**Lab - 1**←

SEARCHING AND SORTING

1. Write a C++ menu-driven program to sort a given array in ascending order. Design proper functions,

maintain boundary conditions and follow coding best practices. The menus are as follows,

a. Bubble Sort

b. Selection Sort

c. Insertion Sort

d. Exit

## Aim:

To write a C++ menu-driven program to sort a given array in ascending order.

## Algorithms:

Algorithm 1- Bubble Sort

**Input:**

arr[] - list of integers

size - number of elements in arr

**Output:**

arr[] - sorted in ascending order

for i from 1 to size - 1

for j from 1 to size - i

if arr[j] > arr[j + 1]

temp ← arr[j]

arr[j] ← arr[j + 1]

arr[j + 1] ← temp

return arr[]

Algorithm 2- Selection Sort

**Input:**

arr[] - list of integers

size - number of elements in list

**Output:**

arr[] - sorted in ascending order

for i from 1 to size - 1

min ← i

for j from i + 1 to size

if arr[j] < arr[min]

min ← j

if min ≠ i

temp ← arr[i]

arr[i] ← arr[min]

arr[min] ← temp

return arr[]

**Algorithm 3- Insertion Sort**

**Input:**

arr[] - list of integers

size - number of elements in arr

**Output:**

arr[] - sorted in ascending order

for i from 2 to size

key ← arr[i]

j ← i - 1

while j ≥ 1 and arr[j] > key

arr[j + 1] ← arr[j]

j ← j - 1

arr[j + 1] ← key

return arr[]

## Code:

// Menu driven program for sorting

#include <stdio.h>

// insertion sort

void insertion(int arr[], int num)

{

    for (int i = 0; i < num; i++)

    {

        int j = i - 1;

        int temp = arr[i];

        while (j >= 0 && arr[j] > temp)

        {

            arr[j + 1] = arr[j];

            j--;

        }

        arr[j + 1] = temp;

    }

}

// Bubble Sort

void bubble(int arr[], int num)

{

    int temp1;

    for (int i = 0; i < num; i++)

    {

        for (int j = 0; j < num - i - 1; j++)

        {

            if (arr[j] > arr[j + 1])

            {

                temp1 = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp1;

            }

        }

    }

}

// Selection Sort

void selection(int arr[], int num)

{

    int temp2;

    for (int i = 0; i < num - 1; i++)

    {

        for (int j = i + 1; j < num; j++)

        {

            if (arr[j]<arr[i])

            {

                temp2 = arr[j];

                arr[j] = arr[i];

                arr[i] = temp2;

            }

        }

    }

}

int main()

{

    int num;

    int choice;

    printf("Enter the number of numbers to be entered\n");

    scanf("%d", &num);

    int arr[num];

    printf("Enter the numbers\n");

    for (int i = 0; i < num; i++)

    {

        scanf("%d", &arr[i]);

    }

    printf("\n");

    printf("1. Insertion Sort\n2. Bubble Sort\n3. Selection Sort\n4. Exit\n");

    scanf("%d", &choice);

    switch (choice)

    {

    case 1:

        insertion(arr, num);

        break;

    case 2:

        bubble(arr, num);

        break;

    case 3:

        selection(arr, num);

        break;

    case 4:

        break;

    }

    for (int i = 0; i < num; i++)

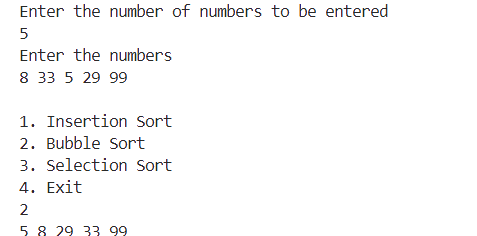
    {

        printf("%d", arr[i]);

    }

}

## Output:



2) Convert the sorting program into a header file and include it into a new cpp file. Write a C++ menu-

driven program for linear and binary search in this new cpp file. Utilize any of the sorting functions in the

included header file to sort the input array before performing a binary search. Design proper functions,

maintain boundary conditions and follow coding best practices. The menu-driven program supports,

a. Linear Search

b. Binary Search

c. Exit

## Aim:

To write a C++ menu-driven program for linear and binary search using functions from an included header file containing sorting algorithms. The input array is sorted before binary search using one of the sorting functions.

## Algorithms:

### ****Algorithm 1 – Bubble Sort**** (from header file)

**Input:**  
arr[] – list of integers  
size – number of elements in arr

**Output:**  
arr[] – sorted in ascending order

for i from 1 to size - 1

for j from 1 to size - i

if arr[j] > arr[j + 1]

temp ← arr[j]

arr[j] ← arr[j + 1]

arr[j + 1] ← temp

return arr[]

### ****Algorithm 2 – Linear Search****

**Input:**  
arr[] – list of integers  
size – number of elements in arr  
key – value to search

**Output:**  
Index of key if found; otherwise -1

for i from 0 to size - 1

if arr[i] == key

return i

return -1

### ****Algorithm 3 – Binary Search****

**Input:**  
arr[] – sorted list of integers  
size – number of elements in arr  
key – value to search

**Output:**  
Index of key if found; otherwise -1

low ← 0

high ← size - 1

while low ≤ high

mid ← (low + high) / 2

if arr[mid] == key

return mid

else if arr[mid] < key

low ← mid + 1

else

high ← mid - 1

return -1

## Code:

#include <stdio.h>

#include "headerfile.h"

void linear(int arr[], int num, int value)

{

    int found = 0;

    for (int i = 0; i < num; i++)

    {

        if (arr[i] == value)

        {

            printf("Found\n");

            found = 1;

            break;

        }

        else

        {

            found = 0;

        }

    }

    if (found == 0)

    {

        printf("Not found\n");

    }

}

void binary(int arr[], int num, int value)

{

    int start = 0, end = num - 1, mid;

    while (start <= end)

    {

        mid = (start + end) / 2; // Calculate mid each time

        if (arr[mid] == value)

        {

            printf("Number found\n");

            return; // Exit the function if the number is found

        }

        else if (arr[mid] > value)

        {

            end = mid - 1; // Search the left half

        }

        else

        {

            start = mid + 1; // Search the right half

        }

    }

    printf("Number not found\n"); // Number not found if loop exits

}

int main()

{

    int choice;

    int num;

    int value;

    printf("Enter the number of numbers to be entered\n");

    scanf("%d", &num);

    int arr[num];

    printf("Enter the numbers\n");

    for (int i = 0; i < num; i++)

    {

        scanf("%d", &arr[i]);

    }

    printf("\n");

    printf("1. Linear Search\n2. Binary search\n");

    scanf("%d", &choice);

    printf("Enter the number to search\n");

    scanf("%d", &value);

    switch (choice)

    {

    case 1:

        linear(arr, num, value);

        break;

    case 2:

        insertion(arr, num);

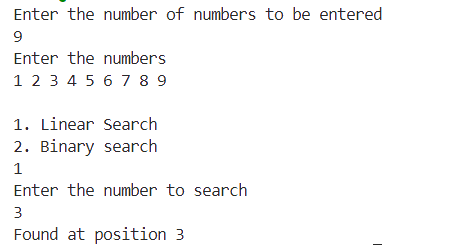
        binary(arr, num, value);

        break;

    }

}

## Output:



**Lab - 2**

1. Write a C++ menu-driven program to determine whether a number is a Palindrome, Armstrong, or Perfect Number. Normal variable and array declarations are not allowed. Utilize dynamic memory allocation (DMA). Design proper functions, maintain boundary conditions, and follow coding best practices. The menu is as follows,

a. Palindrome  
b. Armstrong Number  
c. Perfect Number  
d. Exit

## Aim:

To write a C++ menu-driven program using dynamic memory allocation (DMA) to determine whether a number is a Palindrome, Armstrong Number, or Perfect Number. The program avoids normal variable and array declarations, and uses proper functions and coding practices.

## Algorithms:

### ****Algorithm 1 – Palindrome Check****

**Input:**  
num – dynamically allocated integer

**Output:**  
True if num is a palindrome; otherwise false

Allocate memory for original and reversed number

\*original ← \*num

\*reversed ← 0

\*temp ← \*num

while \*temp ≠ 0

digit ← \*temp % 10

\*reversed ← \*reversed × 10 + digit

\*temp ← \*temp / 10

if \*original == \*reversed

return true

else

return false

### ****Algorithm 2 – Armstrong Number Check****

**Input:**  
num – dynamically allocated integer

**Output:**  
True if num is an Armstrong number; otherwise false

Allocate memory for sum and temp

\*sum ← 0

\*temp ← \*num

n ← number of digits in \*num

while \*temp ≠ 0

digit ← \*temp % 10

\*sum ← \*sum + (digit ^ n)

\*temp ← \*temp / 10

if \*sum == \*num

return true

else

return false

### ****Algorithm 3 – Perfect Number Check****

**Input:**  
num – dynamically allocated integer

**Output:**  
True if num is a perfect number; otherwise false

Allocate memory for sum

\*sum ← 0

for i from 1 to \*num - 1

if \*num % i == 0

\*sum ← \*sum + i

if \*sum == \*num

return true

else

return false

## Code:

#include <iostream>

#include <string>

#include <cmath>

#include <cstring> // For strlen function

using std::cin;

using std::cout;

using std::endl;

int palindrome()

{

    char \*arr = new char[100];

    int i;

    int length = 0;

    int temp;

    bool found = true; // Using boolean for clarity

    cout << "Enter the word: ";

    cin >> arr;

    // Calculate string length correctly using strlen

    length = strlen(arr);

    temp = length - 1;

    // Check for palindrome

    for (i = 0; i < length/2; i++)

    {

        if (arr[i] != arr[temp])

        {

            found = false;

            break;

        }

        temp--;

    }

    if (found)

    {

        cout << "It is a Palindrome" << endl;

    }

    else

    {

        cout << "It is not a Palindrome" << endl;

    }

    // Free allocated memory

    delete[] arr;

    return 0;

}

int armstrong()

{

    int number;

    int digit;

    int sum = 0;

    int temp;

    int count = 0;

    cout << "Enter the number: ";

    cin >> number;

    temp = number;

    // Count the number of digits

    while (temp != 0)

    {

        temp = temp / 10;

        count++;

    }

    temp = number;

    // Calculate the sum of digits raised to the power of count

    while (temp != 0)

    {

        digit = temp % 10;

        temp = temp / 10;

        sum += pow(digit, count);

    }

    if (sum == number)

    {

        cout << "It is an Armstrong number" << endl;

    }

    else

    {

        cout << "It is not an Armstrong number" << endl;

    }

    return 0;

}

int perfect()

{

    int number;

    int sum = 0;

    cout << "Enter the number: ";

    cin >> number;

    // Find the sum of divisors

    for (int i = 1; i < number; i++)

    {

        if (number % i == 0)

        {

            sum += i;

        }

    }

    if (sum == number)

    {

        cout << "It is a perfect number" << endl;

    }

    else

    {

        cout << "It is not a perfect number" << endl;

    }

    return 0;

}

int main()

{

    int choice;

    bool exit = false;

    while (!exit)

    {

        cout << "\nEnter the choice" << endl;

        cout << "1. Palindrome" << endl;

        cout << "2. Armstrong" << endl;

        cout << "3. Perfect Number" << endl;

        cout << "4. Exit" << endl;

        cout << "Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            palindrome();

            break;

        case 2:

            armstrong();

            break;

        case 3:

            perfect();

            break;

        case 4:

            exit = true;

            break;

        default:

            cout << "Invalid choice! Please try again." << endl;

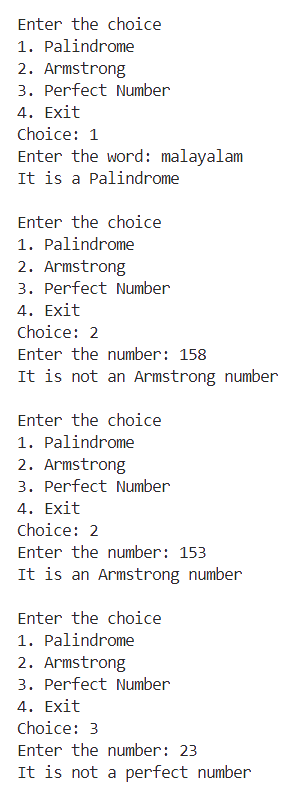
        }

    }

    return 0;

}

## Output:



2. Write a C++ menu-driven program that calculates and displays the area of a square, cube, rectangle, and cuboid. Consider length as the side value for the square and cuboid. Identify proper data members and member functions. Design and create an appropriate class for the given scenario. Maintain proper boundary conditions and follow coding best practices. The menus are as follows, a) Square b) Cube c) Rectangle d) Cuboid e) Exit

## ****AIM:****

To write a C++ menu-driven program that calculates and displays the area of a square, cube, rectangle, and cuboid using a class with appropriate data members and member functions. The program follows coding best practices and proper boundary conditions.

## Algorithms:

### ****Algorithm 1 – Area of a Square****

**Input:**  
length – side length of the square

**Output:**  
area – area of the square

area ← length × length

return area

### ****Algorithm 2 – Surface Area of a Cube****

**Input:**  
length – side length of the cube

**Output:**  
area – surface area of the cube

area ← 6 × length × length

return area

### ****Algorithm 3 – Area of a Rectangle****

**Input:**  
length – length of the rectangle  
breadth – breadth of the rectangle

**Output:**  
area – area of the rectangle

area ← length × breadth

return area

### ****Algorithm 4 – Surface Area of a Cuboid****

**Input:**  
length – length of the cuboid  
breadth – breadth of the cuboid  
height – height of the cuboid

**Output:**  
area – surface area of the cuboid

area ← 2 × (length × breadth + breadth × height + height × length)

return area

## Code:

#include <iostream>

class shape

{

    int len, bre, hei;

public:

    shape()

    {

        len = 1;

        bre = 1;

        hei = 1;

    }

    void setlength(int);

    void setbreadth(int);

    void setheight(int);

    int getlength()

    {

        return len;

    }

    int getbreadth()

    {

        return bre;

    }

    int getheight()

    {

        return hei;

    }

};

void shape::setlength(int length)

{

    len = length;

}

void shape::setbreadth(int breadth)

{

    bre = breadth;

}

void shape::setheight(int height)

{

    hei = height;

}

int main()

{

    class shape s;

    int length;

    int breadth;

    int height;

    int area;

    int volume;

    int exit = 1;

    int choice;

    while (exit == 1)

    {

        std::cout << "Enter the choice\n"

                  << "1. Square\n2. Rectangle\n3. Cube\n4. Cuboid\n5. exit\n";

        std::cin >> choice;

        switch (choice)

        {

        case 1:

            std::cout << "Enter the Length\n";

            std::cin >> length;

            s.setlength(length);

            std::cout << "Enter the Breadth\n";

            std::cin >> breadth;

            s.setbreadth(breadth);

            area = (s.getlength()) \* (s.getbreadth());

            std::cout << "The area of Square is" << area << '\n';

            break;

        case 2:

            std::cout << "Enter the Length\n";

            std::cin >> length;

            s.setlength(length);

            std::cout << "Enter the Breadth\n";

            std::cin >> breadth;

            s.setbreadth(breadth);

            area = (s.getlength()) \* (s.getbreadth());

            std::cout << "The area of Rectangle is" << area << '\n';

            break;

        case 3:

            std::cout << "Enter the Length\n";

            std::cin >> length;

            s.setlength(length);

            std::cout << "Enter the Breadth\n";

            std::cin >> breadth;

            s.setbreadth(breadth);

            std::cout << "Enter the Height\n";

            std::cin >> height;

            volume = (s.getlength()) \* (s.getbreadth()) \* (s.getheight());

            std::cout << "The volume of the cube is" << volume << '\n';

            break;

        case 4:

            std::cout << "Enter the Length\n";

            std::cin >> length;

            s.setlength(length);

            std::cout << "Enter the Breadth\n";

            std::cin >> breadth;

            s.setbreadth(breadth);

            std::cout << "Enter the Height\n";

            std::cin >> height;

            volume = (s.getlength()) \* (s.getbreadth()) \* (s.getheight());

            std::cout << "The volume of the cuboid is" << volume << '\n';

            break;

        case 5:

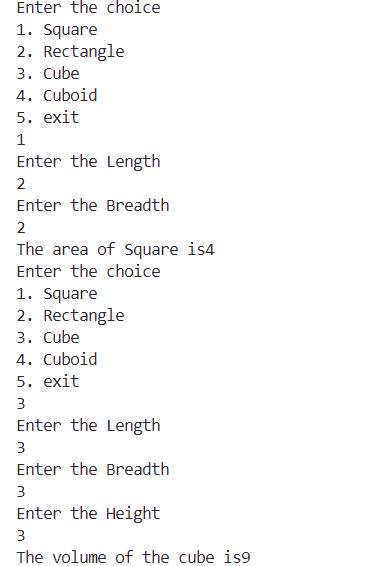
            exit = 0;

        }

    }

}

## Output:



**Lab - 3**

## Aim:

To write a C++ menu-driven program to implement the List Abstract Data Type (ADT) using a fixed-size array of 5 elements, supporting various insertions, deletions, search, display, and rotation operations. The program includes multiple solutions for the rotate operation, including an approach with **O(1) extra space**.

