Society for Computer Technology & Research's

Pune Institute of Computer technology

Department of Information Technology

LAB MANUAL

Data Structure & Applications Laboratory

Class:-SE

P: F–LTL–UG /03/R0

SCRT's Pune Institute of Computer Technology, Pune

Department of Information Technology

Lab Manual

Course Code & Name:

3303203: Data Structures and Applications Laboratory (DSAL)

Class: SE Semester: III

Prepared By:

Ms. J. H. Jadhav
Mr. S. D. Shelake
Mrs. K. Y. Digholkar
Dr. A. S. Ghotkar
Mrs. P. S. Shinde
Ms. S. G. Gaikwad

Compiled By:

Ms. J. H. Jadhav

P: F–LTL–UG /03/R0

VISION AND MISSION OF THE INSTITUTE

Vision:

Pune Institute of Computer Technology aspires to be the leader in higher technical education and research of international repute.

Mission:

To be leading and most sought after Institute of education and research in emerging engineering and technology disciplines that attracts, retains and sustains gifted individuals of significant potential.

VISION AND MISSION OF THE DEPARTMENT

Vision:

The department aspires to be a transformative force in Information Technology education and research, developing globally competitive professionals.

Mission:

1. Inculcate an environment of academic rigor in Information Technology education promoting critical thinking, creativity, and problem-solving skills.
2. Foster research and industry collaboration by promoting multiddisciplinary engagement among students and faculty, driving technological advancements to address societal challenges.
3. Cultivate skilled professionals effective communication, leadership, and ethical values.

PEO’s OF THE DEPARTMENT

1. Have a strong background in science and mathematics and an ability to design and develop solutions for complex problems using modern tools in the field of Information Technology.
2. Attain professional competence by industry collaboration, research, and innovation to create solutions that drive technological progress and address societal and environmental concerns through lifelong learning.
3. Have ability to communicate effectively, work well in team, and to manage IT projects in multidisciplinary environment with ethical awareness.

PSO’s OF THE DEPARTMENT

1. Develop computational solutions that integrate emerging fields such as Artificial Intelligence, Cloud Computing, Cybersecurity, and IoT to address the evolving needs of industry and society.
2. Ability to collaborate effectively in teams, leveraging strong interpersonal skills to deliver solutions for complex IT projects using cutting-edge methodologies, development tools, and techniques.

PROGRAM OUTCOMES

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Objectives

The objective of this course is to provide students with

1. To introduce students to fundamental concepts of data, data objects, and structures, including their classifications, abstract data types, and practical applications of linked structures.
2. To familiarize students with the implementation and applications of stacks and queues and provide an understanding of algorithm design and analysis, focusing on efficiency and complexity.
3. To enable students to comprehend, implement, and analyze a variety of sorting and searching algorithms, including their time and space complexities.
4. To provide students with an understanding of hashing techniques, collision resolution strategies, and file organization methods for efficient data storage and retrieval.

Course Outcomes

1. Analyze algorithms and determine algorithm correctness and time efficiency class.
2. Implement abstract data type (ADT) and data structures for given application.
3. Design algorithms based on techniques like brute -force, divide and conquer, greedy, etc.).
4. Solve problems using algorithmic design techniques and data structures.
5. Analyze of algorithms with respect to time and space complexity.

CO-PO-PSO Mapping

CO NO.	Course Outcomes	Program Outcome												PSO	
		1	2	3	4	5	6	7	8	9	10	11	12	1	2
CO1	Implement and Analyse Linked List to solve real world problems of moderate complexity.	2	2	2	2	1	-	-	-	1	-	2	1	-	2
CO2	Select and implement appropriate data structure from stack and queue for real world problem solving like expression solving and backtracking.	2	2	2	2	1	-	-	-	1	-	2	1	-	2
CO3	Analyze the performance of searching and sorting algorithms for specified problem statement.	2	2	2	3	1	-	-	-	1	-	2	1	-	2
CO4	Implement file for efficient data storage and retrieval and apply an appropriate hashing technique to resolve collisions.	2	2	2	2	1	-	-	-	1	-	2	1	-	2
CO5	Implement complete software solution using an appropriate data structure to solve real world problem.	2	2	3	3	1	1	1	2	3	1	3	2	3	2

IMPORTANT GUIDELINES

- The laboratory assignments are to be submitted by students in the form of journals. The Journal consists of a table of contents (Index), and handwritten write-up of each assignment (Title, Aim, Problem Statement, Outcomes, Date of Implementation, Date of Submission, assessor's sign, Theory- Concept, algorithms, printouts of the code written using coding standards, sample test cases etc.)
- Practical Examination will be based on the assignment conducted in this course with multiple different applications.
- The candidate is expected to know the theory involved in the experiment.
- The practical examination should be conducted if journal of the candidate is completed inall respects and certified by the faculty concerned and head of the department.

Guidelines for Lab /TW Assessment

- Examiners will assess the term work based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of handwritten write-up along with results of implemented assignment, attendance etc.
- Examiners will judge the understanding of the practical performed in the examination by asking some questions related to theory & implementation of experiments he/she has carriedout.
- Appropriate knowledge of usage of software and hardware such as compiler, debugger, coding standards, algorithm to be implemented etc. should be checked by the concerned faculty member(s).

TERM WORK RUBRICS

3303203: DATA STRUCTURE AND APPLICATIONS LABORATORY

Teaching Scheme: Practical: 4Hours/Week02 **Credits: 02**

Examination Scheme: CIE (TW): 25Marks
ESE (PR): 50 Marks

Prerequisites: Students should have prior knowledge of

- **Programming Languages:** C and C++
- **Concepts:** Basic Object-Oriented Programming (OOP)

Assig No.	Title and Description of Assignment	Hrs.	CO
	Practice Assignment • Implement a CPP program for Palindrome Checking for number and character. • Implement a CPP program that reads an integer n and stores the first n Fibonacci numbers in an array • Implement a CPP program for Matrix Addition/Subtraction/Multiplication with Menu-Driven Using Functions • Implement a CPP Convert Decimal to Binary/Octal/Hex • Use class for student databases and implement a CPP program to include functions for addition, display and append with Menu Driven Functionality. • Implement a CPP program for employee databases using constructor, destructor. • Implement a CPP program for employee databases using constructor, destructor.	06	
1	**Linked List** **"Efficient Data Management Using Linked Lists: Implementing Dynamic Operations for Contact Management System"** Utilize **Singly and Doubly Linked Lists** to manage a **Contact Management System**. The system will support key operations such as: • **Creating a contact list** dynamically. • **Adding new contacts** efficiently. • **Deleting contacts** when no longer needed. • **Searching for specific contacts** based on name or number. • **Reversing the contact list** for alternate viewing orders. • **Traversing through the list** to display all stored contacts.	06	CO1

2	**Stack and Queue** **"Implementing Queues and Stacks Using Linked Lists for Real-World Task Management Systems"** This problem involves designing and implementing **queues and stacks using linked lists**, focusing on their practical applications in real-world scenarios such as: • **Task Scheduling System (Queue):** Managing tasks in a first-in, first-out (FIFO) order, such as print job scheduling or process management in an operating system. • **Undo/Redo Functionality (Stack):** Implementing an undo/redo feature in text editors or design software using a last-in, first-out (LIFO) approach.	06	CO1, CO2
3	**Searching and Sorting** Consider a student database of SEIT class (at least 15 records). Database contains different fields of every student like Roll No, Name and SGPA. (array of structure) a) Design a roll call list, arrange list of students according to roll numbers in ascending order (Use Bubble Sort) b) Arrange a list of students alphabetically. (Use Insertion sort) c) Arrange a list of students to find out the first ten toppers from a class. (Use Quick sort) d) Search for students according to SGPA. If more than one student has the same SGPA, then print a list of all students having the same SGPA. e) Search for a particular student according to name using binary search without recursion. (all the student records having the presence of search key should be displayed) (Note: Implement either Bubble sort or Insertion Sort.)	06	CO3
4	**Stack Application** **Conversion and Evaluation of Expressions** • Implement a program to convert an infix expression to prefix and postfix notation. • Evaluate both **prefix** and **postfix** expressions. Use STL for implementation.	08	CO1, CO2
5	**Priority Queue and Double-Ended Queue** **Implementation of a Priority Queue** • Develop a **priority queue** where elements are dequeued based	04	CO1, CO2

	on priority rather than insertion order. • Support operations such as insertion, deletion, and display. **Implementation of a Double-Ended Queue (Deque)** • Implement a **double-ended queue** where insertion and deletion can happen from both ends. Support operations such as **enqueue front, enqueue rear, deque front, dequeue rear**.		
6	**Hashing** **"Designing a Secure User Credential Storage System Using Hashing: Collision Handling with and without Chaining"** **Problem Statement:** Develop a hashing-based system for securely storing user credentials, where usernames act as keys and hashed passwords as values. Implement and demonstrate collision handling using: • Chaining (Array-based collision resolution) • Open Addressing (Linear or Quadratic Probing without chaining)	04	CO4
7	**File Handling** The department maintains student's database. The file contains roll number, name, division and address. Write a program to create a sequential file to store and maintain student data. It should allow the user to add, delete information about students. Display information of a particular student. If the record of the student does not exist an appropriate message is displayed. If a student record is found it should display the student details.	04	CO6
8	**Mini Project** **"Real-World Solution Development Using Fundamental Data Structures and Algorithms"** **Problem Statement:** Design and develop a real-world application that leverages **fundamental concepts of Data structures and Algorithms (DSA)**, including **linear data structures, searching, and sorting techniques** (with a preference for **hashing**). The solution should efficiently handle data storage, retrieval, and processing while optimizing performance.	08	CO5

