The University of Faisalabad

Lab Manual

- **Name:**

  Abdul Haseeb Arif.

- **Reg no.:**

  2023-BS-AI-033.

- **Submitted to:**

  Mam Irsha Qureshi

- **Subect:**

  Data Structures and Algorithms.

- **Department:**

  BS Artificial Intelligence. (Section-A)

- **Semester:**

  3rd

Table of Contents

Lab 1

- **Introduction:**

- **Data structures** in C++ are constructs that allow you to organize and store data in a way that enables efficient access and modification.

- **Common data structures** in C++ include **arrays**, **linked lists**, **stacks, queues**, **trees**, and **graphs**. Each data structure serves different purposes and has its own advantages and disadvantages.

- **Algorithms** in C++ are a set of instructions or procedures for performing a specific task or solving a problem. They can be implemented using various programming constructs and can range from simple operations like sorting and searching to more complex tasks like pathfinding in graphs. Algorithms are crucial for manipulating data stored in data structures.

➢ **Linear data structures** in C++ are those where elements are arranged in a sequential manner. Each element has a single predecessor and a single successor, making it easy to traverse. Examples include:

1. Arrays: A collection of elements identified by index.

2. Linked Lists: A series of nodes where each node contains data and a pointer to the next node.

➢ **Non-linear data structures** in C++ do not arrange elements in a sequential order, allowing for more complex relationships. Examples include:

1. Trees: A hierarchical structure with a root node and child nodes, like binary trees.

2. Graphs: A collection of nodes connected by edges, which can represent various relationships.

Lab 2

- **Array**

➢ **Definition:**

An array is a collection of elements of the same type, stored in contiguous memory locations. It allows you to store multiple values in a single variable, making data management more efficient.

➢ **Syntax of Array in C++:**

data_type array_name[array_size];

- data_type: The type of elements (e.g., int, float, char).

- array_name: The name of the array.

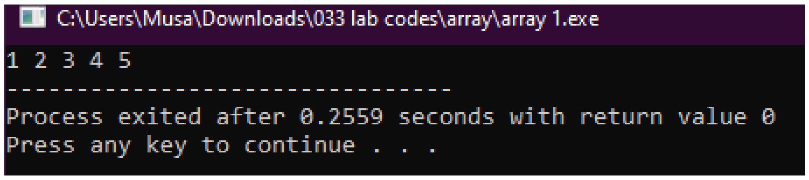
- array_size: The number of elements the array can hold.

➢ **Examples of Arrays in C++:**

▪ **Example 1: Initialize and Print an Integer Array**

```
#include <iostream>

using namespace std;

int main() {

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

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

        cout << numbers[i] << " ";

    }

    return 0;

}
```

- **Output:**

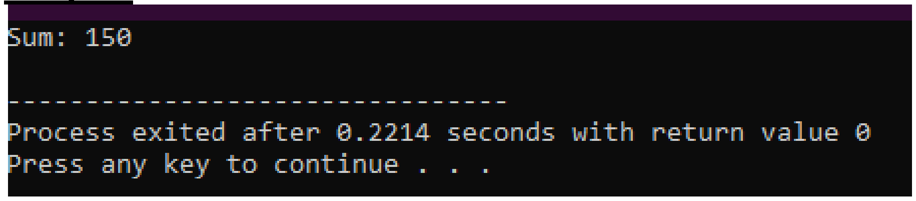


```
C:\Users\Musa\Downloads\033 lab codes\array\array 1.exe
1 2 3 4 5
--------------------------------
Process exited after 0.2559 seconds with return value 0
Press any key to continue . . .
```

- **Example 2: Calculate the Sum of Elements in an Array**

```
#include <iostream>

using namespace std;

int main() {

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

   int sum = 0;

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

      sum += numbers[i];

   }

   cout << "Sum: " << sum << endl;

   return 0;

}
```

- **Output:**



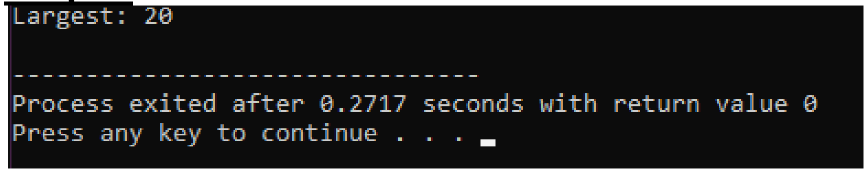
```
Sum: 150

--------------------------------
Process exited after 0.2214 seconds with return value 0
Press any key to continue . . .
```

- **Example 3: Find the Largest Element in an Array**

```
#include <iostream>

using namespace std;

int main() {

   int numbers[5] = {5, 12, 3, 8, 20};
```

```
    int largest = numbers[0];
    for (int i = 1; i < 5; i++) {
        if (numbers[i] > largest) {
            largest = numbers[i];
        }
    }
    cout << "Largest: " << largest << endl;
    return 0;
}
```

- **Output:**

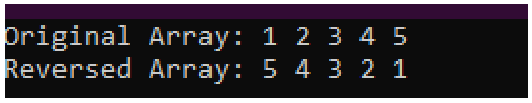


```
Largest: 20

--------------------------------
Process exited after 0.2717 seconds with return value 0
Press any key to continue . . .
```

- **Example 4: Reverse an Array**

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

- **<u>Output:</u>**

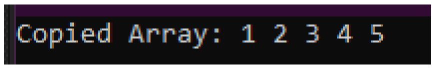


```
Original Array: 1 2 3 4 5
Reversed Array: 5 4 3 2 1
```

- **Example 5: Copying Elements from One Array to Another.**

```
#include <iostream>
using namespace std;
int main() {
    int original[5] = {1, 2, 3, 4, 5};
    int copy[5];
    for (int i = 0; i < 5; i++) {
        copy[i] = original[i];
    }
    cout << "Copied Array: ";
    for (int i = 0; i < 5; i++) {
        cout << copy[i] << " ";
    }
    return 0;
}
```

- **<u>Output:</u>**



```
Copied Array: 1 2 3 4 5
```

Lab 3

- **2d Array:**

- **Definition:**

A 2D array, or two-dimensional array, is a data structure that allows you to store data in a tabular format, consisting of rows and columns. Each element in a 2D array is accessed using two indices: one for the row and one for the column. This structure is useful for representing matrices, grids, or tables of data.

