Data Structures
Course Code: CC-6313

Laboratory Manual

Department of Computer Science

Lahore Garrison University
Main Campus, Sector-C Phase-VI, DHA Lahore

Guidelines for Laboratory Procedure

The laboratory manual is the record of all work on your experiments. A complete, neat, and organized data record is as important as the experiment itself. Please follow these guidelines for efficient performance in the laboratory:

1. Attend the lab orientation to familiarize yourself with the lab setup.
2. Follow the designated lab schedule and complete assignments on time.
3. Write clear and well-documented code, avoiding plagiarism and unauthorized collaboration.
4. Seek help from lab instructors or peers if you encounter difficulties with programming concepts.
5. Regularly back up your code and project files to prevent data loss.
6. Use lab resources responsibly, including computers and software licenses.
7. If collaboration is allowed, work effectively with peers, ensuring each member contributes meaningfully.
8. Maintain a clean and organized workspace for better focus and efficiency.
9. Thoroughly test your code to identify and fix errors before submission.
10. Engage in lab discussions, share insights, and actively participate to enhance the learning experience.

Safety Precautions

1. Handle equipment carefully to prevent damage and avoid placing liquids near electronic devices.
2. Maintain an ergonomic workspace for comfortable and strain-free programming.
3. Save work frequently and use surge protectors to prevent data loss due to power issues.
4. Keep software and antivirus programs up to date and avoid downloading from untrusted sources.
5. Regularly back up code and important files to prevent data loss.
6. Establish clear communication and collaboration guidelines when working with others.
7. Be aware of emergency exits, fire extinguisher locations, and evacuation procedures.

Safety Undertaking

I have read all of the above, and I agree to conform to its contents.

Name: _____

Registration No.: _____

Section: _____

Semester: _____

Student Signature: _____

Lab Instructor: **Mr. Humayun Majeed.**

Lab Instructor Signature: _____

Grading Policy

Assessment	Weightage
Lab Performance	15%
Lab Report	15%
Open Ended Lab	10%
Project	20%
Final Term	40%

Rubrics

Lab Performance (Continuous Assessment) / Performance Test

SrNo	Performance Indicator	Exemplary (3)	Good (2)	Satisfactory (1)	Poor (0)
1	**Completeness and Accuracy**	Student demonstrates excellent capability in analyzing the given problem and designing an appropriate, accurate, and efficient solution	Student demonstrates good capability in analyzing the given problem and designing an appropriate solution	Student demonstrates fair capability in analyzing the given problem and designing a partially correct solution	Student demonstrates poor capability in analyzing the given problem and is unable to design a workable solution
2	**Coding Standards**	Code follows proper standards: well-structured, readable, consistent naming, proper indentation, and meaningful comments	Code mostly follows standards with minor issues in readability, naming, or commenting	Code has several issues with readability, structure, or lack of comments	Code does not follow standards; poorly structured and unreadable

Lab Reports

SrNo	Performance Indicator	Exemplary (3)	Good (2)	Satisfactory (1)	Poor (0)
1	**Demonstration**	Student demonstrates excellent capability in analyzing the given problem and designing an appropriate, accurate, and efficient solution	Student demonstrates good capability in analyzing the given problem and designing an appropriate solution	Student demonstrates satisfactory capability in analyzing the given problem and designing a partially correct solution	Student demonstrates poor capability in analyzing the given problem and is unable to design a workable solution
2	**Efficiency**	Code follows proper standards: well-structured, readable, consistent naming, proper indentation, and meaningful comments	Code mostly follows standards with minor issues in readability, naming, or commenting	Code partially follows standards but has multiple issues with readability, structure, or lack of comments	Code does not follow standards; poorly structured and unreadable

Viva Voce

SrNo	Performance Indicator	Exemplary (3)	Good (2)	Satisfactory (1)	Poor (0)
1	**Responsiveness to Questions/ Accuracy**	Responds confidently and accurately to all questions, showing clear understanding and strong reasoning	Responds accurately to most questions with minor errors or hesitation	Responds to some questions correctly but with noticeable gaps in accuracy or reasoning	Struggles to respond correctly; answers show little or no understanding
2	**Level of understanding of the learned skill**	Demonstrates excellent understanding and can apply the learned skill independently and effectively	Demonstrates good understanding and can apply the learned skill with minor guidance.	Demonstrates basic understanding but requires frequent guidance to apply the skill	Demonstrates poor understanding and is unable to apply the learned skill

OPEN-ENDED LAB's

SrNo	Performance Indicator	Exemplary (3)	Good (2)	Satisfactory (1)	Poor (0)
1	**Implementation of Operations** (Insert, Delete, Traverse)	All operations (insert, delete, forward & backward traversal) implemented correctly and efficiently.	Most operations work correctly with minor issues.	Some operations implemented but with errors.	Implementation is incomplete or incorrect.
2	**Data Handling & Logic** (Duplicate roll numbers, sorting, etc.)	Handles duplicates/sorting logically and efficiently with clear justification.	Attempts handling duplicates/sorting but not fully consistent.	Basic handling attempted but lacks clarity.	No consideration of duplicates/sorting.
3	**Code Quality & Structure** (Readability, modularity, comments)	Code is clean, well-structured, modular, and well-commented.	Code is understandable with minor issues in structure or comments.	Code works but lacks readability/modularity.	Code is messy, hard to read, or poorly structured.
4	**Creativity & Problem-Solving** (Open-ended choices)	Demonstrates innovative solutions or extra features beyond requirements.	Shows some creativity in problem-solving.	Meets only the basic requirements with minimal creativity.	No creativity, only a partial attempt at requirements.

Lab Project

SrNo	Performance Indicator	Exemplary (3)	Good (2)	Satisfactory (1)	Poor (0)
Project Design					
1	**Implementation and Completion**	Project is fully implemented without external assistance; runs correctly with efficient use of appropriate data structures	Project is implemented with minimal assistance; runs correctly with minor inefficiencies	Project is partially implemented or requires frequent assistance; output may be incomplete or inefficient	Project is incomplete or does not run correctly
2	**Appearance and Problem Analysis**	Problem is thoroughly analyzed; solution design is clear, logical, and optimal	Problem is well analyzed; solution design is mostly clear and logical	Problem analysis is basic; solution design is partially correct but limited	Problem is poorly analyzed; solution design is missing or incorrect
Coding Standards					
1	**Code Quality**	Code is well-structured, modular, with proper naming conventions, indentation, and meaningful comments	Code is mostly structured and readable; minor issues in formatting or comments	Code has several readability issues; limited comments; weak modularization	Code is poorly structured, hard to read, with no meaningful comments
Project Report					

1	**Structure and Explanation**	Report is well-organized, clear, and submitted on time; concepts explained thoroughly with strong references	Report is organized, mostly clear, with some references; submitted with minor delays	Report is somewhat disorganized; explanations limited; weak or no references	Report is poorly written, unclear, disorganized, or not submitted
2	**Results and Discussion**	Results are well-documented and analyzed; conclusions are logical and supported with references	Results are clear with some analysis; conclusions mostly logical	Results are presented with limited analysis; weak conclusions	Results missing, unclear, or without conclusions
Project Viva					
1	**Responsiveness to Questions**	Responds confidently, quickly, and accurately to all questions	Responds accurately to most questions with minor hesitation	Responds to few questions; limited accuracy	Struggles to respond; answers mostly incorrect
2	**Understanding of Concepts**	Demonstrates excellent understanding of data structures; explains with clarity and elaboration	Demonstrates good understanding; explains with some elaboration	Demonstrates basic understanding; explanations incomplete	Demonstrates poor understanding; unable to explain concepts
Project Presentation					
1	**Organization**	Presentation is clear, logical, and easy to follow	Presentation is mostly clear; minor confusion in flow	Presentation is somewhat organized; requires effort to follow	Presentation is disorganized and hard to follow