## Algorithms:

**Input:**  
arr[] – array of maximum size 5  
n – current number of elements in the list  
choice – operation selected by user  
Other inputs depend on operation

**Boundary Conditions:**

* No insertion if n == 5
* No deletion if n == 0
* Valid index range: 0 ≤ position < n for deletion/search, 0 ≤ position ≤ n for insertion

**Algorithm 1 – Insert at Beginning**

if n == 5

print "List is full"

return

for i from n - 1 down to 0

arr[i + 1] ← arr[i]

arr[0] ← new\_element

n ← n + 1

**Algorithm 2 – Insert at End**

if n == 5

print "List is full"

return

arr[n] ← new\_element

n ← n + 1

**Algorithm 3 – Insert at Position**

## if n == 5

## print "List is full"

## return

## if position < 0 or position > n

## print "Invalid position"

## return

## for i from n - 1 down to position

## arr[i + 1] ← arr[i]

## arr[position] ← new\_element

## n ← n + 1

**Algorithm 4 – Delete from Beginning**

if n == 0

print "List is empty"

return

for i from 0 to n - 2

arr[i] ← arr[i + 1]

n ← n – 1

**Algorithm 5 – Delete from End**

if n == 0

print "List is empty"

return

n ← n – 1

**Algorithm 6 – Delete at Position**

if n == 0

print "List is empty"

return

if position < 0 or position ≥ n

print "Invalid position"

return

for i from position to n - 2

arr[i] ← arr[i + 1]

n ← n – 1

**Algorithm 7 – Search**

for i from 0 to n - 1

if arr[i] == key

return i

return -1

**Algorithm 8 – Display**

if n == 0

print "List is empty"

return

for i from 0 to n - 1

print arr[i]

**Algorithm 9 – Rotate Right by k (Method 1: Using Extra Array)**

k ← k % n

Create temp[] of size n

for i from 0 to n - 1

temp[(i + k) % n] ← arr[i]

for i from 0 to n - 1

arr[i] ← temp[i]

## Algorithm 10 – Rotate Right by k (Method 2: Repeatedly Shift One Element)

repeat k times

temp ← arr[n - 1]

for i from n - 1 down to 1

arr[i] ← arr[i - 1]

arr[0] ← temp

**Algorithm 11 – Rotate Right by k (Method 3: Using Reverse, O(1) Space)**

k ← k % n

Reverse(arr, 0, n - 1)

Reverse(arr, 0, k - 1)

Reverse(arr, k, n - 1)

Procedure Reverse(arr, start, end):

while start < end

temp ← arr[start]

arr[start] ← arr[end]

arr[end] ← temp

start ← start + 1

end ← end - 1

## Code:

#include <iostream>

using std::cin;

using std::cout;

#define SIZE 5

class List {

    int arr[SIZE];

    int count;

public:

    List() {

        count = 0;

    }

    void insert\_beginning(int);

    void insert\_end(int);

    void insert\_position(int, int);

    void delete\_beginning();

    void delete\_end();

    void delete\_position(int);

    void search(int);

    void rotate(int);

    void display();

};

// Insert at the beginning

void List::insert\_beginning(int value) {

    if (count == SIZE) {

        cout << "The list is full\n";

        return;

    }

    for (int i = count; i > 0; i--) {

        arr[i] = arr[i - 1];

    }

    arr[0] = value;

    count++;

}

// Insert at the end

void List::insert\_end(int value) {

    if (count == SIZE) {

        cout << "The list is full\n";

        return;

    }

    arr[count++] = value;

}

// Insert at a given position

void List::insert\_position(int value, int position) {

    if (count == SIZE || position < 1 || position > count + 1) {

        cout << "Invalid position or list is full\n";

        return;

    }

    for (int i = count; i >= position; i--) {

        arr[i] = arr[i - 1];

    }

    arr[position - 1] = value;

    count++;

}

// Delete from the beginning

void List::delete\_beginning() {

    if (count == 0) {

        cout << "The list is empty\n";

        return;

    }

    for (int i = 0; i < count - 1; i++) {

        arr[i] = arr[i + 1];

    }

    count--;

}

// Delete from the end

void List::delete\_end() {

    if (count == 0) {

        cout << "The list is empty\n";

        return;

    }

    count--;

}

// Delete at a given position

void List::delete\_position(int position) {

    if (count == 0 || position < 1 || position > count) {

        cout << "Invalid position or list is empty\n";

        return;

    }

    for (int i = position - 1; i < count - 1; i++) {

        arr[i] = arr[i + 1];

    }

    count--;

}

// Search for an element

void List::search(int value) {

    if (count == 0) {

        cout << "The list is empty\n";

        return;

    }

    for (int i = 0; i < count; i++) {

        if (arr[i] == value) {

            cout << "Value found at position " << i + 1 << "\n";

            return;

        }

    }

    cout << "Value not found\n";

}

// Rotate the list right by k times using O(1) space

void List::rotate(int k) {

    if (count == 0) {

        cout << "The list is empty\n";

        return;

    }

    k = k % count;  // Handle cases where k > count

    if (k == 0) return;

    // Reverse the entire array

    for (int i = 0, j = count - 1; i < j; i++, j--) {

        std::swap(arr[i], arr[j]);

    }

    // Reverse the first k elements

    for (int i = 0, j = k - 1; i < j; i++, j--) {

        std::swap(arr[i], arr[j]);

    }

    // Reverse the remaining elements

    for (int i = k, j = count - 1; i < j; i++, j--) {

        std::swap(arr[i], arr[j]);

    }

}

// void List::rotate(int k) {

//     if (count == 0) {

//         cout << "The list is empty\n";

//         return;

//     }

//     k = k % count;  // Handle cases where k > count

//     if (k == 0) return;

//     for (int i = 0; i < k; i++) {

//         int lastElement = arr[count - 1];  // Store the last element

//         delete\_end();                      // Remove the last element

//         insert\_beginning(lastElement);     // Insert it at the beginning

//     }

// }

// Display the list

void List::display() {

    if (count == 0) {

        cout << "The list is empty\n";

        return;

    }

    cout << "List: ";

    for (int i = 0; i < count; i++) {

        cout << arr[i] << " ";

    }

    cout << "\n";

}

int main() {

    List l;

    int exit = 1;

    while (exit) {

        int choice, value, position, k;

        cout << "\nEnter choice:\n"

             << "1. Insert Beginning\n2. Insert End\n3. Insert Position\n"

             << "4. Delete Beginning\n5. Delete End\n6. Delete Position\n"

             << "7. Search\n8. Rotate\n9. Display\n10. Exit\n";

        cin >> choice;

        switch (choice) {

        case 1:

            cout << "Enter value: ";

            cin >> value;

            l.insert\_beginning(value);

            break;

        case 2:

            cout << "Enter value: ";

            cin >> value;

            l.insert\_end(value);

            break;

        case 3:

            cout << "Enter value and position: ";

            cin >> value >> position;

            l.insert\_position(value, position);

            break;

        case 4:

            l.delete\_beginning();

            break;

        case 5:

            l.delete\_end();

            break;

        case 6:

            cout << "Enter position: ";

            cin >> position;

            l.delete\_position(position);

            break;

        case 7:

            cout << "Enter value to search: ";

            cin >> value;

            l.search(value);

            break;

        case 8:

            cout << "Enter number of rotations: ";

            cin >> k;

            l.rotate(k);

            break;

        case 9:

            l.display();

            break;

        case 10:

            exit = 0;

            break;

        default:

            cout << "Invalid choice. Try again.\n";

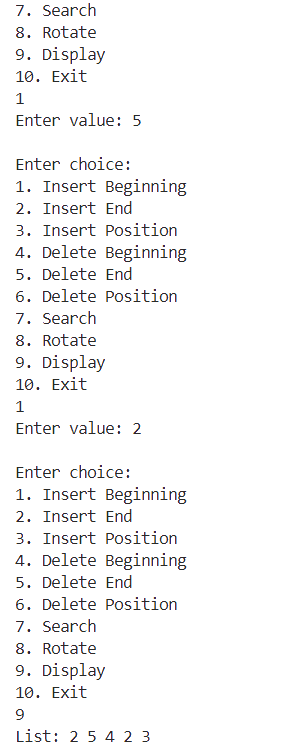
        }

    }

    return 0;

}

## Output:



**Lab - 4**

A. Write a C++ menu-driven program to implement List ADT using a singly linked list. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

1. Insert Beginning
2. Insert End
3. Insert Position
4. Delete Beginning
5. Delete End
6. Delete Position
7. Search
8. Display
9. Display Reverse
10. Reverse Link
11. Exit

## Aim:

To write a C++ menu-driven program to implement the List ADT using a singly linked list with operations including insertion, deletion, search, display, reverse traversal, and reversing the links. The implementation includes proper boundary condition checks and follows best coding practices.

## Algorithms:

**Algorithm 1 – Insert at Beginning**

**Input:**  
value – data to be inserted  
head – pointer to head of list

**Output:**  
Linked list with value inserted at the beginning

Create new\_node with data = value

new\_node.next ← head

head ← new\_node

### ****Algorithm 2 – Insert at End****

**Input:**  
value, head

**Output:**  
Linked list with value inserted at the end

Create new\_node with data = value

new\_node.next ← NULL

if head == NULL

head ← new\_node

return

current ← head

while current.next ≠ NULL

current ← current.next

current.next ← new\_node

**Algorithm 3 – Insert at Position**

**Input:**  
value, position, head

**Output:**  
List with node inserted at given position

if position == 0

Insert at beginning

return

current ← head

for i from 0 to position - 2

if current == NULL

print "Invalid position"

return

current ← current.next

Create new\_node with data = value

new\_node.next ← current.next

current.next ← new\_node

**Algorithm 4 – Delete from Beginning**

if head == NULL

print "List is empty"

return

temp ← head

head ← head.next

Delete temp

**Algorithm 5 – Delete from End**

if head == NULL

print "List is empty"

return

if head.next == NULL

Delete head

head ← NULL

return

current ← head

while current.next.next ≠ NULL

current ← current.next

Delete current.next

current.next ← NULL

**Algorithm 6 – Delete at Position**

**Input:**  
position, head

if head == NULL

print "List is empty"

return

if position == 0

Delete from beginning

return

current ← head

for i from 0 to position - 2

if current == NULL or current.next == NULL

print "Invalid position"

return

current ← current.next

temp ← current.next

current.next ← temp.next

Delete temp

**Algorithm 7 – Search for Element**

**Input:**  
key, head

position ← 0

current ← head

while current ≠ NULL

if current.data == key

return position

current ← current.next

position ← position + 1

return -1 // Not found

**Algorithm 8 – Display**

current ← head

while current ≠ NULL

print current.data

current ← current.next

**Algorithm 9 – Display Reverse (Using Recursion)**

Procedure printReverse(node)

if node == NULL

return

printReverse(node.next)

print node.data

**Algorithm 10 – Reverse the Links of the List**

**Input:**  
head

prev ← NULL

current ← head

while current ≠ NULL

next ← current.next

current.next ← prev

prev ← current

current ← next

head ← prev

## Code:

#include <iostream>

using namespace std;

// Implementing singly linked list

class list

{

    struct node

    {

        int data;

        struct node \*next;

    } \*head;

public:

    list()

    {

        head = NULL;

    }

    // Create New Node

    struct node \*createnewnode(int value)

    {

        struct node \*newnode = new struct node;

        newnode->data = value;

        newnode->next = NULL;

        return newnode;

    }

    // Display

    void display()

    {

        struct node \*temp = head;

        if (head == NULL)

        {

            cout << "The list is empty\n";

        }

        else

        {

            while (temp != NULL)

            {

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

                temp = temp->next;

            }

        }

    }

    // Insert Beginning

    void insert\_beginning(int value)

    {

        struct node \*newnode = createnewnode(value);

        newnode->next = head;

        head = newnode;

    }

    // Insert End

    void insert\_end(int value)

    {

        struct node \*newnode = createnewnode(value);

        if (head == NULL)

        {

            head = newnode;

        }

        else

        {

            struct node \*temp = head;

            while (temp->next != NULL)

            {

                temp = temp->next;

            }

            temp->next = newnode;

        }

    }

    // Insert Position

    void insert\_position(int value, int pos)

    {

        if (pos < 1)

        {

            cout << "Invalid\n";

            return;

        }

        if (pos == 1)

        {

            insert\_beginning(value);

            return;

        }

        int count = 1;

        struct node \*newnode = createnewnode(value);

        struct node \*temp = head;

        while (count < pos - 1 && temp->next != NULL)

        {

            temp = temp->next;

            count++;

        }

        newnode->next = temp->next;

        temp->next = newnode;

    }

    // Delete Beginning

    void delete\_beginning()

    {

        if (head == NULL)

        {

            cout << "The List is empty";

            return;

        }

        struct node \*temp = head;

        head = temp->next;

        free(temp);

    }

    // Delete End

    void delete\_end()

    {

        if (head == NULL)

        {

            cout << "The list is empty\n";

            return;

        }

        if (head->next == NULL)

        {

            head = NULL;

            return;

        }

        struct node \*temp = head;

        while (temp->next->next != NULL)

        {

            temp = temp->next;

        }

        free(temp->next);

        temp->next = NULL;

    }

    // Delete Position

    void delete\_position(int pos)

    {

        if (pos < 1 || head == NULL)

        {

            cout << "Invalid input\n";

            return;

        }

        if (pos == 1)

        {

            delete\_beginning();

            return;

        }

        struct node \*temp = head;

        int count = 1;

        while (count < pos - 1 && temp->next != NULL)

        {

            temp = temp->next;

            count++;

        }

        if (temp->next == nullptr)

        {

            cout << "Position out of bounds!" << endl;

            return;

        }

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

    }

    void search(int value)

    {

        if (head == NULL)

        {

            cout << "The lsit is empty\n";

            return;

        }

        struct node \*temp = head;

        do

        {

            if (temp->data == value)

            {

                cout << "The value is found: " << value << "\n";

                return;

            }

            temp = temp->next;

        } while (temp->next != NULL);

        cout << "The number is not found\n";

    }

    void display\_reverse()

    {

        int arr[50];

        struct node \*temp = head;

        int count = 0;

        if (head == NULL)

        {

            cout << "The list is empty\n";

            return;

        }

        while (temp != NULL)

        {

            arr[count] = temp->data;

            temp = temp->next;

            count++;

        }

        for (int i = count - 1; i >= 0; i--)

        {

            cout << arr[i] << " ";

        }

        cout << "\n";

    }

    void reverse\_link()

    {

        node \*curr = head, \*prev = nullptr, \*next;

        // Traverse all the nodes of Linked List

        while (curr != nullptr)

        {

            // Store next

            next = curr->next;

            // Reverse current node's next pointer

            curr->next = prev;

            // Move pointers one position ahead

            prev = curr;

            curr = next;

        }

        head = prev;

    }

};

int main()

{

    int choice, value, pos, exit = 0;

    class list l;

    while (exit != 1)

    {

        cout << "\n1. Insert Beginning  2. Insert end  3. Insert Position\n4. Delete Beginning  5. Delete End  6. Delete position\n7. Search            8. Display     9.Display Reverse\n10. Reverse Link\n";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value to be inserted\n";

            cin >> value;

            l.insert\_beginning(value);

            break;

        case 2:

            cout << "Enter the value to be inserted\n";

            cin >> value;

            l.insert\_end(value);

            break;

        case 3:

            cout << "Enter the value\n";

            cin >> value;

            cout << "Enter the position\n";

            cin >> pos;

            l.insert\_position(value, pos);

            break;

        case 4:

            l.delete\_beginning();

            break;

        case 5:

            l.delete\_end();

            break;

        case 6:

            cout << "Enter the position\n";

            cin >> pos;

            l.delete\_position(pos);

            break;

        case 7:

            cout << "Enter the value\n";

            cin >> value;

            l.search(value);

            break;

        case 8:

            l.display();

            break;

        case 9:

            l.display\_reverse();

            break;

        case 10:

            l.reverse\_link();

            break;

        case 11:

            exit = 1;

            break;

        }

    }

}

## Output:

## 2. B. Write a C++ menu-driven program to implement List ADT using a singly linked list. You have a gethead() private member function that returns the address of the head value of a list. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

1. Insert Ascending
2. Merge
3. Display
4. Exit

## Aim:

To write a C++ menu-driven program to implement List ADT using a singly linked list with operations for insertion in ascending order, merging two lists, and displaying the contents. The program uses a getHead() private member function to return the head address. The implementation is modular, with logic placed in a header file and included in a separate C++ file which handles three linked lists (List1, List2, List3).

## Algorithms:

**Algorithm 1 – Insert in Ascending Order**

**Input:**  
value – data to be inserted  
head – pointer to the head of the list

**Output:**  
Linked list with value inserted in ascending position

**Time Complexity:** O(n)

Create new\_node with data = value

if head == NULL or value < head.data

new\_node.next ← head

head ← new\_node

return

current ← head

while current.next ≠ NULL and current.next.data < value

current ← current.next

new\_node.next ← current.next

current.next ← new\_node

**Algorithm 2 – Merge Two Sorted Lists into Third List**

**Input:**  
head1 – pointer to first sorted linked list  
head2 – pointer to second sorted linked list

**Output:**  
head3 – pointer to merged sorted linked list

**Time Complexity:** O(n + m)

Create dummy node with next = NULL

tail ← dummy

while head1 ≠ NULL and head2 ≠ NULL

if head1.data < head2.data

tail.next ← head1

head1 ← head1.next

else

tail.next ← head2

head2 ← head2.next

tail ← tail.next

if head1 ≠ NULL

tail.next ← head1

if head2 ≠ NULL

tail.next ← head2

head3 ← dummy.next

### ****Algorithm 3 – Display a List****

**Input:**  
head – pointer to the head of the list

**Output:**  
Prints the list contents

if head == NULL

print "List is empty"

return

current ← head

while current ≠ NULL

print current.data

current ← current.next

## Code:

#include<iostream>

#include "header.h"

using namespace std;

int main()

{

    list l1,l2,l3;

    int choice,num,exit=0;

    while(exit!=1)

    {

        cout << "1.Insert in list 1\n2.Insert in list 2\n3.Merge lists\n4.Display list 1\n5.Display list 2\n6.Display merged list\n7.Exit\n";

        cin >> choice;

        switch(choice)

        {

            case 1:

                cout << "Enter number to insert in list 1: ";

                cin >> num;

                l1.insert\_ascending(num);

                break;

            case 2:

                cout << "Enter number to insert in list 2: ";

                cin >> num;

                l2.insert\_ascending(num);

                break;

            case 3:

                l3=l3.merge(l1,l2);

                break;

            case 4:

                l1.display();

                break;

            case 5:

                l2.display();

                break;

            case 6:

                l3.display();

                break;

            case 7:

                exit=1;

                break;

            default:

                cout << "Invalid choice\n";

        }

    }

}

## Output:



**Lab - 5**

1. Write a C++ menu-driven program to implement List ADT using a doubly linked list with a tail. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

1. Insert Beginning
2. Insert End
3. Insert Position
4. Delete Beginning
5. Delete End
6. Delete Position
7. Search
8. Display
9. Exit

What is the time complexity of each of the operations? (K4)

## Aim:

To write a C++ menu-driven program to implement List ADT using a doubly linked list with tail support. The program includes insertions, deletions, search, and display operations, with appropriate handling of boundary conditions and good coding practices.