- **Syntax:**

The syntax for declaring a 2D array in C++ is as follows:

```
data_type array_name[row_size][column_size];
```

For example, to declare a 2D array of integers with 3 rows and 4 columns, you would write:

```
int myArray[3][4];
```

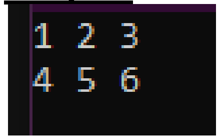
- **Examples**:

- **Example 1:Initializing a 2D Array and Displaying Elements:**

```
#include <iostream>
using namespace std;
int main() {
   int array[2][3] = {{1, 2, 3}, {4, 5, 6}};
   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 3; j++) {
         cout << array[i][j] << " ";
      }
      cout << endl;
   }
```

```
    return 0;
}
```

- **Output:**

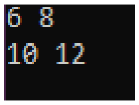


```
1 2 3
4 5 6
```

- **Example 2. Adding Two 2D Arrays:**

```
#include <iostream>
using namespace std;
int main() {
    int a[2][2] = {{1, 2}, {3, 4}};
    int b[2][2] = {{5, 6}, {7, 8}};
    int sum[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum[i][j] = a[i][j] + b[i][j];
        }
    }

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << sum[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

- **Output:**

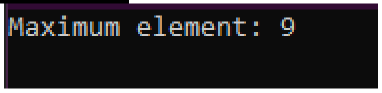


```
6 8
10 12
```

- **Example 3. Finding the Maximum Element in a 2D Array:**

```
#include <iostream>

using namespace std;

int main() {

    int array[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    int max = array[0][0];

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (array[i][j] > max) {

                max = array[i][j];

            }

        }

    }


    cout << "Maximum element: " << max << endl;

    return 0;

}
```

- **Output:**



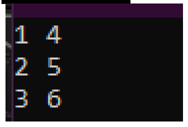
```
Maximum element: 9
```

- **Example 4. Transposing a 2D Array:**

```
#include <iostream>

using namespace std;
```

```
int main() {
    int array[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int transpose[3][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            transpose[j][i] = array[i][j];
        }
    }
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << transpose[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

- **Output:**

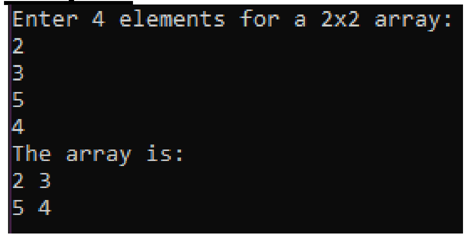


```
1 4
2 5
3 6
```

- **Example 5. Filling a 2D Array with User Input:**

```
#include <iostream>
using namespace std;
int main() {
    int array[2][2];
    cout << "Enter 4 elements for a 2x2 array:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cin >> array[i][j];
        }
    }

    cout << "The array is:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

- **Output:**



```
Enter 4 elements for a 2x2 array:
2
3
5
4
The array is:
2 3
5 4
```

Lab 4

- **Vector:**

➢ Definition:

A Vector in C++ is a dynamic array provided by the Standard Template Library (STL). It can grow and shrink in size, which makes it more flexible than a regular array. Vectors store elements in a contiguous memory location and allow for random access to elements, similar to arrays.

➢ Syntax of Vector:

To use a vector, you need to include the <vector> header and use the std::vector class. Here’s the basic syntax for defining a vector:

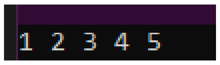
```
#include <vector>

std::vector<data_type> vector_name;

std::vector<int> vec(10); // Creates a vector of size 10 with default values (0)

std::vector<int> vec = {1, 2, 3, 4, 5}; // Initializes a vector with values
```

- **Examples:**

▪ Example 1:Basic Vector Operations:

```
#include <iostream>

#include <vector>

int main() {

   std::vector<int> vec = {1, 2, 3, 4, 5};

   // Displaying vector elements

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

      std::cout << vec[i] << " ";

   }

   return 0;

}
```

▪ Output:

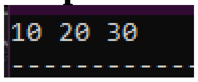


```
1 2 3 4 5
```

- **Example 2. Adding Elements to a Vector:**

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec;

    // Adding elements
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);

    // Displaying vector elements
    for(int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

- **Output:**

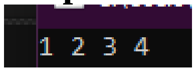


```
10 20 30
-----------
```

- **Example 3. Removing Elements from a Vector:**

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // Removing the last element
    vec.pop_back();

    // Displaying vector elements
    for(int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

- **Output:**

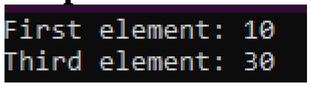


```
1 2 3 4
```

- **Example 4. Accessing Elements by Index:**

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};
    // Accessing elements
    std::cout << "First element: " << vec[0] << std::endl;
    std::cout << "Third element: " << vec[2] << std::endl;

    return 0;
}
```

- **Output:**

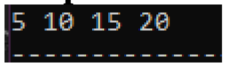


```
First element: 10
Third element: 30
```

- **Example 5. Iterating Over a Vector Using Iterators:**

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec = {5, 10, 15, 20};
    // Iterating using iterators
    for(std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

- **Output:**



```
5 10 15 20
-------------
```

Lab 5

- **List:**

➢ Definition:

A list in C++ is a sequence container that allows non-contiguous memory allocation. It is part of the Standard Template Library (STL) and is implemented as a doubly linked list, which means each element points to both the next and the previous element. This allows for efficient insertions and deletions from anywhere in the list.

