The University of Faisalabad

Submitted by: Zoomer Fiaz

Submitted to: Miss Irsha Qureshi

Registration: 2023-BS-AI-003

Department: Artificial Intelligence

Course Code: CS-216

Lab 01[Introduction]

VARIABLES IN C++

Variables are fundamental building blocks in C++ programming. They are used to store data that can be modified and accessed throughout a program.

- ***Declaring Variables***

To declare a variable in C++, you need to specify the type of the variable followed by its name. The type determines the kind of data the variable can hold.

Syntax:

```
type variableName;
```

- **Initializing Variables**

Variables can be initialized (given a value) at the time of declaration or later in the code. Initialization can be done using the assignment operator =.

Syntax:

```
type variableName = value;
```

TYPES OF VARABLES IN C++

C++ supports various data types for variables, each serving a specific purpose:

- **int**: Used for integers (whole numbers).
- **float**: Used for floating-point numbers (numbers with decimals).
- **double**: Like float but with double precision.
- **char**: Used for single characters.
- **string**: Used for text (requires the #include <string> header).
- **bool**: Used for boolean values (true or false).

FUNCTIONS IN C++

Functions are blocks of code that perform a specific task and can be reused throughout a program. They help in organizing code, making it more readable, and reducing redundancy.

- ***Function Declaration***

A function declaration (or prototype) tells the compiler about the function's name, return type, and parameters. It does not contain the actual body of the function.

Syntax:

```
return_type function_name(parameter_list);
```

Function Definition

A function definition contains the actual body of the function, which includes the statements that perform the task.

Syntax:

```
return_type function_name(parameter_list)

{// Function body}
```

Pointers in C++

Pointers are variables that store the memory address of another variable. They are powerful tools that allow for direct memory access and manipulation, which can lead to more efficient code.

- ***Pointer Declaration***

A pointer is declared by specifying the data type it points to, followed by an asterisk (*) and the pointer's name.

Syntax

```
data_type *pointer_name;
```

- ***Pointer Initialization***

A pointer is initialized by assigning it the address of another variable, using the address-of operator (&).

***Example**:*

```
int value = 42;

int *ptr = &value; // ptr now holds the address of value
```

- ***Dereferencing Pointers***

Dereferencing a pointer means accessing the value at the memory address stored in the pointer, using the dereference operator (*).

***Example**:*

```
int value = 42;

int *ptr = &value;
```

ARRAYS

An array is a linear data structure that stores elements of the same data type in contiguous memory locations. Arrays are used to store lists of related information, **such as** a **shopping list**, a **list of student names**, or a **list of exam grades**.

//EXAMPLE PROGRAM//

```
#include <iostream>
using namespace std;
int main()
 {
   // Declare and initialize an array of integers with 5 elements
   int numbers[5] = {10, 20, 30, 40, 50};
   // Print the elements of the array
   for (int i = 0; i < 5; i++) {
      cout << "Element at index " << i << ": " << numbers[i] << endl;
   }
   return 0;
}
```

TYPES OF ARRAYS

- One-Dimensional Arrays (1D Arrays) These are the simplest form of arrays, consisting of a single line of elements. ...
- Two-Dimensional Arrays (2D Arrays) A two-dimensional array in data structure, often thought of as a matrix, consists of rows and columns. ...
- Multi-Dimensional Arrays.

CHARACTERISTICS OF ARRAY IN DATA STRUCTURES

- **Fixed size**

Once an array is created, its size cannot be changed.

- **Homogeneous elements**

All elements in an array must be of the same data type, such as all integers, all floats, or all characters.

- **Multidimensional arrays**

Arrays can have multiple dimensions, with each dimension represented by a bracket pair. For example, a two-dimensional array is an array of arrays, where each element in the array is itself an array.

- **Array variables**

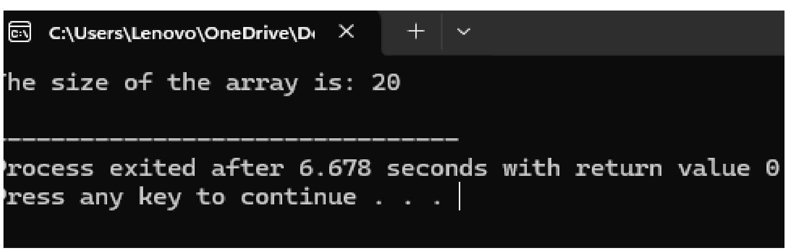
An array variable holds a reference to an array object but does not create the array object itself

Lab 02 [Self Practice]

1. Create a c++ program to check the size of an array.

```
#include <iostream>
using namespace std;
int main()
{
   //Declare an array to check the size
 int Numbers[5] = {15, 25, 38, 42, 54};

   //print the size of the array
   cout << "The size of the array is: " << sizeof(Numbers) << endl;

   return 0;
}
```

Output:

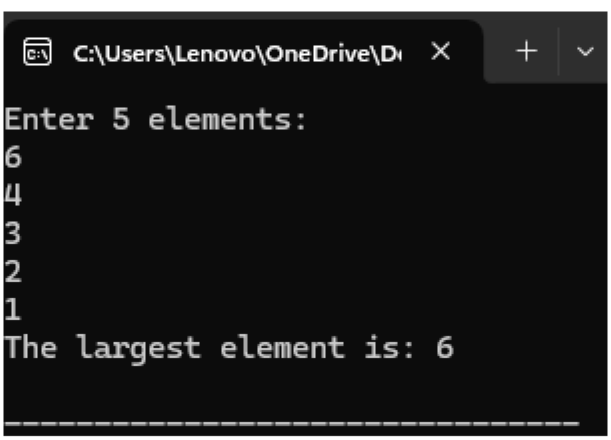


2.Create a c++ program to find the largest element in the array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5], largest;

 cout << "Enter 5 elements:" << endl;

  // Storing 5 elements in an array
 for (int i = 0; i < 5; i++)
```

```
 {
   // Input elements in array
cin >> arr[i];
}
 // Assume that the first element is the largest
largest = arr[0];
 // Loop to find the largest element
for (int i = 1; i < 5; i++)
{
  // Compare the current element with the largest element
if (arr[i] > largest)
{
  // If the current element is greater, update the largest element
largest = arr[i];
}
}
 // Print the largest element of the array
cout << "The largest element is: " << largest << endl;
return 0;
}
```

Output:



3.Create a c++ to store and display the names of five cloth brands.

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

```
using namespace std;
int main()
{
    // Declare an array to store the names of 5 cloth brands
    string cloths[5];

    // Assign cloth brand names to the array elements
    cloths[0] = "Lulusar";
   cloths[1] = "Khaadi";
    cloths[2] = "Sapphire";
    cloths[3] = "Dior";
    cloths[4] = "Cross stich";
    // Loop through the array and print each cloth brand name
    for(int i = 0; i < 5; i++)
{
        cout << cloths[i] << "\n";
    }

    return 0;
}
```

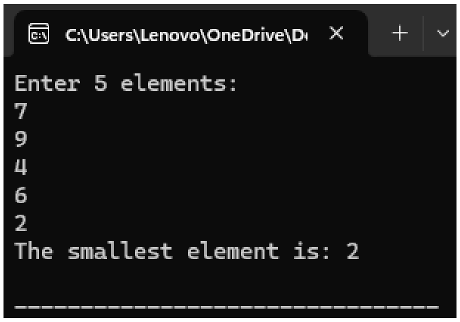
Output:

4. Create a c++ program to find the minimum element in the array.

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

```
int main()
{
 int arr[5], smallest;

 cout << "Enter 5 elements:" << endl;

  // Storing 5 elements in an array
 for (int i = 0; i < 5; i++)
   {
     // Input elements in array
 cin >> arr[i];
 }
  // Assume that the first element is the smallest
 smallest = arr[0];

  // Loop to find the smallest element
 for (int i = 1; i < 5; i++)
 {
   // Compare the current element with the smallest element
 if (arr[i] < smallest)
 {
   // If the current element is smallesr, update the smallest element
 smallest = arr[i];
 }
 }

  // Print the smallest element of the array
 cout << "The smallest element is: " << smallest << endl;

 return 0;
}
```

Output:



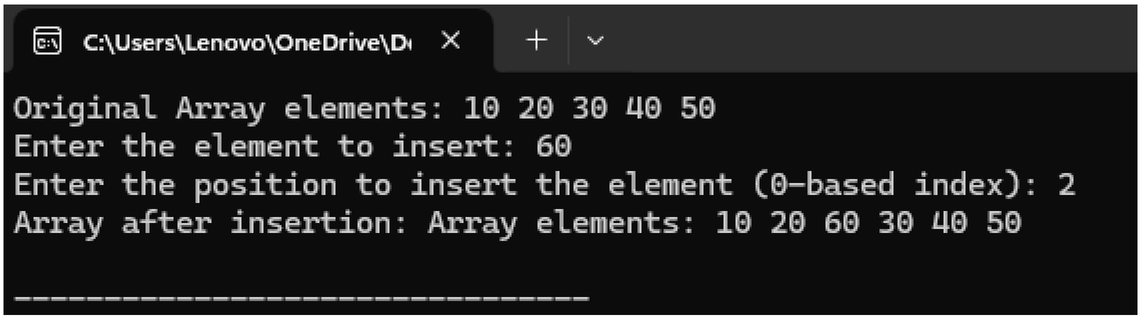
5.Create a c++ program to do insertion in array.

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

void insertElement(int arr[], int& n, int element, int position) {
    // If the position is valid
    if (position < 0 || position > n) {
        cout << "Invalid position!" << endl;
        return;
    }

    // Shift elements to the right to make space for the new element
    for (int i = n; i > position; i--) {
        arr[i] = arr[i - 1];
    }

    // Insert the element at the specified position
    arr[position] = element;

    // Increment the size of the array
    n++;
}

void displayArray(int arr[], int n) {
    cout << "Array elements: ";
```

```
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[100];  // Array of fixed size 100
    int n = 5;  // Initial number of elements in the array

    // Initializing array
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;

    cout << "Original ";
    displayArray(arr, n);

    int element, position;
    cout << "Enter the element to insert: ";
    cin >> element;

    cout << "Enter the position to insert the element (0-based index): ";
    cin >> position;

    // Perform insertion
    insertElement(arr, n, element, position);

    cout << "Array after insertion: ";
```

```
    displayArray(arr, n);

    return 0;
}
```

Output:

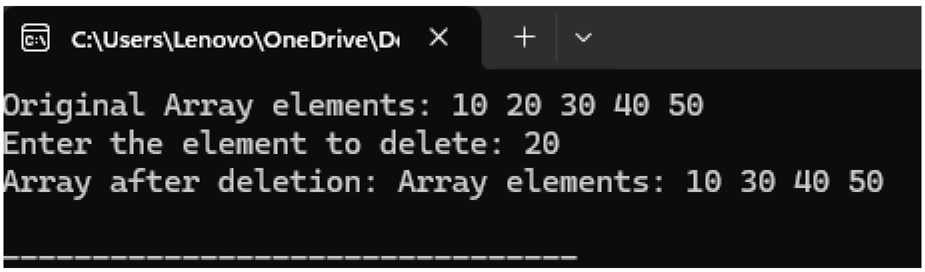


6.Create a c++ program to do deletion in array.