2	**Confidence**	Very confident; explains with clarity; maintains eye contact and gestures effectively	Confident to some extent; maintains some eye contact and gestures	Limited confidence; minimal eye contact and gestures	Lacks confidence; poor delivery with no eye contact
3	**Responsiveness to Audience**	Responds well to questions; clarifies and summarizes when needed	Generally responsive; provides basic clarifications	Reluctantly interacts; responses lack clarity	Avoids interaction; responds poorly
Teamwork					
1	**Sharing Information**	Actively shares relevant and valuable information with teammates	Shares adequate information; mostly relevant	Shares limited or partially relevant information	Rarely shares information; mostly irrelevant
2	**Fulfilling Team Duties**	Performs all assigned team responsibilities effectively	Performs most assigned duties	Performs only some assigned duties	Performs very few or no duties

Lab's Course Learning Outcomes

Course Title: Data Structures

Course Code: CC-6313

Instructor: Mr. Humayun Majeed

Designation: Assistant Lecturer

Department: Computer Science

E-mail: humayunmajeed@lgu.edu.pk

Students will be able to:

To provide a classroom and laboratory environment that enables students to become proficient in applying data structures and algorithms:

- **Apply** appropriate data structures and algorithms to explain, demonstrate their working, and determine their complexities in order to interpret problems and design solutions
- **Practice** and apply appropriate data structures and algorithms to design solutions

CLO1: Apply appropriate data structures and algorithms to explain, demonstrate their working, and determine their complexities in order to interpret problems and design solutions

CLO2: Practice and apply appropriate data structures and algorithms to design solutions

Mapping of CLO to PLO/GA

Course Code	CLOs	PLO 1	PLO 2	PLO 3	PLO 4	PLO 5	PLO 6	PLO 7	PLO 8	PLO 9	PLO 10
CC6313	CLO 1				X						
	CLO 2					X					

PLO1: Academic Education

PLO2: Knowledge for Solving Computing Problems

PLO3: Problem Analysis

PLO4: Design/ Development of Solution

PLO5: Modern Tool Usage

PLO6: Individual and Teamwork

PLO7: Communication

PLO8: Computing Professionalism and Society

PLO9: Ethics

PLO10: Lifelong Learning

Practical No.	Title	CLO
1	**Array Operations** – Traversal, Insertion, and Deletion	1
2	**Searching Algorithms** – Linear Search (Concept and Implementation), Binary Search (Working and Requirement of Sorted Arrays)	1
3	**Linked List** - Singly Linked List (Insertion, Deletion, and Traversal)	1
4	**Linked List -** Doubly Linked List and Circular Linked List (Insertion, Deletion, and Traversal)	1
5	**Stack** – Operations, Implementation (Array and Linked List), Applications (Infix to Postfix Conversion, Expression Evaluation)	1
6	**Queue Structures** – Linear, Circular, and Priority Queues (Array and Linked List Implementations)	1
7	**Basic Sorting Techniques** – Bubble Sort, Selection Sort, and Insertion Sort	1
8	**Recursive Sorting Techniques** – Merge Sort and Quick Sort	2
9	**Advanced Sorting Techniques** – Heap Sort, Max Heap, and Min Heap	2
10	**Binary Search Tree (BST)** – Properties, Insertion, Deletion, and Traversal	2
11	**AVL Trees** – Balancing, Rotations, and Height Property	2
12	**Graph Representations** – Adjacency List, Adjacency Matrix, Depth-First Search (DFS), and Breadth-First Search (BFS)	2
13	**Graph Algorithms** – Minimum Spanning Tree (Prim's and Kruskal's Algorithms)	2
14	**Graph Shortest Path Algorithms** – Dijkstra's Algorithm and Applications	2

PRACTICAL NO.01

Arrays Operation

OBJECTIVE:

PLO	CLO	LL
4	1	C-3

To review the concept of arrays and their declaration in C++.

To implement basic operations on arrays such as insertion, deletion, and updating of elements.

To implement algorithms for array manipulation.

Theory Overview

An array is a data structure that stores a collection of elements, each identified by at least one array index or key. It is a contiguous area of memory consisting of elements of the same type.

Declaration: In C++, arrays are declared using the syntax **type name [size]; ,** where **type** is the data type of the elements, **name** is the name of the array, and **size** is the number of elements in the array.

Initialization: Arrays can be initialized at the time of declaration or later using a loop or by assigning values to individual elements.

Accessing Elements: Elements of an array are accessed using their index. The index starts from 0 for the first element and goes up to **size-1** for the last element.

- **Array Operations**

Insertion

Deletion

Updating:

Traversal:

Searching:

Sorting:

Merging:

Applications: Arrays are used in various applications such as storing and processing data in databases, implementing matrices, representing images, and in many algorithms and program

LABORATORY TASK 01

Problem Statement:

You are required to develop a menu-driven C++ program that performs basic operations on an array of integers. The program should allow the user to perform insertion, deletion, updating, traversal**, searching, and sorting on the array.**

Instructions:

1. **Create an integer** array of size 10.
2. Implement the following operations:
    a. Insertion: Insert an element at a given position.
    b. Deletion: Delete an element from a given position.
    c. Updating: Update an element at a given index with a new value.
    d. Traversal: Display all elements of the array.
    e. Searching: Search for an element (using linear search).
    f. Sorting: Sort the array in ascending order.
3. Use a menu-based system where the user can choose which operation to perform.
4. Display the updated array after each operation.

CONCLUSION:

RUBRICS:

Lab Performance			**Lab Report**		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.02

Searching Algorithm

(Linear & Binary Search)

OBJECTIVE:

PLO	CLO	LL
4	1	C-3

- Understand the concept of **Linear Search** and its implementation.
- Understand and implement the concept of **Binary Search (iterative approach)**.
- Analyze and compare the **time complexity** of both algorithms.
- Demonstrate the **efficiency differences** of linear and binary search for different array sizes.
- Apply search algorithms in **practical scenarios** such as student record management or library databases.
- Illustrate the **best-case, worst-case, and average-case scenarios** of both search algorithms.

Theory Overview

1. Linear Search

Concept of Linear Search: Linear search is a simple search algorithm that sequentially checks each element in a collection until a match is found or the whole collection has been searched. It is also known as a sequential search.

Basic Implementation of Linear Search

- Start from the first element of the array.
- Compare the key element with each element of the array sequentially.
- If the key element matches any element, return the index of that element.
- If the key element is not found after checking all elements, return a specific value (e.g., -1) to indicate that the element is not in the array.

Time Complexity Analysis

Worst-case

If the key element is not present in the array or is present at the last position, the algorithm will have to check all n elements. The worst-case time complexity is O(n).

Best-case

If the key element is found at the first position, the algorithm will perform only one comparison. The best-case time complexity is O(1).

