



DATA STRUCTURE AND ALGORITHM

LAB MANUAL

LAB NO. 1-12

Submitted to:
MAM IRSHA QURESHI

Submitted by:
2023-BS-AI-062

Table of Contents

Lab 1: Introduction	3
Lab 2: Array	6
Lab 3: 2d Array	9
Lab 4: Vector	14
Lab 5: Single Link List.....	19
Lab 6: Double Link list	24
Lab 7: Circular Link list	36
Lab 8: Stacks.....	40
Lab 9: Queues	45
Lab 10: DeQueues.....	48
Lab 11: Binary Trees.....	51
Lab 12: BST	55

LAB MANUAL

LAB 1

Introduction to Data Structures and Algorithms (DSA) in C++

Data Structures and Algorithms (DSA) are fundamental concepts in computer science that help developers design efficient and optimized solutions to problems. DSA is crucial in competitive programming, system design, and software development.

C++ is an excellent language for learning and implementing DSA due to its speed, rich standard library, and low-level features. Here's an introduction to DSA in C++:

1. What are Data Structures?

Data Structures are ways to organize and store data in a computer so that it can be accessed and modified efficiently. Common data structures include:

- **Arrays:** A collection of elements stored in contiguous memory.
 - **Linked Lists:** A sequence of nodes, where each node points to the next.
 - **Stacks:** A collection of elements with Last In, First Out (LIFO) behavior.
 - **Queues:** A collection of elements with First In, First Out (FIFO) behavior.
 - **Trees:** A hierarchical data structure with nodes connected by edges.
 - **Graphs:** A set of nodes (vertices) connected by edges.
 - **Hash Tables:** A data structure that stores data in a key-value pair format for fast access.
-

2. What are Algorithms?

Algorithms are step-by-step procedures or formulas for solving problems. Some common algorithmic topics include:

- **Sorting Algorithms:** Bubble Sort, Quick Sort, Merge Sort, etc.
- **Searching Algorithms:** Binary Search, Linear Search.
- **Dynamic Programming:** Solving problems by breaking them down into simpler subproblems.
- **Greedy Algorithms:** Making the best choice at each step to find the global optimum.
- **Backtracking:** Exploring all possible solutions and reverting when a solution is not feasible.

3. Why C++ for DSA?

- **Standard Template Library (STL):** C++ provides STL, which includes commonly used data structures like vectors, sets, maps, and algorithms such as `sort`, `binary_search`, etc.
- **Efficiency:** C++ offers better performance due to its low-level access to memory.
- **Community and Resources:** A large community and abundant learning resources are available for DSA in C++.

1. Linear Data Structures

A **linear data structure** is one in which the elements are arranged in a sequential manner, where each element has a unique predecessor and successor (except for the first and last elements).

Characteristics:

- Elements are stored in a contiguous or sequential manner.
- Traversal is simple and follows a single level (one after another).
- Suitable for tasks that involve sequential processing of data.

Examples of Linear Data Structures:

1. **Array:**
 - Fixed-size collection of elements of the same data type.
 - Example: `int arr[] = {1, 2, 3, 4};`
 - Memory layout: Contiguous.
2. **Linked List:**
 - A collection of nodes where each node contains data and a pointer to the next node.
 - Example: Singly Linked List, Doubly Linked List.
3. **Stack:**
 - Follows Last In First Out (LIFO) principle.
 - Example: Undo operations in editors.
4. **Queue:**
 - Follows First In First Out (FIFO) principle.
 - Example: Printer queue.

Advantages:

- Simpler implementation and traversal.
- Easy to understand and use.

Disadvantages:

- May not always be the most efficient for complex relationships between elements.

2. Non-Linear Data Structures

A **non-linear data structure** is one in which the elements are not arranged sequentially. Instead, they are arranged in a hierarchical or networked manner, where an element can be connected to multiple elements.

Characteristics:

- Data elements are connected in a non-sequential manner.
- Traversal may involve multiple paths.
- Suitable for representing hierarchical or interconnected relationships.

Examples of Non-Linear Data Structures:

1. **Tree:**
 - A hierarchical structure with nodes connected by edges.
 - Example: Binary Tree, Binary Search Tree, AVL Tree, Heap.
 - Applications: File systems, XML/HTML parsing.
2. **Graph:**
 - A collection of nodes (vertices) and edges.
 - Can be directed or undirected, weighted or unweighted.
 - Example: Social networks, navigation systems.
3. **Hash Table:**
 - Data is stored in a key-value pair format, often implemented using arrays.
 - Example: Dictionaries, caches.

Advantages:

- Efficient representation of complex relationships.
- More flexible data organization.

Disadvantages:

- Implementation and traversal can be more complex.
- May require more memory compared to linear structures.

LAB 2

Topic: Arrays


Definition: An array in C++ is a collection of elements of the same data type stored in contiguous memory locations. Each element can be accessed using an index. Arrays provide a way to store multiple values of the same type in a single variable, making it easier to manage large data sets.

