SHIV NADAR UNIVERSITY
CHENNAI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING

LABORATORY RECORD

B.TECH
(YEAR : 20   - 20   )

NAME : ....

REG. NO. : ....

DEPT. : .... SEM. : .... CLASS & SEC : ....

SHIV NADAR
UNIVERSITY
CHENNAI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING

LABORATORY RECORD

B.TECH
(YEAR : 20 - 20)

NAME	: ....
REG. NO.	: ....
DEPT.	: .... **SEM. :** .... **CLASS & SEC :** ....

SHIV NADAR UNIVERSITY
CHENNAI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING

LABORATORY RECORD

B.TECH
(YEAR : 20 - 20)

NAME	: ....
REG. NO.	: ....
DEPT.	: .... **SEM. :** .... **CLASS & SEC :** ....

Contents

1 Searching and Sorting

1.1 Sort Ascending

1.1.1 Question

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

1.1.2 Algorithm

Algorithm 1 - BubbleSort

Input

1. arr[] - array of integers
2. num - number of elements in array

Output

- sorted array in ascending order

Steps

```
1. For $i$ from 0 to $num - 2$:
    1. For $j$ from 0 to $num - 2 - i$:
        1. If $arr[j] > arr[j + 1]$:
            1. $temp \leftarrow arr[j]$
            2. $arr[j] \leftarrow arr[j + 1]$
            3. $arr[j + 1] \leftarrow temp$
```

Algorithm 2 - InsertionSort

Input

1. arr[] - array of integers
2. num - number of elements in array

Output

- sorted array in ascending order

Steps

```
1. For $i$ from 1 to $num - 1$:
    1. $temp \leftarrow arr[i]$
    2. $j \leftarrow i - 1$
    3. While $j \geq 0$ and $arr[j] > temp$:
        1. $arr[j + 1] \leftarrow arr[j]$
        2. $j \leftarrow j - 1$
    4. $arr[j + 1] \leftarrow temp$
```

Algorithm 3 - SelectionSort

Input

1. arr[] - array of integers
2. num - number of elements in array

Output

- sorted array in ascending order

Steps

```
1. For $i$ from 0 to $num - 1$:
    1. $k \leftarrow i$
    2. $temp \leftarrow arr[i]$
    3. For $j$ from $i + 1$ to $num - 1$:
        1. If $arr[j] < temp$:
            1. $temp \leftarrow arr[j]$
            2. $k \leftarrow j$
    4. $arr[k] \leftarrow arr[i]$
    5. $arr[i] \leftarrow temp$
```

Algorithm 4 - Display

Input

1. arr[] - array of integers
2. num - number of elements in array

Output

- printed array elements

Steps

```
1. For $i$ from 0 to $num - 1$:
    1. Print $arr[i]$ followed by space
2. Print newline
```

1.1.3 Code

main.cpp

```cpp
/*
1. Write a C++ menu-driven program to sort a given array in ascending order. Design proper f

a. Bubble Sort
b. Selection Sort
c. Insertion Sort
d. Exit

*/


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

//function prototypes
void display(const vector<int>& arr);
void bubblesort(vector<int>& arr);
void selectionsort(vector<int>& arr);
void insertionsort(vector<int>& arr);
void ExitMenu();


int main(){
    vector<int> arr;
    int size;
    int choice;

    cout << "Enter the size of the array:";
    cin >> size;
    for(int i = 0; i < size; i++){
        int element;
        cout << "enter element no " << i + 1 << ":" << endl;
        cin >> element;
        arr.push_back(element);
    }
    while(1){
        //defining the menu
        cout << "<===== MENU =====>" << endl;
        cout << "1.Bubblesort the array" << endl;
        cout << "2.Selection sort the array" << endl;
        cout << "3.Insertion sort the array" << endl;
        cout << "4.Exit Menu" << endl;
        cout << "\n";
        cout << "select your options in menu within 1 to 4:" << endl;
        cin >> choice;
        cout << endl;
        switch(choice){
            case 1:
                bubblesort(arr);
                break;
            case 2:
                selectionsort(arr);
                break;
            case 3:
                insertionsort(arr);
                break;
            case 4:
                ExitMenu();
                return 0;
            default:
                cout << "Selected option not in the menu!!\nPLEASE try again!\n\n" << endl;
        }
    }

}

//defining the functions

void display(const vector<int>& arr){

    cout << "SORTED ARRAY: | ";
    for(int num : arr){
        cout << num << " | ";
    }
    cout << endl;
}

void bubblesort(vector<int>& arr){
    int n = arr.size();
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < n - i - 1; j++){
            if(arr[j] > arr[j + 1]){
                int temp;
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    display(arr);
}

void selectionsort(vector<int>& arr){
    int n = arr.size();
    for(int i = 0; i < n - 1; i++){
        int minindex = i;
        for(int j = i + 1; j < n; j++){
            if(arr[j] < arr[minindex]){
                minindex = j;
            }
        }
        int temp;
        temp = arr[i];
        arr[i] = arr[minindex];
        arr[minindex] = temp;
    }

    display(arr);
}

void insertionsort(vector<int>& arr){
    int n = arr.size();
    for(int i = 1; i < n; i++){
        int key = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }

    display(arr);
}
```

```
void ExitMenu(){
    cout << "Menu Exited!!" << endl;
}
```

1.2 Search

1.2.1 Question

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

1.2.2 Algorithm

Algorithm 1 - LinearSearch

Input

1. arr[] - array of integers
2. num - number of elements in the array
3. temp - value to search for

Output

- 1 if temp is found in arr[], 0 otherwise

Steps

```
1. For $i$ from 0 to $num - 1$:
    1. If $arr[i] = temp$:
        1. Return 1
2. Return 0
```

Algorithm 2 - BinarySearch

Input

1. arr[] - array of integers
2. num - number of elements in the array

3. temp - value to search for

Output

- 1 if temp is found in sorted arr[], 0 otherwise

Steps

```
1. Sort $arr[]$ using bubble sort
2. $start \leftarrow 0$
3. $end \leftarrow num - 1$
4. While $start \leq end$:
    1. $mid \leftarrow \lfloor (start + end)/2 \rfloor$
    2. If $arr[mid] = temp$:
        1. Return 1
    3. If $arr[mid] < temp$:
        1. $start \leftarrow mid + 1$
    4. If $arr[mid] > temp$:
        1. $end \leftarrow mid - 1$
5. Return 0
```

1.2.3 Code

main.cpp

```
//program to search a element in an array
#include "head.h"
#include <cstdio>


void linearsearch(int arr[], int num, int num2);
int binarySearch(int arr[], int num, int key);

int main() {
    int num, element, num2, choice;

    printf("Enter the number of elements: ");
    scanf("%d", &num);

    int arr[num], arr2[num];
    printf("Enter the elements: ");
    for (int i = 0; i < num; i++) {
        scanf("%d", &element);
        arr[i] = element;
        arr2[i] = element;
    }
    printf("Enter the element you want to search: ");
    scanf("%d", &num2);

    while (1) {
        printf("\n           MENU\n");
        printf("1. Linear Search\n");
        printf("2. Binary Search\n");
        printf("3. Display Array\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                linearsearch(arr, num, num2);
                break;
            case 2: {
                int position = binarySearch(arr, num, num2);
                if (position != -1) {
                    printf("The element %d is FOUND at position %d\n", num2, position+1);
                } else {
                    printf("The element %d is NOT FOUND\n", num2);
                }
                break;
            }
            case 3:
                 printf("The Sorted Array is : ");
                    for (int i = 0; i<num; i++)
                        {
                            printf("%d  ",arr2[i]);
                        }
                break;
            case 4:
                return 0;
            default:
                printf("INVALID CHOICE. Please try again.\n");
        }
    }
}

void linearsearch(int arr[], int num, int num2) {
    int found = 0;
    for (int i = 0; i < num; i++) {
        if (num2 == arr[i]) {
            printf("The element %d is FOUND at position %d\n", num2, i+1);
```

```
                found = 1;
                break;
            }
        }
        if (!found) {
            printf("The element %d is NOT FOUND\n", num2);
        }
}

int binarySearch(int arr[], int num, int key) {
    Bubblesort(arr, num);
    int left = 0, right = num - 1;

    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == key) {
            return mid;
        }
        if (arr[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

head.h

```


void Bubblesort(int arr2[],int num){

    for (int i = 0; i< num; i++)
    {
        for (int j = 0; j < (num-1); j++)
        {
            if(arr2[j]>arr2[j+1]){
                int temp = arr2[j+1];
                arr2[j+1] = arr2[j];
                arr2[j] = temp;

            }
        }

    }
}

void Selectionsort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min]) {
                min = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
void Insertionsort(int arr[], int N) {

    for (int i = 1; i < N; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

2 Classes with Pointers

2.1 Number Operations

2.1.1 Question

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

2.1.2 Algorithm

Algorithm 1 - CheckPalindrome

Input

1. num - pointer to the number to check

Output

- Message indicating whether the number is a palindrome

Steps

```
1. Allocate memory for $sum$, $rem$, and $temp$ pointers
2. $*sum \leftarrow 0$
3. $*temp \leftarrow *num$
4. While $*temp > 0$:
    1. $*rem \leftarrow (*temp) \bmod 10$
    2. $*sum \leftarrow (*sum) \times 10 + *rem$
    3. $*temp \leftarrow *temp/10$
5. If $*sum = *num$:
    1. Print “It is a palindrome”
6. Else:
    1. Print “It is not a palindrome”
7. Free allocated memory
```

Algorithm 2 - CountDigits

Input

1. num - pointer to the number to count digits of

Output

- Number of digits in the number

Steps

```
1. Allocate memory for temp and len pointers
2. $*temp \leftarrow *num$
3. $*len \leftarrow 0$
4. While $*temp > 0$:
    1. $*temp \leftarrow *temp/10$
    2. $*len \leftarrow *len + 1$
5. Return $*len$
6. Free allocated memory
```

Algorithm 3 - CheckArmstrong

Input

1. num - pointer to the number to check

Output

- Message indicating whether the number is an Armstrong number

Steps

```
1. Allocate memory for sum, rem, temp, and len pointers
2. $*sum \leftarrow 0$
3. $*temp \leftarrow *num$
4. $*len \leftarrow count(num)$
5. While $*temp > 0$:
    1. $*rem \leftarrow *temp \bmod 10$
    2. $*sum \leftarrow *sum + pow(*rem, *len)$
    3. $*temp \leftarrow *temp/10$
6. If $*sum = *num$:
    1. Print “It is an Armstrong number”
7. Else:
    1. Print “It is not an Armstrong number”
8. Free allocated memory
```

Algorithm 4 - CheckPerfect

Input

1. num - pointer to the number to check

Output

- Message indicating whether the number is a perfect number

Steps

```
1. Allocate memory for $sum$ and $i$ pointers
2. $*sum \leftarrow 0$
3. For $*i$ from 1 to $(*num) - 1$:
    1. If $(*num) \bmod (*i) = 0$:
        1. $(*sum) \leftarrow (*sum) + (*i)$
4. If $*sum = *num$:
    1. Print “It is a perfect number”
5. Else:
    1. Print “It is not a perfect number”
6. Free allocated memory
```

2.1.3 Code

main.cpp

```cpp
/*Write a C++ menu-driven program to determine whether a number is a Palindrome, Armstrong,

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

files to be submitted in pointer cpp file*/

#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

// Function prototypes
void check_Palindrome(int* num);
void check_Armstrong(int* num);
void check_Perfect(int* num);
int count_digits(int* num);

int main() {
    // Dynamically allocating memory for number input
    int* number = (int*)malloc(sizeof(int));
    if (!number) {
        cout << "Memory allocation failed!" << endl;
        return 1;
    }

    cout << "Enter the number: ";
    cin >> *number;

    int* selection = (int*)malloc(sizeof(int));

    if (!selection) {
        cout << "Memory allocation failed!" << endl;
        free(number);
        return 1;
    }

    while (true) {
        cout << "\n<===== MENU =====>\n";
        cout << "1. Palindrome\n";
        cout << "2. Armstrong Number\n";
        cout << "3. Perfect Number\n";
        cout << "4. Exit\n";
        cout << "Select a choice: ";
        cin >> *selection;
        cout << endl;

        switch (*selection) {
            case 1:
                check_Palindrome(number);
                break;
            case 2:
                check_Armstrong(number);
                break;
            case 3:
                check_Perfect(number);
                break;
            case 4:
                cout << "Exiting program.\n";
                free(number);
                free(selection);
                return 0;
            default:
                cout << "Selected option does not exist!!\nPlease Try Again\n";
        }
    }
}
```

```
// Function to check if a number is a palindrome
void check_Palindrome(int* num) {
    int* original = (int*)malloc(sizeof(int));
    int* reversed_num = (int*)malloc(sizeof(int));
    int* digit = (int*)malloc(sizeof(int));

    if (!original || !reversed_num || !digit) {
        cout << "Memory allocation failed!" << endl;
        return;
    }

    *original = *num;
    *reversed_num = 0;

    while (*num > 0) {
        *digit = (*num) % 10;
        *reversed_num = (*reversed_num) * 10 + (*digit);
        *num /= 10;
    }

    if (*reversed_num == *original) {
        cout << *original << " is a Palindrome" << endl;
    } else {
        cout << *original << " is not a Palindrome" << endl;
    }

    free(original);
    free(reversed_num);
    free(digit);
}

// Function to check if a number is an Armstrong number
void check_Armstrong(int* num) {
    int* original = (int*)malloc(sizeof(int));
    int* sum = (int*)malloc(sizeof(int));
    int* digit = (int*)malloc(sizeof(int));

    if (!original || !sum || !digit) {
        cout << "Memory allocation failed!" << endl;
        return;
    }

    *original = *num;
    *sum = 0;
    int* count = (int*)malloc(sizeof(int));
    *count = count_digits(num);

    int* temp = (int*)malloc(sizeof(int));
    *temp = *num;

    while (*temp > 0) {
        *digit = *temp % 10;
        *sum += pow(*digit, *count);
        *temp /= 10;
    }

    if (*sum == *original) {
        cout << *original << " is an Armstrong number" << endl;
    } else {
        cout << *original << " is not an Armstrong number" << endl;
    }

    free(original);
    free(sum);
    free(digit);
    free(count);
    free(temp);
}

// Function to check if a number is a perfect number
void check_Perfect(int* num) {
    int* sum = (int*)malloc(sizeof(int));
    int* i = (int*)malloc(sizeof(int));

    if (!sum || !i) {
        cout << "Memory allocation failed!" << endl;
        return;
    }

    *sum = 0;
    *i = 1;

    while (*i < *num) {
        if (*num % *i == 0) {
            *sum += *i;
        }
        (*i)++;
    }

    if (*sum == *num) {
        cout << *num << " is a Perfect number" << endl;
    } else {
        cout << *num << " is not a Perfect number" << endl;
    }

    free(sum);
    free(i);
}

// Function to count the number of digits in a number
int count_digits(int* num) {
    int* count = (int*)malloc(sizeof(int));
    int* temp = (int*)malloc(sizeof(int));

    if (!count || !temp) {
        cout << "Memory allocation failed!" << endl;
        return 0;
    }

    *count = 0;
    *temp = *num;

    while (*temp > 0) {
        (*count)++;
        *temp /= 10;
    }

    int result = *count;
    free(count);
    free(temp);
    return result;
}
```

2.2 Shape Area

2.2.1 Question

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

2.2.2 Algorithm

Algorithm 1 - CalculateSquareArea

Input

1. len - side length of square

Output

- area - area of the square

Steps

```
1. $area \leftarrow len \times len$
2. Return $area$
```

Algorithm 2 - CalculateRectangleArea

Input

1. len - length of rectangle
2. bre - breadth of rectangle

Output

- area - area of the rectangle

Steps

```
1. $area \leftarrow len \times bre$
2. Return $area$
```

Algorithm 3 - CalculateCubeArea

Input

1. len - side length of cube

Output

- area - surface area of the cube

Steps

```
1. $area \leftarrow 6 \times len \times len$
2. Return $area$
```

Algorithm 4 - CalculateCuboidArea

Input

1. len - length of cuboid
2. bre - breadth of cuboid
3. hei - height of cuboid

Output

- area - surface area of the cuboid

Steps

1. $area \leftarrow 2 \times (len \times bre + bre \times hei + hei \times len)$
2. Return $area$

2.2.3 Code

main.cpp

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


class cuboid{
    private:
        int le,br,he;

    public:
        void third_set_parameters(int length, int breadth, int height);
        int third_area();
};

class cube{
    private:
        int s;
    public:
        void two_set_parameters(int side);
        int two_area();
};

class rectangle{
    private:
        int len,bre;
    public:
        void four_set_parameters(int length, int breadth);
        int four_area();
```

