Data Structure 1-9 Lab Notes:

=>Prepared by Shaheer Ali

: _____ :TABLE OF CONTENTS: _____ :

Lab #1: (Programming Principles)
Basic Input & Output, Loops, Arrays.Structure, Functions, Pointers.
Lab #2 (continuation to Lab # 1): Programming Principles:
Overview of C++, and programs to demonstrate C++ Classes, Member function, Inheritance templates.
Lab #3 Linked List lab
List Processing and pointers to structure.
Lab #4 Double Linked List lab
List Processing and pointers to structure.
Lab #5 Stack Lab Using Array
Stack – ADT and application.
Lab #6 Stack Lab Using Linked list
Continuation to the Lab # 6 – and implementation of Stack – ADT using Linked List.
Lab #7 Queues Lab Using Linked list
Implementation of QUEUE– ADT and Application.
Lab #8 Algorithms Lab: Searching
Implementation of searching algorithms(Linear search , Binary search).
Lab #9 Algorithms Lab: Sorting
Implementation of sorting algorithms (Bubble sort, Selection sort.)

Lab No 1:

Certainly! Here's a more specific lab outline for C++:

Lab #1: Programming Principles in C++

1. Basic Input & Output

- **Objective:** Understand how to take input and display output in C++.
- **Example:**

```
#include <iostream>
using namespace std;

int main() {
```

```
    // Input
    int num;
    cout << "Enter a number: ";
    cin >> num;

    // Output
    cout << "You entered: " << num << endl;

    return 0;
}
```

2. Loops

- **Objective:** Learn the use of loops for repetitive tasks in C++.
- **Example:**

```
#include <iostream>
using namespace std;

int main() {
    // While Loop
    int i = 1;
    while (i <= 5) {
        cout << i << endl;
        i++;
    }

    // For Loop
    for (int j = 1; j <= 5; j++) {
        cout << j << endl;
    }

    return 0;
}
```

3. Arrays

- **Objective:** Understand how to work with arrays in C++.
- **Example:**

```
#include <iostream>
using namespace std;

int main() {
    // Declare and initialize an array
    int numbers[] = {1, 2, 3, 4, 5};
```

```
    // Accessing elements
    cout << "Element at index 2: " << numbers[1] << endl;

    // Modifying elements
    numbers[3] = 10;

    // Iterating through the array
    for (int k = 0; k < 5; k++) {
        cout << numbers[k] << endl;
    }

    return 0;
}
```

4. Structure

- **Objective:** Explore the concept of structures in C++.
- **Example:**

```
#include <iostream>
using namespace std;

// Structure
struct Point {
    int x;
    int y;
};

int main() {
    // Create an instance of Point
    Point p1;
    p1.x = 10;
    p1.y = 20;

    // Accessing structure members
    cout << "Coordinates: (" << p1.x << ", " << p1.y << ")" << endl;

    return 0;
}
```

5. Functions

- **Objective:** Learn how to define and use functions in C++.
- **Example:**

```
#include <iostream>
using namespace std;
```

```
// Function declaration
int add(int a, int b) {
    return a + b;
}

int main() {
    // Function usage
    int result = add(5, 3);
    cout << "Sum: " << result << endl;

    return 0;
}
```

6. Pointers

- **Objective:** Introduce the concept of pointers in C++.
- **Example:**

```
#include <iostream>
using namespace std;

int main() {
    int num = 5;
    int *ptr = &num;

    // Accessing value through pointer
    cout << "Value of num: " << *ptr << endl;

    // Modifying value through pointer
    *ptr = 10;
    cout << "Modified value of num: " << num << endl;

    return 0;
}
```

Feel free to modify and expand on these examples based on the specific requirements and goals of your lab. Students can work on these exercises to gain hands-on experience with basic input and output, loops, arrays, structures, functions, and pointers in C++.

Lab No 2:

Certainly! Let's break down each term used in the question and provide detailed explanations along with example programs.

Overview of C++:

C++ is a general-purpose programming language that was developed as an extension of the C programming language. It supports both procedural and object-oriented programming paradigms. Here's a basic structure of a C++ program:

```
#include <iostream>

int main() {
    // Your code here
    return 0;
}
```

- `#include <iostream>`: This is a preprocessor directive that includes the input/output stream library, allowing you to use functions like `cout` and `cin` for input and output.
- `int main()`: This is the main function where the execution of the program begins.