Syntax:

```
dataType arrayName[arraySize];
```

Q1 Simple

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string groccerry[4] = {"colagate", "headnshoulder", "scotchbright", "surf"};
    cout<<groccerry[1]<<endl;
    return 0;
}
```



headnshoulder

Q2 Using loop to print all

```
#include <iostream>
using namespace std;
int main() {
    int marks[4] = {10,20,60,106};
    for(int i=0;i<4;i++){
        cout<<marks[i]<<endl;
```

```

}
return 0;

```

```

10
20
60
106
}

```

Q3 Reverse an Array

```

#include <iostream>
using namespace std;
int main() {
    int n=8;
    int originalArray[n]={1,2,3,4,5,6,7,8};
    int reversedArray[n];
    for (int i = 0; i < n; ++i) {
        reversedArray[i] = originalArray[n - 1 - i];
    }
    cout << "Original Array: ";
    for (int i = 0; i < n; ++i) {
        cout << originalArray[i] << " ";
    }
    cout << endl;
    cout << "Reversed Array: ";
    for (int i = 0; i < n; ++i) {
        cout << reversedArray[i] << " ";
    }
    cout << endl;
    return 0;}

```

```

Enter size: 5
Enter elements: 2 3 4 6 7
Maximum number is: 7
Minimum number is: 2

```

Output

Q4 Count characters in an array

```

#include <iostream>
using namespace std;

int main() {
    char arr[10];
    char check;
    cout<<"Enter 10 characters: ";
    for(int i = 0; i < 10; i++) {
        cin >> arr[i];
    }
    cout<<"Enter character to count: ";
    cin>>check;
    int count = 0;

```

```

Enter 10 characters: a b c d e f a a b b
Enter character to count: a
Occurrences of a: 3

```

```

for(int i = 0; i < 10; i++) {
    if(arr[i] == check) count++;
}
cout << "Occurrences of " << check << ": " << count << endl;
return 0;
}

```

Output

Q5 Find duplicates in an array

```

#include <iostream>
using namespace std;
int main() {
    int arr[10], NewArr[10], Count = 0;
    cout << "Enter 10 integers: ";
    for (int i = 0; i < 10; i++) {
        cin >> arr[i];
        bool Duplicate = false;
        // Check if arr[i] is already in NewArr
        for (int j = 0; j < Count; j++) {
            if (arr[i] == NewArr[j]) {
                Duplicate = true;
                break;
            }
        }
        // If not a duplicate, add to NewArr
        if (!Duplicate) {
            NewArr[Count] = arr[i];
            Count++;
        }
    }
    cout << "Array without duplicates: ";
    for (int i = 0; i < Count; i++) {
        cout << NewArr[i] << " ";
    }
    cout << endl;

    return 0;
}

```

```

Enter 10 integers: 1 2 3 3 4 5 6 7 9 9
Array without duplicates: 1 2 3 4 5 6 7 9

```


LAB 3

2D Arrays in C++

A **2D array** is a collection of elements arranged in a grid or matrix format, consisting of rows and columns. It is essentially an array of arrays.

Declaration

```
data_type array_name[rows][columns];
```

Q1 2d array

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int arr[3][4] = {  
        {1, 2, 3, 4},  
        {5, 6, 7, 8},  
        {9, 10, 11, 12}  
    };  
    for (int i = 0; i < 3; i++) {  
        for (int j = 0; j < 4; j++) {  
            cout << arr[i][j] << " ";  
        }  
        cout << endl;  
    }  
  
    return 0;  
}
```

```
1 2 3 4
5 6 7 8
9 10 11 12
```

Q2

```
#include <iostream>

using namespace std;

int main() {

    int rows = 3, cols = 4;

    // Allocate memory for rows
    int** matrix = new int*[rows];

    // Allocate memory for columns
    for (int i = 0; i < rows; i++) {
        matrix[i] = new int[cols];
    }

    // Initialize the array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = i + j;
        }
    }

    // Print the matrix
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}
```

```

    }

    // Free the allocated memory
    for (int i = 0; i < rows; i++) {
        delete[] matrix[i];
    }
    delete[] matrix;

    return 0;
}

```

Output

Q3 Find the order of matrix in a 2d array

```

#include<iostream>
using namespace std;
int main(){
    int n,m,order;
    cout<<"Enter rows: ";
    cin>>n;
    cout<<"Enter coloumns: ";
    cin>>m;
    int arr[n] [m];
    cout<<"Enter elements: ";
    for(int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
            cin>> arr[i][j];
        }
    }

    cout<<"Matrix is: "<<endl;
    for(int i=0;i<n;i++) {
        for (int j=0; j<m;j++) {
            cout<<arr[i][j]<<" ";
        }
        cout<<"\n";
    }

    order= n*m;
    cout<<"Order of matrix is: "<<order;
}

```

```

Enter rows: 2
Enter coloumns: 2
Enter elements: 1 2 3 6
Matrix is:
1 2
3 6
Order of matrix is: 4

```

Output

Q4 Find the sum of matrices

```
#include<iostream>
using namespace std;
int main() {
    int n, m;
    cout << "Enter rows: ";
    cin >> n;
    cout << "Enter columns: ";
    cin >> m;
    int a[n][m]; int b[n][m]; int c[n][m];
    cout << "Enter elements of 1st matrix: ";
    for(int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> a[i][j];}
        cout << "1st Matrix is: " << endl;
        for(int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cout << a[i][j] << " ";
            }
            cout << "\n";}
        cout << "Enter elements of 2nd matrix: ";
        for(int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cin >> b[i][j];}
            cout << "2nd Matrix is: " << endl;
            for(int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    cout << b[i][j] << " ";
                }
                cout << "\n";}
            cout << "Total of Matrices is: " << endl;
            for(int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    c[i][j] = a[i][j] + b[i][j];
                    cout << c[i][j] << " ";
                }
                cout << "\n";}
            return 0;
        }
    }
```

```
Enter rows: 2
Enter columns: 2
Enter elements of 1st matrix: 1 2 3 4
1st Matrix is:
1 2
3 4
Enter elements of 2nd matrix: 1 2 3 4
2nd Matrix is:
1 2
3 4
Total of Matrices is:
2 4
6 8
```

Q5 Find Average in a 2d array

```
#include <iostream>
using namespace std;
int main() {
    int array[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
};
```

Output

```
The average of the 2D array elements is:5
```

```
int rows = 3, cols = 3;
int sum = 0;
int count = 0;
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        sum += array[i][j];
        count++;
    }
}
float average = (float)sum / count;
cout<<"The average of the 2D array elements is:"<<average;
return 0;
}
```

LAB 4

Vector

Definition: A vector in C++ is a dynamic array provided by the Standard Template Library (STL). Unlike regular arrays, vectors can dynamically resize themselves as elements are added or removed. They are part of the <vector> header.

