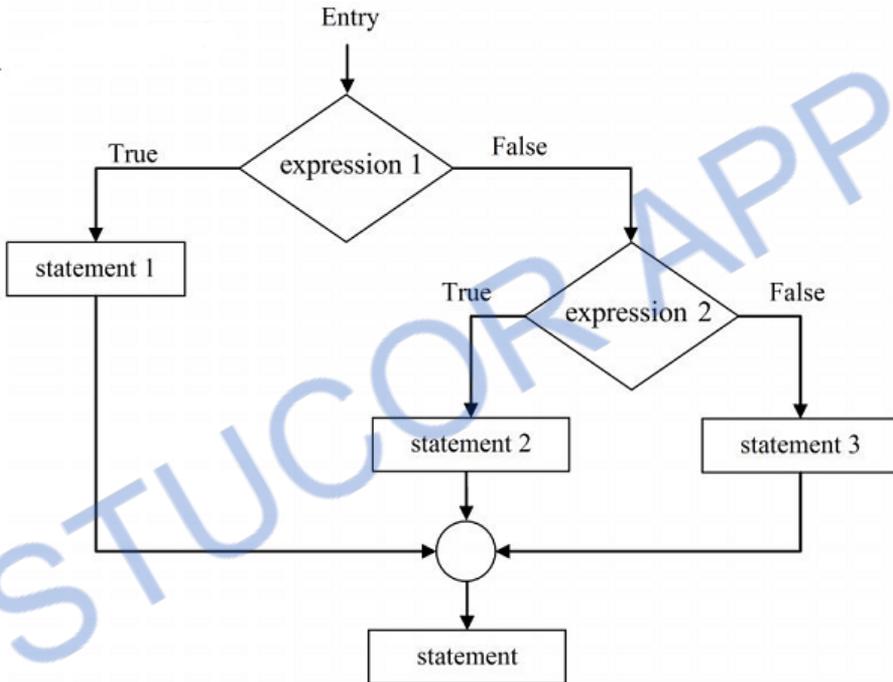


Fig. 1.11 Flowchart for Nested if statement

- In this statement, if expression1 is true, then statement1 is evaluated, otherwise the inner if expression2 is true then statements2 will be executed otherwise inner else statements3 will be executed.

Program 1.17

/* Program to find the biggest of given three numbers */

```

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

```

1.54 C Programming Fundamentals

```
int x,y,z;
printf("\n Enter the three numbers");
scanf("%d%d%d",&x,&y,&z);
if ((x>y) && (x>z))
{
printf("The Biggest number = %d",x);
}
else
{
if(y>z)
{
printf("The Biggest number =%d",y);
}
else
{
printf("The Biggest number =%d",z);
}
}
getch();
}
```

Output

Enter the three numbers: 5 2 8

The Biggest number = 8

❖ Switch () Case Statement

- The switch () case statement is like if statement that allows us to make a decision from a number of choices. The switch statement requires only one argument of any data type, which is checked with a number of case options.
- The switch statement evaluates the expression and then looks for its value among the case constants.

- If the value matches with a case constant, this particular case statement is executed. If not, the default is executed. The general syntax for the switch - case statement is:

```
switch<exprn>
{
  case constant_1:
  {
    statements1;
    break;
  }
  case constant_2:
  {
    statements2;
    break;
  }
  case constant_3:
  {
    statements3;
    break;
  }
  case constant_n:
  {
    statementsn;
    break;
  }
  default:
  {
    default statements;
  }
}
```

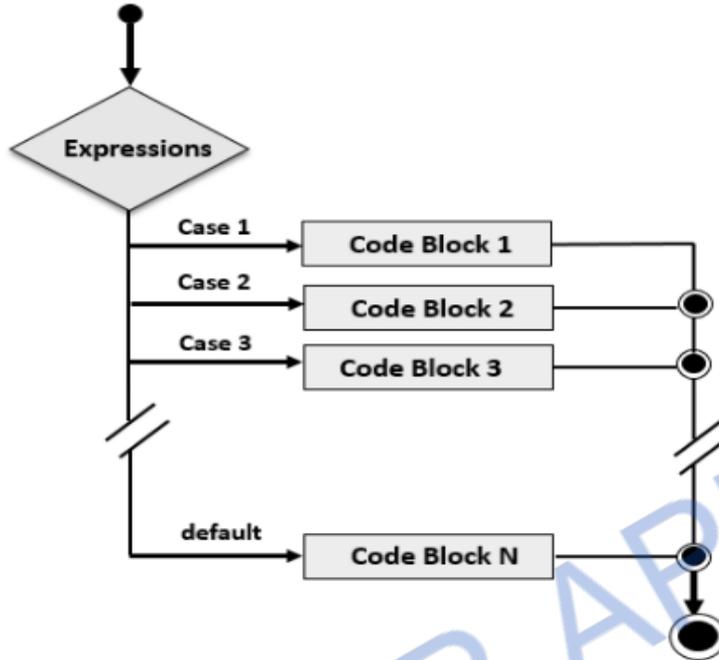


Fig. 1.12 Flowchart for Switch - Case statement

Program 1.18

/ Program to provide multiple functions such as 1. Addition 2. Subtraction 3. Multiplication 4. Division by using switch statements. */*

```

#include<stdio.h>
#include<conio.h>
void main()
{
float c;
int a,b,n;
printf("\n MENU");
printf("\n 1.Addition");
printf("\n 2.Subtraction");
printf("\n 3.Multiplication");
printf("\n 4.Division");
printf("\n 0.Exit");

```

```
printf("\n Enter your choice:");
scanf("%d",&n);
printf("Enter two numbers:");
scanf("%d%d",&a,&b);
switch(n)
{
case 1:
c = a + b;
printf("\n Addition: %d",c);
break;
case 2:
c = a - b;
printf("\n Subtraction: %d",c);
break;
case 3:
c = a * b;
printf("\n Multiplication: %d",c);
break;
case 4:
c = a / b;
printf("\n Division: %d",c);
break;
case 0:
exit();
break;
default:
printf("Invalid choice");
break;
}
```

```
    getch();  
}
```

Output

Menu

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter your choice: 2

Enter two numbers: 40 20

Subtraction: 20

1.8.1.2 Looping Statements

- A loop is defined as a block of statements which are repeatedly executed for certain number of times. The 'C' language supports three types of loop statements.
 1. *for* statement
 2. *while* statement
 3. *do-while* statement

❖ For Loop Statement

- The *for* loop allows to execute a set of instructions until a certain condition is satisfied. Condition may be predefined or open-ended.
- The syntax *for loop* this is follows:

```
for<initial value>;<condition>;<incrementation/decrementation>  
{  
    block of statements;  
}
```

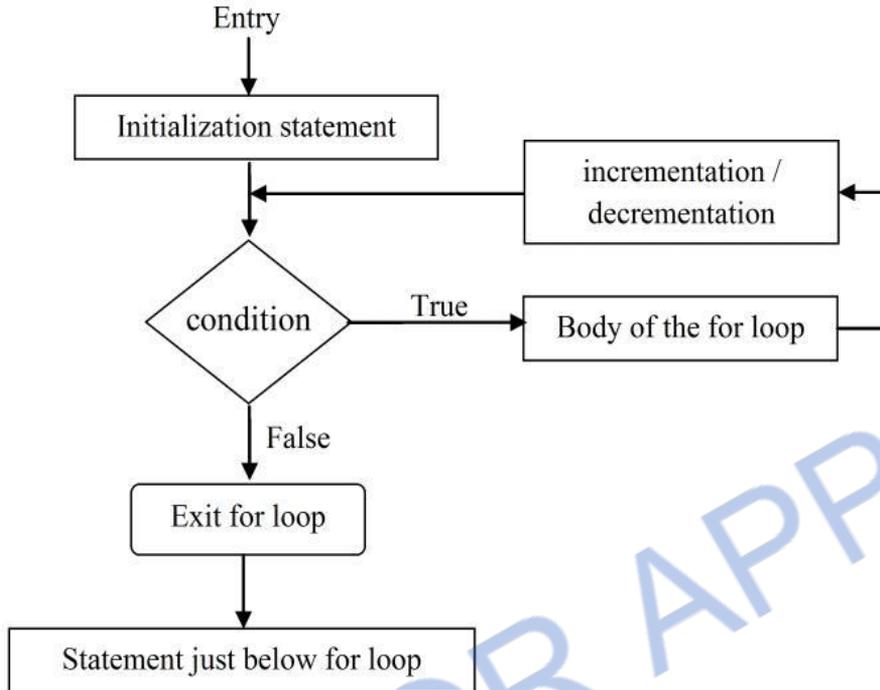


Fig. 1.13 Flowchart of the For loop statement

- Here the initial value means the starting value assigned to the variable and condition in the loop counter to determine whether the loop should continue or not. Incrementation / decrementation is to increment/decrement the loop counter value each time the program segment has been executed.

Program 1.19

/ Program to Generate numbers from 1 to 10 */*

```
#include<stdio.h>  
void main()  
{  
int i,n;  
printf("\n Enter the limit");  
scanf("%d",&n);  
for(i=1;i <=n;i++)  
{
```

```
printf("%d\n",i);  
}  
getch( );  
}
```

Output

Enter the limit: 10

1
2
3
4
5
6
7
8
9
10

❖ While Loop Statement

- The syntax for the *while loop* statement is

```
while(condition)  
{  
    block of statements;  
    incr/decr;  
}
```

- The while loop is often used when the number of times the loop is to be executed is not known in advance. A sequence of statements are executed until some condition is satisfied.
- When the condition specified inside the parenthesis the while loop is satisfied, the control is transferred to the statements inside the loop and executes the body of the loop. The loop continues until the condition is violated. The while tests the condition before each iteration.

- If the condition initially fails the loop is skipped entirely even in the first iteration itself. It is otherwise called as entry controlled loop.

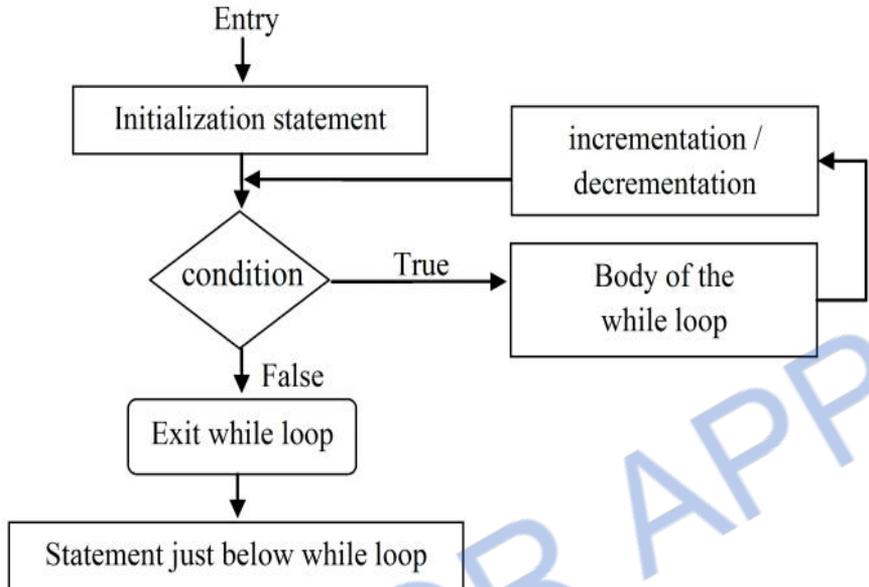


Fig. 1.14 Flowchart of the While loop

Program 1.20

/ Program to Generate the Even numbers to a given limit*/*

```

#include<stdio.h>
#include<conio.h>
void main()
{
int n,i;
printf("\n Enter the limit:");
scanf("%d",&n);
i=1;
while(i<=n)
{
if(i%2==0)

```

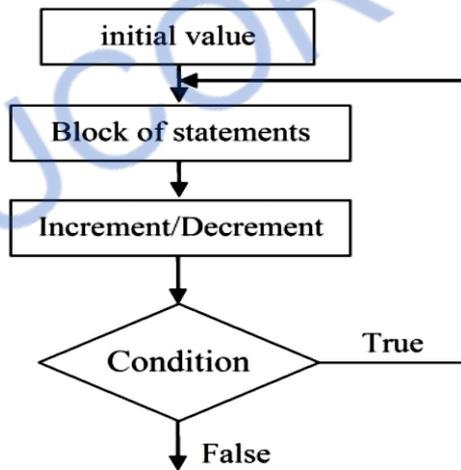
```
printf(“%d\t”,i);  
i++;  
}  
getch( );  
}
```

Output

```
Enter the limit: 10  
2 4 6 8 10
```

❖ Do While Statement

- The *do while loop* varies from the while loop in the checking condition. The condition of the loop is not tested until the body of the loop has been executed once. If the condition is false, after the first loop iteration the loop terminates. The statements are executed atleast once even if the condition fails for the first time itself. It is otherwise called as exit control loop.



Exit from do-while loop

Fig. 1.15 Flowchart of the Do while loop

- The syntax for the do while loop is

```
do  
{  
statements;
```

```
    }  
    while(condition);
```

Program 1.21

/ Program to check the given number is palindrome or not */*

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
int n,t,s,r;  
clrscr();  
printf("\n Enter the number");  
scanf("%d",&n);  
t=n;  
s=0;  
do  
{  
r=n%10;  
s=(s*10)+r;  
n=n/10;  
} while(n>0);  
if(t==s)  
printf("%d is a palindrome",t);  
else  
printf("%d is not a palindrome", t);  
getch( );  
}
```

Output

```
Enter the number 242  
242 is a palindrome
```

1.9 FUNCTIONS

Introduction

- A function is a sub program which contains a set of instructions that are used to perform specified tasks
- A function is used to provide modularity to the software. By using function can divide complex task into manageable tasks. The function can also help to avoid duplication of work.

Advantages of Functions

- ✓ Code reusability
- ✓ Better readability
- ✓ Reduction in code redundancy
- ✓ Easy to debug & test.

How Does Function Work?

- Once a function is called, it takes some data from the calling function and returns back some value to the called function.
- Whenever function is called control passes to the called function and working of the calling function is temporarily stopped, when the execution of the called function is completed then a control return back to the calling function and executes the next statement.
- The function operates on formal and actual arguments and send back the result to the calling function using return() statement.

Types of Functions

Functions are classified into two types

- a) User defined functions
- b) Predefined functions or Library functions or Built-in functions

a) User-defined functions

- User-defined functions are defined by the user at the time of writing a program.

Example: sum(), square()

b) Library functions [Built-in functions]

- Library functions are predefined functions. These functions are already developed by someone and are available to the user for use.

Example: printf(), scanf()

Terminologies used in functions

- A function f that uses another function g is known as the calling function, and g is known as the called function.
- The inputs that a function takes are known as arguments.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass **parameters** to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- **Function declaration** is a declaration statement that identifies a function's name, a list of arguments that it accepts, and the type of data it returns.
- **Function definition** consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

1.9.1 Function prototype

1.9.1.1 User Defined Function

- The function defined by the users according to their requirements is called user defined functions. The users can modify the function according to their requirement.

Need For user Defined Functions

- While it is possible to write any complex program under the main () function and it leads to a number of problems, such as
- The program becomes too large and complex
- The users cannot go through at a glance.
- The task of debugging, testing and maintenance becomes difficult.

Advantages of user Defined Functions

- The length of the source program can be reduced by dividing it into the smaller functions.
- By using functions it is very easy to locate and debug an error.

1.66 C Programming Fundamentals

- The user defined function can be used in many other source programs whenever necessary.
- Functions avoid coding of repeated programming of the similar instructions.
- Functions enable a programmer to build a customized library of repeatedly used routines.
- Functions facilitate top-down programming approach.

1.9.1.2 Elements of user Defined Functions

- In order to write an efficient user defined function, the programmer must be familiar with the following three elements.
 - Function declaration
 - Function definition
 - Function call

Function Declaration

- A function declaration is defined as a prototype of a function which consists of the functions return type, function name and arguments list
- It is always terminated with semicolon (;)
- Function prototypes can be classified into four types
 - a) Function with no arguments and no return values
 - b) Function with arguments and no return values
 - c) Function with arguments and with return values
 - d) Function with no arguments and with return values

Syntax

return_type function_name (parameter_list);

Where,

return_type can be primitive or non-primitive data type

function_name can be any user specified name

parameter_list can consist of any number of parameter of any type

Example

void add(void);

```
void add(int,int);
```

```
int add(int,int);
```

```
int add(void);
```

a) Function with No Arguments and No Return Values

- In this prototype, no data transfer takes place between the calling function and the called function. (i.e) the called program does not receive any data from the calling program and does not send back any value to the calling program.

Syntax:

```
void function_name(void);  
void main()  
{  
.....  
function_name();  
.....  
}  
void function_name(void)  
{  
.....  
.....  
}
```

Program 1.22

*/*Implementation of function with no return type and no argument list*/*

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

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

```
void add(void); //function declaration with no return type and no  
arguments list
```

```
void main()
```

```
{
```

```
add(); //function call
```

```
}  
void add()  
{  
    int a,b,c;  
    printf("Enter the two numbers . . . ");  
    scanf("%d %d",&a,&b);  
    c=a+b;  
    printf("sum is . . . %d",c);  
}
```

Output:

```
Enter the two numbers . . . 10 20  
Sum is . . . 30
```

b) Function with Arguments and No Return Values

- In this prototype, data is transferred from calling function to called function. i.e the called program receives some data from the calling program and does not send back any values to calling program.

Syntax:

```
void function_name(arguments_list);  
void main()  
{  
    .....  
    function_name(argument_list);  
    .....  
}  
void function_name(arguments_list)  
{  
    //function body  
}
```

Program 1.23

*/*Implementation of function with no return type and with argument list*/*

```
#include<stdio.h>
#include<conio.h>
int add(int,int); //function declaration with no return type and with arguments
list
void main()
{
    int a,b;
    clrscr();
    printf("enter the two values: ");
    scanf(" %d %d",&a,&b);
    add(a,b); /*calling a function with arguments */
    getch();
}
void add(int x,int y)
{
    int z;
    z=x+y;
    printf("Sum is . . . . %d",z);
}
```

Output:

Enter two values: 10 20

Sum is 30

c) Function with arguments and With Return Values

- In this prototype, data is transferred between calling function and called function.(i.e) the called program receives some data from the calling program and send back a return value to the calling program.
- Value received from a function can be further used in rest of the program.

Syntax:

```
return_type function_name(arguments_list);  
void main()  
{  
.....  
variable_name=function_name(argument_list);  
.....  
}  
return_type function_name(arguments_list)  
{  
//function body  
}
```

Program 1.24

```
#include<stdio.h>  
#include<conio.h>  
int add(int,int); //function declaration with return type and with arguments list  
void main()  
{  
    int a,b,c;  
    clrscr();  
    printf("enter the two numbers: ");  
    scanf(" %d %d",&a,&b);  
    c=add(a,b); /*calling function with arguments*/  
    printf("sum is . . . %d ", c);  
    getch();  
}  
int add(int x,int y)  
{
```

```
int z;  
z=x+y;  
return(z); /*returning result to calling function*/  
}
```

Output:

Enter the two numbers: 10 20

sum is . . 30

d) Function with No Arguments and with Return Values

- In this prototype, the calling program cannot pass any arguments to the called program. i.e) program may send some return values to the calling program.

Syntax:

```
return_type function_name(void);  
void main()  
{  
.....  
variable_name=function_name();  
.....  
}  
return_type function_name(void)  
{  
//function body  
}
```

Program 1.25

```
/*Implementation of function with return type and no argument list*/  
#include<stdio.h>  
#include<conio.h>  
int add(void); //function declaration with no return type and no arguments list  
void main()  
{
```

1.72 C Programming Fundamentals

```
    int c;
    c=add(); //function call
    printf("sum is . . . %d",c);
}
int add(void)
{
    int a,b,c;
    printf("Enter the two numbers . . . ");
    scanf("%d %d",&a,&b);
    c=a+b;
    return(c);
}
```

Function Definition

- It is the process of specifying and establishing the user defined function by specifying all of its elements and characteristics.
- When a function is defined, space is allocated for that function in the memory.
- A function definition comprises of two parts:
 - ✓ Function header
 - ✓ Function Body

Syntax

```
return_type function_name(argument_list)
{
    //function body
}
```

Example

```
int add(int x, int y)
{
    int z;
    z=x+y;
```

```
return(z);  
}
```

Program 1.26

```
#include<stdio.h>  
// function prototype, also called function declaration  
float square ( float x );  
// main function, program starts from here  
int main( )  
{  
    float m, n ;  
    printf ( "\nEnter some number for finding square \n");  
    scanf ( "%f", &m );  
    // function call  
    n = square ( m );  
    printf ( "\nSquare of the given number %f is %f",m,n );  
}  
float square ( float x ) // function definition  
{  
    float p ;  
    p = x * x ;  
    return ( p );  
}
```

Output

```
Enter some number for finding square  
2  
Square of the given number 2.000000 is 4.000000
```

Function Call

- The function can be called by simply specifying the name of the function, return value and parameters if presence.

1.74 C Programming Fundamentals

- The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Syntax

```
function_name();  
function_name(parameter);  
variable_name=function_name(parameter);  
variable_name=function_name();
```

Example

```
add();  
add(a,b);  
c=add(a,b);  
c=add;
```

- There are two ways that a C function can be called from a program. They are,
 - a) Call by value
 - b) Call by reference

a) **Function-Call by value**

- In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

b) **Function-Call by Reference**

- Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

Program 1.27

//Example program for Actual Parameter and Formal Parameters

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

```
int sum(int a, int b)
{
    int c=a+b;
    return c;
}
int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);
    return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

Program 1.28

//Example of Function call by Value

```
#include <stdio.h>
int increment(int var)
{
    var = var+1;
    return var;
}
int main()
{
    int num1=20;
    int num2 = increment(num1);
```

1.76 C Programming Fundamentals

```
printf("num1 value is: %d", num1);
printf("\nnum2 value is: %d", num2);
return 0;
}
```

Output

```
num1 value is: 20
num2 value is: 21
```

Program 1.29

//Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
    int tempnum ;
    /*Copying var1 value into temporary variable */
    tempnum = var1;
    /* Copying var2 value into var1*/
    var1 = var2;
    /*Copying temporary variable value into var2 */
    var2 = tempnum;
}
int main( )
{
    int num1 = 35, num2 = 45;
    printf("Before swapping: %d, %d", num1, num2);
    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output

Before swapping: 35, 45

After swapping: 45, 35

Program 1.30**Example 2: Function Call by Reference – Swapping numbers**

```
#include
void swapnum ( int *var1, int *var2 )
{
    int tempnum ;
    tempnum = *var1;
    *var1 = *var2;
    *var2 = tempnum;
}
int main( )
{
    int num1 = 35, num2 = 45;
    printf("Before swapping:");
    printf("\nnum1 value is %d", num1);
    printf("\nnum2 value is %d", num2);
    /*calling swap function*/
    swapnum( &num1, &num2 );
    printf("\nAfter swapping:");
    printf("\nnum1 value is %d", num1);
    printf("\nnum2 value is %d", num2);
    return 0;
}
```

Output

Before swapping:

num1 value is 35

num2 value is 45

After swapping:

num1 value is 45

num2 value is 35

1.10 RECURSIVE FUNCTIONS

- Recursion is the process of calling the same function again and again until some condition is satisfied.
- This process is used for repetitive computation.

Syntax:

```
function_name()
{
    function_name();
}
```

Types of Recursion

- a) Direct Recursion
- b) Indirect Recursion

a) Direct Recursion

- A function is directly recursive if it calls itself.

```
Functionname1( )
{
    ....
    Functionname1 ( );    // call to itself
    ....
}
```

Example

- A function is said to be directly recursive if it explicitly calls itself. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func (int n)
```

```
{  
  if (n == 0)  
    return n;  
  else  
    return (Func (n-1));  
}
```

b) Indirect Recursion

- Function calls another function, which in turn calls the original function.

```
Functionname1 ()  
{  
  ...  
  Functionname2 ();  
  ...  
}  
Functionname1 ()  
{  
  ...  
  Functionname1 (); // function (Functionname2) calls (Functionname1)  
  ...  
}
```

Example

- A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it. These two functions are indirectly recursive as they both call each other.

```
int Funcl (int n)  
{  
  if (n == 0)  
    return n;  
  else  
    return Func2(n);  
}
```

```
    }  
    int Func2(int x)  
    {  
        return Func1(x-1);  
    }
```

Program 1.31

*/*C program to find factorial of a given number using recursion*/*

```
#include< stdio.h>  
#include<conio.h>  
void main()  
{  
    int fact(int);  
    int num,f;  
    clrscr();  
    printf("enter the number");  
    scanf("%d",&num);  
    f=fact(num);  
    printf(" the factorial of%d= %d", num, f);  
}  
int fact(int x)  
{  
    int f;  
    if(x==1)  
        return(1);  
    else  
        f=x*fact(x-1); //recursive function call  
    return (f);  
}
```

Output:

Enter the number 5
 The factorial of 5=120

1.11 ARRAYS

Introduction to Arrays

- An Array is a collection of similar data elements
- These data elements have the same data type
- The elements of the array are stored in consecutive memory locations and are referenced by an index

Definition

- An array is a data structure that is used to store data of the same type. The position of an element is specified with an integer value known as index or subscript.

Example

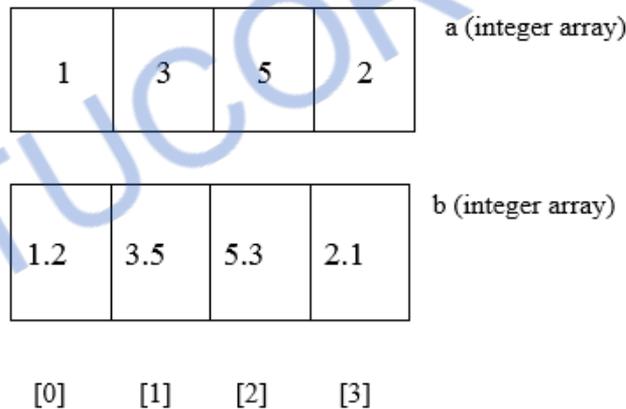


Fig. 1.16 Array Structure

Characteristics

- All the elements of an array share a common name called as array name
- The individual elements of an array are referred based on their position
- The array index in c starts with 0

Advantages of C array

- **Code Optimization** : Less code to access the data

- **Easy to traverse data** : By using the for loop, we can retrieve the elements of an array easily
- **Easy to sort data**: To sort the elements of array, we need a few lines of code only
- **Random Access** : We can access any element randomly using the array

Disadvantages of array

- **Fixed Size**: Whatever size, we define at the time of declaration of array, we can't exceed the limit. So it doesn't grow the size dynamically like Linked List

Classifications

- In general arrays are classified as:
- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

1.11.1 Declaration of an Array

Array has to be declared before using it in C program. Declaring array means specifying three things.

Data_type	Data Type of Each Element of the array
Array_name	Valid variable name
Size	Dimensions of the Array

Arrays are declared using the following syntax:

type name[size]

Here the type can be either int, float, double, char or any other valid data type. The number within the brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array.

- Example:**
- i) `int marks[10]`
 - ii) `int a[5]={10,20,5,56,100}`

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type occupies 2 bytes for each element and for float occupies 4 bytes for each element etc. The size of the array operates the number of elements that can be stored in an array.

1.11.2 Initialization of arrays

Elements of the array can also be initialized at the time of declaration as in the case of every other variable. When an array is initialized, we need to provide a value for every element in the array. Arrays are initialized using the following syntax:

```
type array_name [size] = { list of values};
```

The values are written with curly brackets and every value is separated by a comma. It is a compiler error to specify more number of values than the number of elements in the array.

Example: `int marks [5] = {90, 92, 78, 82, 58};`

1.12 ONE DIMENSIONAL ARRAY

- It is also known as single-dimensional arrays or linear array or vectors
- It consists of fixed number of elements of same type
- Elements can be accessed by using a single subscript

Example

a	1	3	5	2	
	[0]	[1]	[2]	[3]	← subscripts or indices

1.12.1 Declaration of Single Dimensional Array

- An array must be declared before being used. Declaring an array means specifying three things.
 1. Data type
 2. Name
 3. Size

Syntax

```
datatype arrayname [array size];
```

Example

```
int a[4];      // a is an array of 4 integers
char b[6];    // b is an array of 6 characters
```

1.12.2 Initialization of single dimensional array

- Elements of an array can also be initialized. After declaration, the array elements must be initialized otherwise they hold garbage value. An array can be initialized at compile time or at run time.
- Elements of an array can be initialized by using an initialization list. An initialization list is a comma separated list of initializers enclosed within braces.

Example

1. `int a[3]={1,3,4};`
2. `int i[5] = {1, 2, 3, 4, 5};`
3. `float a[5]={1.1, 2.3, 5.5, 6.7, 7.0};`
4. `int b[]={1,1,2,2};`

- In the fourth example the size has been omitted (it can be) and have been declared as an array with 4 elements having 1, 1, 2 and 2 as initial values.
- Character arrays that hold strings allow a shortcut initialization of the form:

char array_name[size]="string"

For example,

char mess[]={'w','e','l','c','o','m','e'};

- If the number of initializers in the list is less than array size, the leading array locations gets initialized with the given values. The rest of the array locations gets initialized to

- 0 - for int array
- 0.0 - for float array
- \0 - for character array

Example

`int a[2]={1};`

a

1	0
---	---

`char b[5]={'A','r','r','\0','\0'};`

b

'A'	'r'	'r'	'\0'	'\0'
-----	-----	-----	------	------

Example Programs**Program 1.32****/*Program to find the maximum number in an array */**

```
#include<stdio.h>
void main( )
{
    int a[5], i, max;
    printf("Enter 5 numbers one by one \n");
    for(i=0;i<5;i++)
    {
        scanf("%d", & a[i]);
    }
    max=a[0];
    for(i=1;i<5;i++)
    {
        if (max<a[i])
            max =a[i];
    }
    printf("\n The maximum number in the array is %d",max);
    getch( );
}
```

Output:

```
Enter 5 numbers one by one
5 7 3 6 4
The maximum number in the array is 7
```

Program 1.33**/*Program for reversing an array*/**

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

```
{
    int a[10], i;
    int n;
    printf("Enter the maximum number of elements\n");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Array in the reverse order\n");
    for(i=n-1; i>=0; i--)
    {
        printf("%d\t", a[i]);
    }
    getch( );
}
```

Output

```
Enter the maximum number of elements
5 11 12 13 14 15
Array in the reverse order
15 14 13 12 11
```

Program 1.34

/* Program to calculate sum of array content */

```
#include<stdio.h>
void main( )
{
    int a[20], n, i, sum = 0;
    print f("\n Enter the size of the array:");
```

```

scanf("%d", &n)
printf("\n Enter the %d numbers one by one:");
for (i=0; i<n; i++)
{
    scanf("%d", &a[i]);
    sum = sum + a[i];
}
printf("The sum of array content = %d", sum);
getch( );
}

```

Output

Enter the size of the array: 5

Enter the 5 number one by one:

10 20 30 40 50

The sum oif the array content = 150

1.13 MULTI-DIMENSIONAL ARRAY

- A multi-dimensional array is an array that has more than one dimension. It is an array of arrays; an array that has multiple levels. The simplest multi-dimensional array is the 2D array, or two-dimensional array and 3D or three-dimensional array.

1.13.1 Two Dimensional Array

- A two dimensional array is an array of one dimensional arrays and can be visualized as a plane that has rows and columns.
- The elements can be accessed by using two subscripts, row subscript (row number), column subscript (column number).
- It is also known as matrix.
- A single dimensional array can store a list of values, whereas two dimensional array can store a table of values.

Example

a[3][5]

1	2	3	6	7
9	10	5	0	4
3	1	2	1	6

Declaration

datatype arrayname [row size][column size]
--

Example: int a [2][3]; //a is an integer array of 2 rows and 3 columns

Number of elements=2*3=6

Initialization

1. By using an initialization list, 2D array can be initialized.

e.g. int a[2][3] = {1,4,6,2}

a

1	4	6
2	0	0

2. The initializers in the list can be braced row wise.

e.g int a[2][3] = {{1,4,6} , {2}};

Program 1.35

/ Example for two dimensional array handling */*

#include <stdio.h>

void main()

{

int a[10][10],i,j,sum,d,n1,n,rowsum,colsum,diasum;

printf("Enter order[row][col] of the matrix\n");

scanf("%d %d",&n,&n1);

*printf("Enter %d elements\n",n1*n);*

```
for(i=0;i<n;i++)
for(j=0;j<n1;j++)
scanf("%d",&a[i][j]);
/* Program module to sum all elements */
sum=0;
for(i=0;i<n;i++)
for(j=0;j<n1;j++)
sum+=a[i][j];
printf("Sum of all elements%d\n",sum);
/* Program to module to sum row wise */
for(i=0;i<n;i++)
{
    rowsum=0;
    for(j=0;j<n1;j++)
    {
        rowsum+=a[i][j];
        printf("row no = %d sum = %d\n",i,rowsum);
    }
}
/* Program module to sum colwise */
for(i=0;i<n;i++)
{
    colsum=0;
    for(j=0;j<n1;j++)
    colsum+=a[j][i];
    printf("col no=%d sum=%d\n ",i,colsum);
}
/* Program module to sum principle diagonal */
```

1.90 C Programming Fundamentals

```
    diasum=0;
    for(i=0;i<n;i++)
    for(j=0;j<n1;j++)
    if(i==j) diasum+=a[i][j];
    printf("Principle diagonal sum %d\n",diasum);
    /* Program module to sum off diagonal */
    diasum=0;
    for(i=0;i<n;i++)
    {
        j= -n1;
        diasum +=a[i][j];
    }
    printf("Off diagonal sum%d\n",diasum);
}
```

Output

Enter order [row][col] of the matrix

3 3

Enter 9 elements

1 2 3 4 5 6 7 8 9

Sum of all elements 45

row no = 0 sum = 6

row no = 1 sum = 15

row no = 2 sum = 24

col no = 0 sum = 12

col no = 1 sum = 15

col no = 2 sum = 18

Principle diagonal sum 15

Off diagonal sum 15

1.13.2 Three-Dimensional Arrays

Initialization of a 3d array

Initialize a three-dimensional array in a similar way to a two-dimensional array.

Example

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Program 1.36

Write a C Program to store and print 12 values entered by the user

```
#include <stdio.h>
int main()
{
    int test[2][3][2];
    printf("Enter 12 values: \n");
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                scanf("%d", &test[i][j][k]);
            }
        }
    }
    // Printing values with the proper index.
    printf("\nDisplaying values:\n");
    for (int i = 0; i < 2; ++i)
    {
```

