# Data Structures

## Lab Manual


**Submitted by:**

**Muhammad Zain**

**Submitted to:**

**Ms. Irsha Qureshi**

**Registration no:**

**2023-bsai-052-III(A)**

**Department :**

**Cs department(AI)**

# Table of Contents

# Lab 1

# Basics

## Variables:

Used as a container to store data.
Can be of any data type.
Must be declared before use.
Name of variable can consist of alphabets, digits and underscore, but should start with alphabet or underscore.

### Syntax:

Declare: variable_name;
Define: variable_name = value;

## Identifiers:

Unique name of the Variable.

## Constants:

Fixed values that can not be changed in a program.

## Data Types:

char: Used to represent characters.
int: Used to represent integral numbers.
float: used to represent decimal numbers.
double: Used to represent decimal numbers.
string: Used to represent number of characters.

## Conditional Statements:

Conditional statements in programming are used to control the flow of a program based on certain conditions. These statements allow the execution of different code blocks depending on whether a specified condition evaluates to true or false, providing a fundamental mechanism for decision-making in algorithms.

### if:

```
if(condition)
{
        statements;
}
```

### if else:

```
if(condition)
{
```

```
        statements;
    }
    else
    {
        statements;
    }
```

**if-else-if:**

```
    if(condition)
    {
        statements;
    }
    else if(condition)
    {
        statements;
    }
```

## Loops:

Loops or Iteration Statements in Programming are helpful when we need a specific task in repetition. They're essential as they reduce hours of work to seconds.

**for:**

```
    for(initialization; condition; increment/decrement)
    {
        statements;
    }
```

**while:**

```
    while(condition)
    {
        statements;
    }
```

**do-while:**

```
    do
    {
        statements;
    }
    while(condtion);
```

## Data Structures:

A data structure is a format that organizes, stores, and processes data in a computer system.

# Types of Data Structures:

## Linear:

The linear data structure is nothing but arranging the data elements linearly one after the other. Here, we cannot arrange the data elements randomly as in the hierarchical order.

Examples: Array, Stack, Queue, Linked List, etc.

## Non-Linear:

A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure.

Examples: Trees, Graphs, etc.

# Lab 2

# Implementation of Array

## Definition:

**Array** is a collection of elements of the same data type, stored in contiguous memory locations. Arrays are a fundamental data structure that provides fixed-size, sequential storage for elements, allowing direct access to any element using an index.

## Syntax:

type arrayName[size];

## Basic Operations:

**Traversal:** Access and process each element of the array sequentially.
**Insertion:** Add a new element to the array at a specific position.
**Deletion:** Remove an element from a specific position in the array.
**Searching:** Find the position of a specific element in the array.
**Updating/Modification:** Replace an element at a specific index with a new value.
**Sorting:** Arrange the elements of the array in ascending or descending order.
**Merging:** Combine two or more arrays into a single array.
**Reversing:** Reverse the order of elements in the array.

## 1: Simple Array

### Code:

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

int main()
{
        // Define the maximum size of the array
        const int SIZE = 100;

        // Declare an array and initialize it with some values
        int numbers[5] = {10, 20, 30, 40, 50};

        // Current size of the array
        int count = 5;

        // Display the array before insertion
        cout << "Array: ";
        for (int i = 0; i < count; i++)
        {
```

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

	return 0;
}
```

```
Array: 10 20 30 40 50
```

## 2: Insertion at Start in Array

**Code:**

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

// Function to insert a value at the start of an array
void insertAtStart(int arr[], int& count, int size, int value)
{
	// Check if there is space in the array
	if (count >= size)
	{
		cout << "Error: Array is full. Cannot insert new element." << endl;
	}
	else
	{
		// Shift all elements one position to the right
		for (int i = count; i > 0; i--)
		{
			arr[i] = arr[i - 1];
		}

		// Insert the new value at the start
		arr[0] = value;

		// Increase the size of the array
		count++;
	}
}

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;
```

```
    // Declare an array and initialize it with some values
    int arr[SIZE] = {1, 2, 3, 4, 5};

    // Current size of the array (number of elements it currently holds)
    int count = 5;

    // Display the array before insertion
    cout << "Array before insertion: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Value to be inserted at the start
    int value;
    cout << "Enter the value to insert at the start: ";
    cin >> value;

    // Call the function to insert the value at the start
    insertAtStart(arr, count, SIZE, value);

    // Display the array after insertion
    cout << "Array after insertion: ";
    for (int i = 0; i <count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
Array before insertion: 1 2 3 4 5
Enter the value to insert at the start: Array after insertion: 0 1 2
```

## 3: Insertion at End in Array

### Code:

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

// Function to insert a value at the end of an array
void insertAtEnd(int arr[], int& count, int size, int value)
{
    // Check if there is space in the array
```

```
	if (count >= size)
	{
		cout << "Error: Array is full. Cannot insert new element." << endl;
	}
	else
	{
		// Insert the new value at the end
		arr[count] = value;

		// Increase the size of the array
		count++;
	}
}

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare an array and initialize it with some values
	int arr[SIZE] = {1, 2, 3, 4, 5};

	// Current size of the array (number of elements it currently holds)
	int count = 5;

	// Display the array before insertion
	cout << "Array before insertion: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	// Value to be inserted at the end
	int value;
	cout << "Enter the value to insert at the end: ";
	cin >> value;

	// Call the function to insert the value at the end
	insertAtEnd(arr, count, SIZE, value);

	// Display the array after insertion
	cout << "Array after insertion: ";
	for (int i = 0; i <count; i++)
	{
		cout << arr[i] << " ";
```

```
	}
	cout << endl;

	return 0;
}
```

```
Array before insertion: 1 2 3 4 5
Enter the value to insert at the end: Array after insertion: 1 2 3 4
```

## 4: Insertion at Mid in Array

### Code:

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

// Function to insert a value before a specified index in the array
void insertAtMid(int arr[], int& count, int Size, int index, int value)
{
	// Check if the array is full
	if (count >= Size)
	{
	cout << "Error: Array is full. Cannot insert new element." << endl;
	}
	else
	{
		// Check if the index is valid
		if (index < 0 || index > count)
		{
		cout << "Error: Invalid index. Cannot insert at this position." << endl;
		}
		else
		{
			// Shift elements to the right starting from the last element to the index
			for (int i = count; i > index; i--)
			{
				arr[i] = arr[i - 1];
			}

			// Insert the value before the specified index
			arr[index] = value;

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

}
```

```
int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare an array and initialize it with some values
	int arr[SIZE] = {1, 2, 3, 4, 5};

	// Current size of the array
	int count = 5;

	// Display the array before insertion
	cout << "Array before insertion: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	// Get the value and index for insertion
	int value, index;
	cout << "Enter the value to insert: ";
	cin >> value;
	cout << "Enter the index to insert before: ";
	cin >> index;

	// Call the function to insert the value before the specified index
	insertAtMid(arr, count, SIZE, index, value);

	// Display the array after insertion
	cout << "Array after insertion: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
 2 3 4 5
: Enter the index to insert before: Array after insertion: 0 1 2 3 4 5
```

## 5: Deletion at Start in Array

**Code:**

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

// Function to delete the first element of the array
void deleteAtStart(int arr[], int& count)
{
	// Check if the array is empty
	if (count <= 0)
	{
		cout << "Error: Array is empty. Cannot delete element." << endl;
	}
	else
	{
		// Shift all elements to the left to remove the first element
		for (int i = 0; i < count - 1; i++)
		{
			arr[i] = arr[i + 1];
		}

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

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare and initialize an array with some values
	int arr[SIZE] = {1, 2, 3, 4, 5};

	// Current size of the array
	int count = 5;

	// Display the array before deletion
	cout << "Array before deletion: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	// Call the function to delete the first element
	deleteAtStart(arr, count);
```

```
    // Display the array after deletion
    cout << "Array after deletion: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
Array before deletion: 1 2 3 4 5
Array after deletion: 2 3 4 5
```

## 6: Deletion at End in Array

**Code:**

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

// Function to delete the last element of the array
void deleteAtEnd(int arr[], int& count)
{
    // Check if the array is empty
    if (count <= 0)
    {
        cout << "Error: Array is empty. Cannot delete element." << endl;
    }
    else
    {
        // Decrease the size of the array
        count--;
    }
}

int main()
{
    // Define the maximum size of the array
    const int SIZE = 100;

    // Declare and initialize an array with some values
    int arr[SIZE] = {1, 2, 3, 4, 5};

    // Current size of the array
```

```
    int count = 5;

    // Display the array before deletion
    cout << "Array before deletion: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Call the function to delete the last element
    deleteAtEnd(arr, count);

    // Display the array after deletion
    cout << "Array after deletion: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
Array before deletion: 1 2 3 4 5
Array after deletion: 1 2 3 4
```

## 7: Deletion at Mid in Array

### Code:

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

// Function to delete an element at a specified index
void deleteAtMid(int arr[], int& count, int index)
{
    // Check if the array is empty
    if (count <= 0)
    {
        cout << "Error: Array is empty. Cannot delete element." << endl;
    }
    else
    {
        // Check if the index is valid
        if (index < 0 || index >= count)
```

```
        {
            cout << "Error: Invalid index. Cannot delete element." << endl;
        }
        else
        {
            // Shift elements to the left starting from the specified index
            for (int i = index; i < count - 1; i++)
            {
                arr[i] = arr[i + 1];
            }

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

int main()
{
    // Define the maximum size of the array (for flexibility)
    const int SIZE = 100;

    // Declare and initialize an array with some values
    int arr[SIZE] = {1, 2, 3, 4, 5};

    // Current size of the array
    int count = 5;

    // Display the array before deletion
    cout << "Array before deletion: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Get the index of the element to delete
    int index;
    cout << "Enter the index to delete: ";
    cin >> index;

    // Call the function to delete the element at the specified index
    deleteAtMid(arr, count, index);

    // Display the array after deletion
    cout << "Array after deletion: ";
```

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

    return 0;
}
```

```
Array before deletion: 1 2 3 4 5
Enter the index to delete: Array after deletion: 2 3 4 5
```

# Lab 3

# Array Practice

## 1: Searching in Array

**Code:**

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

// Function to search for an element in the array
int searchInArray(int arr[], int count, int value)
{
	for (int i = 0; i < count; i++)
	{
		if (arr[i] == value)
		{
			return i; // Return the index if the target is found
		}
	}
	return -1; // Return -1 if the target is not found
}

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare an array and initialize it with some values
	int arr[5] = {1, 2, 3, 4, 5};

	// Current size of the array
	int count = 5;

	// Display the array before insertion
	cout << "Array: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout<<endl;

	// Get the element to search from the user
	int value;
	cout << "Enter the element to search: ";
```

```
    cin >> value;

    // Call the search function
    int result = searchInArray(arr, count, value);

    // Display the result
    if (result != -1)
    {
        cout << "Element " << value << " found at index " << result << "." << endl;
    }
    else
    {
        cout << "Element " << value << " not found in the array." << endl;
    }
    return 0;
}
```

```
Array: 1 2 3 4 5
Element 3 found at index: 2
--------------------------------
```