Basic Syntax

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

int main() {
    vector<int> vec; // Declaring a vector of integers
    vec.push_back(10); // Adding an element
    vec.push_back(20); // Adding another element
    cout << "Size: " << vec.size() << endl; // Displaying size
    return 0;
}
```

Q1

```
#include<iostream>

#include<vector>

using namespace std;

int main(){

    vector <string>students={"Faisal","Hashir","Haseeb","Tayyab","Hanzla"};

    for(int i=0;i<5;i++){

        cout<<students[i]<<endl;

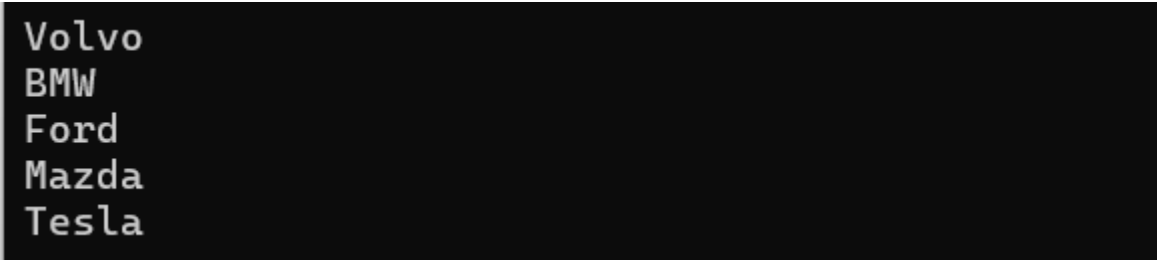
    }

}
```

```
Faisal
Hashir
Haseeb
Tayyab
Hanzla
```

Q2

```
#include <iostream> // Includes the input/output stream library for using cout
#include <string>    // Includes the string library for using the string type
using namespace std; // Allows us to avoid using the std:: prefix before
                    // standard library objects like cout
int main() {
    // Declare an array of strings named 'cars' with 5 elements
    string cars[5];
    // Assign values to each element of the 'cars' array
    cars[0] = "Volvo";
    cars[1] = "BMW";
    cars[2] = "Ford";
    cars[3] = "Mazda";
    cars[4] = "Tesla";
    // Loop through each element in the 'cars' array
    // The loop runs from index 0 to 4 (less than 5)
    for(int i = 0; i < 5; i++) {
        // Output each element in the 'cars' array followed by a newline character
        cout << cars[i] << "\n";
    }
    return 0; // Return 0 indicates that the program ended successfully
}
```



```
Volvo
BMW
Ford
Mazda
Tesla
```

Q3

```

#include <iostream>

using namespace std;

int main() {
    int myNumbers[5] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++) {
        cout << myNumbers[i] << "\n";
    }
    return 0;
}

```

```

10
20
30
40
50

```

Q4

```

#include <iostream>

using namespace std;

int main() {
    int numbers[5] = {10, 20, 30, 40, 50}; // Initializing an array of size 5
    for (int i = 0; i < 5; i++) {
        cout << "Element at index " << i << " is: " << numbers[i] << endl;
    }
    return 0;
}

```

```

Element at index 0 is: 10
Element at index 1 is: 20
Element at index 2 is: 30
Element at index 3 is: 40
Element at index 4 is: 50

```

Q5

```

#include <iostream>

using namespace std;

```



```

int main() {
    int arr[5], largest;
    cout << "Enter 5 numbers:" << endl;
    for (int i = 0; i < 5; i++) {
        cin >> arr[i];
    }
    largest = arr[0];
    for (int i = 1; i < 5; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }
    cout << "The largest number is: " << largest << endl;
    return 0;
}

```

```

Enter 5 numbers:
1
2
3
4
5
The largest number is: 5

```

Q6 front and back function

```

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

int main() {
    list<string> fruits = {"Apple", "Banana", "Cherry"};
    fruits.push_back("h1");
}

```

```
    fruits.push_front('h');  
    for (const string fruit : fruits) {  
        cout << fruit << "\n";  
    }  
  
    return 0;  
}
```

```
my2  
Apple  
Banana  
Cherry  
my
```

LAB 5

Single Linked List in C++

Definition: A **single linked list** is a linear data structure where each element, called a **node**, contains two parts:

1. **Data:** The value stored in the node.
2. **Pointer:** A reference (or pointer) to the next node in the sequence.

The last node in the list has a NULL pointer to indicate the end of the list.

Structure of a Node

In C++, a node in a single linked list can be represented using a struct or class.

```
cpp
Copy code
struct Node {
    int data;    // Data part
    Node* next;  // Pointer to the next node

    Node(int value) { // Constructor to initialize a new node
        data = value;
        next = NULL;
    }
};
```

Q6 Insertion at end Linked list

```
#include <iostream>

using namespace std;

int main (){

    int arr[10]={1,2,3,4,5};

    int size=5;

    int newvalue=6;

    cout<<"before insertion:"<<endl;

    for(int i=0;i<size;i++){

        cout << arr[i];
```

```

    }
    arr[size]=newvalue;
    size++;
    cout << "after insertion:"<<endl;
    for(int i=0;i<size;i++){
        cout << arr[i] ;
    }
    cout<<endl;
}

```

```

before insertion:
1
2
3
4
5
after insertion:
1
2
3
4
5
6

```

Q7 Delete at end LL

```
#include <iostream>
```

```
using namespace std;
```

```

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = 5;

    cout << "Before deletion: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
}

```