```
};

class square{
    private:
        int s;
    public:
        void one_set_parameters(int side);
        int one_area();
};

void cuboid_area();
void cube_area();
void rectangle_area();
void square_area();
void exit_menu();

int main(){
    int choice;
    //menu driven program
    while(1){
        cout << "\n<===== MENU ======>" << endl;
        cout << "1.SQUARE" << endl;
        cout << "2.CUBE" << endl;
        cout << "3.RECTANGLE" << endl;
        cout << "4.CUBOID" << endl;
        cout << "5.EXIT" << endl;
        cout << "select your choice:" << endl;
        cin >> choice;
        cout << endl;
        switch(choice){
            case 1:
                square_area();
                break;
            case 2:
                cube_area();
                break;
            case 3:
                rectangle_area();
                break;
            case 4:
                cuboid_area();
                break;
            case 5:
                exit_menu();
                return 0;
            default:
                cout << "The selected option cease to exist!!\nTRY Again!!" << endl;
            }
        }
    }

//setting parameters for cuboid
void cuboid::third_set_parameters(int length, int breadth, int height){
    le = length;
    br = breadth;
    he = height;
}

//setting parameters for cube
void cube::two_set_parameters(int side){
    s = side;
}

//setting parameters for rectangle
void rectangle::four_set_parameters(int length, int breadth){
    len = length;
    bre = breadth;
}

//setting parameters for square
void square::one_set_parameters(int side){
    s = side;
}


//defining functions to calculate areas:


int cuboid::third_area(){
    int area = 2*(le*br + le*he + he*br);
    return area;
}

int cube::two_area(){
    int area = (6*(s*s));
    return area;
}

int rectangle::four_area(){
    int area = (len * bre);
    return area;
}
```

```
int square::one_area(){
    int area = s*s ;
    return area;
}



//defining the functions to print output:

void cuboid_area(){
    int length, breadth, height;
    cout << "Enter the length";
    cin >> length;
    cout << "ENter the breadth:";
    cin >> breadth;
    cout << "Enter the height: ";
    cin >>  height;
    cuboid one;
    one.third_set_parameters(length,breadth, height);
    cout << "The area of the cuboid is " << one.third_area() << endl;
}

void cube_area(){
    int side;
    cout << "Enter the side of the cube:";
    cin >> side;
    cube cyber1;
    cyber1.two_set_parameters(side);
    cout << "The area of the cube is  " << cyber1.two_area() << endl;
}

void rectangle_area(){
    int length, breadth;
    cout << "Enter the length :" ;
    cin >> length;
    cout << "Enter the breadth: " ;
    cin >> breadth;
    rectangle cyber;
    cyber.four_set_parameters(length, breadth);
    cout << "Area of the rectangle = " << cyber.four_area() << endl;
}

void square_area(){
    int side;
    cout << "Enter the side:";
    cin >> side;
    square cyber;
    cyber.one_set_parameters(side);
    cout << "The area of the square = " << cyber.one_area() << endl;
}

//function to exit

void exit_menu(){
    cout << "MENU EXITED!!\n" << endl;
}
```

3 List ADT with Arrays

3.1 List ADT Implementation

3.1.1 Question

Write a C++ menu-driven program to implement List ADT using an array of size 5. 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. Rotate
10. Exit

The rotate option takes an input 'k' which rotates the entire array to the right by k times. Think of at least 3 solutions. Think of a solution that rotates using O(1) extra space.

3.1.2 Algorithm

Algorithm 1 - ArrayListInsertBeginning

Input

1. num - value to insert
2. arr[] - array storing the list elements
3. cur - current number of elements

Output

- Updated array list with num inserted at beginning

Steps

```
1. If $cur = 5$ (list is full):
    1. Print "List is full"
    2. Return
2. For $i$ from $cur$ down to 1:
    1. $arr[i] \leftarrow arr[i-1]$
3. $arr[0] \leftarrow num$
4. $cur \leftarrow cur + 1$
```

Algorithm 2 - ArrayListInsertEnd

Input

1. num - value to insert
2. arr[] - array storing the list elements
3. cur - current number of elements

Output

- Updated array list with num inserted at end

Steps

```
1. If $cur = 5$ (list is full):
    1. Print “List is full”
    2. Return
2. $arr[cur] \leftarrow num$
3. $cur \leftarrow cur + 1$
```

Algorithm 3 - ArrayListInsertPosition

Input

1. num - value to insert
2. pos - position to insert at
3. arr[] - array storing the list elements
4. cur - current number of elements

Output

- Updated array list with num inserted at position pos

Steps

```
1. If $cur = 5$ (list is full):
    1. Print “List is full”
    2. Return
2. If $pos < 0$ or $pos > cur$:
    1. Print “Invalid position”
    2. Return
3. For $i$ from $cur$ down to $pos + 1$:
    1. $arr[i] \leftarrow arr[i - 1]$
4. $arr[pos] \leftarrow num$
5. $cur \leftarrow cur + 1$
```

Algorithm 4 - ArrayListDeleteBeginning

Input

1. arr[] - array storing the list elements
2. cur - current number of elements

Output

- Updated array list with first element removed

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. For $i$ from 0 to $cur - 2$:
    1. $arr[i] \leftarrow arr[i + 1]$
3. $cur \leftarrow cur - 1$
```

Algorithm 5 - ArrayListDeleteEnd

Input

1. arr[] - array storing the list elements
2. cur - current number of elements

Output

- Updated array list with last element removed

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. $cur \leftarrow cur - 1$
```

Algorithm 6 - ArrayListDeletePosition

Input

1. pos - position to delete
2. arr[] - array storing the list elements
3. cur - current number of elements

Output

- Updated array list with element at position pos removed

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. If $pos < 0$ or $pos \geq cur$:
    1. Print "Invalid position"
    2. Return
3. For $i$ from $pos$ to $cur - 2$:
    1. $arr[i] \leftarrow arr[i + 1]$
4. $cur \leftarrow cur - 1$
```

Algorithm 7 - ArrayListSearch

Input

1. num - value to search for
2. arr[] - array storing the list elements
3. cur - current number of elements

Output

- Status message indicating if num was found or not

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. For $i$ from 0 to $cur - 1$:
    1. If $arr[i] = num$:
        1. Print "Element found"
        2. Return
3. Print "Element not found"
```

Algorithm 8 - ArrayListDisplay

Input

1. arr[] - array storing the list elements
2. cur - current number of elements

Output

- All elements in the list printed

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. For $i$ from 0 to $cur - 1$:
    1. Print $arr[i]$ followed by space
3. Print newline
```

Algorithm 9 - ArrayListRotate

Input

1. k - number of rotations
2. arr[] - array storing the list elements
3. cur - current number of elements

Output

- Updated array list rotated k times

Steps

```
1. If $cur = 0$ (list is empty):
    1. Print "List is empty"
    2. Return
2. $k \leftarrow k \bmod cur$
3. For $i$ from 0 to $k - 1$:
    1. $temp \leftarrow arr[cur - 1]$
    2. For $j$ from $cur - 1$ down to 1:
        1. $arr[j] \leftarrow arr[j - 1]$
    3. $arr[0] \leftarrow temp$
```

3.1.3 Code

main.cpp

```
//Implementation of List ADT using an array of size 5
#include <cstdio>
#include <cstdlib>

class LIST {
private:
    int arr[5];
    int current;

public:
    LIST() { current = -1; }

    void display();
    void insbeg();
    void inspos(int);
    void insend();
    void delbeg();
    void delpos(int);
    void delend();
    int search(int);
    void rotate_v1(int);
    void rotate_v2(int);
    void rotate_v3(int);
};

int main() {
    int choice;
    LIST arr;

    while (true) {
        printf("\n           MENU\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete at Position\n");
        printf("7. Search\n");
        printf("8. Display\n");
        printf("9. Rotate\n");
        printf("10. Exit\n");
        printf("\nENTER YOUR CHOICE: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                arr.insbeg();
                break;
            case 2:
                arr.insend();
                break;
            case 3: {
                int pos;
                printf("Enter the Position: ");
                scanf("%d", &pos);
                arr.inspos(pos);
                break;
            }
            case 4:
                arr.delbeg();
                break;
            case 5:
                arr.delend();
                break;
            case 6: {
                int pos;
                printf("Enter the Position: ");
                scanf("%d", &pos);
                arr.delpos(pos);
                break;
            }
            case 7: {
                int num;
                printf("Enter the Element: ");
                scanf("%d", &num);
                arr.search(num);
                break;
            }
            case 8:
                arr.display();
                break;
            case 9: {
                int rotations, choi;
                printf("Enter Number of Rotations: ");
                scanf("%d", &rotations);
                printf("If You Want to Do it in 3 different Methods 1 enter 1, Method 2 ente
                scanf("%d", &choi);
                if (choi == 1) {
                    arr.rotate_v1(rotations);
                } else if (choi == 2) {
                    arr.rotate_v2(rotations);
                } else if (choi == 3) {
                    arr.rotate_v3(rotations);
                } else {
                    printf("Invalid");
                }
                break;
            }
            case 10:
                return 0;
            default:
                printf("Invalid Choice! Try again.\n");
        }
    }
}

void LIST::display() {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }
    printf("\nThe List is: ");
    for (int i = 0; i <= current; i++) { // Loop to print elements
        printf("%d  ", arr[i]);
    }
    printf("\n");
}

void LIST::insbeg() {
    if (current == 4) {
        printf("The List is Full\n");
        return;
    }

    int temp;
    printf("Enter The Element: ");
    scanf("%d", &temp);

    // Shifting elements to the right to make space at the beginning
    for (int i = current; i >= 0; i--) {
        arr[i + 1] = arr[i];
    }

    arr[0] = temp;
    current++;
}

void LIST::inspos(int pos) {
    if (pos < 0 || pos > current + 1) {
        printf("Invalid Position!\n");
        return;
    }
    if (current == 4) {
        printf("The List is Full\n");
        return;
    }

    int temp;
    printf("Enter The Element: ");
    scanf("%d", &temp);

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

    arr[pos] = temp;
    current++;
}

void LIST::insend() {
    if (current == 4) {
        printf("The List is Full\n");
        return;
    }

    int temp;
    printf("Enter The Element: ");
    scanf("%d", &temp);

    arr[current + 1] = temp;
    current++;
}

void LIST::delbeg() {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }
    for (int i = 0; i < current; i++) {
        arr[i] = arr[i + 1];
    }

    current--;
}

void LIST::delpos(int pos) {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }
    if (pos < 0 || pos > current) {
        printf("Invalid Position!\n");
        return;
    }

    // Shift elements left to remove element at given position
    for (int i = pos; i < current; i++) {
        arr[i] = arr[i + 1];
    }

    current--;
}

void LIST::delend() {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }

    current--;
}

int LIST::search(int num) {
    for (int i = 0; i <= current; i++) { // Loop to search element
        if (arr[i] == num) {
            printf("The Element '%d' is found at Position '%d'\n", num, i);
            return 1;
        }
    }

    printf("Element not found in the list.\n");
    return 0;
}

void LIST::rotate_v1(int num) {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }
    num = num % (current + 1); // Prevent unnecessary rotations
    if (num == 0) return;

    // Rotate right 'num' times
    for (int r = 0; r < num; r++) {
        int temp = arr[current];
        for (int i = current; i > 0; i--) {
            arr[i] = arr[i - 1];
        }
        arr[0] = temp;
    }
}

void LIST::rotate_v2(int num) {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }

    num = num % (current + 1);
    if (num == 0) return;

    int temp[5];

    // Compute new positions and store in temp
    for (int i = 0; i <= current; i++) {
        temp[(i + num) % (current + 1)] = arr[i];
    }

    // Copy temp back to original array
    for (int i = 0; i <= current; i++) {
        arr[i] = temp[i];
    }
}

void LIST::rotate_v3(int num) {
    if (current == -1) {
        printf("The List is Empty\n");
        return;
    }
    num = num % (current + 1);
    if (num == 0) return;

    // Rotate left 'num' times
    for (int i = 0; i < num; i++) {
        int temp = arr[0];
        for (int j = 0; j < current; j++) {
            arr[j] = arr[j + 1];
        }
        arr[current] = temp;
    }
}
```

4 List ADT with Linked Lists

4.1 Reversing

4.1.1 Question

Write a C++ menu-driven program to implement List ADT using a singly linked list. Maintain proper boundary conditions and folloy 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
7. Display
8. Display Reverse
9. Reverse Link
10. Exit

4.1.2 Algorithm

Algorithm 1 - Insert_Beginning

Input

1. num - value to insert
2. head - pointer to head of linked list

Output

- Updated linked list with num inserted at the beginning

Steps

```
1. Create new node $newnode$ with $data = num$
2. $newnode.next \leftarrow head$
3. $head \leftarrow newnode$
```

Algorithm 2 - Insert_End

Input

1. num - value to insert
2. head - pointer to head of linked list

Output

- Updated linked list with num inserted at the end

Steps

```
1. If $head = null$ (list is empty):
    1. Call $insert\_beg(num)$
2. Else:
    1. Create new node $newnode$ with $data = num$ and $next = null$
    2. $temp \leftarrow head$
    3. While $temp.next \neq null$:
        1. $temp \leftarrow temp.next$
    4. $temp.next \leftarrow newnode$
```

Algorithm 3 - Insert_Position

Input

1. num - value to insert
2. pos - position to insert at
3. head - pointer to head of linked list

Output

- Updated linked list with num inserted at position pos

Steps

```
1. Create new node $newnode$ with $data = num$
2. If $pos = 1$:
    1. Call $insert\_beg(num)$
3. Else if $pos = count() + 1$:
    1. Call $insert\_end(num)$
4. Else:
    1. $temp \leftarrow head$
    2. For $i$ from 1 to $pos - 2$:
        1. $temp \leftarrow temp.next$
    3. $newnode.next \leftarrow temp.next$
    4. $temp.next \leftarrow newnode$
```

Algorithm 4 - Delete_Beginning

Input

1. head - pointer to head of linked list

Output

- Updated linked list with first node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print “List is empty”
2. Else:
    1. $temp \leftarrow head.next$
    2. Free $head$
    3. $head \leftarrow temp$
```

Algorithm 5 - Delete_End

Input

1. head - pointer to head of linked list

Output

- Updated linked list with last node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print “List is empty”
2. Else:
    1. $temp \leftarrow head$
    2. If $temp.next = null$ (only one node):
        1. Call $delete\_beg()$
    3. Else:
        1. While $temp.next.next \neq null$:
            1. $temp \leftarrow temp.next$
        2. Free $temp.next$
        3. $temp.next \leftarrow null$
```

Algorithm 6 - Delete_Position

Input

1. pos - position to delete
2. head - pointer to head of linked list

Output

- Updated linked list with node at position pos removed

Steps

```
1. If $pos = 1$:
    1. Call $delete\_beg()$
2. Else if $pos = count()$:
    1. Call $delete\_end()$
3. Else:
    1. $temp \leftarrow head$
    2. For $i$ from 1 to $pos - 1$:
        1. $temp \leftarrow temp.next$
    3. $temp2 \leftarrow temp.next.next$
    4. Free $temp.next$
    5. $temp.next \leftarrow temp2$
```

Algorithm 7 - Display

Input

1. head - pointer to head of linked list

Output

- Elements of the linked list printed

Steps

```
1. $temp \leftarrow head$
2. If $head = null$ (list is empty):
    1. Print "List is empty"
3. Else:
    1. While $temp \neq null$:
        1. Print $temp.data$ followed by " ->"
        2. $temp \leftarrow temp.next$
    2. Print newline
```

Algorithm 8 - Search

Input

1. num - value to search for
2. head - pointer to head of linked list

Output

- Status message indicating if num was found or not

Steps

```
1. $temp \leftarrow head$
2. If $head = null$ (list is empty):
    1. Print “List is empty”
3. Else:
    1. While $temp \neq null$:
        1. If $temp.data = num$:
            1. Print “Element found”
            2. Return
        2. $temp \leftarrow temp.next$
    2. Print “Element not found”
```

Algorithm 9 - Count

Input

1. head - pointer to head of linked list

Output

- Number of nodes in the linked list

Steps

```
1. $temp \leftarrow head$
2. $c \leftarrow 0$
3. While $temp \neq null$:
    1. $c \leftarrow c + 1$
    2. $temp \leftarrow temp.next$
4. Return $c$
```

Algorithm 10 - ReverseLink

Input

1. head - pointer to head of linked list

Output

- Linked list with all links reversed

Steps

```
1. $temp \leftarrow head$
2. $len \leftarrow count()$ (Count number of nodes)
3. Create array $arr$ of size $len$ to store all node pointers
4. For $i$ from 0 to $len - 1$:
        1. $arr[i] \leftarrow temp$
        2. $temp \leftarrow temp.next$
    5. Create array $arr2$ of size $len$ with reversed pointers
    6. For $i$ from 0 to $len - 1$:
        1. $arr2[i] \leftarrow arr[len - 1 - i]$
    7. $head \leftarrow arr2[0]$
    8. $temp \leftarrow head$
    9. For $i$ from 1 to $len - 1$:
        1. $temp.next \leftarrow arr2[i]$
        2. $temp \leftarrow temp.next$
    10. $temp.next \leftarrow null$
```