This is for Student's Journal

Assignment write-up should be in this format

Assignment No:

Title:

Aim:

Objective:

Problem Statement:

Theory / Procedure / Diagrams / Circuits:

Algorithm / Methods / Steps: (if applicable)

Conclusion:

Practice Assignments

Problem Statement: **Without and by using class and Functions**

1. Hello World Program
2. Sum of Two Numbers
3. Find Maximum of Two/Three Numbers
4. Check Even or Odd
5. Simple Calculator (using switch) Using Class or without using class
6. Leap Year Checker
7. Print Multiplication Table
8. Factorial of a Number
9. Fibonacci Series
10. Reverse a Number
11. Prime Number Checker
12. Palindrome Checker (Number or String)
13. Armstrong Number
14. Find Largest Element in an Array
15. Sorting an Array (Bubble, Insertion, Selection)
16. Matrix Addition/Subtraction/Multiplication
17. Linear & Binary Search
18. Count Vowels, Consonants in a String
19. Convert Decimal to Binary/Octal/Hex
20. Simple Menu-Driven Program Using Functions

Problem Statement: Class and Object

1. **Create a Class Student**
   Create a class with attributes: roll_no, name, and marks. Include methods to input and display student details.
2. **Simple Rectangle Class**
   Write a class Rectangle with attributes length and width, and methods to calculate area and perimeter.
3. **Bank Account Class**
   Create a class BankAccount with data members: account number, account holder name, and balance. Add methods to deposit and withdraw money.

4. **Class with Constructor and Destructor**
   Create a class Counter with a constructor to initialize count and a destructor to print a message when an object is destroyed.
5. **Employee Salary Calculation**
   Design a class Employee with data members for ID, name, and basic salary. Calculate gross and net salary using class methods.
6. **Time Class with Member Functions**
   Create a Time class with hours, minutes, and seconds. Write methods to add two time objects.
7. **Complex Number Addition**
   Write a class Complex for complex numbers. Implement methods to read, display, and add two complex numbers.
8. **Student Grading System**
   Implement a class that takes marks of five subjects and calculates total, average, and grade.
9. **Aggregation Example - Library and Book**
   Create two classes Book and Library. A Library contains multiple Book objects. Implement methods to add and display books.
10. **Encapsulation with Getter/Setter**
    Create a class Car with private members and public getter/setter methods to access and modify data.
11. **Library Book Management**
    Create a class `Book` with book ID, title, author, and availability. Add methods to issue and return books.
12. **Student Grading System**
    Create a class `Student` that calculates total marks, percentage, and grade for multiple subjects.
13. **Temperature Conversion**
    Create a class Temperature that has methods to convert Celsius to Fahrenheit and vice versa.

Problem Statement: Constructor and Destructor

1. **Default Constructor Example**
   Create a class Circle with a default constructor that initializes radius to 1 and calculates the area.
2. **Parameterized Constructor**
   Create a class Rectangle with a constructor that takes length and width as parameters and computes area and perimeter.
3. **Constructor Overloading**
   Implement a class Box with multiple constructors: one with no parameters, one with a single dimension (cube), and one with three dimensions.
4. **Constructor with Default Arguments**
   Create a class Account with a constructor that initializes account details, where some arguments have default values.

5. **Print Message from Constructor and Destructor**
   Create a class LifeCycle that prints messages from both its constructor and destructor to observe object lifecycle.

6. **Copy Constructor**
   Create a class Person that has a copy constructor which performs a deep copy of dynamically allocated memory (e.g., copying name as a char*).

7. **Constructor Chaining within Class**
   Create a class Employee that uses constructor initialization list and chains constructors internally.

8. **Destructor for Dynamic Memory Cleanup**
   Write a class Dynamic Array that allocates memory using new in the constructor and deallocates it in the destructor.

9. **Class with Array of Objects Using Constructors**
   Create a class Book and define an array of objects initialized using parameterized constructors.

10. **Object Creation in Loop (Observe Constructor/Destructor Call)**
    Create a class Temp with constructor and destructor printing messages. Create multiple objects inside a loop to observe their calls.

11. **Multiple Constructors with Validation Logic**
    Create a class Temperature with multiple constructors (Celsius/Fahrenheit) and validate input inside the constructors.

Problem Statement: Inheritance

1. **Simple Employee-Manager Inheritance**
   Create a base class Employee and a derived class Manager with additional responsibilities.
2. **Shape to Circle Inheritance**
   Define a base class Shape with color and derived class Circle with radius and area calculation.
3. **Person and Student Inheritance**
   Create a base class Person with name and age. Derive a class Student with roll number and marks.
4. **Vehicle and Car**
   Define a base class Vehicle with brand and speed, and a derived class Car with fuel type and seating capacity.
5. **Account and SavingsAccount**
   Base class Account with balance, derived class SavingsAccount with interest rate and compound interest function.

6. **Multilevel Inheritance: Animal → Mammal → Dog**
   Demonstrate three-level inheritance with relevant properties at each level.
7. **Hierarchical Inheritance: Shape → Circle, Rectangle, Triangle**
   Use a base class `Shape` and multiple derived classes for different shapes.

8. **Educational Institution Example**
   Base class `Institution`, derived classes `School` and `College`, and further derived classes for `EngineeringCollege`, etc.
9. **Employee → Permanent → Manager**
   Use multilevel inheritance to handle employee types with different benefits.
10. **Multiple Inheritance: Printer and Scanner → Copier**
    Combine functionality of two base classes using multiple inheritance.
11. **Hybrid Inheritance Example**
    Combine multiple and hierarchical inheritance to show a diamond problem and solve using virtual base class.

Problem Statement: Friend Functions

1. **Friend Function to Access Private Data**
   Create a class Box with private data members length, breadth, and height. Write a friend function to calculate the volume.
2. **Display Private Member Using Friend Function**
   Define a class Student with a private member marks. Use a friend function to display the marks.
3. **Add Two Distances Using Friend Function**
   Define a class Distance with feet and inches. Use a friend function to add two Distance objects.
4. **Friend Function to Calculate Area of a Circle**
   Create a class Circle with radius as a private member. Write a friend function to compute the area.
5. **Check if Number is Positive Using Friend Function**
   Create a class Number with a private member. Write a friend function to check if it's positive.
6. **Add Two Complex Numbers Using Friend Function**
   Define a class `Complex` with real and imaginary parts. Use a friend function to add two `Complex` objects.
7. **Compare Two Private Members Using Friend Function**
   Create a class `Employee` with private `salary`. Use a friend function to compare salaries of two employees.
8. **Friend Function to Find Greater Value**
   Create a class `Value` with a private member `val`. Write a friend function to find the greater of two `Value` objects.
9. **Bank Transaction Between Two Accounts Using Friend Function**
   Design a class BankAccount. Use a friend function to transfer money from one account to another.

Assignment No: 1

Title: Linked List

Aim: Use of Singly and Doubly Linked List for implementation of Contact Management System

Objective:

1. To understand and implement the concepts of Singly and Doubly Linked Lists for efficient data storage and manipulation.
2. To develop a functional Contact Management System that allows operations such as adding, deleting, searching, and displaying contact details using linked list data structures.
3. Comparing the advantages and limitations of Singly and Doubly Linked Lists in terms of traversal, insertion, and deletion within the Contact Management System

CO Mapped: CO1

Problem Statement:

"Efficient Data Management Using Linked Lists: Implementing Dynamic Operations for Contact Management System"

Utilize **Singly and Doubly Linked Lists** to manage a **Contact Management System**. The system will support key operations such as:

- **Creating a contact list** dynamically.
- **Adding new contacts** efficiently.
- **Deleting contacts** when no longer needed.
- **Searching for specific contacts** based on name or number.
- **Reversing the contact list** for alternate viewing orders.
- **Traversing through the list** to display all stored contacts.