```

    }
    cout << endl;

    size--;

    cout << "After deletion: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}

```

```

Before deletion: 10 20 30 40 50
After deletion: 10 20 30 40

```

Q8 Delete at start

```

#include <iostream>
using namespace std;

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = 5;
    cout << "Before deletion: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < size - 1; i++) {

```

```

        arr[i] = arr[i + 1];
    }
    size--;
    cout << "After deletion: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

```

```

Before deletion: 10 20 30 40 50
After deletion: 20 30 40 50
}

```

Q8

```

#include <iostream>
#include <string>
using namespace std;
int main() {
    string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
    for (int i = 0; i < 5; i++)
    { cout << i << " = " << cars[i] << "\n";
    }
    return 0;
}

```

```

0 = Volvo
1 = BMW
2 = Ford
3 = Mazda
4 = Tesla

```

Q9

```
#include <iostream> // Includes the input/output stream library for using cout
using namespace std; // Allows us to avoid using the std:: prefix before
//standard library objects like cout

int main() {
    // Declare and initialize an array of integers named 'myNumbers' with 5
    //elements
    int myNumbers[5] = {10, 20, 30, 40, 50};
    // Loop through each element in the 'myNumbers' array
    // The loop runs from index 0 to 4 (less than 5)
    for (int i = 0; i < 5; i++) {
        // Output each element in the 'myNumbers' array followed by a newline
        //character
        cout << myNumbers[i] << "\n";
    }
    return 0; // Return 0 indicates that the program ended successfully
}
```

```
10
20
30
40
50
}
```

LAB 6

Double Linked List

A **doubly linked list** is a type of linked data structure that consists of nodes, where each node contains three components:

1. **Data:** The value stored in the node.
2. **Next:** A pointer/reference to the next node in the sequence.
3. **Prev:** A pointer/reference to the previous node in the sequence.

Definition: A Linked List is a linear data structure where elements, called nodes, are connected using pointers. Each node contains two parts.

- Singly link list: A singly link list is the link where each node has a single pointer to the next node. It Traverses is one-directional and the last node's pointer is nullptr.

Q1: INSERTION At ANY POINT

```
#include<iostream>

using namespace std;

class node{
    public:
    int data;
    node *link;
    node *head;
    node *tail;
    node *current;
    node (){
        head = nullptr;
        tail = nullptr;
    }
    void create(int data){
        if(head == nullptr){
            node *n = new node();
```



```

    n->data = data;
    n->link = nullptr;
    head = tail = n;
}
else{
    node *n = new node();
    n->data = data;
    n->link = nullptr;
    tail->link=n;
    tail = n;
}
}

void insertAtBegin(int data) {
    node *n = new node();
    n->data = data;
    n->link = head;
    head = n;
}

void display(){
    current = head;
    while(current!=nullptr){
        cout<<current->data<<" ";
        current = current->link;
    }
}

};


int main(){
    node hashir;

```

```

    hashir.create(5);
    hashir.insertAtBegin(3);
    hashir.display();
}

```



```

3 5
PS C:\CODING\C++\practical\

```

Q2: deletion at start

```

#include<iostream>
using namespace std;
class node{
    public:
    int data;
    node *link;
    node *head;
    node *tail;
    node *current;
    node (){
        head = nullptr;
        tail = nullptr;
    }
    void create(int data){
        if(head == nullptr){
            node *n = new node();
            n->data = data;
            n->link = nullptr;
            head = tail = n;
        }
        else{
            node *n = new node();

```

```

    n->data = data;
    n->link = nullptr;
    tail->link=n;
    tail = n;
}
}

void deleteAtBegin() {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    node *temp = head;
    head = head->link;
    delete temp;
    if (head == nullptr) { // If the list is now empty, set tail to nullptr
        tail = nullptr;
    }
}

void display(){
    current = head;
    while(current!=nullptr){
        cout<<current->data<<" ";
        current = current->link;
    }
}

};

int main(){
    node hashir;

```

```

5
PS C:\CODING\C++\practi

```

```
hashir.create(5);  
hashir.display();  
}
```

Q3: insert at any point

```
#include<iostream>  
using namespace std;  
class node{  
    public:  
    int data;  
    node *link;  
    node *head;  
    node *tail;  
    node *current;  
    node (){  
        head = nullptr;  
        tail = nullptr;  
    }  
    void create(int data){  
        if(head == nullptr){  
            node *n = new node();  
            n->data = data;  
            n->link = nullptr;  
            head = tail = n;  
        }  
        else{  
            node *n = new node();  
            n->data = data;
```

```

    n->link = nullptr;
    tail->link=n;
    tail = n;
}
}

void insertAtPosition(int data, int position) {
    node *n = new node();
    n->data = data;

    if (position == 0) { // Insert at the beginning
        n->link = head;
        head = n;
        if (tail == nullptr) {
            tail = n;
        }
    } else {
        current = head;
        for (int i = 0; i < position - 1 && current != nullptr; ++i) {
            current = current->link;
        }
        if (current != nullptr) {
            n->link = current->link;
            current->link = n;
            if (n->link == nullptr) {
                tail = n;
            }
        } else {
            cout << "Position out of bounds" << endl;

```

```

    }
}
}
void display(){
    current = head;
    while(current!=nullptr){
        cout<<current->data<<" ";
        current = current->link;
    }
}
};
int main(){
    node hashir;
    hashir.create(5);
    hashir.create(5);
    hashir.create(5);
    hashir.insertAtPosition(2,2);
    hashir.display();
}

```

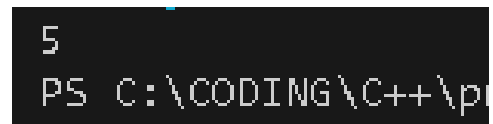
Q4: delete at any point