## 2: Finding Maximum Number

**Code:**

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

// Function to find the maximum number in the array
int findMax(int arr[], int count)
{
    int maxNum = arr[0]; // Assume the first element is the largest
    for (int i = 1; i < count; i++)
    {
        if (arr[i] > maxNum)
        {
            maxNum = arr[i]; // Update maxNum if a larger value is found
        }
    }
    return maxNum;
}

int main()
{
    // Define the maximum size of the array
    const int SIZE = 100;
```

```
    // Declare an array and initialize it with some values
    int arr[] = {1, 2, 3, 4, 5};

    // Current size of the array
    int count = 5;

    // Display the array
    cout << "Array: ";
    for (int i = 0; i < count; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Call the function to find the maximum number
    int maxNumber = findMax(arr, count);

    // Display the result
    cout << "The maximum number in the array is: " << maxNumber << endl;

    return 0;
}
```

```
Array: 1 2 3 4 5
The maximum number in the array is: 5
```

## 3: Sorting in Array

**Code:**

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

// Function to sort the array in ascending order
void sortArray(int arr[], int count)
{
    for (int i = 0; i < count - 1; i++)
    {
        for (int j = 0; j < count - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
                }
            }
        }
}

int main()
{
        // Define the maximum size of the array
        const int SIZE = 100;

        // Declare an array and initialize it with some values
        int arr[] = {9, 7, 3, 2, 5};

        // Current size of the array
        int count = 5;

        // Display the array before sorting
        cout << "Array before sorting: ";
        for (int i = 0; i < count; i++)
        {
                cout << arr[i] << " ";
        }
        cout << endl;

        // Call the function to sort the array
        sortArray(arr, count);

        // Display the array after sorting
        cout << "Array after sorting: ";
        for (int i = 0; i < count; i++)
        {
                cout << arr[i] << " ";
        }
        cout << endl;

        return 0;
}
```

```
Array before sorting: 9 7 3 2 5
Array after sorting: 2 3 5 7 9
```

## 4: Reversing in Array

**Code:**

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

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare an array and initialize it with some values
	int arr[] = {1, 2, 3, 4, 5};

	// Current size of the array
	int count = 5;

	// Display the array before reversing
	cout << "Array before reversing: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	// Display the array after reversing
	cout << "Array after reversing: ";
	for (int i = count-1; i >= 0; i--)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
Array before reversing: 1 2 3 4 5
Array after reversing: 5 4 3 2 1
```

## 5: Updating Index in Array

### Code:

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

// Function to update a specific index in the array
void updateIndex(int arr[], int count, int index, int newValue)
{
	if (index >= 0 && index < count)
```

```
	{
		arr[index] = newValue; // Update the value at the specified index
	}
	else
	{
		cout << "Invalid index!" << endl;
	}
}

int main()
{
	// Define the maximum size of the array
	const int SIZE = 100;

	// Declare an array and initialize it with some values
	int arr[] = {1, 2, 3, 4, 5};

	// Current size of the array
	int count = 5;

	// Display the array before updating
	cout << "Array before update: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;

	// Input index and new value
	int index, newValue;
	cout << "Enter the index to update: ";
	cin >> index;
	cout << "Enter the new value: ";
	cin >> newValue;

	// Update the array at the specified index
	updateIndex(arr, count, index, newValue);

	// Display the array after updating
	cout << "Array after update: ";
	for (int i = 0; i < count; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;
```

```
    return 0;
}
```

```
Array before update: 1 2 3 4 5
Enter the index to update: Enter the new value: Invalid index!
Array after update: 1 2 3 4 5
```

# Lab 4

# Implementation of 2D Array

## Definition:

2D array is an array of arrays, which is essentially a grid-like data structure. It is used to store elements in a two-dimensional table format, where data is organized in rows and columns.

## Syntax:

type arrayName[row][column];

## Basic Operations:

**Traversal:** Access and process each element of the array sequentially.
**Insertion:** Add a new element to the array at a specific position.
**Deletion:** Remove an element from a specific position in the array.
**Searching:** Find the position of a specific element in the array.
**Updating/Modification:** Replace an element at a specific index with a new value.
**Sorting:** Arrange the elements of the array in ascending or descending order.
**Merging:** Combine two or more arrays into a single array.
**Reversing:** Reverse the order of elements in the array.

## 1: Simple 2D Array

### Code:

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

int main()
{
	// Define the dimensions of the 2D array
	const int rows = 3;
	const int cols = 3;

	// Initialize a 2D array with predefined values
	int array[rows][cols] =
	{
	{0, 2, 3},
	{1, 5, 6},
	{4, 8, 9}
	};

	// Display the 2D array
```

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

    return 0;
}
```

```
The 2D array is:
0 2 3
1 5 6
4 8 9
```

## 2: Searching in 2D Arrays

### Code:

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

// Function to search for an element in a 2D matrix
bool searchElement(int matrix[3][3], int value)
{
    // Loop through each row of the matrix
    for (int i = 0; i < 3; i++)
    {
        // Loop through each column of the current row
        for (int j = 0; j < 3; j++)
        {
            // Check if the current element matches the target value
            if (matrix[i][j] == value)
            {
                // If found, print the position (i, j) and return true
                cout << "Element found at position (" << i << ", " << j << ")\n";
                return true;  // Element found, exit the function
            }
        }
    }
    // If the element is not found in the matrix, return false
    return false;
```

```
}

int main()
{
	// Define the dimensions of the 2D array (3x3 matrix)
	const int rows = 3;
	const int cols = 3;

	// Initialize the 2D array (matrix) with predefined values
	int array[rows][cols] =
	{
		{0, 2, 3},
		{1, 5, 6},
		{4, 8, 9}
	};

	// Ask the user to input the value to search for
	int value;
	cout << "Enter the element to search: ";
	cin >> value;

	// Call the search function and check if the element was found
	if (!searchElement(array, value))
	{
		// If the element was not found, print a message
		cout << "Element not found in the matrix.\n";
	}

	return 0;
}
```

```
Enter the element to search: Element not found in the matrix.
```

## 3: Addition of two 2D Arrays

### Code:

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

int main()
{
	// Define the dimensions of the 2D arrays
	const int rows = 3;
	const int cols = 3;

	// Initialize two 2D arrays with predefined values
	int array1[rows][cols] =
```

```
{
	{1, 2, 3},
	{4, 5, 6},
	{7, 8, 9}
};

int array2[rows][cols] =
{
	{3, 8, 7},
	{4, 5, 4},
	{8, 2, 1}
};

// Initialize a result array to store the sum
int result[rows][cols];

// Perform the addition of two arrays
for (int i = 0; i < rows; i++)
{
	for (int j = 0; j < cols; j++)
	{
		result[i][j] = array1[i][j] + array2[i][j];
	}
}

// Display the two input arrays
cout << "Array 1:" << endl;
for (int i = 0; i < rows; i++)
{
	for (int j = 0; j < cols; j++)
	{
		cout << array1[i][j] << " ";
	}
	cout << endl;
}

cout << "\nArray 2:" << endl;
for (int i = 0; i < rows; i++)
{
	for (int j = 0; j < cols; j++)
	{
		cout << array2[i][j] << " ";
	}
	cout << endl;
}
```

```
	// Display the result of the addition
	cout << "\nResultant Array after Addition:" << endl;
	for (int i = 0; i < rows; i++)
	{
		for (int j = 0; j < cols; j++)
		{
			cout << result[i][j] << " ";
		}
		cout << endl;
	}

	return 0;
}
```

```
Resultant Array after Addition:
4 10 10
8 10 10
15 10 10
```

## 4: Multiplication of two 2D Arrays

### Code:

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

int main()
{
	// Define the dimensions of the 2D arrays
	const int rows1 = 2, cols1 = 3;  // First matrix: 2x3
	const int rows2 = 3, cols2 = 2;  // Second matrix: 3x2

	// Initialize two 2D arrays (matrices)
	int matrix1[rows1][cols1] =
	{
		{1, 2, 3},
		{4, 5, 6}
	};

	int matrix2[rows2][cols2] =
	{
		{6, 8},
		{1, 10},
		{1, 12}
	};

	// Initialize a result matrix to store the product
	int result[rows1][cols2] = {0};
```

```
	// Matrix multiplication
	for (int i = 0; i < rows1; i++)
	{
	for (int j = 0; j < cols2; j++)
	{
		for (int k = 0; k < cols1; k++)
		{
			result[i][j] += matrix1[i][k] * matrix2[k][j];
		}
		}
	}

	// Display matrix 1
	cout << "Matrix 1:" << endl;
	for (int i = 0; i < rows1; i++)
	{
		for (int j = 0; j < cols1; j++)
		{
			cout << matrix1[i][j] << " ";
		}
		cout << endl;
	}

	// Display matrix 2
	cout << "\nMatrix 2:" << endl;
	for (int i = 0; i < rows2; i++)
	{
		for (int j = 0; j < cols2; j++)
		{
			cout << matrix2[i][j] << " ";
		}
		cout << endl;
	}

	// Display the result of matrix multiplication
	cout << "\nResultant Matrix after Multiplication:" << endl;
	for (int i = 0; i < rows1; i++)
	{
		for (int j = 0; j < cols2; j++)
		{
			cout << result[i][j] << " ";
		}
		cout << endl;
	}

	return 0;
}
```

```
Resultant Matrix after Multiplication:
11 64
35 154
```

## 5: Transpose of Matrix

### Code:

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

// Function to transpose the given matrix
void transpose(int matrix[3][3], int transposed[3][3])
{
	// Loop through each element in the original matrix
	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < 3; j++)
		{
			// Transpose the element: Row becomes column and column becomes row
			transposed[j][i] = matrix[i][j];
		}
	}
}

// Function to print a matrix
void printMatrix(int matrix[3][3])
{
	// Loop through each element and print it
	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < 3; j++)
		{
			// Print the current element followed by a space
			cout << matrix[i][j] << " ";
		}
		// Print a newline after each row
		cout << endl;
	}
}

int main()
{
	// Initialize a 3x3 matrix with predefined values
	int matrix[3][3] =
	{
	{1, 6, 3},
	{4, 5, 2},
	{7, 8, 9}
	};
```

```
    // Declare a matrix to hold the transposed version of the original matrix
    int transposed[3][3];

    // Display the original matrix
    cout << "Original Matrix:" << endl;
    printMatrix(matrix);

    // Call the transpose function to compute the transposed matrix
    transpose(matrix, transposed);
    cout << "\nTransposed Matrix:" << endl;
    printMatrix(transposed);

    return 0;
}
```

```
Original Matrix:
1 6 3
4 5 2
7 8 9

Transposed Matrix:
1 4 7
6 5 8
3 2 9
```

# Lab 5

# Implementation of Vectors

## Definition:

Vector is a container provided by the Standard Template Library (STL) that represents a dynamic array. Unlike arrays, vectors can dynamically resize themselves to accommodate new elements, making them more flexible and powerful.