Expected Outcome:

1. Clear understanding of the working and structural differences between Singly Linked List and Doubly Linked List.
2. A functional Contact Management System capable of performing operations such as adding, deleting, searching, and displaying contact details using Singly and Doubly Linked Lists.
3. Ability to compare and analyze the performance of Singly and Doubly Linked Lists for contact management system.

Pre-requisites:

1. Basic knowledge of C/C++ Programming
2. Understanding of Data Structures Concepts
3. Knowledge of Pointer Concepts

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- **GeeksforGeeks - Linked Lists Tutorials**
  https://www.geeksforgeeks.org/data-structures/linked-list/
- **Tutorials Point - Data Structure Linked List**
  https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm
- **NPTEL - Data Structures and Algorithms (C Language Based)**
  https://nptel.ac.in/courses/106/102/106102064/

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE (Code::Blocks, Turbo C++, VSCode etc.)

Input: Minimum 5 records

Theory / Procedure / Diagrams:

- Concept of Linked List
- Node Representation in the Linked List
- Concept of Singly and Doubly Linked Lists
- Diagrammatic representation of Singly and Doubly Linked Lists
- Operations on Linked List
- Applications of Linked List

Algorithm / Methods / Steps:

➢ **Insertion in the Linked List**

```
Step 1: Start
Step 2: Create a new node newNode
Step 3: Assign data to newNode.data
Step 4: Set newNode.next = NULL
```

Case 1: Insertion at the Beginning

```
Step 5:
    • If the list is empty (head == NULL)
        o Set head = newNode
    • Else
        o Set newNode.next = head
        o Set head = newNode
```

Case 2: Insertion at the End

```
Step 6:
    • If the list is empty (head == NULL)
        o Set head = newNode
    • Else
        o Initialize temp = head
        o While temp.next != NULL
            ▪ Set temp = temp.next
        o Set temp.next = newNode
```

Case 3: Insertion at a Specific Position (position >= 1)

```
Step 7:
    • If position == 1
        o Perform Insertion at Beginning
    • Else
        o Initialize temp = head
        o Loop from i = 1 to position - 2
            ▪ If temp == NULL, print "Position out of range" and exit
            ▪ Set temp = temp.next
        o Set newNode.next = temp.next
        o Set temp.next = newNode
Step 8: Stop
```

- **Deletion in a Linked List**

```
Step 1: Start
Step 2: Check if the list is empty (head == NULL)
    • If YES, print "List is empty" and stop
    • Else, proceed
```

Case 1: Deletion at the Beginning

```
Step 3:
    • Initialize temp = head
    • Set head = head.next
    • Free memory of temp
```

Case 2: Deletion at the End

```
Step 4:
    • If the list has only one node (head.next == NULL)
        o Free head
        o Set head = NULL
    • Else
        o Initialize temp = head
        o While temp.next.next != NULL
            ▪ Set temp = temp.next
        o Free temp.next
        o Set temp.next = NULL
```

Case 3: Deletion at a Specific Position (position >= 1)

```
Step 5:
    • If position == 1
        o Perform Deletion at Beginning
    • Else
        o Initialize temp = head
        o Loop from i = 1 to position - 2
            ▪ If temp == NULL or temp.next == NULL, print "Position out of range" and
              stop
            ▪ Set temp = temp.next
        o Initialize toDelete = temp.next
        o If toDelete == NULL, print "Position out of range" and stop
        o Set temp.next = toDelete.next
        o Free memory of toDelete
Step 6: Stop
```

- **Searching in a Linked List**

```
Step 1: Start
Step 2: Input the element to be searched, say key
Step 3: Initialize a pointer temp = head
Step 4: Initialize a variable position = 1
Step 5: Repeat while temp != NULL
    • If temp.data == key
        o Print "Element found at position", position
        o Stop
    • Else
        o Set temp = temp.next
        o Increment position by 1
Step 6: If loop ends and element not found
    • Print "Element not found in the list"
Step 7: Stop
```

- **Traverse a Linked List**

```
Step 1: Start
Step 2: Check if the list is empty (head == NULL)
    • If YES, print "List is empty" and stop
    • Else, proceed
Step 3: Initialize a pointer temp = head
Step 4: Repeat while temp != NULL
    • Print temp.data
    • Set temp = temp.next
Step 5: Stop
```

Test Cases:

Consider the following scenarios while performing the operations

Addition:

- In the empty list
- In the existing list

Deletion:

- With empty list
- List with single element
- List with multiple elements
    - Element for deletion is present in the list

    - Element for deletion is not present in the list
- Varying position of element to be deleted
    - First place, last place, in between

Search:

- With empty list
- List with single/multiple elements
    - Element Present
    - Element Not Present

Expected Output: (Results / Visualization)

- Appropriate result with clear message about the status of each of the operation.

Inference:

- Linked Lists provide an efficient way to handle dynamic data where frequent insertion, deletion, and traversal operations are required.
- The Singly Linked List offers a simple structure for basic contact management, while the Doubly Linked List provides improved flexibility, allowing both forward and backward traversal, making deletion and updating operations more convenient.

Oral questions:

1. What is the difference between a singly linked list and a doubly linked list?
2. Why are linked lists preferred over arrays in a Contact Management System?
3. How does dynamic memory allocation help while implementing linked lists?
4. What are the advantages of using a doubly linked list in your Contact Management System?
5. How would you insert a new contact at the beginning of a singly linked list?
6. What changes are required to insert a contact at the end of a doubly linked list?
7. What are the steps to delete a specific contact by name from a doubly linked list?
8. How do you handle memory deallocation when deleting a node from the linked list?
9. How can you modify a contact’s information once it is stored in the linked list?
10. What are the time complexities for insertion, deletion, and search operations in singly and doubly linked lists?
11. Which linked list (singly or doubly) would you prefer if frequent deletions are required? Why?
12. How would you handle duplicate contact names in your Contact Management System

Assignment No: 2

Title: Stack and Queue

Aim: Implementing Stack and Queue Using Linked Lists for Real-World Task Management Systems

Objective:

1. To implement a Queue using linked lists for simulating real-world FIFO task scheduling.
2. To develop a Stack using linked lists for modelling LIFO-based undo/redo operations.
3. To analyse the efficiency and suitability of linked lists in dynamic data structure applications.
4. To demonstrate real-world use cases of queues and stacks in task management systems.

CO Mapped: CO1 and CO2

Problem Statement:

"Implementing Queues and Stacks Using Linked Lists for Real-World Task Management Systems"
This problem involves designing and implementing **queues and stacks using linked lists**, focusing on their practical applications in real-world scenarios such as:

- **Task Scheduling System (Queue):** Managing tasks in a first-in, first-out (FIFO) order, such as print job scheduling or process management in an operating system.
- **Undo/Redo Functionality (Stack):** Implementing an undo/redo feature in text editors or design software using a last-in, first-out (LIFO) approach.

Expected Outcome:

1. Efficient execution of task scheduling and undo/redo operations using linked list-based Queue and Stack implementations.
2. Improved understanding of dynamic memory management and pointer manipulation in real-world data structure applications.
3. Demonstration of how linked lists enhance flexibility and scalability in practical task management systems.

Pre-requisites:

1. Understanding stack/Queue
2. Structure in C/C++

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE (Code::Blocks, Turbo C++, VSCode etc.)

Input: Minimum 5 records

Theory / Procedure / Diagrams:

- **Stack Concepts**

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle, where the last element added is the first to be removed.

When a stack is implemented using a linked list, each element (called a node) contains:

- The data (value),
- A pointer to the next node.

The top of the stack is represented by the head of the linked list.

- **Queue Concepts**

Queue operations using a linked list involve dynamically adding and removing elements while maintaining the FIFO (First In, First Out) principle.