```

#include<iostream>
using namespace std;

class node {
public:
    int data;
    node *link;
    node *head;

```



```

5
PS C:\CODING\C++\p

```

```
node *tail;
```

```
node *current;
```

```
node() {  
    head = nullptr;  
    tail = nullptr;  
}
```

```
void create(int data) {  
    node *n = new node();  
    n->data = data;  
    n->link = nullptr;  
    if (head == nullptr) {  
        head = tail = n;  
    } else {  
        tail->link = n;  
        tail = n;  
    }  
}
```

```
void deleteAtBegin() {  
    if (head == nullptr) {  
        cout << "List is empty." << endl;  
        return;  
    }  
    node *temp = head;  
    head = head->link;  
    if (head == nullptr) { // List had only one node, so update tail
```

```

        tail = nullptr;
    }
    delete temp;
}

void deleteAtPosition(int position) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    if (position == 0) { // Special case for deleting the head
        deleteAtBegin();
        return;
    }

    current = head;
    for (int i = 0; i < position - 1 && current->link != nullptr; ++i) {
        current = current->link;
    }

    if (current->link == nullptr) {
        cout << "Position out of bounds." << endl;
    } else {
        node *temp = current->link;
        current->link = temp->link;
        if (temp->link == nullptr) { // Update tail if we're deleting the last node
            tail = current;
        }
    }
}

```



```

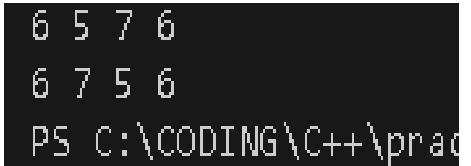
        delete temp;
    }
}

void display() {
    current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->link;
    }
    cout << endl;
}

};

int main() {
    node hashir;
    hashir.create(5);
    hashir.create(10);
    hashir.create(15);
    cout << "Original list: ";
    hashir.display();
    hashir.deleteAtPosition(2);
    cout << "After deleting at position 2: ";
    hashir.display();
    hashir.deleteAtPosition(0);
    cout << "After deleting at position 0: ";
    hashir.display();
    return 0;
}

```



```

6 5 7 6
6 7 5 6
PS C:\CODING\C++\prac

```

```
}
```

Q5: insert at last

```
#include<iostream>
```

```
using namespace std;
```

```
class node{
```

```
    public:
```

```
    int data;
```

```
    node *link;
```

```
    node *head;
```

```
    node *tail;
```

```
    node *current;
```

```
    node (){
```

```
        head = nullptr;
```

```
        tail = nullptr;
```

```
    }
```

```
    void create(int data){
```

```
        if(head == nullptr){
```

```
            node *n = new node();
```

```
            n->data = data;
```

```
            n->link = nullptr;
```

```
            head = tail = n;
```

```
        }
```

```
        else{
```

```
            node *n = new node();
```

```
            n->data = data;
```

```
            n->link = nullptr;
```

```
            tail->link=n;
```

```
            tail = n;
```

```

    }
}

void insertAtEnd(int data) {
    node *n = new node();
    n->data = data;
    n->link = nullptr;
    if (head == nullptr) {
        head = tail = n;
    } else {
        tail->link = n;
        tail = n;
    }
}

void display(){
    current = head;
    while(current!=nullptr){
        cout<<current->data<<" ";
        current = current->link;
    }
}

};

int main(){
    node hashir;
    hashir.create(5);
    hashir.insertAtEnd(6);
    hashir.display();
}

```

```

6 5 7 6
6 7 5 6
PS C:\CODING\C++\pra

```

Lab 7: Circular Link List

Characteristics

- The last node points to the first node.
- Can be singly or doubly linked.
- Enables circular traversal.

- Example Programs

Output

1. Node Structure

```
struct CNode {
```

```
    int data;
```

```
    CNode* next;
```

```
    CNode(int val) : data(val), next(nullptr) {}
```

```
};
```

2. Insertion at end

```
void insertEnd(CNode*& head, int val) {
```

```
    CNode* newNode = new CNode(val);
```

```
    if (head == nullptr) {
```

```
        head = newNode;
```

```
        newNode->next = head;
```

```
        return;
```

```
    }
```

```
    CNode* temp = head;
```

```
    while (temp->next != head) {
```

```
        temp = temp->next;
```

```
    }
```

```
10 20 30
```

```
10 20 30
```

```
Process exited after 0.006958 seconds with return value 0
```

```

temp->next = newNode;
newNode->next = head;
}

```

3. Deletion at value

```

void deleteNode(CNode*& head, int val) {
    if (head == nullptr) return;

    if (head->data == val && head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

```

```

    CNode* temp = head;
    CNode* prev = nullptr;
    do {
        if (temp->data == val) break;
        prev = temp;
        temp = temp->next;
    } while (temp != head);

    if (temp == head && temp->data != val) return;

    if (temp == head) {
        prev = head;
        while (prev->next != head) prev = prev->next;
        head = head->next;
        prev->next = head;
    }

```

Output

```

10 20 30
10 30

```

```

-----
Process exited after 0.062 seconds with return value 0

```

```

    } else {
        prev->next = temp->next;
    }
    delete temp;
} DoublyNode* temp = head;
while (temp != nullptr && temp->data != val) {
    temp = temp->next;
}
if (temp == nullptr) return;
if (temp->next != nullptr) temp->next->prev = temp->prev;
if (temp->prev != nullptr) temp->prev->next = temp->next;
delete temp;
}

```

4. Display the list

```

void display(CNode* head) {
    if (head == nullptr) return;
    CNode* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(head)" << endl;
}

```

Output

```

10 20 30
-----
Process exited after 0.06958 seconds with return value 0

```

5. Insert at position