```
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 2; ++k)
        {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
    }
}
return 0;
}
```

Output

Enter 12 values:

1

2

3

4

5

6

7

8

9

10

11

12

Displaying Values:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 10

test[1][2][0] = 11

test[1][2][1] = 12

STUCOR APP

REVIEW QUESTIONS

PART-A

1. List down the Primary Data Types in C

- **Integer** – We use these for storing various whole numbers, such as 5, 8, 67, 2390, etc.
- **Character** – It refers to all ASCII character sets as well as the single alphabets, such as 'x', 'Y', etc.
- **Double** – These include all large types of numeric values that do not come under either floating-point data type or integer data type.
- **Floating-point** – These refer to all the real number values or decimal points, such as 40.1, 820.673, 5.9, etc.
- **Void** – This term refers to no values at all. We mostly use this data type when defining the functions in a program.

2. What is Variable?

- ✓ Variables are containers for storing data values.
- ✓ Its value can be changed, and it can be reused many times.
- ✓ Syntax for creating variables
 - type variableName = value;
 - Example: int a = 5;

3. What is Operator?

- ✓ An operator is a special symbol that tells the compiler to perform specific mathematical or logical operations.
- ✓ Operators in programming languages are taken from mathematics.
- ✓ C language supports a rich set of built-in operators.

4. List the types of operators supported in C

- ✓ Arithmetic operators
- ✓ Relational operators
- ✓ Logical operators
- ✓ Bitwise operators

- ✓ Assignment operators
- ✓ Type Information Operators(Special operators)

5. What is Ternary operators or Conditional operators?

- ✓ Ternary operators is a conditional operator with symbols? and :
- ✓ **Syntax:** variable = exp1 ? exp2 : exp3
- ✓ If the exp1 is true variable takes value of exp2. If the exp2 is false, variable takes the value of exp3.

6. What is an Operator and Operand?

- ✓ An operator is a symbol that specifies an operation to be performed on operands.
- ✓ Example: *, +, -, / are called arithmetic operators.
- ✓ The data items that operators act upon are called operands.

7. What is type casting?

- ✓ Type casting is the process of converting the value of an expression to a particular data type.
- ✓ Example: int x,y.
c = (float) x/y; where a and y are defined as integers. Then the result of x/y is converted into float.

8. What is the difference between while loop and do while loop?

while	do while
In the while loop the condition is first executed.	In the do...while loop first the statement is executed and then the condition is checked.
If the condition is true, then it executes the body of the loop. When the condition is false it comes of the loop.	The do...while loop will execute at least one time even though the condition is false at the very first time.

9. What is the difference between ++a and a++?

- ✓ ++a means do the increment before the operation (pre increment) a++ means do the increment after the operation (post increment)

10. What is a Function?

- ✓ A function is a block of code which only runs when it is called.
- ✓ It performs a specific task.

11. What is meant by Recursive function?

- ✓ If a function calls itself again and again, then that function is called Recursive function.
- ✓ The syntax for recursive function is:

```
function recurse() {  
    // function code  
    recurse();  
    // function code  
}  
recurse();
```

12. Write short notes about main() function in 'C' program.

Every C program must have main () function.

- ✓ All functions in C, has to end with '(')' parenthesis.
- ✓ It is a starting point of all 'C' programs.
- ✓ The program execution starts from the opening brace '{' and ends with closing brace '}', within which executable part of the program exists.

13. Give the syntax for the 'for' loop statement

```
for (Initialize counter; Test condition; Increment / Decrement)  
{  
    statements;  
}
```

14. What is an Array?

- ✓ An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations.
 - **finite** means data range must be defined.
 - **ordered** means data must be stored in continuous memory addresses.

➤ **homogenous** means data must be of similar data type.

✓ For example: if you want to store marks of 50 students, you can create an array for it.

• `int marks[50];`

15. What are the different types of arrays available in C.

✓ One-dimensional arrays

✓ Multidimensional arrays

16. Write short notes on One-dimensional arrays.

✓ A One-Dimensional Array in C programming is a special type of variable that can store multiple values of only a single data type such as int, float, double, char etc.

✓ The syntax of declaring Two-dimensional arrays is:

➤ `datatype array name [size]`

✓ Example

For example, `int a[5]`

17. Write short notes on Two-dimensional arrays.

✓ A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form.

✓ The general form of declaring Two-dimensional arrays is:

➤ `data_type array_name[x][y];`

✓ Example

`int x[10][20];`

18. What are the key features in the C programming language?

✓ **Portability:** It is a platform-independent language.

✓ **Modularity:** Possibility to break down large programs into small modules.

✓ **Flexibility:** The possibility of a programmer to control the language.

✓ **Speed:** C comes with support for system programming and hence it compiles and executes with high speed when compared with other high-level languages.

✓ **Extensibility:** Possibility to add new features by the programmer.

19. What is a nested loop?

- ✓ A loop that runs within another loop is referred to as a nested loop. The first loop is called the Outer Loop and the inside loop is called the Inner Loop. The inner loop executes the number of times defined in an outer loop.

20. What are the modifiers available in C programming language?

- ✓ Short
- ✓ Long
- ✓ Signed
- ✓ Unsigned
- ✓ long long

21. What is the explanation for prototype function in C?

- ✓ Prototype function is a declaration of a function with the following information to the compiler.
 - Name of the function.
 - The return type of the function.
 - Parameters list of the function.
- ✓ Example
 - `int sum(int,int);`

22. What do you mean by the Scope of the variable?

- ✓ Scope of the variable can be defined as the part of the code area where the variables declared in the program can be accessed directly. In C, all identifiers are lexically (or statically) scoped.

23. Can a C program be compiled or executed in the absence of a main()?

- ✓ The program will be compiled but will not be executed. To execute any C program, main() is required.

24. What is the main difference between the Compiler and the Interpreter?

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.

Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

PART-B

1. Explain the different types of operators with neat examples.
2. Illustrate the different conditional statements available in C with syntax and examples
3. Explain the looping statements with neat examples.
4. What is an Array? Explain Single and Multi-Dimensional arrays with neat examples.
5. Write a C program for Matrix Multiplication with a 3*3 matrix.
6. Create a C program for Matrix Addition.
7. Write a C program to calculate the total, average and grade for 50 Students.
8. Write a C program to calculate the factorial of a given number.
9. Write a C program to check whether a given number is odd or even.
10. Write a C program to check whether a given number is prime or not.
11. Write a C program to check whether a given number is a palindrome or not.
12. Write a C program to check whether a given number is a Armstrong number or not.

II

PROGRAMMING - ADVANCED FEATURES

Structures – Union – Enumerated Data Types – Pointers: Pointers to Variables, Arrays and Functions – File Handling – Preprocessor Directives.

2.1 INTRODUCTION

- Structures, unions and enumerations are known as user defined data types.
- These data types are used to create a flexible new data type.
- Structure can be used for the storage of different data types. The similarity between structure and array is both contain a finite number of elements.
- Union is similar to structures in all aspects except the manner in which their constituent elements are stored.
- In structures, separate memory is allocated to each element, while in unions all the elements are share the same memory.
- Enumeration helps to define a data type whose objects can take a limited set of values.

2.2 STRUCTURE

Definition

- A Structure is a collection of variables of different data types under a single name and provides a convenient way of grouping several of related information together.
- Unlike arrays, it can be used for the storage of heterogeneous data (data of different data types).

2.2.1 Three main aspects of working with structure

1. Defining a structure type (Creating a new type)

2.2 Programming – Advanced Features

2. Initializing structure elements
3. Declaring variables and constants (objects) of the newly created type.

2.2.1.1 Defining Structure

Syntax

```
struct structure_name
{
    element-1;
    element-2;
    element-3; //Variable declarations
    ...
    ...
    element-n;
} v1,v2.....vn;
```

Where element1, element2, element3 are variables of any primitive or derived data types and v1,v2,...vn are structure variable.

Example

```
struct book
{
    char author[40];
    float price;
    int page;
}b1,b2;
```

Rules for defining structure

- Structure definition consists of the keyword struct followed by a structure tag name and a structure declaration list enclosed within braces.
- The structure declaration list consists of one or more variables declaration, possibly of different data types. The variable names declared in the structure declaration list are known as structure members.
- Structure members can be variables of the basic types(eg: char, float, int) or pointer type(eg: char *, int *) or aggregate type(eg: array).

- A structure declaration list cannot contain a member of void type or incomplete type or function type.
- Self referential structure: a structure may contain a pointer to an instance of itself is known as self referential structure.

2.2.1.2 Initializing Structure Elements

Syntax

```
Struct book
{
    int page;
    char author[10];
    float price;
}b1;
```

Example

```
void main()
{
    b1.author="Kalam";
    printf("Enter price:");
    scanf("%f",&b1.price);
    b1.page=178;
}
```

2.2.1.3 Declaring Structure Objects

- Variables (or) constants of the created structure type can be created either at the time of structure definition (or) after the time of structure definition.

Syntax

```
[type qualifier] structure type identifier name [= initialization list]; (or)
variables;
```

Example

```
struct book b1={3,2,1}; // it contains the initialization list
struct book b1,b2,b3; // it contains the variable
```

Rules for declaring structure objects:

- It is important to note that the structure members cannot be initialized during the structure definition; however the members of a structure object can be initialized by providing an initialization list.

2.2.2 Operations on Structures

- Aggregate operations
- Segregate operations

2.2.2.1 Aggregate Operations

- An aggregate operation treats an operand as an entity and operates on the entire operand as whole instead of operating on its constituent members.

Types

- a) Accessing members of an object of a structure
- b) Assigning a structure object to a structure variables
- c) Address of a structure object
- d) Size of a structure.

a) Accessing members of an object of a structure :

The members of a structure object can be accessed by using:

- (i) Direct member access operator (dot operator)
- (ii) Indirect member access operator (arrow operator)

(i) Direct member access operator (dot operator):

Syntax:

struct variable-name.struct-element-name

Example Program 2.1

//C program to print student details using structure

```
#include<stdio.h>
#include<conio.h>
struct student
{
    char name;
```

```
    int rno;
    float mark;
};
struct student s;
void main()
{
    printf("enter the name");
    scanf("%c",&s.name);
    printf("enter the rno");
    scanf("%d",&s.rno);
    printf("enter the mark");
    scanf("%f",&s.mark);
    printf("NAME=%c",s.name);
    printf("RNO=%d",s.rno);
    printf("MARK=%f",s.mark);
    getch();
}
```

OUTPUT

```
enter the name
xyz
enter the rno
20
enter the mark
80
NAME=xyz
RNO=20
MARK=80
```

Example Program 2.2

/*C program to calculate the student's average marks and student details using structure*/

```
#include<stdio.h>
#include<conio.h>
struct student
{
    char name;
    int rno;
    int m1,m2,m3;
};
struct student s;
void main()
{
    float total,average;
    printf("enter the name");
    scanf("%c",&s.name);
    printf("enter the rno");
    scanf("%d",&s.rno);
    printf("enter the marks");
    scanf("%d %d %d",&s.m1,&s.m2,&s.m3);
    total=s.m1+s.m2+s.m3;
    average=total/3;
    printf("NAME=%c",s.name);
    printf("RNO=%d",s.rno);
    printf("AVERAGE MARK=%f",average);
    getch();
}
```

Output:

```
enter the name
xyz
enter the rno
20
enter the marks
80
90
95
NAME=xyz
RNO=20
AVERAGE MARK=88.33
```

Example Program 2.3

```
#include <stdio.h>
struct book //Struct datatype declaration
{
    int x,y;
};
void main()
{
    struct book s1={4,5}; //s1-> variable of structure and values are
    initialized
    int a=10 , b=20 ;
    printf("\na=%d",s1.x+a); // elements are accessed using dot operator(.)
    printf("\nb=%d", s1.y+b);
}
```

Output:

```
a=4
b=5
```

2.8 Programming – Advanced Features

(ii) Indirect member access operator (arrow operator)

Syntax :

struct variable name -> struct element name

(or)

**struct variable name . struct element name*

Example Program 2.4

```
#include<stdio.h>
struct book // structure data type declaration
{
    int x,y;
};
struct book *b1; //pointer to structure
void main()
{
    printf("enter the values");
    scanf("%d", &b1->x);
    scanf("%d",&b1->y); //-> operator used
    printf("\nx=%d", b1->x);
    printf("\ny=%d", *b1.y);
}
```

Output:

Enter the values 10 20

x=10

y=20

b) Assigning a structure object to a structure variables

- Assignment operator (=) is used to assign the values of one variable to another variable. When assignment operator (=) is applied on structure variables, it performs member by member copy.

Example Program 2.5

```
#include<stdio.h>
struct book // struct datatype is declared
{
    char title[25], author[20];
    int price;
};
void main()
{
    struct book b1,b2,b3; //structure variables are declared
    b1={" cutting stone", "Abraham",400};
    b2.author=b1.author;
    b3=b1; // b1 variable values are assigned to b3
    printf("%s by %s is of Rs. %d \n", b1.title,b1.author,b1.price);
    printf("%s is the author of second book",b2.author);
    printf("%s by %s is of Rs. %d \n", b3.title,b3.author,b3.price);
}
```

Output:

```
cutting stone by Abraham is of Rs.400
Abraham is the author of second book
cutting stone by Abraham is of Rs. 400
```

c) Address of a structure object

- The address of operator (&) when applied to a structure object gives its base address. It can also be used to find the address of the constituting members of a structure object.

Example Program 2.6

```
#include<stdio.h>
struct book // struct datatype is declared
{
```

2.10 Programming – Advanced Features

```
char title[25], author[20];
int price;
};
void main()
{
    struct book b1,b2,b3; //structure variables are declared
    b1={" cutting stone", "Abraham",400};
    b2.author=b1.author;
    b3=b1; // b1 variable values are assigned to b3
    printf("%s by %s is of Rs. %d \n", b1.title,b1.author,b1.price);
    printf("\n address of structure's element title %u ",&b1.title);
}
}
```

Output

```
cutting stone by Abraham is of Rs.400
address of structure's element title 1700printf("\n address of whole variable
%u",&b1);
address of structure's element author 1725
address of whole variable 4000
```

d) Size of a structure.

- When the sizeof operator is applied to an operand of a structure type it will produce the result as how much memory space is occupied by that particular object.

Syntax:

```
sizeof (expression);
sizeof type
```

Example:

```
sizeof (struct book); // use structure's name
sizeof b1 // use variable
```

Program 2.7

```
#include<stdio.h>
struct book //structure type declaration
{
    char a; // elements are declared
    int b;
    char c;
    float d;
}; //structure type declarations are terminated
void main()
{
    struct book var; //variable declaratio
    printf("obj of struct book will take %d bytes\n",sizeof(struct book));
    printf("structure variable var takes %d bytes\n", sizeof var);
}
```

Output:

obj of struct pad will take 8 bytes
structure variable var takes 8 bytes

2.2.2.2 Segregate Operations

- A segregate operation operates on the individual members of a structure object.

Program 2.8

```
#include<stdio.h>
struct book
{
    char title[25], author[20];
    int page; float price;
};
void main()
```

```
{  
    struct book b1;  
    printf("enter title, author, page, price");  
    scanf("%s, %s, %d, %f",&b1.title,&b1.author,&b1.page, &b1.price);  
    printf("title is %s, author is %s, page no %d, price %d",b1.title,  
    b1.author, b1.page, b1.price);  
    // operations on individual element  
    b1.page+=100;  
    b1.price+=10;  
    printf("title is %s, author is %s, page no %d",b1.title, b1.author,  
    b1.page);  
    printf("price %d",b1.price);  
}
```

Output

```
Enter title, author, page, price  
Principles of life, prabhu, 145, 200.00  
Title is principles of life, author is prabhu, page no 145, price 200.00  
Title is principles of life, author is prabhu, page no 245, price 210.00
```

2.3 UNION

- Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time. Unions provide an efficient way of using the same memory location for multiple-purpose.
- Union is a user-defined data type, but unlike structures, they share the same memory location.

Defining a Union

- To define a union, you must use the union statement in the same way as did while defining a structure. The union statement defines a new data type with

more than one member for your program. The format of the union statement is as follows:

```
union [union tag] {  
    member definition;  
    member definition;  
    ...  
    member definition;  
} [one or more union variables];
```

- The union tag is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str`.

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

- Now, a variable of `Data` type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

Example Program 2.9 Illustration of Union

```
#include <stdio.h>  
#include <string.h>  
union Data {  
    int i;  
    float f;  
    char str[20];  
};
```

2.14 Programming – Advanced Features

```
void main() {  
    union Data data;  
    data.i = 10;  
    printf( "data.i : %d\n", data.i);  
    data.f = 220.5;  
    printf( "data.f : %f\n", data.f);  
    strcpy( data.str, "Charulatha publication");  
    printf( "data.str : %s\n", data.str);  
}
```

Output

```
data.i : 10  
data.f : 220.500000  
data.str : Charulatha publication
```

Difference between Structure and Union

Sl.No	Structure	Union
1	The member of a structure occupies its own memory space.	The member of union share same memory space.
2	The keyword struct is used to define a structure	The keyword union is used to define a structure
3	All the members of a structure can be initialized.	Only the first member of a union can be initialized.
4	In structure, each member is stored in a separate memory location. So need more memory space.	In union, all members are stored in the same memory locations. So, need less memory space.

2.4 POINTERS

2.4.1 Pointers to Variables

- A pointer is a variable that stores an address of another variable of same type.
- Pointer can have any name that is legal for other variable.

- Pointer variables are declared with prefix of ‘*’ operator.
- Using a pointer variable, we can access the value of another variable assigned to it.

Syntax

```
data_type *pointer_name;
```

Example

```
int *a;
```

- variable *a can store the address of any integer type variable.
- A pointer is a variable whose value is also an address.
- Each variable has two attributes
 - ✓ Value
 - ✓ Address

We can define pointers in two ways.

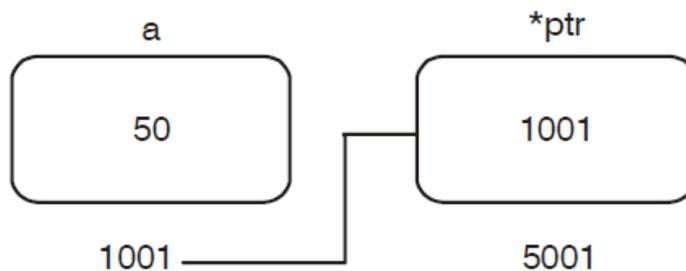
- First a pointer is a variable and assigns different values to a pointer variable.
- Second the value contained by a pointer must be an address which indicates the location of another variable in the memory. So, pointer is called as “address variable”.

Example

```
int a=50;
```

```
int *ptr;
```

```
ptr=&a;
```



- Here ‘a’ is a variable holds a value 50 and stored in a memory location 1001. ‘*ptr’ is pointer variable holds a address of a variable ‘a’.

Advantages of Using Pointers

- Pointers are more compact and efficient code.
- Pointers can be used to achieve clarity and simplicity.
- Pointers are used to pass information between function and its reference point.
- A pointer provides a way to return multiple data items from a function using its function arguments.
- Pointers also provide an alternate way to access an array element.
- A pointer enables us to access the memory directly.

Example Program 2.10

*/*C program for printing value and address of a variable using pointer variable*/*

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=3;
int *ptr;
ptr=&i;
clrscr();
printf("Address of i=%u\n",ptr);
printf("value of i=%d\n",*ptr);
getch();
}
```

Output:

```
Address of i=65524
value of i=3
```

Example Program 2.11

*/*C program for printing value and address of a variable using pointer variable by various methods*/*

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

```
#include<conio.h>
void main()
{
int i=4;
int *j;
j=&i;
clrscr();
printf("Address of i=%u\n",&i);
printf("Address of i=%u\n",j);
printf("Address of j=%u\n",&j);
printf("value of j=%u\n",j);
printf("value of i=%d\n",i);
printf("value of i=%d\n",*(&i));
printf("value of i=%d\n",*j);
getch();
}
```

Output

```
Address of i=65524
Address of i=65524
Address of j=65522
value of j=65524
value of i=4
value of i=4
value of i=4
```

Example Program 2.12

```
/*C program to add two numbers using pointers*/
#include<stdio.h>
#include<conio.h>
```

2.18 Programming – Advanced Features

```
void main()
{
int a,b,*p,*q,sum;
clrscr();
printf("Enter two integers");
scanf("%d %d",&a,&b);
p=&a;
q=&b;
sum=*p+*q;
printf("sum=%d",sum);
getch();
}
```

Output

```
Enter two integers 2 3
sum=5
```

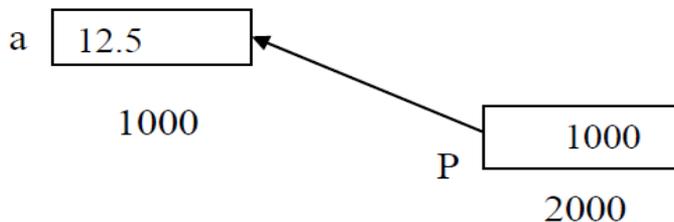
2.4.2 Pointer operators

a) Referencing a pointer

- A pointer variable is made to refer to an object.
- Reference operator(&) is used for this.
- Reference operator is also known as address of (&) operator.

Example

```
float a=12.5;
float *p;
p=&a;
```



b) Dereferencing a pointer

- The object referenced by a pointer can be indirectly accessed by dereferencing the pointer.
- Dereferencing operator (*) is used for this.
- This operator is also known as indirection operator or value- at-operator.

Example

```
int b;  
int a=12;  
a int *p;
```

Example program 2.13

```
#include<stdio.h>  
void main()  
{  
int a=12;  
int *p;  
int **pptr;  
p=&a;  
pptr=&p;  
printf("Value=%d",a);  
printf("value by dereferencing p is %d \n",*p);  
printf("value by dereferencing pptr is %d \n",**pptr);  
printf("value of p is %u \n",p);  
printf("value of pptr is %u\n",pptr);  
}
```

Output

```
Value=12  
value by dereferencing p is 12  
value by dereferencing pptr is 12  
value of p is 1000  
value of pptr is 2000
```

2.4.3 Arrays and pointers

- Array elements are always stored in consecutive memory locations according to the size of the array.
- The size of the variable with the pointer variables refers to, depends on the data type pointed by the pointer.
- A pointer when incremented, always points to a location after skipping the number of bytes required for the data type pointed to by it.

Example

```
int a[5]={ 10,20,30,40,50};
```

a[5] means the array 'a' has 5 elements and of integer data type

Program 2.14

```
/*C program to print the value and address of an array elements*/
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[5]={10,20,30,40,50};
int i;
clrscr();
for(i=0;i<5;i++)
{
printf("The value of a[%d]=%d\n",i,a[i]);
printf("Address of a[%d]=%u\n",i,&a[i]);
}
getch();
}
```

Output

The value of a[0]=10

Address of a[0]=4000

The value of a[1]=20
Address of a[1]=4002
The value of a[2]=10
Address of a[2]=4004
The value of a[3]=10
Address of a[3]=4006
The value of a[4]=10
Address of a[4]=4008

Example Program 2.15

*/*C program to print the value and address of an array elements using pointer*/*

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int arr[5]={10,20,30,40,50};
  int i,*p;
  p=arr;
  clrscr();
  for(i=0;i<=5;i++)
  {
    printf("\nAddress=%u\t",p+i);
    printf("Element=%d",*(p+i));
  }
  getch();
}
```

Output

Address 4000 Element=10
Address 4002 Element=20

2.22 Programming – Advanced Features

Address 4004 Element=30

Address 4006 Element=40

Address 4008 Element=50

Example Program 2.16

*/*C program to add sum of elements of an array using pointer*/*

```
#include<stdio.h>
main()
{
int i,sum;
int arr[5];
int *ptr;
for(i=0;i<5;i++)
{
printf("Enter the number");
scanf("%d",&arr[i]);
}
ptr=arr;
for(i=0;i<5;i++)
{
sum=sum+*ptr
Functions and Pointers 3.29
ptr=ptr+1;
}
printf("Total=%d",sum);
}
```

Output

Enter the number

10

20

```
30
40
50
Total= 150
```

2.4.3.1 Pointers with Multi-Dimensional Array

- A multi-dimensional array can also be represented with an equivalent pointer notation. A two dimensional array can be considered as a collection of one-dimensional arrays.

Syntax

```
data_type (*pointer variable) [expression];
data_type array name[expression 1][expression 2];
```

Example Program 2.17

/*C program to print the value and address of the element using array of pointers*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int * int *a[3];
int b=10,c=20,d=30,i;
a[0]=&b;
a[1]=&c;
a[2]=&d;
clrscr();
for(i=0;i<3;i++)
{
printf("Address=%u\n",a[i]);
printf("Value=%d\n",*(a[i]));
}
```

```
    getch();  
}
```

Output

```
Address=4000  
Value=10  
Address=5000  
Value=20  
Address=6000  
Value=30
```

2.4.4 Functions Pointers

- Function pointers in C can be used to create function calls to which they point. This allows programmers to pass them to functions as arguments. Such functions passed as an argument to other functions are also called callback functions.
- In C programming, it is also possible to pass addresses as arguments to functions. To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses.

Example Program 2.18

Write a C Program for Swapping of two numbers using function pointers.

```
#include <stdio.h>  
void swap(int *n1, int *n2);  
int main()  
{  
    int num1 = 5, num2 = 10;  
    // address of num1 and num2 is passed  
    swap( &num1, &num2);  
    printf("num1 = %d\n", num1);  
    printf("num2 = %d", num2);  
    return 0;
```

```
}  
void swap(int* n1, int* n2)  
{  
    int temp;  
    temp = *n1;  
    *n1 = *n2;  
    *n2 = temp;  
}
```

Output

num1 = 10

num2 = 5

- The address of num1 and num2 are passed to the swap() function using swap(&num1, &num2);
- When *n1 and *n2 are changed inside the swap() function, num1 and num2 inside the main() function are also changed.
- Inside the swap() function, *n1 and *n2 swapped. Hence, num1 and num2 are also swapped.

2.5 ENUMERATED DATA TYPES

- Enumeration or Enum in C is a special kind of data type defined by the user. It consists of constant integrals or integers that are given names by a user. The use of enum in C to name the integer values makes the entire program easy to learn, understand, and maintain by the same or even different programmer.

Syntax to Define Enum in C

- An enum is defined by using the 'enum' keyword in C, and the use of a comma separates the constants within. The basic syntax of defining an enum is:

```
enum enum_name{int_const1, int_const2, int_const3, .... int_constN};
```
- In the above syntax, the default value of int_const1 is 0, int_const2 is 1, int_const3 is 2, and so on. However, you can also change these default values while declaring the enum.

Example

- Below is an example of an enum named cars and how you can change the default values.

```
enum cars{BMW, Ferrari, Jeep, Mercedes-Benz};
```

- Here, the default values for the constants are:

BMW=0, Ferrari=1, Jeep=2, and Mercedes-Benz=3. However, to change the default values, you can define the enum as follows:

```
enum cars{  
    BMW=3,  
    Ferrari=5,  
    Jeep=0,  
    Mercedes-Benz=1  
};
```

2.5.1 Enumerated Type Declaration to Create a Variable

- Similar to pre-defined data types like int and char, you can also declare a variable for enum and other user-defined data types. Here's how to create a variable for enum.
 - **enum condition (true, false);** //declaring the enum
 - **enum condition e;** //creating a variable of type condition
- Suppose we have declared an enum type named condition; we can create a variable for that data type as mentioned above. We can also converge both the statements and write them as: **enum condition (true, false) e;**
- For the above statement, the default value for true will be 1, and that for false will be 0.

2.5.2 Implementing enum using C Program

Example program 2.19: Printing the Values of Weekdays

```
#include <stdio.h>  
  
enum days{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday,  
Saturday};  
  
int main(){
```

```
// printing the values of weekdays
for(int i=Sunday;i<=Saturday;i++){
    printf("%d, ",i);
}
return 0;
}
```

Output

```
Value of cont2 is = 7
Value of cont3 is = 3
Value of hearts is = 5
```

2.6 FILE HANDLING

- A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image.
- The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data.
- The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

2.6.1 Why we need file?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C. It is possible easily move your data from one computer to another without any changes.

2.6.2 File Operations

- In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again.
- However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.
- File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.
 - Creation of the new file
 - Opening an existing file
 - Reading from the file
 - Writing to the file
 - Deleting the file

2.6.2.1 Types of Files

- There are two kinds of files in which data can be stored in two ways either in characters coded in their ASCII character set or in binary format. They are
 1. Text Files (or) ASCII file
 2. Binary Files

Text Files (or) ASCII file

- The file that contains ASCII codes of data like digits, alphabets and symbols is called text file (or) ASCII file.

Binary Files

- A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.
- The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can't be recognized by a word processor.

1. `wb(write)`
This opens a binary file in write mode.
SYNTAX: `fp=fopen("data.dat","wb");`
2. `rb(read)`
This opens a binary file in read mode
SYNTAX: `fp=fopen("data.dat","rb");`
3. `ab(append)`
This opens a binary file in a Append mode i.e. data can be added at the end of file.
SYNTAX: `fp=fopen("data.dat","ab");`
4. `r+b(read+write)`
This mode opens preexisting File in read and write mode.
SYNTAX: `fp=fopen("data.dat","r+b");`
5. `w+b(write+read)`
This mode creates a new file for reading and writing in Binary mode.
SYNTAX: `fp=fopen("data.dat","w+b");`
6. `a+b(append+write)`
This mode opens a file in append mode i.e. data can be written at the end of file.
SYNTAX: `fp=fopen("data.dat","a+b");`

Opening Modes in Standard I/O

r	Open for reading	If the file does not exist, <code>fopen()</code> returns NULL
rb	Open for reading in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.

2.30 Programming – Advanced Features

a	Open for append. i.e, Data is added to the end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary file.	If the file does not exist, fopen() returns NULL
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Closing a File: `fclose(fp);`

- The file (both text and binary) should be closed after reading/writing. Closing a file is performed using library function `fclose()`.

Reading and writing to a text file

- The functions `fprintf()` and `fscanf()` are used to read or write the file They are just the file versions of `printf()` and `scanf()`. The only difference is that, `fprint` and `fscanf` expects a pointer to the structure `FILE`.

Writing to a text file

Program 2.20: Write to a text file using `fprintf()`

```
#include <stdio.h>

int main()
{
    int num;
```

```
FILE *fptr;
fptr = fopen("C:\\program.txt", "w");
if(fptr == NULL)
{
printf("Error!");
exit(1);
}
printf("Enter num: ");
scanf("%d", &num);
fprintf(fptr, "%d", num);
fclose(fptr);
return 0;
}
```

- This program takes a number from user and stores in the file program.txt. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Reading from a text file

Program 2.21: Read from a text file using fscanf()

```
#include <stdio.h>
int main()
{
int num;
FILE *fptr;
if ((fptr = fopen("C:\\program.txt", "r")) == NULL){
printf("Error! opening file");
// Program exits if the file pointer returns NULL.
exit(1);
}
```

2.32 Programming – Advanced Features

```
fscanf(fptr, "%d", &num);
printf("Value of n=%d", num);
fclose(fptr);
return 0;
}
```

Reading and writing to a binary file

- Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

- To write into a binary file, you need to use the function fwrite(). The function takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_data,numbers_data,pointer_to_file);
```

Program 2.22: Writing to a binary file using fwrite()

```
#include <stdio.h>
struct threeNum
{
int n1, n2, n3;
};
int main()
{
int n;
struct threeNum num;
FILE *fptr;
if ((fptr = fopen("C:\\program.bin", "wb")) == NULL){
printf("Error! opening file");
// Program exits if the file pointer returns NULL.
exit(1);
```

```
}  
for(n = 1; n < 5; ++n)  
{  
    num.n1 = n;  
    num.n2 = 5n;  
    num.n3 = 5n + 1;  
    fwrite(&num, sizeof(struct threeNum), 1, fptr);  
}  
fclose(fptr);  
return 0;  
}
```

- We declare a structure three Num with three numbers - *n1*, *n2* and *n3*, and define it in the main function as *num*. Now, inside the for loop, we store the value into the file using *fwrite*.
- The first parameter takes the address of *num* and the second parameter takes the size of the structure three Num. Since, we're only inserting one instance of *num*, the third parameter is 1. And, the last parameter **fptr* points to the file we're storing the data. Finally, we close the file.

Reading from a binary file

- Function *fread()* also take 4 arguments similar to *fwrite()* function as above.
`fread(address_data,size_data,numbers_data,pointer_to_file);`

Program 2.23: Reading from a binary file using *fread()*

```
#include <stdio.h>  
struct threeNum  
{  
    int n1, n2, n3;  
};  
int main()  
{  
    int n;
```

2.34 Programming – Advanced Features

```

struct threeNum num;
FILE *fptr;
if ((fptr = fopen("C:\\program.bin", "rb")) == NULL){
printf("Error! opening file");
// Program exits if the file pointer returns NULL.
exit(1);
}
for(n = 1; n < 5; ++n)
{
fread(&num, sizeof(struct threeNum), 1, fptr);
printf("n1: %d\nn2: %d\nn3: %d", num.n1, num.n2, num.n3);
}
fclose(fptr);
return 0;
}

```

- This program will start reading the records from the file program.bin in the reverse order (last to first)

Text Files

- In C, all components are files, each with a different behavior based on the attached devices. To enable the I/O functions, several standard built-in functions were created and stored in libraries.
- Some of the high level file I/O functions are given in Table 2.1

Table 2.1 High level file I/O functions

S.No	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file

6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

- fopen ()** : It creates a new file for use or opens an existing file for use.
- fclose ()**: It closes a file which has been opened for use.
- fscanf**(file pointer, format string, address of the variable)
Example: fscanf(fp, "%d", &num);
- fprintf**(console output, "format string", file pointer);
Example: fprintf(stdout, "%f \n", f); /*note: stdout refers to screen */
- getw ()**: This function returns the integer value from a given file and increment the file pointer position to the next message.
Syntax: getw (fptr);
Where fptr is a file pointer which takes the integer value from file.
- putw ()**: This function is used for writing an integer value to a given file.
Syntax: putw (value,fptr);

Where fptr is a file pointer Value is an integer value which is written to a given file.

Example Program for getw() and putw()

Program 2.24: Write a program to read integer data from the user and write it into the file using putw() and read the same integer data from the file using getw() and display it on the output screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
```

```
int n;
clrscr();
fp=fopen("c.dat", "wb+");
printf("Enter the integer data");
scanf("%d",&n);
while(n!=0)
{
    putw(n,fp);
    scanf("%d",&n);
}
rewind(fp);
printf("Reading data from file");
while((n=getw(fp))!=EOF)
{
    printf("%d\n",n);
}
fclose(fp);
getch();
}
```

7. fwrite()

- This function is used for writing an entire block to a given file.

Syntax: fwrite(ptr,size,nst,fptr);

ptr is a pointer ,it points to the array of structure.

Size is the size of the structure

nst is the number of the structure

fptr is a filepointer.

8. fread()

- fread(ptr,size,position,fptr);similar to fwrite

9. fflush(stdin);To clean the input stream

Program 2.25: program for fwrite():

Write a program to read an employee details and write them into the file at a time using fwrite().

```
#include<stdio.h>
#include<conio.h>
void main()
    {
    struct emp
    {
    int eno;
    char ename[20];
    float sal;
    }e;
    FILE *fp;
    fp=fopen("emp.dat", "wb");
    clrscr();
    printf("Enter employee number");
    scanf("&d",&e.eno);
    printf("Enter employee name");
    fflush(stdin);
    scanf("%s",e.ename);
    printf("Enter employee salary");
    scanf("%f",&e.sal);
    fwrite(&e,sizeof(e),1,fp);
    printf("One record stored successfully");
    getch();
    }
```

Operations for Search data in a file

1. fseek()
2. ftell()

3. rewind()

fseek() : *Getting data using fseek()*

- When many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record. This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

Syntax of fseek()

fseek(FILE * stream, long int offset, int whence)

fseek(file pointer, displacement, pointer position);

- The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.
- This function is used for seeking the pointer position in the file at the specified byte.

Syntax: fseek(file pointer, displacement, pointer position);

file pointer - It is the pointer which points to the file.

displacement -It is positive or negative.

- This is the number of bytes which are skipped backward (if negative) or forward (if positive) from the current position. This is attached with L because this is a long integer.