Structure of Node:

Each node contains:

- data – value to store
- next – pointer to the next node

Algorithm / Methods / Steps:

Stack operations

PUSH (Insert an element at the top)

```
Procedure PUSH(Stack, value)
    Create newNode
    newNode.data ← value
    newNode.next ← Stack.top
    Stack.top ← newNode
End Procedure
```

POP (Remove and return the top element)

```
Procedure POP(Stack)
    If Stack.top = NULL Then
        Print "Stack Underflow"
        Return
    End If
    temp ← Stack.top
    value ← temp.data
    Stack.top ← Stack.top.next
    Delete temp
    Return value
End Procedure
```

PEEK (Return the top element without removing it)

```
Procedure PEEK(Stack)
    If Stack.top = NULL Then
        Print "Stack is Empty"
        Return
    End If
    Return Stack.top.data
End Procedure
```

IsEmpty (Check if the stack is empty)

```
Function ISEMPTY(Stack)
    Return Stack.top = NULL
End Function
```

Queue Operations:

ENQUEUE Operation (Insert element at rear)

```
Procedure ENQUEUE(Queue, value)
```

```
    newNode ← Allocate memory for new node
    newNode.data ← value
    newNode.next ← NULL
    If Queue.rear = NULL Then
        Queue.front ← newNode
        Queue.rear ← newNode
    Else
        Queue.rear.next ← newNode
        Queue.rear ← newNode
    End If
End Procedure
```

DEQUEUE Operation (Remove element from front)

```
Procedure DEQUEUE(Queue)
    If Queue.front = NULL Then
        Print "Queue Underflow"
        Return
    End If
    temp ← Queue.front
    value ← temp.data
    Queue.front ← Queue.front.next
    If Queue.front = NULL Then
        Queue.rear ← NULL
    End If
    Delete temp
    Return value
End Procedure
```

PEEK Operation (View front element)

```
Function PEEK(Queue)
    If Queue.front = NULL Then
```

```
        Print "Queue is Empty"
        Return
    End If
    Return Queue.front.data
End Function
```

ISEMPTY Operation (Check if queue is empty)

```
Function ISEMPTY(Queue)
    Return Queue.front = NULL
End Function
```

Test Cases:

1. Simple Task Queue (FIFO)
2. Empty Queue Handling
3. Undo Operations (LIFO)
4. Multiple Undo and Empty Stack Check

Expected Output: (Results / Visualization)

Queue (Task Scheduling System - FIFO)

Operations Performed:

- Enqueue tasks: Task1, Task2, Task3
- Dequeue one task
- Peek at the current front task

Expected Output:

```
Enqueued: Task1
Enqueued: Task2
Enqueued: Task3
Dequeued: Task1
Front Task: Task2
```

Stack (Undo/Redo Functionality - LIFO)

Operations Performed:

- Push actions: Edit1, Edit2, Edit3
- Pop one action (Undo)
- Peek at current top action

Expected Output:

```
Pushed: Edit1

Pushed: Edit2

Pushed: Edit3

Popped (Undo): Edit3

Top Action: Edit2
```

Inference:

The implementation showcases the use of linked lists to implement stacks and queues for efficient task management, reinforcing key concepts of dynamic memory and linear data structures.

Oral questions:

1. What is the main difference between a stack and a queue in terms of task order?
2. Why is a linked list preferred over an array in dynamic task handling systems?
3. How does a queue model a print job scheduler?
4. What real-world scenario does the stack in your program simulate?
5. What happens when you dequeue from an empty queue?
6. What type of list traversal is used in stack operations?
7. Explain what 'LIFO' means using your undo/redo system.
8. What would happen if you forget to update the rear pointer after dequeueing the last task?
9. How do you identify an empty stack or queue in your code?
10. Which pointer in the linked list stack points to the most recent action?

Assignment No: 3

Title: Searching and Sorting

Aim: To implement various searching and sorting algorithms on a student database using an array of structures.

Objective:

1. Understand how to use array of structures in C/C++.
2. Implement and apply different sorting algorithms: Bubble Sort, Insertion Sort, and Quick Sort.
3. Implement and apply different searching algorithms: Linear Search and Binary Search.
4. Analyse performance: Passes, swaps, comparisons, and time complexity**.**

CO Mapped: CO3

Problem Statement:

Create a student database of **at least 5 SEIT students**, each having:

- Roll Number (Integer)
- Name (String)
- SGPA (Float)

Perform the following operations on the array of structure:

a**.** Sort students by roll number (ascending order) using Bubble Sort.

b. Sort students alphabetically by name using Insertion Sort.

c. Sort students by SGPA in descending order using Quick Sort to get top 10 toppers.

d. Search for students by SGPA using Linear Search and display all matches.

e. Search for student(s) by name using non-recursive Binary Search, display all partial matches.

Expected Outcome:

1. Understand how to handle complex data using arrays and structures.
2. Implement sorting and searching techniques.
3. Compare algorithm performance (best, worst, average case, number of comparisons, swaps, passes, and memory usage).

Pre-requisites:

1. Understanding of Arrays
2. Structure in C/C++
3. Basic Sorting/Searching Algorithms

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- https://www.geeksforgeeks.org/dsa/sorting-algorithms/

- https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE (Code::Blocks, Turbo C++, VSCode etc.)

Input: Minimum 5 student records

Theory / Procedure / Diagrams:

- Concepts of Structure
- Array of Structure
- Sorting
    - Bubble sort
    - Insertion sort
    - Quick sort
- Searching
    - Linear search
    - Binary search

Algorithm / Methods / Steps:

➢ **Create a structure**

```
Create_database(struct student s[] )
    Step 1: Accept how many records user need to add, say, no of records as n
    Step 2: For i = 0 to n – 1
        i. Accept the record and store it in s[i]
    Step 3: End For
    Step 4: stop
Display_database(struct student s[] ,int n)
    Step 1: For i = 0 to n – 1
        i. Display the fields s.roll_no, s.name, s.sgpa
    Step 2: End For
    Step 3: Stop
```

➢ **Bubble Sort (by Roll No - Ascending)**

```
BubbleSort(Student s[], n)
    Step 1: For Pass = 1 to n-1
    Step 2: For i = 0 to (n – pass – 1)
        i. If s[i].roll_no < s[i+1].roll_no
        ii. Swap (s[i]. s[i+1])
        iii. End if
    Step 3: End for
    Step 4: End For
    Step 5 : Stop
```

➢ **Insertion Sort (by Name - Alphabetical)**

```
insertion_Sort (Struct student S[], int n)

  Step 1: For i = 1 to n-1
          i.  Set key to s[i]
          ii. Set j to i-1
          iii. While j>=0 AND strcmp(s[i].name,key.name)>0
             a. Assign s[j] to s[j+1]
             b. Decrement j
          iv. End While
  Step 2: Assign key to s[j+1]
  Step 3: End for
  Step 4: end of insertion sort
```

➢ **Quick Sort (by SGPA - Descending)**

- Use divide-and-conquer
- Select pivot, partition array, recursively sort

```
partition (struct student s[], int l, int h)
// where s is the array of structure, l is the index of starting element and h is the index of last element.
Step 1: Select s[l].sgpa  as the pivot element
Step 2: Set i = l Step 3: Set j = h-1 Step 4:  While  i ≤ j

    i.  Increment i till s[i].sgpa ≤ pivot element
    ii. Decrement j till s[j].sgpa > pivot element
    iii.    If i < j
    iv. Swap(s[i], s[j])
     v. End if
Step 5: End while

Step 6: Swap(s[j],s[l])

Step 7:  return j

Step 8: end of Partition
```

```
quicksort( struct student s[], int l, int h)

//where s is the array of structure , l is the index of starting element
//and h is the index of last element.
Step 1: If l<h
     i. P=partition(s,l,h)
     ii.  quicksort (s,l,p-1)
     iii. quicksort (s,p+1,h)
Step 2: End if

Step 3: end of quicksort
```

➤ **Searching Algorithms**

Linear Search (by SGPA)

Loop through array, compare s[i].sgpa == key, print all matches.

Linear_search (struct student s[], float key, int n)

```
//Here s is array of structure student, key is sgpa of student to be searched and displayed, n is total
number of students in record
Step 1: Set i to 0 and flag to 0
Step 2: While i<n
        i. If s[i].sgpa==key
                a. Print s[i].roll_no, s[i].name
                b. Set flag to 1
                c. i++
Step 3: End while
Step 4: If flag==0
        i. Print No student found with sgpa=value of key
Step 5: End if
Step 6: End of linear_ sear
```

Binary Search (Non-Recursive - by Name)

Ensure list is sorted by name.
Use iterative binary search to locate any record containing the key string.