Average-case

On average, the algorithm will have to check half of the n elements. The average-case time complexity is O(n/2), which simplifies to O(n).

Efficiency for Different Sizes of Arrays

Linear search is straightforward and easy to implement, but it is not efficient for large datasets. As the size of the array increases, the time taken for linear search also increases linearly.

Practical Application in Student Records or Databases

Linear search can be applied to search for a student's record in a list of student records or to search for a specific entry in a database.

For example, in a student database, linear search can be used to find a student's details based on their student ID or name.

Illustration of Worst-case, Best-case, and Average-case Scenarios

The **worst-case** scenario occurs when the key element is not present in the array or is present at the last position, requiring n comparisons.

The **best-case** scenario occurs when the key element is found at the first position, requiring only one comparison.

The **average-case** scenario occurs when the key element is found in the middle of the array, requiring approximately n/2 comparisons on average.

2. Binary Search

Binary search is an efficient algorithm for finding a target value within a sorted array. It works by repeatedly dividing the search interval in half. If the value of the target is less than the middle element of the interval, the search continues in the lower half. If the value is greater, the search continues in the upper half. This process is repeated until the target value is found or the interval is empty.

Iterative Approach

1. Elements in array must be sorted.
2. Divide the array into two parts.
3. Use variables mid, start and end.
4. Where start = 0 and end = no of elements – 1.
5. For searching use while loop.
6. Set the value of mid = start + end / 2.
7. Compare the element on mid index with target value.
8. If matched display found else compare it with the element present on start and end index.
9. If the element on mid index is smaller than target value than start = mid + 1.
10. If the element on mid index is greater than target value than end = mid - 1.
11. Repeat step 6-10 until the match is found.

Time Complexity

Binary search has a time complexity of O(log n), where n is the number of elements in the array. This makes it significantly more efficient than linear search (O(n)), especially for large arrays.

Applications

Binary search is commonly used in scenarios where the data is sorted and efficient searching is required. It is used in various applications such as searching in databases, finding elements in sorted arrays, and more.

Limitations

Binary search requires the array to be sorted, which can add an additional overhead if the array is frequently modified.

It may not be suitable for small arrays or unsorted data, as the overhead of sorting or maintaining a sorted array may outweigh the benefits of binary search.

Comparisons

Linear Search: Binary search is more efficient than linear search for large arrays, as it has a lower time complexity.

Conclusion

Binary search is a powerful algorithm for efficiently finding elements in a sorted array. Understanding its implementation and limitations can help in applying it effectively to various problem-solving scenarios.

LABORATORY TASK 02

Problem Statement:

A university wants to create a simple search system for its **student ID records**. The system should allow searching for a student ID using both **Linear Search** and **Binary Search** methods.

Instructions:

1. Create an array of **10 student IDs** (e.g., {101, 105, 110, 120, 125, 130, 140, 150, 160, 170}).
2. Implement **Linear Search** to search for a student ID entered by the user.
3. Implement **Binary Search** on the **sorted array** for the same user input.
4. Display the position of the student ID if found, otherwise display "Not Found".
5. Compare the number of comparisons made by both algorithms.

Expected Output (Sample):

Enter Student ID to search: 120
Linear Search → Found at index 3 (Comparisons: 4)
Binary Search → Found at index 3 (Comparison s: 2)

CONCLUSION:

RUBRICS:

Lab Performance			**Lab Report**		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.03

SINGLY LINKED LIST

Objectives:

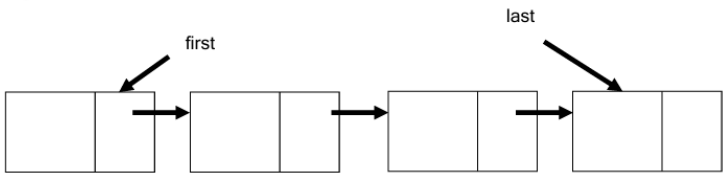
PLO	CLO	LL
4	1	C-3

(a). Understand the concepts of singly linked lists

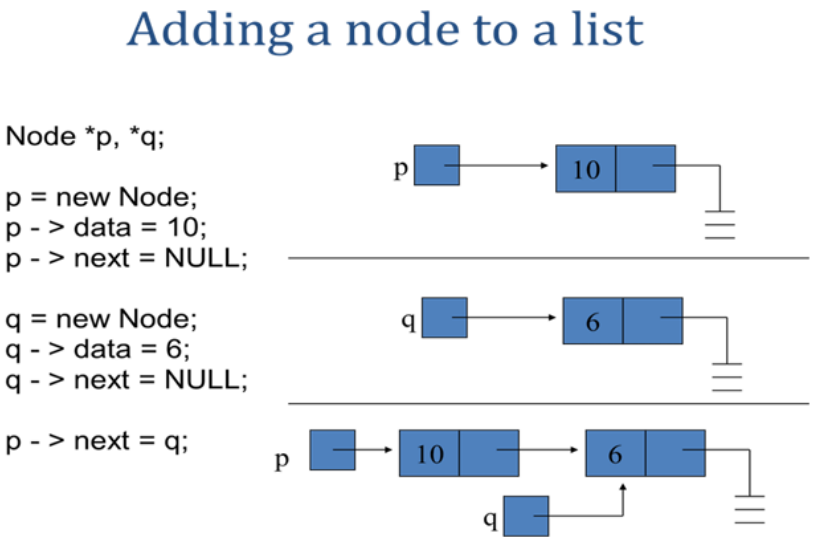
(b). Implement singly linked list using dynamic structures

Theory Overview:

Linked Lists: A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a datum and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.

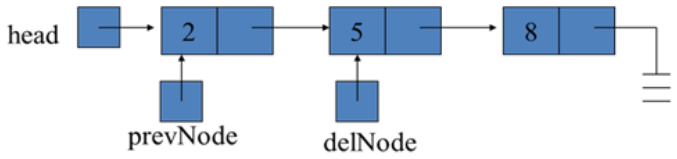


Insertion: To insert data in a linked list, a record is created holding the new item. The nextpointer of the new record is set to link it to the item which is to follow it in the list. The nextpointer of the item which is to precede it must be modified to point to the new item.



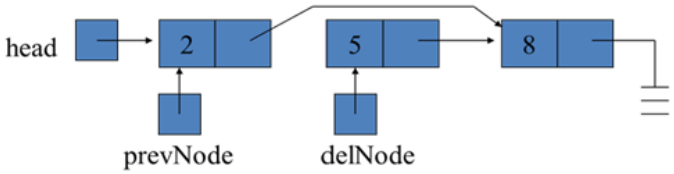
Deletion: To deleted data from list, The nextpointer of the item immediately preceding the one to 6. be deleted is altered and made to point to the item following the deleted item.

Deleting a node from a list



Step 1: Redirect pointer from the Node before the one to be deleted to point to the Node after the one to be deleted.