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

void deleteElement(int arr[], int& n, int element) {
    // Find the position of the element
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == element) {
            break;
        }
    }

    // If the element was not found
    if (i == n) {
        cout << "Element not found!" << endl;
        return;
    }

    // Shift all elements after the found element to the left by one position
    for (int j = i; j < n - 1; j++) {
```

```
        arr[j] = arr[j + 1];
    }

    // Decrease the size of the array
    n--;
}

void displayArray(int arr[], int n) {
    cout << "Array elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[100];  // Array of fixed size 100
    int n = 5;  // Initial number of elements in the array

    // Initializing array
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;

    cout << "Original ";
    displayArray(arr, n);

    int element;
    cout << "Enter the element to delete: ";
```

```
    cin >> element;

    // Perform deletion
    deleteElement(arr, n, element);

    cout << "Array after deletion: ";
    displayArray(arr, n);

    return 0;
}
```



Lab 03 [IMPLEMENTATION OF MULTI-DIMENSIONAL ARRAY]

MULTI-DIMENSIONAL ARRAYS

A multi-dimensional array is an array of arrays. It allows for the representation of more complex data structures like matrices and tables. The most common multi-dimensional arrays are two-dimensional (2D) and three-dimensional (3D) arrays, but arrays can have more dimensions based on the need.

TWO-DIMENSIONAL ARRAYS

A two-dimensional array can be visualized as a table or matrix with rows and columns. Each element in a 2D array is identified by its row and column indices.

SYNTAX:

```
type arrayName[rows][columns] = {
    {value1, value2, value3, ..., valueN},
    {value1, value2, value3, ..., valueN},
    ...
    {value1, value2, value3, ..., valueN}
};
```

ACCESSING ELEMENTS

Elements in a 2D array are accessed using two indices: matrix[row][column].

USAGE

2D arrays are commonly used to store tabular data, such as a spreadsheet, or for operations on matrices in mathematical computations.

THREE-DIMENSIONAL ARRAYS

A three-dimensional array can be visualized as a cube. Each element is identified by three indices representing the dimensions (depth, rows, and columns).

SYNTAX:

```
type arrayName[size1][size2][size3] = {
    {
        {value1, value2, value3, ..., valueN},
        {value1, value2, value3, ..., valueN},
       ... },
    {
```

```
        {value1, value2, value3, ..., valueN},
        {value1, value2, value3, ..., valueN},
        ... },
    ...
};
```

ACCESSING ELEMENTS

Elements in a 3D array are accessed using three indices: cube[depth][row][column].

USAGE

3D arrays are useful for storing multi-layered data, such as volumetric data in simulations, 3D graphics, and scientific computations.

HIGHER-DIMENSIONAL ARRAYS

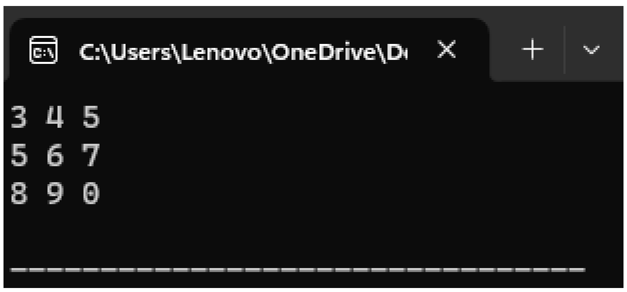
Higher-dimensional arrays (4D, 5D, etc.) follow the same principle, but they are rarely used due to the complexity in managing and understanding them. They are generally used in specialized fields that require handling of multi-dimensional data.

1. **Create a C++ program to define a 3x3 2D array and initialize it with specific values. The program should then print the array in a matrix form using nested loops.**

```
#include <iostream>
using namespace std;
int main()
{
    // Define and initialize a 3x3 2D array
    int arr[3][3] = {
        {3, 4, 5}, // First Row
        {5, 6, 7}, // Second Row
        {8, 9, 0}  // Third Row
    };

    // Loop through the rows of the array
    for (int i = 0; i < 3; i++) {
        // Loop through the columns of the array
        for (int j = 0; j < 3; j++) {
            // Print the current element followed by a space
            cout << arr[i][j] << " ";
```

```
        }

// Move to the next line after printing all columns of the current row
        cout << endl;
    }

    return 0;
}
```

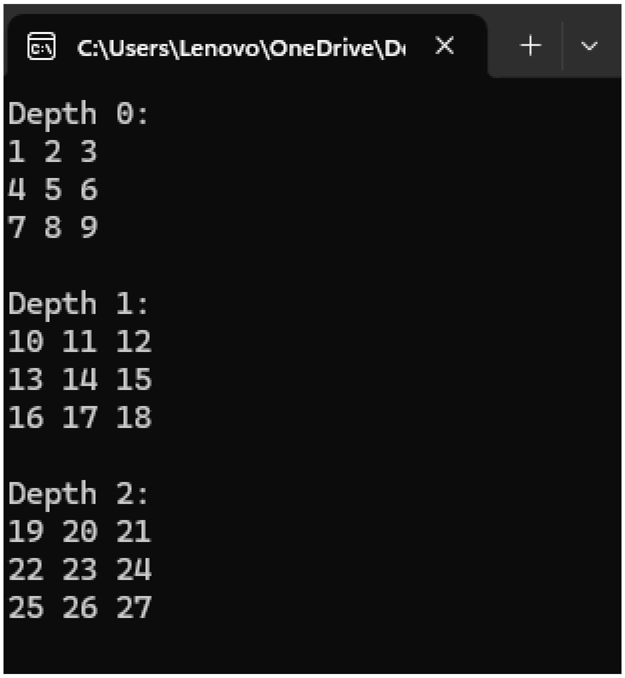
Output:



2.Create a C++ 3D array program that demonstrates how to declare, initialize, and access elements in a three-dimensional array.

```
#include <iostream>
using namespace std;
int main()
{
    // Declare and initialize a 3D array
    int cube[3][3][3] =
 {
        {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        },
        {
            {10, 11, 12},
            {13, 14, 15},
            {16, 17, 18}
```

```
        },
        {
            {19, 20, 21},
            {22, 23, 24},
            {25, 26, 27}
        }
    };

    // Display the 3D array
    for (int i = 0; i < 3; i++)
   {
        cout << "Depth " << i << ":" << endl;
        for (int j = 0; j < 3; j++)
   {
            for (int k = 0; k < 3; k++)
  {
                cout << cube[i][j][k] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    return 0;
}
```

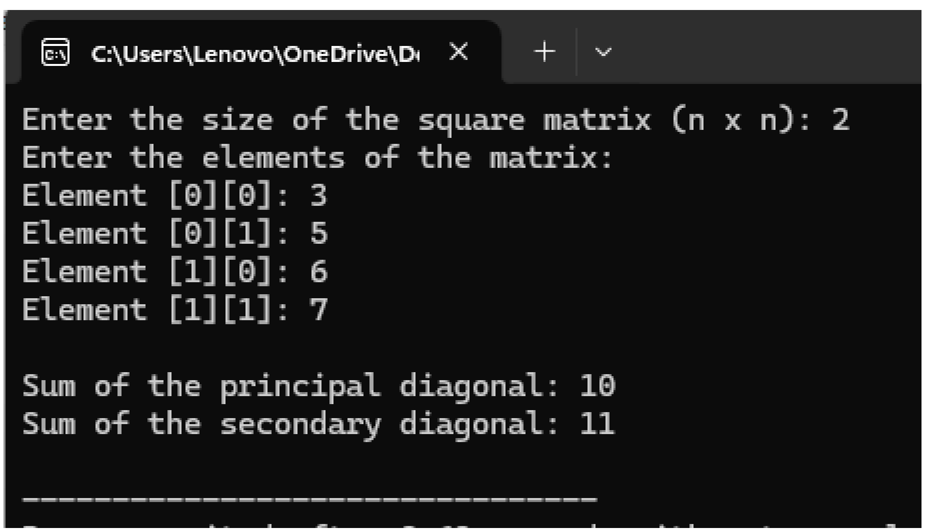


3.Create a C++ program that take sum of diagonal elements in 2 D square matrix

```
#include <iostream>

using namespace std;


int main() {

    int n;


    // Input the size of the square matrix

    cout << "Enter the size of the square matrix (n x n): ";

    cin >> n;


    int matrix[n][n];


    // Input elements of the matrix

    cout << "Enter the elements of the matrix:\n";