## Syntax:

```
#include <vector>
vector<type> vectorName;
```

## Basic Operations:

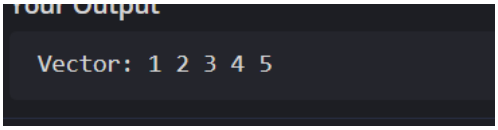
**push_back(val):** Adds val to the end of the vector.
**pop_back():** Removes the last element of the vector.
**size():** Returns the number of elements in the vector.
**capacity():** Returns the total capacity of the vector (memory allocated).
**resize(n):** Resizes the vector to contain n elements.
**empty():** Returns true if the vector is empty, otherwise false.
**front():** Returns the first element of the vector.
**back():** Returns the last element of the vector.
**insert(it, val):** Inserts val at the position pointed to by iterator it.
**erase(it):** Removes the element at the position pointed to by iterator it.
**clear():** Removes all elements from the vector.
**at(index):** Returns the element at the specified index (with bounds checking).
**assign(n, val):** Assigns n copies of val to the vector.

## 1: Simple Vector

### Code:

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the Vector
	cout << "Vector: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
```

```
	}

	return 0;
}
```




## 2: Insertion at Start in Vector

**Code:**

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before insertion
	cout << "Vector before insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Value to insert at the start of the vector
	int value;
	cout << "Enter the value to insert at the start: ";
	cin >> value;

	// Insert the value at the start of the vector
	vec.insert(vec.begin(), value);

	// Display the vector after insertion
	cout << "Vector after insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
```

```
}
```

```
Vector before insertion: 1 2 3 4 5
Enter the value to insert at the start: Vector after insertion: 1 1 2 3 4 5
```

## 3: Insertion at End in Vector

**Code:**

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before insertion
	cout << "Vector before insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Value to insert at the end of the vector
	int value;
	cout << "Enter the value to insert at the end: ";
	cin >> value;

	// Insert the value at the end of the vector
	vec.push_back(value);

	// Display the vector after insertion
	cout << "Vector after insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
Vector before insertion: 1 2 3 4 5
Enter the value to insert at the end: Vector after insertion: 1 2 3 4 5 1
```

## 4: Insertion at Mid in Vector

**Code:**

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before insertion
	cout << "Vector before insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Get the value and index for insertion
	int value, index;
	cout << "Enter the value to insert: ";
	cin >> value;
	cout << "Enter the index to insert before: ";
	cin >> index;

	// Insert the value at the mid of the vector
	vec.insert(vec.begin() + index, value);

	// Display the vector after insertion
	cout << "Vector after insertion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
Vector before insertion: 1 2 3 4 5
Enter the value to insert: Enter the index to insert before: Vector after insertion: 1 5317 2 3 4 5
```

## 5: Deletion at Start in Vector

**Code:**

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before deletion
	cout << "Vector before deletion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Remove the first element of the vector
	vec.erase(vec.begin());

	// Display the vector after deletion
	cout << "Vector after deletion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
Vector before deletion: 1 2 3 4 5
Vector after deletion: 2 3 4 5
```

## 6: Deletion at End in Vector

**Code:**

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

int main()
```

```
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before deletion
	cout << "Vector before deletion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Remove the last element of the vector
	vec.pop_back();

	// Display the vector after deletion
	cout << "Vector after deletion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
}
```

Your Output

```
Vector before deletion: 1 2 3 4 5
Vector after deletion: 1 2 3 4
```

## 7: Deletion at Mid in Vector

### Code:

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

int main()
{
	// Initialize a vector with some integer values
	vector<int> vec = {1, 2, 3, 4, 5};

	// Display the vector before deletion
	cout << "Vector before deletion: ";
	for (int i = 0; i < vec.size(); i++)
```

```
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	// Get the index for deletion
	int value, index;
	cout << "Enter the index to delete: ";
	cin >> index;

	// Remove the given index of the vector
	vec.erase(vec.begin() + index);

	// Display the vector after deletion
	cout << "Vector after deletion: ";
	for (int i = 0; i < vec.size(); i++)
	{
		cout << vec[i] << " ";
	}
	cout << endl;

	return 0;
}
```

```
Vector before deletion: 1 2 3 4 5
Enter the index to delete: Vector after deletion: 1 3 4 5
```

# Lab 6

# Implementation of Stack

## Definition:

Stack is a container provided by the Standard Template Library (STL) that implements a last-in, first-out (LIFO) data structure. This means the last element added to the stack is the first one to be removed.

## Syntax:

```
#include <stack>
stack<type> stackName;
```

## Basic Operations:

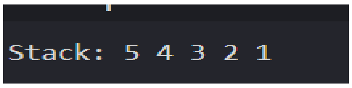
**push(val):** Adds (pushes) val to the top of the stack.
**pop():** Removes (pops) the top element of the stack.
**top():** Returns the value of the top element of the stack.
**size():** Returns the number of elements in the stack.
**empty():** Returns true if the stack is empty, otherwise false.

## 1: Simple Stack

### Code:

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

int main()
{
	// Create a stack of integers
	stack<int> stk;

	// Push elements onto the stack
	stk.push(1);
	stk.push(2);
	stk.push(3);
	stk.push(4);
	stk.push(5);

	// Display Stack by Popping
	cout<<"Stack: ";
	while (!stk.empty())
	{
		cout << stk.top() << " ";
```

```
        stk.pop();
    }

    return 0;
}
```




## 2: Insertion in Stack

### Code:

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

void display(stack<int> stk)
{
    // Display Stack by Popping
    while (!stk.empty())
    {
        cout << stk.top() << " ";
        stk.pop();
    }
    cout<<endl;
}

int main()
{
    // Create a stack of integers
    stack<int> stk;

    // Push elements onto the stack
    stk.push(1);
    stk.push(2);
    stk.push(3);
    stk.push(4);
    stk.push(5);

    // Display before Insertion
    cout<<"Stack before Insertion: ";
    display(stk);

    // Insertion in Stack
    stk.push(19);

    // Display after Insertion
    cout<<"Stack after Insertion: ";
```

```
	display(stk);

	return 0;
}
```

```
Stack before Insertion: 5 4 3 2 1
Stack after Insertion: 19 5 4 3 2 1
```

## 3: Deletion in Stack

**Code:**

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

void display(stack<int> stk)
{
	// Display Stack by Popping
	while (!stk.empty())
	{
		cout << stk.top() << " ";
		stk.pop();
	}
	cout<<endl;
}

int main()
{
	// Create a stack of integers
	stack<int> stk;

	// Push elements onto the stack
	stk.push(1);
	stk.push(2);
	stk.push(3);
	stk.push(4);
	stk.push(5);

	// Display before Deletion
	cout<<"Stack before Deletion: ";
	display(stk);

	// Deletion in Stack
	stk.pop();
```

```
	// Display after Deletion
	cout<<"Stack after Deletion: ";
	display(stk);

	return 0;
}
```

```
Your Output

Stack before Deletion: 5 4 3 2 1
Stack after Deletion: 4 3 2 1
```

## 4: Infix to Postfix

### Code:

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

// Function to check if a character is an operator
bool isOperator(char c)
{
	return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}

// Function to get the precedence of operators
int precedence(char op)
{
	if (op == '+' || op == '-')
	{
		return 1;
	}
	else if (op == '*' || op == '/')
	{
		return 2;
	}
	else if (op == '^')
	{
		return 3;
	}
	return 0;
}

// Function to convert infix expression to postfix expression
```

```
string infixToPostfix(const string& infix)
{
	stack<char> stk;   // Stack to hold operators
	string postfix = "";  // String to store the postfix expression

	for (char c : infix)
	{
		// If the character is an operand (assuming single-character operands)
		if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
		{
			postfix += c; // Add operand to the result
		}

		// If the character is an opening parenthesis, push it to the stack
		else if (c == '(')
		{
			stk.push(c);
		}

		// If the character is a closing parenthesis, pop from the stack until an opening
		parenthesis is encountered
		else if (c == ')')
		{
			while (!stk.empty() && stk.top() != '(')
			{
				postfix += stk.top();
				stk.pop();
			}
			stk.pop();  // Pop the '(' from the stack
		}

		// If the character is an operator
		else if (isOperator(c))
		{
			while (!stk.empty() && precedence(stk.top()) >= precedence(c))
			{
				postfix += stk.top();
				stk.pop();
			}
			stk.push(c);  // Push the current operator to the stack
		}
	}

	// Pop any remaining operators from the stack
	while (!stk.empty())
	{
```

```
                postfix += stk.top();
                stk.pop();
        }

        return postfix;
}

int main()
{
        string infix;

        // Input infix expression
        cout << "Enter an infix expression: ";
        getline(cin, infix);

        // Convert infix to postfix
        string postfix = infixToPostfix(infix);

        // Output the postfix expression
        cout << "Postfix expression: " << postfix << endl;

        return 0;
}
```

```
Infix Expression: A*(B+C)-D/E
Postfix Expression: ABC+*DE/-
```

## 5: Postfix Evaluation

**Code:**

```
#include <iostream>
#include <stack>
#include <string>
#include <cmath> // For power function
using namespace std;

// Function to perform arithmetic operations
int performOperation(int operand1, int operand2, char op)
{
        switch(op)
        {
                case '+': return operand1 + operand2;
                case '-': return operand1 - operand2;
                case '*': return operand1 * operand2;
                case '/': return operand1 / operand2;
                case '^': return pow(operand1, operand2);  // Power operator
```

```
            default: return 0;
        }
}

// Function to evaluate a postfix expression
int evaluatePostfix(const string& postfix)
{
    stack<int> stk;

    for (char c : postfix)
    {
        // If the character is a digit (operand) by checking if it's between '0' and '9'
        if (c >= '0' && c <= '9')
        {
            // Convert character to integer and push to stack
            stk.push(c - '0');  // Convert character to integer (ASCII trick)
        }

        // If the character is an operator
        else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
        {
            // Pop two operands from the stack
            int operand2 = stk.top();
            stk.pop();
            int operand1 = stk.top();
            stk.pop();

            // Perform the operation and push the result back to the stack
            int result = performOperation(operand1, operand2, c);
            stk.push(result);
        }
    }

    // The final result will be the only element left in the stack
    return stk.top();
}

int main()
{
    string postfix;

    // Input postfix expression
    cout << "Enter a postfix expression (single digit operands): ";
    getline(cin, postfix);

    // Evaluate the postfix expression
```

```
    int result = evaluatePostfix(postfix);

    // Output the result
    cout << "The result of the postfix expression is: " << result << endl;

    return 0;
}
```

# Lab 7

# Implementation of List

## Definition:

**List** is a container provided by the **Standard Template Library (STL)** that implements a **doubly linked list**. It is a sequence container that allows for efficient insertion and deletion of elements at both ends and in the middle of the container.

## Syntax:

```
#include <list>
list<type> listName;
```

## Basic Operations:

**push_back(val):** Adds val to the end of the list.
**push_front(val):** Adds val to the beginning of the list.
**pop_back():** Removes the last element from the list.
**pop_front():** Removes the first element from the list.
**insert(it, val):** Inserts val before the position pointed to by iterator it.
**erase(it):** Removes the element at the position pointed to by iterator it.
**remove(val):** Removes all elements with the value val.
**size():** Returns the number of elements in the list.
**clear():** Removes all elements from the list.
**sort():** Sorts the elements of the list in ascending order.
**reverse():** Reverses the order of elements in the list.
**empty():** Returns true if the list is empty, otherwise false.

## 1: Simple List

### Code:

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

int main()
{
	// Create a list of integers
	list<int> List;

	// Inserting elements at the end of the list
	List.push_back(1);
	List.push_back(2);
	List.push_back(3);
	List.push_back(4);
```

```
	List.push_back(5);

	// Displaying the elements of the list
	cout << "Elements of the list: ";
	for (int num : List)
	{
		cout << num << " ";
	}
	cout << endl;

	return 0;
}
```

```
Your Output

Elements of the list: 1 2 3 4 5
```

## 2: Insertion at Start at List

### Code:

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

// Function for Insertion at Start
void display(list<int> List)
{
	for (int num : List)
	{
		cout << num << " ";
	}
	cout << endl;
}

int main()
{
	// Create a list of integers
	list<int> List;

	// Inserting elements at the end of the list
	List.push_back(1);
	List.push_back(2);
	List.push_back(3);
	List.push_back(4);
	List.push_back(5);
```

```
	// Display before Insertion at Start
	cout << "List before Insertion: ";
	display(List);

	// Insertion at Start
	List.push_front(19);

	// Display after Insertion at Start
	cout << "List after Insertion: ";
	display(List);

	return 0;
}
```

Your Output

```
List before Insertion: 1 2 3 4 5
List after Insertion: 19 1 2 3 4 5
```

## 3: Insertion at End

**Code:**

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

// Function for Insertion at End
void display(list<int> List)
{
	for (int num : List)
	{
		cout << num << " ";
	}
	cout << endl;
}

int main()
{
	// Create a list of integers
	list<int> List;

	// Inserting elements at the end of the list
	List.push_back(1);
	List.push_back(2);
	List.push_back(3);
```

```
    List.push_back(4);
    List.push_back(5);

    // Display before Insertion at End
    cout << "List before Insertion: ";
    display(List);

    // Insertion at End
    List.push_back(19);

    // Display after Insertion at End
    cout << "List after Insertion: ";
    display(List);

    return 0;
}
```

Your Output

```
List before Insertion: 1 2 3 4 5
List after Insertion: 1 2 3 4 5 19
```

## 4: Deletion at Start

**Code:**

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

// Function for Deletion at Start
void display(list<int> List)
{
    for (int num : List)
    {
        cout << num << " ";
    }
    cout << endl;
}

int main()
{
    // Create a list of integers
    list<int> List;

    // Inserting elements at the end of the list
    List.push_back(1);
    List.push_back(2);
```

```
	List.push_back(3);
	List.push_back(4);
	List.push_back(5);

	// Display before Deletion at Start
	cout << "List before Deletion: ";
	display(List);

	// Deletion at Start
	List.pop_front();

	// Display after Deletion at Start
	cout << "List after Deletion: ";
	display(List);

	return 0;
}
```

```
Your Output

List before Deletion: 1 2 3 4 5
List after Deletion: 2 3 4 5
```

## 5: Deletion at End

**Code:**

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

// Function for Deletion at End
void display(list<int> List)
{
	for (int num : List)
	{
		cout << num << " ";
	}
	cout << endl;
}

int main()
{
	// Create a list of integers
	list<int> List;

	// Inserting elements at the end of the list
	List.push_back(1);
	List.push_back(2);
```

```
	List.push_back(3);
	List.push_back(4);
	List.push_back(5);

	// Display before Deletion at End
	cout << "List before Deletion: ";
	display(List);

	// Deletion at End
	List.pop_back();

	// Display after Deletion at End
	cout << "List after Deletion: ";
	display(List);
	return 0;
}
```

```
List before Deletion: 1 2 3 4 5
List after Deletion: 1 2 3 4
```

## 6:

## Code:

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

// Function for Deletion at Mid
void display(list<int> List)
{
	for (int num : List)
	{
		cout << num << " ";
	}
	cout << endl;
}
int main()
{
	// Create a list of integers
	list<int> List;
	// Inserting elements at the end of the list
	List.push_back(1);
	List.push_back(2);
	List.push_back(3);
	List.push_back(4);
	List.push_back(5);
	// Display before Deletion at Mid
```

```
        cout << "List before Deletion: ";
        display(List);

        // Get the Value to Delete
        int value;
        cout<<"Enter Value to Delete: ";
        cin>>value;

        // Deletion at Mid
        List.remove(value);
        // Display after Deletion at Mid
        cout << "List after Deletion: ";
        display(List);
        return 0;
}
```

```
List before Deletion: 1 2 3 4 5
Enter Value to Delete: List after Deletion: 1 2 3 4 5
```

# Lab 8

# Implementation of Queue

## Definition:

Queue is a container provided by the Standard Template Library (STL) that implements a first-in, first-out (FIFO) data structure. It is used to store elements in an ordered sequence where elements are added at one end (the rear) and removed from the other end (the front).

## Syntax:

```
#include <queue>
queue<type> queueName;
```

## Basic Operations:

**push(val):** Adds (enqueues) val to the back of the queue.
**pop():** Removes (dequeues) the front element of the queue.
**front():** Returns the value of the front element of the queue.
**back():** Returns the value of the last element of the queue.
**size():** Returns the number of elements in the queue.
**empty():** Returns true if the queue is empty, otherwise false.

## 1: Simple Queue

### Code:

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

int main()
{
	// Create a queue of integers
	queue<int> Queue;

	// Enqueue elements into the queue
	Queue.push(1);
	Queue.push(2);
	Queue.push(3);
	Queue.push(4);
	Queue.push(5);
```

```
    // Display the queue by popping all elements
    cout<<"Queue: ";
    while (!Queue.empty())
    {
        cout<<Queue.front()<<" ";
        Queue.pop();
    }

    return 0;
}
```

```
Queue: 1 2 3 4 5
```

## 2: Insertion in Queue

**Code:**

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

int main()
{
    // Create a queue of integers
    queue<int> Queue;

    // Enqueue elements into the queue
    Queue.push(1);
    Queue.push(2);
    Queue.push(3);
    Queue.push(4);
    Queue.push(5);

    // EnQueue Element in Queue
    Queue.push(19);

    // Display the queue by popping all elements
    cout<<"Queue: ";
    while (!Queue.empty())
    {
        cout<<Queue.front()<<" ";
        Queue.pop();
    }

    return 0;
```

```
}
```

```
Queue: 1 2 3 4 5 19
```

## 3: Deletion in Queue

**Code:**

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

int main()
{
    // Create a queue of integers
    queue<int> Queue;

    // Enqueue elements into the queue
    Queue.push(1);
    Queue.push(2);
    Queue.push(3);
    Queue.push(4);
    Queue.push(5);

    // DeQueue Element in Queue
    Queue.pop();

    // Display the queue by popping all elements
    cout<<"Queue: ";
    while (!Queue.empty())
    {
        cout<<Queue.front()<<" ";
        Queue.pop();
    }

    return 0;
}
```

```
Queue: 2 3 4 5
```

# Lab 9

# Implementation of Singly Linked List

## 1: Simple Linked List

### Code:

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

// Define the structure for a node in the linked list
struct node
{
	int data;		// Data to be stored in the node
	struct node *link; // Pointer to the next node in the list
};

// Global pointers to manage the linked list
struct node *n, *first = NULL, *last = NULL, *current;

// Function to create a new node and add it to the linked list
void create(int data)
{
	n = new node();	// Allocate memory for a new node
	n->data = data;	// Set the data for the node

	if (first == NULL)
	{
		// If the list is empty, create the first node
		n->link = NULL;	// Initialize the link as NULL
		first = last = n;	// Set both first and last pointers to the new node
	}
	else
	{
		// If the list is not empty, add a new node at the end
		n->link = NULL;	// Initialize the link as NULL
		last->link = n;	// Link the last node to the new node
		last = n;		// Update the last pointer to the new node
	}
}

// Function to display the elements of the linked list
void display()
{
	current = first; // Start from the first node
```

```
	while (current != NULL)
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;     // Move to the next node
	}
	cout << endl;
}

int main()
{
	// Create nodes in the linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the linked list after creation
	cout << "Display after Creation: ";
	display();

	return 0;
}
```

```
Display after Creation: 1 2 3 4 5
```

## 2: Insertion at Start

### Code:

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

// Define the structure for a node in the linked list
struct node
{
	int data;          // Data to be stored in the node
	struct node *link; // Pointer to the next node in the list
};

// Global pointers to manage the linked list
struct node *n, *first = NULL, *last = NULL, *current;

// Function to create a new node and add it to the linked list
void create(int data)
{
```

```
	n = new node();     // Allocate memory for a new node
	n->data = data;     // Set the data for the node

	if (first == NULL)
	{
		// If the list is empty, create the first node
		n->link = NULL;     // Initialize the link as NULL
		first = last = n;   // Set both first and last pointers to the new node
	}
	else
	{
		// If the list is not empty, add a new node at the end
		n->link = NULL;     // Initialize the link as NULL
		last->link = n;     // Link the last node to the new node
		last = n;           // Update the last pointer to the new node
	}
}

// Function to insert a new node at the start of the linked list
void insertAtStart(int data)
{
	n = new node();     // Allocate memory for a new node
	n->data = data;     // Set the data for the node
	n->link = first;    // Link the new node to the current first node
	first = n;          // Update the first pointer to the new node
}

// Function to display the elements of the linked list
void display()
{
	current = first; // Start from the first node
	while (current != NULL)
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;     // Move to the next node
	}
	cout << endl; // Add a new line after displaying all elements
}

int main()
{
	// Create nodes in the linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
```

```
	create(5);

	// Display the linked list before insertion
	cout << "Display before Insertion: ";
	display();

	// Insert a new node at the start of the linked list
	insertAtStart(19);

	// Display the linked list after insertion
	cout << "Display after Insertion: ";
	display();

	return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 19 1 2 3 4 5
```

## 3: Insertion at End

**Code:**

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

// Define the structure for a node in the linked list
struct node
{
	int data;		// Data to be stored in the node
	struct node *link; // Pointer to the next node in the list
};

// Global pointers to manage the linked list
struct node *n, *first = NULL, *last = NULL, *current;

// Function to create a new node and add it to the linked list
void create(int data)
{
	n = new node();     // Allocate memory for a new node
	n->data = data;     // Set the data for the node

	if (first == NULL)
	{
		// If the list is empty, create the first node
		n->link = NULL;     // Initialize the link as NULL
```

```
            first = last = n;   // Set both first and last pointers to the new node
        }
        else
        {
            // If the list is not empty, add a new node at the end
            n->link = NULL;     // Initialize the link as NULL
            last->link = n;     // Link the last node to the new node
            last = n;           // Update the last pointer to the new node
        }
}

// Function to insert a new node at the end of the linked list
void insertAtEnd(int data)
{
        n = new node();     // Allocate memory for a new node
        n->data = data;     // Set the data for the node
        n->link = NULL;     // Initialize the link as NULL
        last->link = n;     // Link the current last node to the new node
        last = n;           // Update the last pointer to the new node
}