C++ Classes:

A class is a user-defined data type that encapsulates data and functions operating on that data. Here's a simple example:

```
#include <iostream>

// Class declaration
class Rectangle {
public:
    // Member variables
    double length;
    double width;

    // Member function to calculate area
    double calculateArea() {
        return length * width;
    }
};

int main() {
    // Creating an object of the class
    Rectangle myRectangle;

    // Setting values for the object
    myRectangle.length = 5.0;
    myRectangle.width = 3.0;

    // Using a member function
    double area = myRectangle.calculateArea();

    // Displaying the result
    cout << "Area of the rectangle: " << area << std::endl;
```

```
    return 0;
}
```

- `class Rectangle`: This declares a class named `Rectangle`.
- `public`: This keyword specifies the access level for the members of the class. Members declared as `public` are accessible from outside the class.

Member Functions in C++:

Member functions are functions defined inside a class that operate on class objects. Here's an example:

```
#include <iostream>

// Class declaration
class Rectangle {
private:
    double length;
    double width;

public:
    // Member function to set values
    void setDimensions(double len, double wid) {
        length = len;
        width = wid;
    }

    // Member function to calculate area
    double calculateArea() {
        return length * width;
    }

    // Member function to display information
    void displayInfo() {
        cout << "Length: " << length << ", Width: " << width << std::endl;
    }
};

int main() {
    // Creating an object of the class
    Rectangle myRectangle;

    // Using member function to set values
    myRectangle.setDimensions(5.0, 3.0);

    // Using member function to calculate area
    double area = myRectangle.calculateArea();

    // Using member function to display information
    myRectangle.displayInfo();
```

```
    // Displaying the result
    cout << "Area of the rectangle: " << area << std::endl;

    return 0;
}
```

- `void setDimensions(double len, double wid)`: This is a member function that sets the values of `length` and `width` for a `Rectangle` object.
- `double calculateArea()`: This member function calculates the area of the rectangle.
- `void displayInfo()`: This member function displays information about the rectangle.

Inheritance in C++:

Inheritance is a mechanism in C++ that allows a class to inherit properties and behaviors from another class. Here's an example:

```
#include <iostream>

// Base class
class Shape {
public:
    double area;
    void displayArea() {
        std::cout << "Area: " << area << std::endl;
    }
};

// Derived class
class Circle : public Shape {
public:
    double radius;
    void calculateArea() {
        area = 3.14 * radius * radius;
    }
};

int main() {
    // Creating an object of the derived class
    Circle myCircle;

    // Setting values for the object
    myCircle.radius = 4.0;

    // Calculating the area
    myCircle.calculateArea();

    // Displaying the result
    myCircle.displayArea();
```

```
    return 0;
}
```

- `class Shape`: This is the base class with a member variable `area` and a member function `displayArea`.
- `class Circle : public Shape`: This is a derived class that inherits from the `Shape` class. The `public` keyword indicates that the derived class has public access to the base class.
- `void calculateArea()`: This member function in the derived class calculates the area of the circle.

Templates in C++:

Templates allow you to write generic code that works with any data type. Here's an example of a template function:

```
#include <iostream>

// Template function to find the maximum of two values
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    // Using the template function with different data types
    cout << "Maximum of 5 and 8: " << findMax(5, 8) << std::endl;
    cout << "Maximum of 3.5 and 1.2: " << findMax(3.5, 1.2) << std::endl;

    return 0;
}
```

- `template <typename T>`: This declares a template with a type parameter `T`.
- `T findMax(T a, T b)`: This is a template function that finds the maximum of two values of type `T`.

These examples provide a detailed explanation and demonstration of C++ classes, member functions, inheritance, and templates. Feel free to ask if you have any specific questions or if you'd like further clarification on any topic!

Lab No 3:

Certainly! Let's delve into the details of list processing and pointers to structures in C++. We'll discuss what a linked list is, how it is implemented, and how pointers to structures are used in this context.

Linked List:

A linked list is a data structure that consists of a sequence of elements where each element points to the next element in the sequence. Unlike arrays, linked lists do not have a fixed size, and elements can be easily inserted or removed.