Algorithm 11 - ReverseDisplay

Input

1. head - pointer to head of linked list

Output

- Elements of the linked list displayed in reverse order

Steps

```
1. $len \leftarrow count()$ (Count number of nodes)
2. Create array $arr$ of size $len$ to store node values
3. If $head = null$ (list is empty):
    1. Print "List is empty"
4. Else:
    1. $temp \leftarrow head$
    2. $i \leftarrow 0$
    3. While $temp \neq null$:
        1. $arr[len - 1 - i] \leftarrow temp.data$
        2. $temp \leftarrow temp.next$
        3. $i \leftarrow i + 1$
    4. For $j$ from 0 to $len - 1$:
        1. Print $arr[j]$ followed by space
```

4.1.3 Code

main.cpp

4.2 Ascending Insert and Merging

4.2.1 Question

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

Option 1 inserts a node so the list is always in ascending order. Option 2 takes two lists as input, and merges two lists into a third list. The third list should also be in ascending order. Convert the file into a header file and include it in a C++ file. The second C++ consists of 3 lists and has the following operations,

1. Insert List1
2. Insert List2
3. Merge into List3
4. Display
5. Exit

4.2.2 Algorithm

Algorithm 1 - InsertAscending

Input

1. num - value to insert
2. head - pointer to head of linked list

Output

- Updated linked list with num inserted in ascending order

Steps

```
1. Create new node $newnode$ with $data = num$ and $next = null$
2. If $head = null$ (list is empty) or $num < head.data$:
    1. $newnode.next \leftarrow head$
    2. $head \leftarrow newnode$
    3. Return
3. $temp \leftarrow head$
4. While $temp.next \neq null$ and $temp.next.data < num$:
    1. $temp \leftarrow temp.next$
5. $newnode.next \leftarrow temp.next$
```

```
6. $temp.next \leftarrow newnode$
```

Algorithm 2 - MergeLists

Input

1. l1 - first sorted linked list
2. l2 - second sorted linked list

Output

- l3 - merged sorted linked list containing all elements from l1 and l2

Steps

```
1. Initialize new empty list $l3$
2. $p1 \leftarrow l1.head$
3. $p2 \leftarrow l2.head$
4. While $p1 \neq null$ and $p2 \neq null$:
    1. If $p1.data \leq p2.data$:
        1. $l3.insert\_end(p1.data)$
        2. $p1 \leftarrow p1.next$
    2. Else:
        1. $l3.insert\_end(p2.data)$
        2. $p2 \leftarrow p2.next$
5. While $p1 \neq null$:
    1. $l3.insert\_end(p1.data)$
    2. $p1 \leftarrow p1.next$
6. While $p2 \neq null$:
    1. $l3.insert\_end(p2.data)$
    2. $p2 \leftarrow p2.next$
7. Return $l3$
```

4.2.3 Code

main.cpp

```cpp
#include<iostream>
#include "list.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.
        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";
        }
    }
}
```

list.h

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

class List
{
    struct node
    {
```

```
    int data;
    struct node *next;
}*head;

public:

struct node* gethead()
{
    return head;
}
List()
{
    head=NULL;
}

void insert_ascending(int num)
{
    struct node* newnode=new struct node;
    newnode->data=num;
    if(head==NULL)
    {
        newnode->next=NULL;
        head=newnode;
    }
    else if(head->data>num)
    {
        newnode->next=head;
        head=newnode;
    }
    else
    {
        struct node *temp=head;
        while(temp->next!=NULL && temp->next->data<num)
        {
            temp=temp->next;
        }
        newnode->next=temp->next;
        temp->next=newnode;
    }
}
List merge(List l1,List l2)
{
    List l3;
    struct node *temp1=l1.gethead();
    struct node *temp2=l2.gethead();
    while(temp1!=NULL && temp2!=NULL)
```

```
        {
            if(temp1->data<temp2->data)
            {
                l3.insert_ascending(temp1->data);
                temp1=temp1->next;
            }
            else
            {
                l3.insert_ascending(temp2->data);
                temp2=temp2->next;
            }
        }
        while(temp1!=NULL)
        {
            l3.insert_ascending(temp1->data);
            temp1=temp1->next;
        }
        while(temp2!=NULL)
        {
            l3.insert_ascending(temp2->data);
            temp2=temp2->next;
        }
        return l3;
    }
    void display()
    {
        struct node *temp=head;
        if(head==NULL)
        {
            cout << "List is empty";
        }
        else
        {
            while(temp!=NULL)
            {
                cout << temp->data << " ->";
                temp=temp->next;
            }
            cout << "NULL\n";
        }
    }
};
```

5 List ADT Circular and Doubly Linked Lists

5.1 Doubly Linked List with Tail

5.1.1 Question

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

5.1.2 Algorithm

Algorithm 1 - DoublyLinkedListInsertBeginning

Input

1. num - value to insert
2. head - pointer to head of doubly linked list
3. tail - pointer to tail of doubly linked list

Output

- Updated doubly linked list with num inserted at beginning

Steps

```
1. Create new node $newnode$ with $data = num$, $next = head$, $prev = null$
2. If $head = null$ (list is empty):
    1. $head \leftarrow newnode$
    2. $tail \leftarrow newnode$
3. Else:
    1. $head.prev \leftarrow newnode$
    2. $head \leftarrow newnode$
```

Algorithm 2 - DoublyLinkedListInsertEnd

Input

1. num - value to insert
2. head - pointer to head of doubly linked list
3. tail - pointer to tail of doubly linked list

Output

- Updated doubly linked list with num inserted at end

Steps

```
1. Create new node $newnode$ with $data = num$, $next = null$, $prev = tail$
2. If $head = null$ (list is empty):
    1. $head \leftarrow newnode$
    2. $tail \leftarrow newnode$
3. Else:
    1. $tail.next \leftarrow newnode$
    2. $tail \leftarrow newnode$
```

Algorithm 3 - DoublyLinkedListInsertPosition

Input

1. num - value to insert
2. pos - position to insert at
3. head - pointer to head of doubly linked list
4. tail - pointer to tail of doubly linked list

Output

- Updated doubly linked list with num inserted at position pos

Steps

```
1. If $pos = 1$:
    1. Call $insert\_beginning(num)$
    2. Return
2. $temp \leftarrow head$
3. For $i$ from 1 to $pos - 2$:
    1. $temp \leftarrow temp.next$
4. If $temp.next = null$:
    1. Call $insert\_end(num)$
    2. Return
5. Create new node $newnode$ with $data = num$
6. $newnode.next \leftarrow temp.next$
7. $newnode.prev \leftarrow temp$
8. $temp.next \leftarrow newnode$
```

```
9. $newnode.next.prev \leftarrow newnode$
```

Algorithm 4 - DoublyLinkedListDeleteBeginning

Input

1. head - pointer to head of doubly linked list
2. tail - pointer to tail of doubly linked list

Output

- Updated doubly linked list with first node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print “List is empty”
    2. Return
2. $temp \leftarrow head$
3. If $head.next = null$ (only one node):
    1. $head \leftarrow null$
    2. $tail \leftarrow null$
    3. Delete $temp$
    4. Return
4. $head \leftarrow head.next$
5. Delete $temp$
6. $head.prev \leftarrow null$
```

Algorithm 5 - DoublyLinkedListDeleteEnd

Input

1. head - pointer to head of doubly linked list
2. tail - pointer to tail of doubly linked list

Output

- Updated doubly linked list with last node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print “List is empty”
    2. Return
2. If $head.next = null$ (only one node):
    1. Call $delete\_beginning()$
    2. Return
```

3. *temp* ← *tail*
4. *tail* ← *tail.prev*
5. Delete *temp*
6. *tail.next* ← *null*

5.1.3 Code

main.cpp

```cpp
/*Doubly linked list implementation using head and tail, the menu driven program contains fo
1.Insert beginning
2.Insert end
3.insert position
4.delete beginning
5.delete end
6.delete position
7.display
8.search
9.exit
write the time complexities for each operation
*/


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


//Declaring the class for the list
class doubly{
    private:
        //declaring the structure , head and tail pointers
        struct node{
            int data;
            struct node *next;
            struct node *prev;
        }*head = nullptr, *tail = nullptr;
        int count = 0;

    public:
        //Member functions
        doubly(){
            head = nullptr;
            tail = nullptr;
        }
        void insert_beginning(int value);
        void insert_end(int value);
        void delete_beginning();
        void delete_end();
        void insert_position(int value);
        void delete_position();
        void search();
        void display();
};



int main(){
    int selection;
    int value;
    doubly list;
    //Menu Program
    while(1){
        cout << "\n<======= MENU =======>" << endl;
        cout << "1 -> Insert at the beginning" << endl;
        cout << "2 -> Insert at the end" << endl;
        cout << "3 -> Insert at position" << endl;
        cout << "4 -> Delete at beginning" << endl;
        cout << "5 -> Delete at end" << endl;
        cout << "6 -> Delete at position" << endl;
        cout << "7 -> Search" << endl;
        cout << "8 -> Display" << endl;
        cout << "9 -> Exit" << endl;
        cout << "Enter your choice:";
        cin >> selection;
        switch(selection){
            case 1:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_beginning(value);
                break;
            case 2:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_end(value);
                break;
            case 3:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_position(value);
                break;
            case 4:
                list.delete_beginning();
                break;
            case 5:
                list.delete_end();
                break;
            case 6:
                list.delete_position();
                break;
            case 7:
                list.search();
                break;
            case 8:
                list.display();
                break;
            case 9:
                cout << "Exiting....." << endl;
                return 0;
            default:
                cout << "The option selected CEASE to exist\nTry Again!!" << endl;
        }
    }
}


//Function to insert at beginning
void doubly::insert_beginning(int value){
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    newnode->data = value;
    newnode->next = nullptr;
    newnode->prev = nullptr;
    if(head == nullptr && tail == nullptr){
        head = newnode;
        tail = newnode;
    }else{
        head->prev = newnode;
        newnode->next = head;
        head = newnode;
    }
    count++;
}
//Time Complexity of this function is O(1)
```

```
//function to display the Linked List By traversing
void doubly::display(){
    struct node *temp = head;
    cout << "Doubly Linked List: NULL <-> ";
    while(temp != nullptr){
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}
//Time Complexity of this function is O(n)


//function for insertion at end
void doubly::insert_end(int value){
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = value;
    newnode->next = nullptr;
    newnode->prev = nullptr;
    struct node *temp = tail;
    newnode->prev = temp;
    temp->next = newnode;
    tail = newnode;
    count++;
}
//Time Complexity of this function is O(1)



//function for deletion ar beginning
void doubly::delete_beginning(){
    struct node *temp = head;
    if(head == tail){
        head = nullptr;
        tail = nullptr;
    }else{
        head = temp->next;
        temp->next = nullptr;
        head->prev = nullptr;
    }
    if(count <= 0){
        count = 0;
    }else{
        count--;
    }
}
```

```
//Time Complexity of this function is O(1)



//function for Deletion at the End
void doubly::delete_end(){
    struct node *temp = tail;
    if(head == tail){
        head = nullptr;
        tail = nullptr;
    }else{
        tail = temp->prev;
        tail->next = nullptr;
        temp->prev = nullptr;
    }
    if(count <= 0){
        count = 0;
    }else{
        count--;
    }
}
//Time Complexity of this function is O(1)



//Function for Insertion at a position
void doubly::insert_position(int value){
    int pos;
    cout << "Enter teh position to insert: ";
    cin >> pos;

    if(pos < 1 || pos > count){
        cout << "Warning position exceeds limits" << endl;
    }else if(pos == 1){
        insert_beginning(value);
    }else if(pos == count + 1){
        insert_end(value);
    }else{
        struct node *newnode = (struct node *)malloc(sizeof(struct node));
        newnode->data = value;
        newnode->next = nullptr;
        newnode->prev = nullptr;

        struct node *temp = head;

        for(int i = 1; i < pos - 1; i++){
            temp = temp->next;
        }
        struct node *temp2 = temp->next;
        temp->next = newnode;
        temp2->prev = newnode;
        newnode->prev = temp;
        newnode->next = temp2;
        count++;
    }
}
//Time Complexity for this function is O(n)



//Function to Delete at a position
void doubly::delete_position(){
    int pos;
    cout << "Enter posiion to delete:";
    cin >> pos;

    if(pos < 1 || pos > count){
        cout << "Warning position exceeds limits" << endl;
    }else if(pos == 1){
        delete_beginning();
    }else if(pos == count + 1){
        delete_end();
    }else{
        struct node *temp = head;
        struct node *temp2 = tail;

        for(int i = 1;i < pos - 1;i++){
            temp = temp->next;
        }
        struct node *temp3 = temp->next;
        temp3->next = nullptr;
        temp3->prev = nullptr;
        for(int i = count; i > pos + 1; i--){
            temp2 = temp2->prev;
        }
        temp->next = temp2;
        temp2->prev = temp;
        count--;
    }
}
//Time Complexity for the function is O(2n)
```

```
//Function to Search for an element
void doubly::search(){
    int value;
    cout << "Enter the value: ";
    cin >> value;
    int flag = 0;
    struct node *temp = head;
    while(temp != nullptr){
        if(value == temp->data){
            flag = 1;
            break;
        }
        temp = temp->next;
    }

    if(flag == 1){
        cout << "Element found!!" << endl;
    }else{
        cout << "Element not found!!" << endl;
    }
}
//Time Complexity for the function is O(n)
```

5.2 Circular Linked List

5.2.1 Question

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

5.2.2 Algorithm

Algorithm 1 - CircularLinkedListInsertBeginning

Input

1. num - value to insert
2. head - pointer to head of circular linked list

Output

- Updated circular linked list with num inserted at beginning

Steps

```
1. Create new node $newnode$ with $data = num$
2. If $head = null$ (list is empty):
    1. $head \leftarrow newnode$
    2. $newnode.next \leftarrow head$ (self-reference to make it circular)
3. Else:
    1. $temp \leftarrow head$
    2. While $temp.next \neq head$:
        1. $temp \leftarrow temp.next$
    3. $temp.next \leftarrow newnode$
    4. $newnode.next \leftarrow head$
    5. $head \leftarrow newnode$
```

Algorithm 2 - CircularLinkedListInsertEnd

Input

1. num - value to insert
2. head - pointer to head of circular linked list

Output

- Updated circular linked list with num inserted at end

Steps

```
1. If $head = null$ (list is empty):
    1. Call $insert\_beginning(num)$
2. Else:
    1. Create new node $newnode$ with $data = num$
    2. $newnode.next \leftarrow head$
    3. $temp \leftarrow head$
    4. While $temp.next \neq head$:
        1. $temp \leftarrow temp.next$
    5. $temp.next \leftarrow newnode$
```

Algorithm 3 - CircularLinkedListInsertPosition

Input

1. num - value to insert
2. pos - position to insert at
3. head - pointer to head of circular linked list

Output

- Updated circular linked list with num inserted at position pos

Steps

```
1. If $pos = 1$:
    1. Call $insert\_beginning(num)$
2. Else:
    1. Create new node $newnode$ with $data = num$
    2. $temp \leftarrow head$
    3. For $i$ from 1 to $pos - 2$:
        1. $temp \leftarrow temp.next$
    4. $newnode.next \leftarrow temp.next$
    5. $temp.next \leftarrow newnode$
```

Algorithm 4 - CircularLinkedListDeleteBeginning

Input

1. head - pointer to head of circular linked list

Output

- Updated circular linked list with first node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print "List is empty"
2. Else:
    1. $temp \leftarrow head$
    2. While $temp.next \neq head$:
        1. $temp \leftarrow temp.next$
    3. $temp.next \leftarrow head.next$
    4. Delete $head$
    5. $head \leftarrow temp.next$
```

Algorithm 5 - CircularLinkedListDeleteEnd

Input

1. head - pointer to head of circular linked list

Output

- Updated circular linked list with last node removed

Steps

```
1. If $head = null$ (list is empty):
    1. Print “List is empty”
2. Else:
    1. $temp \leftarrow head$
    2. While $temp.next.next \neq head$:
        1. $temp \leftarrow temp.next$
    3. Delete $temp.next$
    4. $temp.next \leftarrow head$
```

Algorithm 6 - CircularLinkedListDeletePosition

Input

1. pos - position to delete
2. head - pointer to head of circular linked list

Output

- Updated circular linked list with node at position pos removed

Steps

```
1. If $pos = 1$:
    1. Call $delete\_beginning()$
2. Else:
    1. $temp \leftarrow head$
    2. For $i$ from 1 to $pos - 2$:
        1. $temp \leftarrow temp.next$
    3. $temp1 \leftarrow temp.next$
    4. $temp.next \leftarrow temp1.next$
    5. Delete $temp1$
```

Algorithm 7 - CircularLinkedListSearch

Input

1. num - value to search for
2. head - pointer to head of circular linked list

Output

- Position of num in the linked list if found, else message “Element not found”

Steps

```
1. $temp \leftarrow head$
2. $pos \leftarrow 1$
3. While $temp.next \neq head$:
    1. If $temp.data = num$:
        1. Print “Element found at position $pos$”
        2. Return
    2. $temp \leftarrow temp.next$
    3. $pos \leftarrow pos + 1$
4. If $temp.data = num$:
    1. Print “Element found at position $pos$”
5. Else:
    1. Print “Element not found”
```