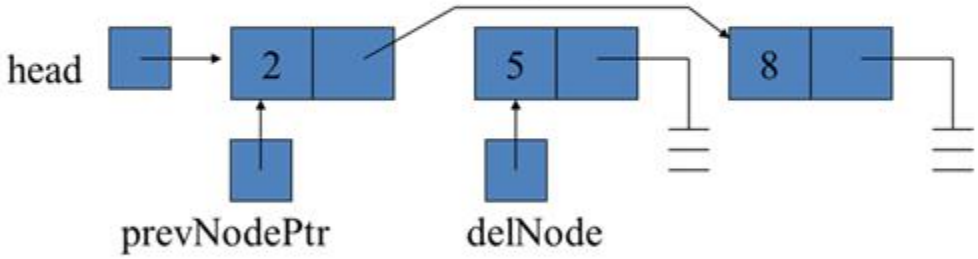
```
prevNode - > next = delNode - > next;
```



Finishing the deletion

Step 2: Remove the pointer from the deleted link.

```
delNode - > next = NULL;
```



Step 3: Free up the memory used for the deleted node:

```
delete delNode;
```

LABORATORY TASK 03

Problem Statement:

Design and implement a comprehensive menu-driven C++ program for a Singly Linked List that supports advanced operations. The program should allow the user to:

1. **Insertion:**
    a. At the beginning, end, or a specific position.
    b. Insert a new node after a given value.
2. **Deletion:**
    a. From the beginning, end, or a specific position.
    b. Delete a node by value (not just by position).
3. **Searching:**
    a. Search for an element and return its position(s).
    b. If multiple nodes have the same value, display all positions.
4. **Updating:**
    a. Update the value of a node at a given position.
    b. Update all nodes with a given value to a new value.
5. **Reversing the list:**
    a. Reverse the entire linked list in place (without creating a new list).
6. **Counting & Statistics:**
    a. Count the total number of nodes in the list.
    b. Find the maximum and minimum values in the list.
7. **Merging:**

    - Create two linked lists and merge them into a single list.

(Lab Task 03 Rubrics Page)

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.04

Double Linked List

PLO	CLO	LL
4	1	C-3

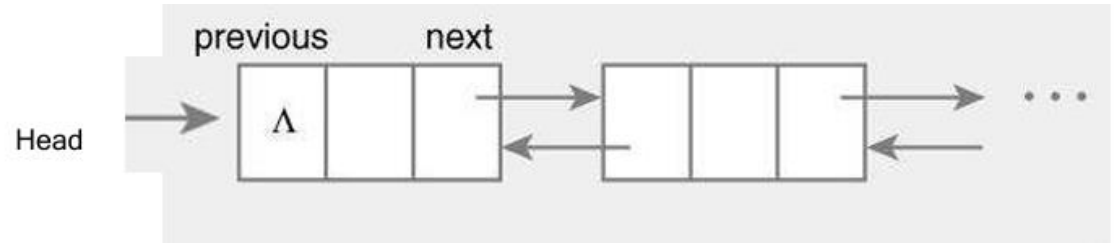
OBJECTIVE:

Understanding the concepts and operations of doubly linked lists.

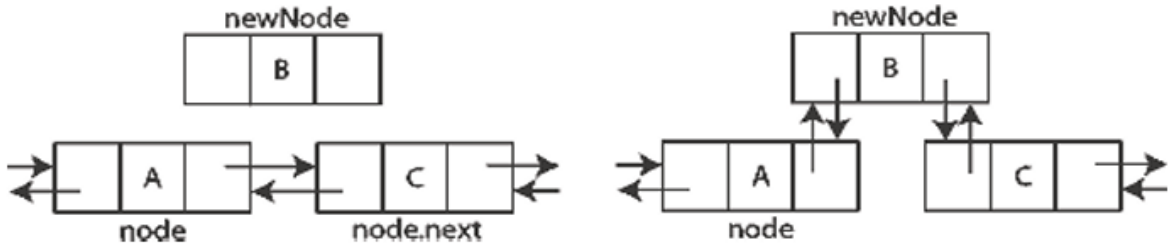
Implement doubly linked list using dynamic structures.

Theory Overview

Doubly Linked Lists: A doubly linked list is a data structure consisting of a group of nodes which together represent a sequence and are connected to each other in both directions (Bi-directional). In Doubly linked list, each node has two pointers. One pointer to its successor (NULL if there is none) and one pointer to its predecessor (NULL if there is none) which enables bi-directional traversing.

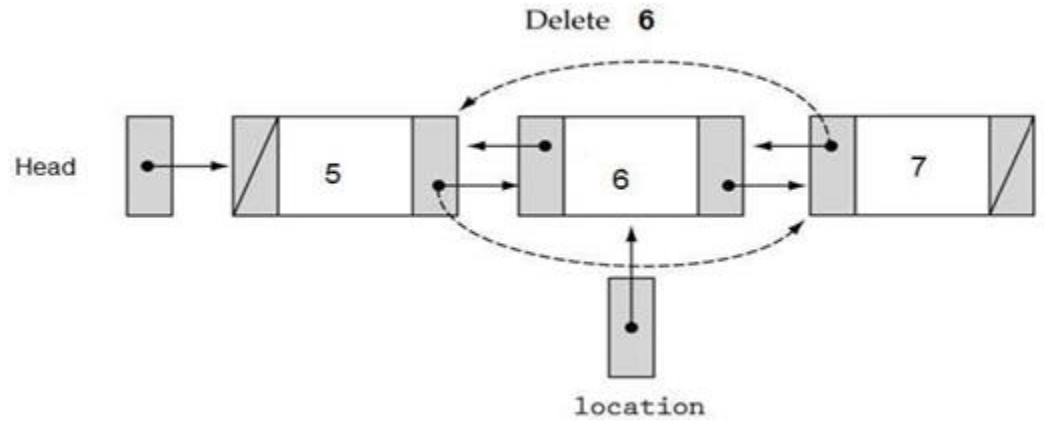


Insertion: To insert data in a doubly linked list, a record is created holding the new item. The next pointer of the new record is set to link it to the item which is to follow it in the list. The pre-pointer of the new record is set to link it to the item which is to before it in the list.



Deletion: In deletion process, element can be deleted from three different places. When the node

is deleted, the memory allocated to that node is released and the previous and next nodes of that node are linked.



LABORATORY TASK 04

Problem Statement:

Write a **menu-driven C++ program** to implement a **Doubly Linked List (DLL)** using dynamic memory allocation. The program should allow the user to perform the following operations:

1. **Insertion:**
    a. At the beginning of the list
    b. At the end of the list
    c. At a specific position
2. **Deletion:**
    a. From the beginning of the list
    b. From the end of the list
    c. From a specific position
3. **Traversal:**
    a. Display the list in **forward direction**
    b. Display the list in **reverse direction**
4. **Searching:**
    a. Search for a given value and display its position(s)
5. **Updating:**
    a. Update the value of a node at a given position
6. **Counting:**
    a. Count the total number of nodes in the list

CONCLUSION:

RUBRICS:

Lab Performance			**Lab Report**		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.05

Stack

OBJECTIVE:

PLO	CLO	LL
4	1	C-3

Understand the applications of stacks.