Pointer position: This sets the pointer position in the file.

Value	Pointer position	Value	Pointer position
0			Beginning of file.
1			Current position
2			End of file

Example:

1. fseek(p,10L,0)

- This 0 means pointer position is on beginning of the file, from this statement pointer position is skipped 10 bytes from the beginning of the file.

2. fseek(p,5L,1)

- This 1 means current position of the pointer position. From this statement pointer position is skipped 5 bytes forward from the current position.

3. fseek(p,-5L,1):

- From this statement pointer position is skipped 5 bytes backward from the current position.

Different When in fseek	
When	Meaning
SEKK_SET	Starts the offset from the beginning of the file.
SEKK_END	Starts the offset from the end of the file.
SEKK_CUR	Starts the offset from the current location of the cursor in the file.

Program 2.26: for fseek()

```

#include <stdio.h>
struct threeNum
{
int n1, n2, n3;
};
int main()
{
int n;

struct threeNum num;
FILE *fptr;
if ((fptr = fopen("C:\\program.bin", "rb")) == NULL){
printf("Error! opening file");
// Program exits if the file pointer returns NULL.
exit(1);
}
// Moves the cursor to the end of the file

```

2.40 Programming – Advanced Features

```
fseek(fptr, sizeof(struct threeNum), SEEK_END);
for(n = 1; n < 5; ++n)
{
fread(&num, sizeof(struct threeNum), 1, fptr);
printf("n1: %d\nn2: %d\nn3: %d", num.n1, num.n2, num.n3);
}
fclose(fptr);
return 0;
}
```

ftell()

- This function is used to move the file pointer to the beginning of the given file. This function returns the value of the current pointer position in the file. The value is count from the beginning of the file.

Syntax: ftell(fptr); fptr is a file pointer.

rewind()

Syntax: rewind(fptr); fptr is a file pointer.

Program 2.27: program for fseek():

Write a program to read last ‘n’ characters of the file using appropriate file functions(Here we need fseek() and fgetc()).

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char ch;
clrscr();
fp=fopen("file1.c", "r");
if(fp==NULL)
```

```
printf("file cannot be opened");
else
{
printf("Enter value of n to read last 'n' characters");
scanf("%d",&n);
fseek(fp,-n,2);
while((ch=fgetc(fp))!=EOF)
{
printf("%c\t",ch);
}
}
fclose(fp);
getch();
}
```

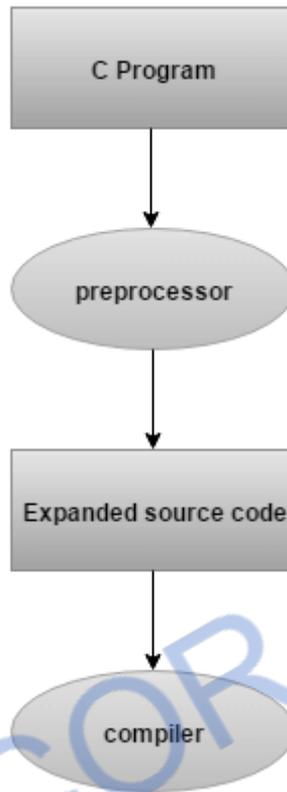
2.7 PREPROCESSOR DIRECTIVES

- The C preprocessor is a microprocessor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.
- **Note:** A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Example

```
#define PI 3.14
```

Here, PI is the macro name which will be replaced by the value 3.14.



- All preprocessor directives starts with hash # symbol. Let's see a list of preprocessor directives.
 - #define: It substitutes a preprocessor using macro.
 - #include: It helps to insert a certain header from another file.
 - #undef: It undefines a certain preprocessor macro.
 - #ifdef: It returns true if a certain macro is defined.
 - #ifndef: It returns true if a certain macro is not defined.
 - #if, #elif, #else, and #endif: It tests the program using a certain condition; these directives can be nested too.
 - #line: It handles the line numbers on the errors and warnings. It can be used to change the line number and source files while generating output during compile time.
 - #error and #warning: It can be used for generating errors and warnings.

- `#error` can be performed to stop compilation.
- `#warning` is performed to continue compilation with messages in the console window.
- `#region` and `#endregion`: To define the sections of the code to make them more understandable and readable, we can use the region using expansion and collapse features.

STUCOR APP

REVIEW QUESTIONS

PART A

1. What is a Structure in C?

- Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful.
- In structure, data is stored in form of records.

2. How to define a Structure?

- **struct** keyword is used to define a structure. **struct** defines a new data type which is a collection of primary and derived data types.

- Syntax

```
struct [structure_tag]
{
    //member variable 1
    //member variable 2
    //member variable 3
    ...
}[structure_variables];
```

3. What is Union?

- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purpose.

4. Give the syntax for creating a union.

```
union [union name]
{
    member definition;
    member definition;
```

```

...
    member definition;
};
    
```

5. Difference between Structure and Union.

Structure	Union
The Keyword struct is used to define the Structure	The Keyword union is used to define the Union
Structure allocates storage space for all its members separately.	Union allocates one storage space for all its members.
Structure occupies high memory space	Union occupies low memory space when compared to Structure
We can access all members of Structure at a time	Only one member of union can be accessed at a time.
Altering the value of a member will not affect other member of a structure	Altering the value of a member will alter other member value in union.

6. What are Enumerated Datatypes?

- Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

7. What is pointer?

- A pointer is a variable that stores the memory address of another variable as its value. A pointer variable points to a data type (like int) of the same type and is created with the * operator.

8. How addresses are assigned to Pointers?

- Example

```

int* p, a;
a= 8;
p = &a;
    
```

- Here, 8 is assigned to the variable a and the address of a is assigned to the pointer p.

9. What are the uses of Pointers?

- Pointers are used to return more than one value to the function
- Pointers are more efficient in handling the data in arrays
- Pointers reduce the length and complexity of the program
- They increase the execution speed
- The pointers saves data storage space in memory.

10. What is the difference between an array and pointer?

Arrays	Pointers
Array allocates space automatically.	Pointer is explicitly assigned to point to an allocated space.
It cannot be resized.	It can be resized using realloc ().
It cannot be reassigned.	Pointers can be reassigned.
Sizeof(array name) gives the number of bytes occupied by the array.	Sizeof(pointer name) returns the number of bytes used to store the pointer variable.

11. What is dangling pointer?

- In C, a pointer may be used to hold the address of dynamically allocated memory. After this memory is freed with the free() function, the pointer itself will still contain the address of the released block. This is referred to as a dangling pointer.
- Using the pointer in this state is a serious programming error. Pointer should be assigned NULL after freeing memory to avoid this bug.

12. What is 'C' functions?

- A function is a self-contained block (or) a sub-program of one or more statements that performs a special task when called.
- To perform a task repetitively then it is not necessary to re-write the particular block of the program again and again. The function defined can be used for any number of times to perform the task.

13. Differentiate library functions and User-defined functions.

Library Functions	User-defined Functions
Library functions are pre-defined set of functions that are defined in C libraries.	The User-defined functions are the functions defined by the user according to his/her requirement.
User can only use the function but cannot change (or) modify this function.	User can use this type of function. User can also modify this function.

14. What are the steps in writing a function in a program?

- Function Declaration (Prototype declaration): Every user-defined function has to be declared before the main().
- Function Calling: The user-defined functions can be called inside any functions like main(), user-defined function, etc.
- Function Definition: The function definition block is used to define the user-defined functions with statements.

15. What is a use of 'return' Keyword?

- The 'return' Keyword is used only when a function returns a value.

16. What is the purpose of the function main()?

- The function main () invokes other functions within it. It is the first function to be called when the program starts execution.
- Features of Main method
 - It is the starting function.
 - It returns an int value to the environment that called the program.
 - Recursive call is allowed for main () also.
 - It is a user-defined function.
 - Program execution ends when the closing brace of the function main() is reached.
 - It has two arguments (a) argument count and (b)argument vector (represents strings passed.)

17. Compare between Array and Structure

Arrays	Structures
An array is a collection of data items of same data type.	A structure is a collection of data items of different data types.
Arrays can only be declared. There is no keyword for arrays.	Structures can be declared and defined. The Keyword for structures is struct.
An array name represents the address of the starting element.	A structure name is known as tag. It is a shorthand notation of the declaration.
An array cannot have bit fields.	A structure may contain bit fields.

18. Is it better to use a macro or a function?

- Macros are more efficient (and faster) than function because their corresponding code is inserted directly at the point where the macro is called. There is no overhead involved in using a macro like there is in placing a call to a function.
- However, macros are generally small and cannot handle large, complex coding constructs. In cases where large, complex constructs are to be handled, functions are more suited, additionally; macros are expanded inline, which means that the code is replicated for each occurrence of a macro.

19. List the characteristics of Arrays.

- All elements of an array share the same name, and they are distinguished from one another with help of an element number.
- Any element of an array can be modified separately without disturbing other elements.

20. What are the types of Arrays?

- One-Dimensional Array
- Two-Dimensional Array
- Multi-Dimensional Array

21. What is File Handling in C?

- A file is nothing but a source of storing information permanently in the form of a sequence of bytes on a disk. The contents of a file are not volatile like the C compiler memory. The various operations available like creating a file, opening a file, reading a file, or manipulating data inside a file is referred to as file handling.

22. What is the need for File Handling in C?

- **Reusability:** It helps in preserving the data or information generated after running the program.
- **Large storage capacity:** Using files, you need not worry about the problem of storing data in bulk.
- **Saves time:** There are certain programs that require a lot of input from the user. You can easily access any part of the code with the help of certain commands.
- **Portability:** You can easily transfer the contents of a file from one computer system to another without having to worry about the loss of data.

23. List some of C File Handling Operations.

- Creating a new file: `fopen()`
- Opening an existing file in your system: `fopen()`
- Closing a file: `fclose()`
- Reading characters from a line: `getc()`
- Writing characters in a file: `putc()`
- Reading a set of data from a file: `fscanf()`
- Writing a set of data in a file: `fprintf()`
- Reading an integral value from a file: `getw()`

24. Give the syntax for Opening a Text File in C.

Syntax

```
*fpointer = FILE *fopen(const char *file_name, const char *mode);
```

- *fpointer is the pointer to the file that establishes a connection between the file and the program.
- *file_name is the name of the file.
- *mode is the mode in which we want to open our file.

25. How to Read and Write a Text File in C?

- The input/output operations in a file help you read and write in a file.
- The simplest functions used while performing operations on reading and writing characters in a file are `getc()` and `putc()` respectively.

2.50 Programming – Advanced Features

- In order to read and write a set of data in a file, we use the `fscanf()` and `fprintf()` operators.

26. Write short notes on Preprocessor Directives.

- The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation (Preprocessor directives are executed before compilation.).
- It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

27. What is macro?

- A macro is a segment of code which is replaced by the value of macro. Macro is defined by `#define` directive.

28. List few preprocessor directives in C.

- `#include`
- Macro's (`#define`)
- `#undef`
- `#ifdef`
- `#ifndef`
- `#if`
- `#else`

PART-B

1. Explain Structure in C with neat program.
2. Write short notes on Union with example program.
3. What is a function? Explain with neat program.
4. Explain call by value and call by reference with example programs.
5. Explain the file handling mechanism in C with programs.
6. Explain preprocessor directives with its types and examples.
7. Explain the concept of pointers with neat programs.
8. Write shorts notes on Arrays.
9. How to write Data into a text file and Read Data from the file? Discuss.
10. How to read and write data to the binary file in a program? Explain.

STUCOR APP

LINEAR DATA STRUCTURES

Abstract Data Types (ADTs) – List ADT – Array-Based Implementation – Linked List – Doubly-Linked Lists – Circular Linked List – Stack ADT – Implementation of Stack – Applications – Queue ADT – Priority Queues – Queue Implementation – Applications.

3.1 ABSTRACT DATA TYPES (ADTS)

- An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.
- In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details.
- The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

3.1.1 Abstract data type model

Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.

Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.

3.2 Linear Data Structures

Figure 3.1 shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program

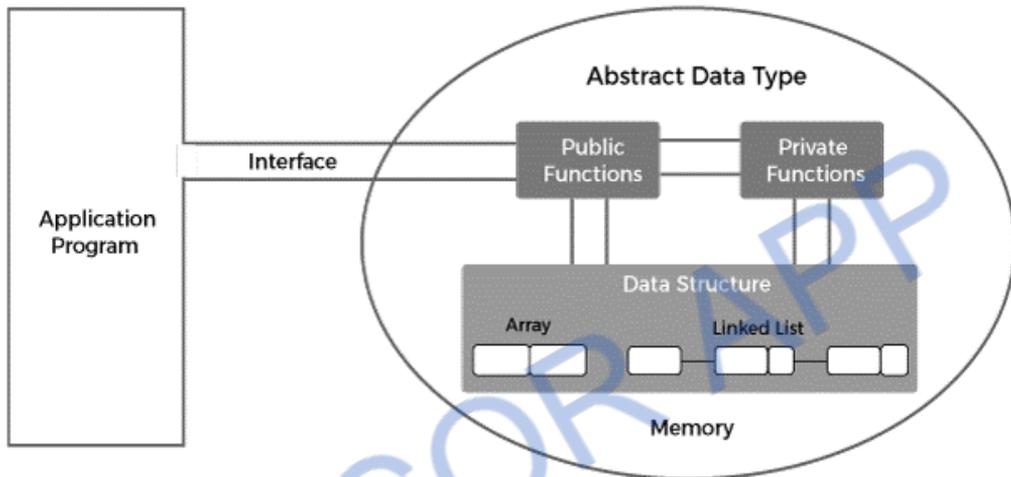


Fig 3.1 Abstract Data Type Model

Example 3.1

Suppose we have an index array of size 4. We have an index location starting from 0, 1, 2, 3. Array is a data structure where the elements are stored in a contiguous location. The memory address of the first element is 1000, second element is 1004, third element is 1008, and the fourth element is 1012. Since it is of integer type so it will occupy 4 bytes and the difference between the addresses of each element is 4 bytes. The values stored in an array are 10, 20, 30 and 40. These values, index positions and the memory addresses are the implementations.

The abstract or logical view of the integer array can be stated as:

- It stores a set of elements of integer type.
- It reads the elements by position, i.e., index.
- It modifies the elements by index
- It performs sorting

3.2 LIST ADT

- The list can be defined as an abstract data type in which the elements are stored in an ordered manner for easier and efficient retrieval of the elements. List Data Structure allows repetition that means a single piece of data can occur more than once in a list.
- In the case of multiple entries of the same data, each entry of that repeating data is considered as a distinct item or entry. It is very much similar to the array but the major difference between the array and the list data structure is that array stores only homogenous data in them whereas the list (in some programming languages) can store heterogeneous data items in its object. List Data Structure is also known as a sequence.
- The list can be called Dynamic size arrays, which means their size increased as we go on adding data in them and we need not to pre-define a static size for the list

For example,

```
numbers = [ 1, 2, 3, 4, 5]
```

- In this example, 'numbers' is the name of the List Data Structure object and it has five items stored in it. In the object named numbers, we have stored all the elements of numeric type. In the list, the indexing starts from zero, which means if we want to access or retrieve the first element of this list then we need to use index zero and similarly whenever we want to access any element from this list named numbers. In other words, we can say that element 1 is on the index 0 and element 2 is on index 1 and similarly for further all elements.

```
Mixed_data = [205, 'Mathu', 8.56]
```

- In this second example, mixed_data is the name of the list object that stores the data of different types. In the mixed_data list, we have stored data of three types, first one is the integer type which is id '205', after the integer data we have stored a string type data having the value 'Mathu' stored at index 1 and at last the index value 2, we have stored a float type data having the value '8.56'.
- To access the elements of the mixed_data list, we need to follow the same approach as defined in the previous example.
- And we can add more data to these defined List objects and that will get appended at the last of the list. For example, if we add another data in the mixed_data list,

3.4 Linear Data Structures

it will get appended after the float value object having value '8.56'. And we can add repeating values to these list-objects.

3.2.1 Operations on the List Data Structure

Add or Insert Operation:

In the Add or Insert operation, a new item (of any data type) is added in the List Data Structure or Sequence object.

Replace or reassign Operation:

In the Replace or reassign operation, the already existing value in the List object is changed or modified. In other words, a new value is added at that particular index of the already existing value.

Delete or remove Operation:

In the Delete or remove operation, the already present element is deleted or removed from the Dictionary or associative array object.

Find or Lookup or Search Operation:

In the Find or Lookup operation, the element stored in that List Data Structure or Sequence object is fetched.

3.3 ARRAY-BASED IMPLEMENTATION

- Arrays are defined as the collection of similar types of data items stored at contiguous memory locations.
- It is one of the simplest data structures where each data element can be randomly accessed by using its index number.
- They are the derived data types in C programming that can store the primitive type of data such as int, char, double, float, etc.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

3.3.1 Properties of array

- Each element in an array is of the same data type and carries the same size that is 4 bytes.

- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

3.3.2 Representation of an array

- Array can be represented in various ways in different programming languages. As an illustration, let's see the declaration of array in C language

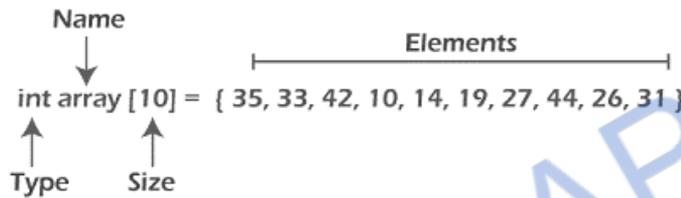


Fig. 3.2: Illustration of an Array

As per the above illustration of array, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

3.3.3 Memory allocation of an array

- Data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.
- We can define the indexing of an array in the below ways
 - 0 (zero-based indexing): The first element of the array will be `arr[0]`.
 - 1 (one-based indexing): The first element of the array will be `arr[1]`.
 - n (n - based indexing): The first element of the array can reside at any random index number
- Fig 3.3 shows the memory allocation of an array `arr` of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the address of `arr[0]`. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

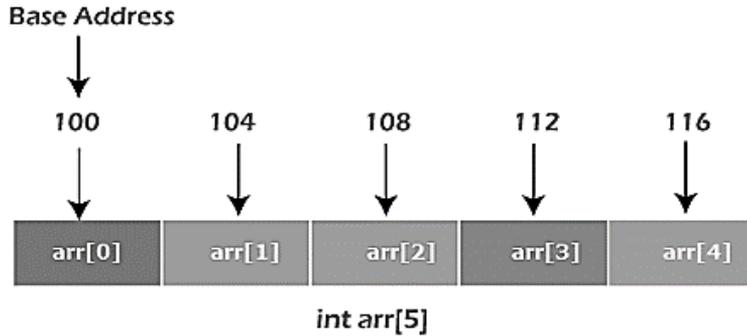


Fig. 3.3: Memory allocation of an array

3.3.4 Access an element from the array

The information given below are required to access any random element from the array

- Base Address of the array.
- Size of an element in bytes.
- Type of indexing, array follows.

The formula to calculate the address to access an array element

$$\text{Byte address of element } A[i] = \text{base address} + \text{size} * (i - \text{first index})$$

Here, size represents the memory taken by the primitive data types. As an instance, int takes 2 bytes, float takes 4 bytes of memory space in C programming.

Example 3.2

Suppose an array, $A[-10 \dots +2]$ having Base address (BA) = 999 and size of an element = 2 bytes, find the location of $A[-1]$.

$$\begin{aligned} L(A[-1]) &= 999 + 2 \times [(-1) - (-10)] \\ &= 999 + 18 \\ &= 1017 \end{aligned}$$

3.3.5 Basic operations of an Array

Basic operations supported in the array

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.

- Update - It updates an element at a particular index.

3.3.6 Complexity of Array operations

Time and space complexity of various array operations are described below.

Time Complexity

Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is O(n).

3.3.7 Limitations of Array

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

3.3.8 Advantages of Array

- Arrays are good for storing multiple values in a single variable.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.

3.3.9 Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.

3.8 Linear Data Structures

- There will be wastage of memory if we store less number of elements than the declared size.

3.4 LINKED LIST

- Linked list is a linear data structure that includes a series of connected nodes.
- Linked list can be defined as the nodes that are randomly stored in the memory.
- A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
- The last node of the list contains a pointer to the null.
- After array, linked list is the second most used data structure.
- In a linked list, every link contains a connection to another link

3.4.1 Representation of a Linked list

- Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below.

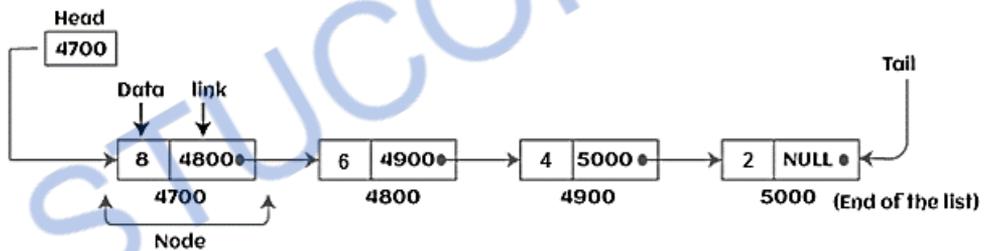


Fig. 3.4: Representation of Linked List

3.4.2 Why use linked list over array?

- Linked list is a data structure that overcomes the limitations of arrays (Refer 3.3.6)
- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration.
- List grows as per the program's demand and limited to the available memory space.

3.4.3 Declare a linked list

- It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e.,
 - Simple variable,
 - Pointer variable.

We can declare the linked list by using the user-defined data type structure.

The declaration of linked list is given as follows

```
struct node
{
int data;
struct node *next;
}
```

In the above declaration, we have defined a structure named as node that contains two variables, one is data that is of integer type, and another one is next that is a pointer which contains the address of next node.

3.4.4 Types of Linked list

Linked list is classified into the following types

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

3.4.4.1 Singly-Linked List

- Singly linked list can be defined as the collection of an ordered set of elements.
- A node in the singly linked list consists of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node
- Link part of the node stores the address of its immediate successor.

3.4.4.2 Doubly Linked List

- Doubly linked list is a complex type of linked list
- Here, a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly-linked list, a node consists of three parts:
 - Node data,
 - Pointer to the next node in sequence (next pointer), and
 - Pointer to the previous node (previous pointer).

3.4.4.3 Circular Singly Linked List

- In a circular singly linked list, the last node of the list contains a pointer to the first node of the list.
- We can have circular singly linked list as well as circular doubly linked list.

3.4.4.4 Circular Doubly Linked List

- Circular doubly linked list is a more complex type of data structure.
- Here a node contains pointers to its previous node as well as the next node.
- Circular doubly linked list doesn't contain NULL in any of the nodes.
- The last node of the list contains the address of the first node of the list.
- The first node of the list also contains the address of the last node in its previous pointer.

3.4.5 Advantages of Linked list

- **Dynamic data structure** - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion** - Unlike arrays, insertion, and deletion in linked list is easier. Elements in the linked list are stored at a random location.
- **Memory efficient** - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation** - We can implement both stacks and queues using linked list.

3.4.6 Disadvantages of Linked list

- **Memory usage** - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal** - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly.
- **Reverse traversing** - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

3.4.7 Applications of Linked list

- Using linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

3.4.8 Operations performed on Linked list

- **Insertion** - This operation is performed to add an element into the list.
- **Deletion** - It is performed to delete an operation from the list.
- **Display** - It is performed to display the elements of the list.
- **Search** - It is performed to search an element from the list using the given key.

3.4.9 Complexity of Linked list

1. Time Complexity

Operation	Average Case	Worst Case
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

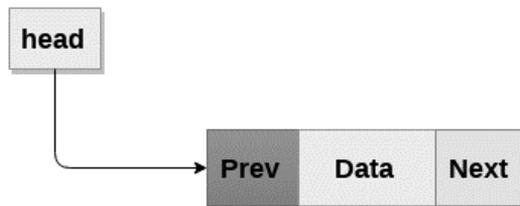
2. Space Complexity

Operation	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

3.5 DOUBLY LINKED LIST

- Doubly linked list is a complex type of linked list
- Here, a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly-linked list, a node consists of three parts:
 - Node data,
 - Pointer to the next node in sequence (next pointer), and
 - Pointer to the previous node (previous pointer).

A sample node in a doubly linked list is shown in Fig. 3.5



Node

Fig. 3.5: Sample node in a doubly linked list

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in Fig 3.6



Fig. 3.6: Doubly Linked List

In C, structure of a node in doubly linked list can be given as:

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

- The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.
- In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list.
- Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

3.5.1 Memory Representation of a doubly linked list

- Memory Representation of a doubly linked list is shown in Fig. 3.7. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
- In Fig 3.7, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added

3.14 Linear Data Structures

to the list therefore the **prev** of the list contains null. The **next** node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

- We can traverse the list in this way until we find any node containing null or -1 in its next part.

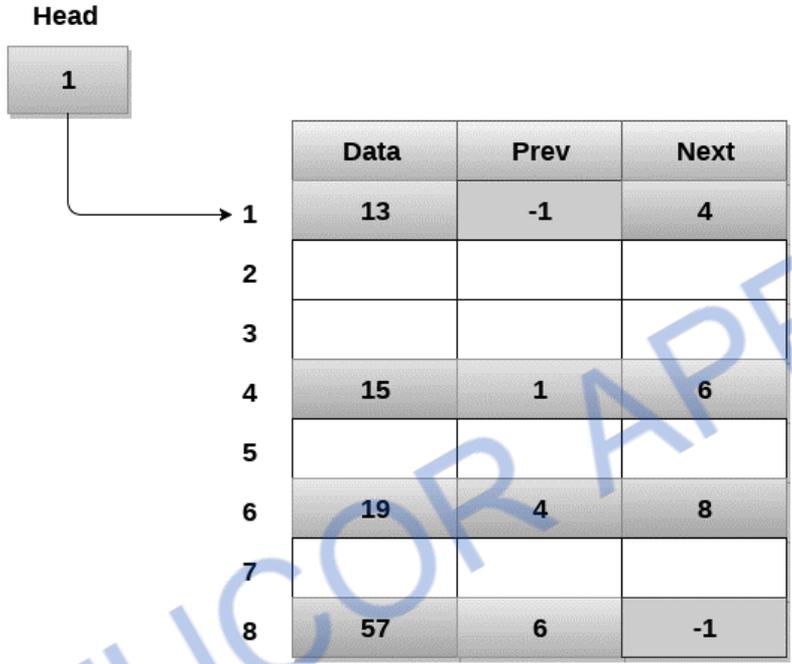


Fig. 3.7: Memory Representation of a doubly linked list

3.5.2 Operations on doubly linked list

Table 3.1 describes all the operations performed on Doubly Linked List

Table 3.1: Operations on Doubly Linked List

Sl. No.	Operation	Description
1.	Insertion at beginning	Adding the node into the linked list at beginning.
2.	Insertion at end	Adding the node into the linked list to the end.
3.	Insertion after specified node	Adding the node into the linked list after the specified node.
4.	Deletion at beginning	Removing the node from beginning of the list
5.	Deletion at the end	Removing the node from end of the list.

6.	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7.	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null
8.	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

3.5.2.1 Insertion at beginning

- There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element.

The following steps to be performed for insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds.
- In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
- In the second scenario, the condition head == NULL become false and the node will be inserted in beginning.
- The next pointer of the node will point to the existing head pointer of the node.
- The prev pointer of the existing head will point to the new node being inserted.
- Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer.
- Hence assign null to its previous part and make the head point to this node.

Algorithm 3.1

```

Step 1: IF ptr = NULL
    Write OVERFLOW
    Go to Step 9
[END OF IF]
    