➢ Syntax:

```
#include <list>
```

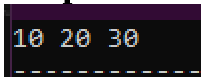
You can declare a list as follows:

```
std::list<data_type> list_name;
```

▪ Example 1: Creating and displaying a list:

```
#include <iostream>

#include <list>

using namespace std;

int main() {

   list<int> myList; // Declare a list of integers

   // Adding elements to the list

   myList.push_back(10);

   myList.push_back(20);

   myList.push_back(30);

   // Displaying the elements of the list

   for (int num : myList) {

      cout << num << " ";

   }

   return 0;
}
```

- **Output:**

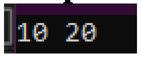


```
10 20 30
```

- **Example 2. Inserting elements at the beginning and end:**

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList;
    myList.push_back(20); // Add 20 at the end
    myList.push_front(10); // Add 10 at the beginning
    // Display the list
    for (int num : myList) {
        cout << num << " ";
    }
    return 0;
}
```

- **Output:**

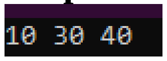


```
10 20
```

- **Example 3. Removing elements from a list:**

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {10, 20, 30, 40};
    myList.remove(20); // Remove the element with value 20

    // Display the list
    for (int num : myList) {
        cout << num << " ";
    }
    return 0;
}
```

- **Output:**

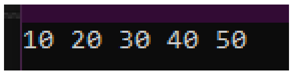


```
10 30 40
```

- **Example 4. Sorting a list:**

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {30, 10, 20, 50, 40};
    myList.sort(); // Sort the list
    // Display the sorted list
    for (int num : myList) {
        cout << num << " ";
    }
    return 0;
}
```

- **Output:**

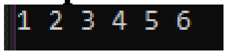


```
10 20 30 40 50
```

- **Example 5. Merging two lists:**

```
#include <iostream>

#include <list>

using namespace std;

int main() {

    list<int> list1 = {1, 2, 3};

    list<int> list2 = {4, 5, 6};

    list1.merge(list2); // Merge list2 into list1

    // Display the merged list

    for (int num : list1) {

        cout << num << " ";

    }

    return 0;

}
```

- **Output:**



```
1 2 3 4 5 6
```

Lab 6

➢ Stack:

- **Definition:**

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. You can think of it like a stack of plates; you can only add or remove the top plate.

- The primary operations of a stack are:

1. Push: Add an element to the top of the stack.

2. Pop: Remove the top element from the stack.

3. Peek/Top: Get the value of the top element without removing it.

4. IsEmpty: Check if the stack is empty.

➢ Syntax :

Here's the basic syntax:

```
#include <stack>

// Creating a stack

std::stack<data_type> stackName;
```

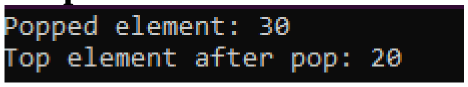
➢ Examples :

▪ Example:Basic Stack Operations (Push and Pop):

```
#include <iostream>

#include <stack>

using namespace std;

int main() {

   stack<int> myStack;

   // Push elements onto the stack

   myStack.push(10);
```

```
    myStack.push(20);

    myStack.push(30);

    // Pop the top element

    cout << "Popped element: " << myStack.top() << endl; // Outputs 30

    myStack.pop();

    // Display the top element

    cout << "Top element after pop: " << myStack.top() << endl; // Outputs 20

    return 0;

}
```

- **Output:**

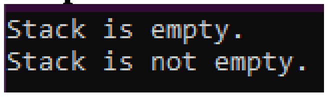


```
Popped element: 30
Top element after pop: 20
```

- **Example 2. Check if Stack is Empty:**

```
#include <iostream>

#include <stack>

using namespace std;

int main() {

    stack<int> myStack;

    // Check if stack is empty

    if (myStack.empty()) {

        cout << "Stack is empty." << endl;

    } else {

        cout << "Stack is not empty." << endl;

    }

    // Push an element

    myStack.push(10);

    // Check again

    if (myStack.empty()) {
        cout << "Stack is empty." << endl;
    } else {
        cout << "Stack is not empty." << endl; // This will be printed
    }
    return 0;
}
```

- **Output:**

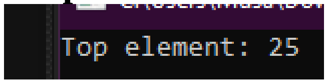


```
Stack is empty.
Stack is not empty.
```

- **Example 3. Peek at the Top Element:**

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> myStack;
    myStack.push(5);
    myStack.push(15);
    myStack.push(25);
    // Peek at the top element
    cout << "Top element: " << myStack.top() << endl; // Outputs 25
    return 0;
}
```

- **Output:**



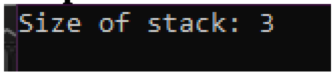
```
Top element: 25
```

- **Example 4. Size of the Stack:**

```
#include <iostream>
#include <stack>
```

```
using namespace std;

int main() {

    stack<int> myStack;

    myStack.push(1);

    myStack.push(2);

    myStack.push(3);

    // Display the size of the stack

    cout << "Size of stack: " << myStack.size() << endl; // Outputs 3

    return 0;

}
```

- **Output:**



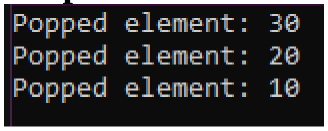
```
Size of stack: 3
```

- **Example 5. Pop All Elements from the Stack:**

```
#include <iostream>

#include <stack>

using namespace std;

int main() {

    stack<int> myStack;

    myStack.push(10);

    myStack.push(20);

    myStack.push(30);

    // Pop all elements

    while (!myStack.empty()) {

        cout << "Popped element: " << myStack.top() << endl; // Outputs 30, 20, 10

        myStack.pop();

    }

    return 0;
```

}

- **Output:**



```
Popped element: 30
Popped element: 20
Popped element: 10
```

Lab 7

- **Queue:**

Definition:

A queue is a linear data structure that follows the First In First Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed. It can be thought of as a line of people waiting for service, where the person who arrives first is served first.