## Algorithms:

**Algorithm 1 – Insert at Beginning**

**Input:**  
value, head, tail

**Output:**  
Node inserted at the beginning

Create new\_node with data = value

new\_node.prev ← NULL

new\_node.next ← head

if head ≠ NULL

head.prev ← new\_node

else

tail ← new\_node

head ← new\_node

**Algorithm 2 – Insert at End**

**Input:**  
value, tail, head

**Output:**  
Node inserted at the end

Create new\_node with data = value

new\_node.next ← NULL

new\_node.prev ← tail

if tail ≠ NULL

tail.next ← new\_node

else

head ← new\_node

tail ← new\_node

**Algorithm 3 – Insert at Position**

**Input:**  
value, position, head

**Output:**  
Node inserted at given position

if position == 0

Insert at beginning

return

current ← head

for i from 0 to position - 2

if current == NULL

print "Invalid position"

return

current ← current.next

Create new\_node with data = value

new\_node.next ← current.next

new\_node.prev ← current

if current.next ≠ NULL

current.next.prev ← new\_node

else

tail ← new\_node

current.next ← new\_node

**Algorithm 4 – Delete from Beginning**

**Input:**  
head, tail

if head == NULL

print "List is empty"

return

temp ← head

head ← head.next

if head ≠ NULL

head.prev ← NULL

else

tail ← NULL

Delete temp

**Algorithm 5 – Delete from End**

**Input:**  
tail, head

if tail == NULL

print "List is empty"

return

temp ← tail

tail ← tail.prev

if tail ≠ NULL

tail.next ← NULL

else

head ← NULL

Delete temp

**Algorithm 6 – Delete at Position**

**Input:**  
position, head

if head == NULL

print "List is empty"

return

if position == 0

Delete from beginning

return

current ← head

for i from 0 to position - 1

if current == NULL

print "Invalid position"

return

current ← current.next

if current == tail

Delete from end

return

current.prev.next ← current.next

current.next.prev ← current.prev

Delete current

**Algorithm 7 – Search for an Element**

**Input:**  
key, head

position ← 0

current ← head

while current ≠ NULL

if current.data == key

return position

current ← current.next

position ← position + 1

return -1 // Not found

**Algorithm 8 – Display**

**Input:**  
head

current ← head

while current ≠ NULL

print current.data

current ← current.next

## Code:

#include <iostream>

using namespace std;

class list

{

    struct node

    {

        struct node \*prev;

        int data;

        struct node \*next;

    } \*head, \*tail;

public:

    list()

    {

        head = NULL;

        tail = NULL;

    }

    struct node \*create\_new\_node(int value)

    {

        struct node \*newnode = new struct node;

        newnode->data = value;

        return newnode;

    }

    void display() // Time complexity: O(n)

    {

        struct node \*temp = head;

        if (temp == NULL)

        {

            cout << "The list empty" << endl;

            return;

        }

        while (temp != NULL)

        {

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

            temp = temp->next;

        }

        cout << "NULL" << endl;

    }

    void insert\_beginning(int value) // Time complexity: O(1)

    {

        struct node \*newnode = create\_new\_node(value);

        newnode->next = head;

        newnode->prev = NULL;

        if (head == NULL)

        {

            head = newnode;

            tail = newnode;

        }

        else

        {

            head = newnode;

        }

    }

    void insert\_end(int value) // Time complexity: O(1)

    {

        struct node \*newnode = create\_new\_node(value);

        if (head == NULL)

        {

            insert\_beginning(value);

            return;

        }

        tail->next = newnode;

        newnode->next = NULL;

        newnode->prev = tail;

        tail = newnode;

    }

    void insert\_position(int value, int pos) // Time complexity: O(n)

    {

        int i;

        struct node \*newnode = create\_new\_node(value);

        struct node \*temp = head;

        if (pos < 1)

        {

            cout << "Invalid" << endl;

            return;

        }

        if (pos == 1)

        {

            insert\_beginning(value);

            return;

        }

        for (i = 1; i < pos - 1 && temp != NULL; i++)

        {

            temp = temp->next;

        }

        if (temp == NULL)

        {

            cout << "Out of bound" << endl;

            return;

        }

        if (temp->next == NULL)

        {

            insert\_end(value);

            return;

        }

        newnode->next = temp->next;

        temp->next->prev = newnode;

        temp->next = newnode;

        newnode->prev = temp;

    }

    void delete\_beginning() // Time complexity: O(1)

    {

        if (head == NULL)

        {

            cout << "The list is empty" << endl;

            return;

        }

        struct node \*temp = head;

        head = head->next;

        delete temp;

    }

    void delete\_end() // Time complexity: O(1)

    {

        if (head == NULL)

        {

            cout << "The list is empty" << endl;

            return;

        }

        if (head->next == NULL)

        {

            head = NULL;

            tail = NULL;

            return;

        }

        struct node \*temp = tail;

        tail = tail->prev;

        tail->next = NULL;

        delete temp;

    }

    void delete\_position(int pos) // Time complexity: O(n)

    {

        struct node \*temp = head;

        int i;

        if (head == NULL)

        {

            cout << "The list is empty" << endl;

            return;

        }

        if (pos < 1)

        {

            cout << "Invalid" << endl;

            return;

        }

        if (pos == 1)

        {

            delete\_beginning();

            return;

        }

        for (i = 1; i < pos && temp != NULL; i++)

        {

            temp = temp->next;

        }

        if (temp == NULL)

        {

            cout << "Out of bound" << endl;

            return;

        }

        if (temp->next == NULL)

        {

            delete\_end();

            return;

        }

        temp->prev->next = temp->next;

        temp->next->prev = temp->prev;

        delete temp;

    }

    void search(int value) // Time complexity: O(n)

    {

        struct node \*temp = head;

        if (head == NULL)

        {

            cout << "The list empty" << endl;

            return;

        }

        while (temp != NULL)

        {

            if (temp->data == value)

            {

                cout << "The value is found" << endl;

                break;

            }

            temp = temp->next;

        }

    }

};

int main()

{

    int value, choice, pos, exit = 1;

    list l;

    while (exit == 1)

    {

        cout << "\n1. Insert Beginning  2. Insert end  3. Insert Position\n4. Delete Beginning  5. Delete End  6. Delete position\n7. Search            8. Display     9. Exit\n";

            cout

             << "Enter the choice" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value" << endl;

            cin >> value;

            l.insert\_beginning(value);

            break;

        case 2:

            cout << "Enter the value" << endl;

            cin >> value;

            l.insert\_end(value);

            break;

        case 3:

            cout << "Enter the Value" << endl;

            cin >> value;

            cout << "Enter the position" << endl;

            cin >> pos;

            l.insert\_position(value, pos);

            break;

        case 4:

            l.delete\_beginning();

            break;

        case 5:

            l.delete\_end();

            break;

        case 6:

            cout << "Enter the position" << endl;

            cin >> pos;

            l.delete\_position(pos);

            break;

        case 7:

            cout << "Enter the Value" << endl;

            cin >> value;

            l.search(value);

            break;

        case 8:

            l.display();

            break;

        case 9:

            exit = 0;

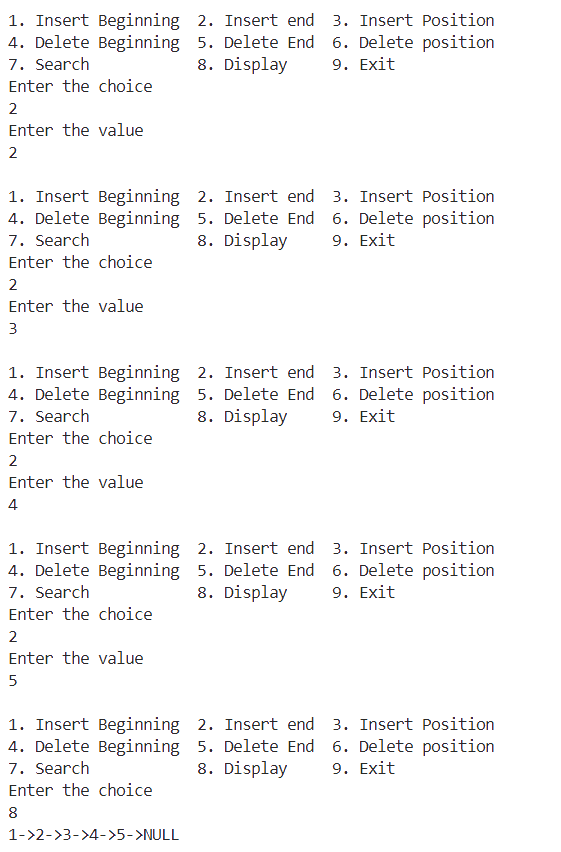
            break;

        }

    }

}

## Output:



2. B. Write a C++ menu-driven program to implement List ADT using a circular linked list.  
Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

1. Insert Beginning
2. Insert End
3. Insert Position
4. Delete Beginning
5. Delete End
6. Delete Position
7. Search
8. Display
9. Exit

## Aim:

To write a C++ menu-driven program to implement the List ADT using a circular linked list. The program includes insertions, deletions, search, and display operations with boundary condition checks and good coding practices.

## Algorithms:

**Algorithm 1 – Insert at Beginning**

**Input:**  
value, head

**Output:**  
Circular list with node inserted at beginning

Create new\_node with data = value

if head == NULL

new\_node.next ← new\_node

head ← new\_node

return

temp ← head

while temp.next ≠ head

temp ← temp.next

new\_node.next ← head

temp.next ← new\_node

head ← new\_node

**Algorithm 2 – Insert at End**

**Input:**  
value, head

**Output:**  
Circular list with node inserted at end

Create new\_node with data = value

if head == NULL

new\_node.next ← new\_node

head ← new\_node

return

temp ← head

while temp.next ≠ head

temp ← temp.next

temp.next ← new\_node

new\_node.next ← head

### ****Algorithm 3 – Insert at Position****

**Input:**  
value, position, head

**Output:**  
Node inserted at given position

if position == 0

Insert at beginning

return

current ← head

for i from 0 to position - 2

if current.next == head

print "Invalid position"

return

current ← current.next

Create new\_node with data = value

new\_node.next ← current.next

current.next ← new\_node

**Algorithm 4 – Delete from Beginning**

**Input:**  
head

if head == NULL

print "List is empty"

return

if head.next == head

Delete head

head ← NULL

return

temp ← head

last ← head

while last.next ≠ head

last ← last.next

head ← head.next

last.next ← head

Delete temp

**Algorithm 5 – Delete from End**

if head == NULL

print "List is empty"

return

if head.next == head

Delete head

head ← NULL

return

prev ← NULL

current ← head

while current.next ≠ head

prev ← current

current ← current.next

prev.next ← head

Delete current

**Algorithm 6 – Delete at Position**

**Input:**  
position, head

if head == NULL

print "List is empty"

return

if position == 0

Delete from beginning

return

prev ← head

for i from 0 to position - 2

if prev.next == head

print "Invalid position"

return

prev ← prev.next

to\_delete ← prev.next

prev.next ← to\_delete.next

Delete to\_delete

**Algorithm 7 – Search for an Element**

**Input:**  
key, head

if head == NULL

return -1

position ← 0

current ← head

do

if current.data == key

return position

current ← current.next

position ← position + 1

while current ≠ head

return -1 // Not found

**Algorithm 8 – Display**

**Input:**  
head

if head == NULL

print "List is empty"

return

current ← head

do

print current.data

current ← current.next

while current ≠ head

## Code:

#include <iostream>

using namespace std;

class list

{

    struct node

    {

        node \*prev;

        int data;

        node \*next;

    } \*head;

public:

    list()

    {

        head = nullptr;

    }

    struct node \*create\_new\_node(int value)

    {

        node \*newnode = new struct node;

        newnode->data = value;

        return newnode;

    }

    void display()

    {

        node \*temp = head;

        if (head == nullptr)

        {

            cout << "The list is empty" << endl;

            return;

        }

        cout << temp->data << " ";

        temp = temp->next;

        while (temp != head)

        {

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

            temp = temp->next;

        }

    }

    void insert\_beginning(int value)

    {

        node \*newnode = create\_new\_node(value);

        node \*temp = head;

        if (head == nullptr)

        {

            head = newnode;

            newnode->next = head;

            return;

        }

        while (temp->next != head)

        {

            temp = temp->next;

        }

        newnode->next = head;

        head = newnode;

        temp->next = head;

    }

    void insert\_end(int value)

    {

        node \*newnode = create\_new\_node(value);

        if (head == nullptr)

        {

            head = newnode;

            newnode->next = head;

            return;

        }

        node \*temp = head;

        newnode->next = head;

        while (temp->next != head)

        {

            temp = temp->next;

        }

        temp->next = newnode;

    }

    void insert\_position(int value, int pos)

    {

        if (pos < 1)

        {

            cout << "Invalid" << endl;

            return;

        }

        if (pos == 1)

        {

            insert\_beginning(value);

            return;

        }

        node \*newnode = create\_new\_node(value);

        node \*temp = head;

        int count = 1;

        while (count < pos - 1 && temp != head)

        {

            temp = temp->next;

            count++;

        }

        if (temp == head)

        {

            cout << "Out Of Bound" << endl;

            return;

        }

        if (temp->next == head)

        {

            insert\_end(value);

            return;

        }

        newnode->next = temp->next;

        temp->next = newnode;

    }

    void delete\_beginning()

    {

        if (head == nullptr)

        {

            cout << "The list is empty" << endl;

            return;

        }

        if (head->next == head)

        {

            head = nullptr;

            return;

        }

        node \*temp = head;

        node \*temp1 = head;

        while (temp->next != head)

        {

            temp = temp->next;

        }

        head = head->next;

        delete temp1;

        temp->next = head;

    }

    void delete\_end()

    {

        if (head == nullptr)

        {

            cout << "The list is empty" << endl;

            return;

        }

        if (head->next == head)

        {

            delete head;

            head = nullptr;

            return;

        }

        node \*temp = head;

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

        {

            temp = temp->next;

        }

        node \*todelete = temp->next;

        temp->next = head;

        delete todelete;

    }

    void delete\_position(int pos)

    {

        if (pos < 1)

        {

            cout << "Invalid" << endl;

            return;

        }

        if (head == nullptr)

        {

            cout << "The list is empty" << endl;

            return;

        }

        if (pos == 1)

        {

            delete\_beginning();

            return;

        }

        node \*temp = head;

        int count = 1;

        while (count < pos - 1 && temp != head)

        {

            temp = temp->next;

            count++;

        }

        if (temp == head)

        {

            cout << "Out of Bound" << endl;

            return;

        }

        node \*temp1;

        temp1 = temp->next;

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

        delete temp1;

    }

    void search(int value)

    {

        if (head == nullptr)

        {

            cout << "The lsit is empty" << endl;

            return;

        }

        struct node \*temp = head;

        do

        {

            if (temp->data == value)

            {

                cout << "The value is found: " << value << "\n";

                return;

            }

            temp = temp->next;

        } while (temp->next != NULL);

        cout << "The number is not found\n";

    }

};

int main()

{

    int value, choice, pos, exit = 1;

    list l;

    while (exit == 1)

    {

        cout << "\n1. Insert Beginning  2. Insert end  3. Insert Position\n4. Delete Beginning  5. Delete End  6. Delete position\n7. Search            8. Display     9. Exit\n";

        cout

            << "Enter the choice" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value" << endl;

            cin >> value;

            l.insert\_beginning(value);

            break;

        case 2:

            cout << "Enter the value" << endl;

            cin >> value;

            l.insert\_end(value);

            break;

        case 3:

            cout << "Enter the Value" << endl;

            cin >> value;

            cout << "Enter the position" << endl;

            cin >> pos;

            l.insert\_position(value, pos);

            break;

        case 4:

            l.delete\_beginning();

            break;

        case 5:

            l.delete\_end();

            break;

        case 6:

            cout << "Enter the position" << endl;

            cin >> pos;

            l.delete\_position(pos);

            break;

        // case 7:

        //     cout << "Enter the Value" << endl;

        //     cin >> value;

        //     l.search(value);

        //     break;

        case 8:

            l.display();

            break;

        case 9:

            exit = 0;

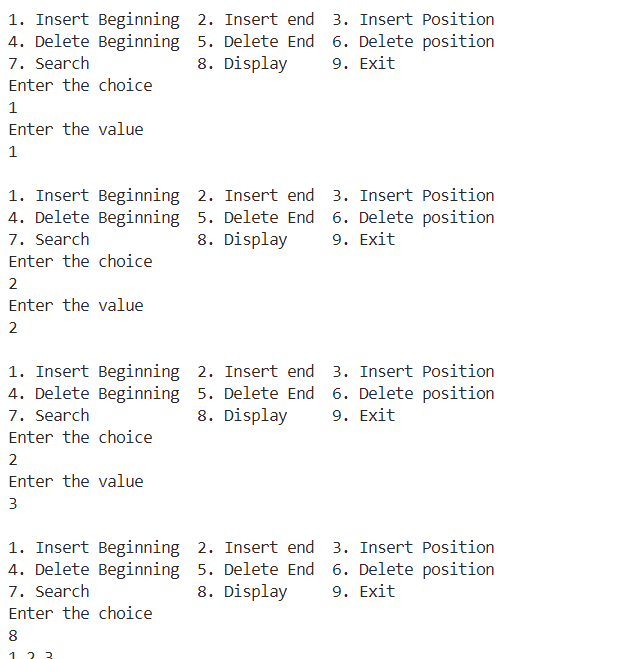
            break;

        }

    }

}

## Output:



3. An operating system allocates a fixed time slot CPU time for processes using a round-robin scheduling algorithm. The fixed time slot will be initialized before the start of the menu-driven program. Implement the round-robin scheduling algorithm using the circular linked list.