Binary_Search (s, n , Key)

```
    // Where s is an array of structure , n is the no of records, and key is
     element to be searched
    Step 1: Set l = 0 & h = n-1
    Step 2:While l ≤ h

            i.   mid = (l + h ) / 2
            ii.  If strcmp (s[mid].name, key)==0)
                    a.  found
                    b. stop
            iii. Else
                a. if (strcmp (key, s[mid].name)<0
                    i. h  = mid – 1
                b. Else
                    ii. l = mid + 1
                c. End if
          iv. End if
  Step 3: End while

  Step 4: not found // search is unsuccessful
```

Test Cases:

Test all algorithms on:

1. Already sorted list
2. Reverse sorted list
3. Partially sorted list
4. Random order list

Expected Output: (Results / Visualization)

1. Test algorithm for above four test cases
2. Analyze the algorithms based on no of comparisons and swapping/shifting required
3. Check for Sort Stability factor
4. No of passes needed
5. Best /average/ worst case of the each algorithm based on above test case
6. Memory space required to sort

Inference:

- Sorting helps in efficient searching and data organization.
- Structures allow handling of heterogeneous student data.
- Performance varies with input; best algorithm depends on context

Oral questions:

- What is searching? And What is the need of sorting
- How to get output after each pass? What do you mean bypass?
- What is recursion? Where the values are stored in recursion?
- Explain the notation Ω, θ, O for time analysis. Explain the time complexity of bubble sort and linear search, binary search.
- What is need of structure? What are the differences between a union and a structure?
- Which operators are used to refer structure member with and without pointer?
- List the advantages and disadvantages of Sequential / Linear Search?
- List the advantages and disadvantages of binary Search?
- What you mean by internal & External Sorting?
- Analyze Quick sort with respect to Time complexity
- In selecting the pivot for Quick Sort, which is the best choice for optimal partitioning:
  a) The first element of the array
  b) The last element of the array
  c) The middle element of the array
  d) The largest element of the array
  e) The median of the array
  f) Any of the above?
- Explain the importance of sorting and searching in computer applications?
- Analyze bubble sort with respect to time complexity.
- What is Divide and Conquer Methodology?
- How the pivot is selected and partition is done in quick sort?
- What is the complexity of quick sort?

Assignment No: 4

Title: Stack: Conversion and Evaluation of Expressions

Aim: To implement a program that converts an infix expression to both prefix and postfix notation and evaluates them using Standard Template Library (STL), demonstrating understanding of expression conversion, stack operations, and expression evaluation.

Objective:

1. To understand the concept and implementation of Stack data structure using SLL.
2. To understand the concept of conversion of expression.
3. To understand the concept of evaluation of expression.

CO Mapped: CO1, CO2

Problem Statement: Implement a program to convert an infix expression to prefix and postfix notation. • Evaluate both prefix and postfix expressions. Use STL for implementation.

Expected Outcome:

1. Understand concept and implementation of stack.
2. Implement Stack as an ADT.
3. Implement applications of Stack i.e. Expression Conversion and Evaluation

Pre-requisites:

1. Understanding of dynamic memory allocation, pointers and STL
2. Concepts of Linked List
3. Working of Stack and stack operations.

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- **https://ds1-iiith.vlabs.ac.in/data-structures-1/**
- **https://ds2-iiith.vlabs.ac.in/data-structures-2/**
- **http://cse01-iiith.vlabs.ac.in/**

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE (Code::Blocks,Turbo C++, VSCode etc.)

Input: Expression in the form of String.

Theory / Procedure / Diagrams:

Expression conversion and stack

Need for expression conversion

One of the disadvantages of the infix notation is that we need to use parentheses to control the evaluation of the operators. We thus have an evaluation method that includes parentheses and two operator priority classes. In the postfix and prefix notations, we do not need parentheses; each provides only one evaluation rule.

Although some high-level languages use infix notation, such expressions cannot be directly evaluated. Rather, they must be analyzed to determine the order in which the expressions are to be evaluated.

What is Polish Notation?

Conventionally, we use the operator symbol between its two operands in an arithmetic expression.

A+B   C–D*E   A*(B+C)

We can use parentheses to change the precedence of the operators. Operator precedence is pre-defined. This notation is called INFIX notation. Parentheses can change the precedence of evaluation. Multiple passes required for evaluation. Named after Polish mathematician Jan Named after Polish mathematician Jan Lukasiewicz

Reverse Polish (POSTFIX) notation refers to the notation in which the operator symbol is placed after its two operands.

AB+   CD*   AB*CD+/

Polish PREFIX notation refers to the notation in which the operator symbol is placed before its two operands.

+AB   *CD   /*AB-CD

Advantages **Polish notations over infix expression**

a. No concept of operator priority.
b. Simplifies the expression evaluation rules.
c. No need of any parenthesis, Hence no ambiguity in the order of evaluation.
d. Evaluation can be carried out using a single scan over the expression string.

Conversion of infix to postfix.

We can also use a stack to convert an expression in standard form (otherwise known as infix) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression

a + b * c + (d * e + f) * g

into postfix. A correct answer is a b c * + d e * f + g * +.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol (+, *, (), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest.

When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on the stack, and which are to be completed prior to the current operator, are popped.

This is illustrated in the following table:

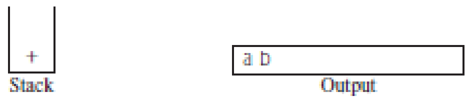
Expression	**Stack When Third Operator Is Processed**	**Action**
a*b-c+d	-	- is completed; + is pushed
a/b+c*d	+	Nothing is completed; * is pushed
a-b*c/d	- *	*is completed; / is pushed
a-b*c+d	- *	*and - are completed; + is pushed

Parentheses simply add an additional complication. We can view a left parenthesis as a high-precedence operator when it is an input symbol (so that pending operators remain pending) and a low-precedence operator when it is on the stack (so that it is not accidentally removed by an operator). Right parentheses are treated as the special case.

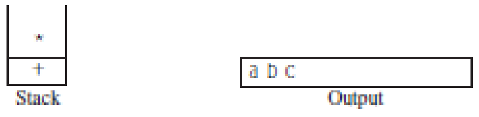
To see how this algorithm performs, we will convert the long infix expression above into its postfix form. First, the symbol a is read, so it is passed through to the output.

Then + is read and pushed onto the stack. Next b is read and passed through to the output.

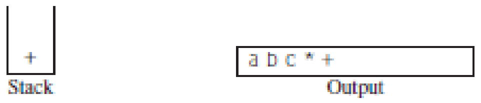
The state of affairs at this juncture is as follows:



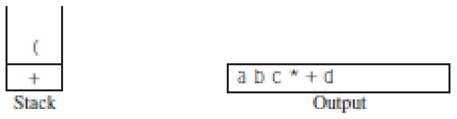
Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have



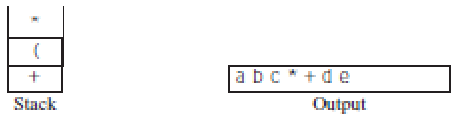
The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of lower but equal priority, on the stack; and then push the +.



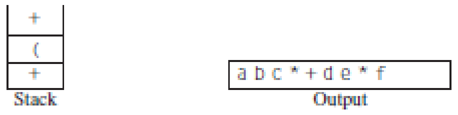
The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.



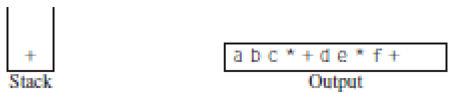
We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



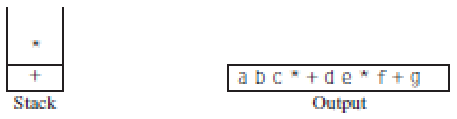
The next symbol read is a +. We pop and output * and then push +. Then we read and output f.



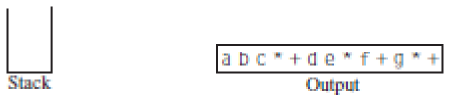
Now we read a), so the stack is emptied back to the (. We output a +.



We read a * next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



As before, this conversion requires only O(N) time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of operators.

Let's work on one more example before we formally develop the algorithm.

A + B * C – D / E converts to A B C * + D E / -

The transformation of this expression is shown in following figure. Because it uses all of the basic arithmetic operators, it is a complete test.