// Function to display the elements of the linked list
void display()
{
        current = first; // Start from the first node
        while (current != NULL)
        {
            cout << current->data << " "; // Print the data of the current node
            current = current->link;     // Move to the next node
        }
        cout << endl; // Add a new line after displaying all elements
}

int main()
{
        // Create nodes in the linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before insertion
        cout << "Display before Insertion: ";
        display();
```

```
    // Insert a new node at the end of the linked list
    insertAtEnd(19);

    // Display the linked list after insertion
    cout << "Display after Insertion: ";
    display();

    return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 1 2 3 4 5 19
```

## 4: Insertion at Mid

### Code:

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

// Define the structure for a node in the linked list
struct node
{
    int data;          // Data to be stored in the node
    struct node *link; // Pointer to the next node in the list
};

// Global pointers to manage the linked list
struct node *n, *first = NULL, *last = NULL, *current;

// Function to create a new node and add it to the linked list
void create(int data)
{
    n = new node();   // Allocate memory for a new node
    n->data = data;   // Set the data for the new node

    if (first == NULL)
    {
        // If the list is empty, initialize the first and last pointers
        n->link = NULL; // Set the link of the new node to NULL
        first = last = n; // Update the first and last pointers
    }
    else
    {
        // If the list is not empty, append the new node at the end
        n->link = NULL;    // Set the link of the new node to NULL
```

```
            last->link = n;    // Update the last node's link to the new node
            last = n;          // Update the last pointer to the new node
        }
}

// Function to insert a new node before a given value in the linked list
void insertAtMid(int value, int newValue)
{
        current = first;       // Start from the first node
        struct node *previous = NULL; // Pointer to keep track of the previous node

        n = new node();        // Allocate memory for the new node
        n->data = newValue;      // Set the data for the new node

        while (current != NULL)
        {
            if (current->data == value)
            {
                // If the current node contains the specified value
                if (previous == NULL)
                {
                    // If inserting before the first node
                    n->link = current; // Link the new node to the first node
                    first = n;         // Update the first pointer to the new node
                }
                else
                {
                    // If inserting somewhere in the middle or end
                    previous->link = n; // Link the previous node to the new node
                    n->link = current;  // Link the new node to the current node
                }
                return; // Exit the loop after insertion
            }

            // Move to the next node
            previous = current;
            current = current->link;
        }
}

// Function to display the elements of the linked list
void display()
{
        current = first; // Start from the first node
        while (current != NULL)
        {
```

```
            cout << current->data << " "; // Print the data of the current node
            current = current->link;     // Move to the next node
        }
        cout << endl; // Add a new line after displaying all elements
}

int main()
{
        // Create nodes in the linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before insertion
        cout << "Display before Insertion: ";
        display();

        // Variables to store the value to insert and the value before which to insert
        int value, newValue;
        cout << "Enter the Value to Insert: ";
        cin >> newValue;
        cout << "Enter Value to Insert before: ";
        cin >> value;

        // Insert the new node before the specified value
        insertAtMid(value, newValue);

        // Display the linked list after insertion
        cout << "Display after Insertion: ";
        display();

        retu
        rn 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Enter the Value to Insert: Enter Value to Insert before: Display after Insertion: 1 2 3 4 5
```

## 5: Deletion at Start

**Code:**

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

```
// Define the structure for a node in the linked list
struct node
{
	int data;		// Data to be stored in the node
	struct node *link; // Pointer to the next node in the list
};

// Global pointers to manage the linked list
struct node *n, *first = NULL, *last = NULL, *current, *temp;

// Function to create a new node and add it to the linked list
void create(int data)
{
	n = new node();   // Allocate memory for a new node
	n->data = data;   // Set the data for the new node

	if (first == NULL)
	{
		// If the list is empty, initialize the first and last pointers
		n->link = NULL; // Set the link of the new node to NULL
		first = last = n; // Update the first and last pointers
	}
	else
	{
		// If the list is not empty, append the new node at the end
		n->link = NULL;    // Set the link of the new node to NULL
		last->link = n;    // Update the last node's link to the new node
		last = n;          // Update the last pointer to the new node
	}
}

// Function to delete the first node in the linked list
void deleteAtStart()
{
	temp = first;           // Store the first node in a temporary pointer
	first = first->link;    // Update the first pointer to point to the next node
	delete temp;            // Free the memory of the removed node
}

// Function to display the elements of the linked list
void display()
{
	current = first; // Start from the first node
	while (current != NULL)
	{
		cout << current->data << " "; // Print the data of the current node
```

```
            current = current->link;     // Move to the next node
        }
        cout << endl; // Add a new line after displaying all elements
}

int main()
{
        // Create nodes in the linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Delete the first node
        deleteAtStart();

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

```
our Output

Display before Deletion: 1 2 3 4 5
Display after Deletion: 2 3 4 5
```

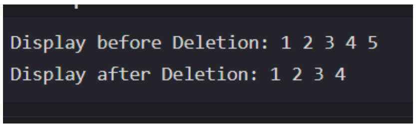
## 6: Deletion at End

**Code:**

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

// Define the structure for a node in the linked list
struct node
{
        int data;           // Data to be stored in the node
        struct node *link; // Pointer to the next node in the list
};

// Global pointers for managing the linked list
```

```
struct node *n, *first = NULL, *last = NULL, *current, *temp;

// Function to create a new node and add it to the linked list
void create(int data)
{
	n = new node();    // Allocate memory for a new node
	n->data = data;    // Set the data for the new node

	if (first == NULL)
	{
		// If the list is empty, initialize the first and last pointers
		n->link = NULL;  // Set the link of the new node to NULL
		first = last = n; // Update both first and last to point to the new node
	}
	else
	{
		// If the list is not empty, append the new node at the end
		n->link = NULL;    // Set the link of the new node to NULL
		last->link = n;    // Update the last node's link to point to the new node
		last = n;          // Update the last pointer to the new node
	}
}

// Function to delete the last node in the linked list
void deleteAtEnd()
{
	current = first; // Start from the first node

	// Traverse to the second last node in the list
	while (current->link != last)
	{
		current = current->link;
	}

	// Delete the last node and update the last pointer
	delete last;        // Free memory of the last node
	last = current;     // Update the last pointer to the second last node
	last->link = NULL;  // Set the new last node's link to NULL
}

// Function to display the elements of the linked list
void display()
{
	current = first; // Start from the first node
	while (current != NULL)
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;     // Move to the next node
	}
	cout << endl; // Add a new line after displaying all elements
```

```
}

int main()
{
	// Create nodes in the linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the linked list before deletion
	cout << "Display before Deletion: ";
	display();

	// Delete the last node
	deleteAtEnd();

	// Display the linked list after deletion
	cout << "Display after Deletion: ";
	display();

	return 0; // Exit the program
}
```




## 7: Deletion at Mid

### Code:

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

// Define a node structure for the linked list
struct node
{
	int data;         // Data stored in the node
	struct node *link; // Pointer to the next node
};

// Global pointers for linked list management
struct node *n, *first = NULL, *last = NULL, *current;

// Function to create a new node and add it to the linked list
void create(int data)
{
	n = new node();     // Allocate memory for a new node
```

```
        n->data = data;     // Set the data for the node

        if (first == NULL)
        {
                // If the list is empty, create the first node
                n->link = NULL;
                first = last = n; // Set both first and last pointers to the new node
        }
        else
        {
                // If the list is not empty, add a new node at the end
                n->link = NULL;
                last->link = n;  // Link the last node to the new node
                last = n;        // Update the last pointer to the new node
        }
}

// Function to delete a node with a specific value from the linked list
void deleteAtMid(int value)
{
        current = first; // Start traversal from the first node
        struct node *previous = NULL; // To keep track of the previous node

        while (current != NULL)
        {
                if (current->data == value)
                {
                        // If the value is found in the current node
                        if (previous == NULL)
                        {
                        // If the node to be deleted is the first node
                        first = first->link; // Move the first pointer to the next node
                        }
                        else
                        {
                        // If the node to be deleted is in the middle or end
                        previous->link = current->link; // Bypass the current node
                        }

                        delete current; // Free memory allocated for the current node
                        return;
                }

                // Move to the next node
                previous = current;
                current = current->link;
        }
}

// Function to display all elements of the linked list
```

```
void display()
{
        current = first; // Start from the first node
        while (current != NULL) {
        cout << current->data << " "; // Print the data of the current node
        current = current->link;     // Move to the next node
        }
        cout << endl; // New line after displaying all elements
}

int main()
{
        // Create a linked list with the given elements
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Input the value to be deleted
        int value;
        cout << "Enter the Value to Delete: ";
        cin >> value;

        // Delete the node with the given value
        deleteAtMid(value);

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

```
Display before Deletion: 1 2 3 4 5
Enter the Value to Delete: Display after Deletion: 1 2 3 4 5
```

# Lab 10

# Implementation of Doubly Linked List

## 1: Simple Doubly Linked List

**Code:**

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

// Define a node structure for the doubly linked list
struct node
{
	int data;          // Data stored in the node
	struct node *next;  // Pointer to the next node
	struct node *prev;  // Pointer to the previous node
};

// Global pointers for managing the linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the list
void create(int data)
{
	n = new node();      // Create a new node
	n->data = data;      // Set the data for the node

	if(first == NULL)    // If the list is empty
	{
		n->next = n->prev = NULL; // Initialize pointers to NULL
		first = last = n;        // Set the new node as the first and last node
	}
	else               // If the list is not empty
	{
		n->next = NULL;         // New node's next is NULL (end of list)
		n->prev = last;        // Link the new node to the last node
		last->next = n;        // Update the last node's next pointer
		last = n;              // Update the last pointer to the new node
	}
}

// Function to display all nodes in the linked list
void display()
{
	current = first;     // Start from the first node
```

```
        while(current != NULL) // Traverse until the end of the list
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next;      // Move to the next node
        }
}

int main()
{
        // Create nodes in the linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list after creation
        cout << "Display after Creation: ";
        display();

        return 0;
}
```

```
Display after Creation: 1 2 3 4 5
```

## 2: Insertion at Start in Doubly Linked List

**Code:**

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

// Define a node structure for the doubly linked list
struct node
{
        int data;           // Data stored in the node
        struct node *next;  // Pointer to the next node
        struct node *prev;  // Pointer to the previous node
};

// Global pointers for managing the linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
```

```
{
    n = new node();       // Create a new node
    n->data = data;       // Set the data for the node

    if(first == NULL)     // If the list is empty
    {
        n->next = n->prev = NULL; // Initialize pointers to NULL
        first = last = n;         // Set the new node as the first and last node
    }
    else                  // If the list is not empty
    {
        n->next = NULL;         // New node's next is NULL (end of list)
        n->prev = last;         // Link the new node to the last node
        last->next = n;         // Update the last node's next pointer
        last = n;               // Update the last pointer to the new node
    }
}

// Function to insert a new node at the start of the list
void insertAtStart(int data)
{
    n = new node();         // Create a new node
    n->data = data;         // Set the data for the node
    n->next = first;        // Link the new node to the current first node
    n->prev = NULL;         // New node's prev is NULL (start of list)
    first->prev = n;        // Update the current first node's prev pointer
    first = n;              // Update the first pointer to the new node
}

// Function to display all nodes in the linked list
void display()
{
    current = first;        // Start from the first node

    while(current != NULL)   // Traverse until the end of the list
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->next;      // Move to the next node
    }
    cout << endl;
}

int main()
{
    // Create nodes in the linked list with the given values
    create(1);
```

```
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the linked list before insertion
	cout << "Display before Insertion: ";
	display();

	// Insert a new node at the start of the linked list
	insertAtStart(19);

	// Display the linked list after insertion
	cout << "Display after Insertion: ";
	display();

	return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 19 1 2 3 4 5
```