```

3.16 Linear Data Structures

- Step 2: SET NEW_NODE = ptr
- Step 3: SET ptr = ptr -> NEXT
- Step 4: SET NEW_NODE -> DATA = VAL
- Step 5: SET NEW_NODE -> PREV = NULL
- Step 6: SET NEW_NODE -> NEXT = START
- Step 7: SET head -> PREV = NEW_NODE
- Step 8: SET head = NEW_NODE
- Step 9: EXIT

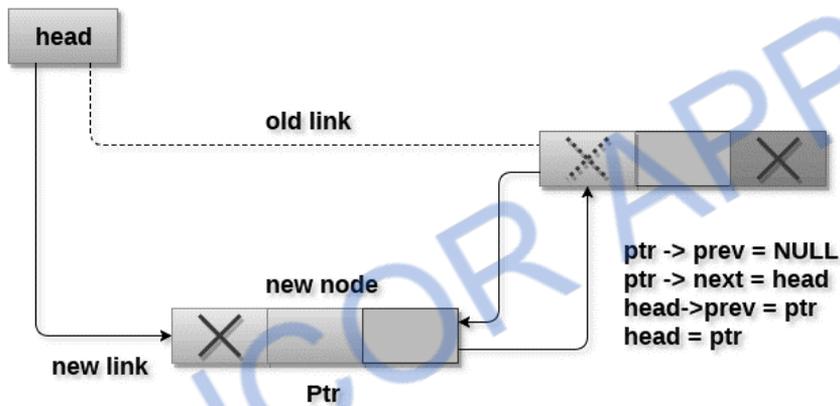


Fig. 3.8: Insertion into Doubly Linked List at the beginning

3.5.2.2 Insertion at end

- To insert a node in doubly linked list at the end, make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.
- Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds.
- In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list.

- For this reason, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.
- Make the next pointer of temp point to the new node being inserted i.e. ptr.
- Make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.
- Make the next pointer of the node ptr point to the null as it will be the new last node of the list.

Algorithm 3.2

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET TEMP = START

Step 7: Repeat Step 8 while TEMP -> NEXT != NULL

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10C: SET NEW_NODE -> PREV = TEMP

Step 11: EXIT

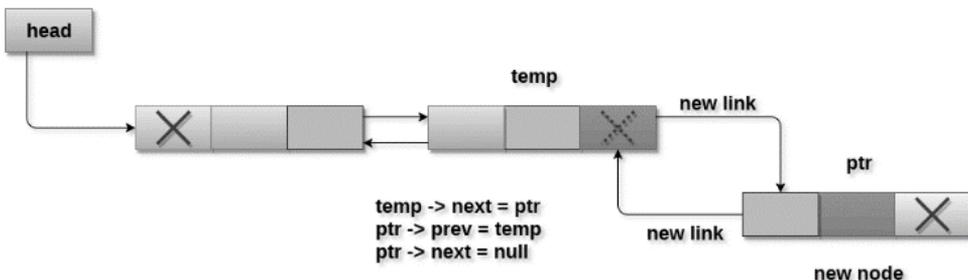


Fig. 3.9: Insertion into Doubly Linked List at the end

3.5.2.3 Insertion after specified node

To insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

The following steps are used for this purpose.

- Allocate the memory for the new node.
- Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
- The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node. Make the next pointer of ptr point to the next node of temp.
- Make the prev of the new node ptr point to temp.
- Make the next pointer of temp point to the new node ptr.
- Make the previous pointer of the next node of temp point to the new node.

Algorithm 3.3

Step 1: IF PTR = NULL

 Write OVERFLOW

 Go to Step 15

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = START

Step 6: SET I = 0

Step 7: REPEAT 8 to 10 until I

Step 8: SET TEMP = TEMP -> NEXT

STEP 9: IF TEMP = NULL

STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

 GOTO STEP 15

[END OF IF]

[END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 12: SET NEW_NODE -> PREV = TEMP

Step 13 : SET TEMP -> NEXT = NEW_NODE

Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 15: EXIT

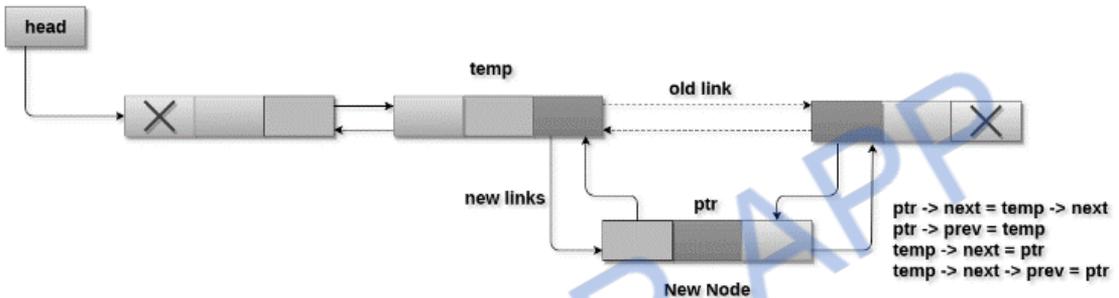


Fig. 3.10: Insertion of Doubly Linked List after specified node

3.5.2.4 Deletion at beginning

- Deletion in doubly linked list at the beginning is the simplest operation.
- Just need to copy the head pointer to pointer ptr and shift the head pointer to its next.
- Make the prev of this new head node point to NULL.
- Now free the pointer ptr by using the free function.

Algorithm 3.4

STEP 1: IF HEAD = NULL
 WRITE UNDERFLOW
 GOTO STEP 6
 STEP 2: SET PTR = HEAD
 STEP 3: SET HEAD = HEAD → NEXT
 STEP 4: SET HEAD → PREV = NULL
 STEP 5: FREE PTR
 STEP 6: EXIT

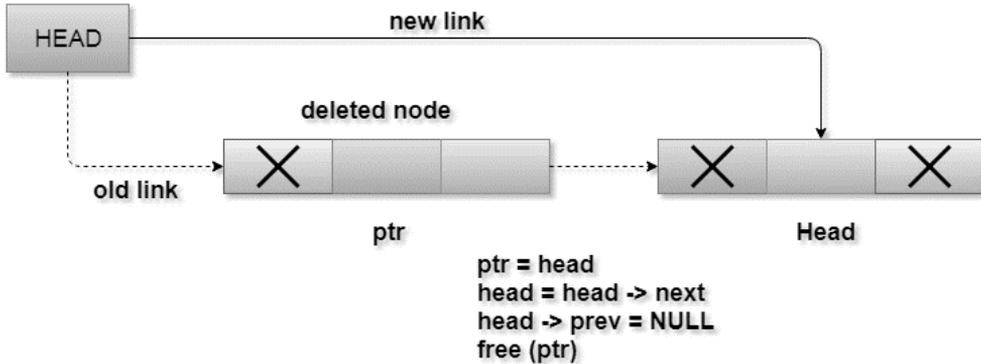


Fig. 3.11: Deletion in Doubly Linked List from beginning

3.5.2.5 Deletion at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position. To delete the last node of the list, following steps are performed.

- If the list is already empty then the condition `head == NULL` will become true and therefore the operation cannot be carried on.
- If there is only one node in the list then the condition `head -> next == NULL` become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list.
- The ptr would point to the last node of the list at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.
- Free the pointer as this the node which is to be deleted.

Algorithm 3.5:

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT

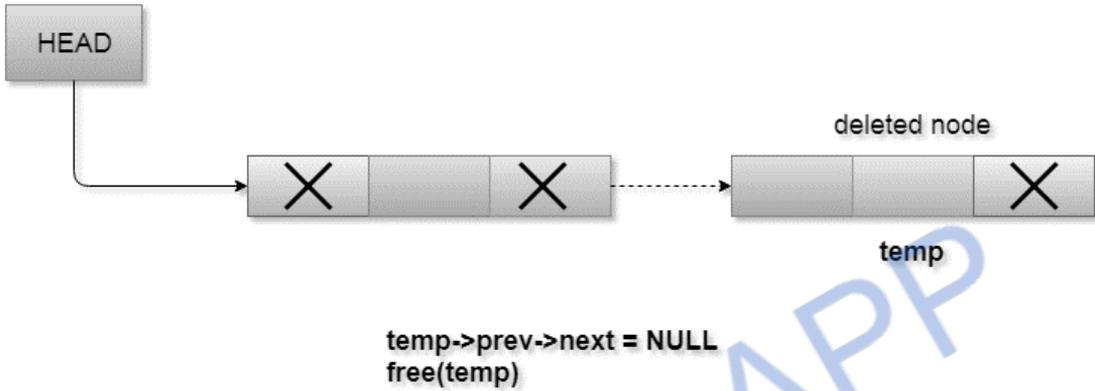


Fig. 3.12: Deletion in Doubly Linked List at the end

3.5.2.6 Deletion of the node having given data

- Copy the head pointer into a temporary pointer temp.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
- Otherwise, make the pointer ptr point to the node which is to be deleted.
- Make the next of temp point to the next of ptr.
- Make the previous of next node of ptr point to temp. free the ptr.

Algorithm 3.6

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

3.22 Linear Data Structures

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP

Step 8: FREE PTR

Step 9: EXIT

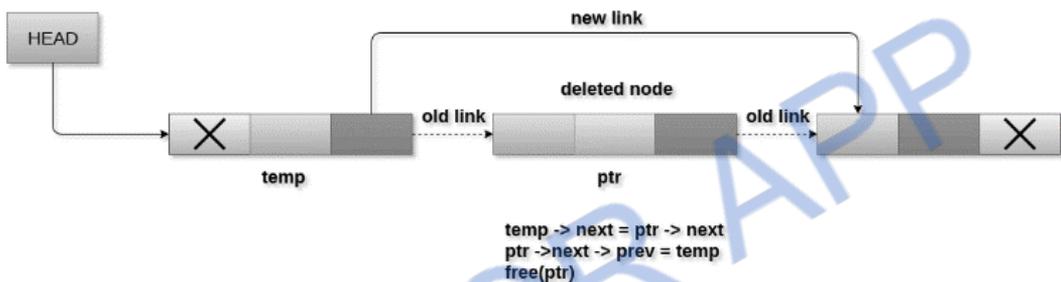


Fig. 3.13: Deletion of a specified node in Doubly Linked List at the end

3.5.2.7 Searching for a specific node

To search for a specific element in the list, traverse the list in order. The following operations to be performed in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm 3.7

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW"

GOTO STEP 8

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Set i = 0

Step 4: Repeat step 5 to 7 while PTR != NULL

Step 5: IF PTR → data = item

 return i

 [END OF IF]

Step 6: i = i + 1

Step 7: PTR = PTR → next

Step 8: Exit

3.5.2.8 Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose,

- Copy the head pointer in any of the temporary pointer ptr.
- Traverse through the list by using while loop.
- Keep shifting value of pointer variable ptr until we find the last node.
- The last node contains null in its next part.

Algorithm 3.8

Step 1: IF HEAD == NULL

 WRITE "UNDERFLOW"

 GOTO STEP 6

 [END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

Step 6: Exit

3.5.3 Advantages of Doubly Linked Lists

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.

3.24 Linear Data Structures

- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List.

3.5.4 Disadvantages of Doubly Linked Lists

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.

3.6 CIRCULAR LINKED LIST

- In a circular singly linked list, the last node of the list contains a pointer to the first node of the list.
- We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly linked list has no beginning and no ending.
- There is no null value present in the next part of any of the nodes.
- Circular linked list are mostly used in task maintenance in operating systems. Fig. 3.14 shows a circular singly linked list.

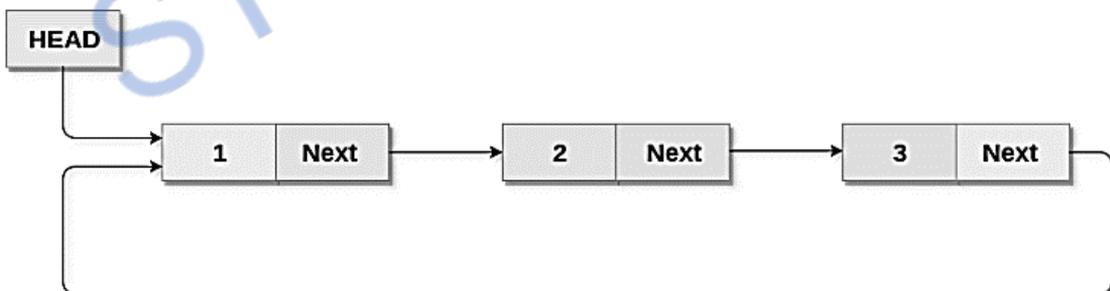


Fig. 3.14: Circular Singly Linked List.

3.6.1 Memory Representation of circular linked list

- Fig 3.15 shows the memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in

the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

- However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

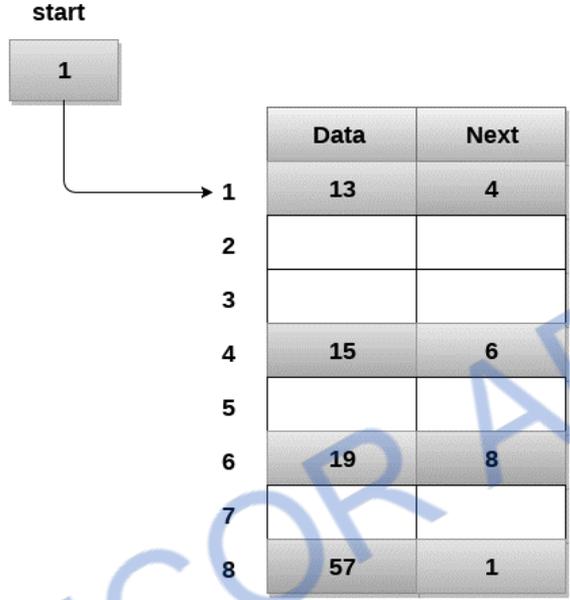


Fig. 3.15: Memory Representation of Circular Linked List

3.6.2 Operations on Circular Singly linked list

Table 3.2 describes all the operations performed on a Doubly Linked List

Table 3.2: Operations on Doubly Linked List

Sl. No.	Operation	Description
1.	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2.	Insertion at end	Adding a node into circular singly linked list at the end.
3.	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
4.	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.

5.	Traversing	Visiting each element of the list at least once in order to perform some specific operation.
----	------------	--

3.6.2.1 Insertion at the beginning

- There are two scenarios in which a node can be inserted in a circular singly linked list at the beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.
- Firstly, allocate the memory space for the new node by using the malloc method of C language.
- In the first scenario, the condition `head == NULL` will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only.
- Also need to make the head pointer point to this node.
- In the second scenario, the condition `head == NULL` will become false which means that the list contains at least one node.
- In this case, traverse the list in order to reach the last node of the list.
- At the end of the loop, the pointer temp would point to the last node of the list.
- Make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list.
- Therefore the next pointer of temp will point to the new node ptr.

Algorithm 3.9

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = HEAD

Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step 11: EXIT

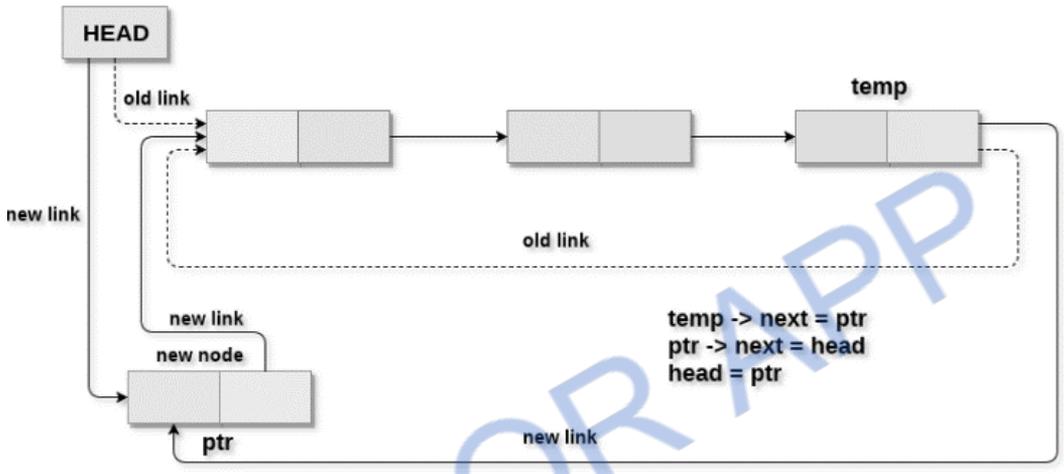


Fig. 3.16: Insertion into a Circular Linked List at the beginning

3.6.2.2 Insertion at the end

- Allocate the memory space for the new node by using the malloc method of C language.
- In the first scenario, the condition head == NULL will be true.
- Also make the head pointer point to this node.
- In the second scenario, the condition head == NULL will become false which means that the list contains at least one node.
- In this case, traverse the list in order to reach the last node of the list.
- At the end of the loop, the pointer temp would point to the last node of the list.
- The existing last node i.e. temp must point to the new node ptr

Algorithm 3.10

Step 1: IF PTR = NULL
Write OVERFLOW

3.28 Linear Data Structures

Go to Step 1

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET TEMP = HEAD

Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10: EXIT

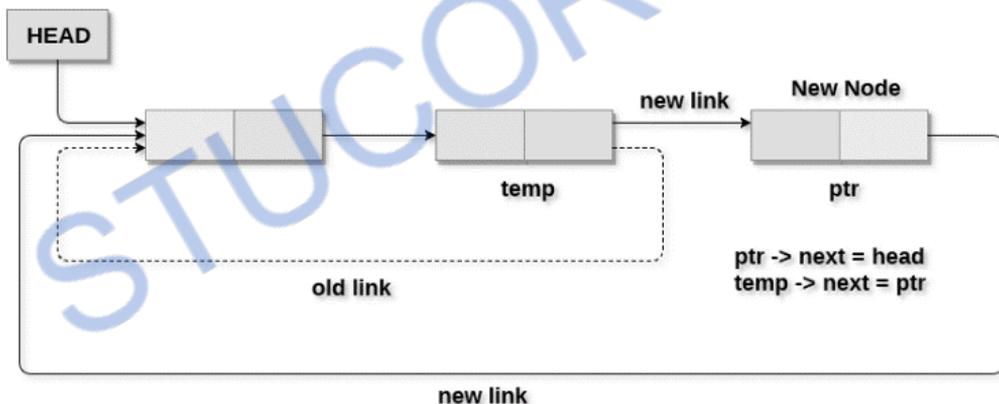


Fig. 3.17: Insertion into a Circular Linked List at the end

3.6.2.3 Deletion at the beginning

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments. There are three scenarios of deleting a node from circular singly linked list at beginning.

- Scenario 1: (The list is Empty) - If the list is empty then the condition head == NULL will become true, in this case, just to print underflow on the screen and make exit.

- Scenario 2: (The list contains single node) - If the list contains single node then, the condition $head \rightarrow next == head$ will become true. In this case, delete the entire list and make the head pointer free.
- Scenario 3: (The list contains more than one node) - If the list contains more than one node then, in that case, traverse the list by using the pointer ptr to reach the last node of the list.
- At the end of the loop, the pointer ptr point to the last node of the list.
- The last node of the list will point to the next of the head node.
- Now, free the head pointer by using the free() method.
- Make the node pointed by the next of the last node, the new head of the list.

Algorithm 3.11

```

Step 1: IF HEAD = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]

Step 2: SET PTR = HEAD
Step 3: Repeat Step 4 while PTR → NEXT != HEAD
Step 4: SET PTR = PTR → next
    [END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT
Step 6: FREE HEAD
Step 7: SET HEAD = PTR → NEXT
Step 8: EXIT
    
```

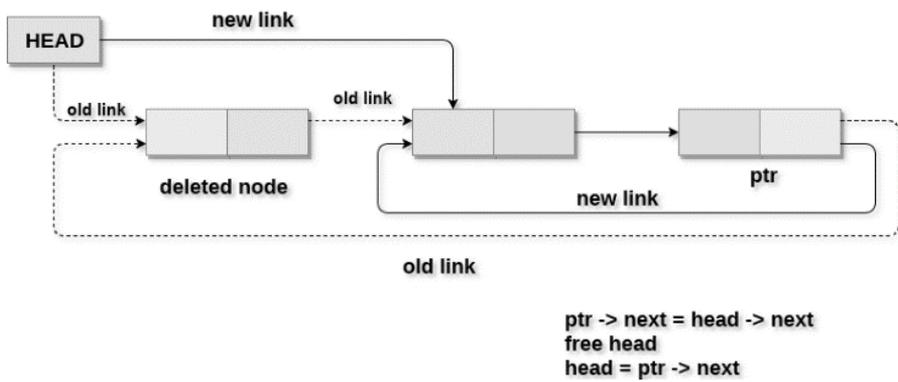


Fig. 3.18: Deletion in a Circular Linked List at beginning

3.6.2.4 Deletion at the end

- Scenario 1 (the list is empty) - If the list is empty then the condition $head == NULL$ will become true, in this case, just to print underflow on the screen and make exit.
- Scenario 2(the list contains single element) - If the list contains single node then, the condition $head \rightarrow next == head$ will become true. In this case, delete the entire list and make the head pointer free.
- Scenario 3(the list contains more than one element) - If the list contains more than one element, then in order to delete the last element, reach the last node. Also keep track of the second last node of the list. For this purpose, the two pointers ptr and $preptr$ are defined.
- Make just one more pointer adjustment. We need to make the next pointer of $preptr$ point to the next of ptr (i.e. $head$) and then make pointer ptr free.

Algorithm 3.12

Step 1: IF $HEAD = NULL$

 Write UNDERFLOW

 Go to Step 8

 [END OF IF]

Step 2: SET $PTR = HEAD$

Step 3: Repeat Steps 4 and 5 while $PTR \rightarrow NEXT \neq HEAD$

Step 4: SET $PREPTR = PTR$

Step 5: SET $PTR = PTR \rightarrow NEXT$

 [END OF LOOP]

Step 6: SET $PREPTR \rightarrow NEXT = HEAD$

Step 7: FREE PTR

Step 8: EXIT

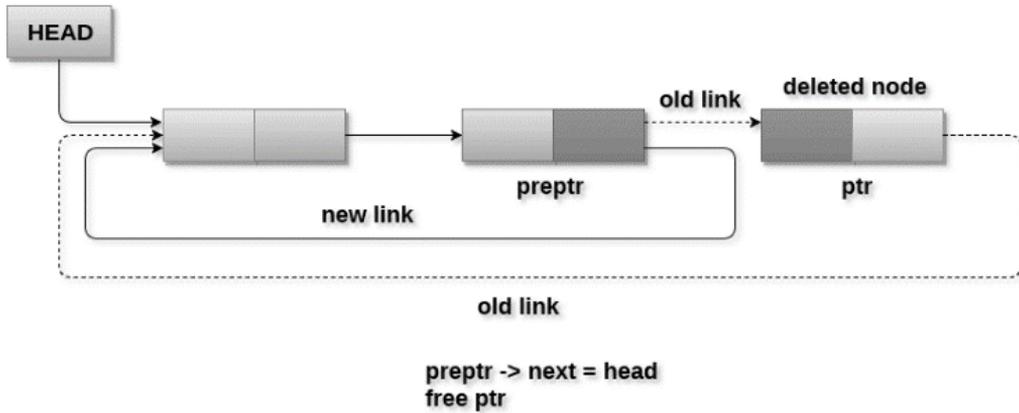


Fig. 3.19: Deletion in a Circular Linked List at the end

3.6.2.5 Searching

- Searching in circular singly linked list needs traversing across the list.
- The item which is to be searched in the list is matched with each node data of the list once.
- If the match found then the location of that item is returned otherwise -1 is returned.

Algorithm 3.13

STEP 1: SET PTR = HEAD

STEP 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head

STEP 6: if ptr → data = item

write i+1

RETURN

3.32 Linear Data Structures

End of IF

STEP 7: $I = I + 1$

STEP 8: $PTR = PTR \rightarrow NEXT$

[END OF LOOP]

STEP 9: EXIT

3.6.2.5 Searching

- Traversing in circular singly linked list can be done through a loop.
- Initialize the temporary pointer variable temp to head pointer and run the while loop until the next pointer of temp becomes head.

Algorithm 3.14

STEP 1: SET $PTR = HEAD$

STEP 2: IF $PTR = NULL$

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL $PTR \rightarrow NEXT \neq HEAD$

STEP 5: PRINT $PTR \rightarrow DATA$

STEP 6: $PTR = PTR \rightarrow NEXT$

[END OF LOOP]

STEP 7: PRINT $PTR \rightarrow DATA$

STEP 8: EXIT

3.6.3 Advantages of Circular Linked Lists

It is possible to traverse from the last node back to the first i.e. the head node.

- The starting node does not matter as we can traverse each and every node despite whatever node we keep as the starting node.
- The previous node can be easily identified.
- There is no need for a NULL function to code. The circular list never identifies a NULL identifier unless it is fully assigned.
- Circular linked lists are beneficial for end operations as start and finish coincide

3.6.4 Disadvantages of Circular Linked Lists

- If the circular linked list is not handled properly then it can lead to an infinite loop as it is circular in nature.
- In comparison with singly-linked lists, doubly linked lists are more complex in nature
- Direct accessing of elements is not possible.
- It is generally a complex task to reverse a circular linked list

3.7 STACK ADT

- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- Stack has one end, whereas the Queue has two ends (front and rear).
- It contains only one pointer top pointer pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

3.7.1 Working of Stack

- Stack works on the LIFO pattern. As we can observe in figure 3.20, there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.
- Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that

3.34 Linear Data Structures

the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

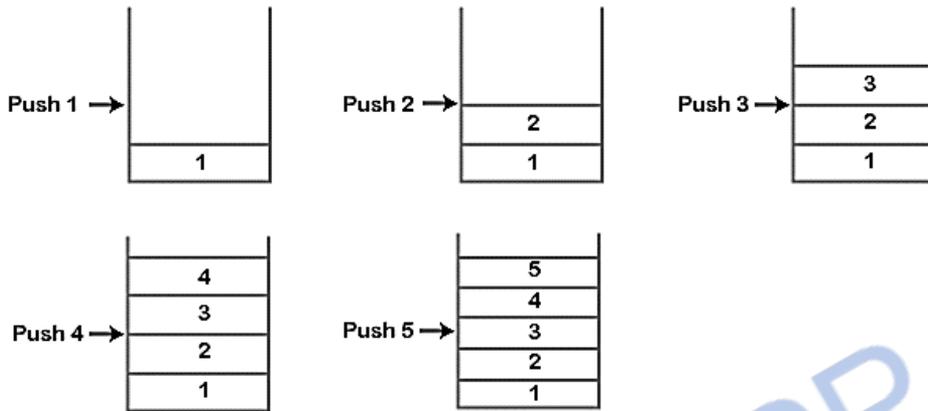


Fig. 3.20: Working principle of a Stack

3.7.2 Operations on Stack

The following are some common operations implemented on the stack:

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack

3.7.2.1 PUSH operation

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.

- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., $top = top + 1$, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.

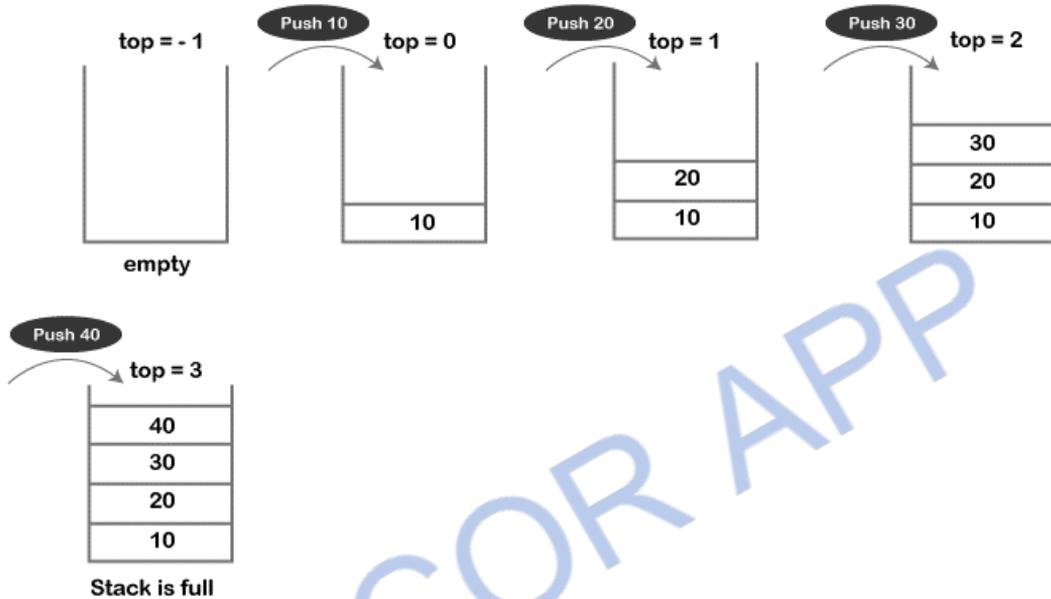
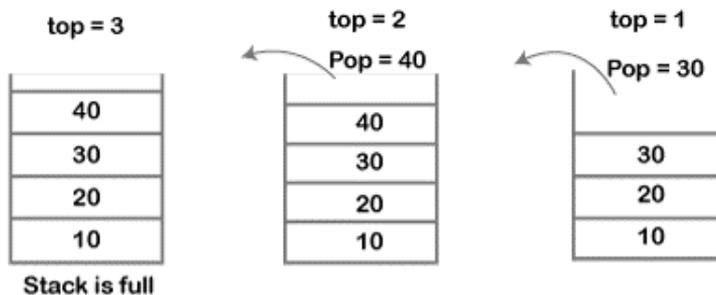


Fig. 3.21 PUSH Operation in Stack

3.7.2.2 POP operation

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top
- Once the pop operation is performed, the top is decremented by 1, i.e., $top = top - 1$.



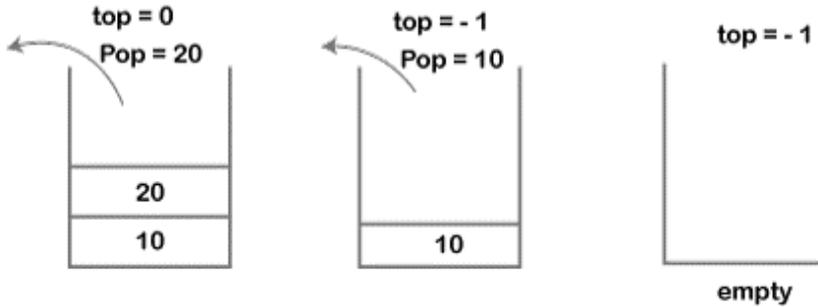


Fig. 3.22 POP Operation in Stack

3.7.3 Applications of Stack

- Balancing of symbols: Stack is used for balancing a symbol.
- String reversal: Stack is also used for reversing a string
- UNDO/REDO: It can also be used for performing UNDO/REDO operations.
- Recursion: The recursion means that the function is calling itself again.
- DFS(Depth First Search): This search is implemented on a Graph, and Graph uses the stack data structure.
- Backtracking: In order to come at the beginning of the path to create a new path, use the stack data structure
- Expression conversion: Stack can also be used for expression conversion.
 - Infix to prefix
 - Infix to postfix
 - Prefix to infix
 - Prefix to postfix
 - Postfix to infix
- Memory management: The stack manages the memory. The memory is assigned in the contiguous memory blocks.

3.8 IMPLEMENTATION OF STACK

- Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

3.8.1 Array implementation of Stack

- In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let's see how each operation can be implemented on the stack using array data structure.

3.8.1.1 Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.
 1. Increment the variable Top so that it can now refer to the next memory location.
 2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm 3.15

```

begin
    if top = n then stack full
    top = top + 1
    stack (top) := item;
end

```

Time Complexity: $O(1)$

3.8.1.1.1 Implementation of push algorithm in C language

```

void push (int val,int n) //n is size of the stack
{
    if (top == n )
        printf("\n Overflow");
    else
    {
        top = top +1;
        stack[top] = val;
    }
}

```

```
    }  
}
```

3.8.1.2 Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation.
- The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.
- The top most element of the stack is stored in another variable and then the top is decremented by 1.
- The operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm 3.16

```
begin  
    if top = 0 then stack empty;  
    item := stack(top);  
    top = top - 1;  
end;
```

Time Complexity: $O(1)$

3.8.1.2.1 Implementation of POP algorithm using C language

```
int pop ()  
{  
    if(top == -1)  
    {  
        printf("Underflow");  
        return 0;  
    }  
    else  
    {
```

```
        return stack[top - - ];  
    }  
}
```

3.8.1.3 Visiting each element of the stack (Peek operation)

- Peek operation involves returning the element which is present at the top of the stack without deleting it.
- Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm 3.17

PEEK (STACK, TOP)

```
Begin  
    if top = -1 then stack empty  
    item = stack[top]  
    return item  
End
```

Time complexity: $O(n)$

Implementation of Peek algorithm in C language

```
int peek()  
{  
    if (top == -1)  
    {  
        printf("Underflow");  
        return 0;  
    }  
    else  
    {  
        return stack [top];  
    }  
}
```

3.8.2 Linked list implementation of stack

- Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node.
- The top most node in the stack always contains null in its address field.

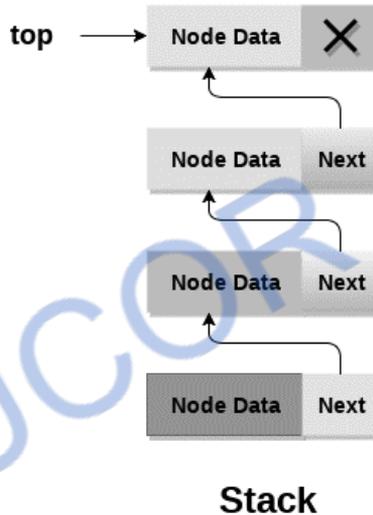


Fig. 3.23: Linked List implementation of Stack

3.8.2.1 Adding a node to the stack (Push operation)

- Adding a node to the stack is referred to as push operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.
 - Create a node first and allocate memory to it.
 - If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

- If there are some nodes in the list already, then add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity: $O(1)$

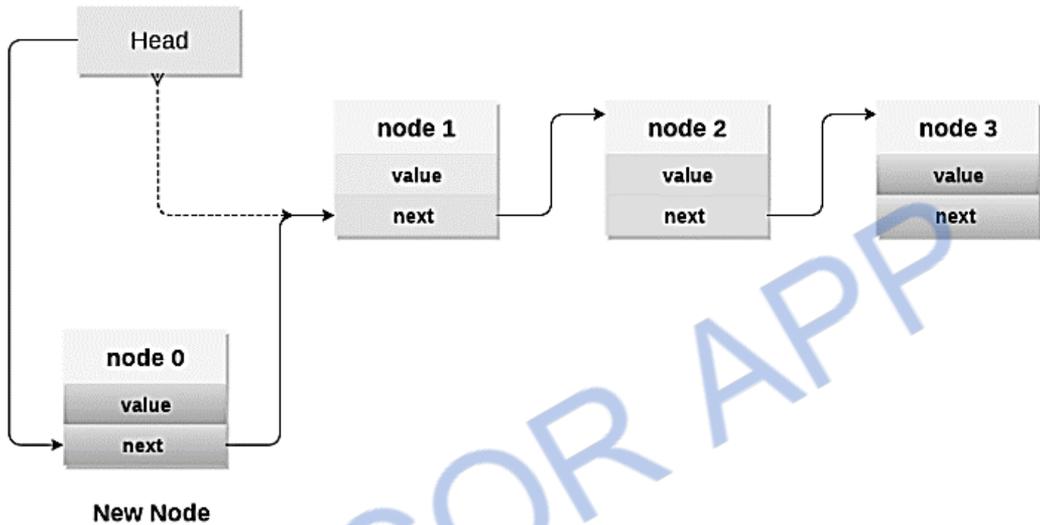


Fig 3.24 Adding new Node to Stack

3.8.2.1.1 Implementation of PUSH in C Language Program

```
void push ()
{
    int val;
    struct node *ptr =(struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
```

```
    if(head==NULL)
    {
        ptr->val = val;
        ptr -> next = NULL;
        head=ptr;
    }
    else
    {
        ptr->val = val;
        ptr->next = head;
        head=ptr;
    }
    printf("Item pushed");
}
}
```

3.8.2.2 Deleting a node from the stack (POP operation)

- Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation.
- In order to pop an element from the stack, do the following steps:
 - **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
 - **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity: O (n)

3.8.2.2.1 Implementation of POP in C Language Program

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");
    }
}
```

3.8.2.3 Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this do the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity: $O(n)$

3.8.2.3.1 Implementation of Display in C Language Program

```
void display()
{
    int i;
```

3.44 Linear Data Structures

```
struct node *ptr;
ptr=head;
if(ptr == NULL)
{
    printf("Stack is empty\n");
}
else
{
    printf("Printing Stack elements \n");
    while(ptr!=NULL)
    {
        printf("%d\n",ptr->val);
        ptr = ptr->next;
    }
}
```

3.9 APPLICATIONS OF STACK

- A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.
- Following are the various Applications of Stack in Data Structure:
 - Evaluation of Arithmetic Expressions
 - Balancing Symbols
 - Processing Function Calls
 - Backtracking
 - Reverse a Data

3.9.1 Evaluation of Arithmetic Expressions

- A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.
- In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example 3.3: $A + (B - C)$

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new notation.

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

3.9.1.1 Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

3.46 Linear Data Structures

Example 3.4: $A + B, (C - D)$ etc.

All these expressions are in infix notation because the operator comes between the operands.

3.9.1.2 Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example 3.5: $+ A B, -CD$ etc.

All these expressions are in prefix notation because the operator comes before the operands.

3.9.1.3 Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example 3.6: $AB +, CD+,$ etc.

All these expressions are in postfix notation because the operator comes after the operands.

Table 3.3 illustrates the conversion of Arithmetic Expression into various Notations

Table 3.3: Conversion of Arithmetic Expression into various Notations

Infix Notation	Prefix Notation	Postfix Notation
$A * B$	$* A B$	$AB*$
$(A+B)/C$	$/+ ABC$	$AB+C/$
$(A*B) + (D-C)$	$+*AB - DC$	$AB*DC-+$

Let's take the example of Converting an infix expression into a postfix expression

	Infix Expression	Stack	Postfix Expression
i)	A + B / C + D * (E - F) ^ G	[]	A
ii)	A + B / C + D * (E - F) ^ G	[/]	AB
iii)	A + B / C + D * (E - F) ^ G	[/]	ABC
iv)	A + B / C + D * (E - F) ^ G	[/ +]	ABC/+
v)	A + B / C + D * (E - F) ^ G	[/ +]	ABC/+D
vi)	A + B / C + D * (E - F) ^ G	[+]	ABC/+D
vii)	A + B / C + D * (E - F) ^ G	[+]	ABC/+D
viii)	A + B / C + D * (E - F) ^ G	[*]	ABC/+D
ix)	A + B / C + D * (E - F) ^ G	[*]	ABC/+D
x)	A + B / C + D * (E - F) ^ G	[(]	ABC/+DE
xi)	A + B / C + D * (E - F) ^ G	[(+]	ABC/+DE
xii)	A + B / C + D * (E - F) ^ G	[(+]	ABC/+DEF
xiii)	A + B / C + D * (E - F) ^ G	[(+]	ABC/+DEF-
xiv)	A + B / C + D * (E - F) ^ G	[(+ ^]	ABC/+DEF-
xv)	A + B / C + D * (E - F) ^ G	[(+ ^]	ABC/+DEF-G
xvi)	A + B / C + D * (E - F) ^ G	[]	ABC/+DEF-G^*+

In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression

- Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.
- Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

3.48 Linear Data Structures

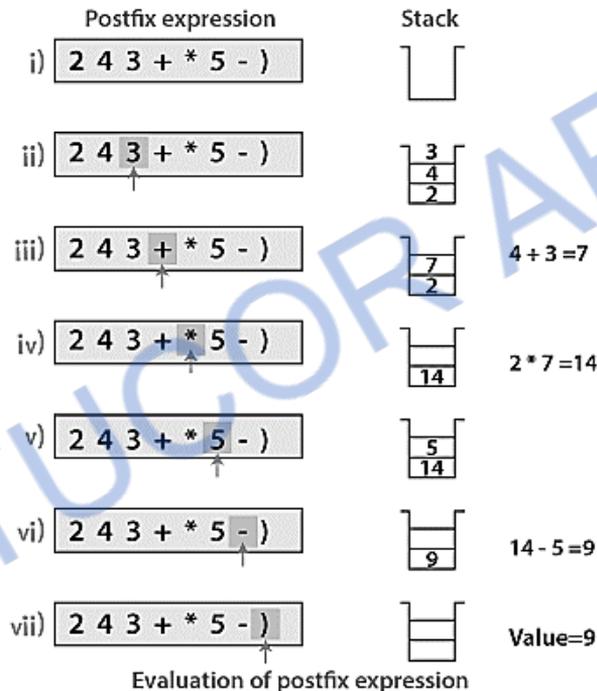
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example 3.7

Now let us consider the following infix expression $2 * (4+3) - 5$.

Its equivalent postfix expression is $2 4 3 + * 5 -$.

The following step illustrates how this postfix expression is evaluated.



3.9.2 Balancing Symbols

- Stacks can be used to check if the given expression has balanced symbols or not.
- The algorithm is very much useful in compilers.
- Each time parser reads one character at a time.
- If the character is an opening delimiter like '(', '{' or '[' then it is PUSHED in to the stack.
- When a closing delimiter is encountered like ')', '}' or ']' is encountered, the stack is popped.

- The opening and closing delimiter are then compared.
- If they match, the parsing of the string continues.
- If they do not match, the parser indicates that there is an error on the line.

A linear time $O(n)$ algorithm based on stack can be given as:-

Create a stack.

while (end of input is not reached) {

 If the character read is not a symbol to be balanced, ignore it.

 If the character is an opening delimiter like (, { or [, PUSH it into the stack.

 If it is a closing symbol like) , } ,] , then if the stack is empty report an error, otherwise POP the stack.

 If the symbol POP-ed is not the corresponding delimiter, report an error.

At the end of the input, if the stack is not empty report an error.

Example 3.8

EXAMPLE	Valid?	Description
(A+B) + (C-D)	Yes	The expression is having balanced symbol
((A+B) + (C-D)	No	One closing brace is missing.
((A+B) + [C-D])	Yes	Opening and closing braces correspond
((A+B) + [C-D])]	No	The last brace does not correspond with the first opening brace.
(A+B) + (C-D)	Yes	The expression is having balanced symbol

For tracing the algorithm let us assume that the input is () (([])

Input Symbol	Operation	Stack	Output
(Push ((
)	Pop (Test if (and A[i] match? YES		
(Push ((

3.50 Linear Data Structures

(Push (((
)	Pop ((
	Test if (and A[i] match? YES		
[Push [([
(Push (([(
)	Pop))	
	Test if (and A[i] match? YES		
]	Pop [(
	Test if [and A[i] match? YES		
)	Pop (
	Test if (and A[i] match? YES		
	Test if Stack is Empty? YES		TRUE

3.9.3 Processing Function Calls

- Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Fig. 3.24: Invoking Function Calls

- When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed.

- Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.
- Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.