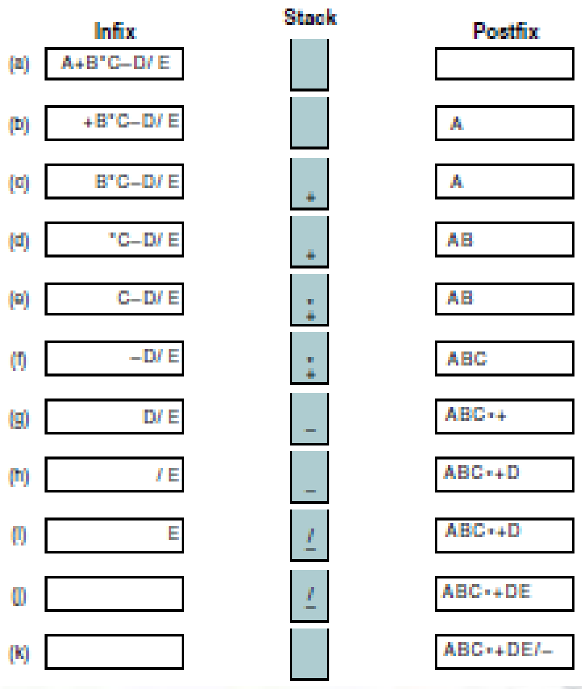


Fig.: Infix Transformation

We begin by copying the first operand, A, to the postfix expression. See Figure (b). The add operator is then pushed into the stack and the second operand is copied to the postfix expression. See Figure (d). At this point we are ready to insert the multiply operator into the stack. As we see in Figure (e), its priority is higher than that of the add operator at the top of the stack, so we simply push it into the stack. After copying the next operand, C, to the postfix expression, we need to push the minus operator into the stack. Because its priority is lower than that of the multiply operator, however, we must first pop the multiply and copy it to the postfix expression. The plus sign is now popped and appended to the postfix expression because the minus and plus have the same priority. The minus is then pushed into the stack. The result is shown in Figure (g). After copying the operand D to the postfix expression, we push the divide operator into the stack because it is of higher priority than the minus at the top of the stack in Figure (i).

After copying E to the postfix expression, we are left with an empty infix expression and two operators in the stack. See Figure (j). All that is left at this point is to pop the stack and copy each operator to the postfix expression. The final expression is shown in Figure (k).

We are now ready to develop the algorithm. We assume only the operators shown below. They have been adapted from the standard algebraic notation.

Priority 2: * /

Priority 1: + -

Priority 0: (

Evaluation postfix and infix with example

Now let's see how we can use stack postponement to evaluate the postfix expressions we developed earlier.

For example, given the expression shown below,

A B C + *

and assuming that A is 2, B is 4, and C is 6, what is the expression value?

The first thing you should notice is that the operands come before the operators. This means that we will have to postpone the use of the operands this time, not the operators. We therefore put them into the stack. When we find an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later.

Following figure traces the operation of our expression. (Note that we push the operand values into the stack, not the operand names. We therefore use the values in the figure.)

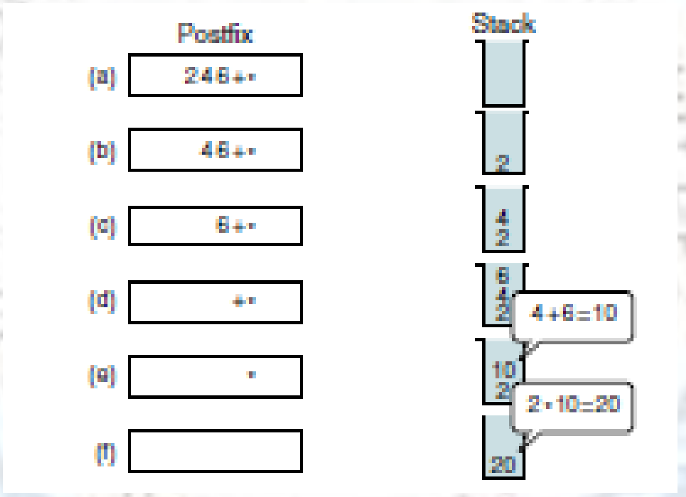


Fig.: Evaluation of Postfix expression

When the expression has been completely evaluated, its value is in the stack.

Algorithm / Methods / Steps:

- **Convert Infix expression to Postfix expression**

Algorithm inToPostFix (formula)

Convert infix formula to postfix.

Pre formula is infix notation that has been edited

to ensure that there are no syntactical errors

Post postfix formula has been formatted as a string

Return postfix formula

```
1 createStack (stack)
2 loop (for each character in formula)
          1 if (character is open parenthesis)
                    1 pushStack (stack, character)
          2 elseif (character is close parenthesis)
                    1 popStack (stack, character)
                    2 loop (character not open parenthesis)
                              1 concatenate character to postFixExpr
                              2 popStack (stack, character)
                    3 end loop
          3 elseif (character is operator) //Test priority of token to token at top of stack
                    1 stackTop (stack, topToken)
                    2 loop (not emptyStack (stack) AND priority(character) <= priority(topToken))
                              1 popStack (stack, tokenOut)
                              2 concatenate tokenOut to postFixExpr
                              3 stackTop (stack, topToken)
                    3 end loop
                    4 pushStack (stack, token)
          4 else  // Character is operand
                    1 Concatenate token to postFixExpr
          5 end if
3 end loop    //Input formula empty.
//Pop stack to postFix
4 loop (not emptyStack (stack))
          1 popStack (stack, character)
          2 concatenate token to postFixExpr
5 end loop
6 return postFix
end inToPostFix
```

➢ **Evaluation of Postfix Expressions**

```
Algorithm postFixEvaluate (expr)
This algorithm evaluates a postfix expression and returns its value.
Pre a valid expression
```

```
Post postfix value computed
Return value of expression
1 createStack (stack)
2 loop (for each character)
        1 if (character is operand)
                1 pushStack (stack, character)
        2 else
                1 popStack (stack, oper2)
                2 popStack (stack, oper1)
                3 operator = character
                4 set value to calculate (oper1, operator, oper2)
                5 pushStack (stack, value)
        3 end if
3 end loop
4 popStack (stack, result)
5 return (result)
end postFixEvaluate
```

Test Cases:

Validation

If Stack Empty	Display message “Stack Empty”
If memory not available	Display message “memory not available”
Parenthesis matching	Display appropriate message open/close parenthesis missing

Infix to postfix /prefix Test Cases

Expected Output: (Results / Visualization)

INPUT:	**POSTFIX OUTPUT:**
(A+B) * (C-D)	AB+CD-*
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$
A-B/(C*D$E)	ABCDE$*/-
A^B^C	ABC^^

INPUT:	**PREFIX OUTPUT:**
(A+B) * (C-D)	*+AB-CD
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG
A-B/(C*D$E)	-A/B*C$DE
A^B^C	^A^BC

Inference:

- Importance of tokenization.
- Need of stack in expression conversion and evaluation.

Oral questions:

1. Define Token Handling Rules used in infix to postfix conversion
2. Explain how precedence and associativity is used in conversion
3. With given example Convert infix expression to postfix.
   3 + 4 * 2 / (1 − 5) ^ 2 ^ 3.", A + B * C
4. How can expression conversion algorithm be used for identifying mismatched parentheses?
5. What is a stack?
6. What are the operations performed on a stack?
7. How is a stack implemented in an array?
8. What is the time complexity of stack operations?
9. What is the time complexity of inserting an element at the bottom of a stack?
10. What are the applications of a stack?
11. What is a stack overflow?
12. What is a stack underflow?
13. How can you implement a stack using two queues?
14. What is a postfix expression, and how can a stack be used to evaluate it?
15. What is a prefix expression, and how can a stack be used to evaluate it?
16. How can a stack be implemented using a linked list?
17. What is the time complexity of converting an infix expression to postfix?
18. What is the time complexity of converting an infix expression to prefix?
19. How can a stack be used to check if an expression containing three types of brackets [, (and { is balanced?

Assignment No: 5

Title: Priority Queue and Double-Ended Queue

Aim: To implement and demonstrate the working of a Priority Queue and a Double-Ended Queue (Deque) supporting standard operations such as insertion, deletion, and display.

Objective:

1. To understand the concept of priority-based data handling using a Priority Queue.
2. To implement insertion and deletion in a Priority Queue based on defined priority levels.
3. To develop a Deque allowing insertions and deletions at both the front and rear ends.

CO Mapped: CO1, CO2

Problem Statement:

Implementation of a Priority Queue

- Develop a **priority queue** where elements are dequeued based on priority rather than insertion order.
- Support operations such as insertion, deletion, and display.

Implementation of a Double-Ended Queue (Deque)

- Implement a **double-ended queue** where insertion and deletion can happen from both ends.
- Support operations such as **enqueue front, enqueue rear, deque front, dequeue rear**.

Expected Outcome:

1. Implement and manipulate Priority Queues and Deques using linear data structures.
2. Understand how different queue types optimize data access in real-world scenarios.