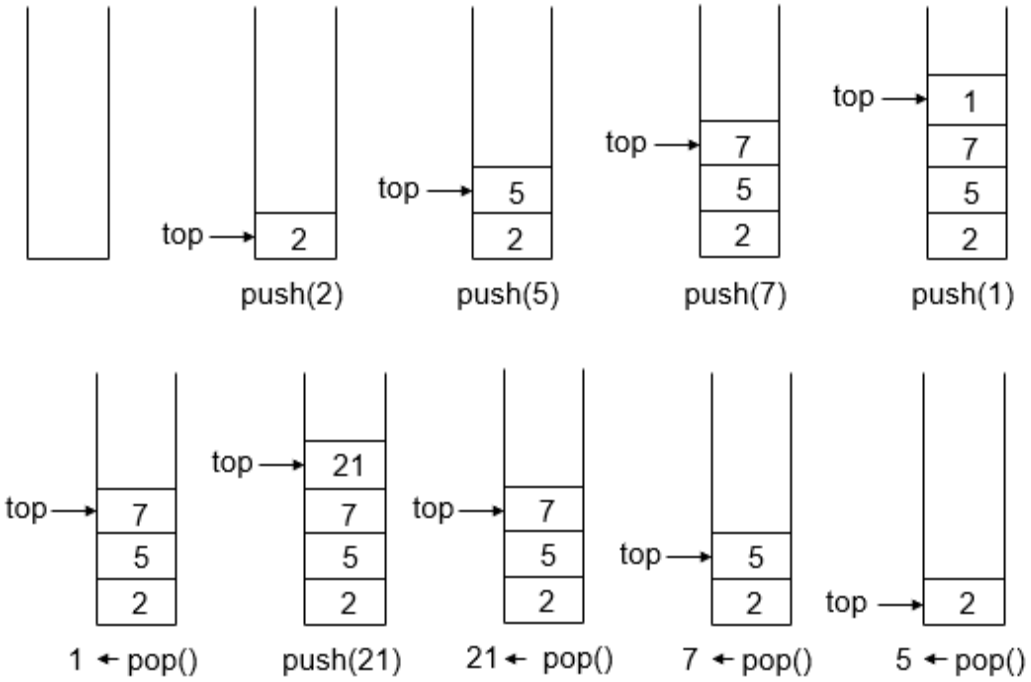
Implement stacks using both static and dynamic implementation.

Theory Overview

Stack is a memory portion, which is used for storing the elements. The elements are stored based on the principle of LIFO (Last In, First Out). In stack insertion and deletion of elements take place at the same end (Top). The last element inserted will be the first to be retrieved.

• This end is called Top

• The other end is called Bottom



Top: The stack top operation gets the data from the top-most position and returns it to the user

without deleting it. The underflow state can also occur in stack top operation if stack is empty.

Push: The push operation adds a new element to the top of the stack or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state.

Pop: The pop operation removes an item from the top of the stack. A pop either returns previously inserted items, or NULL if stack is empty.

Underflow: When stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as overflow.

Stacks can be implemented using arrays or linked lists.

Arrays are simpler but have a fixed size, while linked lists can grow dynamically but require more memory per element.

Time Complexity

The time complexity of push, pop, peek, isEmpty, and size operations is O(1) for both array and linked list implementations.

LABORATORY TASK 05

Q1. Write c++ program to take a string expression as input from user. Using this infix expression, you must convert it into its equivalent postfix notation.

Example: **a + (b * c)**

Read an Expression	Stack	Output
a		a
+	+	a
(	+ (	a
b	+ (	ab
*	+ (*	ab
c	+(*	abc
)	+	abc*
		abc*+

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.06

QUEUES

OBJECTIVE:

PLO	CLO	LL
4	1	C-3

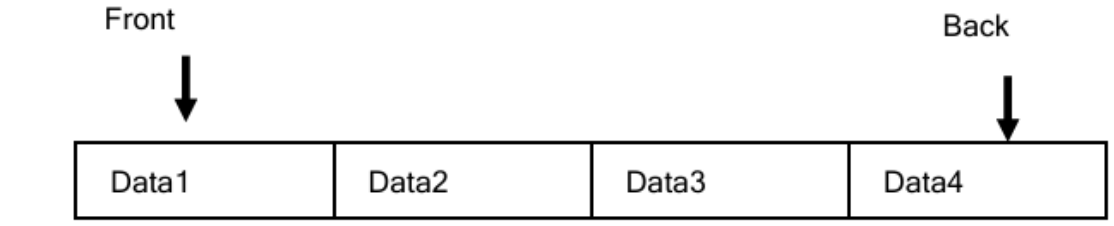
To understand and implement the concepts of queues using data structures (b). To learn priority queue and its implementation

Theory Overview

Queue: A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Static: Queue is implemented by an array, and size of queue remains fix

Dynamic: A queue can be implemented as a linked list, and expand or shrink with each enqueue or dequeue operation.



Enqueue: Enqueue ( ) operation adds a new item to the end of the queue, or initializes the queue if it is empty.

Dequeue: Dequeue ( ) operation removes an item from the front of the queue. A Dequeue operation on an empty stack results in an underflow.

LABORATORY TASK 06

Q1. Write a menu driven program to perform different operations with queue such Enqueue ( ), Dequeue( ) and DisplayQueue( ).

Q2. You must take a single string as input. Using this input string, you have to create multiple queues in which each queue will comprise of separate word appeared in input string. At the end, you will again concatenate all queues to a single queue.

Example:

String = <Data Structure and Algo=

Q1 = D → a → t → a

Q2 = S → t → r → u → c → t → u → r → e

Q3 = a → n → d

Q4 = A → l → g → o

At the end concatenate all queues and display them.

Q1 → Q2 → Q3 → Q4

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.07

Basic Sorting Algorithms
(Bubble, Selection, & Insertion Sort)

PLO	CLO	LL
4	1	C-3

Objective

- To understand the basic **comparison-based sorting algorithms**.
- To implement **Bubble Sort, Selection Sort, and Insertion Sort**.
- To compare the performance (steps/swaps) of these algorithms on the same dataset.

Theory Overview

Sorting is one of the most fundamental operations in computer science. It is the process of arranging data in a specific order (commonly ascending or descending). Sorting makes searching and data organization much more efficient and is used in various applications such as ranking systems, database management, report generation, and data analysis.

Sorting algorithms are broadly classified into two categories:

1. **Comparison-Based Sorting:** Sorting is achieved by comparing elements with each other (e.g., Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Non-Comparison-Based Sorting:** Sorting is achieved without comparisons, using techniques like counting, hashing, or distribution (e.g., Counting Sort, Radix Sort, Bucket Sort).

In this lab, we will focus on **comparison-based basic sorting techniques**: **Bubble Sort, Selection Sort, and Insertion Sort.**

1. Bubble Sort

- Bubble Sort is the simplest sorting technique.
- The algorithm works by repeatedly stepping through the list, comparing each pair of adjacent elements, and swapping them if they are in the wrong order.
- After the first pass, the largest element is moved to the end of the list (like a bubble rising to the surface).
- After the second pass, the second largest element is in place, and so on.
- This continues until no swaps are needed, meaning the array is sorted.

Key Characteristics:

- Easy to implement.
- Works well for small datasets.
- Inefficient for large datasets due to its quadratic time complexity.

Complexity Analysis:

- **Best Case:** O(n) (if the array is already sorted, only one pass is required).

- **Worst Case:** $O(n^2)$ (if the array is sorted in reverse order).
- **Average Case:** $O(n^2)$.
- **Space Complexity:** $O(1)$ (in-place sorting).

2. Selection Sort

- Selection Sort works by dividing the array into two parts: **sorted** and **unsorted**.
- Initially, the sorted part is empty and the unsorted part contains the entire array.
- The algorithm finds the minimum element from the unsorted part and places it at the beginning of the sorted part.
- This process is repeated for the remaining unsorted elements until the array is completely sorted.