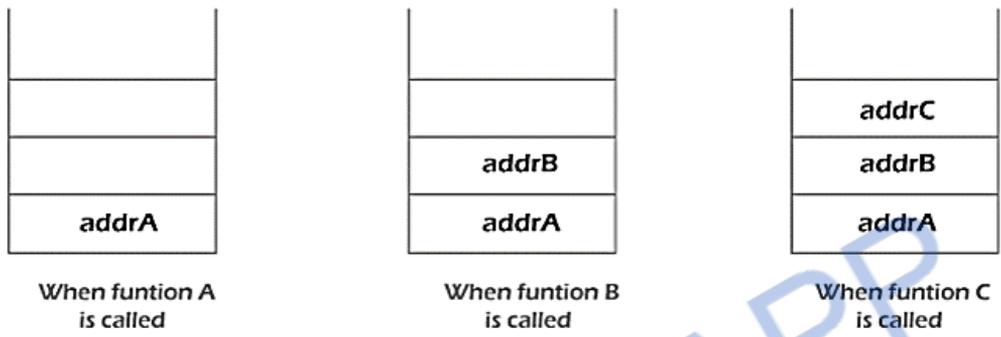


Fig. 3.25: Different states of stack

- Figure 3.26 shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack.
- Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

3.9.4 Backtracking

- Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
- Let's see how Stack is used in Backtracking in the N-Queens Problem
- For the N-Queens problem, one way we can do this is given by the following:
 - For each row, place a queen in the first valid position (column), and then move to the next row
 - If there is no valid position, then one backtracks to the previous row and try the next position

3.52 Linear Data Structures

- If one can successfully place a queen in the last row, then a solution is found. Now backtrack to find the next solution
- We can use a stack to indicate the positions of the queens. Importantly, notice that we only have to put the column positions of the queens on the stack. We can determine each queen's coordinates given only the stack. We simply combine the position of an element in the stack (the row) with the value of that element (the column) for each queen.
- Two examples of this are shown below:

* * * *			* * Q *	2	(3,2)
* Q * *	1	(2,1)	Q * * *	0	(2,0)
* * * Q	3	(1,3)	* * * Q	3	(1,3)
Q * * *	0	(0,0)	* Q * *	1	(0,1)
	stack	queen		stack	queen
		coordinates			coordinates

- Starting with a queen in the first row, first column (represented by a stack containing just "0"), we search left to right for a valid position to place another queen in the next available row.
- If we find a valid position in this row, we push this position (i.e., the column number) to the stack and start again on the next row.
- If we don't find a valid position in this row, we backtrack to the previous row -- that is to say, we pop the col position for the previous row from the stack and search for a valid position further down the row.
- Note, when the stack size gets to n, we will have placed n queens on the board, and therefore have a solution.
- Of course, there is nothing that requires there be only one solution. To find the rest, every time a solution is found, we can pretend it is not a solution, backtrack to the previous row, and proceed to find the next solution.
- Ultimately, every position in the first row will be considered. When there are not more valid positions in the first row and we need to backtrack, that's our cue that there are no more solutions to be found. Thus, we may stop searching when we try to pop from the stack, but can't as it is empty.
- Putting all this into pseudo-code form, we have the following algorithm...

Create empty stack and set current position to 0

```
Repeat {
    loop from current position to the last position until valid position found //current
row
    if there is a valid position {
        push the position to stack, set current position to 0 // move to next row
    }
    if there is no valid position {
        if stack is empty, break // stop search
        else pop stack, set current position to next position // backtrack to previous
row
    }
    if stack has size N { // a solution is found
        pop stack, set current position to next position // backtrack to find next solution
    }
}
```

3.9.5 Reverse a Data

- To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements. For example, suppose we have a string “welcome”, then on reversing it would be “emoclew”.
- There are different reversing applications:
 - Reversing a string
 - Converting Decimal to Binary

3.9.5.1 Reverse a String

- A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the last in first out property of the Stack, the first character of the Stack is on the bottom of the Stack and the last

3.54 Linear Data Structures

character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.

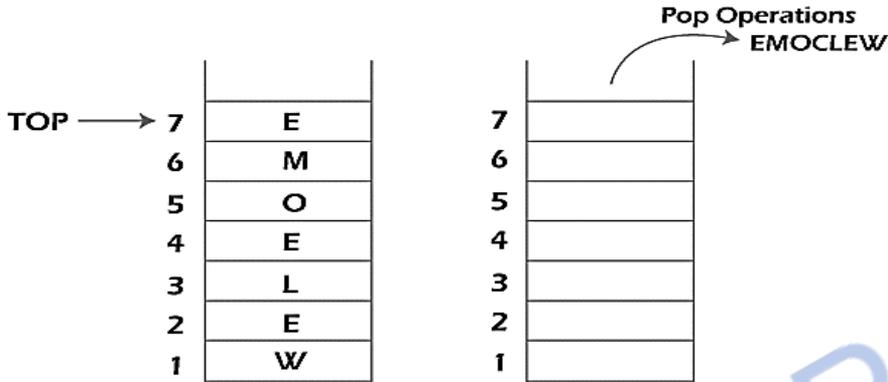


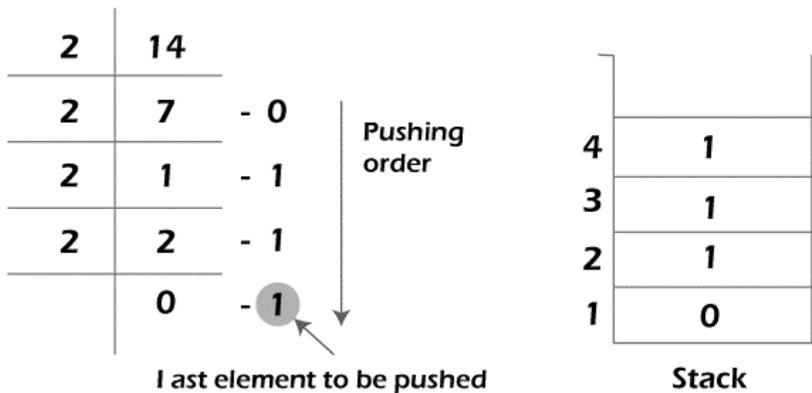
Fig. 3.26: Reverse a String using Stack

3.9.5.2 Converting Decimal to Binary

- Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form.
- For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

Example: 3.9:

Converting 14 number Decimal to Binary:



- In the above example, on dividing 14 by 2, we get seven as a quotient and one as the remainder, which is pushed on the Stack. On again dividing seven by 2, we get three as quotient and 1 as the remainder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number 1110.

3.10 QUEUE ADT

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.
- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue)
- Queue is referred to be as First In First Out list. i.e., the data item stored first will be accessed first.
- For example, people waiting in line for a rail ticket form a queue.

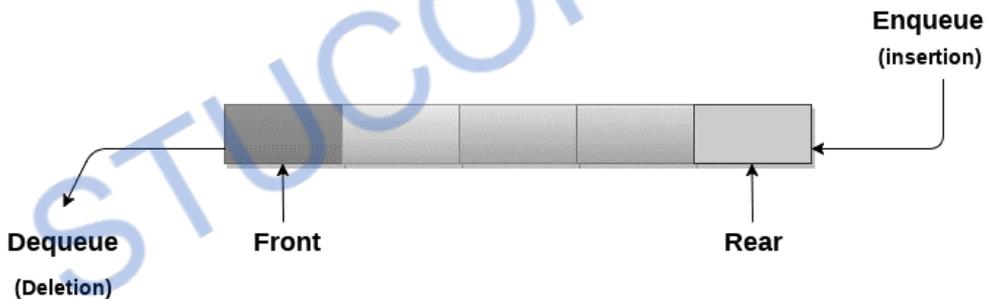


Fig. 3.27: Data Structure of Queue

3.10.1 Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.

3.56 Linear Data Structures

- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
- Queues are used in operating systems for handling interrupts.

3.10.2 Basic Operations in Queue

- **Enqueue** operation: The term "enqueue" refers to the act of adding a new element to a queue.
- **Dequeue** operation: Dequeue is the process of deleting an item from a queue. We must delete the queue member that was put first since the queue follows the FIFO principle. **Front** Operation: This works similarly to the peek operation in stacks in that it returns the value of the first element without deleting it.
- **isEmpty** Operation: The isEmpty() function is used to check if the Queue is empty or not.
- **isFull** Operation: The isFull() function is used to check if the Queue is full or not.

3.10.3 Types of Queue

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

3.10.3.1 Simple Queue or Linear Queue

- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.
- The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue

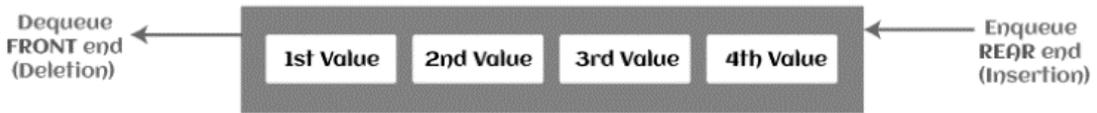


Fig. 3.28 Representation of Linear Queue

3.10.3.2 Circular Queue

- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end.
- The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.



Fig. 3.29 Representation of circular Queue

3.10.3.3 Priority Queue

- It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.
- Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

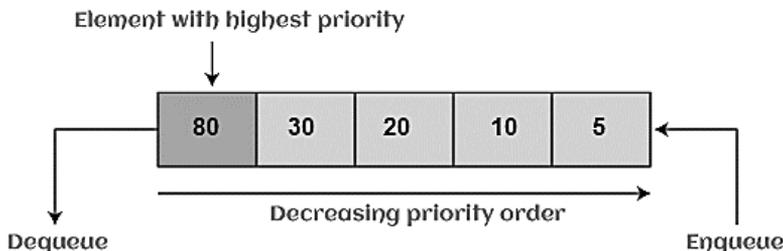


Fig. 3.30 Representation of Priority Queue

There are two types of priority queue

1. **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first.
2. **Descending priority queue** - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first.

3.10.3.4 Deque (or, Double Ended Queue)

- In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends.



Fig. 3.31 Representation of Double Ended Queue

There are two types of Deque

- **Input restricted Deque** - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.
- **Output restricted Deque** - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

3.11 PRIORITY QUEUES

- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

3.11.1 Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

poll(): This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.

add(2): This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.

poll(): It will remove '2' element from the priority queue as it has the highest priority queue.

add(5): It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

3.11.2 Types of Priority Queue

There are two types of priority queue:

Ascending order priority queue

In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

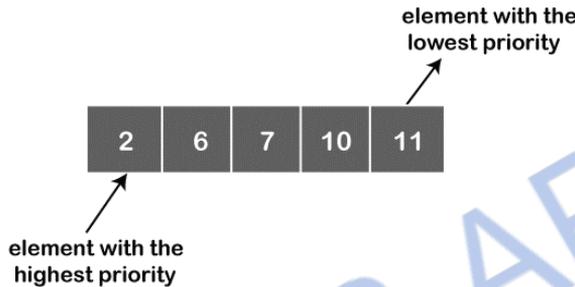


Fig. 3.32 Ascending order priority queue

Descending order priority queue:

In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

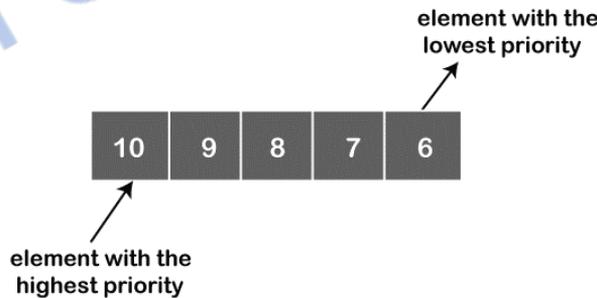


Fig. 3.33 Descending order priority queue

3.11.3 Representation of priority queue

Now, let's see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which INFO list contains the data elements, PRN list contains the priority numbers of each data element available in the INFO list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

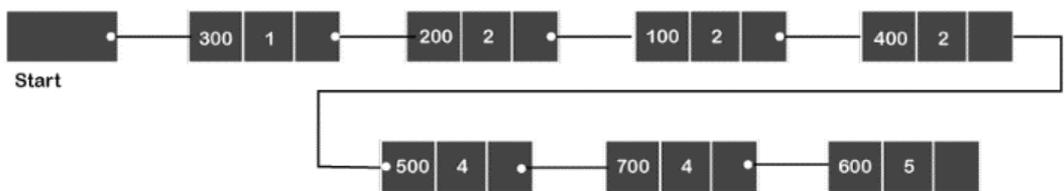
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



3.11.4 Implementation of Priority Queue

- The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the

3.62 Linear Data Structures

priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

3.11.5 Analysis of complexities using different implementations

- A comparative analysis of different implementations of priority queue is given in table 3.4

Table 3.4 Analysis of priority queue

Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

3.11.6 Heap

- A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property.
- If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap.
- It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node.
- Therefore, we can say that there are two types of heaps:

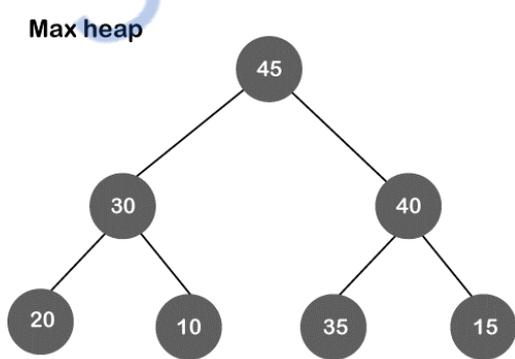


Fig. 3.34 Max heap

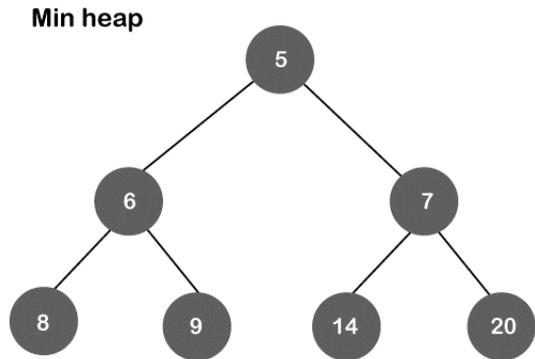


Fig. 3.35 Min heap

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.

- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.

3.11.7 Priority Queue Operations

The common operations that we can perform on a priority queue are

- Insertion,
- Deletion and
- Peek

Let's see how we can maintain the heap data structure.

3.11.7.1 Inserting the element in a priority queue (max heap)

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.

Algorithm 3.18

START

If(no node):

 Create node

Else:

 Insert node at end of heap

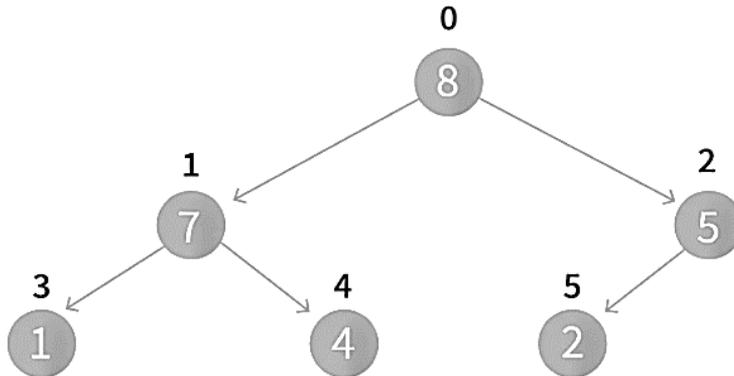
 Heapify

END

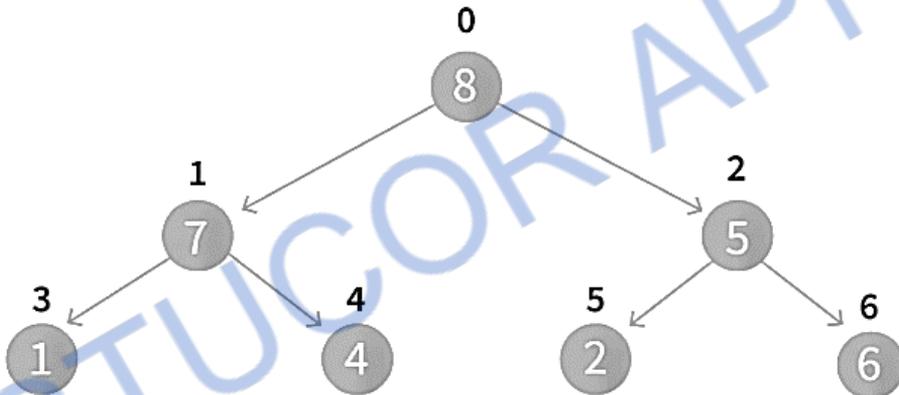
Let us now see with an example how this works:

Example 3.10

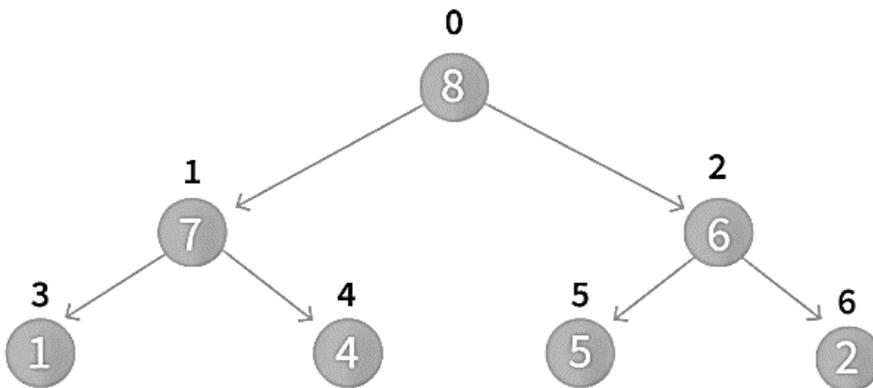
Let's say the elements are 1,4,2,7,8,5. The max-heap of these elements would look like:



Now, let's try to insert a new element, 6. Since there are nodes present in the heap, we insert this node at the end of heap so it looks like this:



Then, heapify operation is implemented. After which, the heap will look like this:



3.11.7.2 Removing the minimum element from the priority queue

In a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Algorithm 3.19:

START

If node that needs to be deleted is a leaf node:

Remove the node

Else:

Swap node that needs to be deleted with the last leaf node present.

Remove the node

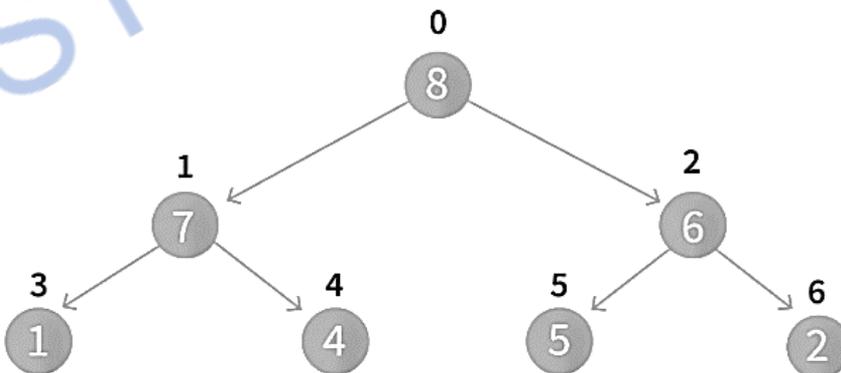
Heapify

END

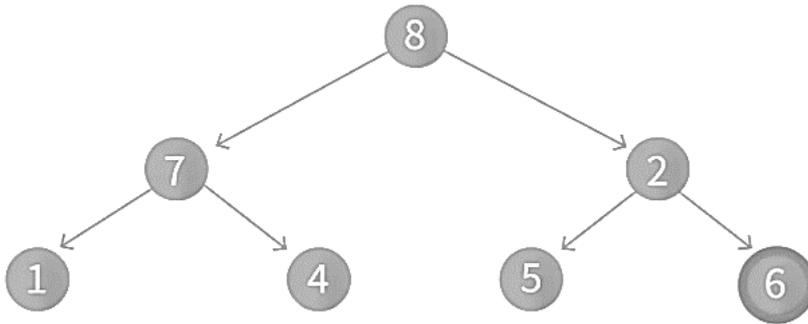
Let us now see with an example how this works:

Example: 3.11

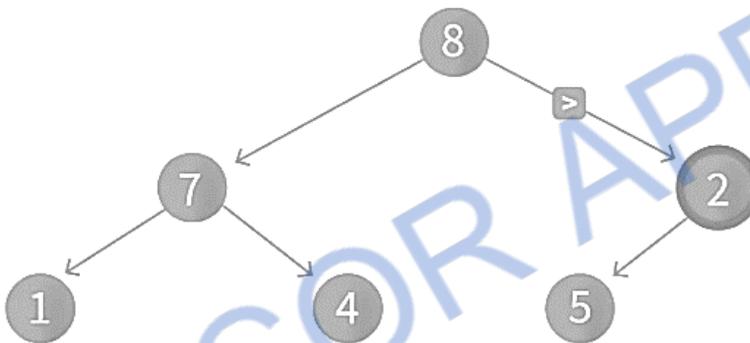
Let's say the elements are 1,4,2,7,8,5,6. The max-heap of these elements would look like:



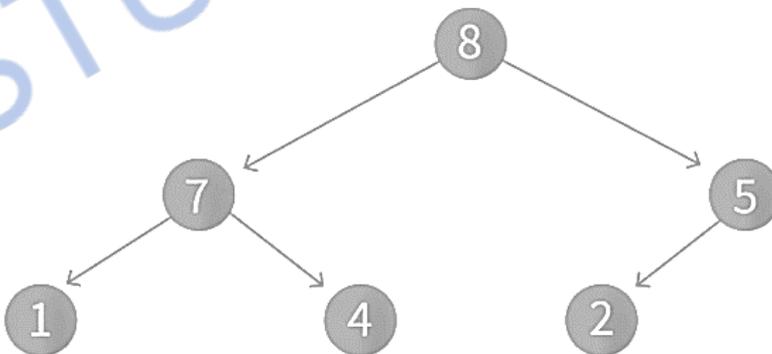
Now, let's try to delete an element, 6. Since this is not a leaf node, we swap it with the last leaf node so it looks like this:



Then, we remove the leaf node so it looks like this:



Then, heapify operation is implemented. After which, the heap will look like this:



3.11.7.3 Peeking the element from a priority queue

This will return the maximum element if a max-heap is used and the minimum number if a min-heap is used. To do both of these, we return root node. This is because in the max-heap or the min-heap the maximum or minimum element will be present at the root node respectively.

3.11.8 Applications of Priority Queue

The following are the applications of a Priority Queue:

- It is used in Dijkstra's Algorithm – To find the shortest path between nodes in a graph.
- It is used in Prim's Algorithm – To find the Minimum Spanning Tree in a weighted undirected graph.
- It is used in Heap Sort – To sort the Heap Data Structure
- It is used in Huffman Coding – A Data Compression Algorithm
- It is used in Operating Systems for:
 - Priority Scheduling – Where processes must be scheduled according to their priority.
 - Load Balancing – Where network or application traffic must be balanced across multiple servers.
 - Interrupt Handling – When a current process is interrupted, a handler is assigned to the same to rectify the situation immediately.
- A* Search Algorithm – A graph traversal and a path search algorithm

3.12 QUEUE IMPLEMENTATION

3.12.1 Array implementation of Queue

- Queue can be represented easily by using linear arrays.
- There are two variables i.e. front and rear that are implemented in the case of every queue.
- Front and rear variables point to the position from where insertions and deletions are performed in a queue.
- Initially, the value of front and queue is -1 which represents an empty queue.

Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in figure 3.36.

The above figure 3.36 shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains -1. However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above

3.68 Linear Data Structures

figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.

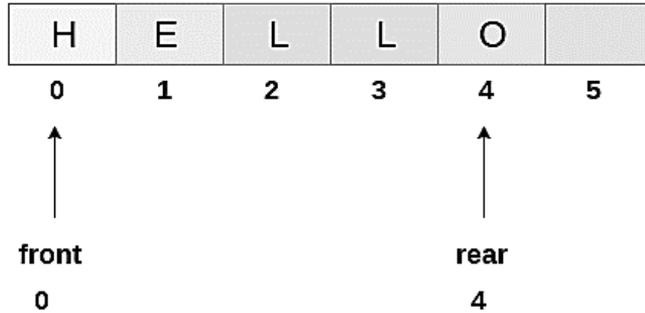


Fig. 3.36: Array representation of Queue

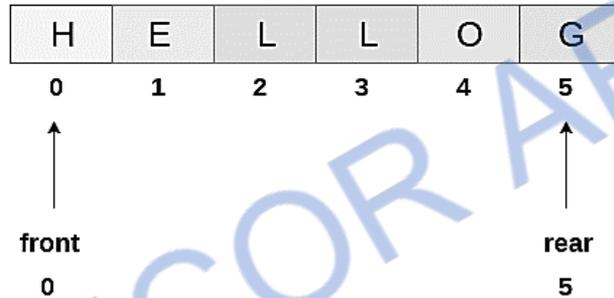


Fig. 3.37: Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. However, the queue will look something like following.

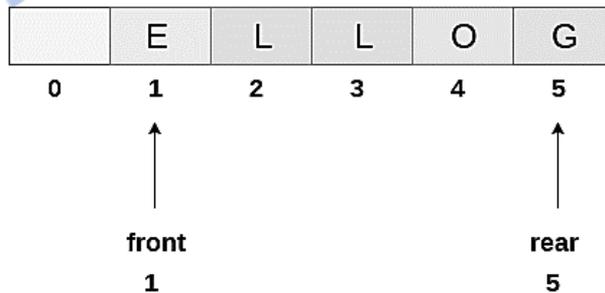


Fig. 3.38: Queue after deleting an element

3.12.1.1 Algorithm to insert any element in a queue

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm 3.20

Step 1: IF REAR = MAX - 1

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT

3.12.1.2 Implementation using C Function:

```
void insert (int queue[], int max, int front, int rear, int item)
```

```
{  
    if (rear + 1 == max)  
    {  
        printf("overflow");  
    }  
    else  
    {  
        if(front == -1 && rear == -1)  
        {  
            front = 0;  
            rear = 0;  
        }  
    }  
}
```

```
    }  
    else  
    {  
        rear = rear + 1;  
    }  
    queue[rear]=item;  
}  
}
```

3.12.1.3 Algorithm to delete an element from the queue

- If, the value of front is -1 or value of front is greater than rear, write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm 3.21

Step 1: IF FRONT = -1 or FRONT > REAR

```
    Write UNDERFLOW  
    ELSE  
    SET VAL = QUEUE[FRONT]  
    SET FRONT = FRONT + 1  
    [END OF IF]
```

Step 2: EXIT

3.12.1.4 Implementation using C Function

```
int delete (int queue[], int max, int front, int rear)  
{  
    int y;  
    if (front == -1 || front > rear)  
    {  
        printf("underflow");  
    }  
}
```

```
    }  
    else  
    {  
        y = queue[front];  
        if(front == rear)  
        {  
            front = rear = -1;  
            else  
            front = front + 1;  
        }  
        return y;  
    }  
}
```

3.12.1.5 Drawback of array implementation

- **Memory wastage:** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.
- **Deciding the array size:** One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place.

3.12.2 Linked List implementation of Queue

- One of the alternative of array implementation is linked list implementation of queue.
- The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.
- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part.

3.72 Linear Data Structures

- Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
- The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively.
- If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the figure 3.39

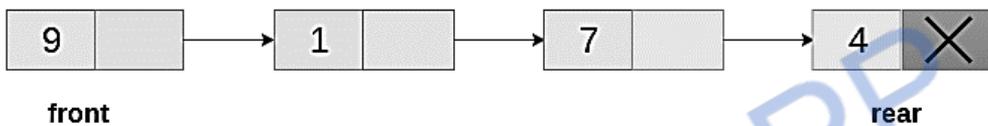


Fig. 3.39 Linked list representation of queue

3.12.2.1 Insertion on Linked Queue

- The insert operation appends the queue by adding an element to the end of the queue.
- The new element will be the last element of the queue.
- There can be two scenarios of inserting this new node ptr into the linked queue.
- In the first scenario, we insert an element into an empty queue. In this case, the condition $\text{front} = \text{NULL}$ becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.
- In the second case, the queue contains more than one element. The condition $\text{front} = \text{NULL}$ becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr.
- Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node ptr.
- Also make the next pointer of rear point to NULL.

Algorithm 3.22

Step 1: Allocate the space for the new node PTR

Step 2: SET PTR -> DATA = VAL

Step 3: IF FRONT = NULL

```
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
```

Step 4: END

3.12.2.2 Implementation using C Function

```
void insert(struct node *ptr, int item; )
{
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
    }
}
```

```
    else
    {
        rear -> next = ptr;
        rear = ptr;
        rear->next = NULL;
    }
}
```

3.13 APPLICATIONS OF QUEUE

There are several algorithms that use queues to give efficient running times. Some simple examples of queue usage are follows.

3.13.1 Type declarations for queue-array implementation

```
struct queue_record
{
    unsigned int q_max_size; /* Maximum # of elements */
    /* until Q is full */
    unsigned int q_front;
    unsigned int q_rear;
    unsigned int q_size; /* Current # of elements in Q */
    element_type *q_array;
};
typedef struct queue_record * QUEUE
```

3.13.2 Routine to test whether a queue is empty-array implementation

```
int
is_empty( QUEUE Q )
{
    return( Q->q_size == 0 );
}
```

3.13.3 Routine to make an empty queue-array implementation

```
void
make_null ( QUEUE Q )
{
Q->q_size = 0;
Q->q_front = 1;
Q->q_rear = 0;
}
```

3.13.4 Routines to enqueue-array implementation

```
unsigned int
succ( unsigned int value, QUEUE Q )
{
if( ++value == Q->q_max_size )
value = 0;
return value;
}

void
enqueue( element_type x, QUEUE Q )
{
if( is_full( Q ) )
error("Full queue");
else
{
Q->q_size++;
Q->q_rear = succ( Q->q_rear, Q );
Q->q_array[ Q->q_rear ] = x;
}
}
```

3.13.5 Other applications

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a line printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.
- Calls to large companies are generally placed on a queue when all operators are busy.
- In large universities, where resources are limited, students must sign a waiting list if all terminals are occupied. The student who has been at a terminal the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.

STUCOR APP

REVIEW QUESTIONS

PART-A

1. **What do you mean by non-linear data structure? Give example.**
 - The non-linear data structure is the kind of data structure in which the data may be arranged in hierarchical fashion. For example- Trees and graphs.
2. **List the various operations that can be performed on data structure.**
 - Various operations that can be performed on the data structure are
 - Create
 - Insertion of element
 - Deletion of element
 - Searching for the desired element
 - Sorting the elements in the data structure
 - Reversing the list of elements.
3. **What is abstract data type**
 - The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations.
4. **What is list ADT in data structure?**
 - The list ADT is a collection of elements that have a linear relationship with each other. A linear relationship means that each element of the list has a unique successor.
5. **What Are Arrays in Data Structures?**
 - An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations. Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.
6. **What is a linked list?**
 - A linked list is a set of nodes where each node has two fields 'data' and 'link'. The data field is used to store actual piece of information and link field is used to store address of next node.

7. What are the pitfall encountered in singly linked list?

- The singly linked list has only forward pointer and no backward link is provided. Hence the traversing of the list is possible only in one direction. Backward traversing is not possible.
- Insertion and deletion operations are less efficient because for inserting the element at desired position the list needs to be traversed. Similarly, traversing of the list is required for locating the element which needs to be deleted.

8. What is Singly Linked List?

- A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).

9. Define doubly linked list.

- Doubly linked list is a kind of linked list in which each node has two link fields. One link field stores the address of previous node and the other link field stores the address of the next node.

10. Write down the steps to modify a node in linked lists.

- Enter the position of the node which is to be modified.
- Enter the new value for the node to be modified.
- Search the corresponding node in the linked list.
- Replace the original value of that node by a new value.
- Display the messages as “The node is modified”.

11. Difference between arrays and lists.

- In arrays any element can be accessed randomly with the help of index of array, whereas in lists any element can be accessed by sequential access only.
- Insertion and deletion of data is difficult in arrays on the other hand insertion and deletion of data is easy in lists.

12. State the properties of LIST abstract data type with suitable example.

- It is linear data structure in which the elements are arranged adjacent to each other.
- It allows to store single variable polynomial.
- If the LIST is implemented using dynamic memory, then it is called linked list. Example of LIST are- stacks, queues, linked list.

13. State the advantages of circular lists over doubly linked list.

- In circular list the next pointer of last node points to head node, whereas in doubly linked list each node has two pointers: one previous pointer and another is next pointer.
- The main advantage of circular list over doubly linked list is that with the help of single pointer field we can access head node quickly. Hence some amount of memory get saved because in circular list only one pointer is reserved.

14. What are the advantages of doubly linked list over singly linked list?

- The doubly linked list has two pointer fields. One field is previous link field, and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in singly linked list only one pointer field is there which stores forward pointer.

15. Why is the linked list used for polynomial arithmetic?

- We can have separate coefficient and exponent fields for representing each term of polynomial. Hence there is no limit for exponent. We can have any number as an exponent.

16. What is the advantage of linked list over arrays?

- The linked list makes use of the dynamic memory allocation. Hence the user can allocate or de allocate the memory as per his requirements. On the other hand, the array makes use of the static memory location. Hence there are chances of wastage of the memory or shortage of memory for allocation.

17. What is the circular linked list?

- The circular linked list is a kind of linked list in which the last node is connected to the first node or head node of the linked list.

18. What is the basic purpose of header of the linked list?

- The header node is the very first node of the linked list. Sometimes a dummy value such - 999 is stored in the data field of header node.
- This node is useful for getting the starting address of the linked list.

19. What is the advantage of an ADT?

- Change: the implementation of the ADT can be changed without making changes in the client program that uses the ADT.

- Understandability: ADT specifies what is to be done and does not specify the implementation details. Hence code becomes easy to understand due to ADT.
- Reusability: the ADT can be reused by some program in future.

20. What is queue ADT?

- Queue is an abstract data structure, somewhat like Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

21. What is priority queue

- A priority queue is a special type of queue in which each element is associated with a priority value. Elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

22. What is stack?

- Stack is an abstract data type that serves as a collection of elements, with two main operations: Push, which adds an element to the collection, and Pop, which removes the most recently added element that was not yet removed.

23. How is Stack represented in Data Structure?

- A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list.

24. List some applications of queue data structure.

- Managing requests on a single shared resource such as CPU scheduling and disk scheduling.
- Handling hardware or real-time systems interrupts.
- Handling website traffic.
- Routers and switches in networking.
- Maintaining the playlist in media players.

25. List some applications of stack data structure.

- A Stack can be used for evaluating expressions consisting of operands and operators.
- Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.

- It can also be used to convert one form of expression to another form. It can be used for systematic Memory Management.

PART B

1. Explain Singly Link List with algorithm and examples.
2. How insertion and deletion are performed on a singly linked list? Explain with pseudocode and examples
3. Explain Doubly Linked List with examples and algorithms.
4. Explain Array based implementation of List ADT with examples and pseudocode.
5. Explain Array based implementation of Stack ADT with examples.
6. Demonstrate the implementation of circular linked list with examples.
7. Explain the applications of stack with examples.
8. Write short notes on queue ADT.
9. How are insertion and deletion operations performed in a queue? Explain with examples and pseudocode.
10. Explain the implementation of priority queue with examples and pseudocode.
11. Explain queue implementation using linked list with pseudocode and examples.
12. Explain queue implementation using stack with pseudocode and examples.

