LAB MANUAL

SUBMITTED TO

MAM. IRSHA QURESHI

SUBMITTED BY

MUNEEB SAJID

SUBJECT:

Data Structures & Algorithm

COURSE CODE:

CS-216

DEPARTEMENT:

BS-Computer Science

Labs No:	Table of Contents	Page No
01	**Introduction**	**03**
02	**Arrays Implementation**	**07**
03	**Arrays practice**	**16**
04	**MULTI-DIMENSIONAL ARRAYS**	**22**
05	**Vectors**	**28**
06	**Stack Implementation**	**33**
07	**QUEUE IMPLEMENTATION**	**47**
08	**Link List Implementation**	**60**
09	**Binary Search Tree**	**75**

Lab No 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).
- **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;

int dereferencedValue = *ptr;
```

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

EXAMPLE

```
#include <iostream>

using namespace std;

int main()

{

//Declare and initialize an array of integers with 5 elements

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

//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 No 02

Arrays Implementation

1. **create a C++ program that allows the user to input subject names and their corresponding marks, then displays the entered information.**

Code:

```
#include <iostream>

using namespace std;

int main()

{

  // Array to store subject names

  string subjects[5] = {"English", "AI", "Dsa", "urdu", "islamiyat"};

   // Array to store the corresponding numbers/marks for the subjects

  int numbers[5];

  for(int m = 0; m < 5; m++)

  {

    // Prompting user to enter the subject name

    cout << "Enter Subject: ";

    cin >> subjects[m];


    // Prompting user to enter the marks for the subject

    cout << "Enter Numbers: ";

    cin >> numbers[m];
```

```
    }



    // Loop to display the subjects and their corresponding marks

    for(int m = 0; m < 5; m++)

    {

        // Displaying the subject name

        cout << "Subject: " << subjects[m];

            // Displaying the marks obtained for the subject

        cout << " Numbers obtained: " << numbers[m] << endl;

    }

    return 0;

}
```

Output

```
Enter Subject: English
Enter Numbers: 40
Enter Subject: urdu
Enter Numbers: 89
Enter Subject: Dsa
Enter Numbers: 67
Enter Subject: ai
Enter Numbers: 90
Enter Subject: islamiyat
Enter Numbers: 79
Subject: English Numbers obtained: 40
Subject: urdu Numbers obtained: 89
Subject: Dsa Numbers obtained: 67
Subject: ai Numbers obtained: 90
Subject: islamiyat Numbers obtained: 79
```

2. **Create a C++ program that manages a customer's account by calculating the transaction history over 5 months.**

Code

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

int main() {
    // Initial balance and variable to store updated transaction history
    int transactionHistory, currentBalance = 50000;

    // Arrays to store months, expenses, and incomes
    string months[5] = {"January", "February", "March", "April", "May"};
    int expenses[5] = {200, 307, 1500, 400, 250};
    int incomes[5] = {800, 700, 900, 1000, 1200};

    // Variables to store customer details
    string customerName, accountNumber;

    // Prompting the user to enter the customer's name
    cout << "Enter Customer Name: ";
    cin >> customerName;

    // Prompting the user to enter the account number
```

```
   cout << "Enter Account Number: ";
   cin >> accountNumber;


   // Displaying customer details
   cout << "Customer Name: " << customerName << endl;
   cout << "Account Number: " << accountNumber << endl;
   cout << "Transaction History" << endl;


   // Loop to calculate and display transaction history for each month
   for (int i = 0; i < 5; i++)
  {
      // Calculate the new balance after accounting for expenses and incomes
      transactionHistory = currentBalance - expenses[i] + incomes[i];


      // Update current balance for the next iteration
      currentBalance = transactionHistory;


      // Display the month and the updated balance
      cout << months[i] << ": " << currentBalance << endl;
   }
   return 0;
}
```

Output