Node Structure:

In C++, a linked list is typically implemented using a structure (or a class) to represent each node in the list. Each node contains two parts:

1. Data: The actual data or value stored in the node.
2. Next: A pointer to the next node in the sequence.

Let's define a simple structure for a linked list node:

```
#include <iostream>

// Node structure
struct Node {
    int data;
    Node* next;
};
```

List Processing with Pointers to Structures:

Creating Nodes:

To create nodes dynamically in C++, we use the `new` keyword to allocate memory for a node and return a pointer to it.

```
// Creating nodes dynamically
Node* createNode(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr; // Initialize the next pointer to null
    return newNode;
}
```

Building a Linked List:

We can create a linked list by linking nodes together. The last node's `next` pointer is set to `nullptr` to indicate the end of the list.

```
// Building a linked list
Node* addToFront(Node* head, int value) {
    Node* newNode = createNode(value);
    newNode->next = head;
```

```
    return newNode;
}

// Displaying the linked list
void displayList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << std::endl;
}
```

Deleting Nodes:

Deleting nodes involves deallocating the memory used by the node and updating the pointers of adjacent nodes.

```
// Deleting a linked list
void deleteList(Node*& head) {
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

Example Usage:

```
int main() {
    // Creating an empty linked list
    Node* myList = nullptr;

    // Adding elements to the front of the list
    myList = addToFront(myList, 3);
    myList = addToFront(myList, 7);
    myList = addToFront(myList, 1);

    // Displaying the linked list
    displayList(myList);

    // Deleting the linked list to free memory
    deleteList(myList);

    return 0;
}
```

In this example, we've created a simple linked list, added elements to the front, displayed the list, and then deleted the list to free the allocated memory.

This covers the basics of linked lists and how pointers to structures are used in C++ for list processing. Feel free to ask if you have more specific questions or if you'd like additional details on any aspect!

Lab No 4:

Certainly! Let's delve into the details of list processing and pointers to structures for a doubly linked list in C++. A doubly linked list is a type of linked list in which each node contains a data element and two pointers, one pointing to the next node and another pointing to the previous node.

Doubly Linked List:

In a doubly linked list, each node has the following structure:

```
#include <iostream>

// Node structure for doubly linked list
struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

- `data`: Holds the actual data or value stored in the node.
- `next`: Points to the next node in the sequence.
- `prev`: Points to the previous node in the sequence.

List Processing with Pointers to Structures:

Creating Nodes:

Creating nodes dynamically involves allocating memory for a node and initializing its `data`, `next`, and `prev` fields.

```
// Creating nodes dynamically
Node* createNode(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr; // Initialize the next pointer to null
    newNode->prev = nullptr; // Initialize the prev pointer to null
    return newNode;
}
```

Building a Doubly Linked List:

Building a doubly linked list involves linking nodes together in both forward and backward directions. The last node's `next` pointer is set to `nullptr` to indicate the end of the list.

```
// Adding a node to the front of the list
Node* addToFront(Node* head, int value) {
    Node* newNode = createNode(value);
    newNode->next = head;
    if (head != nullptr) {
        head->prev = newNode; // Update the previous pointer of the existing head
    }
    return newNode;
}

// Displaying the doubly linked list forward
void displayListForward(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << std::endl;
}

// Displaying the doubly linked list backward
void displayListBackward(Node* tail) {
    Node* current = tail;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->prev;
    }
    cout << std::endl;
}
```

Deleting Nodes:

Deleting nodes in a doubly linked list involves updating the pointers of adjacent nodes and deallocating memory.

```
// Deleting a node from the doubly linked list
void deleteNode(Node*& head, Node*& tail, Node* nodeToDelete) {
    if (nodeToDelete == nullptr) {
        return; // Invalid node to delete
    }

    if (nodeToDelete->prev != nullptr) {
        nodeToDelete->prev->next = nodeToDelete->next;
    } else {
        head = nodeToDelete->next; // Update head if the node to delete is the
first node
    }
```

```
    if (nodeToDelete->next != nullptr) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    } else {
        tail = nodeToDelete->prev; // Update tail if the node to delete is the
last node
    }

    delete nodeToDelete; // Deallocate memory
}
```

Example Usage:

```
int main() {
    // Creating an empty doubly linked list
    Node* myListHead = nullptr;
    Node* myListTail = nullptr;

    // Adding elements to the front of the list
    myListHead = addToFront(myListHead, 3);
    myListHead = addToFront(myListHead, 7);
    myListHead = addToFront(myListHead, 1);

    // Displaying the doubly linked list forward
    displayListForward(myListHead);

    // Displaying the doubly linked list backward
    displayListBackward(myListTail);

    // Deleting a node from the list
    Node* nodeToDelete = myListHead->next;
    deleteNode(myListHead, myListTail, nodeToDelete);

    // Displaying the doubly linked list forward after deletion
    displayListForward(myListHead);

    // Displaying the doubly linked list backward after deletion
    displayListBackward(myListTail);

    return 0;
}
```

This example covers the basics of list processing with pointers to structures for a doubly linked list in C++. Feel free to ask if you have more specific questions or if you'd like additional details on any aspect!