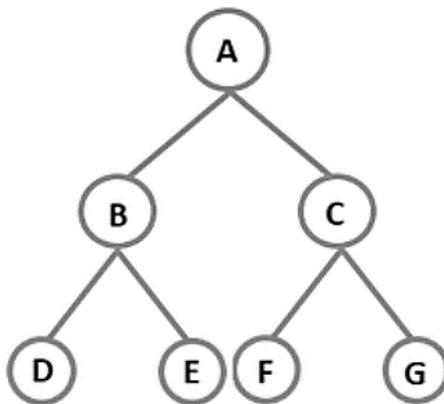
NON-LINEAR DATA STRUCTURES

Trees – Binary Trees – Tree Traversals – Expression Trees – Binary Search Tree – Hashing - Hash Functions – Separate Chaining – Open Addressing – Linear Probing – Quadratic Probing – Double Hashing – Rehashing.

4.1 INTRODUCTION TO TRESS

- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the “children”). This data structure is a specialized method to organize and store data in the computer to be used more effectively.

4.1.1 Example of Tree data structure



Here,

- Node A is the root node
- B is the parent of D and E

4.2 Non – Linear Data Structures

- D and E are the siblings
- D, E, F and G are the leaf nodes
- A and B are the ancestors of E

4.1.2 Basic Terminologies in Tree Data Structure

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node.
- **Descendant:** Any successor node on the path from the leaf node to that node.
- **Sibling:** Children of the same parent node are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

4.1.3 Properties of a Tree

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes, then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

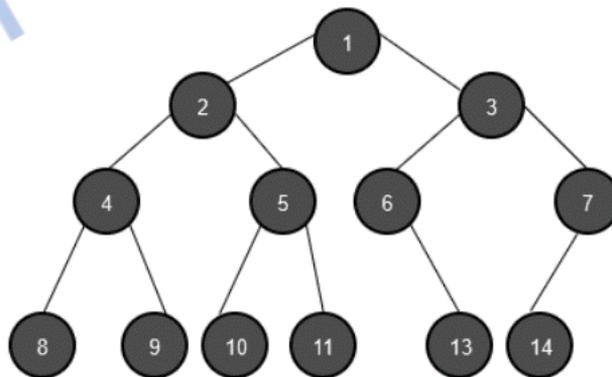
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

4.1.4 Syntax for creating a node

```
struct Node
{
    int data;
    struct Node *left_child;
    struct Node *right_child;
};
```

4.2 BINARY TREES

- Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

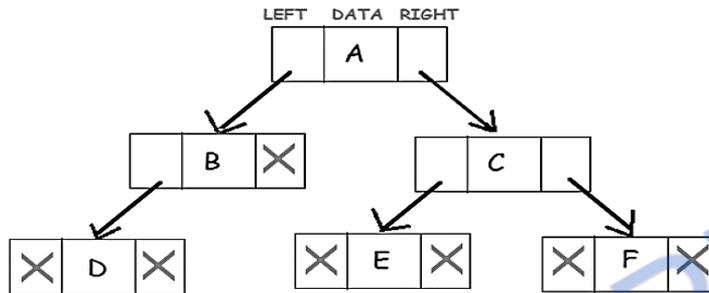


4.2.1 Binary Tree Representation

- A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.

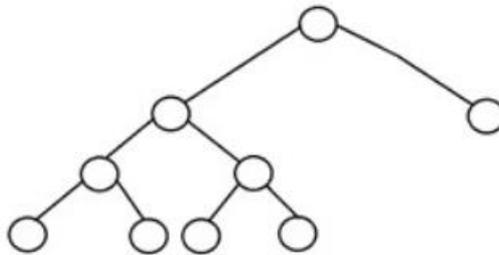
4.4 Non – Linear Data Structures

- Binary Tree node contains the following parts:
 - Data
 - Pointer to left child
 - Pointer to right child

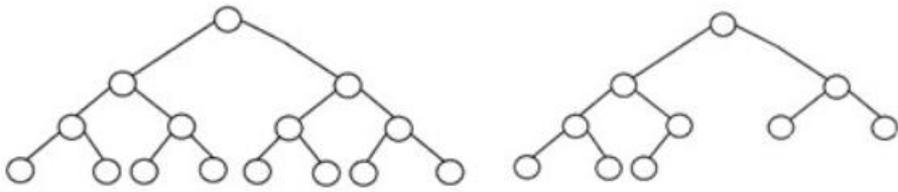


4.2.2 Types of Binary Trees

- There are various types of binary trees, and each of these binary tree types has unique characteristics. Here are each of the binary tree types in detail:
 - **Full Binary Tree**
 - It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

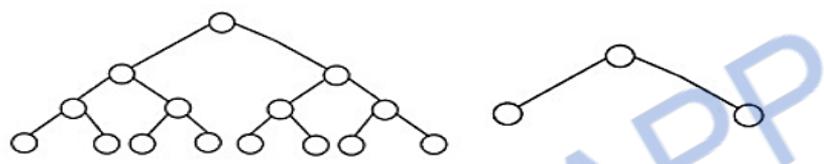


- **Complete Binary Tree**
 - A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.



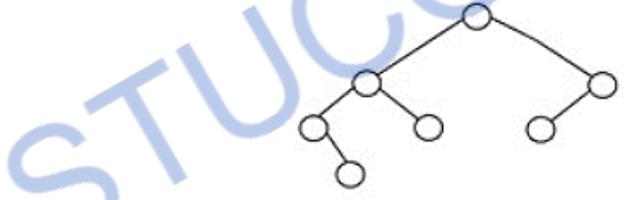
- **Perfect Binary Tree**

- ✓ A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height 'h' has $2^h - 1$ nodes.



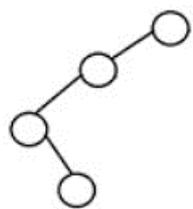
- **Balanced Binary Tree**

- ✓ A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differs by not more than 1.



- **Degenerate Binary Tree**

- ✓ A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child.



4.2.3 Benefits of Binary Trees:

- ❖ The search operation in a binary tree is faster as compared to other trees
- ❖ Only two traversals are enough to provide the elements in sorted order

4.6 Non – Linear Data Structures

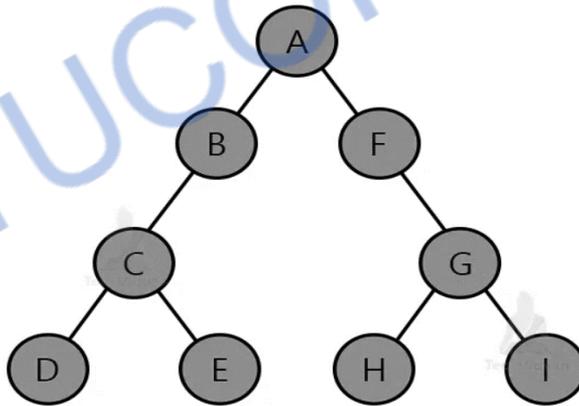
- ❖ It is easy to pick up the maximum and minimum elements
- ❖ Graph traversal also uses binary trees
- ❖ Converting different postfix and prefix expressions are possible using binary trees

4.3 TREE TRAVERSAL

- Tree traversal means visiting each node of the tree. The tree is a non-linear data structure, and therefore its traversal is different from other linear data structures.
- There is only one way to visit each node/element in linear data structures, i.e. starting from the first value and traversing in a linear order.

4.3.1 Types of Tree Traversal

- **Preorder traversal**
 - ✓ In a preorder traversal, we process/visit the root node first. Then we traverse the left subtree in a preorder manner. Finally, we visit the right subtree again in a preorder manner.
 - ✓ For example, consider the following tree:



- ✓ Here, the root node is A. All the nodes on the left of A are a part of the left subtree whereas all the nodes on the right of A are a part of the right subtree. Thus, according to preorder traversal, we will first visit the root node, so A will print first and then move to the left subtree.
- ✓ B is the root node for the left subtree. So B will print next, and we will visit the left and right nodes of B. In this manner, we will traverse the whole left subtree and then move to the right subtree. Thus, the order of visiting the nodes will be **A→B→C→D→E→F→G→H→I**.

❖ Algorithm for Preorder Traversal

- for all nodes of the tree:
 - Step 1: Visit the root node.
 - Step 2: Traverse left subtree recursively.
 - Step 3: Traverse right subtree recursively.

❖ Pseudo-code for Preorder Traversal

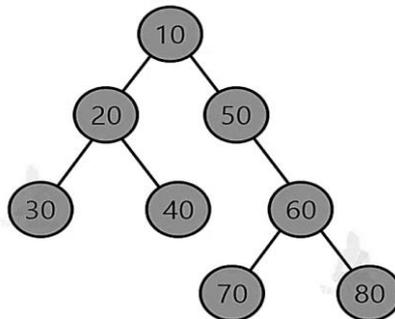
```
void Preorder(struct node* ptr)
{
    if(ptr != NULL)
    {
        printf("%d", ptr->data);
        Preorder(ptr->left);
        Preorder(ptr->right);
    }
}
```

❖ Uses of Preorder Traversal

- If we want to create a copy of a tree, we make use of preorder traversal.
- Preorder traversal helps to give a prefix expression for the expression tree.

• Inorder Traversal

- ✓ In an inorder traversal, we first visit the left subtree, then the root node and then the right subtree in an inorder manner.
- ✓ Consider the following tree:



- ✓ In this case, as we visit the left subtree first, we get the node with the value 30 first, then 20 and then 40. After that, we will visit the root node and print it. Then comes the turn of the right subtree. We will traverse the right subtree in a similar manner. Thus, after performing the inorder traversal, the order of nodes will be **30→20→40→10→50→70→60→80**.

❖ Algorithm for Inorder Traversal

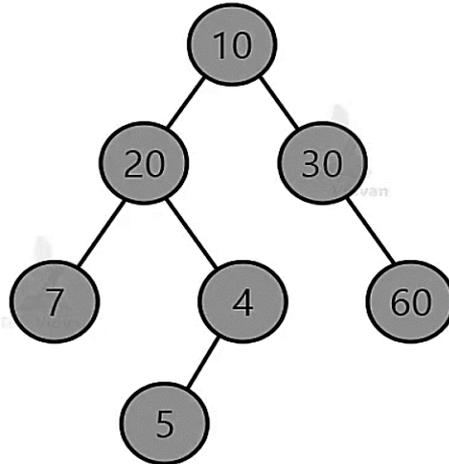
- for all nodes of the tree:
 - Step 1: Traverse left subtree recursively.
 - Step 2: Visit the root node.
 - Step 3: Traverse right subtree recursively.

❖ Pseudo-code for Inorder Traversal

```
void Inorder(struct node* ptr)
{
    if(ptr != NULL)
    {
        Inorder(ptr->left);
        printf("%d", ptr->data);
        Inorder(ptr->right);
    }
}
```

❖ Uses of Inorder Traversal

- It helps to delete the tree.
 - It helps to get the postfix expression in an expression tree.
- **Postorder Traversal**
 - ✓ Postorder traversal is a kind of traversal in which we first traverse the left subtree in a postorder manner, then traverse the right subtree in a postorder manner and at the end visit the root node.
 - ✓ For example, in the following tree:



➤ The postorder traversal will be $7 \rightarrow 5 \rightarrow 4 \rightarrow 20 \rightarrow 60 \rightarrow 30 \rightarrow 10$.

❖ **Algorithm for Postorder Traversal**

- or all nodes of the tree:
 - Step 1: Traverse left subtree recursively.
 - Step 2: Traverse right subtree recursively.
 - Step 3: Visit the root node.

❖ **Pseudo-code for Postorder Traversal**

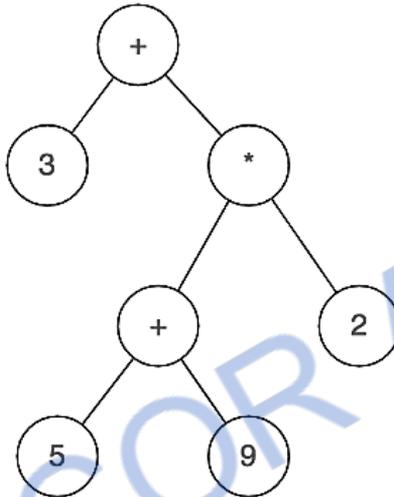
```
void Postorder(struct node* ptr)
{
    if(ptr != NULL)
    {
        Postorder(ptr->left);
        Postorder(ptr->right);
        printf("%d", ptr->data);
    }
}
```

➤ **Uses of Postorder Traversal**

- It helps to delete the tree.
- It helps to get the postfix expression in an expression tree.

4.4 EXPRESSION TREES

- The expression tree is a tree used to represent the various expressions. The tree data structure is used to represent the expressional statements. In this tree, the internal node always denotes the operators. The leaf nodes always denote the operands.
- For example, expression tree for $3 + ((5+9)*2)$ would be:



4.4.1 Properties of an Expression tree

- In this tree, the internal node always denotes the operators.
- The leaf nodes always denote the operands.
- The operations are always performed on these operands.
- The operator present in the depth of the tree is always at the highest priority.
- The operator, which is not much at the depth in the tree, is always at the lowest priority compared to the operators lying at the depth.
- The operand will always present at a depth of the tree; hence it is considered the highest priority among all the operators.

4.4.2 Construction of Expression Tree

- Let us consider a postfix expression is given as an input for constructing an expression tree. Following are the step to construct an expression tree:
 - ❖ Read one symbol at a time from the postfix expression.

- ❖ Check if the symbol is an operand or operator.
 - ❖ If the symbol is an operand, create a one node tree and push a pointer onto a stack
 - ❖ If the symbol is an operator, pop two pointers from the stack namely T1 & T2 and form a new tree with root as the operator, T1 & T2 as a left and right child
 - ❖ A pointer to this new tree is pushed onto the stack
- Thus, an expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

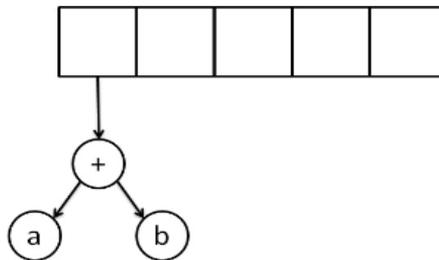
4.4.3 Example - Postfix Expression Construction

➤ **The input is: a b + c ***

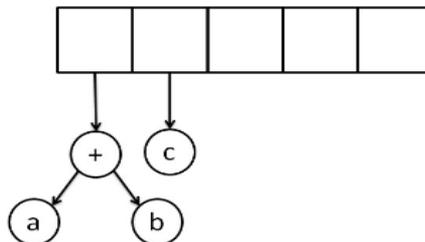
- The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



- Next, read a '+' symbol, so two pointers to tree are popped, a new tree is formed and push a pointer to it onto the stack.

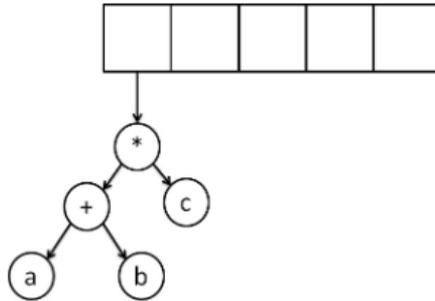


- Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.



4.12 Non – Linear Data Structures

- Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.



4.4.4 Implementation of Expression tree in C Programming language

// C program for expression tree implementation

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

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

```
/* The below structure node is defined as a node of a binary tree consists  
of left child and the right child, along with the pointer next which points to the  
next node */
```

```
struct node
```

```
{
```

```
    char info ;
```

```
    struct node* l ;
```

```
    struct node* r ;
```

```
    struct node* nxt ;
```

```
};
```

```
struct node *head=NULL;
```

```
/* Helper function that allocates a new node with the  
given data and NULL left and right pointers. */
```

```
struct node* newnode(char data)
```

```
{
```

```
struct node* node = (struct node*) malloc ( sizeof ( struct node ) );
node->info = data ;
node->l = NULL ;
node->r = NULL ;
node->nxt = NULL ;
return ( node ) ;
}
void Inorder(struct node* node)
{
    if ( node == NULL)
        return ;
    else
    {
        /* first recur on left child */
        Inorder ( node->l ) ;
        /* then print the data of node */
        printf ( "%c " , node->info ) ;
        /* now recur on right child */
        Inorder ( node->r ) ;
    }
}
void push ( struct node* x )
{
    if ( head == NULL )
        head = x ;
    else
    {
        ( x )->nxt = head ;
```

4.14 Non – Linear Data Structures

```
        head = x ;
    }
    // struct node* temp ;
    // while ( temp != NULL )
    // {
    //     printf ( " %c " , temp->info ) ;
    //     temp = temp->nxt ;
    // }
}
struct node* pop()
{
    // Popping out the top most [pointed with head] element
    struct node* n = head ;
    head = head->nxt ;
    return n ;
}
int main()
{
    char t[] = { 'X' , 'Y' , 'Z' , '*' , '+' , 'W' , '/' } ;
    int n = sizeof(t) / sizeof(t[0]) ;
    int i ;
    struct node *p , *q , *s ;
    for ( i = 0 ; i < n ; i++ )
    {
        // if read character is operator then popping two
        // other elements from stack and making a binary
        // tree
        if ( t[i] == '+' || t[i] == '-' || t[i] == '*' || t[i] == '/' || t[i] == '^' )
```

```
{
    s = newnode ( t [ i ] );
    p = pop() ;
    q = pop() ;
    s->l = q ;
    s->r = p;
    push(s);
}
else {
    s = newnode ( t [ i ] );
    push ( s ) ;
}
}
printf ( " The Inorder Traversal of Expression Tree: " );
Inorder ( s ) ;
return 0 ;
}
```

The output of the above program is

$X + Y * Z / W$

4.4.5 Use of Expression tree

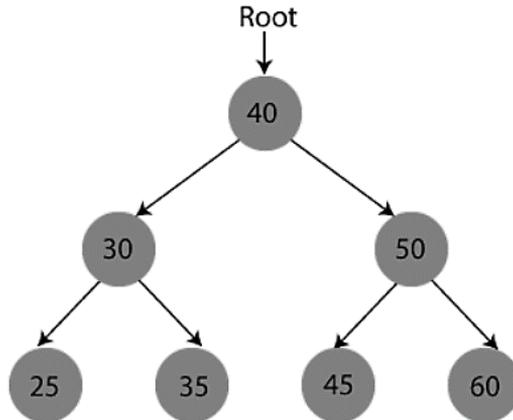
- The main objective of using the expression trees is to make complex expressions and can easily be evaluated using these expression trees.
- It is also used to find out the associativity of each operator in the expression.
- It is also used to solve the postfix, prefix, and infix expression evaluation.

4.5 BINARY SEARCH TREE

- In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

4.16 Non – Linear Data Structures

- Example for Binary Search Tree



- In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.
- Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

4.5.1 Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

4.5.2 Example of creating a binary search tree

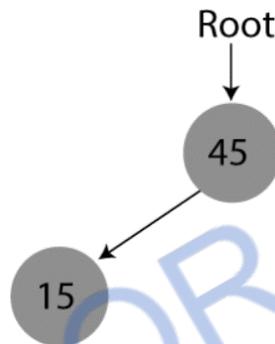
- Now, let's see the creation of binary search tree using an example. Suppose the data elements are : 45, 15, 79, 90, 10, 55, 12, 20, 50
 - First, we have to insert 45 into the tree as the root of the tree.
 - Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
 - Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.
- Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below

- Step 1 - Insert 45.



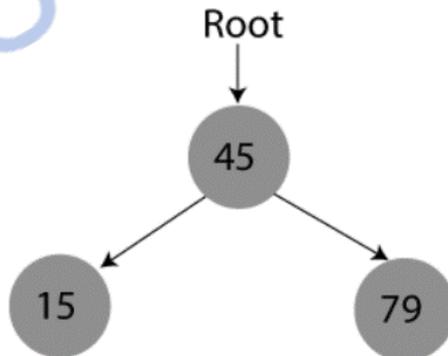
- Step 2 - Insert 15.

- As 15 is smaller than 45, so insert it as the root node of the left subtree.



- Step 3 - Insert 79.

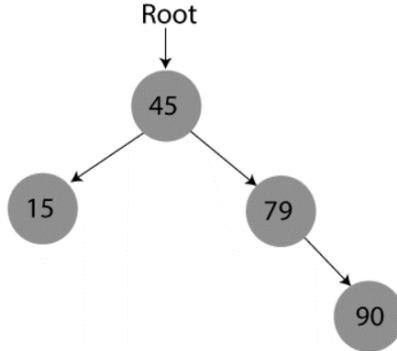
- As 79 is greater than 45, so insert it as the root node of the right subtree.



- Step 4 - Insert 90.

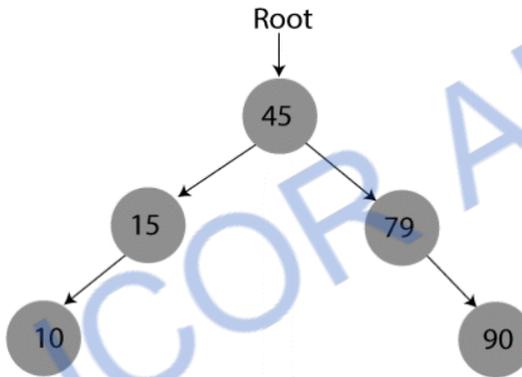
- 90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.

4.18 Non – Linear Data Structures



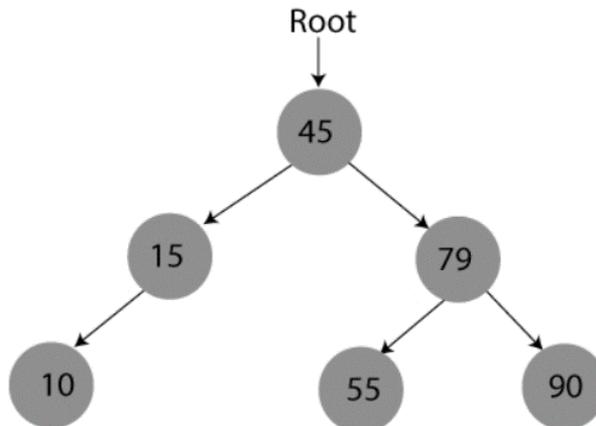
➤ Step 5 - Insert 10

- 10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



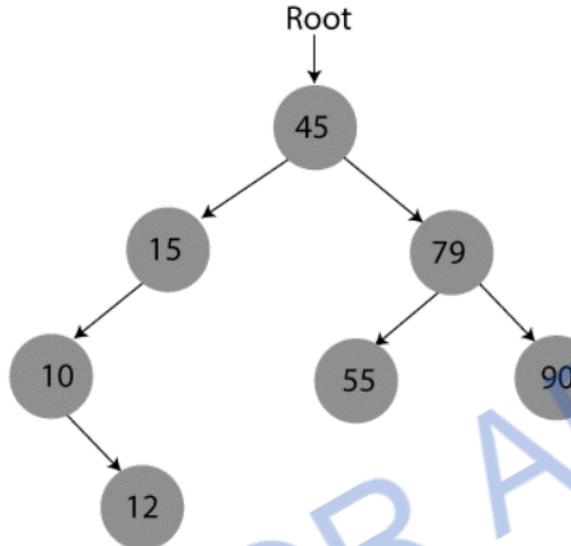
➤ Step 6 - Insert 55

- 55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



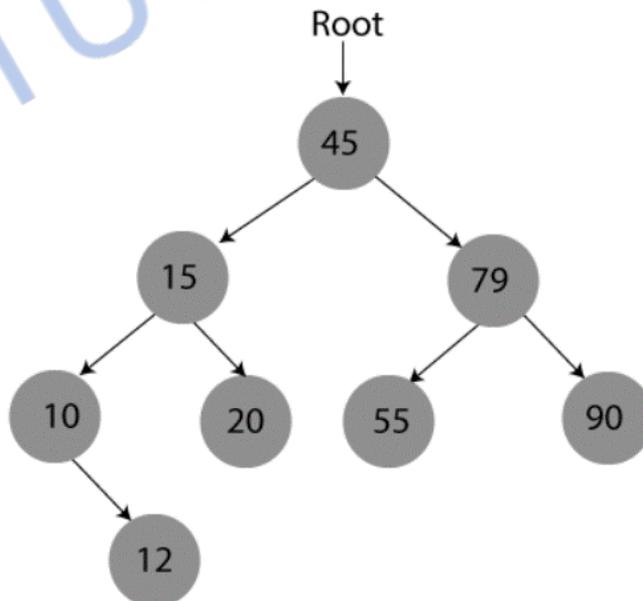
➤ Step 7 - Insert 12

- 12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



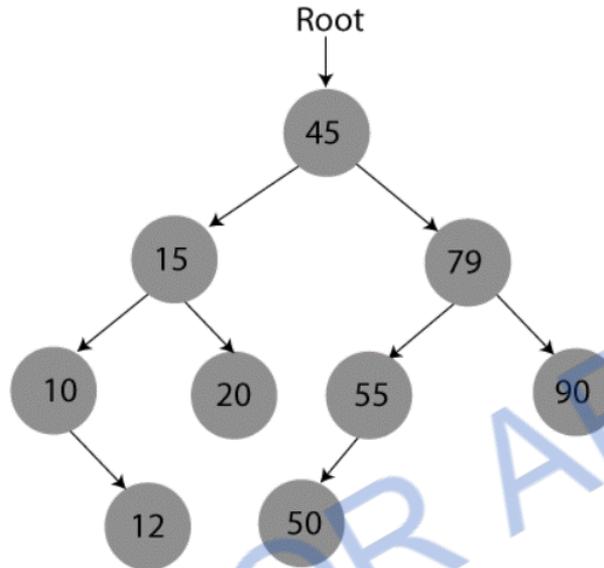
➤ Step 8 - Insert 20

- 20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



4.20 Non – Linear Data Structures

- Step 9 - Insert 50.
 - 50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



- Now, the creation of binary search tree is completed.

4.5.3 Operations performed on a Binary Search Tree

- We can perform insert, delete and search operations on the binary search tree.

4.5.3.1 Searching in Binary search tree

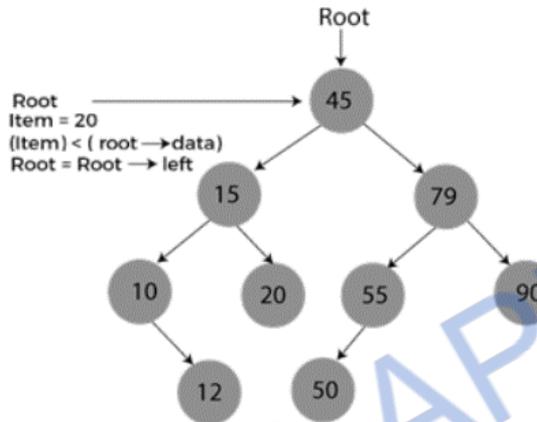
- Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order.

4.5.3.1.1 Steps involved in Searching in a Binary Search Tree

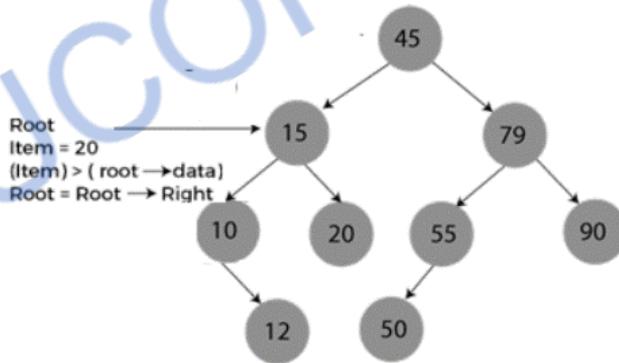
- ❖ First, compare the element to be searched with the root element of the tree.
- ❖ If root is matched with the target element, then return the node's location.
- ❖ If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
- ❖ If it is larger than the root element, then move to the right subtree.
- ❖ Repeat the above procedure recursively until the match is found.
- ❖ If the element is not found or not present in the tree, then return NULL.

- Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

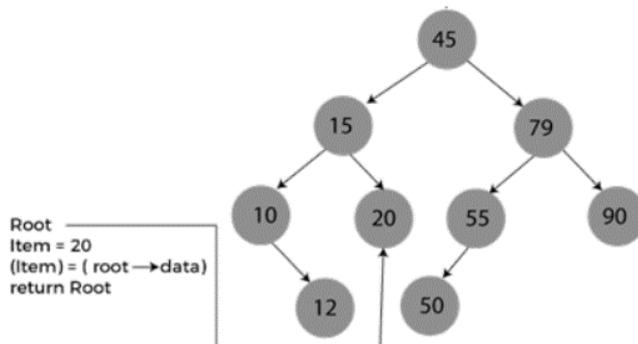
Step1:



Step2:



Step3:



4.5.3.1.2 Algorithm to search an element in Binary search tree

Search (root, item)

Step 1 - if (item = root \rightarrow data) or (root = NULL)

return root

else if (item < root \rightarrow data)

return Search(root \rightarrow left, item)

else

return Search(root \rightarrow right, item)

END if

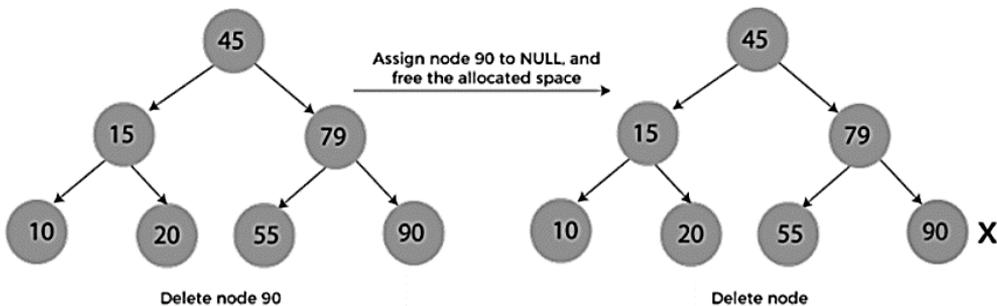
Step 2 - END

4.5.3.2 Deletion in Binary Search tree

- In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -
 - The node to be deleted is the leaf node, or,
 - The node to be deleted has only one child, and,
 - The node to be deleted has two children

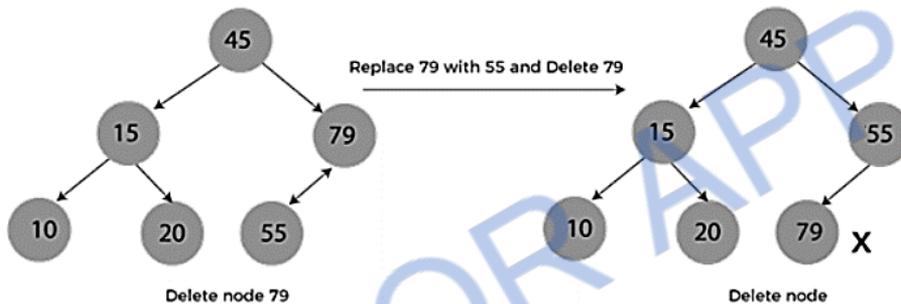
4.5.3.2.1 When the node to be deleted is the leaf node

- It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.
- We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



4.5.3.2.2 When the node to be deleted has only one child

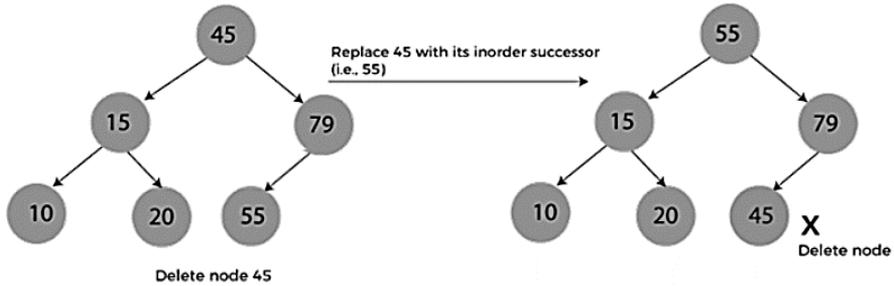
- In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.
- We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
- So, the replaced node 79 will now be a leaf node that can be easily deleted.



4.5.3.2.3 When the node to be deleted has two children

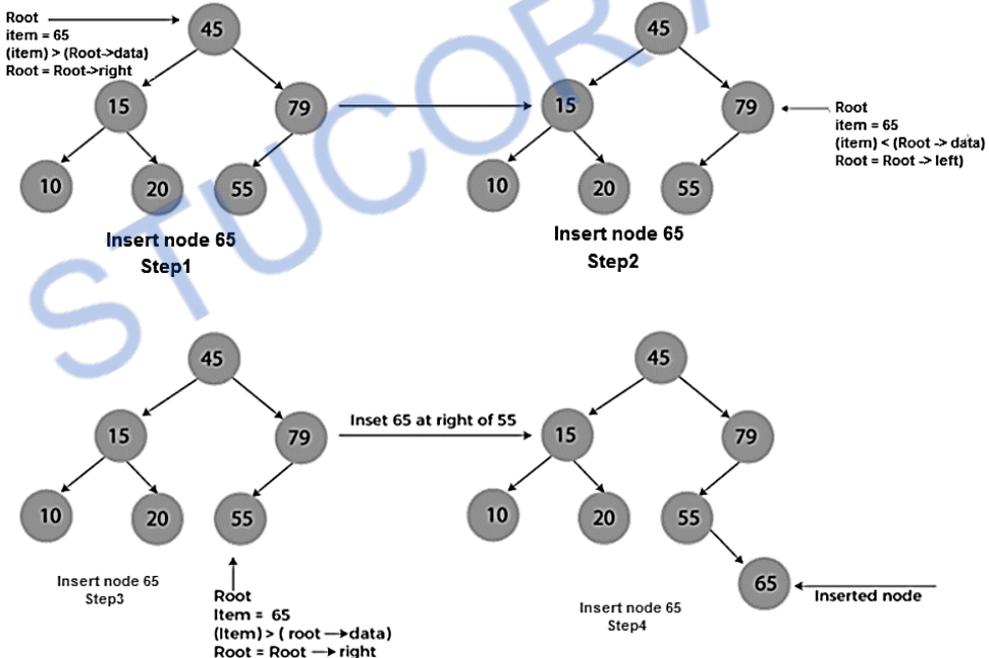
- This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -
 - First, find the inorder successor of the node to be deleted.
 - After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
 - And at last, replace the node with NULL and free up the allocated space.
- The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.
- We can see the process of deleting a node with two children from BST in the below image.
- In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

4.24 Non – Linear Data Structures



4.5.3.3 Insertion in Binary Search tree

- A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree.
- Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.