```
Enter Customer Name: Muneeb
Enter Account Number: 12
Customer Name: Muneeb
Account Number: 12
Transaction History
January: 5600
February: 5993
March: 5393
April: 5993
May: 6943
```

3. **Create a C++ program to initialize and display 5 flowers names.**

Code

```
#include <iostream>

using namespace std;

int main()

{

   // Declaration and initialization of an array of strings

   string Flowers[5] = {"White rose ", "Tulip", "Rose", "Lily", "Jasmine"};

  // Using a for loop to iterate through the array

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

   {

      // Printing the index and the corresponding flower name
```

```
        cout << i << " = " << Flowers[i] << "\n";

    }

  // Additional statement to indicate the end of the program

    cout << "These are the flowers in the list!" << endl;

 return 0;

}
```

Output

```
0 = White rose
1 = Tulip
2 = Rose
3 = Lily
4 = Jasmine
These are the flowers in the list!
```

4. **Write a C++ array program to insert element at the end of an array.**

Code

```
#include <iostream>

using namespace std;

int main()

{

    int arr[10], n, i, x;

 cout << "Enter size of array: ";

    cin >> n;

  cout << "Enter elements of array: ";
```

```
for(i = 0; i < n; i++)

{

    cin >> arr[i];

}

cout << "Enter element to insert at the end of the array: ";

cin >> x;

arr[i] = x;

n++;


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

{

    cout << arr[i] << endl;

}

return 0;

}
```

Output

```
Enter size of array: 5
Enter elements of array: 2
4
5
7
8
Enter element to insert at the end of the array: 9
2
4
5
7
8
9
```

5. Write a C++ program to delete at the end of array.

Code

```
#include <iostream>

using namespace std;

int main()

{

   int arr[10], n, i;



   cout << "Enter size of array: ";

   cin >> n;

  cout << "Enter elements of array: ";

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

   {

       cin >> arr[i];

   }

    if (n > 0)

{

       // Decrease the size of the array to remove the last element

       n--;

   }

else

 {
```

```
        cout << "Array is empty, no element to delete." << endl;
    }


    cout << "Array after deleting the last element: " << endl;
    for(i = 0; i < n; i++)
    {
        cout << arr[i] << endl;
    }
    return 0;
}
```

Output

```
Enter size of array: 4
Enter elements of array: 3
1
3
8
Array after deleting the last element:
3
1
3
```

Lab No 03

Arrays practice

1. **Create a C++ program to store and display the names of five cloth brands.**

Code

```
#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] = "J.";

   cloths[1] = "Khaadi";

    cloths[2] = "Sapphire";

    cloths[3] = "Dior";

    cloths[4] = "Laaj";

     // Loop through the array and print each cloth brand name

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

{
```

```
        cout << cloths[i] << "\n";

    }

     return 0;

}
```

Output

2. **Create a C++ program to check the size of an array.**

Code

```
#include <iostream>

using namespace std;

int main()

{

    //Declare an array to check the size

 int Numbers[5] = {5, 2, 8, 2, 4};

//print the size of the array

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



  return 0;

}
```

Output

```
The size of the array is: 20
```

3. Create a C++ program to find the largest element in the array.

Code

```
#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

```
Enter 5 elements:
4
6
8
1
3
The largest element is: 8
```

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

Code

```
#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

```
Enter 5 elements:
2
6
7
3
1
The smallest element is: 1
```

Lab N0 04

MULTI-DIMENSIONAL ARRAYS

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.

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).

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.

Example Programs

1. **Create a C++ 2D array program of three rows and three columns.**

Code

```
#include <iostream>

using namespace std;

int main()

{

   // Declare and initialize a 2D array

   int matrix[3][3] =

 {

       {4, 7, 3},

       {4, 8, 6},

       {7, 0, 9}

         };

   // Display the 2D array

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

{

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

     {

           cout << matrix[i][j] << " ";

       }
```

```
        cout << endl;

    }

            return 0;

     }
```

Output

```
4 7 3
4 8 6
7 0 9
```

2. **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.**

Code