Syntax:

The syntax for including the queue library is:

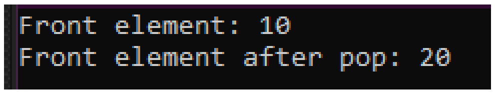
```
#include <queue>
```

Examples :

Example 1.Basic Queue Operations:

```
#include <iostream>

#include <queue>

int main() {

   std::queue<int> q;

   // Adding elements to the queue

   q.push(10);

   q.push(20);

   q.push(30);

   // Displaying the front element

   std::cout << "Front element: " << q.front() << std::endl;

   // Removing an element from the queue

   q.pop();

   std::cout << "Front element after pop: " << q.front() << std::endl;

   return 0;

}
```

- **Output:**

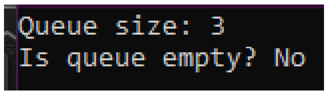


```
Front element: 10
Front element after pop: 20
```

- **<u>Example 2. Queue Size and Empty Check:</u>**

```
#include <iostream>
#include <queue>
int main() {
   std::queue<int> q;
   q.push(1);
   q.push(2);
   q.push(3);
   std::cout << "Queue size: " << q.size() << std::endl;
   std::cout << "Is queue empty? " << (q.empty() ? "Yes" : "No") << std::endl;
   return 0;
}
```

- **Output:**



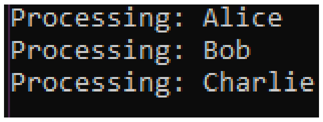
```
Queue size: 3
Is queue empty? No
```

- **<u>Example 3. Queue of Strings:</u>**

```
#include <iostream>
#include <queue>
#include <string>
```

```
int main() {
    std::queue<std::string> q;
    q.push("Alice");
    q.push("Bob");
    q.push("Charlie");
    while (!q.empty()) {
        std::cout << "Processing: " << q.front() << std::endl;
        q.pop();
    }
    return 0;
}
```

- **Output:**

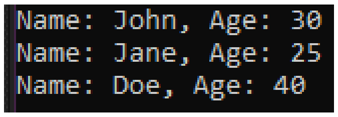


```
Processing: Alice
Processing: Bob
Processing: Charlie
```

- **<u>Example 4. Queue with Custom Data Structure:</u>**

```
#include <iostream>
#include <queue>
struct Person {
    std::string name;
    int age;
};
int main() {
    std::queue<Person> q;
    q.push({"John", 30});
    q.push({"Jane", 25});
    q.push({"Doe", 40});
    while (!q.empty()) {
        Person p = q.front();
        std::cout << "Name: " << p.name << ", Age: " << p.age << std::endl;
        q.pop();
    }
    return 0;
}
```

- **Output:**

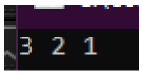


```
Name: John, Age: 30
Name: Jane, Age: 25
Name: Doe, Age: 40
```

- **<u>Example 5. Using Queue to Reverse a Sequence:</u>**

```
#include <iostream>
#include <queue>
#include <stack>
int main() {
    std::queue<int> q;
    std::stack<int> s;
    // Adding elements to the queue
    q.push(1);
    q.push(2);
    q.push(3);
    // Transfer elements to stack to reverse
    while (!q.empty()) {
        s.push(q.front());
        q.pop();
    }
    // Transfer elements back to queue
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }
    // Displaying reversed queue
    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

- **Output:**



```
3 2 1
```

Lab 8

- **De Queue:**

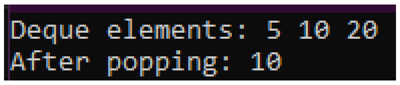
➢ **Definition:**
A deque, short for "double-ended queue," is a data structure that allows insertion and deletion of elements from both ends (front and back). It is part of the C++ Standard Template Library (STL) and provides a flexible way to manage a collection of elements.

➢ **Syntax :**
```
#include <deque>

You can declare a deque like this:
std::deque<data_type> deque_name;
```

➢ **Examples:**

▪ **Example 1.Basic Deque Operations:**
```
#include <iostream>
#include <deque>
int main() {
   std::deque<int> dq;
   // Adding elements to the deque
   dq.push_back(10);
   dq.push_back(20);
   dq.push_front(5);
   std::cout << "Deque elements: ";
   for (int x : dq) {
      std::cout << x << " ";
   }
   std::cout << std::endl;
   // Removing elements
   dq.pop_back();
   dq.pop_front();
   std::cout << "After popping: ";
   for (int x : dq) {
      std::cout << x << " ";
   }
   std::cout << std::endl;

   return 0;
}
```

- **Output:**

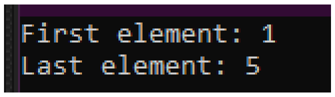


```
Deque elements: 5 10 20
After popping: 10
```

- **Example 2. Accessing Deque Elements:**

```
#include <iostream>
#include <deque>

int main() {
   std::deque<int> dq = {1, 2, 3, 4, 5};

   std::cout << "First element: " << dq.front() << std::endl;
   std::cout << "Last element: " << dq.back() << std::endl;

   return 0;
}
```

- **Output:**

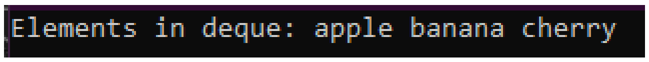


```
First element: 1
Last element: 5
```

- **Example 3. Iterating Through a Deque:**