4.5.3.4 The complexity of the Binary Search tree

- Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

4.5.3.5 Implementation of Binary search tree

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *left;
    Node *right;
};
Node* create(int item)
{
    Node* node = new Node;
    node->data = item;
    node->left = node->right = NULL;
    return node;
}
/*Inorder traversal of the tree formed*/
void inorder(Node *root)
{
    if (root == NULL)
        return;

    inorder(root->left); //traverse left subtree
    cout<< root->data << " "; //traverse root node
    inorder(root->right); //traverse right subtree
}
Node* findMinimum(Node* cur) /*To find the inorder successor*/
{
    while(cur->left != NULL) {
        cur = cur->left;
    }
}
```

```
    return cur;
}
Node* insertion(Node* root, int item) /*Insert a node*/
{
    if (root == NULL)
        return create(item); /*return new node if tree is empty*/
    if (item < root->data)
        root->left = insertion(root->left, item);
    else
        root->right = insertion(root->right, item);
    return root;
}
void search(Node* &cur, int item, Node* &parent)
{
    while (cur != NULL && cur->data != item)
    {
        parent = cur;
        if (item < cur->data)
            cur = cur->left;
        else
            cur = cur->right;
    }
}
void deletion(Node*& root, int item) /*function to delete a node*/
{
    Node* parent = NULL;
    Node* cur = root;
    search(cur, item, parent); /*find the node to be deleted*/
    if (cur == NULL)
```

```
    return;
    if (cur->left == NULL && cur->right == NULL) /*When node has no
children*/
    {
        if (cur != root)
        {
            if (parent->left == cur)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            root = NULL;
        free(cur);
    }
    else if (cur->left && cur->right)
    {
        Node* succ = findMinimum(cur->right);
        int val = succ->data;
        deletion(root, succ->data);
        cur->data = val;
    }
    else
    {
        Node* child = (cur->left)? cur->left: cur->right;
        if (cur != root)
        {
            if (cur == parent->left)
                parent->left = child;
```

```
        else
            parent->right = child;
        }
    else
        root = child;
    free(cur);
}
}
int main()
{
    Node* root = NULL;
    root = insertion(root, 45);
    root = insertion(root, 30);
    root = insertion(root, 50);
    root = insertion(root, 25);
    root = insertion(root, 35);
    root = insertion(root, 45);
    root = insertion(root, 60);
    root = insertion(root, 4);
    printf("The inorder traversal of the given binary tree is - \n");
    inorder(root);
    deletion(root, 25);
    printf("\nAfter deleting node 25, the inorder traversal of the given binary tree is
- \n");
    inorder(root);
    insertion(root, 2);
    printf("\nAfter inserting node 2, the inorder traversal of the given binary tree is
- \n");
    inorder(root);
}
```

```
    return 0;  
}
```

Output

```
The inorder traversal of the given binary tree is -  
4      25      30      35      45      45      50      60  
After deleting node 25, the inorder traversal of the given binary tree is -  
4      30      35      45      45      50      60  
After inserting node 2, the inorder traversal of the given binary tree is -  
2      4      30      35      45      45      50      60
```

4.6 HASHING

- Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.
- It uses hash tables to store the data in an array format. Each value in the array has been assigned a unique index number. Hash tables use a technique to generate these unique index numbers for each value stored in an array format. This technique is called the hash technique.
- You only need to find the index of the desired item, rather than finding the data. With indexing, you can quickly scan the entire list and retrieve the item you wish. Indexing also helps in inserting operations when you need to insert data at a specific location. No matter how big or small the table is, you can update and retrieve data within seconds.
- The hash table is basically the array of elements, and the hash techniques of search are performed on a part of the item i.e. key. Each key has been mapped to a number, the range remains from 0 to table size - 1
- Types of hashing in data structure is a two-step process.
 - The hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.
 - It stores the data in a hash table. You can use a hash key to locate data quickly.

4.6.1 Examples

- In schools, the teacher assigns a unique roll number to each student. Later, the teacher uses that roll number to retrieve information about that student.

4.30 Non – Linear Data Structures

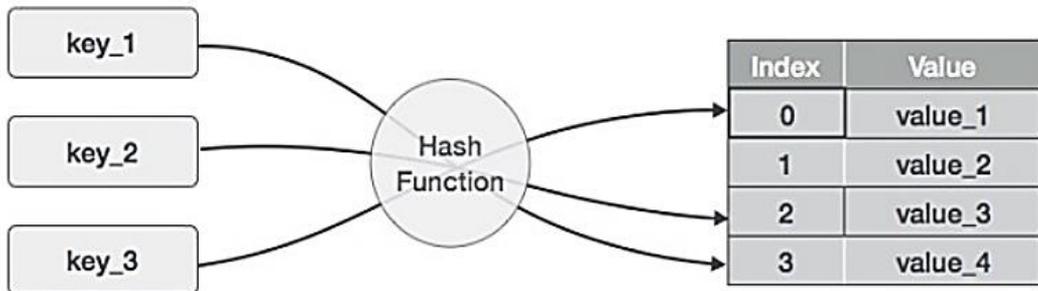
- A library has an infinite number of books. The librarian assigns a unique number to each book. This unique number helps in identifying the position of the books on the bookshelf.

4.7 HASH FUNCTION

- The hash function in a data structure maps the arbitrary size of data to fixed-sized data. It returns the following values: a small integer value (also known as hash value), hash codes, and hash sums. The hashing techniques in the data structure are very interesting, such as:
 - $\text{hash} = \text{hashfunc}(\text{key})$
 - $\text{index} = \text{hash} \% \text{array_size}$
- The hash function must satisfy the following requirements:
 - A good hash function is easy to compute.
 - A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.
 - A good hash function avoids collision when two elements or items get assigned to the same hash value.
- The three characteristics of the hash function in the data structure are:
 - Collision free
 - Property to be hidden
 - Puzzle friendly

4.7.1 Hash Table

- Hashing in data structure uses hash tables to store the key-value pairs. The hash table then uses the hash function to generate an index. Hashing uses this unique index to perform insert, update, and search operations.



- It can be defined as a bucket where the data are stored in an array format. These data have their own index value. If the index values are known then the process of accessing the data is quicker.

4.7.2 How does Hashing in Data Structure Works?

- In hashing, the hashing function maps strings or numbers to a small integer value. Hash tables retrieve the item from the list using a hashing function.
- The objective of hashing technique is to distribute the data evenly across an array. Hashing assigns all the elements a unique key. The hash table uses this key to access the data in the list.
- Hash table stores the data in a key-value pair. The key acts as an input to the hashing function. Hashing function then generates a unique index number for each value stored.
- The index number keeps the value that corresponds to that key. The hash function returns a small integer value as an output. The output of the hashing function is called the hash value.
- Let us understand hashing in a data structure with an example. Imagine you need to store some items (arranged in a key-value pair) inside a hash table with 30 cells. The values are: (3,21) (1,72) (40,36) (5,30) (11,44) (15,33) (18,12) (16,80) (38,99)
- The hash table will look like the following:

Serial Number	Key	Hash	Array Index
1	3	$3\%30 = 3$	3
2	1	$1\%30 = 1$	1
3	40	$40\%30 = 10$	10
4	5	$5\%30 = 5$	5
5	11	$11\%30 = 11$	11
6	15	$15\%30 = 15$	15
7	18	$18\%30 = 18$	18
8	16	$16\%30 = 16$	16
9	38	$38\%30 = 8$	8

4.32 Non – Linear Data Structures

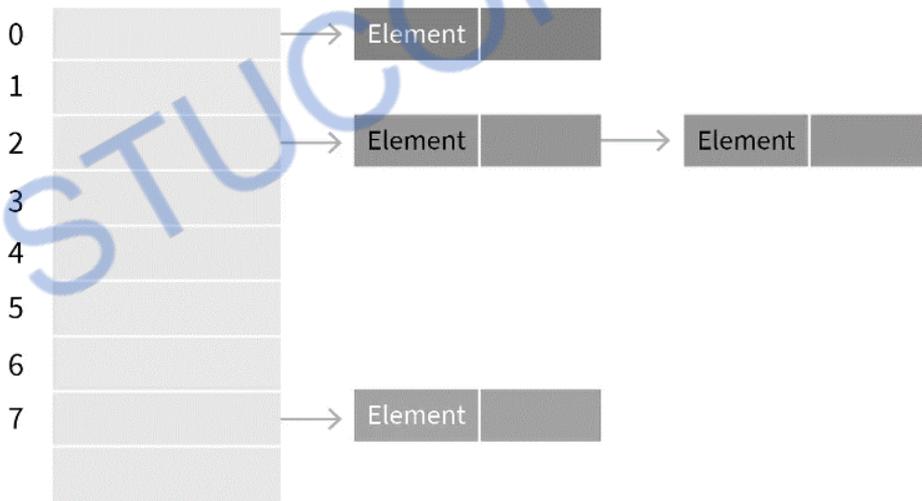
- The process of taking any size of data and then converting that into smaller data value which can be named as hash value. This hash value can be used in an index accessible in hash table. This process defines hashing in data structure.

4.8 SEPARATE CHAINING

- Separate Chaining is the collision resolution technique that is implemented using linked list. When two or more elements are hashed to the same location, these elements are represented into a singly linked list like a chain. Since this method uses extra memory to resolve the collision, therefore, it is also known as open hashing.

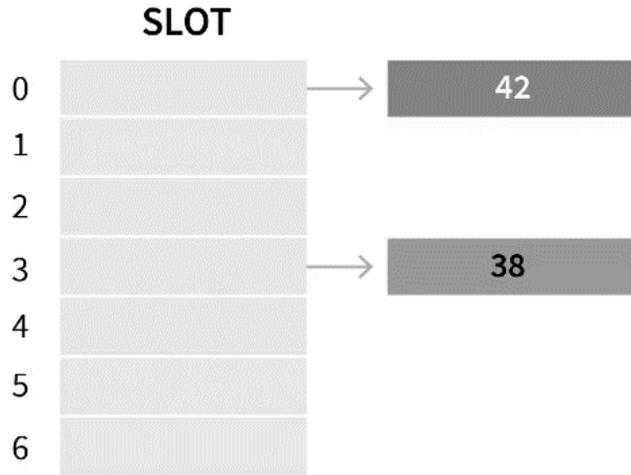
4.8.1 Separate Chaining Hash Table

- In separate chaining, each slot of the hash table is a linked list. We will insert the element into a specific linked list to store it in the hash table. If there is any collision i.e. if more than one element after calculating the hashed value mapped to the same key then we will store those elements in the same linked list. Given below is the representation of the separate chaining hash table.

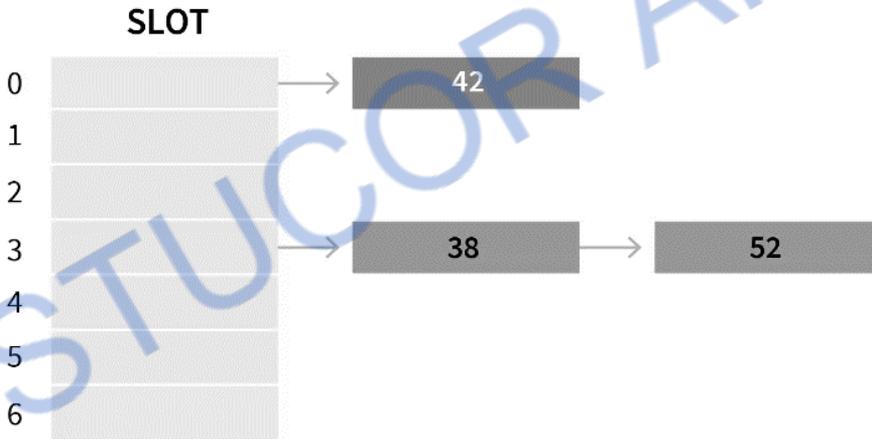


4.8.2 Example for Separate Chaining

- Let's understand with the help of examples. Given below is the hash function:
$$h(\text{key}) = \text{key} \% \text{table size}$$
- In a hash table with size 7, keys 42 and 38 would get 0 and 3 as hash indices respectively.



- If we insert a new element 52, that would also go to the fourth index as $52\%7$ is 3.



- The lookup cost will be scanning all the entries of the selected linked list for the required key. If the keys are uniformly distributed, then the average lookup cost will be an average number of keys per linked list.

4.8.3 How to Avoid Collision in Separate Chaining Method

- Separate chaining method handles the collision by creating a linked list to the occupied buckets. So far we only looked at a simple hash function where collision is imminent.
- It is important to choose a good hash function in order to minimize the number of collisions so that all the key values are evenly distributed in the hash table.

4.34 Non – Linear Data Structures

- Some characteristics of good hash function are:
 - ✓ Minimize collisions
 - ✓ Be easy and quick to compute
 - ✓ key values inserted evenly in the hash table
 - ✓ Have a high load factor for a given set of keys

4.8.4 Practice Problem Based on Separate Chaining

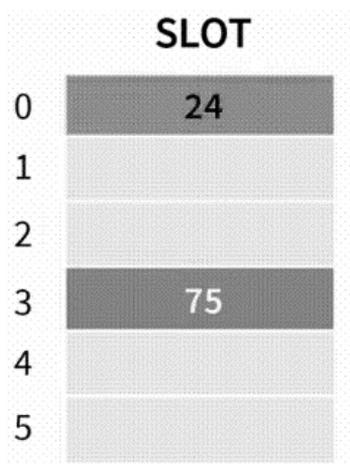
- Let's take an example to understand the concept more clearly. Suppose we have the following hash function, and we have to insert certain elements in the hash table by using separate chaining as the collision resolution technique.
- Hash function = $\text{key} \% 6$ Elements = 24, 75, 65, 81, 42, and 63.
- **Step1:** First we will draw the empty hash table which will have possible range of hash values from 0 to 5 according to the hash function provided.

SLOT	
0	
1	
2	
3	
4	
5	

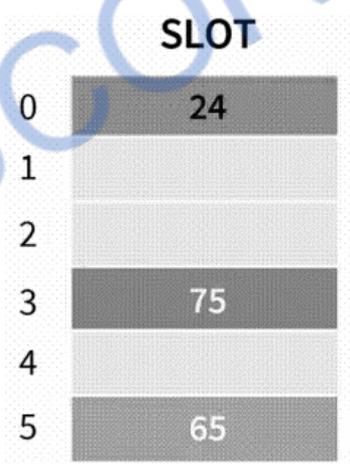
- **Step 2:** Now we will insert all the keys in the hash table one by one. First key to be inserted is 24. It will map to bucket number 0 which is calculated by using hash function $24 \% 6 = 0$.

SLOT	
0	24
1	
2	
3	
4	
5	

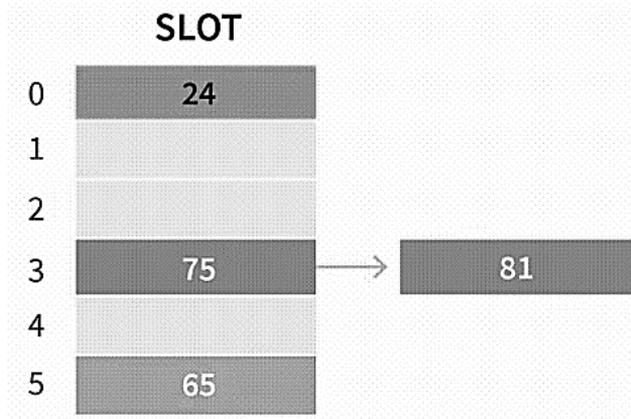
- Step 3: Now the next key that is need to be inserted is 75. It will map to the bucket number 3 because $75\%6=3$. So insert it to bucket number 3.



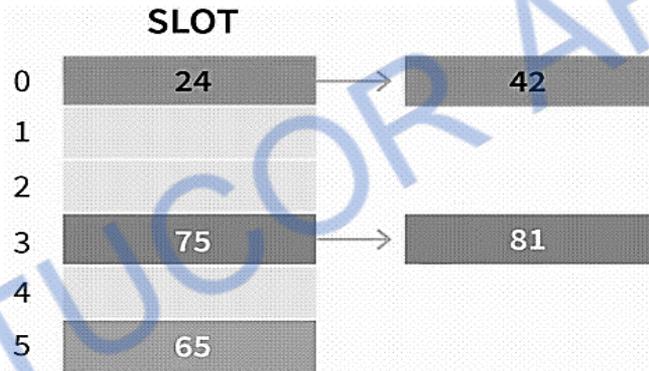
- Step 4: The next key is 65. It will map to bucket number 5 because $65\%6=5$. So, insert it to bucket number 5.



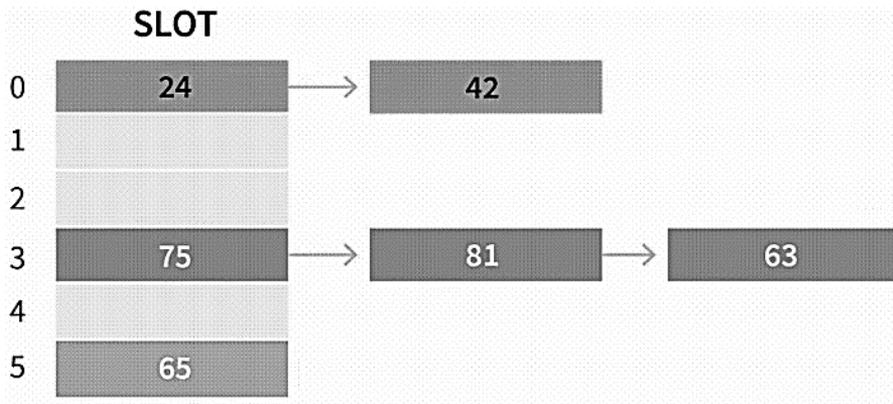
- Step 5: Now the next key is 81. Its bucket number will be $81\%6=3$. But bucket 3 is already occupied by key 75. So separate chaining method will handles the collision by creating a linked list to bucket 3.



- Step 6: Now the next key is 42. Its bucket number will be $42\%6=0$. But bucket 0 is already occupied by key 24. So separate chaining method will again handles the collision by creating a linked list to bucket 0.



- Step 7: Now the last key to be inserted is 63. It will map to the bucket number $63\%6=3$. Since bucket 3 is already occupied, so collision occurs but separate chaining method will handle the collision by creating a linked list to bucket 3.



- In this way the separate chaining method is used as the collision resolution technique.

4.8.5 Advantages and Disadvantages of Separate Chaining

Advantages

- Separate Chaining is one of the simplest methods to implement and understand.
- We can add any number of elements to the chain.
- It is frequently used when we don't know about the number of elements and the number of keys that can be inserted or deleted.

Disadvantages

- The keys in the hash table are not evenly distributed.
- Some amount of wastage of space occurs.
- The complexity of searching becomes $O(n)$ in the worst case when the chain becomes long.

4.9 OPEN ADDRESSING

- The open addressing is another technique for collision resolution. Unlike chaining, it does not insert elements to some other data-structures. It inserts the data into the hash table itself. The size of the hash table should be larger than the number of keys.
- There are three different popular methods for open addressing techniques. These methods are –
 - ✓ Linear Probing
 - ✓ Quadratic Probing
 - ✓ Double Hashing

4.10 LINEAR PROBING

- This is a simple method, sequentially tries the new location until an empty location is found in the table.
- For example: inserting the keys {79, 28, 39, 68, 89} into closed hash table by using same function and collision resolution technique as mentioned before and the table size is 10 (for easy understanding we are not using prime number for table size).Here array or hash table is considered circular because when the last

4.38 Non – Linear Data Structures

slot reached an empty location not found then the search proceeds to the first location of the array.

- The hash function is $h_i(X)=(Hash(X)+F(i)) \% TableSize$ for $i = 0, 1, 2, 3,...etc.$

4.10.1 Solution

0	39
1	68
2	89
3	
4	
5	
6	
7	
8	28
9	79

A Closed Hash Table using Linear Probing

Key	Hash Function $h(X)$	Index	Collision	Alt Index
79	$h_0(79) = (Hash(79) + F(0)) \% 10$ $= ((79 \% 10) + 0) \% 10 = 9$	9		
28	$h_0(28) = (Hash(28) + F(0)) \% 10$ $= ((28 \% 10) + 0) \% 10 = 8$	8		
39	$h_0(39) = (Hash(39) + F(0)) \% 10$ $= ((39 \% 10) + 0) \% 10 = 9$	9	First collision occurs	
	$h_1(39) = (Hash(39) + F(1)) \% 10$ $= ((39 \% 10) + 1) \% 10 = 0$	0		0
68	$h_0(68) = (Hash(68) + F(0)) \% 10$ $= ((68 \% 10) + 0) \% 10 = 8$	8	first collision occurs	
	$h_1(68) = (Hash(68) + F(1)) \% 10$ $= ((68 \% 10) + 1) \% 10 = 9$	9	Again collision occurs	

	$h_2(68)$ $= (\text{Hash}(68)+F(2))\% 10$ $= ((68\% 10)+2)\% 10 = 0$	0	Again collision occurs	
	$h_3(68)$ $= (\text{Hash}(68)+F(3))\% 10$ $= ((68\% 10)+3)\% 10 = 1$	1		1
89	$h_0(89)$ $= (\text{Hash}(89)+F(0))\% 10$ $= ((89\% 10)+0)\% 10 = 9$	9	collision occurs	
	$h_1(89)$ $= (\text{Hash}(89)+F(1))\% 10$ $= ((89\% 10)+1)\% 10 = 0$	0	Again collision occurs	
	$h_2(89)$ $= (\text{Hash}(89)+F(2))\% 10$ $= ((89\% 10)+2)\% 10 = 1$	1	Again collision occurs	
	$h_3(89)$ $= (\text{Hash}(89)+F(3))\% 10$ $= ((89\% 10)+3)\% 10 = 2$	2		2

4.11 QUADTRATIC PROBING

- Quadratic probing is an open addressing method for resolving collision in the hash table. This method is used to eliminate the primary clustering problem of linear probing.
- This technique works by considering of original hash index and adding successive value of an arbitrary quadratic polynomial until the empty location is found. In linear probing, we would use $H+0, H+1, H+2, H+3, \dots, H+K$ hash function sequence.
- Instead of using this sequence, the quadratic probing would use another sequence is that $H+1^2, H+2^2, H+3^2, \dots, H+K^2$. Therefore, the hash function for quadratic probing is

4.40 Non – Linear Data Structures

$$h_i(X) = (\text{Hash}(X) + F(i)^2) \% \text{TableSize for } i = 0, 1, 2, 3, \dots \text{etc.}$$

➤ Let us examine the linear probing with the same example

0	39
1	
2	68
3	89
4	
5	
6	
7	
8	28
9	79

Key	Hash Function h(X)	Index	Collision	Alt Index
79	$h_0(79)$ $= (\text{Hash}(79) + F(0)^2) \% 10$ $= ((79 \% 10) + 0) \% 10$	9		
28	$h_0(28)$ $= (\text{Hash}(28) + F(0)^2) \% 10$ $= ((28 \% 10) + 0) \% 10$	8		
39	$h_0(39)$ $= (\text{Hash}(39) + F(0)^2) \% 10$ $= ((39 \% 10) + 0) \% 10$	9	The first collision occurs	
	$h_1(39)$ $= (\text{Hash}(39) + F(1)^2) \% 10$ $= ((39 \% 10) + 1) \% 10$	0		0
68	$h_0(68)$ $= (\text{Hash}(68) + F(0)^2) \% 10$ $= ((68 \% 10) + 0) \% 10$	8	The collision occurs	

	$h_1(68)$ $= (\text{Hash}(68) + F(1)^2) \% 10$ $= ((68 \% 10) + 1) \% 10$	9	Again collision occurs	
	$h_2(68)$ $= (\text{Hash}(68) + F(2)^2) \% 10$ $= ((68 \% 10) + 4) \% 10$	2		2
89	$h_0(89)$ $= (\text{Hash}(89) + F(0)^2) \% 10$ $= ((89 \% 10) + 0) \% 10$	9	The collision occurs	
	$h_1(89)$ $= (\text{Hash}(89) + F(1)^2) \% 10$ $= ((89 \% 10) + 1) \% 10$	0	Again collision occurs	
	$h_2(89)$ $= (\text{Hash}(89) + F(2)^2) \% 10$ $= ((89 \% 10) + 4) \% 10$	3		3

- Although, the quadratic probing eliminates the primary clustering, it still has the problem.
- When two keys hash to the same location, they will probe to the same alternative location. This may cause secondary clustering. In order to avoid this secondary clustering, double hashing method is created where we use extra multiplications and divisions

4.12 DOUBLE HASHING

- Double Hashing uses 2 hash functions and hence called double hashing. The first hash function determines the initial location to locate the key and the second hash function is to determine the size of the jumps in the probe sequence.

4.12.1 Double Hashing - Hash Function 1 or First Hash Function – formula

- $h_i = (\text{Hash}(X) + F(i)) \% \text{Table Size}$
 were
 - $F(i) = i * \text{hash2}(X)$

4.42 Non – Linear Data Structures

- X is the Key or the Number for which the hashing is done
 - i is the ith time that hashing is done for the same value. Hashing is repeated only when collision occurs
 - Table size is the size of the table in which hashing is done
- This F(i) will generate the sequence such as $\text{hash}_2(X)$, $2 * \text{hash}_2(X)$ and so on.

4.12.2 Double Hashing - Hash Function 2 or Second Hash Function – formula

- Second hash function is used to resolve collision in hashing We use second hash function as
- $\text{hash}_2(X) = R - (X \bmod R)$
- where
- R is the prime number which is slightly smaller than the Table Size.
 - X is the Key or the Number for which the hashing is done

4.12.3 Double Hashing Example - Closed Hash Table

- Let us consider the same example in which we choose $R = 7$.

0	68
1	
2	39
3	89
4	
5	
6	
7	
8	28
9	79

A Closed Hash Table using Double Hashing

Key	Hash Function h(X)	Index	Collision	At Index
79	$h_0(79) = (\text{Hash}(79) + F(0)) \% 10$ $= ((79 \% 10) + 0) \% 10 = 9$	9		
28	$h_0(28) = (\text{Hash}(28) + F(0)) \% 10$ $= ((28 \% 10) + 0) \% 10 = 8$	8		
39	$h_0(39) = (\text{Hash}(39) + F(0)) \% 10$ $= ((39 \% 10) + 0) \% 10 = 9$	9	First collision occurs	
	$h_1(39) = (\text{Hash}(39) + F(1)) \% 10$ $= ((39 \% 10) + 1(7 - (39 \% 7))) \% 10$ $= (9 + 3) \% 10 = 12 \% 10 = 2$	2		2
68	$h_0(68) = (\text{Hash}(68) + F(0)) \% 10$ $= ((68 \% 10) + 0) \% 10 = 8$	8	collision occurs	
	$h_1(68) = (\text{Hash}(68) + F(1)) \% 10$ $= ((68 \% 10) + 1(7 - (68 \% 7))) \% 10$ $= (8 + 2) \% 10 = 10 \% 10 = 0$	0		0
89	$h_0(89) = (\text{Hash}(89) + F(0)) \% 10$ $= ((89 \% 10) + 0) \% 10 = 9$	9	Collision occurs	
	$h_1(89) = (\text{Hash}(89) + F(1)) \% 10$ $= ((89 \% 10) + 1(7 - (89 \% 7))) \% 10$ $= (9 + 2) \% 10 = 10 \% 10 = 0$	0	Again collision occurs	
	$h_2(89) = (\text{Hash}(89) + F(2)) \% 10$ $= ((89 \% 10) + 2(7 - (89 \% 7))) \% 10$ $= (9 + 4) \% 10 = 13 \% 10 = 3$	3		3

- The problem with linear probing is primary clustering. This means that even if the table is empty, any key that hashes to table requires several attempt to resolve the collision because it has to cross over the blocks of occupied cell.
- These blocks of occupied cell form the primary clustering. If any key falls into clustering, then we cannot predict the number of attempts needed to resolve the collision. These long paths affect the performance of the hash table.

4.13 RE-HASHING

- Rehashing is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when the threshold is reached/crossed.

4.13.1 Why Rehashing is done?

- Rehashing is done because whenever a new key value pair is inserted into map, the load factor increases and due to which complexity also increases. And if complexity increases our HashMap will not have constant $O(1)$ time complexity.
- Hence rehashing is done to distribute the items across the hashmap as to reduce both load factor and complexity, So that `get()` and `put()` have constant time complexity of $O(1)$.
- After rehashing is done existing items may fall in the same bucket or different bucket.

4.13.2 What is Load factor in HashMap?

- Load factor in HashMap is basically a measure that decides when exactly to increase the size of the HashMap to maintain the same time complexity of $O(1)$.
- Load factor is defined as (m/n) where n is the total size of the hash table and m is the preferred number of entries which can be inserted before an increment in the size of the underlying data structure is required.
- If you are going to store really large no of elements in the hashmap then it is always good to create HashMap with sufficient capacity upfront as rehashing will not be done frequently, this is more efficient than letting it to perform automatic rehashing.

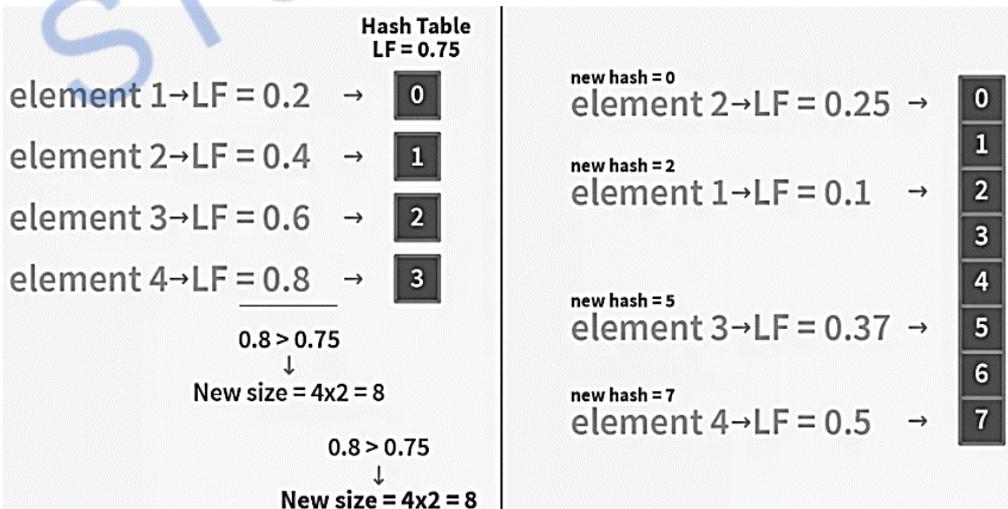
4.13.3 How Rehashing is Done?

- Let's try to understand the above with an example: Say we had HashTable with Initial Capacity of 4. We need to insert 4 Keys: 100, 101, 102, 103
- Hash function used was division method: $\text{Key} \% \text{ArraySize}$
- Element1: $\text{Hash}(100) = 100 \% 6 = 4$, so Element1 will be rehashed and will be stored at 5th Index in this newly resized HashTable, instead of 1st Index as on previous HashTable.

- Element2: $\text{Hash}(101) = 101\%6 = 5$, so Element2 will be rehashed and will be stored at 6th Index in this newly resized HashTable, instead of 2nd Index as on previous HashTable.
- Element3: $\text{Hash}(102) = 102\%6 = 6$, so Element3 will be rehashed and will be stored at 4th Index in this newly resized HashTable, instead of 3rd Index as on previous HashTable.
- Since the Load Balance now is $3/6 = 0.5$, we can still insert the 4th element now.
- Element4: $\text{Hash}(103) = 103\%6 = 1$, so Element4 will be stored at 1st Index in this newly resized HashTable.

4.13.4 Rehashing Steps

- For each addition of a new entry to the map, check the current load factor.
- If it's greater than its pre-defined value, then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucket array.
- Then traverse to each element in the old bucketArray and insert them back so as to insert it into the new larger bucket array.
- If you are going to store a really large number of elements in the HashTable then it is always good to create a HashTable with sufficient capacity upfront as this is more efficient than letting it perform automatic rehashing.



REVIEW QUESTIONS

PART A

1. Define binary tree?

- A binary tree is a tree data structure composed of nodes, each of which has utmost, two children, referred to as left and right nodes. The tree starts off with a single node known as the root.

2. What are the two methods of binary tree implementation?

- Linear representation.
- Linked representation

3. What are the applications of binary tree?

- Binary tree is used in data processing.
 - File index schemes
 - Hierarchical database management system

4. List out few of the Application of tree data-structure?

- The manipulation of Arithmetic expression
- Used for Searching Operation
- Used to implement the file system of several popular operating systems
- Symbol Table construction
- Syntax analysis

5. Define expression tree?

- Expression tree is also a binary tree in which the leaf's terminal nodes or operands and non-terminal intermediate nodes are operators used for traversal.

6. Define tree– traversal and mention the type of traversals?

- Three types of tree traversal
 - Inorder traversal
 - Preoder traversal
 - Postorder traversal.

7. Define in -order traversal?

- In-order traversal entails the following steps;

- Traverse the left subtree
- Visit the root node
- Traverse the right subtree

8. What is pre-order traversal?

- In preorder traversal, first, root node is visited, then left sub-tree and after that right sub-tree is visited. The process of preorder traversal can be represented as:
root → left → right

9. Define threaded binary tree.

- A binary tree is threaded by making all right child pointers that would normally be null point to the in order successor of the node, and all left child pointers that would normally be null point to the in-order predecessor of the node.

10. What are the types of threaded binary tree?

- Right-in threaded binary tree
- Left-in threaded binary tree
- Fully-in threaded binary tree

11. Define Binary Search Tree.

- Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X and the values of all the keys in its right subtree are larger than the key value in X.

12. What is AVL Tree?

- AVL stands for Adelson-Velskii and Landis. An AVL tree is a binary search tree which has the following properties:
 - The sub-trees of every node differ in height by at most one.
 - Every sub-tree is an AVL tree.

13. List out the steps involved in deleting a node from a binary search tree.

- Deleting a node is a leaf node (ie) No children
- Deleting a node with one child.
- Deleting a node with two Childs.

14. Define complete binary tree.

- If all its levels, possible except the last, have maximum number of nodes and if all the nodes in the last level appear as far left as possible

15. Write short notes on Expression Trees.

- A binary expression tree is a specific kind of a binary tree used to represent expressions.
- Two common types of expressions that a binary expression tree can represent are algebraic and boolean. These trees can represent expressions that contain both unary and binary operators.

16. What is Hashing?

- Hashing is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

17. What is Hash Function?

- A hash function is a function that takes a set of inputs of any arbitrary size and fits them into a table or other data structure that contains fixed-size elements.

18. List the advantages of hashing in data structure

- Hash provides better synchronization than other data structures. Hash tables are more efficient than search trees or other data structures. Hash provides constant time for searching, insertion and deletion operations on average.

19. What is separate chaining?

- Separate Chaining is one of the techniques that is used to resolve the collision. It is implemented using linked lists.
- This method combines a linked list with a hash table in order to resolve the collision. In this method, we put all the elements that hash to the same slot in the linked list.

20. What is open addressing in hashing?

- In open addressing,
 - Unlike separate chaining, all the keys are stored inside the hash table.
 - No key is stored outside the hash table.

21. List the techniques used in open addressing.

- Linear Probing
- Quadratic Probing

- Double Hashing

22. What is Linear Probing?

- Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key.

23. Write short notes on Quadratic Probing?

- Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables.
- Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

24. Explain about Double Hashing.

- Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

25. What is rehashing in data structure?

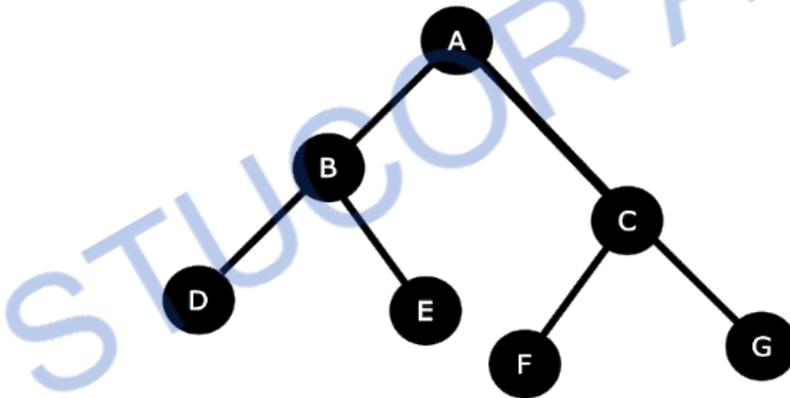
- Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table.

26. List the advantages of Double hashing

- The advantage of Double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of effective method for resolving collisions.

PART-B

1. Explain the tree traversal techniques with an example.
2. Construct an expression tree for the expression $(a+b*c) + ((d*e+f)*g)$. Give the outputs when you apply inorder, preorder and postorder traversals.
3. How to insert and delete an element into a binary search tree and write down the code for the insertion routine with an example.
4. Create a binary search tree for the following numbers start from an empty binary search tree. 45,26,10,60,70,30,40 Delete keys 10, 60 and 45 one after the other and show the trees at each stage.
5. Explain Open Addressing techniques in detail.
6. Find the In-order, Pre-order, and Post-order for the given tree below



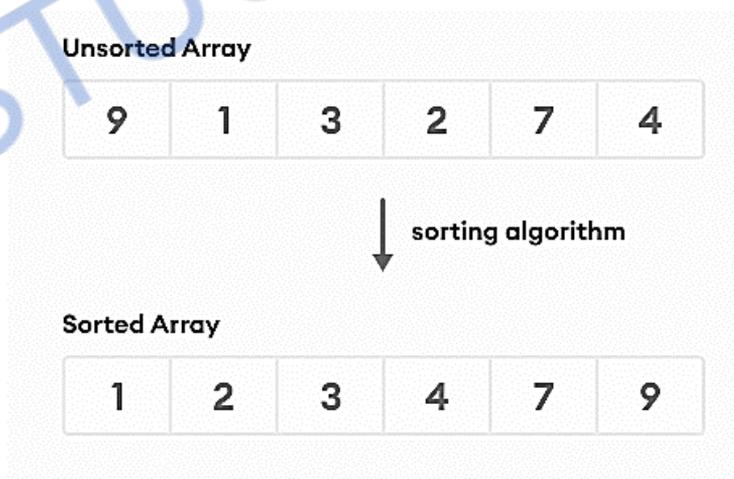
SORTING AND SEARCHING TECHNIQUES

Insertion Sort – Quick Sort – Heap Sort – Merge Sort – Linear Search – Binary Search

5.1 INTRODUCTION TO SORTING

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human – read able input

For example,



5.2 Sorting and Searching Techniques

There are many techniques by using which, sorting can be performed

Sl. No.	Sorting Algorithms	Description
1	Insertion Sort	Insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge
2	Quick Sort	Quick sort follows the divide and conquer approach in which the algorithm is breaking down into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.
3	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements depending upon the choice and the elements are sorted by deleting the root element of the heap.
4	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array

5.2. INSERTION SORT

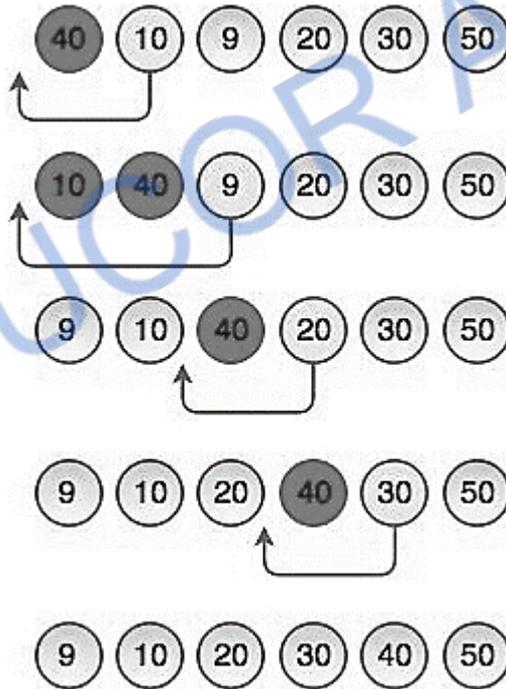
- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

5.2.1 Algorithm:

- Step 1 – If the element is the first one, it is already sorted.
- Step 2 – Move to next element
- Step 3 – Compare the current element with all elements in the sorted array
- Step 4 – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards right
- Step 5 – Insert the value at the correct position
- Step 6 – Repeat until the complete list is sorted

5.2.2 Working of Insertion sort Algorithm

Consider an unsorted array of elements 40, 10, 9, 20, 30, 50



- The above steps represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.

5.4 Sorting and Searching Techniques

- At the first step, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

5.2.3 Analysis of Insertion Sort:

Time Complexity	
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	
	$O(1)$
Stability	
	Yes

5.2.4 Applications

- The insertion sort is used when:
 - The array is has a small number of elements
 - There are only a few elements left to be sorted

Example Program 5.1