```
#include <iostream>

using namespace std;

int main()

{

    // Define and initialize a 3x3 2D array

    int arr[3][3] = {

        {3, 4, 5}, // First Row

        {5, 3, 7}, // Second Row

        {8, 1, 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;

}
```

<u>Output</u>

```
3 4 5
5 3 7
8 1 0
```

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

Code

```
#include <iostream>

using namespace std;

int main()

{
```

```
// Declare and initialize a 3D array
 int cube[3][3][3] =
{
       {
           {1, 3, 4},
           {4, 5, 5},
           {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;
}
```

<u>Output</u>

```
Depth 0:
1 3 4
4 5 5
7 8 9

Depth 1:
10 11 12
13 14 15
16 17 18

Depth 2:
19 20 21
22 23 24
25 26 27
```

Lab N0 05

Vectors

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, like 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).

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

Code

```
#include <iostream>

#include <vector>

using namespace std;

int main()

{

  vector<string> cars = {"MUSTANG", "BMW", "FORD", "NISSAN"};
```

```
  // Change the value of the first element

 cars[0] = "Honda";

   cout << cars[0];

 return 0;}
```

Output

```
Honda
```

2. **Create a C++ program to check size of vectors and remove element and update it.**

Code

```
#include <iostream>

#include <vector>

using namespace std;

int main()

{

    // Create a vector to store integers

    vector<int> numbers;

    // Add elements to the vector

    numbers.push_back(23);

    numbers.push_back(26);

    numbers.push_back(28);

    // Display the elements of the vector

    cout << "Vector elements: ";
```

```
    for (int i = 0; i < numbers.size(); i++)
 {
// Use size() to get the number of elements
        cout << numbers[i] << " ";
    }
    cout << endl;
    // Remove the last element
    numbers.pop_back();
    // Display the updated vector
    cout << "After pop_back, elements: ";
    for (int i = 0; i < numbers.size(); i++)
{
        cout << numbers[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output

```
Vector elements: 23 26 28
After pop_back, elements: 23 26
```

3. Inserting Elements at Specific Position

Code:

```
#include <iostream>

#include <vector>

using namespace std;

 int main() {

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

    vec.insert(vec.begin() + 2, 3); // Insert 3 at the third position

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

        cout << "Element " << i << ": " << vec[i] << endl;

    }

  return 0;

}
```

Output:

```
Element 0: 1
Element 1: 2
Element 2: 3
Element 3: 4
Element 4: 5
```

4. **Sorting Elements**

Code:

```
#include <iostream>

#include <vector>
```

```
#include <algorithm>

using namespace std;

 int main() {

    vector<int> vec = {4, 6, 7, 9, 3};

     sort(vec.begin(), vec.end());

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

        cout << "Element " << i << ": " << vec[i] << endl;

    }

  return 0;

}
```

Output:

```
Element 0: 3
Element 1: 4
Element 2: 6
Element 3: 7
Element 4: 9
```

Lab N0 06

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.

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).**

Code

```
#include <iostream>

using namespace std;

#include <string>

bool isPalindrome(const string& word)

 {

   string reversedWord;
```

```
int length = word.length();

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

        reversedWord.push_back(word[length - 1 - i]);

    }

      return word == reversedWord;

}

int main()

{

    string userWord;

cout << "Enter a word to check if it's a palindrome: ";

    cin >> userWord;

    if (isPalindrome(userWord))

{

cout << "'" << userWord << "' is a palindrome!" << endl;

    } else {

cout << "'" << userWord << "' is not a palindrome." << endl;

    }

     return 0;

}
```

Output

```
Enter a word to check if it's a palindrome: Muneeb
'Muneeb' is not a palindrome.
```

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

Code

```
#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
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

```
Enter a string of parentheses: {*+-}(
The string is not balanced.
```

3. **Write a C++ program to insert an element into a stack.**

Code