5.2.3 Code

main.cpp

```cpp
/*Circular linked list*/

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

//class for circular linked list
class circular{
    private:
        struct node{
            int data;
            struct node* link;
        }*head = nullptr;
        int count = 0;

    public:
        circular(){
            head = nullptr;
        }
        //member functions
        void insert_beginning(int value);
        void insert_end(int value);
        void insert_position(int value);
        void delete_beginning();
        void delete_end();
        void delete_position();
        void search();
        void display();

};

int main(){
    int selection;
    int value;
    circular list;
    while(1){
        cout << "\n<======= MENU =======>" << endl;
        cout << "1 -> Insert at the beginning" << endl;
        cout << "2 -> Insert at the end" << endl;
        cout << "3 -> Insert at position" << endl;
        cout << "4 -> Delete at beginning" << endl;
        cout << "5 -> Delete at end" << endl;
        cout << "6 -> Delete at position" << endl;
        cout << "7 -> Search" << endl;
        cout << "8 -> Display" << endl;
        cout << "9 -> Exit" << endl;
        cout << "Enter your choice:";
        cin >> selection;
        switch(selection){
            case 1:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_beginning(value);
                break;
            case 2:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_end(value);
                break;
            case 3:
                cout << "Enter the value to insert:";
                cin >> value;
                list.insert_position(value);
                break;
            case 4:
                list.delete_beginning();
                break;
            case 5:
                list.delete_end();
                break;
            case 6:
                list.delete_position();
                break;
            case 7:
                list.search();
                break;
            case 8:
                list.display();
                break;
            case 9:
                cout << "Exiting....." << endl;
                return 0;
            default:
                cout << "The option selected CEASE to exist\nTry Again!!" << endl;
        }
    }
}


//Function for Insertion at beginning
void circular::insert_beginning(int value){
    int num = value;
    struct node *cnode = (struct node*)malloc(sizeof(struct node));
    cnode->data = num;
    cnode->link = nullptr;
    if(head == nullptr){
        head = cnode;
        cnode->link = head;
        count++;
    }else{
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        temp = head;
        cnode->link = temp;
        while(temp->link != head){
            temp = temp->link;
        }
        temp->link = cnode;
        head = cnode;
        count++;
    }
```

```
}
//Time Complexity = O(n)


//Function for insertion at end
void circular::insert_end(int value){
    int num = value;
    struct node *cnode = (struct node*)malloc(sizeof(struct node));
    cnode->data = num;
    cnode->link = nullptr;
    if(head == nullptr){
        head = cnode;
        cnode->link = head;
        count++;
    }else{
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        temp = head;
        while(temp->link != head){
            temp = temp ->link;
        }
        temp->link = cnode;
        cnode->link = head;
        count++;
    }
}
//Time complexity = O(n)


//function for insertion at position
void circular::insert_position(int value){
    int pos;
    cout << "Enter the position for insertion:";
    cin >> pos;
    if(pos <= 0 || pos > count + 1){
        cout << "Position entered exceeds Limits\nTry Again" << endl;
    }else{
        if(pos == 1){
            insert_beginning(value);
            return;
        }
        int num = value;
        struct node *cnode = (struct node*)malloc(sizeof(struct node));
        cnode->data = num;
        cnode->link = nullptr;
        struct node *temp = head;
        struct node *temp2;
        for(int i = 1; i < pos - 1; i++) {
            temp = temp->link;
        }
        temp2 = temp->link;
        temp->link = cnode;
        cnode->link = temp2;
        count++;
    }
}
//Time complexity = O(n)


//Function to Delete at beginning
void circular::delete_beginning(){
    if(count <= 0){
        cout << "The list is empty" << endl;
    }else{
        struct node *temp3 = head;
        struct node *temp = head;
        struct node *temp2 = temp->link;
        while(temp3->link != temp){
            temp3 = temp3->link;
        }
        temp3->link = temp2;
        head = temp2;
        temp->link = nullptr;
        count--;
    }
}
//Time complexity is O(n)

//function for deletion at the end
void circular::delete_end(){
    if(count <= 0){
        cout << "The list is empty" << endl;
    }else{
        struct node *temp = head;
        struct node *temp2 = head;
        while(temp->link != head){
            temp = temp->link;
        }
        temp->link = nullptr;
        for(int i = 1;i < count; i++ ){
            temp2 = temp2->link;
        }
        temp2->link = head;
        count--;
    }
}
//Time complexity is O(n)

//Function for deletion at position
void circular::delete_position(){
    int pos;
    cout << "Enter the position to delete:";
    cin >> pos;
    if(pos <= 0 || pos > count){
        cout << "Position entered exceeds Limits\nPlease Try again" << endl;
    }else if(pos == 1){
        delete_beginning();
    }else if(pos == count){
        delete_end();
    }else{
        struct node *temp = head;
        struct node *prev = nullptr;
        for(int i = 1; i < pos; i++){
            prev = temp;
            temp = temp->link;
        }
        prev->link = temp->link;
        temp->link = nullptr;
        free(temp);
        count--;
    }
}
//Time complexity is O(n)


//Function for displaying the linked list
void circular::display(){
    struct node *temp = (struct node*)malloc(sizeof(struct node));
    temp = head;
    int tcount = count;
    cout << "Circular List";
    while(temp ->link != nullptr && tcount--){
        cout << " -> " << temp->data ;
        temp = temp ->link;
    }
}
//time complexity for the display is O(n)
```

```
//Function for searching
void circular::search() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    struct node *temp = head;
    int value;
    cout << "Enter the value to search: ";
    cin >> value;

    while (true) {
        if (temp->data == value) {
            cout << "Element found in the list!!" << endl;
            return;
        }
        temp = temp->link;

        if (temp == head)
            break;
    }

    cout << "Element not found in the list!!" << endl;
}
//time complexity is O(n)
```

5.3 Round Robin Scheduler

5.3.1 Question

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 Proces
2. Execute
3. Exit

The “Insert Process” will get an integer time from the user and add it to the queue. The “Exccute” operation will exceute a deletion in the beginning operation and subtract the fixcd time from the process. If the processing time falls below 0 then the process is considered to have completed its execution, otherwise, the remaining time after subtraction is inserted at the end of the circular linked list.

5.3.2 Algorithm

Algorithm 1 - RoundRobinScheduler

Input

1. processList - circular linked list containing process burst times
2. timeSlot - time quantum for each execution

Output

- Execution order of processes

Steps

```
1. Initialize circular linked list $processList$
2. Input time quantum $timeSlot$
3. While $processList$ is not empty:
    1. $val \leftarrow processList.delete\_beginning()$ (Extract first process)
    2. $remainingTime \leftarrow val - timeSlot$
    3. If $remainingTime > 0$:
        1. $processList.insert\_end(remainingTime)$ (Reinsert process at end with updated time)
    4. Else:
        1. Process completed execution
```

5.3.3 Code

main.cpp

```
/*C. An operating system allocates a fixed time slot CPU time for processes using a round-r

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

Insert Process
Execute
Exit
The "Insert Process" will get an integer time from the user and add it to the queue.

The "Execute" operation will execute a deletion in the beginning operation and subtract the

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

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

```
//class for Round-Robin-Schedule
class schedule{
    private:
        struct process{
            int time;
            struct process *link;
        }*head = nullptr, *tail = nullptr;
        int pro_count = 0;
        int timeslice = 0;

    public:
    //Constructor
        schedule(){
            head = nullptr;
            tail = nullptr;
        }
    //member function protoypes
        void getimeslice();
        void insert(int pro_time);
        void delete_process();
        void execute();
        void display();


};


//main function
int main(){
    int selection;
    schedule cpu;
    cpu.getimeslice();
    while(1){
        //Menu-Driven program
        cout << "\n<===== MENU =====>" << endl;
        cout << "1.Insert Process" << endl;
        cout << "2.Execute" << endl;
        cout << "3.Exit" << endl;
        cout << "Enter your choice: ";
        cin >> selection;
        switch(selection){
            case 1:
            //inserting the process
                int process_time;
                cout << "Enter the process time: ";
```

```
                    cin >> process_time;
                    cpu.insert(process_time);
                    break;
                case 2:
                //executing the process
                    cpu.execute();
                    break;
                case 3:
                //exiting the menu
                    cout << "Exiting....." << endl;
                    return 0;
                default:
                    cout << "Selected choice Cease to exist\nplease Try Again" << endl;
            }
        }
}


//member function to get the time alloted for each process
void schedule::getimeslice(){
    cout << "Enter the time slice a process: ";
    cin >> timeslice;
}


//member function for inserting the process
void schedule::insert(int pro_time){
    struct process *newproccess = (struct process *)malloc(sizeof(struct process));
    newproccess->time = pro_time;
    newproccess->link = nullptr;
    if(head == nullptr && tail == nullptr){
        head = newproccess;
        tail = newproccess;
        newproccess->link = head;
    }else{
        tail->link = newproccess;
        tail = newproccess;
        newproccess->link = head;
    }
    pro_count++;
    //display();
}
//Time Complexity is O(1)
```

```
//The below display function is to check how to process are running inside the circular lin

/*void schedule::display(){
    struct process *temp = head;
    cout << "Process times to complete in queue: ";
    while(temp != tail){
        cout << temp->time << " <- ";
        temp = temp->link;
    }
    if(temp == nullptr){
        return;
    }else{
        cout << temp->time << endl;
    }
}
//Time Complexity is O(n)
*/


//member function to delete a process
void schedule::delete_process(){
    if(head == tail){
        head->link = nullptr;
        head = nullptr;
        tail = nullptr;
    }else{
        struct process *temp = head;
        head = temp->link;
        temp->link = nullptr;
        tail->link = head;
        free(temp);
    }
    pro_count--;
}
//Time Complexity is O(1)


//member function to execute a process
void schedule::execute(){
    struct process *temp = head;
    struct process *temp2 = head;
    if(temp == nullptr){
        cout << "There are no process to execute!!" << endl;
    }else{
        temp->time = temp->time - timeslice;
        if(temp->time <= 0){
            delete_process();
        }else{
            head = temp->link;
            while(temp != tail){
                temp = temp->link;
            }
            temp->link->link = temp2;
            tail = temp2;
        }

        cout << "Execution Completed!!" << endl;
        cout << "No of process pending --> " << pro_count << endl;
    }
    //display();
}
//Time Complexity is O(n + 1)
//1 form display
//n for the execute function
```

6 Stack ADT

6.1 Character Array Stack ADT

6.1.1 Question

Write a separate C++ menu-driven program to implement stack ADT using a character array of size 5. Maintain proper boundary conditions and follow good coding practices. Stack ADT has the following operations

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

6.1.2 Algorithm

Algorithm 1 - ArrayStackPush

Input

1. ch - character to push
2. arr[] - array storing the stack elements
3. cur - current position (top) of stack

Output

- Updated stack with character ch added

Steps

```
1. If $cur = 5$ (stack is full):
    1. Print "List is full"
    2. Return
2. $arr[cur] \leftarrow ch$
3. $cur \leftarrow cur + 1$
```

Algorithm 2 - ArrayStackPop

Input

1. arr[] - array storing the stack elements
2. cur - current position (top) of stack

Output

- Character from the top of stack and updated stack

Steps

```
1. If $cur = 0$ (stack is empty):
    1. Print "List is empty"
    2. Return $0$
2. $cur \leftarrow cur - 1$
3. Return $arr[cur]$
```

Algorithm 3 - ArrayStackPeek

Input

1. arr[] - array storing the stack elements
2. cur - current position (top) of stack

Output

- Character from the top of stack (without removing it)

Steps

```
1. If $cur = 0$ (stack is empty):
    1. Print "List is empty"
    2. Return
2. Print $arr[cur - 1]$
```

6.1.3 Code

main.cpp

```cpp
/*A. Write a separate C++ menu-driven program to implement stack ADT using a character arra

1.Push
2.Pop
3.Peek
4.Exit
What is the time complexity of each of the operations? (K4)

*/

#include <iostream>
using namespace std;

#define size 5//making a fixed size of 5

//creating the stack class
class stack{
    private:
        //data members
        char arr[size];
        int count;
        public:
        //Constructor of the stack
        stack(){
            count = 0;
        }
        //member functions:
        void push(char ch);
        void pop();
        void peek();
        void display();
};

int main(){
    int choice;
    stack s;
    while(1){
        cout << "\n<===== MENU =====>" << endl;
        cout << "1.Push" << endl;
        cout << "2.Pop" << endl;
        cout << "3.Peek" << endl;
        cout << "4.Exit" << endl;
        cout << "Select your choice" << endl;
        cin >> choice;
        switch(choice){
            case 1:
                char c;
                cout << "Enter a character to push into the stack: ";
                cin >> c;
                s.push(c);
                break;
            case 2:
                s.pop();
                break;
            case 3:
                s.peek();
                break;
            case 4:
                cout << "Exiting...." << endl;
                return 0;
            default:
                cout << "Selected Option Cease to exist\nPlease Try Again" << endl;
        }
    }
}

//Defining the push function
void stack::push(char ch){
    if(count == size){
        cout << "This action causes stack overflow\nProcess terminated" << endl;
    }else{
        arr[count] = ch;
        count++;
        cout << "The element " << ch << "pushed into the stack" << endl;
    }
    display();
}
//Time complexity of the push function is O(1)



//defining the pop function
void stack::pop(){
    if(count <= 0){
        cout << "The Stack is Empty\nPush elements and Try Again" << endl;
    }else{
        arr[count] = 0;
        count--;
        cout << "Top element of the stack is removed/poped" << endl;
    }
    display();
}
//Time complexity if the pop function is O(1)



//Defining the peek function
void stack::peek(){
    if(count <= 0){
        cout << "The stack is empty \n" << endl;
    }else{
        cout << "Top element of the stack is " << arr[count - 1] << endl;
    }
}
//Time complexity of the peek function is O(1)



//Defining the Display function
```

```
void stack::display(){
    int loopcount = count;
    cout << "STACK --->" << endl << endl;
    while(loopcount--){
        cout << "|" << arr[loopcount] << "|" << endl;
        cout << "---" << endl;
    }
}
//TIme complexity of the display function is O(1)
```

6.2 Character Linked List Stack ADT

6.2.1 Question

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

6.2.2 Algorithm

Algorithm 1 - StackPush

Input

1. ch - character to push
2. head - pointer to top of stack

Output

- Updated stack with character ch added

Steps

```
1. Create new node $newnode$ with $data = ch$ and $next = null$
2. If $head = null$ (stack is empty):
    1. $head \leftarrow newnode$
3. Else:
    1. $temp \leftarrow head$
    2. While $temp.next \neq null$:
        1. $temp \leftarrow temp.next$
    3. $temp.next \leftarrow newnode$
```

Algorithm 2 - StackPop

Input

1. head - pointer to top of stack

Output

- Character from the top of stack and updated stack

Steps

```
1. If $head = null$ (stack is empty):
    1. Print “List is empty”
    2. Return
2. $temp \leftarrow head$
3. If $temp.next = null$ (only one element):
    1. $ch \leftarrow head.data$
    2. Free $head$
    3. $head \leftarrow null$
    4. Return $ch$
4. Else:
    1. While $temp.next.next \neq null$:
        1. $temp \leftarrow temp.next$
    2. $ch \leftarrow temp.next.data$
    3. Free $temp.next$
    4. $temp.next \leftarrow null$
    5. Return $ch$
```

Algorithm 3 - StackPeek

Input

1. head - pointer to top of stack

Output

- Character from the top of stack (without removing it)

Steps

```
1. If $head = null$ (stack is empty):
    1. Print “List is empty”
    2. Return 0
2. $temp \leftarrow head$
3. While $temp.next \neq null$:
    1. $temp \leftarrow temp.next$
4. Return $temp.data$
```

6.2.3 Code

main.cpp

```
/*B. Write a separate C++ menu-driven program to implement stack ADT using a character sing

Push
Pop
Peek
Exit
What is the time complexity of each of the operations? (K4)*/



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


//Creatuing a class for stack using singly linked list
class singlystack{
    private:
        //data members
        struct node{
            char ch;
            struct node *link;
        }*head = nullptr, *tail = nullptr;
        int count = 0;

    public:
        //Constructor
        singlystack(){
            head = nullptr;
            tail = nullptr;
        }
        //member functions
        void push(char c);
        void pop();
        void peek();
        void reverse_link();
        void display();
};



int main(){
    int choice;
    singlystack stack;
    //Menu Program
    while(1){
        cout << "\n<===== MENU =====>" << endl;
        cout << "1.Push" << endl;
        cout << "2.Pop" << endl;
        cout << "3.Peek" << endl;
        cout << "4.Exit" << endl;
        cout << "Select your choice" << endl;
        cin >> choice;
        switch(choice){
            case 1:
                char ch;
                cout << "Enter a character to push into the stack: ";
                cin >> ch;
                stack.push(ch);
                break;
            case 2:
                stack.pop();
                break;
            case 3:
                stack.peek();
                break;
            case 4:
                cout << "Exiting...." << endl;
                return 0;
            default:
                cout << "Selected Option Cease to exist\nPlease Try Again" << endl;
        }
    }
}