Implement the program by including the appropriate header file. It consists of the following operations.

1. Insert Process
2. Execute
3. Exit

## Aim:

To write a C++ menu-driven program to implement the round-robin CPU scheduling algorithm using a circular linked list. The program uses a fixed time slot and performs insertions and executions based on CPU burst times.

## Algorithms:

**Algorithm 1 – Insert Process**

**Input:**  
time\_required – burst time of a process

Create new\_node with data = time\_required

if head == NULL

new\_node.next ← new\_node

head ← new\_node

tail ← new\_node

else

new\_node.next ← head

tail.next ← new\_node

tail ← new\_node

**Algorithm 2 – Execute**

**Input:**  
head, tail, time\_slot

if head == NULL

print "Queue is empty"

return

temp ← head

head ← head.next

tail.next ← head

execution\_time ← temp.data - time\_slot

if execution\_time > 0

Create new\_node with data = execution\_time

new\_node.next ← head

tail.next ← new\_node

tail ← new\_node

print "Process executed for", time\_slot

else

print "Process completed"

Delete temp

## Code:

// program to perform round-robin scheduling

# include <cstdio>

# include <cstdlib>

# include "circular\_list.h"

int main()

{

    sll l1;

    int choice, time, res;

    while (1)

    {

        printf("\n1. Insert process\n2. Execute\n3. Exit\n");

        printf("Enter menu number of operation you want to perform: ");

        scanf("%d", &choice);

        while (choice < 1 || choice > 3)

        {

            printf("Error. Enter a valid menu number: ");

            scanf("%d", &choice);

        }

        switch (choice)

        {

        case 1:

        {

            printf("Enter integer time: ");

            scanf("%d", &time);

            while (time <= 0)

            {

                printf("Error. Enter a valid process time: ");

                scanf("%d", &time);

            }

            res = l1.insert\_end(time);

            if (res == 1)

            {

                printf("%d has been inserted.\n", time);

            }

            break;

        }

        case 2:

        {

            int temp;

            printf("Enter fixed processing time: ");

            scanf("%d", &time);

            while (time <= 0)

            {

                printf("Error. Enter a valid process time: ");

                scanf("%d", &time);

            }

            res = l1.execute(time);

            if (res == -1)

            {

                printf("List is empty.\n");

            }

            break;

        }

        case 3:

        {

            exit(0);

            break;

        }

        default:

        {

            printf("Error.\n");

            break;

        }

        }

    }

}

**Lab - 6**

## Aim:

To write a C++ menu-driven program to implement Stack ADT using a character array of size 5.

## Algorithms:

**Algorithm – Push**

**Input:**  
element – a character to be pushed  
top – index of the top of the stack  
stack[] – array of size 5

**Output:**  
Element is pushed into the stack

if top == 4

print "Stack Overflow"

else

top ← top + 1

stack[top] ← element

**Algorithm – Pop**

**Input:**  
top – index of the top of the stack  
stack[] – array of characters

**Output:**  
Top element is removed

if top == -1

print "Stack Underflow"

else

element ← stack[top]

top ← top - 1

print element

**Algorithm – Peek**

**Input:**  
top, stack[]

**Output:**  
Top element is displayed

if top == -1

print "Stack is empty"

else

print stack[top]

## Code:

#include <iostream>

using namespace std;

class stack

{

    char arr[5];

    int cur;

public:

    stack()

    {

        cur = 0;

    }

    void display()

    {

        cout << endl;

        if (cur == 0)

        {

            cout << "The stack is empty" << endl;

            return;

        }

        for (int i = 0; i < cur; i++)

        {

            cout << arr[i] << " ";

        }

        cout << endl;

    }

    void push(int value)

    {

        if (cur == 5)

        {

            cout << "Stack overflow" << endl;

            return;

        }

        for (int i = cur; i > 0; i--)

        {

            arr[i] = arr[i - 1];

        }

        arr[0] = value;

        cur++;

    }

    char pop()

    {

        if (cur == 0)

        {

            cout << "Stack underflow" << endl;

            return 'N';

        }

        else

        {

            char temp = arr[0];

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

            {

                arr[i] = arr[i + 1];

            }

            cur--;

            return temp;

        }

    }

    char peek()

    {

        if (cur == 0)

        {

            cout << "Stack underflow" << endl;

            return 'N';

        }

        else

        {

            return arr[0];

        }

    }

};

int main()

{

    int choice, exit = 0;

    char value,temp;

    stack s;

    while (exit == 0)

    {

        cout << "1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value" << endl;

            cin >> value;

            s.push(value);

            break;

        case 2:

            temp = s.pop();

            cout << "Deleted value: " << temp << endl;

            break;

        case 3:

            temp = s.peek();

            cout << "The top value: " << temp << endl;

            break;

        case 4:

            s.display();

            break;

        case 5:

            exit = 1;

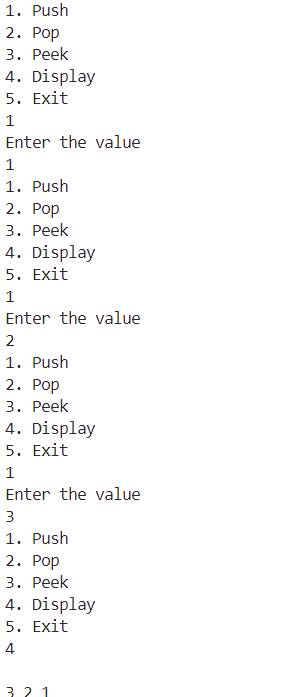
            break;

        }

    }

}

## Output:



2. Write a separate C++ menu-driven program to implement stack ADT using a character singly linked list. Maintain proper boundary conditions and follow good coding practices. Stack ADT has the following operations,

1. Push
2. Pop
3. Peek
4. Exit

## Aim:

To write a C++ menu-driven program to implement Stack ADT using a character singly linked list.

## Algorithms:

**Algorithm – Push**

**Input:**  
element – a character to be pushed  
head – pointer to the top of the stack

**Output:**  
Element is inserted at the beginning (top of the stack)

Create new node

new\_node → data ← element

new\_node → next ← head

head ← new\_node

**Algorithm – Pop**

**Input:**  
head – pointer to the top of the stack

**Output:**  
Top element is deleted and displayed

if head is NULL

print "Stack Underflow"

else

temp ← head

head ← head → next

print temp → data

delete temp

**Algorithm – Peek**

**Input:**  
head – pointer to the top of the stack

**Output:**  
Top element is displayed

if head is NULL

print "Stack is empty"

else

print head → data

## Code:

#include <iostream>

using namespace std;

class stack

{

    struct node

    {

        char data;

        struct node \*next;

    } \*head;

public:

    stack()

    {

        head = nullptr;

    }

    node \*create\_new\_node(char value)

    {

        node \*newnode = new struct node;

        newnode->data = value;

        return newnode;

    }

    void display()

    {

        if (head == nullptr)

        {

            cout << "The stack is empty" << endl;

            return;

        }

        node \*temp = head;

        while (temp != nullptr)

        {

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

            temp = temp->next;

        }

        cout << endl;

    }

    void push(char value)

    {

        node \*newnode = create\_new\_node(value);

        newnode->next = head;

        head = newnode;

    }

    char pop()

    {

        if (head == nullptr)

        {

            cout << "Stack underflow" << endl;

            return 'N';

        }

        node \*temp = head;

        char value = head->data;

        head = head->next;

        delete temp;

        return value;

    }

    char peek()

    {

        if (head == nullptr)

        {

            cout << "Stack underflow" << endl;

            return 'N';

        }

        return head->data;

    }

};

int main()

{

    int choice, exit = 0;

    char value,temp;

    stack s;

    while (exit == 0)

    {

        cout << "1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value" << endl;

            cin >> value;

            s.push(value);

            break;

        case 2:

            temp = s.pop();

            cout << "Deleted value: " << temp << endl;

            break;

        case 3:

            temp = s.peek();

            cout << "The top value: " << temp << endl;

            break;

        case 4:

            s.display();

            break;

        case 5:

            exit = 1;

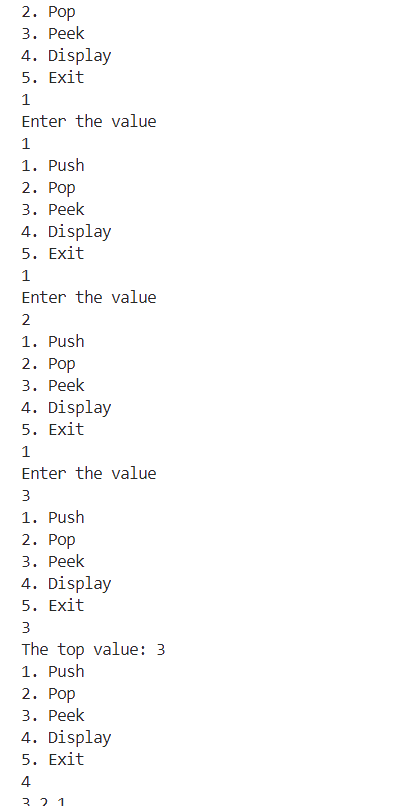
            break;

        }

    }

}

## Output:



3.Write a C++ menu-driven program to implement infix to postfix and postfix evaluation. Use the singly linked list (SLL) to implement the stack ADT as a header file. Maintain proper boundary conditions and follow good coding practices. The program has the following operations, Get Infix Convert Infix Evaluate Postfix Exit The Get Infix option gets a valid infix expression and stores it efficiently. The Convert Infix option converts the stored infix expression into a postfix expression. It prints the postfix expression at the end after conversion. The Evaluate Postfix expression calculates and prints the output of the converted infix expression.

## Aim:

To write a C++ menu-driven program to implement infix to postfix conversion and postfix evaluation using stack ADT implemented with a singly linked list (SLL).

## Algorithms:

**Algorithm 1 – Get Infix**

**Input:**  
A valid infix expression (e.g., "3+(4\*5)")

**Output:**  
Stores the infix expression

Prompt user for infix expression

Read the expression as a string

Store it in a character array or string variable

**Algorithm 2 – Infix to Postfix Conversion**

**Input:**  
A valid infix expression  
Operator precedence: ^ > \* / % > + -

**Output:**  
Converted postfix expression

Initialize empty postfix expression and stack

For each character 'ch' in infix expression:

if ch is operand:

Append ch to postfix

else if ch is '(':

Push ch onto stack

else if ch is ')':

While top of stack ≠ '(':

Pop from stack and append to postfix

Pop '(' from stack

else if ch is operator:

While stack is not empty and precedence of ch ≤ precedence of top of stack:

Pop from stack and append to postfix

Push ch onto stack

While stack is not empty:

Pop from stack and append to postfix

Return postfix expression

**Algorithm 3 – Evaluate Postfix**

**Input:**  
A valid postfix expression (e.g., "345\*+")

**Output:**  
Evaluated result of the postfix expression

Initialize empty stack

For each character 'ch' in postfix expression:

if ch is operand:

Push ch (converted to integer) onto stack

else if ch is operator:

op2 ← Pop from stack

op1 ← Pop from stack

result ← op1 <operator> op2

Push result onto stack

Final result is on top of the stack

Return stack top as output

## Code:

#include <iostream>

#include "stck\_sll.h"

#include <string>

#include <cmath>

using namespace std;

int precedence(char op)

{

    if (op == '^')

        return 3; // Highest precedence

    if (op == '\*' || op == '/')

        return 2; // Multiplication/Division

    if (op == '+' || op == '-')

        return 1; // Addition/Subtraction

    return 0;

}

string get\_infix()

{

    string op;

    cout << "Enter the string" << endl;

    cin >> op;

    return op;

}

string convert\_infix(string op)

{

    string exp;

    stack s;

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

    {

        if (isdigit(op[i]))

        {

            exp = exp + op[i];

        }

        else

        {

            int pre = precedence(op[i]);

            if (s.peek() == '^' && op[i] == '^')

            {

                s.push(op[i]);

            }

            else

            {

                while (precedence(s.peek()) >= pre && s.peek() != 'N')

                {

                    exp += s.pop();

                }

                s.push(op[i]);

            }

        }

    }

    while (s.peek() != 'N')

    {

        exp += s.pop();

    }

    return exp;

}

int eval\_postfix(string conv)

{

    stack s;

    string exp;

    int value, a, b;

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

    {

        if (isdigit(conv[i]))

        {

            s.push(conv[i]);

        }

        else

        {

            b = s.pop();

            a = s.pop();

            if (conv[i] == '+')

            {

                s.push((a - '0') + (b - '0'));

            }

            else if (conv[i] == '-')

            {

                s.push((a - '0') - (b - '0'));

            }

            else if (conv[i] == '\*')

            {

                s.push((a - '0') \* (b - '0'));

            }

            else if (conv[i] == '/')

            {

                s.push((a - '0') / (b - '0'));

            }

            else if (conv[i] == '^')

            {

                s.push(pow((a - '0'), (b - '0')));

            }

        }

    }

    return s.peek();

}

int main()

{

    int exit = 0, choice;

    string op, conv;

    int val;

    while (exit == 0)

    {

        cout << "1. Get Infix\n2. Convert Infix\n3. Evaluate Postfix\n4. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            op = get\_infix();

            cout << op << endl;

            break;

        case 2:

            conv = convert\_infix(op);

            cout << conv << endl;

            break;

        case 3:

            val = eval\_postfix(conv);

            cout << val << endl;

            break;

        case 4:

            exit = 1;

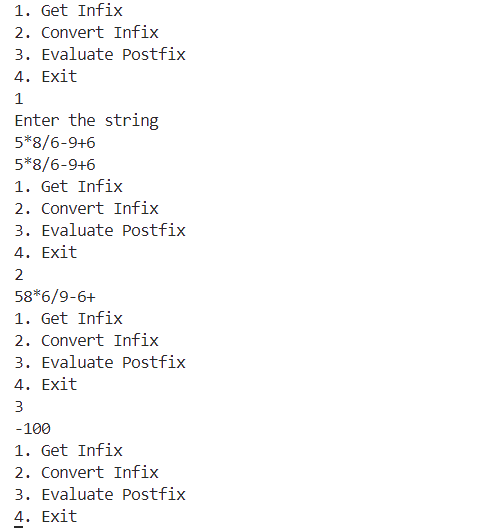
            break;

        }

    }

}

## Output:



4.Write a C++ menu-driven program to check balanced parentheses using optimal ADT. The program has:

1. Check Balance
2. Exit

Requirements:

* Input: String of '(' and ')'
* Output: Balance status
* Use optimal data structure (stack recommended)
* Leverage previous header files
* Implement boundary conditions
* Follow coding best practices

## Aim:

To write a C++ menu-driven program to check for balanced parentheses using an optimal data structure (stack) implemented via a header file.

## Algorithms:

**Algorithm – Check Balanced Parentheses**

**Input:**  
A string containing only '(' and ')'

**Output:**  
Whether the parentheses are balanced or not

Initialize empty stack

For each character ch in the input string:

if ch == '(':

Push onto stack

else if ch == ')':

if stack is empty:

Return "Unbalanced"

else:

Pop from stack

After traversing the string:

if stack is empty:

Return "Balanced"

else:

Return "Unbalanced"

## Code:

#include <iostream>

#include "stck\_sll.h"

#include <string>

using namespace std;

void check\_balance(string str)

{

    stack s;

    char temp;

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

    {

        if (str[i] == '(')

        {

            s.push(str[i]);

        }

        else

        {

            if (s.peek() == 'N')

            {

                cout << "It is not Balanced" << endl;

                return;

            }

            temp = s.pop();

        }

    }

    if (s.peek() == 'N')

    {

        cout << "It is Balanced" << endl;

    }

    else

    {

        cout << "It is not Balanced" << endl;

    }

}

int main()

{

    string str;

    int exit = 0, choice;

    while (exit == 0)

    {

        cout << "1. Check Balance\n2. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the string" << endl;

            cin >> str;

            check\_balance(str);

            break;

        case 2:

            exit = 1;

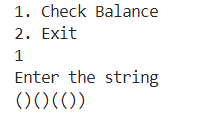
            break;

        }

    }

}

## Output:



**Lab - 7**

1.Write a separate C++ menu-driven program to implement Queue ADT using an integer array of size 5. Maintain proper boundary conditions and follow good coding practices. The Queue ADT has the following operations,

1. Enqueue
2. Dequeue
3. Peek
4. Exit

## Aim:

To write a C++ menu-driven program to implement Queue ADT using an integer array of size 5 with proper boundary conditions and coding practices.