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

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

            cout << "Element [" << i << "][" << j << "]: ";

            cin >> matrix[i][j];

        }
```

```
    }

    // Compute the sum of the diagonals
    int principalDiagonalSum = 0, secondaryDiagonalSum = 0;

    for (int i = 0; i < n; i++) {
        principalDiagonalSum += matrix[i][i]; // Principal diagonal: matrix[i][i]
        secondaryDiagonalSum += matrix[i][n - i - 1]; // Secondary diagonal: matrix[i][n-i-1]
    }

    // Output the results
    cout << "\nSum of the principal diagonal: " << principalDiagonalSum << endl;
    cout << "Sum of the secondary diagonal: " << secondaryDiagonalSum << endl;

    return 0;
}
```

Output:

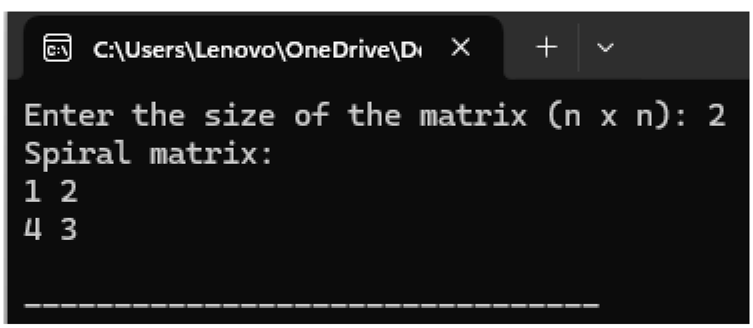


4.Create a C++ program filling a 2 D array in a spiral pattern

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

```
void fillSpiral(int n) {
    int matrix[n][n];
    int value = 1;
    int top = 0, bottom = n - 1, left = 0, right = n - 1;

    while (top <= bottom && left <= right) {
        // Fill the top row
        for (int i = left; i <= right; i++) {
            matrix[top][i] = value++;
        }
        top++;

        // Fill the right column
        for (int i = top; i <= bottom; i++) {
            matrix[i][right] = value++;
        }
        right--;

        // Fill the bottom row
        if (top <= bottom) {
            for (int i = right; i >= left; i--) {
                matrix[bottom][i] = value++;
            }
            bottom--;
        }

        // Fill the left column
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                matrix[i][left] = value++;
            }
```

```
            left++;
        }
    }

    // Output the matrix in spiral order
    cout << "Spiral matrix:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int n;
    cout << "Enter the size of the matrix (n x n): ";
    cin >> n;

    fillSpiral(n);

    return 0;
}
```



VECTORS

Vectors in C++ are dynamic arrays provided by the Standard Template Library (STL). They offer a convenient way to manage a collection of elements where the size can change dynamically. Vectors provide many functionalities that make them powerful and flexible for various programming needs.

KEY FEATURES OF VECTORS

Dynamic Sizing

Unlike static arrays, vectors can automatically resize themselves when elements are added or removed.

Random Access

Elements in a vector can be accessed using an index, similar to arrays.

Memory Management

Vectors handle memory allocation and deallocation automatically, reducing the risk of memory leaks.

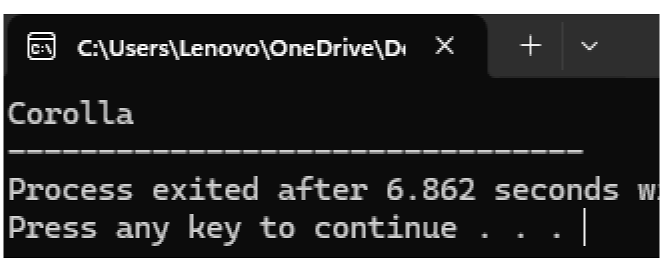
Flexibility

Vectors can store any data type, including user-defined types, and can be nested (i.e., vectors of vectors).

5.Create a C++ program that utilizes the vector container from the Standard Template Library (STL) to store a list of car brands.

```
#include <iostream>

#include <vector>

using namespace std;


int main()

{

  vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};


  // Change the value of the first element

  cars[0] = "Corolla";


  cout << cars[0];

  return 0;

}
```

Output:



Lab 04 [Stack Implementation]

STACK

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 will be the first one to be removed. Stacks are used in various applications such as expression evaluation, backtracking algorithms, and function call management.

KEY OPERATIONS

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the element from the top of the stack.
- **Peek/Top**: Retrieve the element at the top of the stack without removing it.
- **isEmpty**: Check if the stack is empty.
- **size**: Get the number of elements in the stack.

IMPLEMENTATION OF STACK USING STL LIBRARY

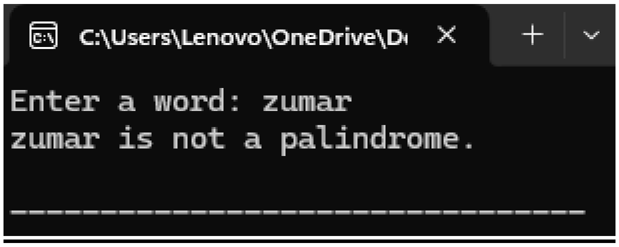
The Standard Template Library (STL) in C++ provides a built-in stack container that simplifies stack implementation.

Example:

1. **Use a stack to check if a word entered by the user is a palindrome (a word that reads the same backward and forward).**

```
#include  <iostream>

#include <stack>

using namespace std;


bool isPalindrome(const string& word)

{

   stack<char> s;


   // Push each character of the word onto the stack

   for (int i = 0; i < word.length(); i++)

   {

      s.push(word[i]);

   }
```

```
    // Check if the word reads the same forward and backward
    for (int i = 0; i < word.length(); i++)
    {
        if (word[i] != s.top())
                {
            return false;  // If characters don't match, it's not a palindrome
        }
        s.pop();
    }

    return true;
}

int main()
 {
    string word;
    cout << "Enter a word: ";
    cin >> word;

    if (isPalindrome(word))
    {
        cout << word << " is a palindrome." << endl;
    } else
    {
        cout << word << " is not a palindrome." << endl;
    }

    return 0;
}
```

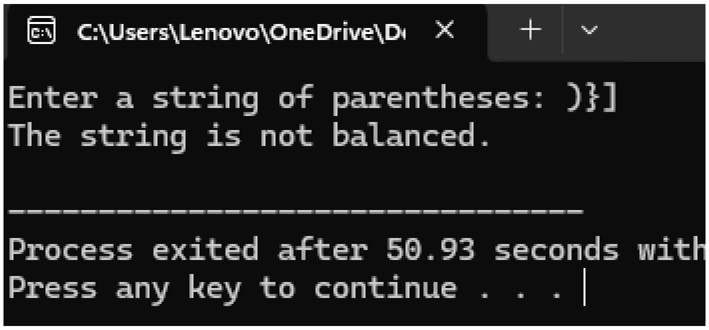


2.Write a program using a stack to check if a string of parentheses (`(), {}, []`) is balanced. For example, `(())` is balanced, but `(()` is not.

```
#include <iostream>

#include <stack>

using namespace std;

bool isBalanced(const string& str)

{

   stack<char> s;


   for (int i = 0; i < str.length(); i++)

   {

      char ch = str[i];


      // If it's an opening bracket, push it onto the stack

      if (ch == '(' || ch == '{' || ch == '[')

            {

         s.push(ch);

      }

            else if (ch == ')' || ch == '}' || ch == ']')

            {

         // If it's a closing bracket, check for balance

         if (s.empty()) return false;  // No matching opening bracket


         char top = s.top();

         s.pop();


         // Check if the top of the stack matches the corresponding opening bracket
```

```
            if ((ch == ')' && top != '(') ||
                (ch == '}' && top != '{') ||
                (ch == ']' && top != '['))
                                {
                return false;
            }
        }
    }

    return s.empty();
}
int main()
{
    string str;
    cout << "Enter a string of parentheses: ";
    cin >> str;

    if (isBalanced(str))
    {
        cout << "The string is balanced." << endl;
    }
    else
    {
        cout << "The string is not balanced." << endl;
    }
    return 0;
}
```

Output:

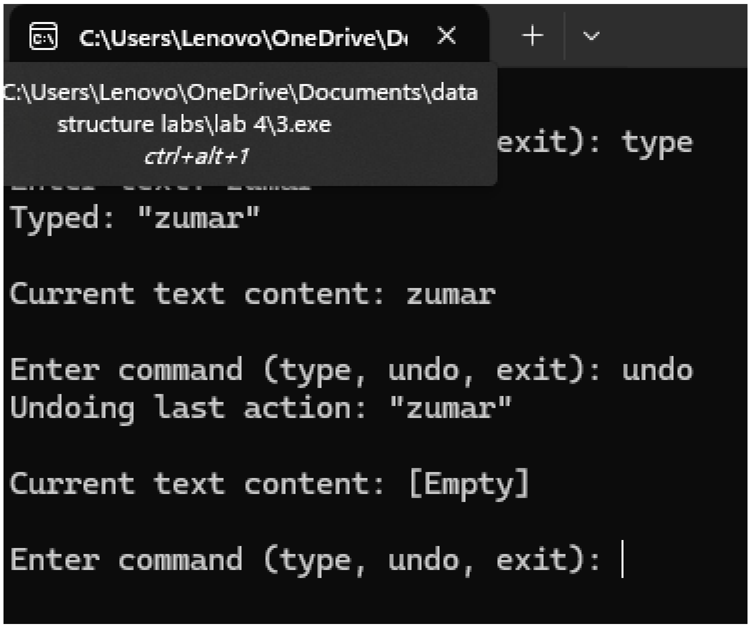


3.Implement an "Undo" function for a text editor. Each action (input by the user) is pushed to the stack, and when the user chooses "Undo," the last action is popped.