```
#include <iostream>
#include <deque>
int main() {
   std::deque<std::string> dq = {"apple", "banana", "cherry"};

   std::cout << "Elements in deque: ";
   for (std::deque<std::string>::iterator it = dq.begin(); it != dq.end(); ++it) {
      std::cout << *it << " ";
   }
   std::cout << std::endl;

   return 0;
}
```

- **Output:**

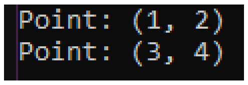


```
Elements in deque: apple banana cherry
```

- **Example 4. Deque with Custom Data Type:**

```
#include <iostream>
#include <deque>
struct Point {
    int x, y;
};

int main() {
    std::deque<Point> dq;
    dq.push_back({1, 2});
    dq.push_back({3, 4});

    for (const Point& p : dq) {
        std::cout << "Point: (" << p.x << ", " << p.y << ")" << std::endl;
    }

    return 0;
}
```

- **Output:**

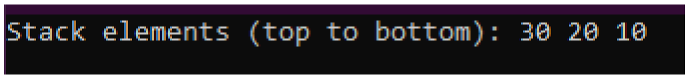


```
Point: (1, 2)
Point: (3, 4)
```

- **Example 5. Deque as a Stack:**

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq;

    // Using deque as a stack
    dq.push_back(10);
    dq.push_back(20);
    dq.push_back(30);

    std::cout << "Stack elements (top to bottom): ";
    while (!dq.empty()) {
        std::cout << dq.back() << " ";
        dq.pop_back();
    }
    std::cout << std::endl;

    return 0;
}
```

- **Output:**



```
Stack elements (top to bottom): 30 20 10
```

Lab 9

- **Tree:**

Definition:

A tree is a data structure that simulates a hierarchical tree structure, consisting of nodes connected by edges. Each tree has a root node, and every node can have zero or more child nodes.

- **Characteristics**

The main characteristics of a tree include:

1. Root: The top node of the tree.
2. Node: An element of the tree that contains data and may link to other nodes.
3. Edge: The connection between two nodes.
4. Leaf: A node that does not have any children.
5. Height: The length of the longest path from the root to a leaf node.

Syntax :

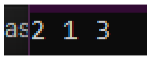
```
struct TreeNode {
    int data; // The data stored in the node
    TreeNode* left; // Pointer to the left child
    TreeNode* right; // Pointer to the right child

    // Constructor to initialize the node
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};
```

Examples:

Example 1. Creating a Simple Binary Tree:

```
#include <iostream>
using namespace std;
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

void printTree(TreeNode* node) {
    if (node == nullptr) return;
    printTree(node->left);
    cout << node->data << " ";
    printTree(node->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    printTree(root);
    return 0;
}
```

- **Output:**

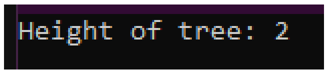


```
as2 1 3
```

- <u>**Example 2. Calculating the Height of a Binary Tree:**</u>

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

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

int height(TreeNode* node) {
    if (node == nullptr) return 0;
    return 1 + max(height(node->left), height(node->right));
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    cout << "Height of tree: " << height(root) << endl;
    return 0;
}
```

- **Output:**



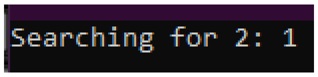
```
Height of tree: 2
```

- <u>**Example 3. Searching for a Value in a Binary Search Tree:**</u>

```
#include <iostream>
```

```
using namespace std;

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

bool search(TreeNode* node, int value) {
    if (node == nullptr) return false;
    if (node->data == value) return true;
    return search(node->left, value) || search(node->right, value);
}

int main() {
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(6);
    cout << "Searching for 2: " << search(root, 2) << endl;
    return 0;
}
```

- **Output:**

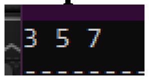


```
Searching for 2: 1
```

- **Example 4. Inserting a Value into a Binary Search Tree:**

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

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

TreeNode* insert(TreeNode* node, int value) {
    if (node == nullptr) return new TreeNode(value);
    if (value < node->data) {
        node->left = insert(node->left, value);
    } else {
        node->right = insert(node->right, value);
    }
    return node;
}

void printTree(TreeNode* node) {
    if (node == nullptr) return;
    printTree(node->left);
    cout << node->data << " ";
    printTree(node->right);
}

int main() {
    TreeNode* root = nullptr;
    root = insert(root, 5);
    insert(root, 3);
    insert(root, 7);
    printTree(root);
    return 0;
}
```

- **Output:**



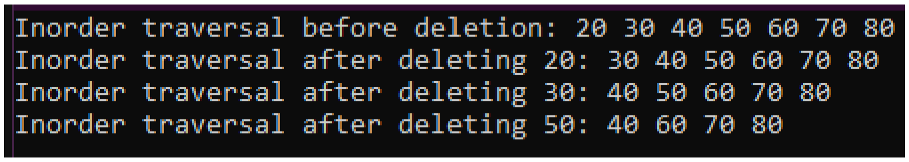
```
3 5 7
```

<u>5. Deleting a Node from a Binary Search Tree:</u>

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

// Definition of the structure for a tree node
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Function to insert a node in the BST
Node* insert(Node* root, int data) {
    if (root == nullptr) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }
    return root;
}

// Function to find the minimum value node in the BST
Node* findMin(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the BST
Node* deleteNode(Node* root, int data) {
    if (root == nullptr) {
        return root;
    }