## Algorithms:

**Algorithm 1 – Enqueue**

**Input:**  
An integer value to insert

**Output:**  
Queue with new element at the rear

If (rear == size - 1):

Return "Queue Overflow"

Else:

rear ← rear + 1

queue[rear] ← value

If (front == -1):

front ← 0

**Algorithm 2 – Dequeue**

**Input:**  
Queue with elements

**Output:**  
Queue with front element removed

If (front == -1 or front > rear):

Return "Queue Underflow"

Else:

value ← queue[front]

front ← front + 1

**Algorithm 3 – Peek**

**Input:**  
Queue

**Output:**  
Element at front of queue

If (front == -1 or front > rear):

Return "Queue is empty"

Else:

Return queue[front]

## Code:

#include <iostream>

using namespace std;

#define size 5

class queue

{

    int arr[5];

    int cur;

public:

    queue()

    {

        cur = 0;

    }

    void enqueue(int);

    void dequeue();

    int peek();

    void display();

};

void queue::enqueue(int value)

{

    if (cur == size)

    {

        cout << "The queue is full\n";

    }

    arr[cur] = value; // This can alos be written as arr[cur++] = value

    cur++;

    display();

}

void queue::dequeue()

{

    if (cur == 0)

    {

        cout << "The queue emptyt\n";

    }

    for (int i = 0; i < (cur - 1); i++)

    {

        arr[i] = arr[i + 1];

    }

    cur--;

    display();

}

int queue::peek()

{

    if (cur == 0)

    {

        cout << "The Queue is empty" << endl;

        return 404;

    }

    return arr[cur - 1];

}

void queue::display()

{

    for(int i = 0; i<5; i++)

    {

        cout << arr[i];

    }

    cout << endl;

}

int main()

{

    queue ca;

    int exit = 1;

    while (exit == 1)

    {

        int choice, val;

        cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value: ";

            cin >> val;

            ca.enqueue(val);

            break;

        case 2:

            ca.dequeue();

            break;

        case 3:

            val = ca.peek();

            if (val != 404)

            {

                cout << "Front element: " << val << endl;

            }

            break;

        case 4:

            exit = 0;

            break;

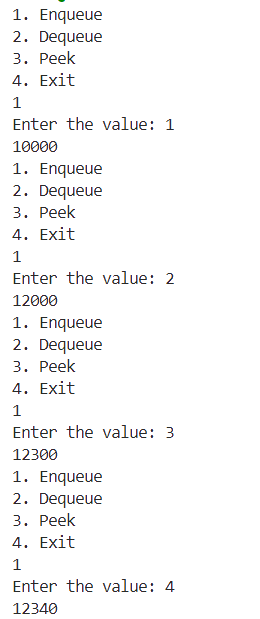
        }

    }

    return 0;

}

## Output:



2.Write a separate C++ menu-driven program to implement Circular Queue ADT using an integer array of size 5. Maintain proper boundary conditions and follow good coding practices. The Circular Queue ADT has the following operations,

1. Enqueue
2. Dequeue
3. Peek
4. Exit

## Aim:

To write a C++ menu-driven program to implement Circular Queue ADT using an integer array of size 5, maintaining proper boundary conditions and good coding practices.

## Algorithms:

**Algorithm 1 – Enqueue (Circular Queue)**

**Input:**  
An integer value to insert

**Output:**  
Queue with new element at the rear

If ((rear + 1) % size == front):

Return "Queue Overflow"

Else if (front == -1 and rear == -1):

front ← 0

rear ← 0

Else:

rear ← (rear + 1) % size

queue[rear] ← value

**Algorithm 2 – Dequeue (Circular Queue)**

**Input:**  
Queue with elements

**Output:**  
Queue with front element removed

If (front == -1):

Return "Queue Underflow"

value ← queue[front]

If (front == rear):

front ← -1

rear ← -1

Else:

front ← (front + 1) % size

**Algorithm 3 – Peek (Front Element)**

**Input:**  
Queue

**Output:**  
Element at front of queue

If (front == -1):

Return "Queue is empty"

Else:

Return queue[front]

## Code:

#include <iostream>

using namespace std;

class circ\_arr

{

    int front = 0; // Front index

    int rear = 0;  // Rear index

    int cur = 0;   // Tracks current number of elements

    int arr[5];    // Array of size 5

public:

    void enqueue(int value)

    {

        if (cur == 5)

        {

            cout << "The Queue is full" << endl;

            return;

        }

        arr[rear] = value;

        rear = (rear + 1) % 5; // Circular increment

        cur++;                 // Increment element count

        display();

    }

    int dequeue()

    {

        if (cur == 0)

        {

            cout << "The Queue is empty" << endl;

            return 404;

        }

        int temp = arr[front];

        front = (front + 1) % 5; // Circular increment

        cur--;                   // Decrement element count

        display();

        return temp;

    }

    int peek()

    {

        if (cur == 0)

        {

            cout << "The Queue is empty" << endl;

            return 404;

        }

        return arr[front];

    }

    void display()

    {

        if (cur == 0)

        {

            cout << "Queue is empty" << endl;

            return;

        }

        int i = front;

        for (int count = 0; count < cur; count++)

        {

            cout << arr[i] << " ";

            i = (i + 1) % 5; // Circular increment

        }

        cout << endl;

    }

};

int main()

{

    circ\_arr ca;

    int exit = 1;

    while (exit == 1)

    {

        int choice, val;

        cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter the value: ";

            cin >> val;

            ca.enqueue(val);

            break;

        case 2:

            val = ca.dequeue();

            if (val != 404)

            {

                cout << "Dequeued: " << val << endl;

            }

            break;

        case 3:

            val = ca.peek();

            if (val != 404)

            {

                cout << "Front element: " << val << endl;

            }

            break;

        case 4:

            exit = 0;

            break;

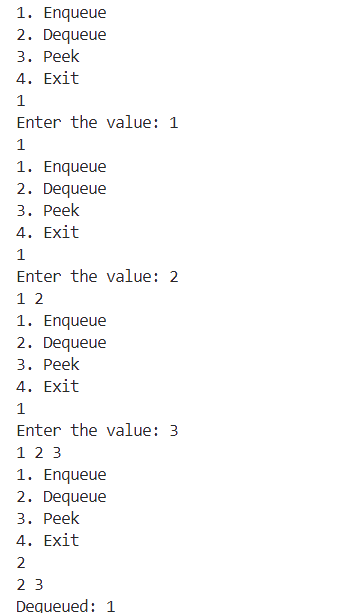
        }

    }

    return 0;

}

## Output:



3.Write a separate C++ menu-driven program to implement Queue ADT using an integer-linked list. Maintain proper boundary conditions and follow good coding practices. The Queue ADT has the following operations, Enqueue Dequeue Peek Exit

## Aim:

To write a C++ menu-driven program to implement Queue ADT using a singly linked list, ensuring boundary conditions and good coding practices.

## Algorithms:

**Algorithm 1 – Enqueue**

**Input:**  
An integer value to insert

**Output:**  
Queue with new element at the rear

Create newNode with given value

If (rear == NULL):

front ← newNode

rear ← newNode

Else:

rear → next ← newNode

rear ← newNode

**Algorithm 2 – Dequeue**

**Input:**  
Queue with elements

**Output:**  
Queue with front element removed

If (front == NULL):

Return "Queue Underflow"

temp ← front

front ← front → next

If (front == NULL):

rear ← NULL

Delete temp

**Algorithm 3 – Peek**

**Input:**  
Queue

**Output:**  
Element at front of queue

If (front == NULL):

Return "Queue is empty"

Else:

Return front → data

## Code:

#include <iostream>

using namespace std;

class queue

{

    struct node

    {

        int data;

        struct node \*next;

    } \*head;

public:

    queue()

    {

        head = nullptr;

    }

    // Create New Node

    struct node \*createnewnode(int value)

    {

        struct node \*newnode = new struct node;

        newnode->data = value;

        newnode->next = nullptr;

        return newnode;

    }

    //Display

    void display()

    {

        if (head = nullptr)

        {

            cout << "The queue is empty" << endl;

            return;

        }

        node\* temp = head;

        while (temp!=nullptr)

        {

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

            temp = temp->next;

        }

        cout << endl;

    }

    // Insert End

    void enqueue(int value)

    {

        struct node \*newnode = createnewnode(value);

        if (head == nullptr)

        {

            head = newnode;

        }

        else

        {

            struct node \*temp = head;

            while (temp->next != nullptr)

            {

                temp = temp->next;

            }

            temp->next = newnode;

        }

        display();

    }

    // Delete Beginning

    void dequeue()

    {

        if (head == nullptr)

        {

            cout << "The queue is empty";

            return;

        }

        struct node \*temp = head;

        head = temp->next;

        free(temp);

        display();

    }

    int peek()

    {

        if (head==nullptr)

        {

            cout << "The queue is empty" << endl;

            return 404;

        }

        node\* temp = head;

        while (temp->next!=nullptr)

        {

            temp = temp->next;

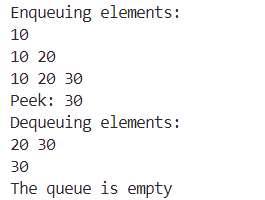
        }

        return temp->data;

    }

};

## Output:



4.Take a string from the user that consists of the '+' symbol. Process the string such that the final string does not include the '+' symbol and the immediate left non- '+' symbol.  
Select and choose the optimal ADT. Implement the program by including the appropriate header file.

## Aim:

To write a C++ program that processes a string containing the '+' symbol such that the final string excludes both the '+' and the immediate left non-'+' character, using an optimal Abstract Data Type (ADT).

## Algorithms:

**Algorithm – Process String Using Stack ADT**

**Input:**  
A string str consisting of '+' and other characters.

**Output:**  
A string with all '+' symbols and their immediate left non-'+' characters removed.

Initialize an empty stack S

For each character ch in str:

If ch is not '+':

Push ch into stack S

Else if ch == '+':

If stack S is not empty:

Pop from stack S // Remove immediate left non-'+' character

// Skip adding '+' to stack

Initialize result as an empty string

While stack S is not empty:

Pop from stack and prepend to result

Return result

## Code:

#include <iostream>

#include <stack>

#include <string>

#include <algorithm>

using namespace std;

int main()

{

    stack<int> stk;

    string str = "";

    cout << "Enter the string" << endl;

    cin >> str;

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

    {

        if (str[i]!='+')

        {

            stk.push(str[i]);

        }

        if (str[i]=='+' and (not stk.empty()))

        {

            stk.pop();

        }

    }

    str = "";

    while((not stk.empty()))

    {

        str+=(stk.top());

        stk.pop();

    }

    reverse(str.begin(), str.end());

    cout << str << endl;

    return 0;

}

## Output:



**Lab - 8**

1. There are nn block towers, numbered from 1 to nn. The ii-th tower consists of aiai​ blocks. In one move, you can move one block from tower ii to tower jj, but only if ai>ajai​>aj​. That move increases ajaj​ by 1 and decreases aiai​ by 1. You can perform as many moves as you would like (possibly, zero). What's the largest amount of blocks you can have on the tower 1 after the moves?

## Aim:

To write a C++ program using STL that finds the **maximum number of blocks** that can be moved to **Tower 1** by following the given rules.

## Algorithms:

**Algorithm – Maximize Blocks on Tower 1**

**Input:**

* t – number of test cases
* For each test case:
  + n – number of towers
  + a[] – array of integers representing blocks on each tower

**Output:**

* For each test case, print the **maximum number of blocks possible on Tower 1**

for i = 1 to t:

 read n

 read a[0...n-1]

 sum ← 0

 for j = 0 to n-1:

  sum ← sum + a[j]

 max\_blocks ← sum - (n - 1)

 print max\_blocks

## Code:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to calculate the maximum number of blocks in tower 1

long long solve(int n, vector<long long>& blocks) {

    // Initially, blocks[0] represents the number of blocks in tower 1

    long long max\_blocks\_in\_tower1 = blocks[0];

    // Create a vector of pairs (tower\_index, number\_of\_blocks)

    vector<pair<int, long long>> tower\_blocks;

    for (int i = 0; i < n; i++) {

        tower\_blocks.push\_back({i + 1, blocks[i]});

    }

    // Sort towers by number of blocks in descending order

    sort(tower\_blocks.begin(), tower\_blocks.end(),

         [](const pair<int, long long>& a, const pair<int, long long>& b) {

             return a.second > b.second;

         });

    // For each tower with more blocks than tower 1, we can move blocks to tower 1

    for (int i = 0; i < n; i++) {

        // Skip if this is tower 1

        if (tower\_blocks[i].first == 1) {

            continue;

        }

        // Find tower 1's current position in our sorted array

        int tower1\_index = -1;

        for (int j = 0; j < n; j++) {

            if (tower\_blocks[j].first == 1) {

                tower1\_index = j;

                break;

            }

        }

        // If we can't find tower 1 or tower 1 has more blocks than current tower, skip

        if (tower1\_index == -1 || tower\_blocks[tower1\_index].second >= tower\_blocks[i].second) {

            continue;

        }

        // We can move a block from this tower to tower 1

        // Each move increases tower 1's blocks by 1 and decreases the other tower by 1

        // We can keep doing this until they have the same number of blocks

        // After that, they'll keep swapping positions and we can't increase tower 1 further

        long long diff = tower\_blocks[i].second - tower\_blocks[tower1\_index].second;

        // Integer division by 2 gives us how many blocks tower 1 gains

        long long blocks\_to\_move = diff / 2;

        // Update tower 1's blocks

        tower\_blocks[tower1\_index].second += blocks\_to\_move;

        // Update max if necessary

        max\_blocks\_in\_tower1 = max(max\_blocks\_in\_tower1, tower\_blocks[tower1\_index].second);

    }

    return max\_blocks\_in\_tower1;

}

// Optimized solution

long long solveOptimized(int n, vector<long long>& blocks) {

    long long tower1\_blocks = blocks[0];

    // For each tower, find how many blocks we can add to tower 1

    for (int i = 1; i < n; i++) {

        // If current tower has more blocks than tower 1

        if (blocks[i] > tower1\_blocks) {

            // Calculate how many blocks we can move

            long long diff = blocks[i] - tower1\_blocks;

            // We can add diff/2 blocks to tower 1

            tower1\_blocks += diff / 2;

        }

    }

    return tower1\_blocks;

}

int main() {

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

    int t;

    cin >> t;  // Number of test cases

    while (t--) {

        int n;

        cin >> n;  // Number of towers

        vector<long long> blocks(n);

        for (int i = 0; i < n; i++) {

            cin >> blocks[i];  // Number of blocks in each tower

        }

        // Calculate and output the result

        cout << solveOptimized(n, blocks) << endl;

    }

    return 0;

}

## Output:



1.Write a separate C++ menu-driven program to implement Tree ADT using a character binary tree. Maintain proper boundary conditions and follow good coding practices. The Tree ADT has the following operations,

1. Insert
2. Preorder
3. Inorder
4. Postorder
5. Search
6. Exit

## Aim:

To implement Tree ADT using a character binary tree with basic operations.

## Algorithms:

**Algorithm 1 – Insert Node**

**Input:** root, char val  
**Output:** updated tree

if root == NULL →

 create new node with val

 return node

else

 if val < root->data → root->left ← insert(root->left, val)

 else → root->right ← insert(root->right, val)

return root

**Algorithm 2 – Preorder Traversal**

**Input:** root  
**Output:** nodes in preorder

if root ≠ NULL →

 print root->data

 preorder(root->left)

 preorder(root->right)

**Algorithm 3 – Inorder Traversal**

**Input:** root  
**Output:** nodes in inorder

if root ≠ NULL →

 inorder(root->left)

 print root->data

 inorder(root->right)

**Algorithm 4 – Postorder Traversal**

**Input:** root  
**Output:** nodes in postorder

if root ≠ NULL →

 postorder(root->left)

 postorder(root->right)

 print root->data

**Algorithm 5 – Search Node**

**Input:** root, char key  
**Output:** found / not found

if root == NULL → return false

if root->data == key → return true

if key < root->data → return search(root->left, key)

else → return search(root->right, key)

## Code:

#include <iostream>

#include <queue>

using namespace std;

class binary\_tree

{

    struct node

    {

        int data;

        node \*left;

        node \*right;

    };

public:

    node\* root;

    binary\_tree()

    {

        root = nullptr;

    }

    void insert(node \*&root, int value)

    {

        node \*newNode = new node;

        newNode->data = value;

        newNode->left = newNode->right = NULL;

        if (root == NULL)

        {

            root = newNode;

            return;

        }

        queue<node \*> q;

        q.push(root);

        while (!q.empty())

        {

            node \*temp = q.front();

            q.pop();

            if (temp->left == NULL)

            {

                temp->left = newNode;

                break;

            }

            else

            {

                q.push(temp->left);

            }

            if (temp->right == NULL)

            {

                temp->right = newNode;

                break;

            }

            else

            {

                q.push(temp->right);

            }

        }

    }

    void inorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        inorder(root->left);

        cout << root->data << endl;

        inorder(root->right);

    }

    void preorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        cout << root->data << endl;

        preorder(root->left);

        preorder(root->right);

    }

    void postorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        postorder(root->left);

        postorder(root->right);

        cout << root->data << endl;

    }

    bool search(node \*root, int value)

    {

        if (root == nullptr)

        {

            return false;

        }

        queue<node \*> q;

        q.push(root);

        while (!q.empty())

        {

            node \*temp = q.front();

            q.pop();

            if (temp->data == value)

            {

                return true;

            }

            if (temp->left)

            {

                q.push(temp->left);

            }

            if (temp->right)

            {

                q.push(temp->right);

            }

        }

        return false;

    }

};

int main()

{

    binary\_tree bt;

    int choice, value;

    while (true)

    {

        cout << "\n1. Insert\n2. Inorder\n3. Preorder\n4. Postorder\n5. Search\n6. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter value to insert: ";

            cin >> value;

            bt.insert(bt.root, value); // Inserting into the tree

            break;

        case 2:

            cout << "Inorder Traversal: ";

            bt.inorder(bt.root);

            cout << endl;

            break;

        case 3:

            cout << "Preorder Traversal: ";

            bt.preorder(bt.root);

            cout << endl;

            break;

        case 4:

            cout << "Postorder Traversal: ";

            bt.postorder(bt.root);

            cout << endl;

            break;

        case 5:

            cout << "Enter value to search: ";

            cin >> value;

            if (bt.search(bt.root, value))

                cout << "Value found!" << endl;

            else

                cout << "Value not found!" << endl;

            break;

        case 6:

            cout << "Exiting..." << endl;

            return 0;

        default:

            cout << "Invalid choice. Try again." << endl;

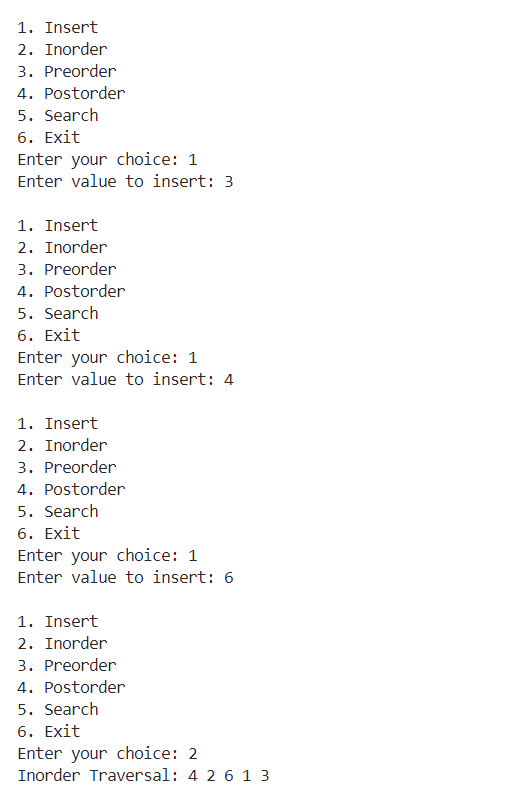
        }

    }

    return 0;

}

## Output:



**Lab - 9**

Nene invented a new game based on an increasing sequence of integers a1,a2,…,aka1​,a2​,…,ak​.