```
#include <iostream>

#include <stack>

#include <string>

using namespace std;


int main() {

    stack<string> actions;

    string command, text;


    while (true) {

        cout << "\nEnter command (type, undo, exit): ";

        cin >> command;


        if (command == "type") {

            cout << "Enter text: ";

            cin.ignore();  // Ignore the newline from previous input

            getline(cin, text);

            actions.push(text);

            cout << "Typed: \"" << text << "\"" << endl;

        }

        else if (command == "undo") {

            if (actions.empty()) {

                cout << "Nothing to undo." << endl;

            } else {
```

```
                cout << "Undoing last action: \"" << actions.top() << "\"" << endl;
                actions.pop();
            }
        }
        else if (command == "exit") {
            break;
        }
        else {
            cout << "Invalid command. Try 'type', 'undo', or 'exit'." << endl;
        }

        // Display the current content
        cout << "\nCurrent text content: ";
        if (actions.empty()) {
            cout << "[Empty]" << endl;
        } else {
            stack<string> temp = actions;
            string output;
            while (!temp.empty()) {
                output = temp.top() + " " + output;
                temp.pop();
            }
            cout << output << endl;
        }
    }

    return 0;
}
```

Output:



4.stack with minimum element tracking.

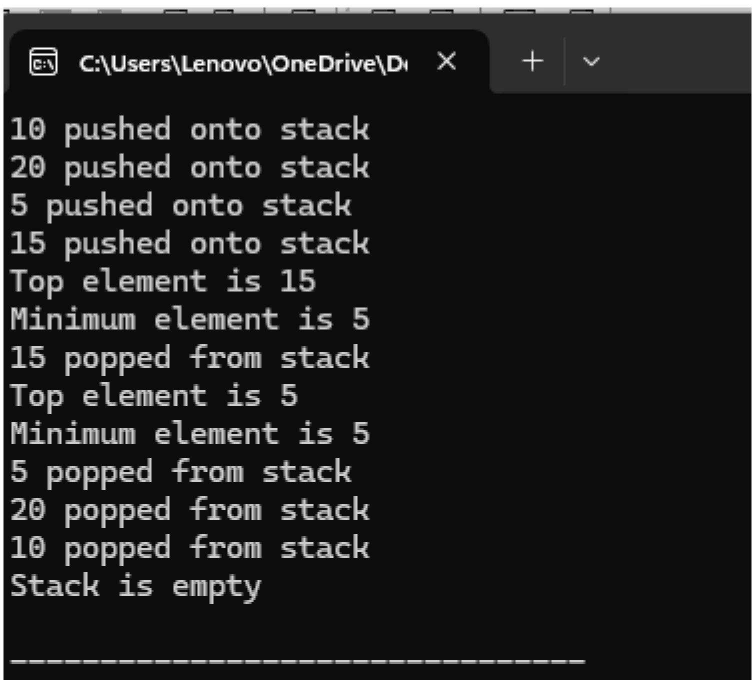
```
#include <iostream>
using namespace std;

class Stack {
private:
   int* arr;
   int* minArr;
   int top;
   int capacity;

public:
   Stack(int size = 10) {
      capacity = size;
      arr = new int[capacity];
      minArr = new int[capacity];
      top = -1;
   }

   ~Stack() {
      delete[] arr;
```

```
    delete[] minArr;
}

// Push operation
void push(int value) {
    if (top == capacity - 1) {
        cout << "Stack Overflow\n";
        return;
    }

    arr[++top] = value;

    if (top == 0) {
        minArr[top] = value;
    } else {
        minArr[top] = (value < minArr[top - 1]) ? value : minArr[top - 1];
    }

    cout << value << " pushed onto stack\n";
}

// Pop operation
void pop() {
    if (top == -1) {
        cout << "Stack Underflow\n";
        return;
    }

    cout << arr[top--] << " popped from stack\n";
}

// Peek operation
```

```
    void peek() {
        if (top == -1) {
            cout << "Stack is empty\n";
        } else {
            cout << "Top element is " << arr[top] << endl;
        }
    }

    // Get minimum element
    void getMin() {
        if (top == -1) {
            cout << "Stack is empty\n";
        } else {
            cout << "Minimum element is " << minArr[top] << endl;
        }
    }

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

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(5);
    stack.push(15);

    stack.peek();
```

```
    stack.getMin();

    stack.pop();

    stack.peek();

    stack.getMin();

    stack.pop();

    stack.pop();

    stack.pop();

    stack.getMin();


    return 0;

}
```

Output:



Lab 05 [Queue]

QUEUE

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Queues are commonly used in scenarios where order needs to be preserved, such as in scheduling algorithms, print spooling, and handling requests in a multi-user environment.

KEY OPERATIONS

- **Enqueue:** Add an element to the end of the queue.
- **Dequeue:** Remove the element from the front of the queue.
- **Front**: Retrieve the front element of the queue without removing it.
- **isEmpty**: Check if the queue is empty.
- **size**: Get the number of elements in the queue.

Example:

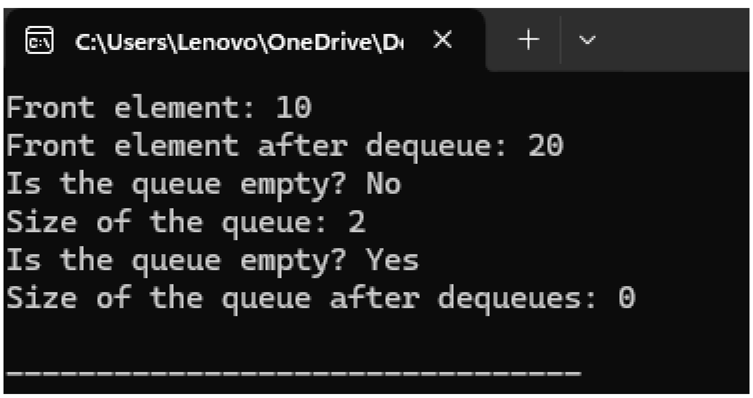
1.Create a C++ program to manage a queue of integers. Declare a queue of integers, Add elements to the queue, Display the front element of the queue, Remove an element from the front of the queue, Check if the queue is empty, Display the size of the queue.

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

class Queue {
private:
    int* arr;
    int front;
    int rear;
    int size;
    int capacity;

public:
    // Constructor to initialize the queue
    Queue(int cap) {
        capacity = cap;
```

```
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Destructor to free allocated memory
    ~Queue() {
        delete[] arr;
    }

    // Add an element to the queue
    void enqueue(int value) {
        if (size == capacity) {
            cout << "Queue is full!" << endl;
            return;
        }
        rear = (rear + 1) % capacity;  // circular increment
        arr[rear] = value;
        size++;
    }

    // Remove an element from the front of the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }
        front = (front + 1) % capacity;  // circular increment
        size--;
    }
```

```
    // Get the front element of the queue
    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return -1; // Returning -1 for empty queue
        }
        return arr[front];
    }

    // Check if the queue is empty
    bool isEmpty() {
        return size == 0;
    }

    // Get the size of the queue
    int getSize() {
        return size;
    }
};

int main() {
    Queue q(5); // Queue with capacity of 5

    q.enqueue(10); // Add elements to the queue
    q.enqueue(20);
    q.enqueue(30);

    cout << "Front element: " << q.peek() << endl; // Display the front element

    q.dequeue(); // Remove the front element
```

```
    cout << "Front element after dequeue: " << q.peek() << endl;

    cout << "Is the queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;
    cout << "Size of the queue: " << q.getSize() << endl;

    q.dequeue(); // Remove another element
    q.dequeue(); // Remove another element

    cout << "Is the queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;
    cout << "Size of the queue after dequeues: " << q.getSize() << endl;

    return 0;
}
```



2.Create a c++ program tol add names to the queue, display the front element, dequeue an element, and show the final state of the queue.

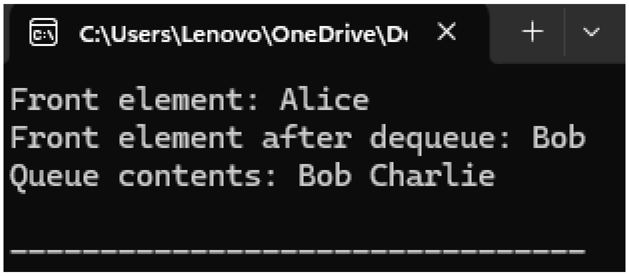
```
#include <iostream>
#include <string>
using namespace std;

class Queue {
private:
    string* arr;
    int front;
    int rear;
```

```
    int size;
    int capacity;

public:
    // Constructor to initialize the queue
    Queue(int cap) {
        capacity = cap;
        arr = new string[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Destructor to free allocated memory
    ~Queue() {
        delete[] arr;
    }

    // Add a name to the queue
    void enqueue(string name) {
        if (size == capacity) {
            cout << "Queue is full!" << endl;
            return;
        }
        rear = (rear + 1) % capacity;  // circular increment
        arr[rear] = name;
        size++;
    }

    // Remove a name from the front of the queue
    void dequeue() {
```

```
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return;
    }
    front = (front + 1) % capacity;  // circular increment
    size--;
}

// Get the front name of the queue
string peek() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return ""; // Return empty string for empty queue
    }
    return arr[front];
}

// Check if the queue is empty
bool isEmpty() {
    return size == 0;
}

// Get the size of the queue
int getSize() {
    return size;
}

// Display the queue contents
void displayQueue() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
```

```
            return;
        }
        cout << "Queue contents: ";
        for (int i = 0; i < size; i++) {
            cout << arr[(front + i) % capacity] << " ";
        }
        cout << endl;
    }
};

int main() {
    Queue q(5); // Queue with capacity of 5

    // Add names to the queue
    q.enqueue("Alice");
    q.enqueue("Bob");
    q.enqueue("Charlie");

    // Display the front element
    cout << "Front element: " << q.peek() << endl;

    // Remove the front element (dequeue)
    q.dequeue();
    cout << "Front element after dequeue: " << q.peek() << endl;

    // Show the final state of the queue
    q.displayQueue();

    return 0;
}
```

Output:



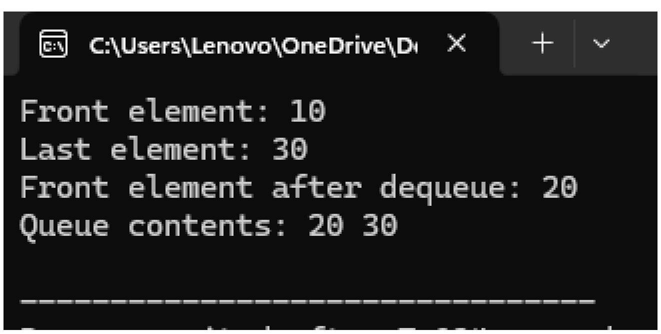
3.Create a c++ program to add numbers to the queue, display the front and last elements, dequeue an element, and show the final state of the queue.