## 3: Insertion at End in Doubly Linked List

### Code:

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

// Define a node structure for the doubly linked list
struct node
{
	int data;           // Data stored in the node
	struct node *next;  // Pointer to the next node
	struct node *prev;  // Pointer to the previous node
};

// Global pointers for managing the linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
	n = new node();       // Create a new node
	n->data = data;       // Set the data for the node
```

```
    if(first == NULL)     // If the list is empty
    {
        n->next = n->prev = NULL; // Initialize pointers to NULL
        first = last = n;         // Set the new node as the first and last node
    }
    else                  // If the list is not empty
    {
        n->next = NULL;         // New node's next is NULL (end of list)
        n->prev = last;         // Link the new node to the last node
        last->next = n;         // Update the last node's next pointer
        last = n;               // Update the last pointer to the new node
    }
}

// Function to insert a new node at the end of the list
void insertAtEnd(int data)
{
    n = new node();         // Create a new node
    n->data = data;         // Set the data for the node
    n->next = NULL;         // New node's next is NULL (end of list)
    n->prev = last;         // Link the new node to the current last node
    last->next = n;         // Update the last node's next pointer
    last = n;               // Update the last pointer to the new node
}

// Function to display all nodes in the linked list
void display()
{
    current = first;        // Start from the first node

    while(current != NULL)   // Traverse until the end of the list
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->next;      // Move to the next node
    }
    cout << endl;
}

int main()
{
    // Create nodes in the linked list with the given values
    create(1);
    create(2);
    create(3);
    create(4);
    create(5);
```

```
    // Display the linked list before insertion
    cout << "Display before Insertion: ";
    display();

    // Insert a new node at the end of the linked list
    insertAtEnd(19);

    // Display the linked list after insertion
    cout << "Display after Insertion: ";
    display();

    return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 1 2 3 4 5 19
```

## 4: Insertion at Mid in Doubly Linked List

**Code:**

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

// Define the structure for a doubly linked list node
struct node
{
    int data;          // Data stored in the node
    struct node *next;  // Pointer to the next node
    struct node *prev;  // Pointer to the previous node
};

// Global pointers for managing the linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
    n = new node();      // Create a new node
    n->data = data;      // Assign the data to the node

    if(first == NULL)    // If the list is empty
    {
        n->next = n->prev = NULL; // Initialize the node's pointers to NULL
        first = last = n;         // Set the node as the first and last node
    }
```

```
    else            // If the list is not empty
    {
        n->next = NULL;       // New node's next is NULL (end of list)
        n->prev = last;       // Link the new node to the last node
        last->next = n;       // Update the last node's next pointer
        last = n;             // Update the last pointer to the new node
    }
}

// Function to insert a new node before a given value in the list
void insertAtMid(int value, int newValue)
{
    current = first;  // Start from the first node

    n = new node();   // Create a new node
    n->data = newValue; // Assign the new value to the node

    while (current != NULL)  // Traverse through the list
    {
        if (current->data == value)  // If the current node's data matches the target value
        {
            if (current->prev == NULL)  // Inserting at the start of the list
            {
                n->next = current;     // Point the new node to the current first node
                n->prev = NULL;        // New node's previous pointer is NULL
                current->prev = n;     // Update the current node's previous pointer
                first = n;             // Update the first pointer to the new node
            }
            else  // Inserting in the middle of the list
            {
                n->next = current;     // Point the new node to the current node
                n->prev = current->prev; // Link the new node to the previous node
                current->prev->next = n; // Update the previous node's next pointer
                current->prev = n;     // Update the current node's previous pointer
            }
            return;  // Exit after insertion
        }
        current = current->next;  // Move to the next node in the list
    }
    // If the value is not found in the list
    cout << "Value " << value << " not found in the list." << endl;
}

// Function to display all nodes in the linked list
void display()
{
```

```
    current = first;            // Start from the first node

    while(current != NULL)      // Traverse until the end of the list
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->next;    // Move to the next node
    }
    cout << endl;
}

int main()
{
    // Create nodes in the linked list with the given values
    create(1);
    create(2);
    create(3);
    create(4);
    create(5);

    // Display the linked list before insertion
    cout << "Display before Insertion: ";
    display();

    // Variables to store the value to insert and the value before which to insert
    int value, newValue;
    cout << "Enter the Value to Insert: ";
    cin >> newValue;
    cout << "Enter Value to Insert before: ";
    cin >> value;

    // Insert the new node before the specified value
    insertAtMid(value, newValue);

    // Display the linked list after insertion
    cout << "Display after Insertion: ";
    display();

    return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Enter the Value to Insert: Enter Value to Insert before: Value -1454936214 not found in the list.
Display after Insertion: 1 2 3 4 5
```

## 5: Deletion at Mid in Doubly Linked List

### Code:

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

// Define the structure for a doubly linked list node
struct node
{
	int data;		// Data stored in the node
	struct node *next;	// Pointer to the next node
	struct node *prev;	// Pointer to the previous node
};

// Global pointers for managing the doubly linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the end of the list
void create(int data)
{
	n = new node();		// Allocate memory for a new node
	n->data = data;		// Assign the data to the new node

	if(first == NULL)	// If the list is empty
	{
		n->next = n->prev = NULL; // Initialize the new node's pointers to NULL
		first = last = n;	// Set the new node as the first and last node
	}
	else			// If the list is not empty
	{
		n->next = NULL;		// Set the new node's next pointer to NULL
		n->prev = last;		// Link the new node to the last node
		last->next = n;		// Update the last node's next pointer
		last = n;		// Update the last pointer to the new node
	}
}

// Function to delete a node with a specific value from the list
void deleteAtMid(int value)
{
	current = first;	// Start from the first node

	while(current != NULL)	// Traverse the list
	{
		if(current->data == value) // If the node with the given value is found
		{
```

```
                if(current == first)   // Case 1: Deleting the first node
                {
                        temp = first;      // Temporarily store the first node
                        first = first->next; // Update the first pointer to the next node
                        if (first != NULL)  // Check if the list is not empty after deletion
                        first->prev = NULL; // Set the previous pointer of the new first node to
                        NULL
                        delete(temp);      // Free the memory of the old first node
                }
                else if(current == last) // Case 2: Deleting the last node
                {
                        temp = last;       // Temporarily store the last node
                        last = last->prev; // Update the last pointer to the previous node
                        if (last != NULL)  // Check if the list is not empty after deletion
                        last->next = NULL; // Set the next pointer of the new last node to
                        NULL
                        delete(temp);      // Free the memory of the old last node
                }
                else                  // Case 3: Deleting a node in the middle
                {
                        current->prev->next = current->next; // Update the previous node's
                        next pointer
                        current->next->prev = current->prev; // Update the next node's
                        previous pointer
                        delete current;     // Free the memory of the current node
                }
                return;               // Exit the function after deletion
            }
            current = current->next;  // Move to the next node
        }
}

// Function to display all nodes in the linked list
void display()
{
        current = first;          // Start from the first node

        while(current != NULL)    // Traverse the list until the end
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next;     // Move to the next node
        }
        cout << endl;
}

int main()
```

```
{
    // Create a linked list with the given elements
    create(1);
    create(2);
    create(3);
    create(4);
    create(5);

    // Display the linked list before deletion
    cout << "Display before Deletion: ";
    display();

    // Input the value to be deleted
    int value;
    cout << "Enter the Value to Delete: ";
    cin >> value;

    // Delete the node with the given value
    deleteAtMid(value);

    // Display the linked list after deletion
    cout << "Display after Deletion: ";
    display();

    return 0;
}
```

```
Display before Deletion: 1 2 3 4 5
Enter the Value to Delete: Display after Deletion: 1 2 3 4 5
```

# Lab 11

# Implementation of Circular Linked List

## 1: Simple Circular Linked List

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
	int data;	// Data stored in the node
	struct node *link; // Pointer to the next node
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node();	// Allocate memory for a new node
	n->data = data;	// Assign the data to the new node

	if(first == NULL)	// If the list is empty
	{
		n->link = n;	// Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
	}
	else		// If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n;  // Update the last node's link to point to the new node
		last = n;	// Update the last pointer to the new node
	}
}

// Function to display all nodes in the circular linked list
void display()
{
	current = first;	// Start from the first node

	do		// Traverse the list until we return to the first node
```

```
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;      // Move to the next node
	}
	while(current != first);  // Stop when we loop back to the first node
}

int main()
{
	// Create nodes in the circular linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the circular linked list after creation
	cout << "Display after Creation: ";
	display();

	return 0;
}
```

```
Display after Creation: 1 2 3 4 5
```

## 2: Insertion at Start in Circular Linked List

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
	int data;       // Data stored in the node
	struct node *link; // Pointer to the next node
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node();      // Allocate memory for a new node
	n->data = data;      // Assign the data to the new node
```

```
    if(first == NULL)    // If the list is empty
    {
        n->link = n;     // Point the new node to itself (circular link)
        first = last = n; // Set the new node as both the first and last node
    }
    else                // If the list is not empty
    {
        n->link = first; // Point the new node to the first node
        last->link = n;  // Update the last node's link to point to the new node
        last = n;        // Update the last pointer to the new node
    }
}

// Function to insert a new node at the start of the circular linked list
void insertAtStart(int data)
{
    n = new node();      // Allocate memory for a new node
    n->data = data;      // Assign the data to the new node
    n->link = first;     // Point the new node to the current first node
    last->link = n;      // Update the last node's link to point to the new node
    first = n;           // Update the first pointer to the new node
}

// Function to display all nodes in the circular linked list
void display()
{
    current = first;       // Start from the first node

    do                     // Traverse the list until we return to the first node
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->link;      // Move to the next node
    }
    while(current != first);  // Stop when we loop back to the first node
}

// Main function
int main()
{
    // Create nodes in the circular linked list with the given values
    create(1);
    create(2);
    create(3);
    create(4);
    create(5);
```

```
	// Display the circular linked list before insertion
	cout << "Display before Insertion: ";
	display();

	// Insert a new node at the start of the circular linked list
	insertAtStart(19);