In this game, initially nn players are lined up in a row. In each of the rounds of this game, the following happens:

* Nene finds the a1a1​-th, a2a2​-th, ……, akak​-th players in a row. They are kicked out of the game simultaneously. If the ii-th player in a row should be kicked out, but there are fewer than ii players in a row, they are skipped.

Once no one is kicked out of the game in some round, all the players that are still in the game are declared as winners.

For example, consider the game with a=[3,5]a=[3,5] and n=5n=5 players. Let the players be named player A, player B, ……, player E in the order they are lined up initially. Then,

* Before the first round, players are lined up as ABCDE. Nene finds the 3-rd and the 5-th players in a row. These are players C and E. They are kicked out in the first round.
* Now players are lined up as ABD. Nene finds the 3-rd and the 5-th players in a row. The 3-rd player is player D and there is no 5-th player in a row. Thus, only player D is kicked out in the second round.
* In the third round, no one is kicked out of the game, so the game ends after this round.
* Players A and B are declared as the winners.

## Aim:

To modify each element of array n such that n[i] becomes less than the smallest element in array a.

## Algorithms:

**Algorithm – Process Arrays**

**Input:** Integers k, q; arrays a[k], n[q]  
**Output:** Modified n

1. Read k, q

2. Input a[k], then sort a

3. Input n[q]

4. For each n[i] →

  if n[i] ≥ a[0] → n[i] ← a[0]-1

5. Print n[]

## Code:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

    int k,q;

    cout << "Enter k and q" << endl;

    cin >> k >> q;

    vector<int> a(k);

    vector<int> n(q);

    cout << "Enter the values of a" << endl;

    for (int i = 0; i<k; i++)

    {

        cin >> a[i];

    }

    sort(a.begin(),a.end());

    cout << "Enter the values of n" << endl;

    for (int i = 0; i<q; i++)

    {

        cin >> n[i];

    }

    for (int i = 0; i<q; i++)

    {

        if (n[i]>=a[0])

        {

            n[i] = a[0]-1;

        }

    }

    for (int i = 0; i<q; i++)

    {

        cout << n[i] << " ";

    }

    cout << endl;

}

2.Write a separate C++ menu-driven program to implement Tree ADT using a binary search tree. Maintain proper boundary conditions and follow good coding practices. The Tree ADT has the following operations,

1. Insert
2. Preorder
3. Inorder
4. Postorder
5. Search
6. Exit

## Aim:

To implement Tree ADT using Binary Search Tree (BST) with basic operations.

## Algorithms:

**Algorithm – BST Operations**

**Input:** Integer/key  
**Output:** Tree with BST properties maintained

1. Insert →

  If root == NULL → new node ← root

  Else traverse: key < curr → left, key > curr → right

2. Preorder →

  Visit → Left → Right (recursively)

3. Inorder →

  Left → Visit → Right (recursively)

4. Postorder →

  Left → Right → Visit (recursively)

5. Search →

  If key == curr → found

  Else if key < curr → search left

  Else → search right

## Code:

#include <iostream>

using namespace std;

class binary\_search\_tree

{

    struct node

    {

        node \*left;

        node \*right;

        int value;

    };

public:

    node \*root;

    binary\_search\_tree()

    {

        root = nullptr;

    }

    void insert(int value)

    {

        node \*newnode = new node;

        newnode->left = newnode->right = nullptr;

        newnode->value = value;

        if (root == nullptr)

        {

            root = newnode;

            return;

        }

        node \*temp = root;

        while (temp != nullptr)

        {

            if (temp->value > value)

            {

                if (temp->left == nullptr)

                {

                    temp->left = newnode;

                    return;

                }

                temp = temp->left;

            }

            else

            {

                if (temp->right == nullptr)

                {

                    temp->right = newnode;

                    return;

                }

                temp = temp->right;

            }

        }

    }

    void inorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        inorder(root->left);

        cout << root->value << endl;

        inorder(root->right);

    }

    void preorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        cout << root->value << endl;

        preorder(root->left);

        preorder(root->right);

    }

    void postorder(node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        postorder(root->left);

        postorder(root->right);

        cout << root->value << endl;

    }

    void search(int value)

    {

        if (root == nullptr)

        {

            cout << "Empty" << endl;

            return;

        }

        node \*temp = root;

        while (temp != nullptr)

        {

            if (temp->value == value)

            {

                cout << "The value is found" << endl;

                return;

            }

            else if (value < temp->value)

            {

                temp = temp->left;

            }

            else if (value > temp->value)

            {

                temp = temp->right;

            }

        }

        if (temp == nullptr)

        {

            cout << "The value is not found" << endl;

            return;

        }

    }

};

int main()

{

    binary\_search\_tree bst;

    int choice, value;

    while (true)

    {

        cout << "\n1. Insert\n2. Inorder\n3. Preorder\n4. Postorder\n5. Search\n6. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter value to insert: ";

            cin >> value;

            bst.insert(value); // Inserting into the tree

            break;

        case 2:

            cout << "Inorder Traversal: " << endl;

            bst.inorder(bst.root);

            cout << endl;

            break;

        case 3:

            cout << "Preorder Traversal: " << endl;

            bst.preorder(bst.root);

            cout << endl;

            break;

        case 4:

            cout << "Postorder Traversal: " << endl;

            bst.postorder(bst.root);

            cout << endl;

            break;

        case 5:

            cout << "Enter value to search: ";

            cin >> value;

            bst.search(value);

            break;

        case 6:

            cout << "Exiting..." << endl;

            return 0;

        default:

            cout << "Invalid choice. Try again." << endl;

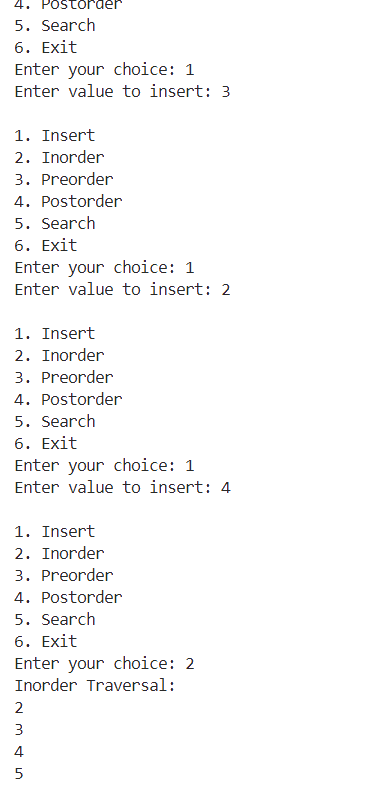
        }

    }

    return 0;

}

## Output:



3.Add a "construct expression tree" method to the binary tree data structure from the previous lab code—import stack from the standard template library (STL) to construct the expression tree. Import the Tree ADT program into another program that gets a valid postfix expression, constructs, and prints the expression tree. It consists of the following operations.

1. Postfix Expression
2. Construct Expression Tree
3. Preorder
4. Inorder
5. Postorder
6. Exit

## Aim:

To construct an expression tree using a postfix expression and perform standard traversals using a binary tree.

## Algorithms:

**Construct Expression Tree →**

**Input:** Postfix expression  
**Output:** Root of expression tree

1. Init empty stack<TreeNode\*> stk

2. For each char ch in postfix:

  a. If ch is operand →

   Create new node with ch → push to stk

  b. Else if ch is operator →

   Pop 2 nodes (right, left)

   Create new node with ch as root

   Set left, right children

   Push new root to stk

1. Root of expression tree ← stk.top()

Visit → Left → Right

Left → Visit → Right

Left → Right → Visit

## Code:

#include <iostream>

#include <stack>

#include <cctype> // For isalpha(), isdigit()

#include <string>

using namespace std;

class ExpressionTree

{

private:

    // Define a tree node structure

    struct Node

    {

        char value;

        Node \*left;

        Node \*right;

        Node(char val) : value(val), left(nullptr), right(nullptr) {}

    };

public:

    Node \*root;

    // Constructor

    ExpressionTree() : root(nullptr) {}

    // Public function to build the expression tree from a postfix string

    void buildFromPostfix(const string &postfix)

    {

        // Use a stack to hold tree nodes

        stack<Node \*> st;

        for (char ch : postfix)

        {

            // If it's an operand (letter or digit), create a new node and push

            if (isalpha(ch))

            {

                Node \*newNode = new Node(ch);

                st.push(newNode);

            }

            // If it's an operator, pop two nodes, make them children, push back

            else if (isdigit(ch))

            {

                // Pop two nodes

                if (st.size() < 2)

                {

                    cerr << "Invalid postfix expression!" << endl;

                    return;

                }

                Node \*rightNode = st.top();

                st.pop();

                Node \*leftNode = st.top();

                st.pop();

                // Create a new node for the operator

                Node \*newNode = new Node(ch);

                newNode->left = leftNode;

                newNode->right = rightNode;

                // Push the operator node back onto the stack

                st.push(newNode);

            }

            // Ignore any whitespace or invalid characters

        }

        // After processing all characters, the stack should have exactly one node

        if (st.size() == 1)

        {

            root = st.top();

            st.pop();

        }

        else

        {

            cerr << "Invalid postfix expression or mismatch in operands/operators!" << endl;

            root = nullptr;

        }

    }

    void inorder(Node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        inorder(root->left);

        cout << root->value << endl;

        inorder(root->right);

    }

    void preorder(Node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        cout << root->value << endl;

        preorder(root->left);

        preorder(root->right);

    }

    void postorder(Node \*root)

    {

        if (root == nullptr)

        {

            return;

        }

        postorder(root->left);

        postorder(root->right);

        cout << root->value << endl;

    }

};

int main()

{

    ExpressionTree et;

    string postfix;

    int choice;

    while (true)

    {

        cout << "\n---- MENU ----\n"

             << "1. Postfix Expression\n"

             << "2. Construct Expression Tree\n"

             << "3. Preorder\n"

             << "4. Inorder\n"

             << "5. Postorder\n"

             << "6. Exit\n"

             << "Enter your choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "Enter a valid postfix expression (single-char operands): ";

            cin >> postfix;

            break;

        case 2:

            et.buildFromPostfix(postfix);

            cout << "Expression tree constructed.\n";

            break;

        case 3:

            cout << "Preorder Traversal: ";

            et.preorder(et.root);

            break;

        case 4:

            cout << "Inorder Traversal: ";

            et.inorder(et.root);

            break;

        case 5:

            cout << "Postorder Traversal: ";

            et.postorder(et.root);

            break;

        case 6:

            cout << "Exiting...\n";

            return 0;

        default:

            cout << "Invalid choice. Try again.\n";

        }

    }

    return 0;

}

**Lab - 10**

Write a C++ program (using STL if needed) to solve the following: Polycarp has a sequence of integers a of length n (1 ≤ aᵢ ≤ n). The sequence makes him happy only if all elements are distinct. To achieve this, he can perform moves where he removes the first (leftmost) element of the sequence. Task: Find the minimum number of moves needed so the remaining sequence has all distinct elements. Input: First line: t (number of test cases, 1 ≤ t ≤ 10⁴) For each test case: n (sequence length, 1 ≤ n ≤ 2·10⁵) a₁, a₂, ..., aₙ (sequence elements, 1 ≤ aᵢ ≤ n) Constraints: Sum of n across all test cases ≤ 2·10⁵ Output: For each test case, print the minimum number of elements to remove from the beginning to make all remaining elements distinct.

## Aim:

To write a C++ program using STL to find the minimum number of prefix deletions required so that the remaining elements in a sequence are all distinct.

## Algorithms:

**Input:**

* t: test cases
* Each test case:  
   • n: length of array  
   • a[]: array of integers

**Output:**

* Minimum number of prefix deletions such that remaining elements are all distinct

for each test case:

read n and a[0...n-1]

initialize set seen

start from right end:

for i from n-1 to 0:

if a[i] not in seen:

insert a[i] in seen

else:

break

result ← i+1 (minimum deletions)

print result

## Code:

#include <iostream>

#include <unordered\_set>

#include <vector>

using namespace std;

int main()

{

    int test\_case;

    cin>>test\_case;

    for(int i  = 0 ; i < test\_case;i++)

    {

        int size;

        vector <int> v1;

        unordered\_set<int>u1;

        cin >> size;

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

        {

            int element;

            cin >> element;

            v1.push\_back(element);

        }

        int count = 1;

        u1.insert(v1[size-1]);

        if(size == 1)

        {

            cout<<0<<endl;

        }

        for (int k = size -2; k >= 0; k--)

        {

            if(u1.count(v1[k]) == 0)

            {

                u1.insert(v1[k]);

                count ++;

                if(count == size){

                    cout<<size-count<<endl;

                    break;

                }

            }

            else{

                cout<<(size - count)<<endl;

                break;

            }

        }

        v1.clear();

        u1.clear();

    }

    return 0;

}

2.C. Write a separate C++ menu-driven program to implement Priority Queue ADT using a max heap. Maintain proper boundary conditions and follow good coding practices. The Priority Queue ADT has the following operations,

1. Insert
2. Delete (extract max)
3. Display
4. Search
5. Sort (Heap Sort)
6. Exit

## Aim:

Implement **Priority Queue ADT** using **Max Heap** with operations:

## Algorithms:

 **Insert:**

* Add element at end -> **heapify-up** to maintain max-heap.

 **Delete (Extract Max):**

* Remove root -> Replace root with last element -> **heapify-down** to restore heap.

 **Display:**

* Print elements in the heap.

 **Search:**

* Linear search through heap.

 **Sort (Heap Sort):**

* Build max-heap -> Extract max -> Place at end -> **heapify-down** -> Repeat.

## Code:

#include <iostream>

#include <vector>

using namespace std;

class maxheap {

    vector<int> heap = {0};  // 1-based indexing: index 0 is a placeholder

    int cur;                 // cur is the next available index

public:

    maxheap() {

        cur = 1;           // Heap is empty when cur == 1

    }

    void heapifyup() {

        int temp = cur - 1;  // new element is at cur-1

        while (temp > 1 && heap[temp / 2] < heap[temp]) {

            // Swap the current element with its parent

            int temp1 = heap[temp];

            heap[temp] = heap[temp / 2];

            heap[temp / 2] = temp1;

            temp = temp / 2;

        }

    }

    void heapifydown() {

        int index = 1;  // start at the root

        while (index \* 2 < cur) {  // while there is at least a left child

            int left = index \* 2;

            int right = left + 1;

            int maxIndex = left;  // assume left child is larger

            // If right child exists and is greater than left child, update maxIndex

            if (right < cur && heap[right] > heap[left]) {

                maxIndex = right;

            }

            // If the current node is larger than the larger child, break

            if (heap[index] >= heap[maxIndex]) {

                break;

            }

            // Swap current node with the larger child

            int temp = heap[index];

            heap[index] = heap[maxIndex];

            heap[maxIndex] = temp;

            // Continue heapifying down from the swapped child index

            index = maxIndex;

        }

    }

    void insert(int value) {

        heap.push\_back(value);

        cur++;

        heapifyup();

    }

    int del() {

        if (cur == 1) {

            cout << "The heap is empty" << endl;

            return -1;

        }

        int maxvalue = heap[1];         // store the max value (root)

        heap[1] = heap[cur - 1];          // move the last element to the root

        heap.pop\_back();                  // remove the last element

        cur--;                          // decrease heap size

        heapifydown();                  // restore the heap property

        return maxvalue;

    }

    void display() {

        if (cur == 1) {

            cout << "Empty" << endl;

            return;

        }

        // Display elements from index 1 to cur - 1

        for (int i = 1; i < cur; i++) {

            cout << heap[i] << " ";

        }

        cout << endl;

    }

    void search(int value) {

        for (int i = 1; i < cur; i++) {

            if (heap[i] == value) {

                cout << "The value " << value << " is found." << endl;

                return;

            }

        }

        cout << "Not found" << endl;

    }

    void sort() {

        vector<int> sorted;

        int originalCur = cur;

        vector<int> originalHeap = heap;

        // Repeatedly delete the max and store it

        while (cur > 1) {

            int maxVal = del();

            sorted.push\_back(maxVal);

        }

        cout << "Sorted (descending order): ";

        for (int val : sorted) {

            cout << val << " ";

        }

        cout << endl;

        // Restore the original heap (if needed)

        heap = originalHeap;

        cur = originalCur;

    }

};

int main() {

    maxheap heap;

    int choice, value;

    while (true) {

        cout << "\n--- Priority Queue Menu ---\n";

        cout << "1. Insert\n";

        cout << "2. Delete\n";

        cout << "3. Display\n";

        cout << "4. Search\n";

        cout << "5. Sort (Heap Sort)\n";

        cout << "6. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                cout << "Enter value to insert: ";

                cin >> value;

                heap.insert(value);

                break;

            case 2:

                value = heap.del();

                if (value != -1)

                    cout << "Deleted: " << value << endl;

                break;

            case 3:

                cout << "Heap elements: ";

                heap.display();

                break;

            case 4:

                cout << "Enter value to search: ";

                cin >> value;

                heap.search(value);

                break;

            case 5:

                heap.sort();

                break;

            case 6:

                cout << "Exiting..." << endl;

                return 0;

            default:

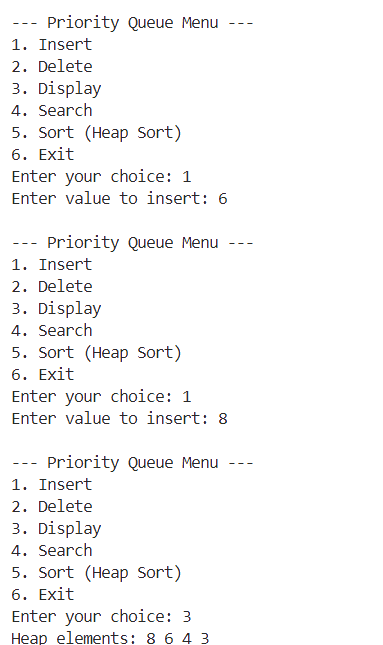
                cout << "Invalid choice. Please try again." << endl;

        }

    }

}

## Output:



**Lab - 11**

Write a separate C++ menu-driven program to implement Hash ADT with Linear Probing.  
Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert
2. Delete
3. Search
4. Display
5. Exit

## Aim:

Implement **Hash ADT** with **Linear Probing** for collision resolution

## Algorithms:

**Insert:**

If table is full → return false.  
Compute index: index = key % table\_size.  
If table[index] is empty → insert key at index.  
Else → probe linearly:  
While table[index] is not empty → index = (index + 1) % table\_size.  
Insert key at new index.