Lab No 5:

Sure, let's go into details about implementing a Stack using an array in C++, covering the Stack ADT (Abstract Data Type) and its applications.

Stack ADT:

A stack is a Last-In-First-Out (LIFO) data structure that follows the principle of "last in, first out," meaning the last element added is the first one to be removed. It supports two main operations:

1. **Push**: Adds an element to the top of the stack.
2. **Pop**: Removes the element from the top of the stack.

Additional operations may include:

- **Top/Peek**: Retrieve the element from the top without removing it.
- **isEmpty**: Checks if the stack is empty.
- **isFull**: Checks if the stack is full (in the case of a fixed-size array implementation).
- **Size**: Returns the number of elements in the stack.

Stack Implementation Using Array:

Let's implement a simple stack using an array in C++.

```
#include <iostream>

class Stack {
private:
    static const int MAX_SIZE = 100; // Maximum size of the stack
    int arr[MAX_SIZE];
    int top; // Index of the top element

public:
    // Constructor
    Stack() : top(-1) {}

    // Push operation
    void push(int value) {
        if (top == MAX_SIZE - 1) {
            cout << "Stack overflow! Cannot push " << value << ".\n";
            return;
        }
        arr[++top] = value;
        cout << value << " pushed to the stack.\n";
    }

    // Pop operation
    void pop() {
        if (isEmpty()) {
            cout << "Stack underflow! Cannot pop from an empty stack.\n";
            return;
        }
        cout << arr[top--] << " popped from the stack.\n";
    }
```

```
    // Top/Peek operation
    int peek() const {
        if (isEmpty()) {
            cout << "The stack is empty.\n";
            return -1; // Return a sentinel value
        }
        return arr[top];
    }

    // Check if the stack is empty
    bool isEmpty() const {
        return top == -1;
    }

    // Check if the stack is full
    bool isFull() const {
        return top == MAX_SIZE - 1;
    }

    // Get the size of the stack
    int size() const {
        return top + 1;
    }
};
```

Example Usage:

```
int main() {
    Stack myStack;

    myStack.push(10);
    myStack.push(20);
    myStack.push(30);

    cout << "Top element: " << myStack.peek() << "\n";
    cout << "Stack size: " << myStack.size() << "\n";

    myStack.pop();
    myStack.pop();

    std::cout << "Is the stack empty? " << (myStack.isEmpty() ? "Yes" : "No") <<
"\n";

    return 0;
}
```

This example demonstrates the basic operations of a stack: push, pop, peek, isEmpty, and size.

Applications of Stack:

1. **Function Call Management (Call Stack):** Used to manage function calls and returns in programs.
2. **Expression Evaluation:** Infix to Postfix conversion and evaluation.
3. **Undo Mechanisms:** Used in applications where an "undo" feature is required.
4. **Backtracking:** Used in algorithms that involve backtracking, such as the depth-first search algorithm.
5. **Memory Management:** Used in managing memory during the execution of a program.

Implementing a stack using an array provides a simple and efficient way to manage data with a Last-In-First-Out (LIFO) structure. The example above covers the basics, and you can extend it based on your specific requirements or application needs. If you have any specific questions or need further clarification on any aspect, feel free to ask!

Lab No 6:

Certainly! Let's continue the implementation of a Stack using a linked list in C++. This involves defining a Node structure, creating a Stack class, and implementing the basic stack operations.