	// Display the circular linked list after insertion
	cout << "\nDisplay after Insertion: ";
	display();

	return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 19 1 2 3 4 5
```

## 3: Deletion at Start in Circular Linked List

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
	int data;		// Data stored in the node
	struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
	n = new node();	// Allocate memory for a new node
	n->data = data;	// Assign the data to the new node

	if(first == NULL)	// If the list is empty
	{
		n->link = n;	// Point the new node to itself (circular link)
		first = last = n; // Set the new node as both the first and last node
```

```
	}
	else			// If the list is not empty
	{
		n->link = first; // Point the new node to the first node
		last->link = n;  // Update the last node's link to point to the new node
		last = n;	// Update the last pointer to the new node
	}
}

// Function to delete the first node in the circular linked list
void deleteAtStart()
{
temp = first;		// Store the first node in a temporary pointer
first = first->link;	// Update the first pointer to the second node
last->link = first;	// Update the last node's link to point to the new first node
delete(temp);		// Free the memory of the old first node
}

// Function to display all nodes in the circular linked list
void display()
{
	current = first;	// Start from the first node

	do
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;	// Move to the next node
	}
	while(current != first);	// Stop when we loop back to the first node
	cout << endl;
}

// Main function
int main()
{
	// Create nodes in the circular linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the circular linked list before deletion
	cout << "Display before Deletion: ";
	display();
```

```
    // Delete the first node in the circular linked list
    deleteAtStart();

    // Display the circular linked list after deletion
    cout << "Display after Deletion: ";
    display();

    return 0;
}
```

```
Display before Deletion: 1 2 3 4 5
Display after Deletion: 2 3 4 5
```

## 4: Deletion at End in Circular Linked List

**Code:**

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

// Define the structure for a circular linked list node
struct node
{
    int data;       // Data stored in the node
    struct node *link; // Pointer to the next node in the circular linked list
};

// Global pointers for managing the circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the circular linked list
void create(int data)
{
    n = new node();     // Allocate memory for a new node
    n->data = data;     // Assign the data to the new node

    if(first == NULL)   // If the list is empty
    {
        n->link = n;    // Point the new node to itself (circular link)
        first = last = n; // Set the new node as both the first and last node
    }
    else                // If the list is not empty
    {
        n->link = first; // Point the new node to the first node
        last->link = n;  // Update the last node's link to point to the new node
        last = n;        // Update the last pointer to the new node
    }
```

```
}

// Function to delete the last node in the circular linked list
void deleteAtEnd()
{
	current = first;		// Start from the first node

	while(current->link != last)	// Traverse the list to find the second-to-last node
	{
		current = current->link;
	}

	delete(last);			// Free the memory of the last node
	current->link = first;		// Update the second-to-last node's link to point to the first node
	last = current;			// Update the last pointer to the second-to-last node
}

// Function to display all nodes in the circular linked list
void display()
{
	current = first;		// Start from the first node

	do
	{
		cout << current->data << " "; // Print the data of the current node
		current = current->link;	// Move to the next node
	}
	while(current != first);	// Stop when we loop back to the first node
	cout << endl;
}

// Main function
int main()
{
	// Create nodes in the linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the linked list before deletion
	cout << "Display before Deletion: ";
	display();

	// Delete the last node in the linked list
```

```
    deleteAtEnd();

    // Display the linked list after deletion
    cout << "Display after Deletion: ";
    display();

    return 0;
}
```

```
Display before Deletion: 1 2 3 4 5
Display after Deletion: 1 2 3 4
```

# Lab 12

# Implementation of Double Circular Linked List

## 1: Simple Double Circular Linked List

**Code:**

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

// Define a structure for a node in a doubly circular linked list
struct node
{
	int data;        // Data stored in the node
	struct node *next; // Pointer to the next node
	struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the doubly circular linked list
void create(int data)
{
	n = new node();    // Allocate memory for a new node
	n->data = data;    // Assign data to the node

	if(first == NULL)  // If the list is empty
	{
		n->next = n;   // Point the node's `next` to itself (circular link)
		n->prev = n;   // Point the node's `prev` to itself
		first = last = n; // Set the new node as the first and last node
	}
	else              // If the list is not empty
	{
		n->next = first; // Point the new node's `next` to the first node
		n->prev = last;  // Point the new node's `prev` to the last node
		last->next = n;  // Update the last node's `next` to the new node
		first->prev = n; // Update the first node's `prev` to the new node
		last = n;        // Update the `last` pointer to the new node
	}
}

// Function to display the data in the doubly circular linked list
void display()
```

```
{
        current = first;     // Start from the first node

        do
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next;      // Move to the next node
        }
        while(current != first);          // Stop when we loop back to the first node
        cout << endl;                     // Print a newline after displaying all nodes
}

// Main function
int main()
{
        // Create nodes in the doubly circular linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list after creation
        cout << "Display after Creation: ";
        display();

        return 0;
}
```

**Your Output**

```
Display after Creation: 1 2 3 4 5
```

## 2: Insertion at Start in Double Circular Linked List

**Code:**

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

// Define a structure for a node in a doubly circular linked list
struct node
{
        int data;          // Data stored in the node
        struct node *next; // Pointer to the next node
        struct node *prev; // Pointer to the previous node
```

```
};

// Global pointers for managing the doubly circular linked list
struct node *n, *first, *last, *current;

// Function to create a new node and add it to the doubly circular linked list
void create(int data)
{
	n = new node();    // Allocate memory for a new node
	n->data = data;    // Assign data to the node

	if(first == NULL)  // If the list is empty
	{
		n->next = n;   // Point the node's `next` to itself (circular link)
		n->prev = n;   // Point the node's `prev` to itself
		first = last = n; // Set the new node as the first and last node
	}
	else               // If the list is not empty
	{
		n->next = first; // Point the new node's `next` to the first node
		n->prev = last;  // Point the new node's `prev` to the last node
		last->next = n;  // Update the last node's `next` to the new node
		first->prev = n; // Update the first node's `prev` to the new node
		last = n;        // Update the `last` pointer to the new node
	}
}

// Function to insert a new node at the start of the doubly circular linked list
void insertAtStart(int data)
{
	n = new node();     // Allocate memory for a new node
	n->data = data;     // Assign data to the node
	n->next = first;    // Point the new node's `next` to the current first node
	n->prev = last;     // Point the new node's `prev` to the last node
	last->next = n;     // Update the last node's `next` to the new node
	first->prev = n;    // Update the first node's `prev` to the new node
	first = n;          // Update the `first` pointer to the new node
}

// Function to display the data in the doubly circular linked list
void display()
{
	current = first;    // Start from the first node

	do
	{
```

```
		cout << current->data << " "; // Print the data of the current node
		current = current->next;      // Move to the next node
	}
	while(current != first);          // Stop when we loop back to the first node
	cout << endl;                     // Print a newline after displaying all nodes
}

// Main function
int main()
{
	// Create nodes in the doubly circular linked list with the given values
	create(1);
	create(2);
	create(3);
	create(4);
	create(5);

	// Display the linked list before insertion
	cout << "Display before Insertion: ";
	display();

	// Insert a new node at the start of the linked list
	insertAtStart(19);

	// Display the linked list after insertion
	cout << "Display after Insertion: ";
	display();

	return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 19 1 2 3 4 5
```

## 3: Insertion at End in Double Circular Linked List

**Code:**

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

// Define a structure for a node in a doubly circular linked list
struct node
{
	int data;         // Data stored in the node
```

```
    struct node *next; // Pointer to the next node
    struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly circular linked list
struct node *n, *first, *last, *current;

// Function to create a new node and add it to the doubly circular linked list
void create(int data)
{
    n = new node();     // Allocate memory for a new node
    n->data = data;     // Assign data to the node

    if(first == NULL)   // If the list is empty
    {
        n->next = n;    // Point the node's `next` to itself (circular link)
        n->prev = n;    // Point the node's `prev` to itself
        first = last = n; // Set the new node as both the first and last node
    }
    else               // If the list is not empty
    {
        n->next = first; // Point the new node's `next` to the first node
        n->prev = last;  // Point the new node's `prev` to the last node
        last->next = n;  // Update the last node's `next` to point to the new node
        first->prev = n; // Update the first node's `prev` to point to the new node
        last = n;        // Update the `last` pointer to the new node
    }
}

// Function to insert a new node at the end of the doubly circular linked list
void insertAtEnd(int data)
{
    n = new node();      // Allocate memory for a new node
    n->data = data;      // Assign data to the node
    n->next = first;     // Point the new node's `next` to the first node
    n->prev = last;      // Point the new node's `prev` to the last node
    last->next = n;      // Update the last node's `next` to point to the new node
    first->prev = n;     // Update the first node's `prev` to point to the new node
    last = n;            // Update the `last` pointer to the new node
}

// Function to display the data in the doubly circular linked list
void display()
{
    current = first;     // Start from the first node
```

```
        do
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next;      // Move to the next node
        }
        while(current != first);          // Stop when we loop back to the first node
        cout << endl;                     // Print a newline after displaying all nodes
}

// Main function
int main()
{
        // Create nodes in the doubly circular linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before insertion
        cout << "Display before Insertion: ";
        display();

        // Insert a new node at the end of the linked list
        insertAtEnd(19);

        // Display the linked list after insertion
        cout << "Display after Insertion: ";
        display();

        return 0;
}
```

```
Display before Insertion: 1 2 3 4 5
Display after Insertion: 1 2 3 4 5 19
```

## 4: Deletion at End in Double Circular Linked List

**Code:**

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

// Define a structure for a node in a doubly circular linked list
struct node
{
```

```
    int data;       // Data stored in the node
    struct node *next; // Pointer to the next node
    struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the doubly circular linked list
void create(int data)
{
    n = new node();     // Allocate memory for a new node
    n->data = data;     // Assign data to the node

    if(first == NULL)   // If the list is empty
    {
        n->next = n;    // Point the node's `next` to itself (circular link)
        n->prev = n;    // Point the node's `prev` to itself
        first = last = n; // Set the new node as both the first and last node
    }
    else            // If the list is not empty
    {
        n->next = first; // Point the new node's `next` to the first node
        n->prev = last;  // Point the new node's `prev` to the last node
        last->next = n;  // Update the last node's `next` to point to the new node
        first->prev = n; // Update the first node's `prev` to point to the new node
        last = n;        // Update the `last` pointer to the new node
    }
}

// Function to delete the last node from the doubly circular linked list
void deleteAtEnd()
{
    temp = last;          // Store the last node in a temporary pointer
    last = last->prev;    // Update the `last` pointer to the previous node
    last->next = first;   // Update the last node's `next` to point to the first node
    first->prev = last;   // Update the first node's `prev` to point to the new last node
    delete(temp);         // Delete the old last node (free memory)
}

// Function to display the data in the doubly circular linked list
void display()
{
    current = first;      // Start from the first node

    do
```

```
        {
                cout << current->data << " "; // Print the data of the current node
                current = current->next;      // Move to the next node
        }
        while(current != first);          // Stop when we loop back to the first node
        cout << endl;
}

// Main function
int main()
{
        // Create nodes in the doubly circular linked list with the given values
        create(1);
        create(2);
        create(3);
        create(4);
        create(5);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Delete the last node in the doubly circular linked list
        deleteAtEnd();