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

class Queue {
private:
    int* arr;
    int front;
    int rear;
    int size;
    int capacity;

public:
    // Constructor to initialize the queue
    Queue(int cap) {
        capacity = cap;
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Destructor to free allocated memory
    ~Queue() {
        delete[] arr;
    }
```

```
// Add a number to the queue
void enqueue(int num) {
    if (size == capacity) {
        cout << "Queue is full!" << endl;
        return;
    }
    rear = (rear + 1) % capacity;  // circular increment
    arr[rear] = num;
    size++;
}

// Remove a number from the front of the queue
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return;
    }
    front = (front + 1) % capacity;  // circular increment
    size--;
}

// Get the front element of the queue
int peekFront() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return -1; // Return -1 for empty queue
    }
    return arr[front];
}
```

```
// Get the last element of the queue
int peekLast() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return -1; // Return -1 for empty queue
    }
    return arr[rear];
}

// Check if the queue is empty
bool isEmpty() {
    return size == 0;
}

// Get the size of the queue
int getSize() {
    return size;
}

// Display the queue contents
void displayQueue() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
        return;
    }
    cout << "Queue contents: ";
    for (int i = 0; i < size; i++) {
        cout << arr[(front + i) % capacity] << " ";
    }
    cout << endl;
}
```

```
};

int main() {
    Queue q(5); // Queue with a capacity of 5

    // Add numbers to the queue
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    // Display the front and last elements
    cout << "Front element: " << q.peekFront() << endl;
    cout << "Last element: " << q.peekLast() << endl;

    // Remove the front element (dequeue)
    q.dequeue();
    cout << "Front element after dequeue: " << q.peekFront() << endl;

    // Show the final state of the queue
    q.displayQueue();

    return 0;
}
```



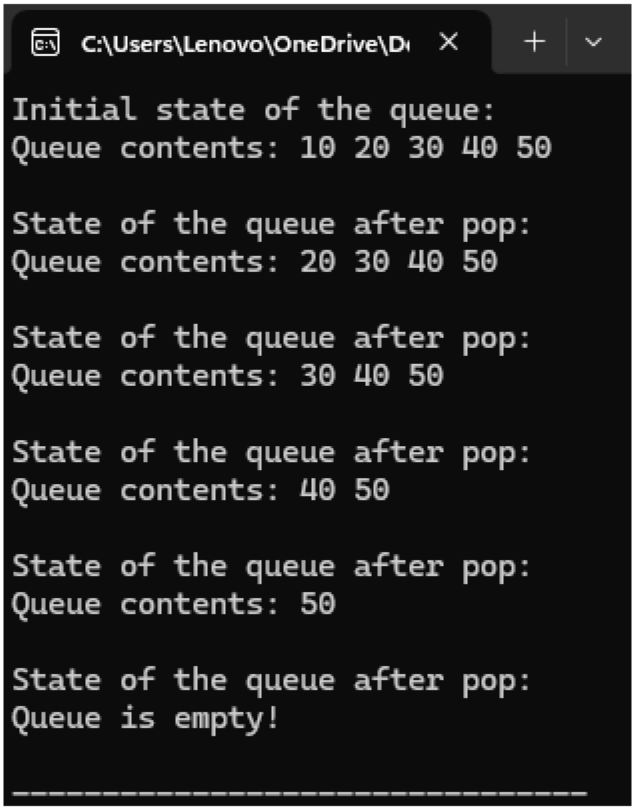
4.Create a c++ program to manage a queue of integers, add several elements to the queue, display the elements, and then repeatedly pop elements while displaying the state of the queue after each pop operation.

```
#include <iostream>
```

```
using namespace std;

class Queue {
private:
    int* arr;
    int front;
    int rear;
    int size;
    int capacity;

public:
    // Constructor to initialize the queue
    Queue(int cap) {
        capacity = cap;
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Destructor to free allocated memory
    ~Queue() {
        delete[] arr;
    }

    // Add an element to the queue
    void enqueue(int num) {
        if (size == capacity) {
            cout << "Queue is full!" << endl;
            return;
        }
        rear = (rear + 1) % capacity;  // circular increment
        arr[rear] = num;
        size++;
    }

    // Remove an element from the front of the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }
        front = (front + 1) % capacity;  // circular increment
        size--;
    }
```

```
    // Check if the queue is empty
    bool isEmpty() {
        return size == 0;
    }

    // Display the queue contents
    void displayQueue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }
        cout << "Queue contents: ";
        for (int i = 0; i < size; i++) {
            cout << arr[(front + i) % capacity] << " ";
        }
        cout << endl;
    }

    // Get the current size of the queue
    int getSize() {
        return size;
    }
};

int main() {
    Queue q(5); // Queue with a capacity of 5

    // Add several elements to the queue
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);

    // Display the elements of the queue
    cout << "Initial state of the queue:" << endl;
    q.displayQueue();

    // Repeatedly pop elements and display the state of the queue
    while (!q.isEmpty()) {
        cout << "\nState of the queue after pop:" << endl;
        q.dequeue();
        q.displayQueue();
    }
```

```
    return 0;
}
```


C:\Users\Lenovo\OneDrive\D
Initial state of the queue:
Queue contents: 10 20 30 40 50
State of the queue after pop:
Queue contents: 20 30 40 50
State of the queue after pop:
Queue contents: 30 40 50
State of the queue after pop:
Queue contents: 40 50
State of the queue after pop:
Queue contents: 50
State of the queue after pop:
Queue is empty!

Lab 06 Linked List Implementation]

<u>LINK LIST</u>

A linked list is a linear data structure where elements, called nodes, are stored in separate objects rather than in a contiguous memory location like arrays. Each node contains two parts: a data part that stores the value and a pointer part that points to the next node in the sequence. The linked list allows for efficient insertion and deletion of elements.

<u>Types of Linked Lists</u>

- ***Singly Linked List.***
- ***Doubly Linked List.***
- ***Circular Linked List.***

<u>Singly Linked List</u>

A singly linked list contains nodes that point to the next node in the list. Here's how to implement a basic linked list in C++.

<u>Doubly Linked List</u>

A doubly linked list is a type of linked list in which each node contains pointers to both the previous and next nodes. This allows for more efficient traversal in both directions (forward and backward) and simplifies certain operations such as deletion.

<u>Circular Linked List</u>

A circular linked list is a variation of the linked list where the last node points back to the first node, forming a circle. This structure allows for efficient traversal and can be used in scenarios where a cyclic iteration of elements is needed, such as in round-robin scheduling.

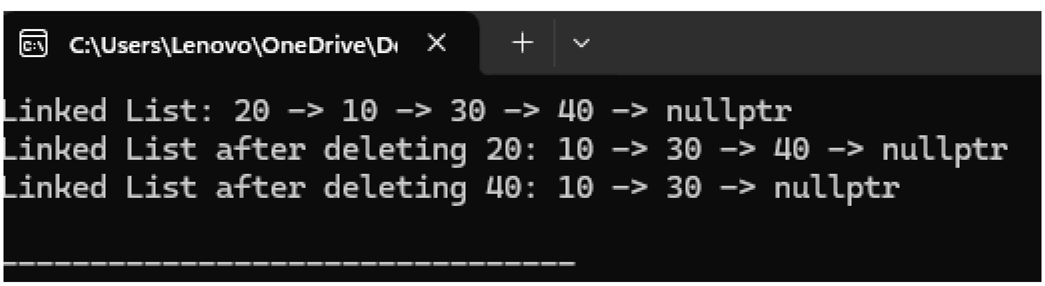
1. ***Create a c++ program in which singly linked list contains nodes that point to the next node in the list.***

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


struct Node {
   int data;
   Node* next;
};
class LinkedList {
private:
```

```
    Node* head;
public:
    LinkedList() { head = nullptr; }

    // Function to add a new node at the beginning
    void insertAtBeginning(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = head;
        head = newNode;
    }

    // Function to add a new node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = nullptr;
        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }

    // Function to delete a node by its value
    void deleteNode(int value) {
        if (head == nullptr) return;
```

```cpp
        if (head->data == value) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        Node* temp = head;
        while (temp->next != nullptr && temp->next->data != value) {
            temp = temp->next;
        }

        if (temp->next == nullptr) return;

        Node* nodeToDelete = temp->next;
        temp->next = temp->next->next;
        delete nodeToDelete;
    }

    // Function to display the linked list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "nullptr" << endl;
    }
};
int main() {
```

```
    LinkedList list;
    list.insertAtBeginning(10);
    list.insertAtBeginning(20);
    list.insertAtEnd(30);
    list.insertAtEnd(40);

    cout << "Linked List: ";
    list.display();

    list.deleteNode(20);
    cout << "Linked List after deleting 20: ";
    list.display();

    list.deleteNode(40);
    cout << "Linked List after deleting 40: ";
    list.display();

    return 0;
}
```

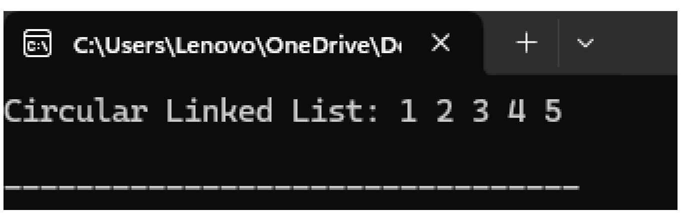


2.Write a c++ program to create a circular linked list with values.