//function to push character into a stack
```

```
void singlystack::push(char c){
    struct node *chnode = (struct node*)malloc(sizeof(struct node));
    chnode->ch = c;
    chnode->link = nullptr;
    struct node *temp = tail;
    if(head == nullptr){
        head = chnode;
        tail = chnode;
    }else{
        temp->link = chnode;
        tail = chnode;
    }
    count++;
    display();
}
//Time complexity is O(1)without display, It is O(n + 1)with display



//defining the pop function
void singlystack::pop(){
    if((head == nullptr && tail == nullptr)){
        cout << "The Stack is empty\n" << endl;
        return;
    }else{
        if(count == 1){
            head->link = nullptr;
            head = nullptr;
            tail = nullptr;
            count--;
        }else{
            struct node *temp = head;
            while(temp->link != tail){
                temp = temp->link;
            }
            temp->link = nullptr;
            tail = temp;
            count--;
        }
        display();
    }
}
//Time complexity is O(n)


//Defining the function for peek
```

```
void singlystack::peek(){
    if(count == 0){
        cout << "The stack is empty" << endl;
    }else{
        struct node *temp = head;
        while(temp->link != nullptr){
            temp = temp->link;
        }
        cout << "Top element of the stack is " << temp->ch << endl;
    }
}
//TIme complexity is O(n)


//Defining the function for display
void singlystack::display(){
    reverse_link();
    struct node *temp = head;
    cout << "Current stack :\n" << endl;
    while(temp != nullptr){
        cout << "|" << temp->ch << "|" << endl;
        cout << "---" << endl;
        temp = temp->link;
    }
    reverse_link();
}
//Time complexity if O(n)


//Defining the function to reverse a link
void singlystack::reverse_link(){
    struct node *current = head;
    struct node *prev = nullptr;
    struct node *next = nullptr;
    head = tail;
    tail = current;
    while(current != nullptr){
        next = current->link;
        current->link = prev;
        prev = current;
        current = next;
    }
    head = prev;
}
//TIme complexity is O(n)
```

6.3 Infix to Postfix

6.3.1 Question

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,

1. Get Infix
2. Convert Infix
3. Evaluate Postfix
4. Exit

6.3.2 Algorithm

Algorithm 1 - InfixToPostfix

Input

1. infix - infix expression string

Output

- postfix - equivalent postfix expression string

Steps

```
1. Initialize empty stack $s$
2. Initialize empty string $postfix$
3. For each character $c$ in $infix$:
    1. If $c$ is a digit (operand):
        1. Append $c$ to $postfix$
    2. Else if $c =' ('$:
        1. Push $c$ onto stack $s$
    3. Else if $c =')'$:
        1. While $s$ is not empty and top of $s$ is not '(':
            1. Append $s.pop()$ to $postfix$
        2. If $s$ is not empty and top of $s$ is '(':
            1. Pop '(' from $s$
    4. Else if $c$ is an operator $(+, -, *, /, \%)$:
        1. While $s$ is not empty and top of $s$ is not '(' and $precedence(s.peek()) \geq precedence(c)$:
            1. Append $s.pop()$ to $postfix$
        2. Push $c$ onto stack $s$
4. While $s$ is not empty:
    1. If top of $s$ is not '(':
        1. Append $s.pop()$ to $postfix$
    2. Else:
        1. Pop ‘(’ from $s$
5. Return $postfix$
```

Algorithm 2 - EvaluatePostfix

Input

1. postfix - postfix expression string

Output

- result - numerical result after evaluating the expression

Steps

```
1. Initialize empty stack $s$
2. For each character $c$ in $postfix$:
    1. If $c$ is a digit (operand):
        1. Convert $c$ to integer and push onto stack $s$
    2. Else ($c$ is an operator):
        1. $op2 \leftarrow s.pop()$
        2. $op1 \leftarrow s.pop()$
        3. Apply operator $c$ on $op1$ and $op2$:
            1. If $c =' +'$: $s.push(op1 + op2)$
            2. If $c =' -'$: $s.push(op1 - op2)$
            3. If $c =' *'$: $s.push(op1 * op2)$
            4. If $c =' /'$: $s.push(op1/op2)$
            5. If $c =' \%'$: $s.push(op1 \bmod op2)$
3. Return $s.pop()$ as the final result
```

6.3.3 Code

main.cpp

stack.h

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

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

```
class Stack {
private:
    Node* top;
public:
    Stack() { top = nullptr; }

    bool isEmpty() { return top == nullptr; }

    void push(char data) {
        Node* newNode = new Node();
        if (!newNode) {
            cout << "Memory allocation failed\n";
            return;
        }
        newNode->data = data;
        newNode->next = top;
        top = newNode;
    }

    char pop() {
        if (isEmpty()) {
            cout << "Stack underflow\n";
            return '\0';
        }
        Node* temp = top;
        char poppedData = temp->data;
        top = top->next;
        delete temp;
        return poppedData;
    }

    char peek() {
        return isEmpty() ? '\0' : top->data;
    }
};
```

6.4 Parenthesis Balance

6.4.1 Question

Write a C++ menu-driven program to get a string of '(' and ')' parenthesis from the user and check whether they are balanced. Identify the optimal ADT and data structure to solve the mentioned problem. You can consider all previous header files for the solution's implementation. Maintain proper boundary conditions and follow good coding practices. The program has the following operations,

1. Check Balance

2. Exit

The Check Balance operations get a string of open and closed parentheses. Additionally, it displays whether the parenthesis is balanced or not. Explore at least two designs (solutions) before implementing your solution.

6.4.2 Algorithm

Algorithm 1 - ParenthesisBalance

Input

1. s - string containing parentheses

Output

- Boolean indicating whether the parentheses are balanced

Steps

```
1. Initialize empty stack $st$
2. For each character $c$ in string $s$:
    1. If $c$ is an opening bracket ('(' or '{' or '['):
        1. Push $c$ onto stack $st$
    2. Else if $c$ is a closing bracket (')' or '}' or ']'):
        1. If $st$ is empty:
            1. Return $false$
        2. If (st.peek() = '(' and c = ')') or (st.peek() = '{' and c = '}') or
           (st.peek() = '[' and c = ']'):
            1. Pop top element from $st$
        3. Else:
            1. Return $false$
3. Return $st.empty()$
```

6.4.3 Code

main.cpp

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

class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        for (char c : s) {
            if (c == '(') {
                st.push(c);
            } else {
                if (st.empty()) return false;
                char top = st.top();
                if ((c == ')' && top == '(')) {
                    st.pop();
                } else {
                    return false;
                }
            }
        }
        return st.empty();
    }
};

int main() {
    Solution sol;
    int choice;
    string expression;

    while (true) {
        cout << "\nMenu:" << endl;
        cout << "1. Check Balance" << endl;
        cout << "2. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter expression: ";
                cin >> expression;
                if (sol.isValid(expression)) {
                    cout << "Balanced" << endl;
                } else {
                    cout << "Not Balanced" << endl;
                }
                break;
            case 2:
                cout << "Exiting program..." << endl;
                return 0;
            default:
                cout << "Invalid choice. Please try again." << endl;
        }
    }
    return 0;
}
```

7 Queue ADT

7.1 Integer Array Queue ADT

7.1.1 Question

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

7.1.2 Algorithm

Algorithm 1 - QueueEnqueue

Input

1. num - element to insert into queue
2. arr[] - array storing the queue elements
3. cur - current number of elements

Output

- Updated queue with num inserted at the end

Steps

```
1. If $cur = 5$:
    1. Print “Queue is full”
    2. Return
2. $arr[cur] \leftarrow num$
3. $cur \leftarrow cur + 1$
```

Algorithm 2 - QueueDequeue

Input

1. arr[] - array storing the queue elements
2. cur - current number of elements

Output

- Updated queue with front element removed

Steps

```
1. If $cur = 0$:
    1. Print "Queue is empty"
    2. Return
2. For $i$ from 0 to $cur - 2$:
    1. $arr[i] \leftarrow arr[i+1]$
3. $cur \leftarrow cur - 1$
```

Algorithm 3 - QueuePeek

Input

1. arr[] - array storing the queue elements
2. cur - current number of elements

Output

- Value of the front element (without removing it)

Steps

```
1. If $cur = 0$:
    1. Print "Queue is empty"
    2. Return
2. Print $arr[0]$
```

7.1.3 Code

main.cpp

```cpp
/*A. Write a separate C++ menu-driven program to implement Queue ADT using an integer array

Enqueue
Dequeue
Peek
Exit
What is the time complexity of each of the operations? (K4)

*/

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


//Defining the class for Queue
```

```
class queue{
    private:
        //Data members
        vector<int> arr;
        int front;
        int rear;

    public:
        //CONSTRUCTOR
        queue(){
            front = -1;
            rear = -1;
        }
        bool isempty();//Function to check if queue is Empty
        bool isfull();//Funtion to check if the queue is Full
        void enqueue(int val);//Enqueue element into the list
        void dequeue();//Dequeue element from the list
        void peek();//Info about first element
        void display();//Display funtion to see the queue
};


//Main Function for Menu Program
int main(){
    int choice;
    queue obj;
    while(1){
        cout << "\n\n<===== MENU =====>" << endl;
        cout << "1.Enqueue" << endl;
        cout << "2.Dequeue" << endl;
        cout << "3.Peek" << endl;
        cout << "4.Exit" << endl;
        cout << "Select your choice: " << endl;
        cin >> choice;
        switch(choice){
            case 1:
                int value;
                cout << "Enter a value to enqueue!" << endl;
                cin >> value;
                obj.enqueue(value);
                break;
            case 2:
                obj.dequeue();
                break;
            case 3:
                obj.peek();
                break;
            case 4:
                cout << "Exiting..." << endl;
                return 0;
            default:
                cout << "The selelcted choice cease to Exist\nPlease Try Again" << endl;
        }
    }
}



//DEFINING THE FUNCTION

bool queue::isempty(){
    if(rear == -1 && front == -1){
        return true;
    }else{
        return false;
    }
}

bool queue::isfull(){
    if(rear >= 4){
        return true;
    }else{
        return false;
    }
}

void queue::enqueue(int val){
    if(isfull()){
        cout << "The queue is Full!, Dequeue and Try Again!!" << endl;
    }else{
        arr.push_back(val);
        rear++;
    }
    display();
}

void queue::display(){
    for(int i = 0;i <= rear;i++){
        cout << arr[i]<< " | ";
    }
}
```

```
void queue::dequeue(){
    if(isempty()){
        cout << "The queue is empty! Enqueue and Try Again!!" << endl;
    }else{
        int temp = arr[0];
        for(int i = 0;i < rear; i++){
            arr[i] = arr[i + 1];
        }
        rear--;
        cout << temp << " Dequeued from the queue!!" << endl;
        display();
    }
}


void queue::peek(){
    if(isempty()){
        cout << "The queue is empty!" << endl;
    }else{
        cout << "Front in the queue is " << arr[0] << endl;
    }
}
```

7.2 Integer Array Circular Queue ADT

7.2.1 Question

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

7.2.2 Algorithm

Algorithm 1 - CircularQueueEnqueue

Input

1. x - element to insert into queue
2. arr[] - array storing the queue elements
3. front - front index of queue
4. rear - rear index of queue
5. size - current number of elements

6. MAX_SIZE - maximum capacity of queue

Output

- Updated circular queue with x inserted

Steps

```
1. If $size = MAX\_SIZE$:
    1. Print “Queue is full”
    2. Return
2. If $front = -1$ (queue is empty):
    1. $front \leftarrow 0$
3. $rear \leftarrow (rear + 1) \bmod MAX\_SIZE$
4. $arr[rear] \leftarrow x$
5. $size \leftarrow size + 1$
```

Algorithm 2 - CircularQueueDequeue

Input

1. arr[] - array storing the queue elements
2. front - front index of queue
3. rear - rear index of queue
4. size - current number of elements
5. MAX_SIZE - maximum capacity of queue

Output

- Updated circular queue with front element removed

Steps

```
1. If $size = 0$:
    1. Print “Queue is empty”
    2. Return
2. $front \leftarrow (front + 1) \bmod MAX\_SIZE$
3. $size \leftarrow size - 1$
```

Algorithm 3 - CircularQueuePeek

Input

1. arr[] - array storing the queue elements
2. front - front index of queue
3. size - current number of elements

Output

- Value of the front element (without removing it)

Steps

```
1. If $size = 0$:
    1. Print “Queue is empty”
    2. Return
2. Print $arr[front]$
```

7.2.3 Code

main.cpp

```cpp
/*B. Write a separate C++ menu-driven program to implement Circular Queue ADT using an inte

Enqueue
Dequeue
Peek
Exit
What is the time complexity of each of the operations? (K4)

*/

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

#define SIZE 5  // Fixed size for the circular queue

// Circular Queue Class
class CircularQueue {
private:
    vector<int> arr;
    int front, rear;

public:
    // Constructor
    CircularQueue() {
        arr.resize(SIZE, 0); // Initialize vector with fixed size
        front = -1;
        rear = -1;
    }

    bool isEmpty();  // Check if queue is empty
    bool isFull();   // Check if queue is full
    void enqueue(int val); // Insert element
    void dequeue();        // Remove element
    void peek();           // Get front element
    void display();        // Display queue elements
};

// Main Menu Function
int main() {
    CircularQueue obj;
    int choice, value;

    while (true) {
        cout << "\n\n<===== MENU =====>" << endl;
        cout << "1. Enqueue" << endl;
        cout << "2. Dequeue" << endl;
        cout << "3. Peek" << endl;
        cout << "4. Exit" << endl;
        cout << "Select your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter a value to enqueue: ";
                cin >> value;
                obj.enqueue(value);
                break;
            case 2:
                obj.dequeue();
                break;
            case 3:
                obj.peek();
                break;
            case 4:
                cout << "Exiting..." << endl;
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    }
}

// Check if queue is empty
bool CircularQueue::isEmpty() {
    return front == -1;
}
```

```cpp
// Check if queue is full
bool CircularQueue::isFull() {
    return (rear + 1) % SIZE == front;
}

// Enqueue function
void CircularQueue::enqueue(int val) {
    if (isFull()) {
        cout << "Queue is Full! Cannot enqueue " << val << endl;
        return;
    }
    if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % SIZE;
    }
    arr[rear] = val;
    cout << val << " enqueued successfully." << endl;
    display();
}

// Dequeue function
void CircularQueue::dequeue() {
    if (isEmpty()) {
        cout << "Queue is Empty! Cannot dequeue." << endl;
        return;
    }
    cout << arr[front] << " dequeued successfully." << endl;
    if (front == rear) { // Only one element was present
        front = rear = -1;
    } else {
        front = (front + 1) % SIZE;
    }
    display();
}

// Peek function
void CircularQueue::peek() {
    if (isEmpty()) {
        cout << "Queue is Empty! No front element." << endl;
    } else {
        cout << "Front element: " << arr[front] << endl;
    }
}
```

```cpp
// Display function
void CircularQueue::display() {
    if (isEmpty()) {
        cout << "Queue is Empty!" << endl;
        return;
    }
    cout << "Queue elements: ";
    int i = front;
    while (true) {
        cout << arr[i] << " ";
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    cout << endl;
}
```

7.3 Integer Linked List Queue ADT

7.3.1 Question

Write a separate G++ 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:

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

7.3.2 Algorithm

Algorithm 1 - QueueEnqueue

Input

1. x - element to insert into queue
2. front - pointer to front of queue
3. rear - pointer to rear of queue

Output

- Updated queue with x inserted at the rear

Steps

```
1. Create new node $temp$ with $data = x$ and $next = null$
2. If $front = null$ (queue is empty):
    1. $front \leftarrow temp$
```

```
        2. $rear \leftarrow temp$
        3. Return
3. $rear.next \leftarrow temp$
4. $rear \leftarrow temp$
```

Algorithm 2 - QueueDequeue

Input

1. front - pointer to front of queue
2. rear - pointer to rear of queue

Output

- Updated queue with front element removed

Steps

```
1. If $front = null$ (queue is empty):
    1. Print "Queue is empty"
    2. Return
2. $temp \leftarrow front$
3. $front \leftarrow front.next$
4. Delete $temp$
```

Algorithm 3 - QueuePeek

Input

1. front - pointer to front of queue

Output

- Value of the front element (without removing it)

Steps

```
1. If $front = null$ (queue is empty):
    1. Print "Queue is empty"
    2. Return
2. Print $front.data$
```

7.3.3 Code

main.cpp

```
/*C. Write a separate C++ menu-driven program to implement Queue ADT using an integer-linke

Enqueue
Dequeue
Peek
Exit*/

#include <iostream>
using namespace std;

//Class for implementing Queue using linked list
class queue{
    private:
        //Data Memebers
        struct node{
            int data;
            struct node *link;
        }*front, *rear;

    public:
        //Constructor
        queue(){
            front = nullptr;
            rear = nullptr;
        }
        void Enqueue(int val);//FUnction to Enqueue an element
        void Dequeue();//Function to Dequeue an element
        void peek();//Function to show the Element at front
        void display();//Function to dispaly the Queue
        bool isempty();//Function to check is the Queue is Empty
};


//Main code Block for Menu Program
int main(){
    int choice;
    queue obj;// creating an Instance of a class named obj
    while(1){
        cout << "\n\n<===== MENU =====>" << endl;
        cout << "1.Enqueue" << endl;
        cout << "2.Dequeue" << endl;
        cout << "3.Peek" << endl;
        cout << "4.Exit" << endl;
        cout << "Select an option:";
        cin >> choice;
```

```cpp
        switch(choice){
            case 1:
                //Asking user for val to Enqueue in the Queue
                int val;
                cout << "Enter value to enqueue: ";
                cin >> val;
                obj.Enqueue(val);
                break;
            case 2:
                obj.Dequeue();
                break;
            case 3:
                obj.peek();
                break;
            case 4:
                cout << "Exiting...." << endl;
                return 0;
            default:
                cout << "The selected choice Cease to Exist!!Please Try Again" << endl;
        }
    }
}


//Defining The Member Functions

bool queue::isempty(){//Defining the Factor for boundary Condition
    if(front == nullptr){
        return true;
    }else{
        return false;
    }
}

void queue::Enqueue(int val){
    if(isempty()){//Boundary conditions
        struct node * newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = val;
        newnode->link = nullptr;
        front = newnode;
        rear = newnode;
    }else{
        struct node * newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = val;
        newnode->link = nullptr;
        struct node *temp = rear;
        temp->link = newnode;
        rear = newnode;
    }
    display();
}

void queue::display(){
    struct node *temp = front;
    cout << "QUEUE:)>    ";
    while(temp != nullptr){
        cout << temp->data << " | ";
        temp = temp->link;
    }
}

void queue::Dequeue(){
    if(isempty()){//Boundary conditions
        cout << "The queue is Empty! Enqueue and Try Again!!" << endl;
    }else{
        struct node *temp = front;
        front = temp->link;
        temp->link = nullptr;
        display();
    }
}

void queue::peek(){
    if(isempty()){//Boundary conditions
        cout << "The queue is Empty!" << endl;
    }else{
        struct node *temp = front;
        cout << "Front Element: " << temp->data << endl;
    }
}
```

7.4 String Plus Symbol

7.4.1 Question

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.