    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children: get the inorder successor (smallest in the right subtree)
        Node* temp = findMin(root->right);
        root->data = temp->data; // Copy the inorder successor's content to this node
        root->right = deleteNode(root->right, temp->data); // Delete the inorder successor
    }
    return root;
```

```
}

// Function to print the inorder traversal of the BST
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    cout << "Inorder traversal before deletion: ";
    inorder(root);
    cout << endl;

    root = deleteNode(root, 20);
    cout << "Inorder traversal after deleting 20: ";
    inorder(root);
    cout << endl;

    root = deleteNode(root, 30);
    cout << "Inorder traversal after deleting 30: ";
    inorder(root);
    cout << endl;

    root = deleteNode(root, 50);
    cout << "Inorder traversal after deleting 50: ";
    inorder(root);
    cout << endl;

    return 0;
```

- **Output:**



```
Inorder traversal before deletion: 20 30 40 50 60 70 80
Inorder traversal after deleting 20: 30 40 50 60 70 80
Inorder traversal after deleting 30: 40 50 60 70 80
Inorder traversal after deleting 50: 40 60 70 80
```

Lab 10

- **Binary Search Trees:**

➢ **Definition:** A Binary Search Tree (BST) is a type of binary tree in which each node follows a specific ordering property:

Left Subtree: Contains nodes with keys less than the parent node's key.

Right Subtree: Contains nodes with keys greater than the parent node's key. This property makes BSTs highly efficient for searching, insertion, and deletion operations.

➢ **Key Characteristics of a Tree:**

Ordered Structure: Nodes are arranged in a way that enables efficient operations.

Recursive Representation: BST operations like traversal and manipulation are naturally implemented using recursion.

No Duplicates: In a standard BST, duplicate elements are not allowed.

Traversals: Common traversal methods include in-order, pre-order, and post-order, where in-order traversal yields a sorted sequence of elements.

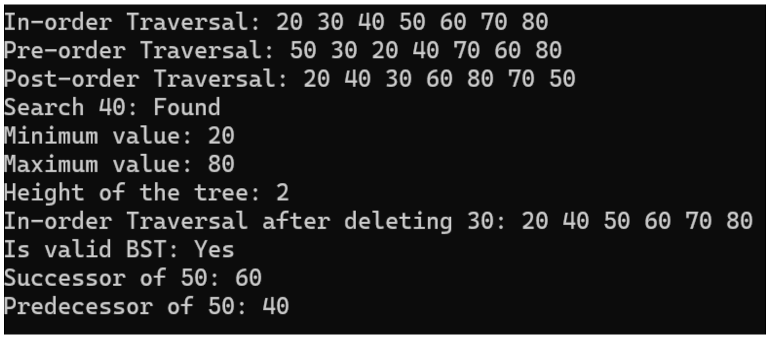
➢ **Examples:**

▪ **Example 1. Insertion in BST:**

```
struct Node {
   int data;
   Node* left;
   Node* right;
   Node(int value) :
data(value), left(nullptr), right(nullptr) {}
};
Node* insert(Node* root, int value) {
   if (root == nullptr) return new Node(value);
   if (value < root->data)
      root->left = insert(root->left, value);
   else
```

```
        root->right = insert(root->right, value);
    return root;}
```

- **Output:**

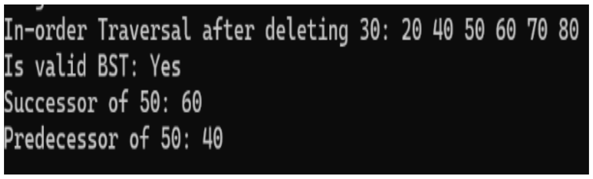


```
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
Search 40: Found
Minimum value: 20
Maximum value: 80
Height of the tree: 2
In-order Traversal after deleting 30: 20 40 50 60 70 80
Is valid BST: Yes
Successor of 50: 60
Predecessor of 50: 40
```

- **<u>Example 2.Deletion in BST:</u>**

```
Node* findMin(Node* root) {
    while (root->left != nullptr) root = root->left;
    return root;
}
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) return root;
    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else {
        if (root->left == nullptr) return root->right;
        if (root->right == nullptr) return root->left;
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

- **<u>Output:</u>**

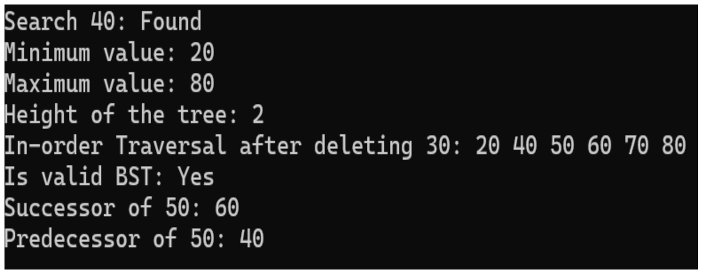


```
In-order Traversal after deleting 30: 20 40 50 60 70 80
Is valid BST: Yes
Successor of 50: 60
Predecessor of 50: 40
```

- **<u>Example 3.Searching in BST</u>**

```
bool search(Node* root, int value) {
    if (root == nullptr) return false;
    if (root->data == value) return true;
    if (value < root->data)
        return search(root->left, value);
    return search(root->right, value);}
```

Output

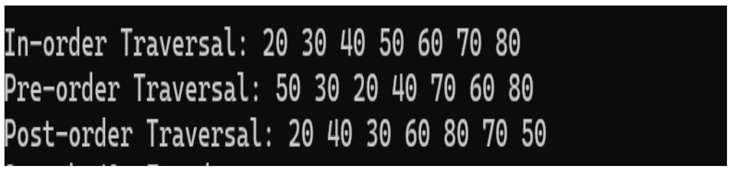


```
Search 40: Found
Minimum value: 20
Maximum value: 80
Height of the tree: 2
In-order Traversal after deleting 30: 20 40 50 60 70 80
Is valid BST: Yes
Successor of 50: 60
Predecessor of 50: 40
```