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

// Node structure for circular linked list
```

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

// Function to create a circular linked list
void create(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        head = newNode;
        head->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
}

// Function to display elements of the circular linked list
void display(Node* head) {
    if (head == nullptr) return;

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
```

```
        } while (temp != head);
        cout << endl;
    }

    int main() {
        Node* head = nullptr;

        // Create circular linked list with values
        create(head, 1);
        create(head, 2);
        create(head, 3);
        create(head, 4);
        create(head, 5);

        // Display elements of the circular linked list
        cout << "Circular Linked List: ";
        display(head);

        return 0;
    }
```



3. Write a function to delete a node by its value in a circular linked list.

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

// Define a structure for the circular linked list node
struct Node {
    int data;  // The data part of the node
    Node* next;  // Pointer to the next node
```

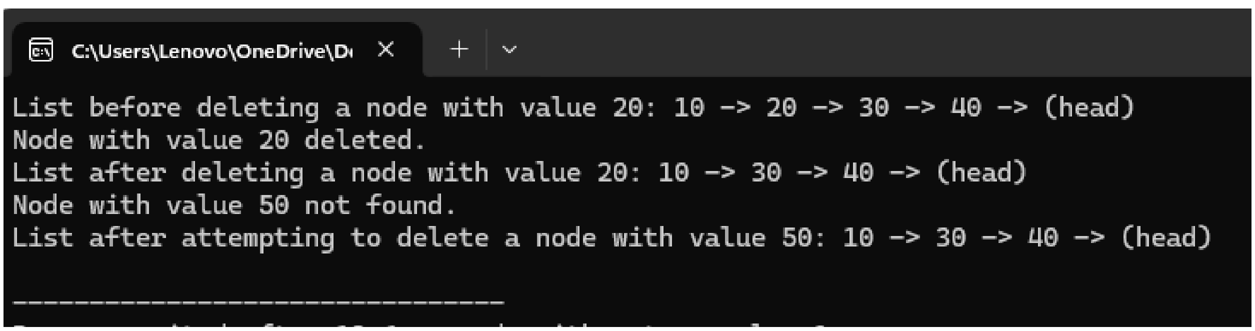
```
    // Constructor to create a new node
    Node(int val) : data(val), next(nullptr) {}
};

// Class to represent the Circular Linked List
class CircularLinkedList {
private:
    Node* head;  // Pointer to the first node in the list

public:
    // Constructor to initialize the list
    CircularLinkedList() : head(nullptr) {}

    // Function to add a node at the end of the list (to form the circular nature)
    void append(int data) {
        Node* newNode = new Node(data);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;  // The last node points to the head to form a circle
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;  // Make the list circular
        }
    }

    // Function to delete a node by its value in the circular linked list
```

```
void deleteByValue(int value) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    // If the node to delete is the head node
    if (head->data == value) {
        // If there's only one node
        if (head->next == head) {
            delete head;
            head = nullptr;
            cout << "Node with value " << value << " deleted. List is now empty." << endl;
            return;
        }

        // If there are multiple nodes, update the last node's next pointer
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }

        // Temp now points to the last node
        temp->next = head->next;
        Node* oldHead = head;
        head = head->next;
        delete oldHead;
        cout << "Node with value " << value << " deleted." << endl;
        return;
    }
```

```
    // If the node to delete is not the head node
    Node* current = head;
    Node* previous = nullptr;

    // Traverse the list to find the node
    do {
        previous = current;
        current = current->next;
        if (current->data == value) {
            // Remove the node from the list
            previous->next = current->next;
            delete current;
            cout << "Node with value " << value << " deleted." << endl;
            return;
        }
    } while (current != head);

    // If the value is not found in the list
    cout << "Node with value " << value << " not found." << endl;
}

// Function to display the list
void display() {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }

    Node* temp = head;
    do {
        cout << temp->data << " -> ";
```

```
            temp = temp->next;
        } while (temp != head);
        cout << "(head)" << endl;
    }

    // Destructor to free up the list memory
    ~CircularLinkedList() {
        if (head == nullptr) return;

        Node* temp = head;
        do {
            Node* nextNode = temp->next;
            delete temp;
            temp = nextNode;
        } while (temp != head);
    }
};

int main() {
    CircularLinkedList list;

    // Add some nodes to the circular linked list
    list.append(10);
    list.append(20);
    list.append(30);
    list.append(40);

    // Display the list before deletion
    cout << "List before deleting a node with value 20: ";
    list.display();
```

```
    // Delete the node with value 20
    list.deleteByValue(20);

    // Display the list after deletion
    cout << "List after deleting a node with value 20: ";
    list.display();

    // Try deleting a node that does not exist
    list.deleteByValue(50);

    // Display the list again
    cout << "List after attempting to delete a node with value 50: ";
    list.display();

    return 0;
}
```



```
C:\Users\Lenovo\OneDrive\D
List before deleting a node with value 20: 10 -> 20 -> 30 -> 40 -> (head)
Node with value 20 deleted.
List after deleting a node with value 20: 10 -> 30 -> 40 -> (head)
Node with value 50 not found.
List after attempting to delete a node with value 50: 10 -> 30 -> 40 -> (head)

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

4.Create a c++ program in which doubly linked list allows traversal in both directions, making it more versatile than a singly linked list.

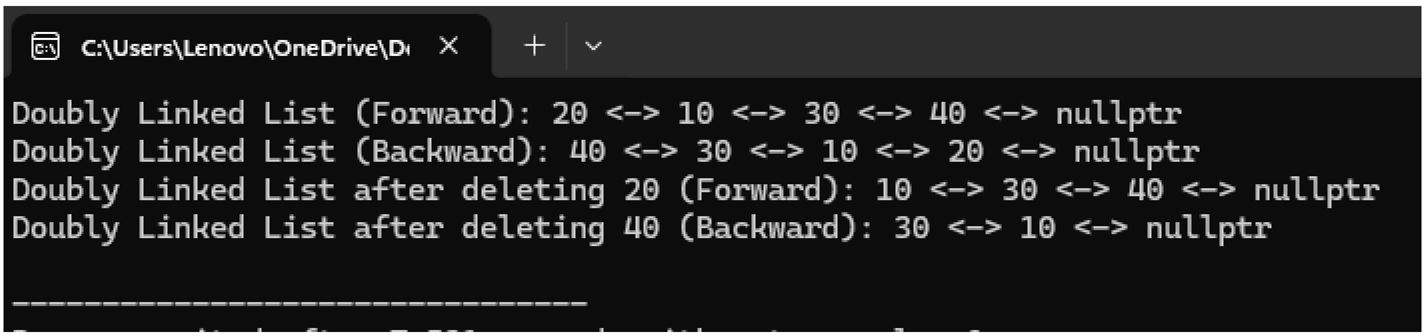
```
#include <iostream>
using namespace std;

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

```
};
class DoublyLinkedList {
private:
    Node* head;
public:
    DoublyLinkedList() { head = nullptr; }

    // Function to add a new node at the beginning
    void insertAtBeginning(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = head;
        newNode->prev = nullptr;
        if (head != nullptr) {
            head->prev = newNode;
        }
        head = newNode;
    }

    // Function to add a new node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = nullptr;
        if (head == nullptr) {
            newNode->prev = nullptr;
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != nullptr) {
```

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

// Function to delete a node by its value
void deleteNode(int value) {
    Node* temp = head;
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }
    if (temp == nullptr) return; // Node not found
    if (temp->prev != nullptr) {
        temp->prev->next = temp->next;
    } else {
        head = temp->next;
    }
    if (temp->next != nullptr) {
        temp->next->prev = temp->prev;
    }
    delete temp;
}

// Function to display the linked list from beginning to end
void displayForward() {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
```

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

    // Function to display the linked list from end to beginning
    void displayBackward() {
        if (head == nullptr) return;
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        while (temp != nullptr) {
            cout << temp->data << " <-> ";
            temp = temp->prev;
        }
        cout << "nullptr" << endl;
    }
};
int main() {
    DoublyLinkedList list;
    list.insertAtBeginning(10);
    list.insertAtBeginning(20);
    list.insertAtEnd(30);
    list.insertAtEnd(40);

    cout << "Doubly Linked List (Forward): ";
    list.displayForward();

    cout << "Doubly Linked List (Backward): ";
    list.displayBackward();

    list.deleteNode(20);
```

```
    cout << "Doubly Linked List after deleting 20 (Forward): ";
    list.displayForward();

    list.deleteNode(40);
    cout << "Doubly Linked List after deleting 40 (Backward): ";
    list.displayBackward();