```

void insertAtPosition(int data, int position) {
    node *n = new node();
    n->data = data;
}

```

Output

```

10 20 30
-----
Process exited after 0.06958 seconds with return value 0
Press any key to continue . . . |

```

```

if (position == 0) { // Insert at the beginning
    n->link = head;
    head = n;
    if (tail == nullptr) {
        tail = n;
    }
} else {
    current = head;
    for (int i = 0; i < position - 1 && current != nullptr; ++i) {
        current = current->link;
    }
    if (current != nullptr) {
        n->link = current->link;
        current->link = n;
        if (n->link == nullptr) {
            tail = n;
        }
    } else {
        cout << "Position out of bounds" << endl;

```

LAB 8

STACK

A **stack** is a data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added to the stack will be the first one to be removed. In C++, you can implement a stack using arrays or linked lists, but the **Standard Template Library (STL)** provides a built-in stack container.

Here's a breakdown of the stack in C++:

Stack in C++ (using STL)

The C++ Standard Library provides the `stack` class, which allows you to perform basic operations like:

- **push**: Adds an element to the top.
- **pop**: Removes the element from the top.
- **top**: Returns the top element without removing it.
- **empty**: Checks if the stack is empty.
- **size**: Returns the size of the stack.

Q1

```
#include<iostream>

#include<string>

#include<stack>

using namespace std;

int main(){

    stack<int>marks;

    marks.push(87);

    marks.push(95);

    marks.push(76);

    marks.push(88);

    marks.pop();
```



```

marks.pop();

marks.pop();

marks.pop();

cout<<"Size of stack is: "<<marks.size()<<"\n";

if(marks.empty()==true){

    cout<<"\nStack is empty";

}

else

    cout<<"\nStack is not empty";

```

```

Size of stack is: 0
Stack is empty
}

```

Q2

```

#include<iostream>

#include<string>

#include<stack>

using namespace std;

int main(){

    stack<string>sports;

    sports.push("Cricket");

    sports.push("Football");

    sports.push("Tennis");

    sports.push("Hockey");

```

```

sports.push("Squash");

sports.push("Basketball");

sports.pop();

sports.pop();

cout<<"Size of stack is: "<<sports.size()<<"\n";

cout<<"Top element of stack is: "<<sports.top();

if(sports.empty()==true){

    cout<<"\nStack is empty";

}

else

    cout<<"\nStack is not empty";

```

```

}
Size of stack is: 4
Top element of stack is: Hockey
Stack is not empty

```

Q3

```

#include<iostream>

#include<string>

#include<stack>

using namespace std;

int main(){

    stack<string>universities;

    universities.push("TUF");

```

```

universities.push("FAST");
universities.push("NUST");
universities.push("LUMS");
universities.push("NTU");
universities.push("Riphah");
universities.push("BNU");
universities.pop();
universities.pop();
universities.pop();

cout<<"Size of stack is: "<<universities.size()<<"\n";
cout<<"Top element in stack is: "<<universities.top();

if(universities.empty()==true){

    cout<<"\nStack is empty";

}

else

    cout<<"\nStack is not empty";

```

```

}
Size of stack is: 4
Top element in stack is: LUMS
Stack is not empty

```

Q4

```

#include<iostream>

#include<string>

#include<stack>

```

```
using namespace std;

int main(){

    stack<string>bikes;

    bikes.push("CD70");

    bikes.push("YBZ");

    bikes.push("Suzuki");

    bikes.push("Yamaha");

    bikes.pop();

    bikes.pop();

    cout<<"Size of stack is: "<<bikes.size()<<"\n";

    cout<<"Top element in stack is: "<<bikes.top();

    if(bikes.empty()==true){

        cout<<"\nStack is empty";

    }

    else

        cout<<"\nStack is not empty";

}
```

```
Size of stack is: 2
Top element in stack is: YBZ
Stack is not empty
```

LAB 9

QUEUE

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means that elements are added to the rear (end) of the queue and removed from the front. Queues are commonly used in scenarios where order needs to be preserved, such as task scheduling, buffering, or resource sharing.

Basic Operations in a Queue

1. **Enqueue**: Add an element to the rear of the queue.
2. **Dequeue**: Remove and return the element from the front of the queue.
3. **Peek/Front**: Retrieve the element at the front without removing it.
4. **IsEmpty**: Check if the queue is empty.
5. **IsFull** (optional): Check if the queue is full (for bounded queues).

Codes

Q1

```
#include<iostream>

#include<string>

#include<queue>

using namespace std;

int main(){

    queue<string>colleges;

    colleges.push("KIPS");

    colleges.push("TIPS");

    colleges.push("IVY");

    colleges.push("Alley");

    cout<<"First Element of queue is: "<<colleges.front()<<"\n";

    cout<<"Last Element of queue is: "<<colleges.back()<<"\n";
```

```

colleges.front()="PGC";

colleges.back()="NIXOR";

cout<<"New First Element of queue is: "<<colleges.front()<<"\n";

cout<<"New Back Element of queue is: "<<colleges.back()<<"\n";

colleges.pop();

cout<<"After popping the new front element is: "<<colleges.front();

if(colleges.empty()==true){

    cout<<"\nQueue is empty";

}

else

    cout<<"\nQueue is not empty";

```

```

First Element of queue is: KIPS
Last Element of queue is: Alley
New First Element of queue is: PGC
New Back Element of queue is: NIXOR
After popping the new front element is: TIPS
Queue is not empty
}

```

Q2