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

Your Output

```
Display before Deletion: 1 2 3 4 5
Display after Deletion: 1 2 3 4
```

## 5: Deletion at Mid in Double Circular Linked List

**Code:**

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

// Define a structure for a node in a doubly circular linked list
struct node
{
```

```
    int data;          // Data stored in the node
    struct node *next; // Pointer to the next node
    struct node *prev; // Pointer to the previous node
};

// Global pointers for managing the doubly circular linked list
struct node *n, *first, *last, *current, *temp;

// Function to create a new node and add it to the doubly circular linked list
void create(int data)
{
    n = new node();     // Allocate memory for a new node
    n->data = data;     // Assign data to the node

    if(first == NULL)   // If the list is empty
    {
        n->next = n;    // Point the node's `next` to itself (circular link)
        n->prev = n;    // Point the node's `prev` to itself
        first = last = n; // Set the new node as both the first and last node
    }
    else                // If the list is not empty
    {
        n->next = first; // Point the new node's `next` to the first node
        n->prev = last;  // Point the new node's `prev` to the last node
        last->next = n;  // Update the last node's `next` to point to the new node
        first->prev = n; // Update the first node's `prev` to point to the new node
        last = n;        // Update the `last` pointer to the new node
    }
}

// Function to delete a node from the list based on the value
void deleteAtMid(int value)
{
    current = first;  // Start from the first node

    do
    {
        if(current->data == value)  // If the current node contains the value to be deleted
        {
            if(current == first)    // If the node to be deleted is the first node
            {
                first = first->next;   // Update the first pointer to the next node
                last->next = first;    // Update the last node's `next` to point to the new
                first node
                first->prev = last;    // Update the new first node's `prev` to point to the
                last node
```

```
                }
                else if(current == last) // If the node to be deleted is the last node
                {
                        last = last->prev;      // Update the last pointer to the previous node
                        last->next = first;     // Update the last node's `next` to point to the first
                        node
                        first->prev = last;     // Update the first node's `prev` to point to the new
                        last node
                }
                else                    // If the node to be deleted is a middle node
                {
                        current->prev->next = current->next;  // Bypass the current node in the
                        `next` direction
                        current->next->prev = current->prev;  // Bypass the current node in the
                        `prev` direction
                }
                delete(current);           // Delete the current node (free memory)
                return;                    // Exit after the node is deleted
        }
        current = current->next;    // Move to the next node
    }
    while(current != first);         // Stop when we loop back to the first node
}

// Function to display the data in the doubly circular linked list
void display()
{
    current = first;      // Start from the first node

    do
    {
        cout << current->data << " "; // Print the data of the current node
        current = current->next;      // Move to the next node
    }
    while(current != first);          // Stop when we loop back to the first node
    cout << endl;
}

// Main function
int main()
{
    // Create a doubly circular linked list with the given values
    create(1);
    create(2);
    create(3);
    create(4);
```

```
        create(5);

        // Display the linked list before deletion
        cout << "Display before Deletion: ";
        display();

        // Input the value of the node to be deleted
        int value;
        cout << "Enter the Value to Delete: ";
        cin >> value;

        // Delete the node with the given value from the linked list
        deleteAtMid(value);

        // Display the linked list after deletion
        cout << "Display after Deletion: ";
        display();

        return 0;
}
```

```
Display before Deletion: 1 2 3 4 5
Enter the Value to Delete: Display after Deletion: 1 2 3 4 5
```

# Lab 13

# Implementation of Binary Search Tree

## 1: Insertion in Binary Search Tree

**Code:**

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

// Structure for node
struct Node
{
	int data;		// Value of the node
	struct Node* left = NULL;  // Pointer to the left child
	struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
	if (root == NULL)  // If empty tree or reaching a leaf node
	{
		root = new Node(); // Create a new node
		root->data = value; // Assign the value
		return root;	// Return the new node as the root
	}

	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
	{
		root->left = insert(root->left, value);
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = insert(root->right, value);
	}

	return root; // Return the root after insertion
}

// Function for in-order traversal
```

```
void inorder(Node* root)
{
	if (root == NULL)   // If tree is empty
	{
		return;
	}

	inorder(root->left);         // Traverse the left subtree
	cout << root->data << " ";   // Print the current node's data
	inorder(root->right);        // Traverse the right subtree
}

int main()
{
	Node* root = NULL; // Initialize an empty BST

	// Insert nodes into the BST
	root = insert(root, 9);
	root = insert(root, 3);
	root = insert(root, 52);
	root = insert(root, 28);
	root = insert(root, 6);
	root = insert(root, 16);

	// Display the tree nodes using in-order traversal
	cout << "Display In-Order: ";
	inorder(root);

	return 0;
}
```

```
Display In-Order: 3 6 9 16 28 52
```

## 2: Deletion in Binary Search Tree

### Code:

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

// Structure for Node
struct Node
{
	int data;               // Value of the node
	struct Node* left = NULL;  // Pointer to the left child
```

```
	struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
	if (root == NULL)   // If tree is empty or reaching a leaf
	{
		root = new Node();  // Create a new node
		root->data = value; // Assign the value to the node
		return root;        // Return the new node as the root
	}

	// Recur on the left subtree if the value is smaller
	else if (value < root->data)
	{
		root->left =insert(root->left, value);
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = insert(root->right, value);
	}

	return root; // Return the root after insertion
}

// Function for in-order successor
Node* getSuccessor(Node* root)
{
	root = root->right; // Start from the right subtree
	while (root != nullptr && root->left != nullptr)
	{
		root = root->left; // Traverse left to find the smallest value
	}
	return root; // Return the in-order successor
}

// Function for Deletion
Node* deletion(Node* root, int value)
{
	if (root == NULL)   // If tree is empty
	{
		return root;
	}

	// Recur on the left subtree if the value is smaller
	if (value < root->data)
	{
		root->left = deletion(root->left, value);
```

```
	}
	// Recur on the right subtree if the value is larger
	else if (value > root->data)
	{
		root->right = deletion(root->right, value);
	}
	// Node to be deleted is found
	else
	{
		// Case 1: Node has no children or only one child
		if (root->left == NULL)
		{
			Node* temp = root->right; // Replace with right child
			delete root;            // Delete the node
			return temp;
		}
		else if (root->right == NULL)
		{
			Node* temp = root->left; // Replace with left child
			delete root;           // Delete the node
			return temp;
		}

		// Case 2: Node has two children
		Node* temp = getSuccessor(root); // Find in-order successor
		root->data = temp->data;       // Replace data with successor's value
		root->right = deletion(root->right, temp->data); // Delete successor
	}
	return root; // Return the updated root
}

// Function for in-order traversal
void inorder(Node* root)
{
	if (root == NULL)    // If tree is empty
	{
		return;
	}

	inorder(root->left);        // Traverse the left subtree
	cout << root->data << " ";   // Print the current node's data
	inorder(root->right);       // Traverse the right subtree
}

int main()
{
	Node* root = NULL; // Initialize an empty BST

	// Insert nodes into the BST
	root = insert(root, 9);
```

```
    root = insert(root, 33);
    root = insert(root, 2);
    root = insert(root, 28);
    root = insert(root, 6);
    root = insert(root, 6);

    // Display the BST before deletion
    cout << "Before Deletion: ";
    inorder(root);

    // Delete nodes with values 52 and 33
    root = deletion(root, 52);
    root = deletion(root, 33);

    // Display the BST after deletion
    cout << "\nAfter Deletion: ";
    inorder(root);

    return 0;
}
```

```
Before Deletion: 2 6 9 28 33
After Deletion: 2 6 9 28
```

## 3: Searching in Binary Search Tree

**Code:**

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

// Structure for node
struct Node
{
    int data;               // Value of the node
    struct Node* left = NULL;  // Pointer to the left child
    struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
    if (root == NULL)   // If tree is empty or reaching a leaf
    {
        root = new Node();  // Create a new node
        root->data = value; // Assign the value to the node
```

```
        return root;        // Return the new node as the root
    }

    // Recur on the left subtree if the value is smaller
    else if (value < root->data)
    {
            root->left = insert(root->left, value);
    }
    // Recur on the right subtree if the value is larger
    else if (value > root->data)
    {
        root->right = insert(root->right, value);
    }

    return root; // Return the root after insertion
}

// Function for searching
Node* searching(Node* root, int value)
{
    // If tree is empty or value matches the current node
    if (root == NULL || value == root->data)
    {
        return root;
    }
    // Recur on the left subtree if the value is smaller
    else if (value < root->data)
    {
        return searching(root->left, value);
    }
    // Recur on the right subtree if the value is larger
    else
    {
        return searching(root->right, value);
    }
}

int main()
{
    Node* root = NULL; // Initialize an empty BST

    // Insert nodes into the BST
    root = insert(root, 19);
    root = insert(root, 5);
    root = insert(root, 46);
    root = insert(root, 9);
```

```
    root = insert(root, 25);
    root = insert(root, 9);

    // Search for a value in the BST
    cout << "Searching Node: ";
    Node* search = searching(root, 46); // Search for the value 46

    // Display search result
    if (search != NULL)
    {
        cout << "Value Exists!";
    }
    else
    {
        cout << "Value Doesn't Exist!";
    }

    return 0;
}
```

```
Searching Node: Value Exists!
```

## 4: Duplication in Binary Search Tree

**Code:**

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

// Structure for node
struct Node
{
    int data;                // Value of the node
    int count;           // Count of occurrences of the value
    struct Node* left = NULL;  // Pointer to the left child
    struct Node* right = NULL; // Pointer to the right child
};

// Function for Insertion
Node* insert(Node* root, int value)
{
    if (root == NULL)    // If tree is empty or reaching a leaf
    {
        root = new Node();  // Create a new node
        root->data = value; // Assign the value to the node
        return root;        // Return the new node as the root
```

```
    }

    // If the value already exists, increment its count
    else if (value == root->data)
    {
        root->count++;
        return root;
    }
    // Recur on the left subtree if the value is smaller
    else if (value < root->data)
    {
        root->left = insert(root->left, value);
    }
    // Recur on the right subtree if the value is larger
    else
    {
        root->right = insert(root->right, value);
    }

    return root; // Return the root after insertion
}

// Function for in-order traversal
void inorder(Node* root)
{
    if (root == NULL)    // If tree is empty
    {
        return;
    }

    inorder(root->left);         // Traverse the left subtree
    cout << root->data << "(" << root->count << ") "; // Print the data and its count
    inorder(root->right);        // Traverse the right subtree
}

int main()
{
    Node* root = NULL; // Initialize an empty BST

    // Insert nodes into the BST
    root = insert(root, 10);
    root = insert(root, 7);
    root = insert(root, 3);
    root = insert(root, 6);
    root = insert(root, 37);
    root = insert(root, 49);
    root = insert(root, 10); // Duplicate value
    root = insert(root, 7); // Duplicate value
    root = insert(root, 3); // Duplicate value
```

```
    // Display the BST in in-order traversal
    cout << "Display In-Order: ";
    inorder(root);

    return 0; // Exit the program
}
```

```
Display In-Order: 3(1) 6(0) 7(1) 10(1) 37(0) 49(0)
```