Pre-requisites:

1. Understanding of basic data structures
2. Basic knowledge of different types of queues and their Algorithmic Thinking for different operations on it

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- https://www.geeksforgeeks.org/dsa/priority-queue-set-1-introduction/
- https://www.geeksforgeeks.org/dsa/deque-set-1-introduction-applications/

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C/C++ IDE (CodeBlocks,Turbo C++, VSCode etc.)

Input: Minimum 5 Records

Theory / Procedure / Diagrams:

- **Priority Queue:** Concept, Types, Internal representation, operations, Use cases
- **Double Ended Queue:** Concept, Types, Internal representation, operations, Use cases
- Difference between Queue, Stack, Priority Queue, and Deque
- Time and Space Complexity of operations

Algorithm / Methods / Steps:

Priority Queue:

A normal priority queue maintains elements in order of their priority, but unlike heaps, it can be implemented more simply using arrays or linked lists — with less efficient performance but easier algorithms.

Assuming a max-priority queue (higher value = higher priority), here's the algorithm using a **sorted array (descending order of priority):**

- **insert(item, priority)**

```
1. Create a new element (item, priority).
2. Find the correct position in the array to insert:
    a. Start from beginning, find first element with lower priority.
3. Insert the new element at that position (shift others right).
```

- **peek()**

```
1. Return the first element in the array (index 0).
```

- **extract_max()**

```
1. Store the first element (index 0) as max.
2. Remove it from the array (shift elements left).
3. Return the stored element.
```

- **is_empty()**

```
1. Return true if array size == 0, else false.
```

Double-ended queue (deque)

It is a linear data structure that allows **insertion and deletion from both ends** — front and rear.

Below is the **algorithm using a circular array** (fixed-size array implementation)

Assume size = MAX

Let:

- arr[MAX] = storage
- front = index of front element
- rear = index of rear element
- Initially: front = -1, rear = -1

➢ **is_full()**

```
if ((front == 0 and rear == MAX - 1) or (front == rear + 1)):
    return True
else:
    return False
```

➢ **is_empty()**

```
if (front == -1):
    return True
else:
    return False
```

➢ **insert_front(x)**

```
if is_full():
    report overflow
else if is_empty():
    front = rear = 0
else if front == 0:
    front = MAX - 1
else:
    front = front - 1


arr[front] = x
```

➢ **insert_rear(x)**

```
if is_full():
    report overflow
```

```
else if is_empty():
    front = rear = 0
else if rear == MAX - 1:
    rear = 0
else:
    rear = rear + 1

arr[rear] = x
```

- **delete_front()**

```
if is_empty():
    report underflow
else if front == rear:
    front = rear = -1
else if front == MAX - 1:
    front = 0
else:
    front = front + 1
```

- **delete_rear()**

```
if is_empty():
    report underflow
else if front == rear:
    front = rear = -1
else if rear == 0:
    rear = MAX - 1
else:
    rear = rear – 1
```

- **get_front()**

```
if is_empty():
    report underflow
```

```
else:
    return arr[front]
```

➢ **get_rear()**

```
if is_empty():
    report underflow
else:
    return arr[rear]
```

Test Cases and Expected Output: (Results / Visualization)

Priority Queue – Test Cases

Test Case	Operation	Input	Expected Output
1	Insert elements	(10, priority 2), (5, 1), (20, 3)	Queue: 20 → 10 → 5
2	Peek (get highest priority)	—	20
3	Extract max	—	20 removed; Queue: 10 → 5
4	Increase priority	Increase 5 to priority 4	Queue: 5 → 10
5	Insert duplicate priority	(30, 2)	Queue: 5 → 30 → 10

Double-Ended Queue – Test Cases

Test Case	Operation	Input	Expected Output
1	Insert at rear	10, 20	Deque: 10 → 20
2	Insert at front	5	Deque: 5 → 10 → 20
3	Delete from rear	—	20 removed; Deque: 5 → 10
4	Delete from front	—	5 removed; Deque: 10
5	Mix operations	Insert 15 rear, Insert 25 front, Delete front	Deque: 10 → 15

Inference:

The implementation demonstrated how Priority Queues manage elements based on priority, making them ideal for scheduling tasks, while Deques allow insertion and deletion from both ends, offering flexibility in various algorithms. Implementing and testing these structures highlighted their key operations and practical applications.

Oral questions:

1. What is a priority queue? How does it differ from a normal queue?
2. Explain the difference between a max-priority queue and a min-priority queue.
3. Can a priority queue be implemented using an array or linked list? How?
4. What are the time complexities of insert, peek, and extract in a binary heap?
5. Why is a heap a preferred data structure for priority queues?
6. Describe how the insert operation works in a max-heap priority queue.
7. What is the use of the heapify operation in a priority queue?
8. How do you implement increase_key or decrease_key in a priority queue?
9. What happens if two elements have the same priority?
10. How do you implement a priority queue using a binary heap?
11. Name real-world applications of priority queues.
12. How is a priority queue used in Dijkstra's algorithm?
13. How does a priority queue work in task scheduling or CPU scheduling?
14. What is a double-ended queue (deque)?
15. How is a deque different from a queue or stack?
16. What are the primary operations supported by a deque?
17. What are the two types of deques? (Input-restricted and Output-restricted)
18. What data structures can be used to implement a deque?
19. How do you insert at the front and rear in a deque implemented as a circular array?
20. How do you check if a deque is full or empty?
21. What is the advantage of using a deque over a stack or a queue?
22. Can a deque be used to implement both stack and queue operations? How?
23. What are the differences between using a deque as a circular array vs. a doubly linked list?

Assignment No: 6

Title: Hashing

Aim: To design and implement a secure system for storing user credentials using hashing techniques and to demonstrate collision resolution using chaining and open addressing

Objective:

3. To securely store user credentials using hashing.
4. To explore and implement two methods of collision handling:
   - Chaining (using an array of linked lists)
   - Open Addressing (using linear or quadratic probing)
5. To understand and evaluate the performance of both methods.

CO Mapped: CO4

Problem Statement:

Develop a hashing-based system for securely storing user credentials, where usernames act as keys and hashed passwords as values. Implement and demonstrate collision handling using:

• Chaining (Array-based collision resolution)

• Open Addressing (Linear or Quadratic Probing without chaining)

Expected Outcome:

1. User credentials stored securely using hashing
2. Collision resolution using both chaining and open addressing
3. Performance comparison between the two methods

Pre-requisites:

1. Understanding of hash functions and hashing mechanisms
2. Knowledge of data structures such as arrays, linked lists
3. Basic knowledge of security and hashing algorithms (e.g., SHA-256)

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- https://www.geeksforgeeks.org/hashing-data-structure/
- https://www.youtube.com/watch?v=RVkQISfOCnI
- https://www.youtube.com/watch?v=shs0KM3wKv8
- Book: “Data Structures and Algorithm Analysis in C++” by Mark Allen Weiss
- https://www.programiz.com/dsa/hash-table

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C/C++ IDE (CodeBlocks,Turbo C++, VSCode etc.)

Input:

Theory / Procedure / Diagrams:

1. Hash Function: Maps a username (key) to an index
2. Password Hashing: Convert password into secure hashed form (e.g., SHA-256)
3. Chaining: Use linked list/array at each hash table index to handle collisions
4. Open Addressing: Use linear or quadratic probing to find alternate index

Algorithm / Methods / Steps:

Chaining Method:
1. Compute hash index from username
2. Hash password using secure algorithm
3. Insert user and hashed password in linked list at index

Open Addressing (Linear Probing):
1. Compute hash index from username
2. Hash password
3. If slot occupied, search next slot (i + 1)
4. Insert when empty slot is found

Test Cases:

Username	Password	Expected Index	Collision Handling
user1	pass123	3	Inserted directly
user2	abc@456	3	Resolved (chaining or probing)
user3	qwerty	4	Inserted directly

Expected Output: (Results / Visualization)

- Table with users and hashed passwords stored securely
- Collision resolution behavior visible (linked list or probing chain)
- Console output showing the hash index and storage steps

Inference:

Hashing provides an efficient way to store and retrieve user credentials. Chaining allows multiple entries at one index using linked lists, suitable for high-collision environments. Open addressing avoids extra space but may lead to longer search times under high load.

Oral questions:

1. What is the need for hashing passwords?
2. How does chaining differ from open addressing?
3. What are the advantages of using SHA-256 for password hashing?
4. Which collision resolution strategy is more memory efficient?
5. How do you ensure no two users have the same username?