```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100;
void push(int stack[], int& top, int element) {
  if (top >= MAX_SIZE - 1) {
      cout << "Stack overflow, cannot insert element." << endl;
      return;
  }
  stack[++top] = element;
  cout << "Inserted " << element << " into the stack." << endl;
}
void displayStack(const int stack[], int top) {
```

```
    if (top == -1) {
        cout << "Stack is empty." << endl;
        return;
    }
    cout << "Stack elements: ";
    for (int i = 0; i <= top; i++) {
        cout << stack[i] << " ";
    }
    cout << endl;
}
int main() {
    int stack[MAX_SIZE];
    int top = -1;  // Initialize stack pointer
    int choice, element;
        while (true) {
        cout << "Menu:\n1. Push Element\n2. Display Stack\n3. Exit\nEnter your choice: ";
        cin >> choice;
          switch (choice) {
           case 1:
                cout << "Enter element to insert: ";
                cin >> element;
                push(stack, top, element);
                break;
```

```
    case 2:
            displayStack(stack, top);
            break;
        case 3:
            return 0;
        default:
            cout << "Invalid choice. Please try again." << endl;
      }
   }
     return 0;
}
```

Output

```
Menu:
1. Push Element
2. Display Stack
3. Exit
Enter your choice: 1
Enter element to insert: 56
Inserted 56 into the stack.
Menu:
1. Push Element
2. Display Stack
3. Exit
Enter your choice: 2
Stack elements: 56
Menu:
1. Push Element
2. Display Stack
3. Exit
Enter your choice: 
```

4. Write a C++ program to delete an element from the stack.

Code

```
#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

```
Enter command (type, undo, exit): type
Enter text: Muneeb
Typed: "Muneeb"

Current text content: Muneeb

Enter command (type, undo, exit): exit
```

5. **Write C++ program of infix to postfix conversion.**

Code

```
    #include <iostream>

    #include <string>

    using namespace std;

const int MAX_SIZE = 100;

class Stack {

      char arr[MAX_SIZE];

      int top;

    public:

      Stack() : top(-1) {}

      void push(char c) {

        if (top >= MAX_SIZE - 1) {
```

```
            cout << "Stack overflow!" << endl;
            return;
        }
        arr[++top] = c;
    }
    char pop() {
        if (top == -1) {
            cout << "Stack underflow!" << endl;
            return '\0';
        }
        return arr[top--];
    }
    char peek() {
        if (top == -1) {
            return '\0';
        }
        return arr[top];
    }
    bool isEmpty() {
        return top == -1;
    }
  };
int precedence(char op) {
```

```
    switch (op) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            case '^':
                return 3;
            default:
                return 0;
        }
    }
bool isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
    }
string infixToPostfix(const string& infix) {
        Stack stack;
        string postfix = "";
    for (char c : infix) {
            if (isalnum(c)) {  // If the character is an operand, add it to postfix
                postfix += c;
            } else if (c == '(') {
```

```
                stack.push(c);
            } else if (c == ')') {
                stack.push(c);
                while (!stack.isEmpty() && stack.peek() != '(') {
                    postfix += stack.pop();
                }
                stack.pop();  // Pop the '(' from the stack
            } else if (isOperator(c)) {
                while (!stack.isEmpty() && precedence(stack.peek()) >= precedence(c)) {
                    postfix += stack.pop();
                }
                stack.push(c);
            }
        }

        while (!stack.isEmpty()) {
            postfix += stack.pop();
        }
    return postfix;
    }
int main() {
        string infix;
        cout << "Enter an infix expression: ";
        cin >> infix;
```

```
    string postfix = infixToPostfix(infix);

      cout << "Postfix expression: " << postfix << endl;

return 0;

   }
```