Steps:

1. Find the smallest element in the array.
2. Swap it with the element at the first position.
3. Move the boundary of the sorted part one step ahead.
4. Repeat until all elements are sorted.

Key Characteristics:

- Fewer swaps compared to Bubble Sort.
- Performance does not depend on initial arrangement of elements.
- Still inefficient for large datasets because of $O(n^2)$ comparisons.

Complexity Analysis:

- **Best Case:** $O(n^2)$.
- **Worst Case:** $O(n^2)$.
- **Average Case:** $O(n^2)$.
- **Space Complexity:** $O(1)$.

3. Insertion Sort

- Insertion Sort works the way we arrange playing cards in our hands.
- It builds the sorted array one item at a time by comparing the new element with already sorted elements and inserting it at its correct position.
- The sorted portion grows as each new element is inserted.

Steps:

1. Assume the first element is already sorted.
2. Pick the next element.
3. Compare it with elements in the sorted part and shift the elements to make space.
4. Insert the element in its correct position.
5. Repeat until all elements are sorted.

Key Characteristics:

- Very efficient for small datasets or nearly sorted data.
- Performs better than Bubble and Selection sort in many real-world cases.
- Adaptive: requires fewer steps when the list is already partially sorted.

Complexity Analysis:

- **Best Case:** O(n) (when the array is already sorted).
- **Worst Case:** $O(n^2)$ (when the array is in reverse order).
- **Average Case:** $O(n^2)$.
- **Space Complexity:** O(1).

LABORATORY TASK 07

Scenario:
Imagine you are developing a **Student Result Management System**. You have a list of students along with their marks in a course. For report generation, the system must display the students in **ascending order of marks**. Since you want to explore different methods of sorting, you will implement **Bubble Sort, Selection Sort, and Insertion Sort** to order the student marks.

Tasks to Perform

1. **Input**:
    a. Take n student marks from the user (array input).
2. **Sorting Options (Menu Driven Program):**
    a. **Option 1:** Sort using **Bubble Sort**
    b. **Option 2:** Sort using **Selection Sort**
    c. **Option 3:** Sort using **Insertion Sort**
3. **Output:**
    a. Display the sorted marks in ascending order.
    b. Show the **number of comparisons** and **swaps** for analysis.

CONCLUSION:

RUBRICS:

Lab Performance			**Lab Report**		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.08

Recursive Sort (Merge, & Quick Sort)

PLO	CLO	LL
5	2	P-3

OBJECTIVE:

Understanding the concepts of recursion and its practical usage

Implementation of recursion

Theory Overview:

Recursion: Recursion is a programming technique in which procedures and functions call themselves. When a function calls itself, it is making a recursive call. This approach can be applied to solve many types of problems. Most computer programming languages support recursion by allowing a function to call itself within the program.

Recursive Call:A function call in which the function being called is the same as the one making the call. Base Case: • The case for which the solution can be stated non-recursively • The case for which the answer is explicitly known. Recursive Case:A recursive function definition also contains one or more recursive cases; meaning input(s) for which the program recurs (calls itself). The job of the recursive cases can be seen as breaking down complex inputs into simpler ones. In a properly-designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached. For example, the factorial function can be defined recursively by the equations 0! = 1 and, for all n > 0, n! = n (n − 1)! None of equation by itself constitutes a complete definition, the first is the base case, and the second is the recursive case.

Recursive Sort

Recursive Sort includes Merge sort and Quick sort

Quick Sort

Concept: Quick Sort is a divide-and-conquer algorithm that 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. The sub-arrays are then sorted recursively.

Steps:

1. Choose a pivot element from the array.

2. Rearrange the elements so that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Merge Sort

Concept: Merge Sort is also a divide-and-conquer algorithm that works by dividing the array into two halves, sorting each half, and then merging the sorted halves to produce a sorted array.

Steps:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into a single sorted array.

LABORATORY TASK 08

You are assigned the task of developing a **Library Book Management System** in C++. The library maintains a record of book IDs, which must be arranged in ascending order for efficient searching.

1. Implement **Merge Sort** to arrange the list of book IDs.
2. Implement **Quick Sort** to arrange the same list of book IDs.
3. Compare the execution time of both algorithms for different dataset sizes (e.g., 10 books, 100 books, 1000 books).
4. Analyze which algorithm performs better and under what circumstances.

Expected Output:

- Unsorted list of book IDs.
- Sorted list using Merge Sort.
- Sorted list using Quick Sort.
- Time taken by both algorithms for comparison.

CONCLUSION:

RUBRICS:

Lab Performance			**Lab Report**		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.09

Advanced Sorting Techniques

(Heap Sort, Max Heap, and Min Heap)

Objectives

PLO	CLO	LL
5	2	P-3

1. To understand the concept of Heap Data Structure and its types (Max Heap and Min Heap).
2. To implement Heap Sort for efficient sorting of large datasets.
3. To learn the process of building a Max Heap and a Min Heap from an array.
4. To apply heap operations (insertion, deletion, heapify) in practical problem-solving.
5. To compare Heap Sort with other sorting algorithms in terms of time complexity and space efficiency.

Theory Overview

Sorting plays a critical role in computer science applications, ranging from database indexing to scheduling and optimization. For large datasets, efficient algorithms like Heap Sort are often preferred. Heap Sort is based on the Heap Data Structure, a specialized complete binary tree that satisfies the Heap Property.

1. Heap Data Structure

A Heap is a complete binary tree where all levels are completely filled except possibly the last, which is filled from left to right.

- **Max Heap:**

The value of each parent node is greater than or equal to the values of its children. The largest element is always at the root.

Example: [90, 70, 50, 40, 60, 30, 20] → Root = 90 (largest element).

- **Min Heap:**

The value of each parent node is less than or equal to the values of its children. The smallest element is always at the root.

Example: [10, 20, 15, 30, 40] → Root = 10 (smallest element).

2. Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses the Heap data structure to sort elements. It works in two phases:

Steps of Heap Sort:

1. Build a Max Heap from the given array.
2. The root element (maximum value) is swapped with the last element.
3. The heap size is reduced, and the heap property is restored (heapify).
4. Repeat until the array is sorted.

Example:
Unsorted array: [4, 10, 3, 5, 1]

- Build Max Heap: [10, 5, 3, 4, 1]
- Swap root (10) with last → [1, 5, 3, 4, 10]
- Heapify → [5, 4, 3, 1, 10]
- Repeat until sorted: [1, 3, 4, 5, 10]

3. Complexity Analysis

- Heap Construction: O(n)
- Heapify Operation: O(log n)

- Heap Sort: O(n log n)

Best Case: O(n log n)

Worst Case: O(n log n)

Average Case: O(n log n)

Space Complexity: O(1) (in-place sorting).

4. Applications

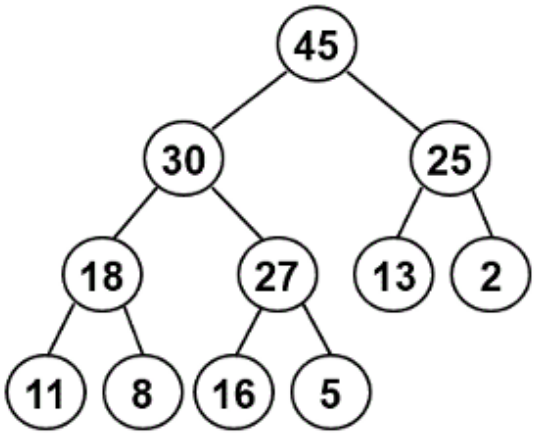
- Scheduling problems (e.g., CPU job scheduling, priority queues).
- Graph algorithms (e.g., Dijkstra's shortest path uses Min Heap).
- Finding Kth largest or smallest element in an array.
- Real-world systems like memory management, bandwidth allocation, and task prioritization.