7.4.2 Algorithm

Algorithm 1 - RemovePlusSymbols

Input

1. s - string with characters and '+' symbols

Output

- Modified string after applying '+' operations

Steps

```
1. Initialize empty stack $s1$
2. For $i$ from 0 to $s.length() - 1$:
    1. If $s[i] =' +'$:
        1. Pop last character from $s1$
    2. Else:
        1. Push $s[i]$ onto $s1$
3. Initialize empty result string $result$
4. While $s1$ is not empty:
    1. Print character at top of $s1$
    2. Move to next node in stack
```

7.4.3 Code

main.cpp

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

string processString(string str) {
    deque<char> q;  // Queue ADT (deque used for back removal)

    for (char ch : str) {
        if (ch == '+') {
            if (!q.empty()) q.pop_back(); // Remove last inserted non-'+' element
        } else {
            q.push_back(ch); // Insert into queue
        }
    }

    // Build the final output string
    string result = "";
    while (!q.empty()) {
        result += q.front();
        q.pop_front();
    }
    return result;
}

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

    string output = processString(input);
    cout << "Output: " << output << endl;

    return 0;
}
```

8 Tree ADT

8.1 Tower

8.1.1 Question

There are n block towers, numbered from 1 to n. The i-th tower consists of a; blocks. In one move, you can move one block from tower i to tower j, but only if a; > a;. That move increases a; by 1 and decreases a; 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?

Input:

The first line contains a single integer t (1 <= t <= 10ˆ4) - the number of testcases. The first line of each testcase contains a single integer n (2 <= n <= 2 * 10ˆ5) — the number of towers. The second line contains n integers a1, a2, . . . , an (1 <= a <= 10ˆ9) — the number of blocks on each tower. The sum of n over all testcases doesn't exceed 2ˆ10°.

Output:

For each testcase, print the largest amount of blocks you can have on the tower 1 after you make any number of moves (possibly, zero).

8.1.2 Algorithm

Algorithm 1 - MaximizeTowerHeight

Input

1. tower[] - array of n integers representing tower heights

Output

- Maximum possible height of tower[0] after operations

Steps

```
1. For $k$ from 1 to $n-1$:
    1. While $tower[k] > tower[0]$:
        1. $tower[0] \leftarrow tower[0] + 1$
        2. $tower[k] \leftarrow tower[k] - 1$
2. Return $tower[0]$
```

8.1.3 Code

main.cpp

```cpp
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int solve(vector<int>& a, int size);

void sort(vector<int>& a, int size);

int main(){
    int t;
    cin >> t;
    while(t--){
        vector<int> arr;
        int n;
        cin >> n;
        int max = 0;
        for(int i = 0;i < n;i++){
            int e;
            cin >> e;
            arr.push_back(e);
        }
        sort(arr, arr.size());

        cout << solve(arr, n) << endl;
    }
}


int solve(vector<int>& a, int size){
    for(int i = 0;i < size; i++){
        while(a[0] < a[i]){
            a[0] = a[0] + 1;
            a[i] = a[i] - 1;
        }
    }
    int max_blocks = a[0];
    return max_blocks;
}

void sort(vector<int>& a, int size){
    sort(a+1, a.size() + a);
}
```

8.2 Character Binary Tree ADT

8.2.1 Question

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

8.2.2 Algorithm

Algorithm 1 - BinaryTreeInsert

Input

1. x - character to insert
2. root - root node of binary tree

Output

- Updated binary tree with x inserted

Steps

```
1. Create new node $temp$ with $data = x$, $left = null$, $right = null$
2. If $root = null$:
     1. $root \leftarrow temp$
3. Else:
     1. $p \leftarrow root$
     2. While $p.left \neq null$ AND $p.right \neq null$:
          1. If $p.left \neq null$:
               1. $p \leftarrow p.left$
          2. Else:
               1. $p \leftarrow p.right$
     3. If $p.left = null$:
          1. $p.left \leftarrow temp$
     4. Else:
          1. $p.right \leftarrow temp$
```

Algorithm 2 - InorderTraversal

Input

1. p - root node of binary tree or subtree

Output

- Inorder traversal sequence of the tree

Steps

```
1. If $p \neq null$:
    1. InorderTraversal($p.left$)
    2. Print $p.data$
    3. InorderTraversal($p.right$)
```

Algorithm 3 - PostorderTraversal

Input

1. p - root node of binary tree or subtree

Output

- Postorder traversal sequence of the tree

Steps

```
1. If $p \neq null$:
    1. PostorderTraversal($p.left$)
    2. PostorderTraversal($p.right$)
    3. Print $p.data$
```

Algorithm 4 - PreorderTraversal

Input

1. p - root node of binary tree or subtree

Output

- Preorder traversal sequence of the tree

Steps

```
1. If $p \neq null$:
    1. Print $p.data$
    2. PreorderTraversal($p.left$)
    3. PreorderTraversal($p.right$)
```

Algorithm 5 - TreeSearch

Input

1. x - character to search for
2. root - root node of binary tree

Output

- Status message indicating if x was found

Steps

```
1. $p \leftarrow root$
2. While $p \neq null$:
    1. If $p.data = x$:
        1. Print “Element found”
        2. Return
    2. Else:
        1. If $p.left \neq null$:
            1. $p \leftarrow p.left$
        2. Else:
            1. $p \leftarrow p.right$
3. Print “Element not found”
```

8.2.3 Code

main.cpp

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

class Tree {
    struct node {
        char data;
        struct node* left;
        struct node* right;

        node(char val) {
            data = val;
            left = right = nullptr;
        }
    };

public:
    node* root; // Root node of the tree

    Tree() { root = nullptr; } // Constructor

    void insert(char key);
    void inorder(node* root);
    void preorder(node* root);
    void postorder(node* root);
    bool search(char key);
    void menu();
};

void Tree::insert(char key) {
    node* newNode = new node(key);
    if (!root) {
        root = newNode;
        return;
    }

    queue<node*> q;
    q.push(root);

    while (!q.empty()) {
        node* temp = q.front();
        q.pop();

        if (!temp->left) {
            temp->left = newNode;
            return;
        } else {
            q.push(temp->left);
        }

        if (!temp->right) {
            temp->right = newNode;
            return;
        } else {
            q.push(temp->right);
        }
    }
}

void Tree::inorder(node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

void Tree::preorder(node* root) {
    if (!root) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void Tree::postorder(node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

bool Tree::search(char key) {
    if (!root) return false;

    queue<node*> q;
    q.push(root);

    while (!q.empty()) {
        node* temp = q.front();
        q.pop();

        if (temp->data == key)
            return true;

        if (temp->left)
            q.push(temp->left);
        if (temp->right)
            q.push(temp->right);
    }

    return false;
}

void Tree::menu() {
    int choice;
    char value;
    do {
        cout << "\n--- Binary Tree Menu ---\n";
        cout << "1. Insert a node\n";
        cout << "2. Inorder Traversal\n";
        cout << "3. Preorder Traversal\n";
        cout << "4. Postorder Traversal\n";
        cout << "5. Search for an element\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter a character to insert: ";
                cin >> value;
                insert(value);
                break;

            case 2:
                cout << "Inorder Traversal: ";
                inorder(root);
                cout << endl;
                break;

            case 3:
                cout << "Preorder Traversal: ";
                preorder(root);
                cout << endl;
                break;

            case 4:
                cout << "Postorder Traversal: ";
                postorder(root);
                cout << endl;
                break;

            case 5:
                cout << "Enter a character to search: ";
                cin >> value;
                if (search(value))
                    cout << value << " found in the tree.\n";
                else
                    cout << value << " not found in the tree.\n";
                break;

            case 6:
                cout << "Exiting program...\n";
                break;

            default:
```

```
                cout << "Invalid choice! Please enter a valid option.\n";
        }
    } while (choice != 6);
}

int main() {
    Tree tree;
    tree.menu(); // Start menu-driven program
    return 0;
}
```

9 Binary Search Tree ADT

9.1 Nenes Game

9.1.1 Question

Nene invented a new game based on an increasing sequence of integers a_1, a_2, . . . , a_k. In this game, initially, n players are lined up in a row. In each of the rounds of this game, the following happens:

Nene finds the a_1-th, a_2-th, . . . , a_k-th players in the row. They are kicked out of the game simultaneously. If the i-th player in the row should be kicked out, but there are fewer than i players in the 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] and n = 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 3rd and the 5th players in the 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 3rd and the 5th players in the row. The 3rd player is player D and there is no 5th player in the row. Thus, only player D is kicked out in the second round. Now players are lined up as AB. In the third round, Nene finds the 3rd and 5th players. There are none. No one is kicked out of the game, so the game ends after this round. Players A and B are declared as the winners.

Nene has not yet decided how many people would join the game initially. Nene gave you q integers n_1, n_2, . . . , n_q and you should answer the following question for each 1 <= i <= q independently: "How many people would be declared as winners if there are n_i players in the game initially?"

Input:

Each test contains multiple test cases. The first line contains the number of test cases t (1 <= t <= 250). The description of test cases follows.

The first line of each test case contains two integers k and q (1 <= k, q <= 100) ̆2014 the length of the sequence a and the number of values n_i you should solve this problem for.

The second line contains k integers a_1, a_2, . . . , a_k (1 <= a_1 < a_2 < . . . < a_k <= 100) ̆2014 the sequence a.

The third line contains q integers n_1, n_2, . . . , n_q (1 <= n_i <= 100).

Output:

For each test case, output q integers: the i-th (1 <= i <= q) of them should be the number of players declared as winners if initially n_i players join the game.

(Self-correction note: I initially assumed the example typo ARD should be ABD based on the rules and fixed it. Also corrected variable inconsistencies like g vs q and index notation.)

9.1.2 Algorithm

Algorithm 1 - NenesGameSimulation

Input

1. a[] - array of k integers representing Nene's moves
2. b[] - array of q integers representing initial pile sizes

Output

- Final pile sizes after all possible moves

Steps

```
1. For each query $i$ from 0 to $q-1$:
    1. Create vector $c$ of size $b[i]$ with all zeros
    2. Set $flag \leftarrow 1$
    3. While $flag = 1$:
        1. $flag \leftarrow 0$
        2. For $j$ from $k-1$ down to 0:
            1. If $a[j] \leq c.size()$:
                1. Erase element at position $(a[j]-1)$ from vector $c$
                2. $flag \leftarrow 1$
    4. Output the final size of vector $c$
```

9.1.3 Code

main.cpp

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


int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    cin >> t;
    while(t--){
        int k;
        int q;
        cin >> k >> q;
        vector<int> a;
        vector<int> n;
        for(int i = 0;i < k;i++){
            int e;
            cin >> e;
            a.push_back(e);
        }

        for(int i = 0;i < q;i++){
            int e;
            cin >> e;
            n.push_back(e);
        }

        int min = a[0];

        for(int num : n){
            vector<int> temp;
            for(int i = 0;i < num;i++){
                temp.push_back(i);
            }
            int count = 0;
            for(int i = 0;i < a[0] && i <= temp.size();i++){
                count++;
            }
            cout << count - 1<< " ";
        }
        cout << endl;
    }
}
```

9.2 Advantages Game

9.2.1 Question

There are n participants in a competition, participant i having a strength of s_i.

Every participant wonders how much of an advantage they have over the next strongest participant (excluding themselves). In other words, each participant i wants to know the difference between their strength s_i and the maximum strength s_j among all other participants (j != i). Note that this difference can be negative.

So, they ask you for your help! For each i (from 1 to n), calculate and output the difference s_i - max(s_j for j != i).

Input:

The input consists of multiple test cases. The first line contains an integer t (1 <= t <= 1000) 2̆014 the number of test cases. The descriptions of the test cases follow.

The first line of each test case contains an integer n (2 <= n <= 2 * 10ˆ5) 2̆014 the number of participants (length of the array).

The following line contains n space-separated positive integers s_1, s_2, . . . , s_n (1 <= s_i <= 10ˆ9) 2̆014 the strengths of the participants.

It is guaranteed that the sum of n over all test cases does not exceed 2 * 10ˆ5.

Output:

For each test case, output n space-separated integers. For each i (1 <= i <= n), output the difference between s_i and the maximum strength of any other participant.

9.2.2 Algorithm

Algorithm 1 - CalculateAdvantages

Input

1. nums[] - array of n integers

Output

- advantages[] - array of n integers representing the advantages

Steps

```
1. Create a copy $max[] \leftarrow nums[]$
2. Sort $max[]$ in descending order using bubble sort:
    1. For $i$ from 0 to $n-2$:
        1. For $j$ from 0 to $n-i-2$:
            1. If $max[j] < max[j+1]$:
                1. Swap $max[j]$ and $max[j+1]$
3. For each element $nums[i]$ from 0 to $n-1$:
    1. $dif \leftarrow nums[i] - max[0]$
    2. If $dif = 0$ (current element is the maximum):
        1. $dif \leftarrow nums[i] - max[1]$ (calculate advantage over second maximum)
    3. Store $dif$ as the advantage for element $nums[i]$
4. Return array of advantages
```

9.2.3 Code

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<int> arr(n);
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        // Find the first and second maximum
        int max1 = -1, max2 = -1;
        for (int num : arr) {
            if (num > max1) {
                max2 = max1;
                max1 = num;
            } else if (num > max2) {
                max2 = num;
            }
        }

        for (int i = 0; i < n; i++) {
            if (arr[i] == max1) {
                cout << arr[i] - max2 << " ";
            } else {
                cout << arr[i] - max1 << " ";
            }
        }
        cout << "\n";
    }
    return 0;
}
```

9.3 03 BST Implementation

9.3.1 Question

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

9.3.2 Algorithm

Algorithm 1 - BSTInsert

Input

1. root - root of binary search tree
2. num - value to insert

Output

- Updated binary search tree with num inserted

Steps

```
1. If $root = null$:
    1. Create new node $newnode$
    2. $newnode.data \leftarrow num$
    3. $newnode.left \leftarrow null$
    4. $newnode.right \leftarrow null$
    5. Return $newnode$
2. If $num < root.data$:
    1. $root.left \leftarrow insert(root.left, num)$
3. Else if $num > root.data$:
    1. $root.right \leftarrow insert(root.right, num)$
```

```
4. Return $root$
```

Algorithm 2 - BSTSearch

Input

1. root - root of binary search tree
2. num - value to search for

Output

- Boolean indicating whether num is found in the tree

Steps

```
1. $p \leftarrow root$
2. While $p \neq null$:
    1. If $p.data = num$:
        1. Print "Element found"
        2. Return
    2. Else if $num < p.data$:
        1. $p \leftarrow p.left$
    3. Else:
        1. $p \leftarrow p.right$
3. Print "Element not found"
```

Algorithm 3 - InorderTraversal

Input

1. root - root of binary search tree

Output

- Inorder traversal of the tree (sorted values)

Steps

```
1. If $root \neq null$:
    1. InorderTraversal($root.left$)
    2. Print $root.data$
    3. InorderTraversal($root.right$)
```