    return 0;
}
```



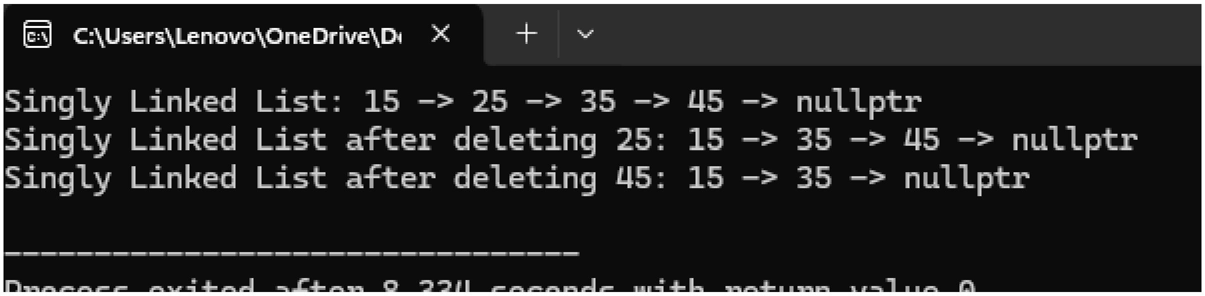
5.*Create a c++ single linked list program to delete a node by its value.*

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

struct Node {
    int data;
    Node* next;
};
class SinglyLinkedList {
private:
    Node* head;
public:
    SinglyLinkedList() { head = nullptr; }

    // Function to add a new node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node();
```

```
    newNode->data = value;
    newNode->next = nullptr;
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to delete a node by its value
void deleteNode(int value) {
    if (head == nullptr) return;

    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr && temp->next->data != value) {
        temp = temp->next;
    }

    if (temp->next == nullptr) return;
```

```cpp
        Node* nodeToDelete = temp->next;
        temp->next = temp->next->next;
        delete nodeToDelete;
    }

    // Function to display the linked list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "nullptr" << endl;
    }
};
int main() {
    SinglyLinkedList list;
    list.insertAtEnd(15);
    list.insertAtEnd(25);
    list.insertAtEnd(35);
    list.insertAtEnd(45);

    cout << "Singly Linked List: ";
    list.display();

    list.deleteNode(25);
    cout << "Singly Linked List after deleting 25: ";
    list.display();

    list.deleteNode(45);
    cout << "Singly Linked List after deleting 45: ";
```

```
    list.display();


    return 0;
}
```



6..How would you delete a node at a specific position in a doubly linked list? Show it in code.

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


// Define a structure for the doubly linked list node
struct Node {
    int data;  // The data part of the node
    Node* next;  // Pointer to the next node
    Node* prev;  // Pointer to the previous node


    // Constructor to create a new node
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};


// Class to represent the Doubly Linked List
class DoublyLinkedList {
private:
    Node* head;  // Pointer to the first node in the list


public:
    // Constructor to initialize the list
```

```
DoublyLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list
void append(int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

// Function to delete a node at a specific position (1-based index)
void deleteAtPosition(int position) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    Node* temp = head;
    int count = 1;

    // If position is 1, we delete the head node
    if (position == 1) {
        head = temp->next;
        if (head != nullptr) {
```

```
        head->prev = nullptr;
    }
    delete temp;
    cout << "Node at position " << position << " deleted." << endl;
    return;
}

// Traverse the list to find the node at the given position
while (temp != nullptr && count < position) {
    temp = temp->next;
    count++;
}

// If the position is greater than the number of nodes
if (temp == nullptr) {
    cout << "Position out of bounds." << endl;
    return;
}

// If the node is not the last one
if (temp->next != nullptr) {
    temp->next->prev = temp->prev;
}
// If the node is not the first one
if (temp->prev != nullptr) {
    temp->prev->next = temp->next;
}

// Delete the node
delete temp;
cout << "Node at position " << position << " deleted." << endl;
```

```
    }

    // Function to display the list
    void display() {
        if (head == nullptr) {
            cout << "List is empty." << endl;
            return;
        }

        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " <-> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }

    // Destructor to free up the list memory
    ~DoublyLinkedList() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main() {
    DoublyLinkedList list;

    // Add some nodes to the list
```

```
    list.append(10);
    list.append(20);
    list.append(30);
    list.append(40);

    // Display the list before deletion
    cout << "List before deleting a node at position: ";
    list.display();

    // Delete the node at position 3
    list.deleteAtPosition(3);

    // Display the list after deletion
    cout << "List after deleting the node at position 3: ";
    list.display();

    // Try to delete a node at an out-of-bounds position
    list.deleteAtPosition(10);

    // Display the list after attempting to delete at an out-of-bounds position
    cout << "List after attempting to delete at position 10: ";
    list.display();

    return 0;
}
```

```
C:\Users\Lenovo\OneDrive\D
List before deleting a node at position: 10 <-> 20 <-> 30 <-> 40 <-> NULL
Node at position 3 deleted.
List after deleting the node at position 3: 10 <-> 20 <-> 40 <-> NULL
Position out of bounds.
List after attempting to delete at position 10: 10 <-> 20 <-> 40 <-> NULL

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

Lab 07[binary search tree]

BST (Binary Search Tree)

A **Binary Search Tree (BST)** is a special kind of binary tree where each node has a maximum of two children. BSTs are used to maintain a sorted order of elements, allowing efficient search, insertion, and deletion operations. The left subtree of a node contains only nodes with values less than the node’s value, while the right subtree contains only nodes with values greater than the node’s value.

KEY OPERATIONS IN BST

A Binary Search Tree (BST) is a node-based data structure that maintains a sorted order of elements and supports efficient search, insertion, and deletion operations.

//key operations//

- Insertion.
- Search.
- Deletion.
- In-order traversal.

1.Create a C++ propgram to insert value in binary search tree.

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
class BinarySearchTree {
private:
    Node* root;

    // Helper function to insert a new node
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
```

```
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

public:
    BinarySearchTree() : root(nullptr) {}

    // Function to insert a value into the tree
    void insert(int value) {
        root = insert(root, value);
    }

    // Function to perform in-order traversal
    void inOrderTraversal(Node* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            cout << node->data << " ";
            inOrderTraversal(node->right);
        }
    }

    // Function to initiate in-order traversal
    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
```

```
        }
    };
    int main() {
        BinarySearchTree bst;
        bst.insert(50);
        bst.insert(30);
        bst.insert(70);
        bst.insert(20);
        bst.insert(40);
        bst.insert(60);
        bst.insert(80);

        cout << "In-order traversal: ";
        bst.inOrderTraversal();

        return 0;
    }
```

2.Create a C++ program to delete a value in binary seach tree.

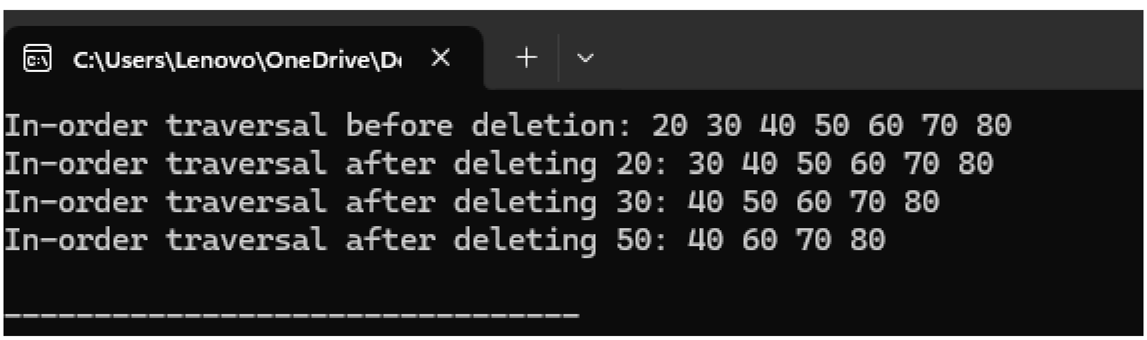
```
#include <iostream>
using namespace std;

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

```
class BinarySearchTree {
private:
    Node* root;

    // Helper function to insert a new node
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Helper function to find the in-order successor
    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) {
            current = current->left;
        }
        return current;
    }

    // Helper function to delete a node
    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 {
        // 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 in-order successor
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Helper function for in-order traversal
void inOrderTraversal(Node* node) {
    if (node != nullptr) {
        inOrderTraversal(node->left);
        cout << node->data << " ";
        inOrderTraversal(node->right);
    }
```

```
    }

public:
    BinarySearchTree() : root(nullptr) {}

    // Function to insert a value into the tree
    void insert(int value) {
        root = insert(root, value);
    }

    // Function to delete a value from the tree
    void deleteNode(int value) {
        root = deleteNode(root, value);
    }

    // Function to perform in-order traversal
    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
    }
};
int main() {
    BinarySearchTree bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);
```

```
    cout << "In-order traversal before deletion: ";
    bst.inOrderTraversal();


    bst.deleteNode(20);
    cout << "In-order traversal after deleting 20: ";
    bst.inOrderTraversal();


    bst.deleteNode(30);
    cout << "In-order traversal after deleting 30: ";
    bst.inOrderTraversal();


    bst.deleteNode(50);
    cout << "In-order traversal after deleting 50: ";
    bst.inOrderTraversal();


    return 0;
}
```

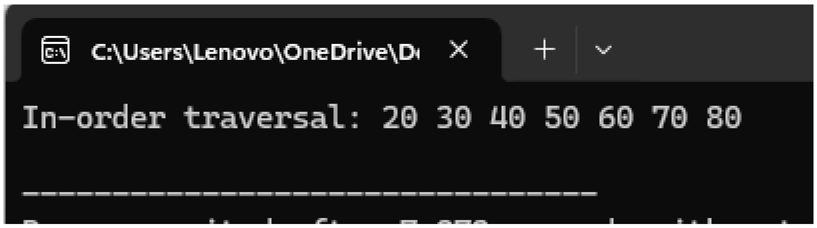


3.Create a C++ program for in-order traversal in binary search tree.