Output

```
Enter an infix expression: (A+b){+-*
Postfix expression: Ab++*-
```

Lab NO 07

QUEUE IMPLEMENTATION

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

EXAMPLES

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.**

Code

```
#include <iostream>

using namespace std;

class Queue {

private:

    static const int MAX_SIZE = 100; // Maximum size of the queue

    int arr[MAX_SIZE];              // Array to store queue elements
```

```
    int frontIdx;                  // Index of the front element
    int rearIdx;                   // Index of the rear element
    int count;                     // Number of elements in the queue


public:
    // Constructor
    Queue() {
        frontIdx = 0;
        rearIdx = -1;
        count = 0;
    }
// Enqueue an element
    void push(int value) {
        if (count == MAX_SIZE) {
            cout << "Queue is full, cannot enqueue." << endl;
            return;
        }
        rearIdx = (rearIdx + 1) % MAX_SIZE; // Circular increment
        arr[rearIdx] = value;
        count++;
    }
  // Dequeue an element
    void pop() {
```

```
    if (count == 0) {
        cout << "Queue is empty, cannot dequeue." << endl;
        return;
    }
    frontIdx = (frontIdx + 1) % MAX_SIZE; // Circular increment
    count--;
  }
 // Get the front element
  int front() {
    if (count == 0) {
        cout << "Queue is empty, no front element." << endl;
        return -1; // Sentinel value
    }
    return arr[frontIdx];
  }
 // Check if the queue is empty
  bool empty() {
    return count == 0;
  }
// Get the size of the queue
  int size() {
    return count;
  }
```

```
};
int main() {
    Queue q;
 // Enqueue elements
    q.push(10);
    q.push(20);
    q.push(30);


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


    // Dequeue an element
    q.pop();
    cout << "Front element after dequeue: " << q.front() << endl;


    // Check if the queue is empty
    if (q.empty()) {
        cout << "Queue is empty" << endl;
    } else {
        cout << "Queue is not empty" << endl;
    }
  // Get the size of the queue
    cout << "Queue size: " << q.size() << endl;
```

```
return 0;

}
```

Output

```
Front element: 10
Front element after dequeue: 20
Queue is not empty
Queue size: 2
```

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.***

Code

```
#include <iostream>

#include <string>

using namespace std;

class NameQueue {

private:

   static const int MAX_SIZE = 100; // Maximum size of the queue

   string arr[MAX_SIZE];          // Array to store queue elements

   int frontIdx;                  // Index of the front element

   int rearIdx;                   // Index of the rear element

   int count;                     // Number of elements in the queue


public:

   // Constructor
```

```
  NameQueue() {
    frontIdx = 0;
    rearIdx = -1;
    count = 0;
  }
// Enqueue a name
  void enqueue(const string& name) {
    if (count == MAX_SIZE) {
      cout << "Queue is full, cannot add more names." << endl;
      return;
    }
    rearIdx = (rearIdx + 1) % MAX_SIZE; // Circular increment
    arr[rearIdx] = name;
    count++;
  }
 // Dequeue a name
  void dequeue() {
    if (count == 0) {
      cout << "Queue is empty, cannot dequeue." << endl;
      return;
    }
    frontIdx = (frontIdx + 1) % MAX_SIZE; // Circular increment
    count--;
```

```
    }
// Get the front name
    string front() {
        if (count == 0) {
            return "Queue is empty, no front element.";
        }
        return arr[frontIdx];
    }

    // Check if the queue is empty
    bool isEmpty() {
        return count == 0;
    }
// Display the final state of the queue
    void display() {
        if (count == 0) {
            cout << "Queue is empty." << endl;
            return;
        }
        cout << "Queue contents: ";
        for (int i = 0; i < count; i++) {
            cout << arr[(frontIdx + i) % MAX_SIZE] << " ";
        }
```

```
        cout << endl;   }
};int main() {
    NameQueue queue;


    // Add names to the queue
    queue.enqueue("Muneeb");
    queue.enqueue("Haseeb");
    queue.enqueue("Tayyab");
    queue.enqueue("Zain");
 // Display the front element
    cout << "Front element: " << queue.front() << endl;
    // Dequeue an element
    queue.dequeue();
    cout << "Front element after dequeue: " << queue.front() << endl;
   // Show the final state of the queue
    queue.display();
return 0;
}
```

OUTPUT

```
Front element: Muneeb
Front element after dequeue: Haseeb
Queue contents: Haseeb Tayyab Zain
```

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.