Algorithm 4 - PreorderTraversal

Input

1. root - root of binary search tree

Output

- Preorder traversal of the tree

Steps

```
1. If $root \neq null$:
    1. Print $root.data$
    2. PreorderTraversal($root.left$)
    3. PreorderTraversal($root.right$)
```

Algorithm 5 - PostorderTraversal

Input

1. root - root of binary search tree

Output

- Postorder traversal of the tree

Steps

```
1. If $root \neq null$:
    1. PostorderTraversal($root.left$)
    2. PostorderTraversal($root.right$)
    3. Print $root.data$
```

9.3.3 Code

main.cpp

```
/*C. Write a separate C++ menu-driven program to implement Tree ADT using a binary search t

Insert

Preorder

Inorder

Postorder

Search

Exit

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

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


class tree{
    private:
        struct node{
            int data;
            node* left;
            node* right;
        }*root = nullptr;

        node* insert(node* newroot, int value);
        node* createnewnode(int val);
        void preorder(node* current);
        void inorder(node* current);
        void postorder(node* current);

    public:
        tree(){
            root = nullptr;
        }
        void inserthelper(int val);
        void preorderhelper();
        void inorderhelper();
        void postorderhelper();
        void search(int value);


};




int main(){
    tree adt;
    int choice;
    while(1){
        cout << "\n<===== MENU =====>" << endl;
        cout << "1.Insert" << endl;
        cout << "2.Preorder" << endl;
        cout << "3.Inorder" << endl;
        cout << "4.Postorder" << endl;
        cout << "5.Search" << endl;
```

```
            cout << "6.Exit" << endl;
            cout << "Enter your choice: ";
            cin >> choice;

            switch(choice){
                case 1:
                    int value;
                    cout << "Enter the value to insert: ";
                    cin >> value;
                    adt.inserthelper(value);
                    break;
                case 2:
                    adt.preorderhelper();
                    break;
                case 3:
                    adt.inorderhelper();
                    break;
                case 4:
                    adt.postorderhelper();
                    break;
                case 5:
                    int v;
                    cout << "Enter the value you want to search: ";
                    cin >> v;
                    adt.search(v);
                    break;
                case 6:
                    cout << "Exiting...." << endl;
                    return 0;
                default:
                    cout << "The selected choice cease to Exist\nPlease Try Again" << endl;
            }
        }
}




tree::node* tree::insert(node* newroot, int value){
    if(newroot == nullptr){
        newroot = createnewnode(value);
        return newroot;
    }


    if(value < newroot->data){
        newroot->left = insert(newroot->left, value);
    }else{
        newroot->right = insert(newroot->right, value);
    }

    return newroot;
}

void tree::inserthelper(int val){
    root = insert(root, val);
}


tree::node* tree::createnewnode(int val){
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = val;
    newnode->left = newnode->right = nullptr;
    return newnode;
}


void tree::preorder(node* current){
    if(current != nullptr){
        cout << current->data << " ";
        preorder(current->left);
        preorder(current->right);
    }
}

void tree::preorderhelper(){
    cout << "Preorder Traversal: ";
    preorder(root);
    cout << endl;
}

void tree::inorder(node* current){
    if(current != nullptr){
        inorder(current->left);
        cout << current->data << " ";
        inorder(current->right);
    }
}

void tree::inorderhelper(){
    cout << "Inorder Traversal: ";
    inorder(root);
    cout << endl;
}


void tree::postorder(node* current){
    if(current != nullptr){
        postorder(current->left);
        postorder(current->right);
        cout << current->data << " ";
    }
}



void tree::postorderhelper(){
    cout << "Postorder Traversal: ";
    postorder(root);
    cout << endl;
}


void tree::search(int value){
    struct node* temp = root;
    int flag = 0;
    while(temp != nullptr){
        if(value == temp->data){
            flag = 1;
            break;
        }else{
            if(value < temp->data){
                temp = temp->left;
            }else if(value > temp->data){
                temp = temp->right;
            }
        }
    }

    if(flag){
        cout << value << " is found in the tree" << endl;
    }else{
        cout << value << " is not found in the tree" << endl;
    }

}
```

9.4 Expression Tree

9.4.1 Question

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

9.4.2 Algorithm

Algorithm 1 - CreateExpressionTree

Input

1. postfix - String containing postfix expression

Output

- Root node of the expression tree

Steps

```
1. Initialize empty stack $s$
2. For each character $c$ in postfix expression:
    1. If $c$ is a digit (operand):
        1. Create new node $newnode$ with $data = c$
        2. $newnode.left \leftarrow null$
        3. $newnode.right \leftarrow null$
        4. Push $newnode$ onto stack $s$
    2. Else ($c$ is an operator):
        1. Create new node $newnode$ with $data = c$
        2. $newnode.right \leftarrow s.top()$
        3. Pop element from stack $s$
        4. $newnode.left \leftarrow s.top()$
        5. Pop element from stack $s$
        6. Push $newnode$ onto stack $s$
3. $root \leftarrow s.top()$
4. Return $root$
```

Algorithm 2 - InorderTraversal

Input

1. root - Root node of the expression tree

Output

- Inorder representation of the expression tree

Steps

```
1. If $root \neq null$:
    1. InorderTraversal($root.left$)
    2. Print $root.data$
    3. InorderTraversal($root.right$)
```

Algorithm 3 - PreorderTraversal

Input

1. root - Root node of the expression tree

Output

- Preorder representation of the expression tree

Steps

```
1. If $root \neq null$:
    1. Print $root.data$
    2. PreorderTraversal($root.left$)
    3. PreorderTraversal($root.right$)
```

Algorithm 4 - PostorderTraversal

Input

1. root - Root node of the expression tree

Output

- Postorder representation of the expression tree

Steps

```
1. If $root \neq null$:
    1. PostorderTraversal($root.left$)
    2. PostorderTraversal($root.right$)
    3. Print $root.data$
```

9.4.3 Code

main.cpp

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

// Expression Tree class extends the binary tree functionality
class ExpressionTree {
private:
  struct Node {
    string data; // Value stored in the node (operand or operator)
    Node *left;  // Pointer to left child
    Node *right; // Pointer to right child
  } *root;

  // Helper methods
  void preorder(Node *current);
  void inorder(Node *current);
  void postorder(Node *current);
  bool isOperator(const string &c);

public:
  ExpressionTree() { root = nullptr; }
  ~ExpressionTree();

  // Deallocates the tree recursively
  void deleteTree(Node *node);

  // Constructs an expression tree from a postfix expression
  void constructTree(const string &postfix);

  // Public traversal methods
  void preorderTraversal();
  void inorderTraversal();
  void postorderTraversal();
```

```
};

ExpressionTree::~ExpressionTree() { deleteTree(root); }

void ExpressionTree::deleteTree(Node *node) {
  if (node) {
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
  }
}

bool ExpressionTree::isOperator(const string &c) {
  return c == "+" || c == "-" || c == "*" || c == "/" || c == "^";
}

// Constructs expression tree from postfix expression
// Time Complexity: O(n) where n is the length of the postfix expression
void ExpressionTree::constructTree(const string &postfix) {
  stack<Node *> st;
  Node *t1, *t2, *temp;

  // Process each token in the postfix expression
  for (size_t i = 0; i < postfix.length(); i++) {
    if (postfix[i] == ' ')
      continue;

    // Extract token (operand or operator)
    string token;
    if (isdigit(postfix[i])) {
      // If it's a digit, extract the complete number
      while (i < postfix.length() &&
             (isdigit(postfix[i]) || postfix[i] == '.')) {
        token += postfix[i];
        i++;
      }
      i--; // Adjust for the loop increment
    } else {
      token = postfix[i];
    }

    // Create a new node with this token
    temp = new Node;
    temp->data = token;
    temp->left = temp->right = nullptr;
    if (!isOperator(token)) {
      // If operand, push to stack
      st.push(temp);
    } else {
      // If operator, pop two nodes from stack
      t1 = st.top();
      st.pop(); // First operand (right child)
      t2 = st.top();
      st.pop(); // Second operand (left child)

      // Make them children of the operator node
      temp->right = t1;
      temp->left = t2;

      // Push the operator node back to stack
      st.push(temp);
    }
  }

  // The final node in stack is the root
  root = st.top();
  st.pop();
}

// Preorder traversal: Root-Left-Right
// Time Complexity: O(n) where n is number of nodes
void ExpressionTree::preorder(Node *current) {
  if (current) {
    cout << current->data << " ";
    preorder(current->left);
    preorder(current->right);
  }
}

void ExpressionTree::preorderTraversal() {
  cout << "Preorder: ";
  preorder(root);
  cout << endl;
}

// Inorder traversal: Left-Root-Right
// For expression trees, need to add parentheses to preserve operator precedence
// Time Complexity: O(n) where n is number of nodes
void ExpressionTree::inorder(Node *current) {
  if (current) {
    if (isOperator(current->data))
```

```
      cout << "(";

    inorder(current->left);
    cout << current->data << " ";
    inorder(current->right);

    if (isOperator(current->data))
      cout << ")";
  }
}

void ExpressionTree::inorderTraversal() {
  cout << "Inorder: ";
  inorder(root);
  cout << endl;
}

// Postorder traversal: Left-Right-Root
// Time Complexity: O(n) where n is number of nodes
void ExpressionTree::postorder(Node *current) {
  if (current) {
    postorder(current->left);
    postorder(current->right);
    cout << current->data << " ";
  }
}

void ExpressionTree::postorderTraversal() {
  cout << "Postorder: ";
  postorder(root);
  cout << endl;
}

int main() {
  ExpressionTree tree;
  string postfix;
  int choice;

  while (true) {
    cout << "\n<===== EXPRESSION TREE MENU =====>" << endl;
    cout << "1. Postfix Expression" << endl;
    cout << "2. Construct Expression Tree" << endl;
    cout << "3. Preorder" << endl;
    cout << "4. Inorder" << endl;
    cout << "5. Postorder" << endl;
    cout << "6. Exit" << endl;
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
    case 1:
      cin.ignore(); // Clear the input buffer
      cout << "Enter a postfix expression (separate tokens with spaces): ";
      getline(cin, postfix);
      break;

    case 2:
      cout << "Constructing expression tree from \"" << postfix << "\"" << endl;
      tree.constructTree(postfix);
      cout << "Expression tree constructed successfully!" << endl;
      break;

    case 3:
      tree.preorderTraversal();
      break;

    case 4:
      tree.inorderTraversal();
      break;

    case 5:
      tree.postorderTraversal();
      break;

    case 6:
      cout << "Exiting..." << endl;
      return 0;

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

  return 0;
}
```

10 Priority Queue ADT

10.1 Polycarp Numbers

10.1.1 Question

Polycarp was presented with some sequence of integers a of length n. A sequence can make Polycarp happy only if it consists of different numbers (i.e., distinct numbers).

In order to make his sequence like this, Polycarp is going to make some (possibly zero) number of moves. In one move, he can:

remove the first (leftmost) element of the sequence.

For example, in one move, the sequence [3, 1, 4, 3] will produce the sequence [1, 4, 3], which consists of different numbers.

Determine the minimum number of moves he needs to make so that in the remaining sequence all elements are different. In other words, find the length of the smallest prefix of the given sequence a, after removing which all values in the remaining sequence will be unique.

Input:

The first line of the input contains a single integer t (1 <= t <= 10ˆ4) ˇ2014 the number of test cases.

Each test case consists of two lines. The first line contains an integer n (1 <= n <= 2 * 10ˆ5) ˇ2014 the length of the given sequence a. The second line contains n integers a_1, a_2, . . . , a_n (1 <= a_i <= n) ˇ2014 elements of the given sequence a.

It is guaranteed that the sum of n values over all test cases does not exceed 2 * 10ˆ5.

Output:

For each test case print your answer on a separate line ˇ2014 the minimum number of elements that must be removed from the beginning of the sequence so that all remaining elements are different.

10.1.2 Algorithm

Algorithm 1 - MinimumElementsToRemove

Input

1. a[] - array of n integers

Output

- k - minimum number of elements to remove from beginning

Steps

```
1. Initialize unordered_set $b \leftarrow \emptyset$
2. For $i \leftarrow n - 1$ down to 0:
    1. If $a[i] \notin b$:
        1. $b \leftarrow b \cup \{a[i]\}$
    2. Else:
        1. Break loop
3. Return $n - |b|$
```

10.1.3 Code

main.cpp

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

int main(){
    int t;
    cin >> t;
    while(t--){
        int n;
        cin >> n;
        vector<int> a;
        for(int i = 0;i < n;i++){
            int e;
            cin >> e;
            a.push_back(e);
        }

        unordered_set<int> seen;
        int dups_RtoL = 0;
        for(int i = n - 1;i >= 0;i--){
            if(seen.find(a[i]) == seen.end()){
                dups_RtoL++;
                seen.insert(a[i]);
            }else{
                break;
            }
        }

        cout << n - dups_RtoL << endl;
    }
}
```

10.2 Word Game

10.2.1 Question

Three guys play a game: first, each person writes down n distinct words of length 3. Then, they total up the number of points as follows:

If a word was written by one person ˇ2014 that person gets 3 points.

If a word was written by two people ˇ2014 each of the two gets 1 point.

If a word was written by all three ˇ2014 nobody gets any points.

In the end, how many points does each player have?

Input:

The input consists of multiple test cases. The first line contains an integer t (1 <= t <= 100) ˇ2014 the number of test cases. The description of the test cases follows.

The first line of each test case contains an integer n (1 <= n <= 1000) ˇ2014 the number of words written by each person.

The following three lines each contain n distinct strings ˇ2014 the words written by each person. Each string consists of 3 lowercase English characters.

Output:

For each test case, output three space-separated integers ˇ2014 the number of points each of the three guys earned. You should output the answers in the same order as the input; the i-th integer should be the number of points earned by the i-th guy.

10.2.2 Algorithm

Algorithm 1 - WordGameScoring

Input

1. a[][] - 3×n matrix of strings where a[i] represents words written by player i

Output

- p[] - array of 3 integers representing points of each player

Steps

```
1. Initialize score array $p[0...2] \leftarrow 0$
2. For player 0's words (i from 0 to $n-1$):
    1. If word $a[0][i]$ appears in player 1's list but not in player 2's list:
        1. $p[0] \leftarrow p[0] + 1$
        2. $p[1] \leftarrow p[1] + 1$
    2. Else if word $a[0][i]$ appears in player 2's list but not in player 1's list:
        1. $p[0] \leftarrow p[0] + 1$
        2. $p[2] \leftarrow p[2] + 1$
    3. Else if word $a[0][i]$ appears in neither player 1's nor player 2's list:
        1. $p[0] \leftarrow p[0] + 3$
3. For player 1's words (i from 0 to $n-1$):
    1. If word $a[1][i]$ appears in player 2's list but not in player 0's list:
        1. $p[1] \leftarrow p[1] + 1$
        2. $p[2] \leftarrow p[2] + 1$
    2. Else if word $a[1][i]$ appears in neither player 0's nor player 2's list:
        1. $p[1] \leftarrow p[1] + 3$
4. For player 2's words (i from 0 to $n-1$):
    1. If word $a[2][i]$ appears in neither player 0's nor player 1's list:
        1. $p[2] \leftarrow p[2] + 3$
5. Return array $p$
```

10.2.3 Code

main.cpp

```cpp
#include<iostream>
#include<map>
#include<vector>
using namespace std;
int main()
{
    int t;
    cout<<"enter hte number of testcases\n";
    cin>>t;
    while(t--)
    {
        int n;
        cout<<"enter the number of 3 letter strings\n";
        cin>>n;
        vector<vector<string>>words(3,vector<string>(n));
        map<string,int>wordcount;
        cout<<"enter the words\n";
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<n; j++)
            {
                cin>>words[i][j];
                wordcount[words[i][j]]++;
            }
        }
        vector<int>scores(3,0);
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<n; j++)
            {
                string word=words[i][j];
                if( wordcount[word]==1)
                {
                    scores[i]+=3;
                }
                else if(wordcount[word]==2)
                {
                    scores[i]+=1;
                }
            }
        }
        cout<<"their scores are\n";
        cout<<scores[0]<<" "<<scores[1]<<" "<<scores[2]<<"\n";
    }
    return 0;
}
```

10.3 Priority Queue ADT Implementation

10.3.1 Question

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
3. Display
4. Search
5. Sort (Heap Sort)
6. Exit

10.3.2 Algorithm

Algorithm 1 - MaxHeapInsert

Input

1. x - element to insert
2. array[] - heap represented as an array
3. n - current size of the heap

Output

- Updated heap with new element inserted

Steps

```
1. If $array$ is empty:
    1. $array.push\_back(x)$
2. Else:
    1. $array.push\_back(x)$
    2. $i \leftarrow array.size() - 1$
    3. While $((i + 1)/2 - 1) \geq 0$:
        1. If $array[i] > array[((i + 1)/2 - 1)]$:
            1. Swap $array[i]$ and $array[((i + 1)/2 - 1)]$
            2. $i \leftarrow (i + 1)/2 - 1$
        2. Else:
            1. Break loop
```