LABORATORY TASK 09

Q1.

Your work for this task should be done on paper. Please show your work to a staff member before you leave the lab.

Consider the following heap:



Which node would be in position 4 of the corresponding array?

1. Given the node in position 4, how could you use arithmetic to determine:
   - the position of its left child (if any) in the array
   - the position of its right child (if any) in the array
   - the position of its parent in the array
2. If we call the remove() method on this heap:
   - Which item will be removed?
   - Which item will be copied into the position of the removed item and then be sifted down to restore the heap?
   - What will the tree look like after this call to remove()?

3. What will the heap look like after a second call to remove()? After a third call?

4. After completing the three calls to remove(), we then use the insert() method to add an item with a key of 21. What will the tree look like after that insertion?

(Lab Task 09 Rubrics Page)

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.10

BINARY SEARCH TREE

OBJECTIVE:

PLO	CLO	LL
5	2	P-3

a). Understand concepts and usages of binary search tree (b). Practice the implementation of binary search tree

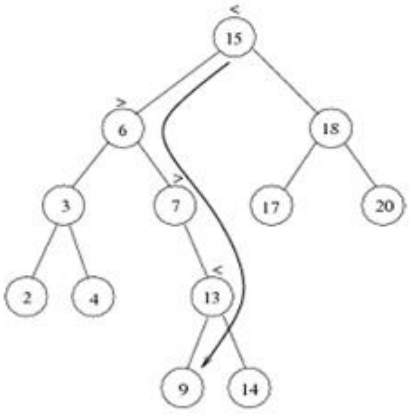
Theory Overview:

Binary Search Trees: Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently. Binary search tree is either empty or each node N of tree satisfies the following property

• The Key value in the left child is not more than the value of root

• The key value in the right child is more than or identical to the value of root

• All the sub-trees, i.e. left and right sub-trees follow the two rules mention above.



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Minimum: The minimum element of a binary search tree is the last node of the left roof

Maximum: The maximum element is the last node of the right roof.

LABORATORY TASK 10

Q1. Write a menu driven program of binary search tree that involves following operations:

1. Inserting and deleting an element
2. Breadth First Traversal and Depth First Traversal
3. Finding height of a tree
4. Maximum and Minimum node

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.11

AVL Tree

OBJECTIVE:

PLO	CLO	LL
5	2	P-3

Understand the concepts if AVL Tree.

Implementation of AVL Tree.

Theory Overview:

An AVL tree is a self-balancing binary search tree (BST), named after its inventors Adelson-Velsky and Landis. It maintains balance by ensuring the heights of the left and right subtrees of any node differ by at most one. This balance is achieved through rotations during insertion and deletion operations, which guarantees that the tree remains approximately balanced, providing efficient search, insertion, and deletion operations with a time complexity of O(log n).

Properties of AVL Trees

1. **Binary Search Tree Property**:
   a. For any node NNN, all elements in the left subtree of NNN are less than NNN and all elements in the right subtree are greater than NNN.
2. **Balance Factor**:
   a. The balance factor of a node is defined as the height difference between its left and right subtrees.
   b. For every node NNN, the balance factor BF(N)BF(N)BF(N) is calculated as: BF(N)=height(left subtree)−height(right subtree)BF(N) = \text{height(left subtree)} - \text{height(right subtree)}BF(N)=height(left subtree)−height(right subtree)
   c. The tree is balanced if the balance factor of every node is either -1, 0, or 1.
3. **Height**
   a. The height of an AVL tree with nnn nodes is O(log⁡n)O(\log n)O(logn).

Rotations in AVL Trees

To maintain the balance factor within the permissible range, AVL trees perform rotations when nodes are inserted or deleted. There are four types of rotations:

1. Right Rotation (Single Rotation):

- **a.** Used to balance a left-heavy subtree.
- **b.** Applied when a left child's left subtree causes imbalance.
- **c.** Procedure: Perform a right rotation on the unbalanced node.

2. Left Rotation (Single Rotation):

- **a.** Used to balance a right-heavy subtree.
- **b.** Applied when a right child's right subtree causes imbalance.
- **c.** Procedure: Perform a left rotation on the unbalanced node.

3. Left-Right Rotation (Double Rotation):

- **a.** Used to balance a left-heavy subtree with a right-heavy left child.
- **b.** Applied when a left child's right subtree causes imbalance.
- **c.** Procedure: Perform a left rotation on the left child.
- **d.** Perform a right rotation on the unbalanced node.

4. Right-Left Rotation (Double Rotation):

- **a.** Used to balance a right-heavy subtree with a left-heavy right child.
- **b.** Applied when a right child's left subtree causes imbalance.
- **c.** Procedure: Perform a right rotation on the right child.
- **d.** Perform a left rotation on the unbalanced node.

AVL Tree Operations

1. Insertion:

- **a.** Insert the new node as in a standard BST.
- **b.** Update the height of the ancestor nodes.
- **c.** Rebalance the tree by performing the necessary rotations if any node becomes unbalanced.

2. Deletion:

- **a.** Delete the node as in a standard BST.
- **b.** Update the height of the ancestor nodes.
- **c.** Rebalance the tree by performing the necessary rotations if any node becomes unbalanced.

3. Search:

- **a.** Perform the search operation as in a standard BST.
- **b.** Time Complexity: O(logn) due to the balanced nature of the AVL tree.

LABORATORY TASK 11

Q1. Build an AVL tree with the following values:

15, 20, 24, 10, 13, 7, 30, 36, 25

CONCLUSION:

_____

_____

_____

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.12

Graph

OBJECTIVE:

Understand the concepts if graphs

Implementation of graphs

PLO	CLO	LL
5	2	P-3

Theory Overview

Breadth-First Search (BFS)

Concept: BFS is a graph traversal algorithm that explores the nodes level by level. It starts from a given source node and explores all its neighbors before moving on to the neighbors' neighbors.

Steps:

1. Initialize a queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty:
    - Dequeue a node from the queue.
    - For each unvisited neighbor of the dequeued node:
        - Mark the neighbor as visited.
        - Enqueue the neighbor.

Key Points:

- **Queue**: Uses a queue to keep track of nodes to be explored.
- **Level-wise Traversal**: Explores nodes level by level, ensuring that all nodes at the current depth are processed before moving to the next depth level.
- **Time Complexity**: O(V+E)O(V + E)O(V+E), where VVV is the number of vertices and EEE is the number of edges.
- **Space Complexity**: O(V)O(V)O(V), for storing the queue and the visited list.

Applications:

- Finding the shortest path in unweighted graphs.
- Level-order traversal of a tree.
- Solving problems like finding connected components in a graph.

Advantages:

- Guarantees finding the shortest path in unweighted graphs.
- Simple to implement.

Disadvantages:

- Requires additional memory for the queue and visited list.
- May explore a large number of nodes in graphs with high branching factors.