Code

```
#include <iostream>

#include <queue>

using namespace std;

int main() {

    // Declare a queue of integers

    queue<int> numbers;

    // Enqueue elements (integers)

    numbers.push(10);

    numbers.push(20);

    numbers.push(30);

    numbers.push(40);


    // Display the front element

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


    // Display the last element (since queue does not have a built-in function for this,

    // we need to use additional steps)

    cout << "Last element: ";

    if (!numbers.empty()) {
```

```
    int last;
    // Temporarily dequeue elements to find the last one
    for (int i = 0; i < numbers.size(); i++) {
        last = numbers.front();
        numbers.push(last);
        numbers.pop();
    }
    cout << last << endl;
}

// Dequeue an element
numbers.pop();
cout << "Front element after dequeue: " << numbers.front() << endl;

// Check if the queue is empty
if (numbers.empty()) {
    cout << "Queue is empty" << endl;
} else {
    cout << "Queue is not empty" << endl;
}
 // Get the size of the queue
cout << "Queue size: " << numbers.size() << endl;
```

```
    // Display all elements in the queue

    cout << "Remaining elements in the queue:" << endl;

    while (!numbers.empty()) {

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

        numbers.pop();

    }

    cout << endl;

 return 0;

}
```

Output

```
Front element: 10
Last element: 40
Front element after dequeue: 20
Queue is not empty
Queue size: 3
Remaining elements in the queue:
20 30 40
```

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.**

Code

```
#include <iostream>

#include <queue>

using namespace std;

int main() {
```

```
  // Declare a queue of integers
queue<int> numbers;
   // Enqueue elements (integers)
  numbers.push(1);
  numbers.push(2);
  numbers.push(3);
  numbers.push(4);
  numbers.push(5);
  // Display all elements in the queue
  cout << "Initial queue elements: ";
  queue<int> tempQueue = numbers; // Use a temporary queue to display elements
  while (!tempQueue.empty()) {
      cout << tempQueue.front() << " ";
      tempQueue.pop();
  }
  cout << endl;
 // Pop elements one by one and display the queue after each pop
  while (!numbers.empty()) {
      numbers.pop();
      cout << "Queue after pop: ";
      tempQueue = numbers; // Reset the temporary queue
      while (!tempQueue.empty()) {
          cout << tempQueue.front() << " ";
```

```
            tempQueue.pop();

        }

        cout << endl;

    }

    return 0;

}
```

Output

```
Initial queue elements: 1 2 3 4 5
Queue after pop: 2 3 4 5
Queue after pop: 3 4 5
Queue after pop: 4 5
Queue after pop: 5
Queue after pop:
```

Lab N0 08

Link List Implementation

LINK LIST

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.

Types of Linked Lists

- **Single Linked List.**
- **Double Linked List.**
- **Circular Linked List.**

Single Linked List

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++.

Double Linked List

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.

Circular Linked List

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.

<u>Code</u>

```
#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;
    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;

}
```

Output

```
Linked List: 20 -> 10 -> 30 -> 40 -> nullptr
Linked List after deleting 20: 10 -> 30 -> 40 -> nullptr
Linked List after deleting 40: 10 -> 30 -> nullptr
```

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

Code

```
#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, 3);
    create(head, 2);
    create(head, 5);
    create(head, 7);
    create(head, 8);
    // Display elements of the circular linked list
    cout << "Circular Linked List: ";
    display(head);
    return 0;
}
```

<u>Output</u>

```
Circular Linked List: 3 2 5 7 8
```

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

Code

```
#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;
    }
```

```
        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;

}
```