- **<u>Example 4.Traversing in BST(in-order, pre-order, post-order):</u>**

```
void inOrder(Node* root) {
   if (root == nullptr) return;
inOrder(root->left);
std::cout << root->data << " ";
inOrder(root->right);
}

void preOrder(Node* root) {
   if (root == nullptr) return;
std::cout << root->data << " ";
preOrder(root->left);
preOrder(root->right);
}
void postOrder(Node* root) {
   if (root == nullptr) return;
postOrder(root->left);
postOrder(root->right);
std::cout << root->data << " ";
}
```

- **Output:**

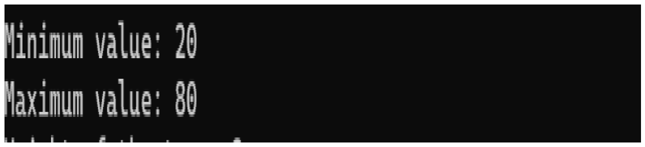


```
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
```

- **Example 5.Minimum and Maximum:**

```
int findMinValue(Node* root) {
    while (root->left != nullptr) root = root->left;
    return root->data;
}
int findMaxValue(Node* root) {
    while (root->right != nullptr) root = root->right;
    return root->data;
}
```

➢ **Output**



```
Minimum value: 20
Maximum value: 80
```

Lab 11

- **Link List:**
- **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.

- **Syntax:**

- **Examples:**

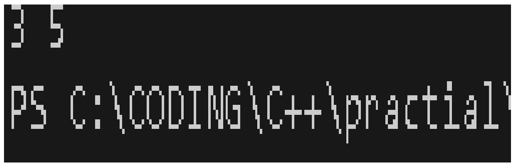
Q1: Insertion at any point:

```
#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 faisal;
    hateem.create(5);
    hateem.insertAtBegin(3);
    hateem.display();
}
```

- **Output:**



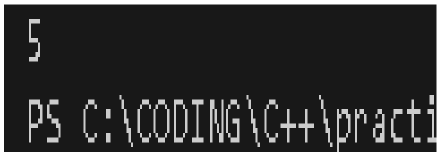
```
3 5
PS C:\CODING\C++\practial
```

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

- **Output:**



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

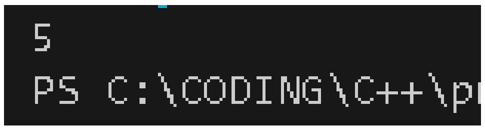
Q3: insert at any point

```
#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 hateem;
    hateem.create(5);
    hateem.create(5);
    hateem.create(5);
    hateem.insertAtPosition(2,2);
    hateem.display();
}
```

- **Output:**



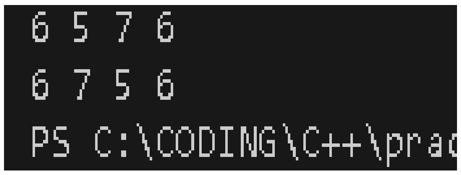
```
5
PS C:\CODING\C++\pr
```

Q4: delete 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) {
    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 hateem;

    hateem.create(5);

    hateem.create(10);

    hateem.create(15);

    cout<< "Original list: ";

    hateem.display();

    hateem.deleteAtPosition(2);

    cout<< "After deleting at position 2: ";

    hateem.display();

    hateem.deleteAtPosition(0);

    cout<< "After deleting at position 0: ";

    hateem.display();

    return 0;

}
```

- **Output:**



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

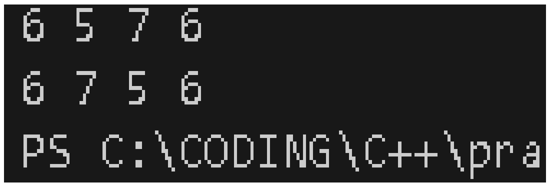
Q5: insert at last

```
#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 hateem;
    hateem.create(5);
    hateem.insertAtEnd(6);
    hateem.display();
}
```

- **<u>Output:</u>**



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

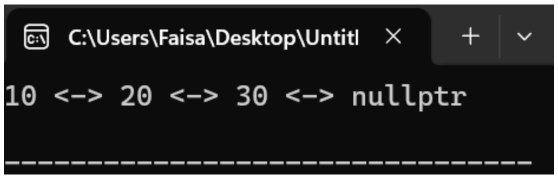
Lab 12

- **Doubly Link List**
- **Characteristics**
  1. Each node has pointers to both the previous and next nodes.
  2. Traversal is bidirectional.
  3. The head's prev pointer and the last node's next pointer are nullptr.
- **Example Programs**

1. Node Structure

```
struct DoublyNode {
   int data;
DoublyNode* prev;
DoublyNode* next;

DoublyNode(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

Output

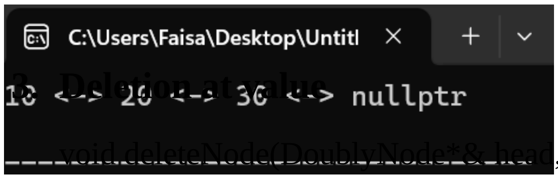


```
10 <-> 20 <-> 30 <-> nullptr

--------------------------------
```

2. Insertion at end

```
void insertEnd(DoublyNode*& head, int val) {
DoublyNode* newNode = new DoublyNode(val);
   if (head == nullptr) {
      head = newNode;
      return;
   }
DoublyNode* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
newNode->prev = temp;}
```