Assignment No: 7

Title: FILE Handling

Aim: To implement Sequential File using CPP and perform various operation on it.

Objective:

1. Understand the concept of File Data structure
2. To learn Sequential file organization.
3. Create a sequential file and perform various operations on file.
4. Design the application using file data structure for database management.

CO Mapped: CO4

Problem Statement: Department maintains student database. The file contains roll number, name, division and address. Implement a CPP program to -

1. Create a sequential file to store and maintain student data.
2. It should allow the user to add and delete information of students.
3. Display information of particular student.
   i. If the student record does not exist an appropriate message is displayed.
   ii. If student record is found it should display the student details.

Expected Outcome:

- **Adding a student**: New student details are appended to the file.
- **Deleting a student**: The corresponding student record is removed, and the file is updated.
- **Displaying a student**: Information about a student is displayed if found; otherwise, a "not found" message is shown.
- **Invalid Input Handling**: If the user enters an invalid option or an unknown student roll number, appropriate messages are displayed.
- **Modifying Student Records:** The corresponding student record is modified, and the file is updated.

Pre-requisites:

- Understanding of File Operations
- File Modes in C++
- File Handling Classes in C++
- Understanding C++ Standard Library Functions

Suggested Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

- https://www.geeksforgeeks.org/cpp/file-handling-c-classes/

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE (Code::Blocks, Turbo C++, VSCode etc.)

Input: Minimum 5 records

Theory / Procedure / Diagrams:

- Concept of File
- Types of Files
- File organization (Concept, feature, drawback, operations)
- Applications of file

Algorithm / Methods / Steps:

Note: StudentData is file a name

➢ ***Create a file CreateAFile()***

```
Step 1: Open StudentData for output
Step 2:  If(file opened) then
Step 3:  scan noofReords
Step 4: For i =1 to noofRecords
        i. Display "Please enter the information of student: "
        ii. Get rollno, name, divison, address, date of birth, percentage, grade
        iii. Write rollno, name, division, address, date of birth, percentage, grade into StudentData
Step 5: end for
Step 6: Close file
Step 7: END
```

➢ ***Display a file DisplayFileContents()***

```
Step 1: Open StudentData for input
Step 2: If FileNotPresent
        i. Display error message
        ii. Exit
Step 3:  end if
Step 4: while Not EndofFile StudentData
        i. Read Student Information from StudentData
        ii. Display Student Information
Step 5: end while
Step 6: Close StudentData
Step 7: END
```

➢ ***Add a record AddNewRecords()***

```
Step 1: Open StudentData for append // file pointer will automatically moved to the end of file
Step2: If FileNotPresent
        i. Display error message
        ii. Exit
Step 3: end if
Step 4: Display "Please enter the information of student: "
        i. Get rollno, name, division, address, date of birth, percentage, grade
        ii. Write rollno, name, division, address, date of birth, percentage, grade into StudentData
Step 5: Close StudentData
Step 6: END
```

➢ ***Search a record SearchRecord(key)***

```
Step 1: Open StudentData for input
Step 2: if FileNotPresent
        i. Display error message
        ii. Exit
Step 3: end if
Step 4: while Not EndofFile StudentData
        i. Read Student Information from StudentData
        ii. if StudentRecord contains key, it can be unique roll no or if 'name' there can be multiple
        records displayed
        a. Display Student Information
        iii. end if
Step 5: end while
Step 6: Close StudentData
Step 7: END
```

➢ ***Modify a Record ModifyRecord(key)***

```
Step 1: Open StudentData for input
Step 2:  if FileNotPresent
        i. Display error message
        ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
        i. Read Student Information from StudentData
        ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Display Student Information
        b. Get new information for modification
        c. Write information into Temporary file
        iii. else
        a. Write information into Temporary file
        iv. end if
Step 6: end while
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10:END
```

➢ ***Delete a Record Delete Record(key)***

```
Step 1: Open StudentData for input
Step 2: if FileNotPresent
        i. Display error message
        ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
        i. Read Student Information from StudentData
        ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Continue      // read next record
        iii. else
        a. Write information into Temporary file
        iv. end if
Step 6: end while
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10: END
```

Test Cases:

1. File does not exist
2. File is open in read mode for writing
3. Appending records in the File

Expected Output: (Results / Visualization)

- Reading and Writing Data to Files
- Handling Large Data
- Data Persistence

Inference:

The implementation demonstrates the use of file handling in C++ to manage student records. It allows adding, deleting, and displaying data, reinforcing concepts of object-oriented programming and basic database operations.

Oral questions:

1. Which header file is required to use file I/O operations?
2. Which class is used to create an output stream?
3. Which class is used to create a stream that performs both input and output operations?
4. By default, all the files in C++ are opened in____ mode.

5. What is the use of ios::trunc mode?
6. What is the return type of open() method?
7. Which is the default mode of the opening using the fstream class?
8. Which is the default mode of the opening using the ifstream class?

Assignment No: 8

Title: Mini Project

Aim: Real-World Solution Development Using Fundamental Data Structures and Algorithms

Objective:

1. **Implement core linear data structures** to efficiently store and process data.
2. **Apply searching algorithms** to enable fast data retrieval and minimize search time.
3. **Use sorting techniques** to organize data for optimized processing and retrieval.
4. **Optimize performance** to ensure the solution can handle large datasets efficiently.
5. **Design a scalable and maintainable application** using fundamental DSA principles.

CO Mapped: CO5

Problem Statement:

Design and develop a real-world application that leverages **fundamental concepts of Data structures and Algorithms (DSA)**, including **linear data structures, searching, and sorting techniques** (with a preference for **hashing**). The solution should efficiently handle data storage, retrieval, and processing while optimizing performance.

Expected Outcome:

6. Gain practical experience in applying core data structures and algorithms—such as arrays, linked lists, searching, sorting, and hashing—to solve real-world problems.
7. Demonstrate efficient data storage, fast retrieval using hashing, and improved performance through appropriate algorithm selection, reinforcing the importance of algorithmic thinking and optimization in software development.

Pre-requisites: All concepts of Linear Data structures

Implementation Requirements: (Components / Digital Kits / Platform / Software / Hardware)

Platform: Any C++ IDE as per project requirements.

Input: As per project requirements

Instructions:

- Students have to form a group of **3 to 4 members**.
- The group should appoint a **Team Leader** responsible for coordinating and submitting the project.
- Members should distribute tasks evenly (e.g., coding, documentation, testing).
- **Presentation:** Each group will present their project, explaining their approach and findings.

Submit a **Project Report** including:

- Introduction
- Problem Definition
- Algorithm Explanation
- Code Implementation
- Test Cases and Results
- Conclusion and Future Scope

Note: Template will be provided for writing project report.

Sample problem statements:

Implementing a Library Management System using Data Structures

- **Description**: Design and implement a simple library management system. The system should support basic operations like adding new books, searching for books, borrowing books, and returning books.
- **Data Structures Involved**: Linked List for storing book information, Queue for borrowing/returning operations, and HashMap for fast book search.
- **Challenges**:
    - Implement efficient searching, borrowing, and returning of books.
    - Handle book availability (e.g., how to keep track of borrowed and available books).
    - Optimize the system for large datasets.

1. Student Management System

Problem Statement:
Design a system to store and manage student records (roll number, name, department, grades).

- Use arrays or linked lists to store records.
- Implement hashing for fast search by roll number.
- Support sorting by name or GPA.

◆ 2. Library Book Tracking System

Problem Statement:
Build a system to track books in a library, including title, author, ISBN, and availability.

- Use linear data structures to store book data.

- Hashing for fast lookup by ISBN.
- Allow sorting books by title or author name.

◆ 3. Contact Management System

Problem Statement:
Develop an application to store and manage contact information (name, phone, email).

- Use linked lists to store dynamic data.
- Implement hashing for quick search by name or phone number.
- Support alphabetical sorting of contacts.

◆ 4. Hospital Patient Record System

Problem Statement:
Create a system to manage patient records with details like patient ID, name, age, and diagnosis.

- Store data using arrays or linked lists.
- Use hashing for quick access by patient ID.
- Allow sorting patients by age or name.

◆ 5. Inventory Management System

Problem Statement:
Design a system to manage products in a store (ID, name, quantity, price).

- Use a linear data structure for storage.
- Hash product IDs for fast retrieval.
- Enable sorting by price or quantity.

◆ 6. Attendance Tracker

Problem Statement:
Create a tool to track attendance of students/employees over time.

- Use arrays/lists to store records.
- Search using hashing by ID or name.
- Sort by attendance percentage.