Depth-First Search (DFS)

Concept: DFS is a graph traversal algorithm that explores as far down a branch as possible before backtracking. It starts from a given source node and explores each branch to its end before backtracking to explore other branches.

Steps:

1. Initialize a stack (or use recursion) and push the starting node.
2. Mark the starting node as visited.
3. While the stack is not empty:
    - Pop a node from the stack.
    - For each unvisited neighbor of the popped node:
        - Mark the neighbor as visited.
        - Push the neighbor onto the stack.

Key Points:

- **Stack/Recursion**: Uses a stack or recursion to keep track of nodes to be explored.

- **Depth-wise Traversal**: Explores nodes along a branch as deep as possible before backtracking.
- **Time Complexity**: O(V+E)O(V + E)O(V+E), where VVV is the number of vertices and EEE is the number of edges.
- **Space Complexity**: O(V)O(V)O(V) in the worst case for the stack or recursion call stack.

Applications:

- Detecting cycles in a graph.
- Topological sorting of a Directed Acyclic Graph (DAG).
- Finding connected components in a graph.
- Solving puzzles and problems like mazes and games where backtracking is required.

Advantages:

- Requires less memory than BFS in sparse graphs.
- Can be more efficient than BFS in deep graphs with fewer branches.

Disadvantages:

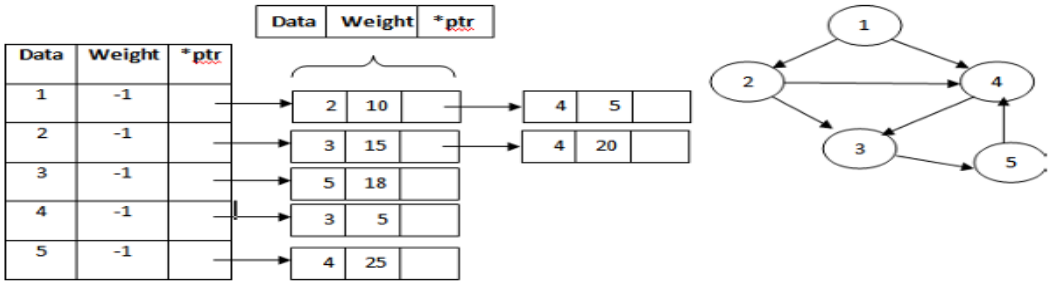
- Does not guarantee the shortest path in unweighted graphs.
- May get stuck in deep or infinite loops without proper checks.

LABORATORY TASK 12

Write a function called DelAllMinEdges( ). This function will first search all the edges in the graph and will delete all those edges having minimum weight. Consider that graph is implemented using Adjacency List. Consider the following structure for both Vertices and Edges

```
StructGNode
{
      int weight;
      GNode *ptr;
}*GList;
```

Note: In the following example DelAllMinEdges( ) will delete the edge between vertex 4 to 3 and 1to 4 because of minimum weight.



CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.13

GRAPH- MST

OBJECTIVE:

Understand the concepts and implementation of MST

PLO	CLO	LL
5	2	P-3

Theory Overview

Spanning Tree: A spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph.
Minimum Spanning Tree (MST):A minimum spanning tree is a subgraph of an undirected weighted graph G, such that
• it is a tree (i.e., it is acyclic)
• it covers all the vertices V – contains |V| - 1 edges
• the total cost associated with tree edges is the minimum among all possible spanning trees Kruskal'sAlgorithm:Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.
Algorithm:
• create a forest F (a set of trees), where each vertex in the graph is a separate tree as well
• create a set S containing all the edges in the graph
• while S is nonempty – remove an edge with minimum weight from S – if that edge connects two different trees, then add it to the forest, combining two trees into a single tree – otherwise discard that edge
• In the subsequent example, a green edge means it has been included into the forest of trees that will ultimately form a single, minimum spanning tree.

How to Generate a MST?

LABORATORY TASK 13

Q1. Write a program, which reads an adjacency matrix representation of a graph and applies Kruskal's algorithm to find minimum spanning tree of the input graph.

CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		

PRACTICAL NO.14

DIJKSTRA ALGO

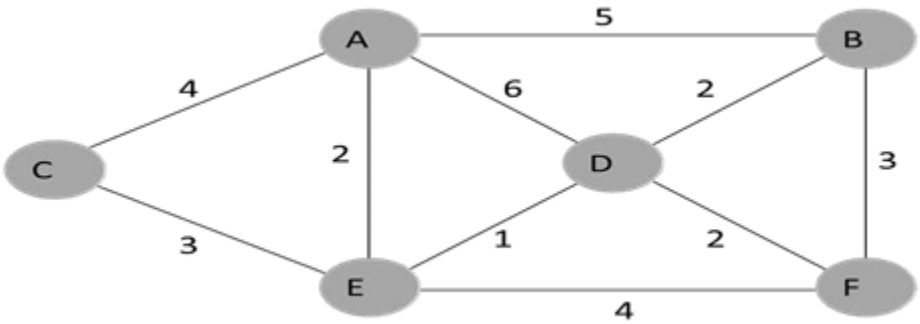
OBJECTIVE:

Find shortest path in Graph using Dijkstra Algo.

PLO	CLO	LL
5	2	P-3

Theory Overview

Dijkstra Algorithm - Shortest Path Calculation: This algorithm is used to find the shortest path between nodes. The shortest path may be computed based on number of hops, the distance, bandwidth available, queue lengths etc. It is widely used in Dynamic routing algorithm.



Dijkstra Algorithm

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes except the initial node.

3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances.

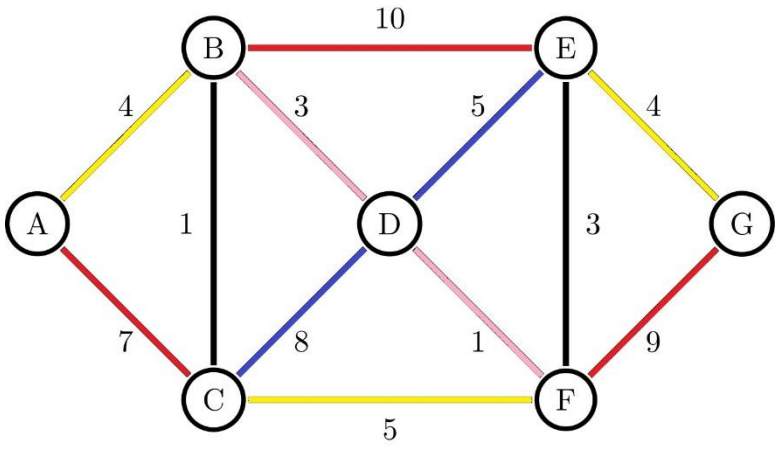
4. When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again; its distance recorded now is final and minimal.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal), then stop. The algorithm has finished.

6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.

LABORATORY TASK 14

Q1. Consider the following graph and calculate the shortest path using Dijkstra Algorithm from **A** to **G**. Show all the steps involved in the calculations.



CONCLUSION:

RUBRICS:

Lab Performance			Lab Report		
Description	**Total Marks**	**Marks Obtained**	**Description**	**Total Marks**	**Marks Obtained**
Completeness and Accuracy	**3**		**Demonstration**	**3**	
Coding Standards	**3**		**Efficiency**	**3**	
Total Marks obtained			**Total Marks Obtained**		