**Delete:**

Compute index: index = key % table\_size.  
If table[index] is empty → return false (key not found).  
If table[index] == key → mark table[index] as deleted (set to a sentinel value like -1).  
Else → probe linearly:  
While table[index] is not empty →  
If table[index] == key → delete key at index.  
Else → index = (index + 1) % table\_size.

**Search:**

Compute index: index = key % table\_size.  
If table[index] is empty → return false.  
If table[index] == key → return true.  
Else → probe linearly:  
While table[index] is not empty →  
If table[index] == key → return true.  
Else → index = (index + 1) % table\_size.  
Return false if key not found after probing.

**Display:**

For each index in the table:  
If table[index] is not empty → print table[index].  
If no keys are present → print "Table is empty."

## Code:

#include <iostream>

using namespace std;

const int SIZE = 10;

const int EMPTY = -1;

const int DELETE = -2;

class linear\_probbing

{

    int table[SIZE];

    int hash(int num)

    {

        int index = num % SIZE;

        return index;

    }

public:

    linear\_probbing()

    {

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

        {

            table[i] = EMPTY;

        }

    }

    void display()

    {

        for (int i = 0; i<SIZE; i++)

        {

            if(table[i] == EMPTY)

            {

                cout << "Empty" << " ";

            }

            else if(table[i] == DELETE)

            {

                cout << "Deleted" << " ";

            }

            else

            {

                cout << table[i] << " ";

            }

        }

        cout << endl;

    }

    void insert(int key)

    {

        int index = hash(key);

        int temp = index;

        while(table[index] != EMPTY and table[index] != DELETE)

        {

            index = (index+1)%SIZE;

            if(index == temp)

            {

                cout << "The table is full" << endl;

                break;

            }

        }

        table[index] = key;

    }

    void del(int key)

    {

        int index = hash(key);

        int temp = index;

        while (table[index] != key)

        {

            index = (index+1)%SIZE;

            if(index == temp)

            {

                cout << "The key is not found" << endl;

                break;

            }

        }

        table[index] = DELETE;

    }

    void search(int key)

    {

        int index = hash(key);

        int temp = index;

        while (table[index] != key)

        {

            index = (index+1)%SIZE;

            if(index == temp)

            {

                cout << "The key is not found" << endl;

                return;

            }

        }

        cout << "The element is found" << endl;

    }

};

int main()

{

    int exit = 1;

    int choice, value;

    linear\_probbing lp;

    while(exit)

    {

        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch(choice)

        {

            case(1):

                cout << "Enter the value" << endl;

                cin >> value;

                lp.insert(value);

                break;

            case(2):

                cout << "Enter the value to delete" << endl;

                cin >> value;

                lp.del(value);

                break;

            case(3):

                cout << "Enter the value to search" << endl;

                cin >> value;

                lp.search(value);

                break;

            case(4):

                lp.display();

                break;

            case(5):

                exit = 0;

        }

    }

}

2. Write a separate C++ menu-driven program to implement Hash ADT with Quadratic Probing. Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert
2. Delete
3. Search
4. Display
5. Exit

## Aim:

Write a separate C++ menu-driven program to implement Hash ADT using **Quadratic Probing**. Maintain proper boundary conditions and follow good coding practice

## Algorithms:

**nsert(key):**  
index → key % size  
If table[index] is empty → table[index] = key  
Else →  
 i = 1  
 while i < size →  
  new\_index → (index + i\*i) % size  
  If table[new\_index] is empty → table[new\_index] = key → return  
  i++

**Delete(key):**  
index → key % size  
If table[index] == key → table[index] = DELETED  
Else →  
 i = 1  
 while i < size and table[(index + i*i) % size] ≠ EMPTY →  
  If table[(index + i*i) % size] == key → table[...] = DELETED → return  
  i++

**Search(key):**  
index → key % size  
If table[index] == key → return true  
Else →  
 i = 1  
 while i < size and table[(index + i\*i) % size] ≠ EMPTY →  
  If table[...] == key → return true  
  i++  
return false

**Display():**  
for i = 0 to size - 1 →  
 If table[i] is not EMPTY → print table[i]  
 Else → print “EMPTY”

## Code:

#include <iostream>

using namespace std;

const int SIZE = 10;

const int EMPTY = -1;

const int DELETE = -2;

class quad\_prob

{

    int table[SIZE];

    int hash(int key)

    {

        int index = key % SIZE;

        return index;

    }

public:

    quad\_prob()

    {

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

        {

            table[i] = EMPTY;

        }

    }

    void display()

    {

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

        {

            if (table[i] == EMPTY)

            {

                cout << "Empty" << " ";

            }

            else if (table[i] == DELETE)

            {

                cout << "Deleted" << " ";

            }

            else

            {

                cout << table[i] << " ";

            }

        }

        cout << endl;

    }

    void insert(int key)

    {

        int i = 1;

        int index = hash(key);

        int newindex = index;

        while (table[newindex] != EMPTY and table[newindex] != DELETE)

        {

            if (i == SIZE)

            {

                cout << "The table is full" << endl;

                return;

            }

            newindex = (index + (i \* i)) % SIZE;

            i++;

        }

        table[newindex] = key;

    }

    void del(int key)

    {

        int i = 1;

        int index = hash(key);

        int newindex = index;

        while (table[newindex] != key)

        {

            if (i == SIZE)

            {

                cout << "Element not found" << endl;

                return;

            }

            newindex = (index + (i \* i)) % SIZE;

            i++;

        }

        table[newindex] = DELETE;

    }

    void search(int key)

    {

        int i = 1;

        int index = hash(key);

        int newindex = index;

        while (table[newindex] != key)

        {

            if (i == SIZE)

            {

                cout << "Element not found" << endl;

                return;

            }

            newindex = (index + (i \* i)) % SIZE;

            i++;

        }

        cout << "The element is found" << endl;

    }

};

int main()

{

    int exit = 1;

    int choice, value;

    quad\_prob qp;

    while (exit)

    {

        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case (1):

            cout << "Enter the value" << endl;

            cin >> value;

            qp.insert(value);

            break;

        case (2):

            cout << "Enter the element to delete" << endl;

            cin >> value;

            qp.del(value);

            break;

        case (3):

            cout << "Enter the element to search" << endl;

            cin >> value;

            qp.search(value);

            break;

        case (4):

            qp.display();

            break;

        case(5):

            exit = 0;

        }

    }

}

3.Write a separate C++ menu-driven program to implement Hash ADT with Separate Chaining. Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert
2. Delete
3. Search
4. Display
5. Exit

## Aim:

Write a separate C++ menu-driven program to implement Hash ADT using **Separate Chaining**. Maintain proper boundary conditions and follow good coding practices

## Algorithms:

**Insert(key):**  
index → key % size  
append key to chain at table[index]

**Delete(key):**  
index → key % size  
search for key in chain at table[index]  
if found → delete key from chain

**Search(key):**  
index → key % size  
search for key in chain at table[index]  
if found → return true  
else → return false

**Display():**  
for i = 0 to size - 1 →  
 print "Index i →"  
 for each element in chain at table[i] → print element

## Code:

#include <iostream>

using namespace std;

const int SIZE = 10;

const int EMPTY = -1;

const int DELETE = -2;

class sep\_chain

{

    struct node

    {

        int data;

        node \*next;

    } \*head;

    node \*table[SIZE];

    int hash(int key)

    {

        int index = key % SIZE;

        return index;

    }

public:

    sep\_chain()

    {

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

        {

            table[i] = nullptr;

        }

    }

    void display()

    {

        node \*temp;

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

        {

            if (table[i] != nullptr)

            {

                temp = table[i];

                while (temp != nullptr)

                {

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

                    temp = temp->next;

                }

            }

            else

            {

                cout << "Empty";

            }

            cout << endl;

        }

    }

    void insert(int key)

    {

        int index = hash(key);

        node \*newnode = new node;

        newnode->data = key;

        newnode->next = nullptr;

        node \*temp;

        if (table[index] == nullptr)

        {

            table[index] = newnode;

        }

        else

        {

            temp = table[index];

            table[index] = newnode;

            newnode->next = temp;

        }

    }

    void del(int key)

    {

        int index = hash(key);

        node\* temp1 = table[index];

        node\* temp2 = table[index];

        if (table[index] == nullptr)

        {

            cout << "No element found" << endl;

        }

        else if (table[index]->data == key)

        {

            table[index] = table[index]->next;

        }

        else

        {

            while(temp1->data != key or temp1!=nullptr)

            {

                temp2 = temp1;

                temp1 = temp1->next;

            }

            if(temp1->next == nullptr)

            {

                temp2->next = nullptr;

                delete temp1;

            }

            else if (temp1->next != nullptr)

            {

                temp2->next = temp1->next;

                delete temp1;

            }

        }

    }

    void search(int key)

    {

        int index = hash(key);

        node\* temp = table[index];

        while(temp!=nullptr)

        {

            if(temp->data == key)

            {

                cout << "The value is found" << endl;

                return;

            }

        }

        cout << "The value is not found" << endl;

    }

};

int main()

{

    int exit = 1;

    int choice, value;

    sep\_chain sc;

    while (exit)

    {

        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case (1):

            cout << "Enter the value" << endl;

            cin >> value;

            sc.insert(value);

            break;

        case(2):

            cout << "Enter the value to delete" << endl;

            cin >> value;

            sc.del(value);

            break;

        case(3):

            cout << "Enter the value to search" << endl;

            cin >> value;

            sc.search(value);

            break;

        case (4):

            sc.display();

            break;

        case(5):

            exit = 0;

        }

    }

}

**Lab - 12**

Write a separate C++ menu-driven program to implement Graph ADT with an adjacency matrix. Maintain proper boundary conditions and follow good coding practices. The Graph ADT has the following operations,

1. Insert Vertex/Edge
2. Delete Vertex/Edge
3. Search Vertex/Edge
4. Display Graph
5. Exit

## Aim:

Write a C++ menu-driven program to implement Graph ADT using an **adjacency matrix**. Maintain proper boundary conditions and follow good coding practices. Operations include Insert, Delete, Search, Display, and Exit.

## Algorithms:

**Insert Vertex():**  
Input → vertex label  
if vertex\_count < MAX →  
 add vertex to vertex\_list  
 increase vertex\_count  
 Output → "Vertex inserted"  
else → Output → "Vertex limit reached"

**Insert Edge():**  
Input → start\_vertex, end\_vertex  
if both vertices exist →  
 adj[start][end] → 1  
 adj[end][start] → 1 (if undirected)  
 Output → "Edge inserted"  
else → Output → "Invalid vertices"

**Delete Vertex():**  
Input → vertex  
if vertex exists →  
 remove row and column of vertex from matrix  
 shift remaining vertices  
 decrease vertex\_count  
 Output → "Vertex deleted"  
else → Output → "Vertex not found"

**Delete Edge():**  
Input → start\_vertex, end\_vertex  
if edge exists →  
 adj[start][end] → 0  
 adj[end][start] → 0 (if undirected)  
 Output → "Edge deleted"  
else → Output → "Edge not found"

**Search Vertex():**  
Input → vertex  
if vertex in vertex\_list →  
 Output → "Vertex found"  
else → Output → "Vertex not found"

**Search Edge():**  
Input → start\_vertex, end\_vertex  
if adj[start][end] == 1 →  
 Output → "Edge found"  
else → Output → "Edge not found"

**Display():**  
Input → none  
for i = 0 to vertex\_count  
 print row i of adj matrix  
Output → "Adjacency matrix of graph"

## Code:

#include <iostream>

#include <vector>

using namespace std;

class adj\_mat

{

    int row = 5;

    int col = 5;

    int mat[5][5];

public:

    adj\_mat()

    {

        for (int q = 0; q<row; q++)

        {

            for (int k = 0; k<col; k++)

            {

                mat[q][k] = 0;

            }

        }

    }

    void insert(int i, int j)

    {

        if (mat[i][j] == 1)

        {

            cout << "Already exist" << endl;

            return;

        }

        mat[i][j] = 1;

        mat[j][i] = 1;

    }

    void del(int i, int j)

    {

        mat[i][j] = 0;

        mat[j][i] = 0;

    }

    void search(int i, int j)

    {

        if (mat[i][j]==1)

        {

            cout << "Available" << endl;

            return;

        }

        cout << "Not Available" <<endl;

    }

    void display()

    {

        for (int a = 0; a<row; a++)

        {

            for (int h = 0; h<col; h++)

            {

                cout << mat[a][h] << " ";

            }

            cout << endl;

        }

    }

};

int main()

{

    int choice, i,j,exit = 1;

    adj\_mat am;

    while(exit==1)

    {

        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case (1):

            cout << "Enter the vertex values: i, j" << endl;

            cin >> i;

            cin >> j;

            am.insert(i-1,j-1);

            break;

        case(2):

            cout << "Enter the vertex values to delete: i, j" << endl;

            cin >> i;

            cin >> j;

            am.del(i-1,j-1);

            break;

        case(3):

            cout << "Enter the vertex values to search: i, j" << endl;

            cin >> i;

            cin >> j;

            am.search(i-1,j-1);

            break;

        case (4):

            am.display();

            break;

        case(5):

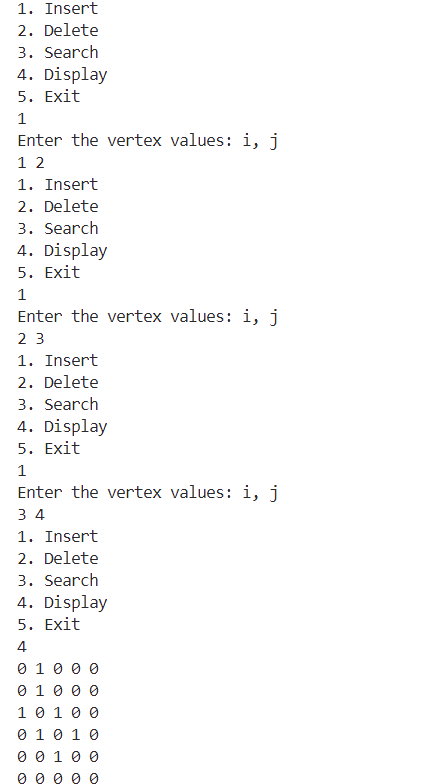
            exit = 0;

        }

    }

}

## Output:



2.Write a separate C++ menu-driven program to implement Graph ADT with an adjacency list. Maintain proper boundary conditions and follow good coding practices. The Graph ADT has the following operations,

1. Insert Vertex/Edge
2. Delete Vertex/Edge
3. Search Vertex/Edge
4. Display Graph
5. Exit

**AIM:**

Write a C++ menu-driven program to implement **Graph ADT using an adjacency list**. Maintain proper boundary conditions and follow good coding practices. The operations include Insert, Delete, Search, Display, and Exit.