Output

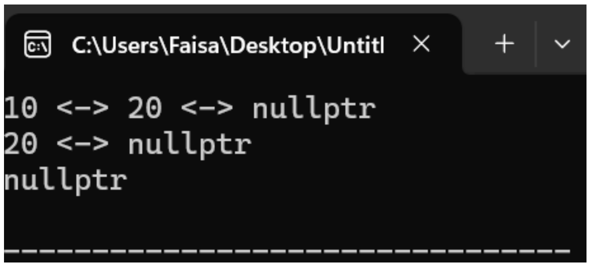


```
10 <-> 20 <-> 30 <-> nullptr
```

3. Deletion at value

```
void deleteNode(DoublyNode*& head, int val) {
    if (head == nullptr) return;
    if (head->data == val) {
DoublyNode* temp = head;
        head = head->next;
        if (head != nullptr) head->prev = nullptr;
        delete temp;
        return;
    }
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;

}
```

Output



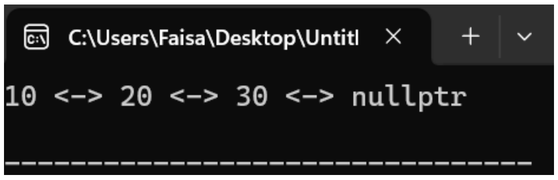
C:\Users\Faisa\Desktop\Untitl

```
10 <-> 20 <-> nullptr
20 <-> nullptr
nullptr

--------------------------------
```

4. Display the list

```
void display(DoublyNode* head) {

DoublyNode* temp = head;

while (temp != nullptr) {

  cout << temp->data << " <-> ";

  temp = temp->next;

}

cout << "nullptr" <<endl;}
```

Output



C:\Users\Faisa\Desktop\Untitl

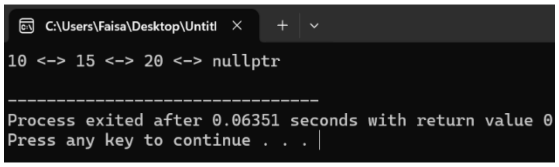
```
10 <-> 20 <-> 30 <-> nullptr

--------------------------------
```

5. Insert at postion

```
void insertatposition(int position,int data){
        node *n = new node();
        n->data=data;
        if(position==1){
            n->next=head;
            if(head!=NULL){
                head->prev=n;}
            head = n;
            if(tail==NULL){
                tail=n;}}
        else{
            current = head;
            int count = 1;
            while(current!=nullptr&& count<position-1){
                current = current->next;
                count++;}
            if(current==NULL){
                cout<<"position out of bounds";
                delete n;
                return;}
            n->next=current->next;
            n->prev=current;
            if (current->next != nullptr) {
    current->next->prev = n;} else { // If inserting at the tail
```

```
        tail = n;}

    current->next = n;}}
```

Output



C:\Users\Faisa\Desktop\Untitl

```
10 <-> 15 <-> 20 <-> nullptr

--------------------------------
Process exited after 0.06351 seconds with return value 0
Press any key to continue . . .
```

Lab 13

- **Circular Link List**

➢ **Characteristics**

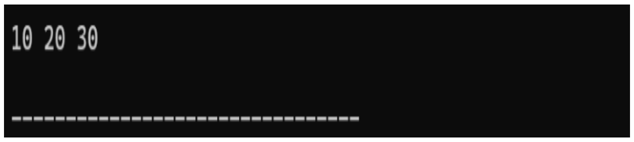
- The last node points to the first node.
- Can be singly or doubly linked.
- Enables circular traversal.

- **Example Programs**

1. Node Structure

```
struct CNode {
    int data;
CNode* next;

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

Output

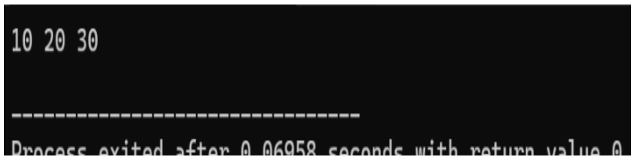


```
10 20 30

-----------------------------
```

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

Output

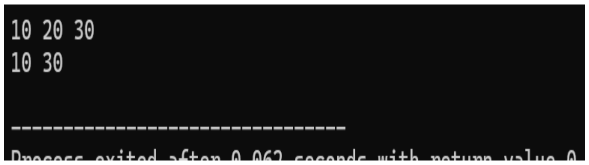


```
10 20 30

--------------------------------
Process exited after 0.06958 seconds with return value 0
```

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;
    } 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;
}
```

Output

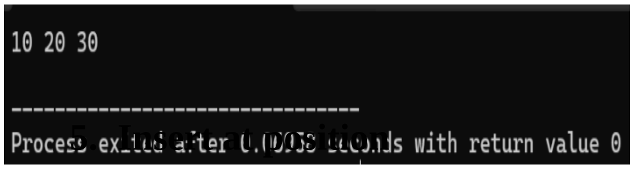


```
10 20 30
10 30

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

4. <u>Display the list</u>

```
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

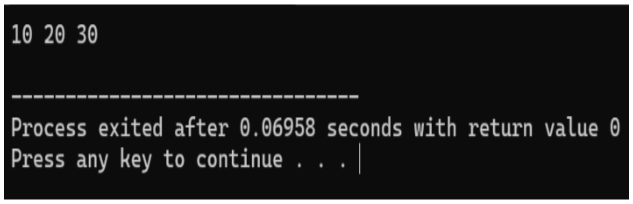


```
10 20 30

------------------------------
Process exited after 0.[illegible] seconds with return value 0
```

5. Insert at position

```
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;
```

- **<u>Output:</u>**



```
10 20 30

--------------------------------
Process exited after 0.06958 seconds with return value 0
Press any key to continue . . .
```

>---<