Stack ADT Using Linked List:

Node Structure:

```
#include <iostream>

// Node structure for the linked list
struct Node {
    int data;
    Node* next;

    // Constructor
    Node(int val) : data(val), next(nullptr) {}
};
```

Stack Implementation:

```
class Stack {
private:
    Node* top; // Pointer to the top of the stack

public:
    // Constructor
    Stack() : top(nullptr) {}

    // Destructor to deallocate memory
    ~Stack() {
        while (!isEmpty()) {
```

```
            pop();
        }
    }

    // Push operation
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
        cout << value << " pushed to the stack.\n";
    }

    // Pop operation
    void pop() {
        if (isEmpty()) {
            cout << "Stack underflow! Cannot pop from an empty stack.\n";
            return;
        }
        Node* temp = top;
        top = top->next;
        cout << temp->data << " popped from the stack.\n";
        delete temp;
    }

    // Top/Peek operation
    int peek() const {
        if (isEmpty()) {
            cout << "The stack is empty.\n";
            return -1; // Return a sentinel value
        }
        return top->data;
    }

    // Check if the stack is empty
    bool isEmpty() const {
        return top == nullptr;
    }

    // Get the size of the stack
    int size() const {
        int count = 0;
        Node* current = top;
        while (current != nullptr) {
            count++;
            current = current->next;
        }
        return count;
    }
};
```

Example Usage:

```
int main() {
    Stack myStack;

    myStack.push(10);
    myStack.push(20);
    myStack.push(30);
cout << "Top element: " << myStack.peek() << "\n";cout << "Stack size: " <<
myStack.size() << "\n";

    myStack.pop();
    myStack.pop();

    cout << "Is the stack empty? " << (myStack.isEmpty() ? "Yes" : "No") << "\n";

    return 0;
}
```

This example demonstrates the basic operations of a stack using a linked list: push, pop, peek, isEmpty, and size. The linked list implementation provides dynamic memory management, making it suitable for situations where the size of the stack may vary during runtime.

Applications of Stack Using Linked List:

1. **Undo Mechanisms:** Used in applications where an "undo" feature is required.
2. **Function Call Management (Call Stack):** Used to manage function calls and returns in programs.
3. **Expression Evaluation:** Infix to Postfix conversion and evaluation.
4. **Backtracking:** Used in algorithms that involve backtracking, such as the depth-first search algorithm.
5. **Memory Management:** Used in managing memory during the execution of a program.

Using a linked list for the stack allows for efficient memory allocation and deallocation as elements are pushed and popped. The destructor in the Stack class takes care of freeing memory when the stack goes out of scope.

Feel free to ask if you have any specific questions or if you'd like additional details on any aspect!

Lab No 7:

Certainly! Let's go through the implementation of a Queue using a linked list in C++. We'll cover the Queue ADT (Abstract Data Type) and its applications.

Queue ADT Using Linked List:

Node Structure:

```
#include <iostream>
```

```
// Node structure for the linked list
struct Node {
    int data;
    Node* next;

    // Constructor
    Node(int val) : data(val), next(nullptr) {}
};
```

Queue Implementation:

```
class Queue {
private:
    Node* front; // Front of the queue
    Node* rear;  // Rear of the queue

public:
    // Constructor
    Queue() : front(nullptr), rear(nullptr) {}

    // Destructor to deallocate memory
    ~Queue() {
        while (!isEmpty()) {
            dequeue();
        }
    }

    // Enqueue operation
    void enqueue(int value) {
        Node* newNode = new Node(value);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        cout << value << " enqueued to the queue.\n";
    }

    // Dequeue operation
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue underflow! Cannot dequeue from an empty queue.\n";
            return;
        }
        Node* temp = front;
        front = front->next;
        if (front == nullptr) {
            rear = nullptr; // Queue becomes empty after dequeue
        }
        cout << temp->data << " dequeued from the queue.\n";
```

```
        delete temp;
    }

    // Front operation
    int getFront() const {
        if (isEmpty()) {
            cout << "The queue is empty.\n";
            return -1; // Return a sentinel value
        }
        return front->data;
    }

    // Check if the queue is empty
    bool isEmpty() const {
        return front == nullptr;
    }

    // Get the size of the queue
    int size() const {
        int count = 0;
        Node* current = front;
        while (current != nullptr) {
            count++;
            current = current->next;
        }
        return count;
    }
};
```

Example Usage:

```
int main() {
    Queue myQueue;

    myQueue.enqueue(10);
    myQueue.enqueue(20);
    myQueue.enqueue(30);

    cout << "Front element: " << myQueue.getFront() << "\n";
    cout << "Queue size: " << myQueue.size() << "\n";

    myQueue.dequeue();
    myQueue.dequeue();

    cout << "Is the queue empty? " << (myQueue.isEmpty() ? "Yes" : "No") << "\n";

    return 0;
}
```

This example demonstrates the basic operations of a queue using a linked list: enqueue, dequeue, getFront, isEmpty, and size. The linked list implementation allows for dynamic memory management, making it suitable for situations where the size of the queue may vary during runtime.