**Algorithm:**

**Insert Vertex():**  
Input → vertex  
if vertex not in graph →  
 create empty list for vertex  
 Output → "Vertex inserted"  
else → Output → "Vertex exists"

**Insert Edge():**  
Input → vertex1, vertex2  
if both vertices exist →  
 add vertex2 to vertex1's list  
 add vertex1 to vertex2's list (if undirected)  
 Output → "Edge inserted"  
else → Output → "Invalid vertex/vertices"

**Delete Vertex():**  
Input → vertex  
if vertex in graph →  
 remove all edges pointing to vertex  
 delete vertex from list  
 Output → "Vertex deleted"  
else → Output → "Vertex not found"

**Delete Edge():**  
Input → vertex1, vertex2  
if edge exists →  
 remove vertex2 from vertex1’s list  
 remove vertex1 from vertex2’s list (if undirected)  
 Output → "Edge deleted"  
else → Output → "Edge not found"

**Search Vertex():**  
Input → vertex  
if vertex in graph → Output → "Vertex found"  
else → Output → "Vertex not found"

**Search Edge():**  
Input → vertex1, vertex2  
if vertex2 in vertex1’s list → Output → "Edge found"  
else → Output → "Edge not found"

**Display():**  
Input → none  
for each vertex in graph →  
 print vertex and its adjacency list  
Output → "Adjacency list displayed"

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <functional>

using namespace std;

class adj\_list

{

    struct Node

    {

        vector<int> data;

        Node \*next;

        Node() : data(2) {}

    };

    Node \*lst[7];

public:

    adj\_list()

    {

        for (int i = 0; i < 7; i++)

        {

            lst[i] = nullptr;

        }

    }

    bool search(int i, int j)

    {

        Node \*head = lst[i];

        while (head != nullptr)

        {

            if ((head->data)[0] == j)

            {

                return true;

            }

            head = head->next;

        }

        return false;

    }

    Node \*insert\_linked\_list(Node \*head, int i, int j, int wght)

    {

        Node \*newnode = new Node;

        newnode->next = head;

        newnode->data[0] = i;

        (newnode->data)[1] = j;

        (newnode->data)[2] = wght;

        head = newnode;

        return head;

    }

    void insert(int i, int j, int wght)

    {

        if (!search(i, j))

        {

            lst[i] = insert\_linked\_list(lst[i], i, j, wght);

            lst[j] = insert\_linked\_list(lst[j], j, i, wght);

        }

    }

    void del(int i, int j)

    {

        Node \*head1 = lst[i];

        Node \*head2 = lst[j];

        Node \*temp;

        bool found;

        if (head1 == nullptr or head2 == nullptr)

        {

            cout << "Not Available" << endl;

            return;

        }

        if (head1->data[1] == j)

        {

            lst[i] = head1->next;

            found = true;

        }

        while (head1->next != nullptr and found != true)

        {

            if (head1->next->data[1] == j)

            {

                temp = head1->next->next;

                head1->next = temp;

                lst[i] = head1;

                break;

            }

            head1 = head1->next;

        }

        if (head2->data[1] == i)

        {

            lst[j] = head2->next;

            return;

        }

        while (head2->next != nullptr)

        {

            if (head2->next->data[1] == i)

            {

                temp = head2->next->next;

                head2->next = temp;

                lst[j] = head2;

                return;

            }

            head2 = head2->next;

        }

        cout << "Value Not found" << endl;

    }

    void display()

    {

        for (int i = 0; i < 7; i++)

        {

            Node \*head = lst[i];

            cout << i << " - ";

            if (head == nullptr)

            {

                cout << "nullptr" << " ";

            }

            while (head != nullptr)

            {

                cout << "(" << head->data[0] << " " << head->data[1] << " " << head->data[2] << ") ";

                head = head->next;

            }

            cout << endl;

        }

    }

    void prims()

    {

        priority\_queue<vector<int>, vector<vector<int>>, std::greater<vector<int>>> pq;

        vector<int> visited(7, 0); // Assuming 7 vertices: 0 through 6

        vector<vector<int>> path;

        Node \*head;

        int sum = 0;

        pq.push({0, 0, 0}); // cost, from, to

        while (!pq.empty())

        {

            vector<int> top = pq.top();

            pq.pop();

            int cost = top[0];

            int from = top[1];

            int to = top[2];

            if (visited[to] == 0)

            {

                visited[to] = 1;

                sum += cost;

                if (from != to) // skip the initial self-edge

                    path.push\_back({from, to});

                head = lst[to];

                while (head != nullptr)

                {

                    pq.push({head->data[2], to, head->data[1]}); // cost, from=to, to=neighbor

                    head = head->next;

                }

            }

        }

        for (const auto &edge : path)

        {

            cout << "(" << edge[0] << "," << edge[1] << ") ";

        }

        cout << endl;

        cout << "The sum: " << sum << endl;

    }

};

int main()

{

    adj\_list al;

    int choice, i, j, wght, exit = 1;

    while (exit == 1)

    {

        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Prims\n6. Exit" << endl;

        cin >> choice;

        switch (choice)

        {

        case (1):

            cout << "Enter the vertex values: i, j, wght" << endl;

            cin >> i;

            cin >> j;

            cin >> wght;

            al.insert(i, j, wght);

            break;

        case (2):

            cout << "Enter the vertex values to delete: i, j" << endl;

            cin >> i;

            cin >> j;

            al.del(i, j);

            break;

        case (3):

            cout << "Enter the vertex values to search: i, j" << endl;

            cin >> i;

            cin >> j;

            if (al.search(i, j))

            {

                cout << "Available" << endl;

            }

            else

            {

                cout << "No" << endl;

            }

            break;

        case (4):

            al.display();

            break;

        case (5):

            al.prims();

            break;

        case (6):

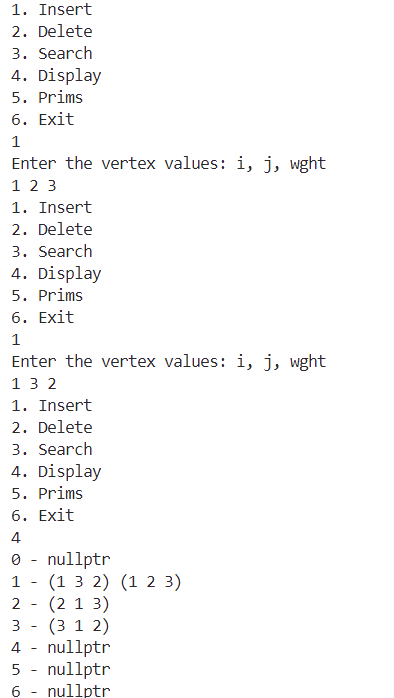
            exit = 0;

        }

    }

}

**Output:**

****

**3.Write a separate C++ menu-driven program to implement Graph ADT with Prim’s, Kruskal’s, and Dijkstra’s algorithms. Maintain proper boundary conditions and follow good coding practices**

**AIM:**

Write a separate C++ menu-driven program to implement **Graph ADT** with **Prim’s**, **Kruskal’s**, and **Dijkstra’s algorithms**. Maintain proper boundary conditions and follow good coding practices.

**Algorithm:**

**Insert Vertex():**

Input → vertex  
if vertex not in graph → add vertex  
else → Output → "Vertex exists"

**Insert Edge():**

Input → vertex1, vertex2, weight  
if both vertices exist →  
 add edge (vertex1, vertex2, weight)  
else → Output → "Invalid vertices"

**Prim’s Algorithm:**

Input → weighted graph, start vertex  
Initialize → visited[], key[], parent[]  
for each vertex v → key[v] = ∞, visited[v] = false  
key[start] = 0, parent[start] = -1  
repeat (V - 1) times →  
 u = vertex with min key and not visited  
 mark u visited  
 for all neighbors v of u →  
  if weight(u, v) < key[v] and v not visited →  
   key[v] = weight(u, v), parent[v] = u  
Output → edges in MST and total cost

**Kruskal’s Algorithm:**

Input → weighted graph  
Initialize → parent[], rank[]  
sort all edges by weight  
for each edge (u, v) in sorted list →  
 if find(u) ≠ find(v) →  
  union(u, v), include (u, v) in MST  
Output → edges in MST and total cost

**Dijkstra’s Algorithm:**

Input → weighted graph, source vertex  
Initialize → dist[], visited[]  
for each vertex v → dist[v] = ∞, visited[v] = false  
dist[source] = 0  
for (V - 1) times →  
 u = vertex with min dist[] and not visited  
 mark u visited  
 for each neighbor v of u →  
  if not visited and dist[u] + weight(u,v) < dist[v] →  
   dist[v] = dist[u] + weight(u,v)  
Output → shortest path distances from source

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <limits>

#include <iomanip>

using namespace std;

// Class to represent a Graph

class Graph {

private:

    int vertices;                   // Number of vertices

    vector<vector<int>> adjMatrix;  // Adjacency matrix

    // Utility function to find the vertex with minimum distance value

    int minDistance(const vector<int>& dist, const vector<bool>& visited) {

        int min = numeric\_limits<int>::max();

        int min\_index = -1;

        for (int v = 0; v < vertices; v++) {

            if (!visited[v] && dist[v] <= min) {

                min = dist[v];

                min\_index = v;

            }

        }

        return min\_index;

    }

    // Utility function for Kruskal's algorithm to find the parent of a node

    int find(vector<int>& parent, int i) {

        if (parent[i] == i)

            return i;

        return find(parent, parent[i]);

    }

    // Utility function for Kruskal's algorithm to perform union of two sets

    void unionSets(vector<int>& parent, vector<int>& rank, int x, int y) {

        int rootX = find(parent, x);

        int rootY = find(parent, y);

        if (rootX == rootY) return;

        if (rank[rootX] < rank[rootY]) {

            parent[rootX] = rootY;

        } else if (rank[rootX] > rank[rootY]) {

            parent[rootY] = rootX;

        } else {

            parent[rootY] = rootX;

            rank[rootX]++;

        }

    }

    // Utility function to display the adjacency matrix

    void displayMatrix() {

        cout << "\nAdjacency Matrix:" << endl;

        cout << "    ";

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

            cout << setw(3) << i;

        }

        cout << endl;

        cout << "   ";

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

            cout << "---";

        }

        cout << endl;

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

            cout << setw(2) << i << " |";

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

                if (adjMatrix[i][j] == numeric\_limits<int>::max()) {

                    cout << setw(3) << "∞";

                } else {

                    cout << setw(3) << adjMatrix[i][j];

                }

            }

            cout << endl;

        }

    }

public:

    // Constructor

    Graph(int v) : vertices(v) {

        // Initialize adjacency matrix with infinite weight (no edge)

        adjMatrix.resize(v, vector<int>(v, numeric\_limits<int>::max()));

        // Set diagonal elements to 0 (distance to self is 0)

        for (int i = 0; i < v; i++) {

            adjMatrix[i][i] = 0;

        }

    }

    // Function to add an edge to the graph

    void addEdge(int u, int v, int weight) {

        if (u < 0 || u >= vertices || v < 0 || v >= vertices) {

            cout << "Invalid vertex!" << endl;

            return;

        }

        adjMatrix[u][v] = weight;

        adjMatrix[v][u] = weight; // For undirected graph

    }

    // Function to remove an edge from the graph

    void removeEdge(int u, int v) {

        if (u < 0 || u >= vertices || v < 0 || v >= vertices) {

            cout << "Invalid vertex!" << endl;

            return;

        }

        adjMatrix[u][v] = numeric\_limits<int>::max();

        adjMatrix[v][u] = numeric\_limits<int>::max(); // For undirected graph

    }

    // Display the graph

    void displayGraph() {

        displayMatrix();

        cout << "\nGraph Edges:" << endl;

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

            for (int j = i+1; j < vertices; j++) {

                if (adjMatrix[i][j] != numeric\_limits<int>::max()) {

                    cout << i << " -- " << j << " : " << adjMatrix[i][j] << endl;

                }

            }

        }

    }

    // Prim's Algorithm for Minimum Spanning Tree

    void primMST() {

        vector<int> parent(vertices);    // Array to store constructed MST

        vector<int> key(vertices);       // Key values used to pick minimum weight edge

        vector<bool> mstSet(vertices);   // To represent set of vertices included in MST

        // Initialize all keys as INFINITE

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

            key[i] = numeric\_limits<int>::max();

            mstSet[i] = false;

        }

        // Always include first vertex in MST

        key[0] = 0;       // Make key 0 so this vertex is picked as first vertex

        parent[0] = -1;   // First node is always root of MST

        // The MST will have vertices-1 edges

        for (int count = 0; count < vertices - 1; count++) {

            // Pick the minimum key vertex from the set of vertices not yet included in MST

            int u = minDistance(key, mstSet);

            // Add the picked vertex to the MST Set

            mstSet[u] = true;

            // Update key value and parent index of the adjacent vertices of the picked vertex

            for (int v = 0; v < vertices; v++) {

                // Update the key only if the vertex is not in MST, there is an edge from u to v,

                // and weight of edge u-v is less than current key value of v

                if (adjMatrix[u][v] != numeric\_limits<int>::max() && !mstSet[v] && adjMatrix[u][v] < key[v]) {

                    parent[v] = u;

                    key[v] = adjMatrix[u][v];

                }

            }

        }

        // Check if MST is connected

        bool isConnected = true;

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

            if (key[i] == numeric\_limits<int>::max()) {

                isConnected = false;

                break;

            }

        }

        if (!isConnected) {

            cout << "\nThe graph is not connected. Cannot form a Minimum Spanning Tree." << endl;

            return;

        }

        // Print the constructed MST

        cout << "\nPrim's Minimum Spanning Tree:" << endl;

        int totalWeight = 0;

        for (int i = 1; i < vertices; i++) {

            cout << parent[i] << " -- " << i << " : " << adjMatrix[i][parent[i]] << endl;

            totalWeight += adjMatrix[i][parent[i]];

        }

        cout << "Total Weight of MST: " << totalWeight << endl;

    }

    // Kruskal's Algorithm for Minimum Spanning Tree

    void kruskalMST() {

        vector<pair<int, pair<int, int>>> edges; // (weight, (u, v))

        // Collect all edges from the adjacency matrix

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

            for (int j = i+1; j < vertices; j++) {

                // If there's an edge between i and j

                if (adjMatrix[i][j] != numeric\_limits<int>::max()) {

                    edges.push\_back({adjMatrix[i][j], {i, j}});

                }

            }

        }

        // Sort edges by weight

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

        vector<int> parent(vertices);

        vector<int> rank(vertices, 0);

        // Create V single-item sets

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

            parent[i] = i;

        }

        cout << "\nKruskal's Minimum Spanning Tree:" << endl;

        int totalWeight = 0;

        int edgeCount = 0;

        for (auto& edge : edges) {

            int u = edge.second.first;

            int v = edge.second.second;

            int weight = edge.first;

            int rootU = find(parent, u);

            int rootV = find(parent, v);

            // If including this edge doesn't cause a cycle

            if (rootU != rootV) {

                cout << u << " -- " << v << " : " << weight << endl;

                totalWeight += weight;

                unionSets(parent, rank, rootU, rootV);

                edgeCount++;

            }

        }

        // Check if MST is complete

        if (edgeCount != vertices - 1) {

            cout << "The graph is not connected. Cannot form a complete Minimum Spanning Tree." << endl;

        } else {

            cout << "Total Weight of MST: " << totalWeight << endl;

        }

    }

    // Dijkstra's Algorithm for Shortest Path

    void dijkstra(int src) {

        if (src < 0 || src >= vertices) {

            cout << "Invalid source vertex!" << endl;

            return;

        }

        vector<int> dist(vertices, numeric\_limits<int>::max());

        vector<bool> visited(vertices, false);

        vector<int> parent(vertices, -1);

        // Distance of source vertex from itself is always 0

        dist[src] = 0;

        // Find shortest path for all vertices

        for (int count = 0; count < vertices - 1; count++) {

            // Pick the minimum distance vertex from the set of vertices not yet processed

            int u = minDistance(dist, visited);

            // If we couldn't find a valid vertex, the graph is disconnected

            if (u == -1) break;

            // Mark the picked vertex as processed

            visited[u] = true;

            // Update dist value of the adjacent vertices of the picked vertex

            for (int v = 0; v < vertices; v++) {

                // Update dist[v] only if it's not in visited, there is an edge from u to v,

                // and total weight of path from src to v through u is smaller than current value of dist[v]

                if (!visited[v] && adjMatrix[u][v] != numeric\_limits<int>::max() &&

                    dist[u] != numeric\_limits<int>::max() && dist[u] + adjMatrix[u][v] < dist[v]) {

                    dist[v] = dist[u] + adjMatrix[u][v];

                    parent[v] = u;

                }

            }

        }

        // Print the constructed distance array

        cout << "\nDijkstra's Shortest Paths from vertex " << src << ":" << endl;

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

            if (i != src) {

                if (dist[i] == numeric\_limits<int>::max()) {

                    cout << "Vertex " << i << ": No path exists" << endl;

                } else {

                    cout << "Vertex " << i << ": Distance = " << dist[i] << ", Path: ";

                    // Print path

                    vector<int> path;

                    int current = i;

                    while (current != -1) {

                        path.push\_back(current);

                        current = parent[current];

                    }

                    // Print path in correct order

                    for (int j = path.size() - 1; j >= 0; j--) {

                        cout << path[j];

                        if (j > 0) cout << " -> ";

                    }

                    cout << endl;

                }

            }

        }

    }

    // Check if the graph is connected

    bool isConnected() {

        if (vertices == 0) return true;

        vector<bool> visited(vertices, false);

        // Start DFS from vertex 0

        queue<int> q;

        q.push(0);

        visited[0] = true;

        while (!q.empty()) {

            int u = q.front();

            q.pop();

            for (int v = 0; v < vertices; v++) {

                if (adjMatrix[u][v] != numeric\_limits<int>::max() && !visited[v]) {

                    visited[v] = true;

                    q.push(v);

                }

            }

        }

        // Check if all vertices are visited

        for (bool v : visited) {

            if (!v) return false;

        }

        return true;

    }

};

int main() {

    int choice, vertices, u, v, weight, src;

    Graph\* graph = nullptr;

    while (true) {

        cout << "\n===== GRAPH ADT MENU =====" << endl;

        cout << "1. Create a Graph" << endl;

        cout << "2. Add Edge" << endl;

        cout << "3. Remove Edge" << endl;

        cout << "4. Display Graph" << endl;

        cout << "5. Prim's MST Algorithm" << endl;

        cout << "6. Kruskal's MST Algorithm" << endl;

        cout << "7. Dijkstra's Shortest Path Algorithm" << endl;

        cout << "8. Exit" << endl;

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                cout << "Enter number of vertices: ";

                cin >> vertices;

                if (vertices <= 0) {

                    cout << "Number of vertices must be positive!" << endl;

                    break;

                }

                // Delete previous graph if exists

                if (graph != nullptr) {

                    delete graph;

                }

                graph = new Graph(vertices);

                cout << "Graph with " << vertices << " vertices created." << endl;

                break;

            case 2:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                cout << "Enter edge (u v weight): ";

                cin >> u >> v >> weight;

                if (weight <= 0) {

                    cout << "Weight must be positive!" << endl;

                    break;

                }

                graph->addEdge(u, v, weight);

                cout << "Edge added: " << u << " -- " << v << " with weight " << weight << endl;

                break;

            case 3:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                cout << "Enter edge to remove (u v): ";

                cin >> u >> v;

                graph->removeEdge(u, v);

                cout << "Edge removed: " << u << " -- " << v << endl;

                break;

            case 4:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                graph->displayGraph();

                break;

            case 5:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                if (!graph->isConnected()) {

                    cout << "The graph is not connected. Prim's algorithm requires a connected graph." << endl;

                    break;

                }

                graph->primMST();

                break;

            case 6:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                graph->kruskalMST();

                break;

            case 7:

                if (graph == nullptr) {

                    cout << "Please create a graph first!" << endl;

                    break;

                }

                cout << "Enter source vertex: ";

                cin >> src;

                graph->dijkstra(src);

                break;

            case 8:

                cout << "Exiting program..." << endl;

                if (graph != nullptr) {

                    delete graph;

                }

                return 0;

            default:

                cout << "Invalid choice! Please try again." << endl;

        }

    }

    return 0;

}