```

#include<iostream>

#include<string>

#include<queue>

using namespace std;

```

```

int main(){

    queue<string>schools;

    schools.push("Allied");

    schools.push("City School");

    schools.push("Lyceum");

    cout<<"First Element of queue is: "<<schools.front()<<"\n";

    cout<<"Last Element of queue is: "<<schools.back()<<"\n";

    schools.front()="Beacon House";

    schools.back()="Generations";

    cout<<"New First Element of queue is: "<<schools.front()<<"\n";

    cout<<"New Back Element of queue is: "<<schools.back()<<"\n";

    schools.pop();

    cout<<"After popping the new front element is: "<<schools.front()<<"\n";

    cout<<"Size of queue is: "<<schools.size();

    if(schools.empty()==true){

        cout<<"\nQueue is empty";

    }

    else

        cout<<"\nQueue is not empty";

}

```

```

First Element of queue is: Allied
Last Element of queue is: Lyceum
New First Element of queue is: Beacon House
New Back Element of queue is: Generations
After popping the new front element is: City School
Size of queue is: 2
Queue is not empty

```

LAB 10

DEQUEUE

A **deque** (or **deque**, short for **double-ended queue**) is a data structure that allows elements to be added or removed from both ends — **front** and **rear**. This flexibility makes it a hybrid of a stack and a queue.

Basic Operations in a Dequeue

1. **Insert at the front:** Add an element at the front.
2. **Insert at the rear:** Add an element at the rear.
3. **Delete from the front:** Remove an element from the front.
4. **Delete from the rear:** Remove an element from the rear.
5. **Peek at the front/rear:** Access elements at either end without removing them.
6. **IsEmpty:** Check if the deque is empty.
7. **IsFull** (optional): Check if the deque is full (for bounded dequeues).

Q1

```
#include <iostream>
```

```
#include <deque>
```

```
using namespace std;
```

```
int main() {
```

```
    deque<int> dq;
```

```
    dq.push_back(10);
```

```
    dq.push_back(20);
```

```
    dq.push_back(30);
```

```
    dq.push_back(40);
```



```

dq.push_back(50);


dq.pop_front();

cout << "Remaining elements: ";
for (int elem : dq) {
    cout << elem << " ";
}

cout << endl;

return 0;
}

```



Remaining elements: 20 30 40 50

Q2

```

#include<iostream>

#include<deque>

using namespace std;

int main(){

```

```
deque<int>numbers;

numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
numbers.push_back(40);
numbers.push_back(50);

cout<<numbers.front()<<" ";

cout<<numbers.back()<<" ";

} 10 50
```

LAB 11

Binary Tree

A **Binary Tree** is a hierarchical data structure where each node has at most two children, referred to as the **left child** and the **right child**.

Properties of a Binary Tree

1. Node Structure:

- Each node consists of:
 - **Data:** The value stored in the node.
 - **Left Child:** A pointer/reference to the left child node.
 - **Right Child:** A pointer/reference to the right child node.

2. Types of Binary Trees:

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Perfect Binary Tree:** All internal nodes have two children, and all leaf nodes are at the same level.
- **Complete Binary Tree:** All levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
- **Skewed Binary Tree:** All nodes have only one child (either left-skewed or right-skewed).
- **Balanced Binary Tree:** The height difference between the left and right subtrees of any node is at most 1.

3. Traversal Methods:

- **Inorder Traversal:** Left → Root → Right.
- **Preorder Traversal:** Root → Left → Right.
- **Postorder Traversal:** Left → Right → Root.
- **Level Order Traversal:** Nodes are visited level by level.

All Traversals

```
#include <iostream>
```

```
#include <queue>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```

Node* right;

Node(int value) {

    data = value;

    left = nullptr;

    right = nullptr;

}

};

void inorderTraversal(Node* root) {

    if (root == nullptr) return;

    inorderTraversal(root->left);

    cout << root->data << " ";

    inorderTraversal(root->right);

}

void preorderTraversal(Node* root) {

    if (root == nullptr) return;

    cout << root->data << " ";

    preorderTraversal(root->left);

    preorderTraversal(root->right);

}

void postorderTraversal(Node* root) {

    if (root == nullptr) return;

```

```

    postorderTraversal(root->left);

    postorderTraversal(root->right);

    cout << root->data << " ";

}

void levelOrderTraversal(Node* root) {

    if (root == nullptr) return;

    queue<Node*> q;

    q.push(root);

    while (!q.empty()) {

        Node* current = q.front();

        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr) q.push(current->left);

        if (current->right != nullptr) q.push(current->right);

    }

}

int main() {

    // Create the binary tree

    Node* root = new Node(1);

```

```

root->left = new Node(2);
root->right = new Node(3);
root->left->left = new Node(4);
root->left->right = new Node(5);
cout << "Inorder Traversal: ";
inorderTraversal(root);
cout << endl;
cout << "Preorder Traversal: ";
preorderTraversal(root);
cout << endl;
cout << "Postorder Traversal: ";
postorderTraversal(root);
cout << endl;
cout << "Level Order Traversal: ";
levelOrderTraversal(root);
cout << endl;
return 0;

```

```

Inorder Traversal: 4 2 5 1 3
Preorder Traversal: 1 2 4 5 3
Postorder Traversal: 4 5 2 3 1
Level Order Traversal: 1 2 3 4 5
}

```

LAB 12

Binary Search Tree

A **Binary Search Tree (BST)** is a binary tree with the following properties:

1. **Binary Tree:** Each node can have at most two children: left and right.
2. **BST Property:**
 - For every node N :
 - All nodes in the left subtree of N contain values less than N .
 - All nodes in the right subtree of N contain values greater than N .
3. **No Duplicates:** Typically, a BST does not allow duplicate values.

Key Operations in a BST

1. **Insertion:** Add a new node to the BST while maintaining its properties.
2. **Search:** Check if a value exists in the BST.
3. **Deletion:** Remove a node from the BST while maintaining its properties.
4. **Traversal:**
 - **Inorder Traversal:** Left \rightarrow Root \rightarrow Right (results in sorted order for BST).
 - **Preorder Traversal:** Root \rightarrow Left \rightarrow Right.
 - **Postorder Traversal:** Left \rightarrow Right \rightarrow Root.

Q1 Simple