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

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
class BinarySearchTree {
private:
```

```
    Node* root;

    // Helper function to insert a new node
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Helper function for in-order traversal
    void inOrderTraversal(Node* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            cout << node->data << " ";
            inOrderTraversal(node->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    // Function to insert a value into the tree
    void insert(int value) {
        root = insert(root, value);
    }

    // Function to perform in-order traversal
    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
    }
};
int main() {
    BinarySearchTree bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
```

```
        bst.insert(60);
        bst.insert(80);

        cout << "In-order traversal: ";
        bst.inOrderTraversal();

        return 0;
    }
```



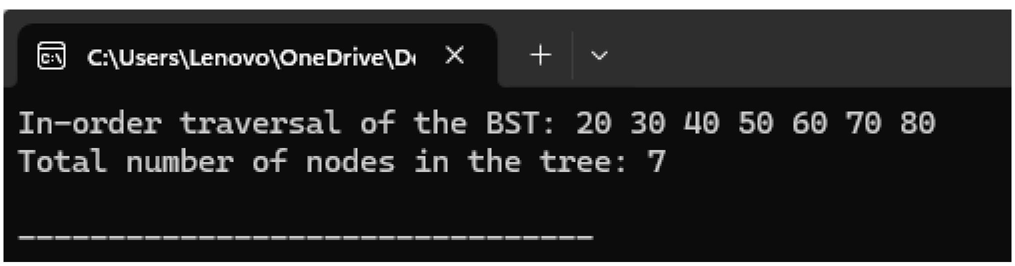
4. Write a program to count all the nodes in a binary search tree.

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

// Structure for the Node of the Binary Search Tree
struct Node {
   int data;   // Data of the node
   Node* left; // Pointer to the left child
   Node* right; // Pointer to the right child

   // Constructor to create a new node
   Node(int value) {
      data = value;
      left = right = nullptr;
   }
};

// Class for Binary Search Tree
class BST {
private:
```

```
    Node* root; // Root node of the tree

    // Helper function to count nodes recursively
    int countNodesHelper(Node* node) {
        if (node == nullptr) {
            return 0; // Base case: if the node is NULL, return 0
        }
        // Recursively count the nodes in the left and right subtrees and add 1 for the current node
        return 1 + countNodesHelper(node->left) + countNodesHelper(node->right);
    }

public:
    // Constructor to initialize the tree
    BST() {
        root = nullptr;
    }

    // Function to insert a node into the Binary Search Tree
    void insert(int value) {
        root = insertHelper(root, value);
    }

    // Helper function to insert a node
    Node* insertHelper(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value); // Create a new node if the position is empty
        }
        if (value < node->data) {
            node->left = insertHelper(node->left, value); // Insert in the left subtree
        } else {
            node->right = insertHelper(node->right, value); // Insert in the right subtree
```

```
        }
        return node;
    }

    // Public function to count all nodes in the tree
    int countNodes() {
        return countNodesHelper(root);
    }

    // Function to display the tree (In-order traversal for visualization)
    void inorder() {
        inorderHelper(root);
        cout << endl;
    }

    // Helper function for In-order traversal
    void inorderHelper(Node* node) {
        if (node != nullptr) {
            inorderHelper(node->left);
            cout << node->data << " ";
            inorderHelper(node->right);
        }
    }
};

int main() {
    BST tree;

    // Inserting nodes into the BST
    tree.insert(50);
    tree.insert(30);
```

```
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    // Displaying the tree using In-order traversal
    cout << "In-order traversal of the BST: ";
    tree.inorder();

    // Counting the nodes in the BST
    int nodeCount = tree.countNodes();
    cout << "Total number of nodes in the tree: " << nodeCount << endl;

    return 0;
}
```



5.Write a program to check if there are duplicate values in a binary search tree.

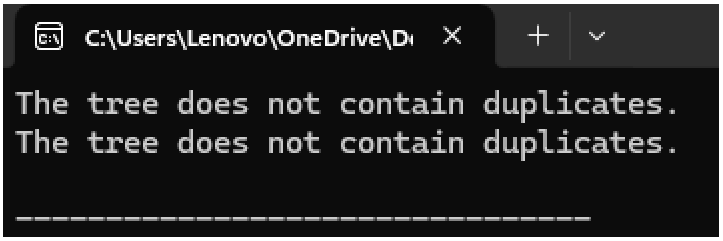
```
#include <iostream>
#include <unordered_set>
using namespace std;

// Structure for the Node of the Binary Search Tree
struct Node {
    int data;   // Data of the node
    Node* left; // Pointer to the left child
    Node* right; // Pointer to the right child
```

```
    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Class for Binary Search Tree
class BST {
private:
    Node* root; // Root node of the tree

    // Helper function to insert a node into the Binary Search Tree
    Node* insertHelper(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value); // Create a new node if the position is empty
        }
        if (value < node->data) {
            node->left = insertHelper(node->left, value); // Insert in the left subtree
        } else if (value > node->data) {
            node->right = insertHelper(node->right, value); // Insert in the right subtree
        }
        // If value is equal to node's data, it means it's a duplicate in the BST
        return node;
    }

    // Helper function to perform In-order traversal and check for duplicates
    bool inorderHelper(Node* node, unordered_set<int>& visited) {
        if (node == nullptr) {
            return true;
        }
```

```
        // Traverse left subtree
        if (!inorderHelper(node->left, visited)) {
            return false;
        }

        // Check if the current node's data is already in the set
        if (visited.find(node->data) != visited.end()) {
            return false; // Duplicate found
        }

        // Add the current node's data to the set
        visited.insert(node->data);

        // Traverse right subtree
        return inorderHelper(node->right, visited);
    }

public:
    // Constructor to initialize the tree
    BST() {
        root = nullptr;
    }

    // Function to insert a node into the Binary Search Tree
    void insert(int value) {
        root = insertHelper(root, value);
    }

    // Function to check if the BST contains duplicates
    bool containsDuplicates() {
```

```
        unordered_set<int> visited;
        return inorderHelper(root, visited);
    }
};

int main() {
    BST tree;

    // Inserting nodes into the BST
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    // Check if the tree contains duplicates
    if (tree.containsDuplicates()) {
        cout << "The tree does not contain duplicates." << endl;
    } else {
        cout << "The tree contains duplicates." << endl;
    }

    // Insert a duplicate value into the tree
    tree.insert(40);

    // Check again if the tree contains duplicates
    if (tree.containsDuplicates()) {
        cout << "The tree does not contain duplicates." << endl;
    } else {
```

```
        cout << "The tree contains duplicates." << endl;
    }

    return 0;
}
```



6. How can you search for a specific value in a binary search tree?

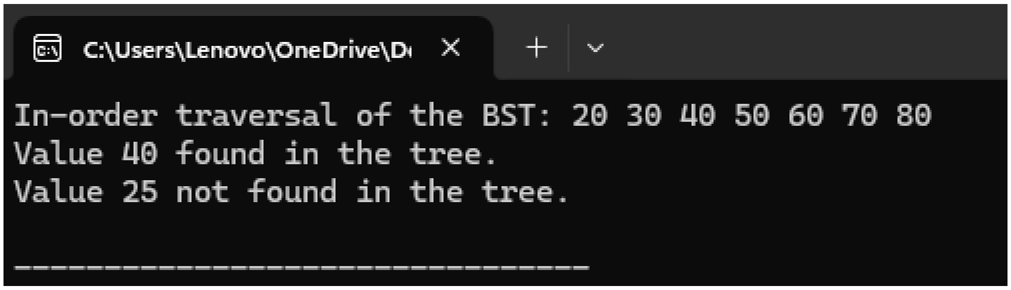
```
#include <iostream>
using namespace std;

// Structure for the Node of the Binary Search Tree
struct Node {
    int data;   // Data of the node
    Node* left; // Pointer to the left child
    Node* right; // Pointer to the right child

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Class for Binary Search Tree
class BST {
private:
    Node* root; // Root node of the tree
```

```
    // Helper function to insert a node into the Binary Search Tree
    Node* insertHelper(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value); // Create a new node if the position is empty
        }
        if (value < node->data) {
            node->left = insertHelper(node->left, value); // Insert in the left subtree
        } else {
            node->right = insertHelper(node->right, value); // Insert in the right subtree
        }
        return node;
    }

    // Helper function to search for a value in the Binary Search Tree
    Node* searchHelper(Node* node, int value) {
        if (node == nullptr) {
            return nullptr; // Base case: the value is not found
        }
        if (node->data == value) {
            return node; // The value is found
        }
        if (value < node->data) {
            return searchHelper(node->left, value); // Search in the left subtree
        } else {
            return searchHelper(node->right, value); // Search in the right subtree
        }
    }

public:
    // Constructor to initialize the tree
    BST() {
```

```
        root = nullptr;
    }

    // Function to insert a node into the Binary Search Tree
    void insert(int value) {
        root = insertHelper(root, value);
    }

    // Function to search for a value in the Binary Search Tree
    bool search(int value) {
        Node* result = searchHelper(root, value);
        return result != nullptr; // If the result is not null, the value was found
    }

    // Function to display the tree (In-order traversal for visualization)
    void inorder() {
        inorderHelper(root);
        cout << endl;
    }

    // Helper function for In-order traversal
    void inorderHelper(Node* node) {
        if (node != nullptr) {
            inorderHelper(node->left);
            cout << node->data << " ";
            inorderHelper(node->right);
        }
    }
};

int main() {
```

```
    BST tree;

    // Inserting nodes into the BST
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    // Displaying the tree using In-order traversal
    cout << "In-order traversal of the BST: ";
    tree.inorder();

    // Searching for a specific value in the BST
    int valueToSearch = 40;
    if (tree.search(valueToSearch)) {
        cout << "Value " << valueToSearch << " found in the tree." << endl;
    } else {
        cout << "Value " << valueToSearch << " not found in the tree." << endl;
    }

    // Searching for another value in the BST
    valueToSearch = 25;
    if (tree.search(valueToSearch)) {
        cout << "Value " << valueToSearch << " found in the tree." << endl;
    } else {
cout << "Value " << valueToSearch << " not found in the tree." << endl;
    }
```

```
    return 0;
}
```


C:\Users\Lenovo\OneDrive\D
In-order traversal of the BST: 20 30 40 50 60 70 80
Value 40 found in the tree.
Value 25 not found in the tree.