Output

```
Doubly Linked List (Forward): 20 <-> 10 <-> 30 <-> 40 <-> nullptr
Doubly Linked List (Backward): 40 <-> 30 <-> 10 <-> 20 <-> nullptr
Doubly Linked List after deleting 20 (Forward): 10 <-> 30 <-> 40 <-> null
ptr
Doubly Linked List after deleting 40 (Backward): 30 <-> 10 <-> nullptr
```

4. ***Create a C++ single linked list program to delete a node by its value.***

Code

```
#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;
    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;

}
```

Output

```
Singly Linked List: 15 -> 25 -> 35 -> 45 -> nullptr
Singly Linked List after deleting 25: 15 -> 35 -> 45 -> nullptr
Singly Linked List after deleting 45: 15 -> 35 -> nullptr
```

Lab N0 09

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++ program to insert value in binary search tree.***

Code

```
#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(5);
    bst.insert(3);
    bst.insert(70);
    bst.insert(3);
    bst.insert(4);
    bst.insert(6);
    bst.insert(8);
```

```
    cout << "In-order traversal: ";
    bst.inOrderTraversal();
return 0;
}
```

<u>Output</u>

```
In-order traversal: 3 4 5 6 8 70
```

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

Code

```
#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(5);
    bst.insert(3);
    bst.insert(7);
    bst.insert(2);
    bst.insert(4);
    bst.insert(6);
    bst.insert(8);
 cout << "In-order traversal before deletion: ";
    bst.inOrderTraversal();
    bst.deleteNode(2);
    cout << "In-order traversal after deleting 2: ";
    bst.inOrderTraversal();
   bst.deleteNode(3);
    cout << "In-order traversal after deleting : ";
    bst.inOrderTraversal();
bst.deleteNode(5);
    cout << "In-order traversal after deleting 5: ";
    bst.inOrderTraversal();
  return 0;
}
```

Output

```
In-order traversal before deletion: 2 3 4 5 6 7 8
In-order traversal after deleting 2: 3 4 5 6 7 8
In-order traversal after deleting : 4 5 6 7 8
In-order traversal after deleting 5: 4 6 7 8
```

3. ***Create a C++ program to search for a value in binary search tree.***

Code

```
#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 search for a value
bool search(Node* node, int value) {
    if (node == nullptr) {
        return false;
    }
    if (node->data == value) {
        return true;
    }
    if (value < node->data) {
        return search(node->left, value);
    } else {
        return search(node->right, value);
    }
}
// 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 search for a value in the tree
    bool search(int value) {
        return search(root, value);
    }
    // Function to perform in-order traversal
    void inOrderTraversal() {
        inOrderTraversal(root);
        cout << endl;
    }
};
int main() {
```

```
    BinarySearchTree bst;

    bst.insert(5);

    bst.insert(3);

    bst.insert(7);

    bst.insert(2);

    bst.insert(4);

    bst.insert(6);

    bst.insert(8);

  cout << "In-order traversal: ";

    bst.inOrderTraversal();

    cout << "Search for 4: " << (bst.search(4) ? "Found" : "Not Found") << endl;

    cout << "Search for 2: " << (bst.search(2) ? "Found" : "Not Found") << endl;

    return 0;

}
```

Output

```
In-order traversal: 2 3 4 5 6 7 8
Search for 4: Found
Search for 2: Found
```

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

Code

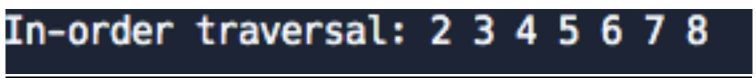
```
#include <iostream>

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(5);
    bst.insert(3);
    bst.insert(7);
```

```
  bst.insert(2);

  bst.insert(4);

  bst.insert(6);

  bst.insert(8);

 cout << "In-order traversal: ";

  bst.inOrderTraversal();

  return 0;

}
```

Output


In-order traversal: 2 3 4 5 6 7 8