```
#include <iostream>
```

```
using namespace std;
```

```
struct TreeNode {
```

```
    int data;
```

```
    TreeNode *left;
```

```
    TreeNode *right;
```

```
TreeNode(int value) {  
    data = value;  
    left = nullptr;  
    right = nullptr;  
}  
};
```

```
TreeNode* insertNode(TreeNode* root, int value) {  
    if (!root) {  
        return new TreeNode(value);  
    }  
    if (value < root->data) {  
        root->left = insertNode(root->left, value);  
    } else {  
        root->right = insertNode(root->right, value);  
    }  
    return root;  
}
```



```

void inOrder(TreeNode* root) {
    if (root) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}

```

```

int main() {
    TreeNode* root = nullptr;

    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);
}

```

```
cout << "In-order Traversal: ";
```

```
inOrder(root);
```

```
cout << endl;
```

```
return 0;
```

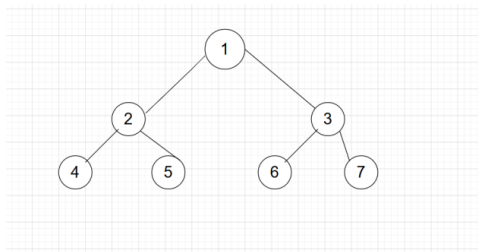
```
}
```

```
In-order Traversal: 20 30 40 50 60 70 80
```

Q2

TRAVERSE

PREORDER



```
#include <iostream>
```

```
using namespace std;
```

```
// Node Structure
```

```
struct Node {
```

```
    int data;    // value
```

```
    Node* left;  // left pointer
```

```
    Node* right; // right Pointer
```

```
    Node(int val) { // Constructor
```

```
        data = val;
```

```

    left = NULL;
    right = NULL;
}
};

void preorder(struct Node* root){
    if(root == NULL){
        return;
    }
    cout << root->data << " "; // visit root node
    preorder(root->left); // visit left subtree
    preorder(root->right); // visit right subtree
}

int main(){
    struct Node* root = new Node(1);
    root->left= new Node(2);
    root->right= new Node(3);
    root->left->left= new Node(4);
    root->left->right= new Node(5);
    root->right->left= new Node(6);
    root->right->right= new Node(7);
    preorder(root);
}

```

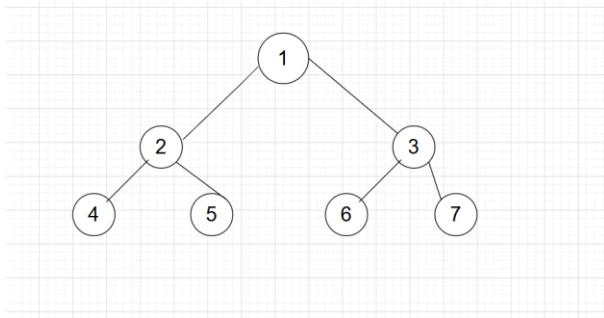
1 2 4 5 3 6 7

ALGORITHM

1. Start.
2. Node Structure.
3. Visit the root node.
4. Traverse the left subtree.
5. Traverse the right subtree.

6. End.

POST ORDER



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

```
// Node Structure
```

```
struct Node {
```

```
    int data;    // value
```

```
    Node* left;  // left pointer
```

```
    Node* right; // right Pointer
```

```
    Node(int val) { // Constructor
```

```
        data = val;
```

```
        left = NULL;
```

```
        right = NULL;
```

```
    }
```

```
};
```

```
void postorder(struct Node* root){
```

```
    if (root == NULL) {
```

```
        return;
```

```
    }
```

```
    postorder(root->left);    // visit left subtree
```

```
    postorder(root->right);    // visit right subtree
```

```

        cout << root->data << " "; // visit root node

    }

int main(){

    struct Node* root = new Node(1);

    root->left= new Node(2);

    root->right= new Node(3);

    root->left->left= new Node(4);

    root->left->right= new Node(5);

    root->right->left= new Node(6);

    root->right->right= new Node(7);

    postorder(root);

}

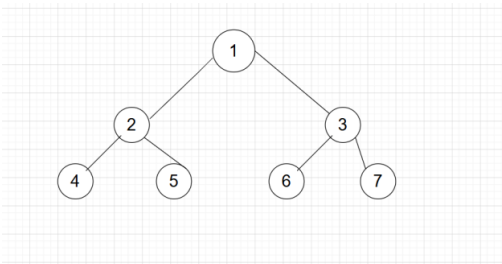
```

4 5 2 6 7 3 1

ALGORITHM

1. Start.
2. Node Structure.
3. Traverse the left subtree.
4. Traverse the right subtree.
5. Visit the root node.
6. End.

INORDER



```

#include <iostream>

using namespace std;

```

```

// Node Structure

struct Node {

    int data;      // value

    Node* left;    // left pointer

    Node* right;   // right Pointer

    Node(int val) { // Constructor

        data = val;

        left = NULL;

        right = NULL;

    }

};

void inorder(Node* root) {

    if (root == NULL) {

        return;

    }

    inorder(root->left);    // visit left subtree

    cout << root->data << " "; // visit root node

    inorder(root->right);   // visit right subtree

}

int main(){

    struct Node* root = new Node(1);

    root->left= new Node(2);

    root->right= new Node(3);

    root->left->left= new Node(4);

    root->left->right= new Node(5);

    root->right->left= new Node(6);

    root->right->right= new Node(7);

    inorder(root);

```

}

4 2 5 1 6 3 7