```
#include <stdio.h>
int main()
{
    int n, array[1000], c, d, t;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
    {
        scanf("%d", &array[c]);
    }
    for (c = 1 ; c <= n - 1; c++)
    {
```

```
d = c;
while ( d > 0 && array[d] < array[d-1])
{
    t = array[d];
    array[d] = array[d-1];
    array[d-1] = t;
    d--;
}
}
printf("Sorted list in ascending order:\n");
for (c = 0; c <= n - 1; c++)
{
    printf("%d\n", array[c]);
}
return 0;
}
```

Output

Enter the number of elements

5

Enter 5 integers

40

30

20

10

40

Sorted list in ascending order

10

20

30

40

40

5.3 QUICK SORT

- Quick sort is also known as Partition-exchange sort based on the rule of Divide and Conquer.
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only $O(n \log n)$ space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

5.3.1 Algorithm for Quick Sort:

Step 1: Choose the highest index value as pivot.

Step 2: Take two variables to point left and right of the list excluding pivot.

Step 3: Left points to the low index.

Step 4: Right points to the high index.

Step 5: While value at left < (Less than) pivot move right.

Step 6: While value at right > (Greater than) pivot move left.

Step 7: If both Step 5 and Step 6 does not match, swap left and right.

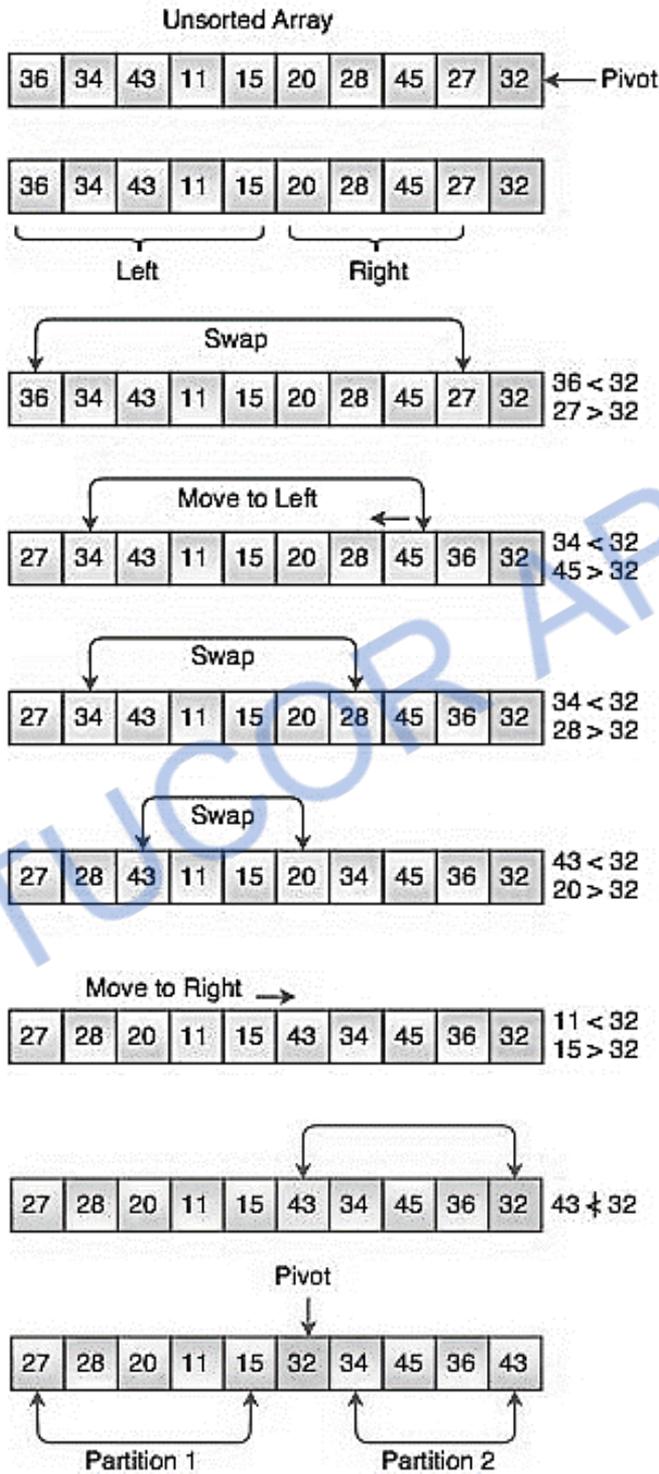
Step 8: If left = (Less than or Equal to) right, the point where they met is new pivot.

5.3.2 Working of Quick sort Algorithm

Consider an unsorted array as follows

36, 34, 43, 11, 15, 20, 28, 45, 27, 32

- The following steps represents how to find the pivot value in an array. As we see, pivot value divides the list into two parts (partitions) and then each part is processed for quick sort. Quick sort is a recursive function. We can call the partition function again.



5.3.3 Quicksort Complexity

Time Complexity	
Best	$O(n \cdot \log n)$
Worst	$O(n^2)$
Average	$O(n \cdot \log n)$
Space Complexity	$O(\log n)$
Stability	No

5.3.4 Applications of quick sort:

Quicksort algorithm is used when

- the programming language is good for recursion
- time complexity matters
- space complexity matters

Example Program 5.2: Program for implementing Quick Sort

```
#include<stdio.h>
#include<conio.h>
//quick Sort function to Sort Integer array list
void quicksort(int array[], int firstIndex, int lastIndex)
{
    //declaaring index variables
    int pivotIndex, temp, index1, index2;
    if(firstIndex < lastIndex)
    {
        //assigninh first element index as pivot element
        pivotIndex = firstIndex;
        index1 = firstIndex;
        index2 = lastIndex;
        //Sorting in Ascending order with quick sort
```

```
while(index1 < index2)
{
    while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
    {
        index1++;
    }
    while(array[index2]>array[pivotIndex])
    {
        index2--;
    }
    if(index1<index2)
    {
        //Swapping operation
        temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;
    }
}
//At the end of first iteration, swap pivot element with index2 element
temp = array[pivotIndex];
array[pivotIndex] = array[index2];
array[index2] = temp;
//Recursive call for quick sort, with partitioning
quicksort(array, firstIndex, index2-1);
quicksort(array, index2+1, lastIndex);
}
}
int main()
{
```

5.10 Sorting and Searching Techniques

```
//Declaring variables
int array[100],n,i;
//Number of elements in array form user input
printf("Enter the number of element you want to Sort : ");
scanf("%d",&n);
//code to ask to enter elements from user equal to n
printf("Enter Elements in the list : ");
for(i = 0; i < n; i++)
{
    scanf("%d",&array[i]);
}
//calling quickSort function defined above
quicksort(array,0,n-1);
//print sorted array
printf("Sorted elements: ");
for(i=0;i<n;i++)
    printf(" %d",array[i]);
getch();
return 0;
}
```

Output

Enter the number of element you want to sort: 5

Enter the elements in the list:

7

10

3

21

15

Sorted elements: 3 7 10 15 21

5.4 HEAP SORT

- Heap sort is a comparison based sorting algorithm.
- It is a special tree-based data structure.
- Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array
- Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array
- Heap sort basically recursively performs two main operations
 - Build a heap H, using the elements of array.
 - Repeatedly delete the root element of the heap formed in 1st phase.

5.4.1 What is a heap?

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

5.4.2 Working of Heap sort Algorithm

- In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows:
- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

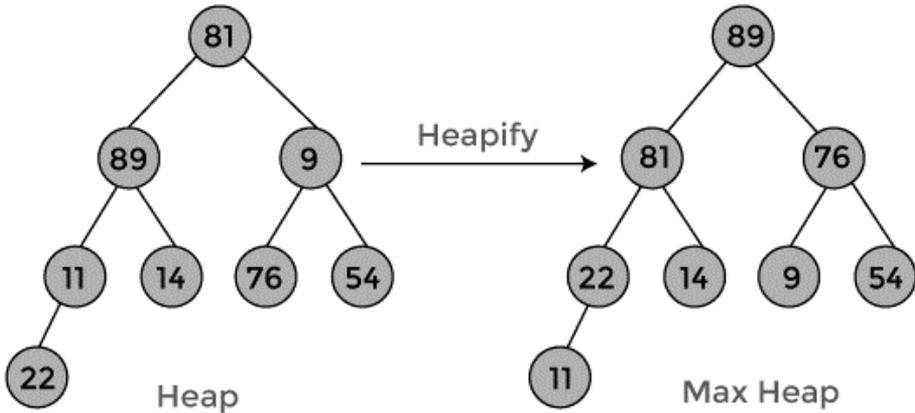
Consider an unsorted array as follows

81, 89, 9, 11, 14, 76, 54, 22

Given array is

81	89	9	11	14	76	54	22
----	----	---	----	----	----	----	----

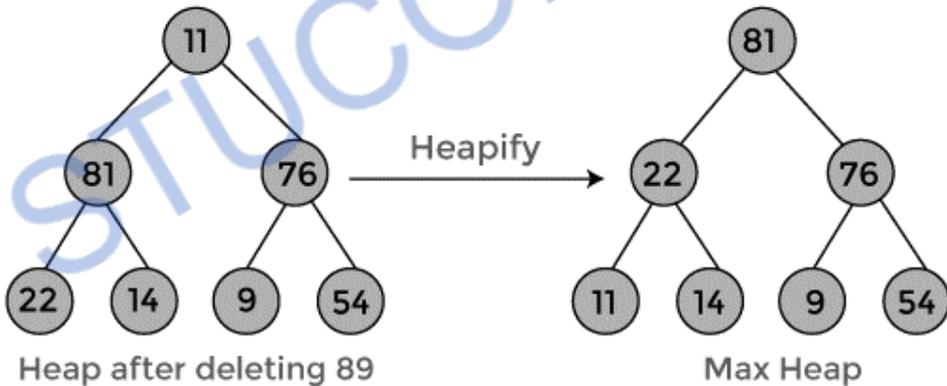
First, construct a heap from the given array and convert it into max heap



After converting the given heap into max heap, the array elements are

89	81	76	22	14	9	54	11
----	----	----	----	----	---	----	----

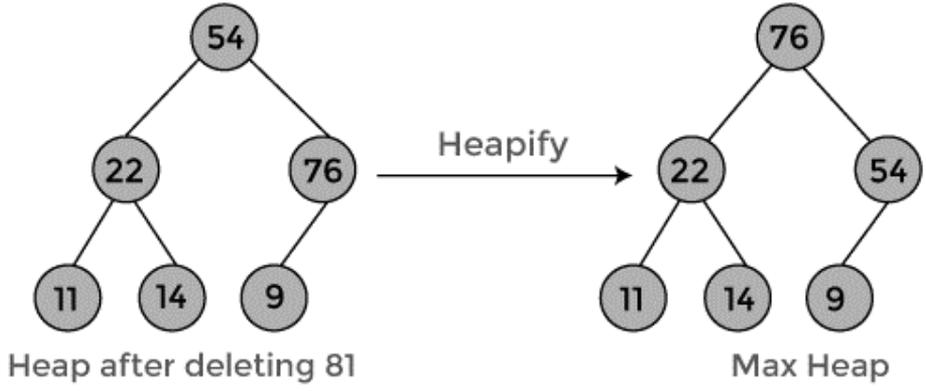
- Next step is to delete the root element (89) from the max heap. To delete this node, swap it with the last node, i.e. (11). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are

81	22	76	11	14	9	54	89
----	----	----	----	----	---	----	----

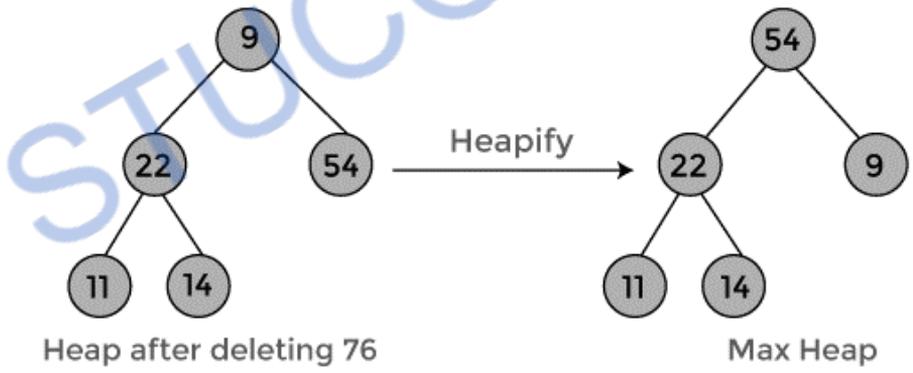
- In the next step, again delete the root element (81) from the max heap. To delete this node, swap it with the last node, i.e. (54). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are

76	22	54	11	14	9	81	89
----	----	----	----	----	---	----	----

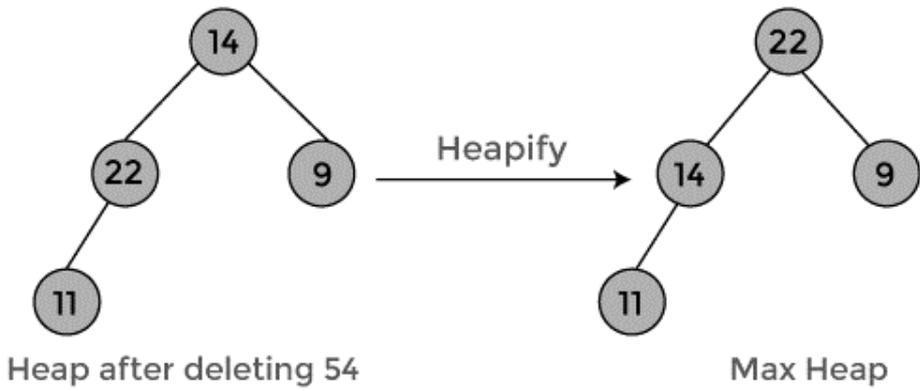
- In the next step, delete the root element (76) from the max heap again. To delete this node, swap it with the last node, i.e. (9). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are

54	22	9	11	14	76	81	89
----	----	---	----	----	----	----	----

- In the next step, again delete the root element (54) from the max heap. To delete this node, swap it with the last node, i.e. (14). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are

22	14	9	11	54	76	81	89
----	----	---	----	----	----	----	----

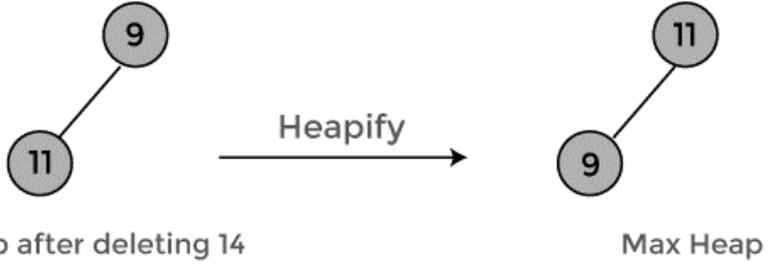
- In the next step, again delete the root element (22) from the max heap. To delete this node, swap it with the last node, i.e. (11). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are

14	11	9	22	54	76	81	89
----	----	---	----	----	----	----	----

- In the next step, again delete the root element (14) from the max heap. To delete this node, swap it with the last node, i.e. (9). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are

11	9	14	22	54	76	81	89
----	---	----	----	----	----	----	----

- In the next step, again delete the root element (11) from the max heap. To delete this node, swap it with the last node, i.e. (9). After deleting the root element, again heapify it to convert it into max heap.



- After swapping the array element 11 with 9, the elements of array are

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

- Now, heap has only one element left. After deleting it, heap will be empty.



- After completion of sorting, the array elements are

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Now, the array is completely sorted

5.4.3 Heapsort Complexity:

Time Complexity	
Best	$O(n \log n)$
Worst	$O(n \log n)$
Average	$O(n \log n)$
Space Complexity	$O(1)$
Stability	No

5.4.4 Heap Sort Applications:

- Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort.
- Because of the $O(n \log n)$ upper bound on Heapsort's running time and constant $O(1)$ upper bound on its auxiliary storage.
- Although Heap Sort has $O(n \log n)$ time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort).

Example Program 5.3: Program for implementing Heap Sort

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
```

```
// If right child is larger than root
if (right < n && a[right] > a[largest])
    largest = right;
// If root is not largest
if (largest != i) {
    // swap a[i] with a[largest]
    int temp = a[i];
    a[i] = a[largest];
    a[largest] = temp;

    heapify(a, n, largest);
}
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
}
/* function to print the array elements */
```

5.18 Sorting and Searching Techniques

```
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }
}

int main()
{
    int a[] = {42, 8, 26, 39, 28, 23, 7};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output

Before sorting array elements are
42, 8, 26, 39, 28, 23, 7
After sorting array elements are
7, 8, 23, 26, 28, 39, 42

5.5 MERGE SORT

- Merge sort is a sorting technique based on divide and conquer technique.
- With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

- Merge sort first divides the array into equal halves and then combines them in a sorted manner

5.5.1 Algorithm

- Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too

Step 1 – if it is only one element in the list it is already sorted, return.

Step 2 – divide the list recursively into two halves until it can no more be divided.

Step 3 – merge the smaller lists into new list in sorted order.

5.5.2 Working of Merge sort Algorithm

Consider an unsorted array elements 12, 31, 25, 8, 32, 17, 40 and 42

Let the elements of array are

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

First divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

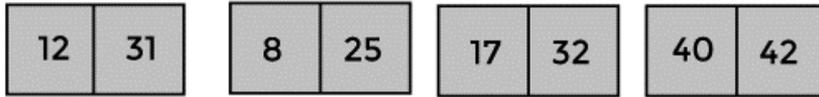
Now, again divide these arrays to get the atomic value that cannot be further divided.

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

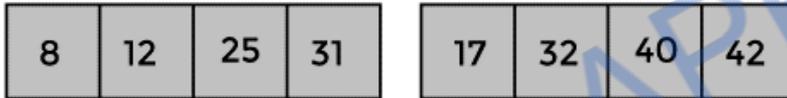
5.20 Sorting and Searching Techniques

Now, combine them in the same manner they were broken. First compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like



5.5.3 Merge sort complexity

Time Complexity	
Best	$O(n \cdot \log n)$
Worst	$O(n \cdot \log n)$
Average	$O(n \cdot \log n)$
Space Complexity	$O(n)$
Stability	YES

5.5.4 Merge Sort Applications

- Inversion count problem
- External sorting
- E-commerce applications

Example Program 5.4: Program for implementing Merge Sort

```
#include <stdio.h>

/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2]; //temporary arrays
    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
        LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
        RightArray[j] = a[mid + 1 + j];
    i = 0; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg; /* initial index of merged sub-array */
    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
    }
}
```

```
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }
    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
/* Function to print the array */
void printArray(int a[], int n)
{
    int i;
```

```
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
int main()
{
    int a[] = { 10, 35, 23, 5, 31, 19, 40, 43 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArray(a, n);
    mergeSort(a, 0, n - 1);
    printf("After sorting array elements are - \n");
    printArray(a, n);
    return 0;
}
```

Output

Before sorting array elements are

10, 35, 23, 5, 31, 19, 40, 43

After sorting array elements are

5, 10, 19, 23, 31, 35, 40, 43

5.6 INTRODUCTION TO SEARCHING

- Searching in data structure refers to the process of finding the required information from a collection of items stored as elements in the computer memory.
- These sets of items are in different forms, such as an array, linked list, graph, or tree. Another way to define searching in the data structures is by locating the desired element of specific characteristics in a collection of items

5.6.1 Searching Methods

- Searching in the data structure can be done by applying searching algorithms to check for or extract an element from any form of stored data structure. These algorithms are classified according to the type of search operation they perform, such as:
- **Sequential search** - The list or array of elements is traversed sequentially while checking every component of the set. For example – Linear Search.
- **Interval Search** - The interval search includes algorithms that are explicitly designed for searching in sorted data structures. In terms of efficiency, these algorithms are far better than linear search algorithms. Example- Logarithmic Search, Binary search.

These methods are evaluated based on the time taken by an algorithm to search an element matching the search item in the data collections and are given by,

- The best possible time
- The average time
- The worst-case time

The primary concerns are with worst-case times, which provide guaranteed predictions of the algorithm's performance and are also easier to calculate than average times.

5.7 LINEAR SEARCH

- Linear search is also called as sequential search algorithm.
- It is the simplest searching algorithm.
- In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted.
- The worst-case time complexity of linear search is $O(n)$.

5.7.1 Steps used in the implementation of Linear Search

- First, we have to traverse the array elements using for loop.
- In each iteration of for loop, compare the search element with the current array element, and
 - If the element matches, then return the index of the corresponding array element.
 - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return -1.

5.7.2 Algorithm

Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i <= n

Step 4: if a[i] == val

 set pos = i

 print pos

 go to step 6

 [end of if]

 set ii = i + 1

 [end of loop]

Step 5: if pos = -1

 print "value is not present in the array "

 [end of if]

Step 6: exit

5.7.3 Working of Linear search

Consider an array of elements 70, 40, 30, 11, 57, 41, 25, 14, 52

Let the elements of array are

5.26 Sorting and Searching Techniques

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is $K = 41$

Now, start from the first element and compare K with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

The value of K , i.e., **41**, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 40$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 11$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 57$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K = 41$

Now, the element to be searched is found. So algorithm will return the index of the element matched.

5.7.4 Linear Search complexity

Time Complexity	
Best	O(1)
Worst	O(n)
Average	O(n)
Space Complexity	
	O(1)

5.7.5 Applications of Linear Search Algorithm

- Linear search can be applied to both single-dimensional and multi-dimensional arrays.
- Linear search is easy to implement and effective when the array contains only a few elements.
- Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

5.7.6 Advantages and Disadvantages

Sl. No.	Advantages	Disadvantages
1.	Will perform fast searches of small to medium lists	Time consuming for the enormous arrays.
2.	The list does not need to sorted	Slow searching of big lists
3.	Not affected by insertions and deletions	A key element doesn't matches any element then Linear search algorithm is a worst case

Example Program 5.5: Program for Implementation of Linear Search

```
#include <stdio.h>

int linearSearch(int a[], int n, int val) {
    // Going through array sequentially
    for (int i = 0; i < n; i++)
```

```
{
    if (a[i] == val)
        return i+1;
}
return -1;
}
int main() {
    int a[] = {59, 40, 33, 11, 57, 41, 27, 18, 53}; // given array
    int val = 41; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = linearSearch(a, n, val); // Store result
    printf("The elements of the array are - ");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\nElement to be searched is - %d", val);
    if (res == -1)
        printf("\nElement is not present in the array");
    else
        printf("\nElement is present at %d position of array", res);
    return 0;
}
```

Output

The elements of the array are - 59, 40, 33, 11, 57, 41, 27, 18, 53

Element to be searched is – 41

Element is present at 6 position of array

5.8 BINARY SEARCH

- Binary search is the search technique that works efficiently on sorted lists.
- Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.
- If the match is found then, the location of the middle element is returned.
- Otherwise, we search into either of the halves depending upon the result produced through the match.

5.8.1 Algorithm

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

 set pos = mid

 print pos

 go to step 6

 else if a[mid] > val

 set end = mid - 1

 else

 set beg = mid + 1

 [end of if]

 [end of loop]

Step 5: if pos = -1

 print "value is not present in the array"

 [end of if]

Step 6: exit

5.8.2 Working of Binary search

- To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

5.30 Sorting and Searching Techniques

- There are two methods to implement the binary search algorithm -
 - Iterative method
 - Recursive method

The recursive method of binary search follows the divide and conquer approach
Consider an array of elements 10, 12, 24, 29, 39, 40, 51, 56, 69

Let the elements of array are

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, **K = 56**

Use the below formula to calculate the mid of the array

$$\text{mid} = (\text{beg} + \text{end})/2$$

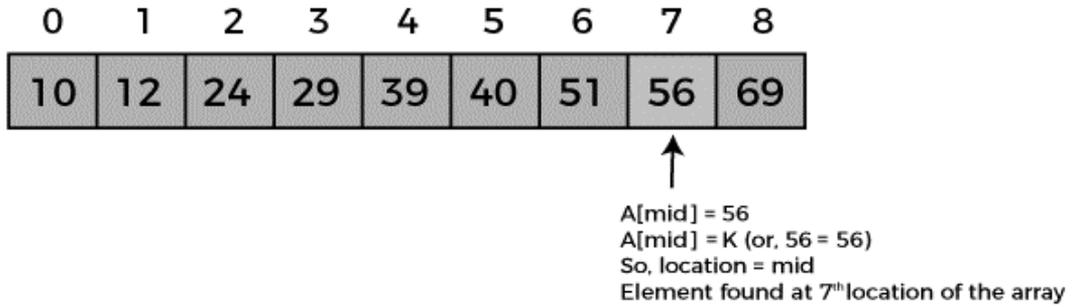
In the given array beg = 0, end = 8. So mid = (0+8)/2 = 4

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
A[mid] = 39
A[mid] < K (or, 39 < 56)
So, beg = mid + 1 = 5, end = 8
Now, mid = (beg + end)/2 = 13/2 = 6

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
A[mid] = 51
A[mid] < K (or, 51 < 56)
So, beg = mid + 1 = 7, end = 8
Now, mid = (beg + end)/2 = 15/2 = 7



Now, the element to search is found. So algorithm will return the index of the element matched.

5.8.3 Binary Search complexity:

Time Complexity	
Best	O(1)
Worst	O(logn)
Average	O(logn)
Space Complexity	
O(1)	

5.8.4 Advantages and Disadvantages

Sl. No.	Advantages	Disadvantages
1.	It is a much faster algorithm	It can be used only when data is sorted
2.	It works on the divide and conquers principle	It is more complicated
3.	It is efficient	If random access is not supported then efficiency might be lost
4.	It is a simple algorithm to understand	It can be implemented only for two-way transversal data structures

Example Program 5.6: Program for implementation of Binary Search

```
#include <stdio.h>

int binarySearch(int a[], int beg, int end, int val)
{
```

5.32 Sorting and Searching Techniques

```
int mid;
if(end >= beg)
{
    mid = (beg + end)/2;
/* if the item to be searched is present at middle */
    if(a[mid] == val)
    {
        return mid+1;
    }
    /* if the item to be searched is smaller than middle, then it can only be in
left subarray*/
    else if(a[mid] < val)
    {
        return binarySearch(a, mid+1, end, val);
    }
    /* if the item to be searched is greater than middle, then it can only be in
right subarray*/
    else
    {
        return binarySearch(a, beg, mid-1, val);
    }
}
return -1;
}

int main() {
    int a[] = {21, 14, 35, 30, 40, 51, 55, 57, 70}; // given array
    int val = 40; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = binarySearch(a, 0, n-1, val); // Store result
    printf("The elements of the array are - ");
```

```

for (int i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\nElement to be searched is - %d", val);
if (res == -1)
printf("\nElement is not present in the array");
else
printf("\nElement is present at %d position of array", res);
return 0;
}
    
```

Output

The elements of the array are - 21, 14, 35, 30, 40, 51, 55, 57, 70
 Element to be searched is – 40
 Element is present at 5 position of array

5.8.5 Linear Search vs Binary Search

Sl. No.	Linear Search	Binary Search
1.	In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
2.	It is also called sequential search.	It is also called half-interval search.
3.	It is preferable for the small-sized data sets.	It is preferable for the large-size data sets.
4.	The time complexity of linear search $O(n)$.	The time complexity of binary search $O(\log n)$.
5.	Multidimensional array can be used.	Only single dimensional array is used.
6.	Linear search performs equality comparisons	Binary search performs ordering comparisons
7.	It is less complex.	It is more complex.
8.	It is very slow process.	It is very fast process

REVIEW QUESTIONS

PART A

1. What is sorting?

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

2. Define insertion sort?

- Successive element in the array to be sorted and inserted into its proper place with respect to the other already sorted element. We start with second element and put it in its correct place, so that the first and second elements of the array are in order.

3. Write short notes on quick sort.

- Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort.

4. What is Time Complexity for Quick Sort?

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

5. What is Merge sort?

- The Merge Sort function repeatedly divides the array into two halves until we reach a stage where we try to perform Merge Sort on a subarray of size 1

6. What is Time Complexity for Merge Sort?

- Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of $O(n \log n)$.

7. What is Linear Search?

- The Linear search algorithm works by sequentially iterating through the whole array or list from one end until the target element is found.
- If the element is found, it returns its index, else -1.

8. What is binary Search?

- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.
- Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

9. What is Heap Sort?

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
- It is like the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

10. What is Time Complexity for Heap Sort?

- The time complexity for Heap sort in average, best-case, and worst-case time complexity of $O(n \log n)$.

11. What is Time Complexity for Insertion Sort?

Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

12. Why quick sort is preferred for arrays and merge sort for linked lists?

- Quick sort is an in-place sorting algorithm, i.e. which means it does not require any additional space, whereas Merge sort does, which can be rather costly. In merge sort, the allocation and deallocation of the excess space increase the execution time of the algorithm.
- Unlike arrays, in linked lists, we can insert elements in the middle in $O(1)$ extra space and $O(1)$ time complexities if we are given a reference/pointer to the previous node. As a result, we can implement the merge operation in the merge sort without using any additional space.

13. In which case insertion sort is used?

- Insertion sort has a fast best-case running time and is a good sorting algorithm to use if the input list is already mostly sorted.

5.36 Sorting and Searching Techniques

14. What is the advantage of using Quick sort algorithm?

- Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

15. Mention the various types of searching techniques in C.

- Linear search
- Binary search

16. Define Searching.

- Searching in data structure refers to the process of finding the required information from a collection of items stored as elements in the computer memory.
- These sets of items are in different forms, such as an array, linked list, graph, or tree.

17. Compare Quick sort and Merge Sort.

Basis for comparison	Quick Sort	Merge Sort
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists

18. Mention the different ways to select a pivot element.

- Pick the first element as pivot
- Pick the last element as pivot
- Pick the Middle element as pivot
- Median-of-three elements
- Pick three elements, and find the median x of these elements
- Use that median as the pivot.
- Randomly pick an element as pivot.

19. What is divide-and-conquer strategy?

- Divide a problem into two or more sub problems
- Solve the sub problems recursively
- Obtain solution to original problem by combining these solutions

PART B

1. Explain Insertion sort with algorithm and examples.
2. Sort the sequence 13,11,74,37,85,39,22,56,25 using insertion sort.
3. Explain the operation and implementation of merge sort.
4. Write quick sort algorithm and explain with an example.
5. Trace the quick sort algorithm for the following list of numbers.
90,77,60,99,55,88,66
6. Explain linear search algorithm with an example.
7. Explain Binary search algorithm with an example.
8. Write down the merge sort algorithm and give its worst case, best case and average case analysis.
9. Explain Heap Sort algorithm with an example