Applications of Queue Using Linked List:

1. **Breadth-First Search (BFS):** Used in graph algorithms for traversing or searching graph structures.
2. **Job Scheduling:** Queues are used to manage tasks in systems where tasks are processed in the order they arrive.
3. **Print Queue:** Used in printers to manage the order of print jobs.
4. **Buffer Management:** Queues are used to manage data in a first-in, first-out (FIFO) manner.
5. **Task Management in Operating Systems:** Queues are used to manage tasks or processes in an operating system.

Using a linked list for the queue allows for efficient memory allocation and deallocation as elements are enqueued and dequeued. The destructor in the `Queue` class takes care of freeing memory when the queue goes out of scope.

Feel free to ask if you have any specific questions or if you'd like additional details on any aspect!

Lab No 8:

Certainly! Let's discuss the implementation of two commonly used searching algorithms in C++: Linear Search and Binary Search.

Linear Search:

Linear Search is a simple search algorithm that looks for a target value in a list and returns its position if found, or -1 if not found. It works by iterating through each element in the list until the target value is found or the end of the list is reached.

Implementation:

```
#include <iostream>
#include <vector>

// Linear Search function
int linearSearch(const std::vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) {
            return i; // Element found, return its index
        }
    }
    return -1; // Element not found
}
```

```
int mainvector<int> array = {10, 20, 30, 40, 50};

    int target = 30;
    int result = linearSearch(array, target);

    if (result != -1cout << "Element " << target << " found at index " << result
<< std::endl;
    } else {
        cout << "Element " << target << " not found in the array." << std::endl;
    }

    return 0;
}
```

Binary Search:

Binary Search is a more efficient search algorithm but requires a sorted list. It works by repeatedly dividing the search range in half. If the value to be searched is equal to the middle element, the search is successful. If the value is less than the middle element, the search continues in the lower half; otherwise, it continues in the upper half.

Implementation:

```
#include <iostream>
#include <vector>

// Binary Search function
int binarySearch(const std::vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Element found, return its index
        } else if (arr[mid] < target) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }

    return -1; // Element not found
}

int main() {
    vector<int> array = {10, 20, 30, 40, 50};

    int target = 30;
    int result = binarySearch(array, target);
```

```
    if (result != -1) {
        cout << "Element " << target << " found at index " << result << std::endl;
    } else {
        cout << "Element " << target << " not found in the array." << std::endl;
    }

    return 0;
}
```

In both implementations, a vector is used as the data structure, and the target element is searched within it. The `linearSearch` function performs a linear search, while the `binarySearch` function performs a binary search. It's important to note that for binary search, the array must be sorted.

These algorithms demonstrate basic searching techniques. Depending on the context and the characteristics of the data, other searching algorithms might be more suitable. If you have specific questions or need further clarification, feel free to ask!

Lab No 9:

Certainly! Let's go through the implementation of two simple sorting algorithms in C++: Bubble Sort and Selection Sort.

Bubble Sort:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Implementation:

```
#include <iostream>
#include <vector>

// Bubble Sort function
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    vector<int> array = {64, 34, 25, 12, 22, 11, 90};
```

```
    cout << "Original array: ";
    for (int num : array) {
        cout << num << " ";
    }
    cout << std::endl;

    bubbleSort(array);

    cout << "Sorted array: ";
    for (int num : array) {
        cout << num << " ";
    }
    cout << std::endl;

    return 0;
}
```

Selection Sort:

Selection Sort is a simple sorting algorithm that divides the input into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and swaps it with the first element of the unsorted region.

Implementation:

```
#include <iostream>
#include <vector>

// Selection Sort function
void selectionSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;

        // Find the index of the minimum element in the unsorted region
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with the first element
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    using namespace std; // Use the namespace std
```

```
    vector<int> array = {64, 25, 12, 22, 11};

    cout << "Original array: ";
    for (int num : array) {
        cout << num << " ";
    }
    cout << endl;

    selectionSort(array);

    cout << "Sorted array: ";
    for (int num : array) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

In both implementations, a vector is used as the data structure to be sorted. The sorting functions, `bubbleSort` and `selectionSort`, modify the input vector to produce a sorted result.

These algorithms provide a basic understanding of sorting techniques. Depending on the context and the characteristics of the data, other sorting algorithms might be more suitable. If you have specific questions or need further clarification, feel free to ask!