Algorithm 2 - MaxHeapDeleteRoot

Input

1. array[] - heap represented as an array
2. n - current size of the heap

Output

- Updated heap with root element removed

Steps

```
1. If $array$ is empty:
    1. Print “The Queue is empty”
2. Else:
    1. $array[0] \leftarrow array.back()$
    2. $array.pop\_back()$
    3. $i \leftarrow 0$
    4. While $2i + 1 < array.size()$:
```

```
1. $j \leftarrow 2i + 1$
2. If $j + 1 < array.size()$:
    1. If $array[i] < array[j]$ and $array[i] < array[j + 1]$:
        1. If $array[j] > array[j + 1]$:
            1. Swap $array[i]$ and $array[j]$
            2. $i \leftarrow j$
        2. Else:
            1. Swap $array[i]$ and $array[j + 1]$
            2. $i \leftarrow j + 1$
    2. Else if $array[i] < array[j]$:
        1. Swap $array[i]$ and $array[j]$
        2. $i \leftarrow j$
    3. Else if $array[i] < array[j + 1]$:
        1. Swap $array[i]$ and $array[j + 1]$
        2. $i \leftarrow j + 1$
    4. Else:
        1. Break loop
3. Else:
    1. If $array[i] < array[j]$:
        1. Swap $array[i]$ and $array[j]$
        2. $i \leftarrow j$
    2. Else:
        1. Break loop
```

Algorithm 3 - HeapSort

Input

1. array[] - heap represented as an array
2. n - current size of the heap

Output

- Sorted array in descending order

Steps

```
1. Initialize empty vector $sorted$
2. While $array$ is not empty:
    1. $sorted.push\_back(array[0])$
    2. Call $delete\_root()$ to remove the maximum element
3. $array \leftarrow sorted$
```

10.3.3 Code

main.cpp

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

class PriorityQueue {
    int *heap;
    int capacity;
    int size;

public:
    PriorityQueue(int cap) {
        capacity = cap;
        heap = new int[capacity];
        size = 0;
    }

    ~PriorityQueue() {
        delete[] heap;
    }

    int parent(int i) { return (i - 1) / 2; }
    int left(int i) { return 2 * i + 1; }
    int right(int i) { return 2 * i + 2; }

    void swap(int &x, int &y) {
        int temp = x;
        x = y;
        y = temp;
    }

    void insert(int key) {
        if (size == capacity) {
            cout << "Priority Queue is full\n";
            return;
        }

        heap[size] = key;
        int i = size;
        size++;

        // Heapify Up
        while (i != 0 && heap[parent(i)] < heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }
```

```
void deleteMax() {
    if (size <= 0) {
        cout << "Priority Queue is empty\n";
        return;
    }

    cout << "Deleted Max: " << heap[0] << endl;
    heap[0] = heap[size - 1];
    size--;
    heapifyDown(0);
}

void heapifyDown(int i) {
    int largest = i;
    int l = left(i);
    int r = right(i);

    if (l < size && heap[l] > heap[largest])
        largest = l;
    if (r < size && heap[r] > heap[largest])
        largest = r;

    if (largest != i) {
        swap(heap[i], heap[largest]);
        heapifyDown(largest);
    }
}

void display() {
    if (size == 0) {
        cout << "Priority Queue is empty\n";
        return;
    }

    cout << "Priority Queue (Heap): ";
    for (int i = 0; i < size; i++)
        cout << heap[i] << " ";
    cout << endl;
}

void heapSort() {
    int* temp = new int[size];
    int tempSize = size;

    for (int i = 0; i < tempSize; i++) {
            temp[i] = heap[0];
            heap[0] = heap[size - 1];
            size--;
            heapifyDown(0);
        }

        cout << "Heap Sorted Order (Descending): ";
        for (int i = 0; i < tempSize; i++)
            cout << temp[i] << " ";
        cout << endl;

        // Restore original heap
        delete[] heap;
        heap = new int[capacity];
        size = 0;
        for (int i = 0; i < tempSize; i++)
            insert(temp[i]);

        delete[] temp;
    }
};

// Driver code
int main() {
    PriorityQueue pq(100);

    pq.insert(40);
    pq.insert(20);
    pq.insert(60);
    pq.insert(30);
    pq.insert(10);

    pq.display();

    pq.deleteMax();
    pq.display();

    pq.insert(70);
    pq.display();

    pq.heapSort();
    pq.display();

    return 0;
}
```

11 Hash Map

11.1 Linear Probing

11.1.1 Question

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

11.1.2 Algorithm

Algorithm 1 - LinearProbingInsert

Input

1. num - value to insert
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Updated hash table with num inserted (if possible)

Steps

```
1. $index \leftarrow num \bmod SIZE$
2. If $table[index] = EMPTY$:
    1. $table[index] \leftarrow num$
3. Else:
    1. $temp \leftarrow index$
    2. While $table[index] \neq EMPTY$:
        1. $index \leftarrow (index + 1) \bmod SIZE$
        2. If $index = temp$:
            1. Print "Table is full"
            2. Return
    3. $table[index] \leftarrow num$
```

Algorithm 2 - LinearProbingDelete

Input

1. num - value to delete
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Updated hash table with num removed (if found)

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. If $table[i] = num$:
        1. $table[i] \leftarrow EMPTY$
        2. Return
2. Print "Element not found"
```

Algorithm 3 - LinearProbingSearch

Input

1. num - value to search for
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Status message indicating if num was found or not

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. If $table[i] = num$:
        1. Print "The element is present at index $i$"
        2. Return
2. Print "The element is not present"
```

Algorithm 4 - LinearProbingDisplay

Input

1. table[] - hash table array
2. SIZE - size of hash table

Output

- Visual representation of the hash table

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. If $table[i] = EMPTY$:
        1. Print "$i$ -> EMPTY"
    2. Else:
        1. Print "$i$ -> $table[i]$"
```

11.1.3 Code

main.cpp

```cpp
/* menu-driven C++ program to implement a Hash ADT using Linear Probing, followed by the ti
with operations:
    1.Insert
    2.Delete
    3.Search
    4.Display
    5.Exit*/


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

    const int SIZE = 10;



    class hashtable{
        private:
            struct slot{
                int value;
                bool isoccupied;
                bool isdeleted;
            };

            slot table[SIZE];

            int hashfunction(int key){
                return key % SIZE;
            }

        public:
            hashtable(){
                for(int i = 0;i < SIZE;i++){
```

```
                table[i].isoccupied = false;
                table[i].isdeleted = false;
            }
        }

        void insert(int val);
        void display();
        void remove(int key);
        void search(int key);

};




int main(){
    int choice;
    int val;
    hashtable obj;
    while(1){
        cout << "\n<======== MENU ========>" << endl;
        cout << "1.Insert " << endl;
        cout << "2.Delete" << endl;
        cout << "3.Search" << endl;
        cout << "4.Display" << endl;
        cout << "5.Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                cout << "Enter a value to insert: ";
                cin >> val;
                obj.insert(val);
                break;
            case 2:
                cout << "Enter a value to remove: ";
                cin >> val;
                obj.remove(val);
                break;
            case 3:
                cout << "Enter the value to search: ";
                cin >> val;
                obj.search(val);
                break;
            case 4:
                obj.display();
                break;
            case 5:
                cout << "Exiting...." << endl;
                return 0;
            default:
                cout << "Selected choice cease to exist\nPlease Try Again" << endl;
        }
    }
}



void hashtable::insert(int key){
    int index = hashfunction(key);
    int start = index;
    while(table[index].isoccupied || table[index].isdeleted){
        index = (index + 1) % SIZE;
        if(index == start){
            cout << "The hash table is full" << endl;
            return;
        }
    }
    table[index].value = key;
    table[index].isoccupied = true;
    table[index].isdeleted = false;
    cout << "Inserted " << key << " at index " << index << endl;
}


void hashtable::display(){
    cout << "Hashtable: " << endl;
    for(int i = 0;i < SIZE;i++){
        if(table[i].isoccupied == true && table[i].isdeleted == false){
            cout << i << " ---> " << table[i].value << endl;
        }else{
            cout << i << " ---> " << endl;
        }
    }
}

void hashtable::remove(int key){
    int index = hashfunction(key);
    int start = index;

    while(table[index].isdeleted || table[index].isoccupied){
        if(key == table[index].value){
            table[index].isoccupied = false;
            table[index].isdeleted = true;
            cout << "Deleted" << key << " at index " << index << endl;
            return;
        }
        index = (index + 1)%SIZE;
        if(index == start){
            cout << "The hashtable is Empty" << endl;
            return;
        }
    }
}


void hashtable::search(int key){
    int index = hashfunction(key);
    int start = index;

    while(table[index].isoccupied == true){
        if(key == table[index].value){
            cout << "Found Element " << key << " at index " << index << endl;
            return;
        }
        index = (index + 1)%SIZE;
        if(index == start){
            break;
        }
    }
    cout << "The element " << key << " not found !" << endl;
}
```

11.2 Quadratic Probing

11.2.1 Question

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

11.2.2 Algorithm

Algorithm 1 - QuadraticProbingInsert

Input

1. num - value to insert
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Updated hash table with num inserted (if possible)

Steps

```
1. $index \leftarrow num \bmod SIZE$
2. If $table[index] = EMPTY$:
   1. $table[index] \leftarrow num$
3. Else:
   1. $temp \leftarrow index$
   2. $col \leftarrow 1$
   3. While $table[index] \neq EMPTY$:
      1. $index \leftarrow (index + col + col^2) \bmod SIZE$
      2. $col \leftarrow col + 1$
      3. If $index = temp$:
         1. Print “Table is full”
         2. Return
   4. $table[index] \leftarrow num$
```

Algorithm 2 - QuadraticProbingDelete

Input

1. num - value to delete
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Updated hash table with num removed (if found)

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
   1. If $table[i] = num$:
      1. $table[i] \leftarrow EMPTY$
      2. Return
```

```
2. Print “Element not found”
```

Algorithm 3 - QuadraticProbingSearch

Input

1. num - value to search for
2. table[] - hash table array
3. SIZE - size of hash table

Output

- Status message indicating if num was found or not

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. If $table[i] = num$:
        1. Print “The element is present at index $i$”
        2. Return
2. Print “The element is not present”
```

Algorithm 4 - QuadraticProbingDisplay

Input

1. table[] - hash table array
2. SIZE - size of hash table

Output

- Visual representation of the hash table

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. If $table[i] = EMPTY$:
        1. Print “$i$ -> EMPTY”
    2. Else:
        1. Print “$i$ -> $table[i]$”
```

11.2.3 Code

main.cpp

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

```cpp
const int TABLE_SIZE = 10;
const int EMPTY = -1;
const int DELETED = -2;

class HashTable {
private:
    int table[TABLE_SIZE];
    int hashFunc(int key);  // O(1)

public:
    HashTable();            // O(n)
    void insert(int key);   // Average: O(1), Worst: O(n)
    void remove(int key);   // Average: O(1), Worst: O(n)
    void search(int key);   // Average: O(1), Worst: O(n)
    void display();         // O(n)
};

// ===== MAIN MENU =====
int main() {
    HashTable ht;
    int choice, key;

    do {
        cout << "\n=== Hash Table Menu (Quadratic Probing) ===\n";
        cout << "1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter key to insert: ";
                cin >> key;
                ht.insert(key);
                break;
            case 2:
                cout << "Enter key to delete: ";
                cin >> key;
                ht.remove(key);
                break;
            case 3:
                cout << "Enter key to search: ";
                cin >> key;
                ht.search(key);
                break;
            case 4:
                ht.display();
                break;
            case 5:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 5);

    return 0;
}

HashTable::HashTable() {    // O(n)
    for (int i = 0; i < TABLE_SIZE; i++)
        table[i] = EMPTY;
}

int HashTable::hashFunc(int key) {  // O(1)
    return key % TABLE_SIZE;
}

void HashTable::insert(int key) {   // Average: O(1), Worst: O(n)
    int index = hashFunc(key);
    int i = 0;

    while (table[(index + i * i) % TABLE_SIZE] != EMPTY &&
           table[(index + i * i) % TABLE_SIZE] != DELETED &&
           table[(index + i * i) % TABLE_SIZE] != key) {
        i++;
        if (i == TABLE_SIZE) {
            cout << "Hash table is full. Cannot insert.\n";
            return;
        }
    }

    int newIndex = (index + i * i) % TABLE_SIZE;
    if (table[newIndex] == key) {
        cout << "Duplicate key. Already exists.\n";
    } else {
        table[newIndex] = key;
        cout << "Inserted key " << key << " at index " << newIndex << ".\n";
    }
}

void HashTable::remove(int key) {   // Average: O(1), Worst: O(n)
```

```
    int index = hashFunc(key);
    int i = 0;

    while (table[(index + i * i) % TABLE_SIZE] != EMPTY) {
        int probeIndex = (index + i * i) % TABLE_SIZE;

        if (table[probeIndex] == key) {
            table[probeIndex] = DELETED;
            cout << "Key " << key << " deleted from index " << probeIndex << ".\n";
            return;
        }
        i++;
        if (i == TABLE_SIZE) break;
    }

    cout << "Key " << key << " not found.\n";
}

void HashTable::search(int key) {   // Average: O(1), Worst: O(n)
    int index = hashFunc(key);
    int i = 0;

    while (table[(index + i * i) % TABLE_SIZE] != EMPTY) {
        int probeIndex = (index + i * i) % TABLE_SIZE;

        if (table[probeIndex] == key) {
            cout << "Key " << key << " found at index " << probeIndex << ".\n";
            return;
        }
        i++;
        if (i == TABLE_SIZE) break;
    }

    cout << "Key " << key << " not found.\n";
}

void HashTable::display() {         // O(n)
    cout << "\nHash Table Contents:\n";
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << i << ": ";
        if (table[i] == EMPTY)
            cout << "EMPTY";
        else if (table[i] == DELETED)
            cout << "DELETED";
        else
            cout << table[i];
        cout << endl;
    }
}
```

11.3 Separate Chaining

11.3.1 Question

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

11.3.2 Algorithm

Algorithm 1 - SeparateChainingInsert

Input

1. num - value to insert
2. table[] - hash table with linked lists
3. SIZE - size of hash table

Output

- Updated hash table with num inserted

Steps

```
1. $index \leftarrow num \bmod SIZE$
2. Create $newnode$ with $data = num$
3. $newnode.next \leftarrow table[index]$
4. $table[index] \leftarrow newnode$
```

Algorithm 2 - SeparateChainingDelete

Input

1. num - value to delete
2. table[] - hash table with linked lists
3. SIZE - size of hash table

Output

- Updated hash table with num removed (if found)

Steps

```
1. $index \leftarrow num \bmod SIZE$
2. If $table[index] = null$:
   1. Print "Element not found"
   2. Return
3. $temp \leftarrow table[index]$
4. If $temp.next = null$:
   1. If $temp.data = num$:
      1. Delete $temp$
      2. $table[index] \leftarrow null$
      3. Return
   2. Else:
      1. Print "Element not found"
      2. Return
5. If $temp.data = num$:
   1. $table[index] \leftarrow temp.next$
   2. Delete $temp$
   3. Return
6. While $temp.next \neq null$:
   1. If $temp.next.data \neq num$:
      1. $temp \leftarrow temp.next$
   2. Else:
      1. Break loop
7. If $temp.next \neq null$ and $temp.next.data = num$:
   1. $temp2 \leftarrow temp.next$
   2. $temp.next \leftarrow temp2.next$
   3. Delete $temp2$
8. Else:
   1. Print "Element not found"
```

Algorithm 3 - SeparateChainingSearch

Input

1. num - value to search for
2. table[] - hash table with linked lists
3. SIZE - size of hash table

Output

- Status message indicating if num was found or not

Steps

```
1. $index \leftarrow num$ mod $SIZE$
2. $temp \leftarrow table[index]$
3. While $temp \neq null$:
    1. If $temp.data = num$:
        1. Print "Element found at index $index$"
        2. Return
    2. $temp \leftarrow temp.next$
4. Print "Element not found"
```

Algorithm 4 - SeparateChainingDisplay

Input

1. table[] - hash table with linked lists
2. SIZE - size of hash table

Output

- Visual representation of the hash table with all chains

Steps

```
1. For $i$ from 0 to $SIZE - 1$:
    1. Print "Index $i$:"
    2. $temp \leftarrow table[i]$
    3. If $temp = null$:
        1. Print "Empty"
    4. Else:
        1. While $temp \neq null$:
            1. Print $temp.data$ followed by " -> "
            2. $temp \leftarrow temp.next$
        2. Print "NULL"
    5. Print newline
```

11.3.3 Code

main.cpp

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

#define SIZE 10
```

```
class hashtable{
    private:

        struct slot{
            vector<int> v;
        };

        slot table[SIZE];

        int hashfunction(int key){
            return key % SIZE;
        }

    public:
        void insert(int key);
        void display();
        void search(int key);
        void remove(int key);
};




int main(){
    int choice;
    int value;
    hashtable obj;

    while(1){
        cout << "\n<====== MENU ======>" << endl;
        cout << "1.Insert" << endl;
        cout << "2.Delete" << endl;
        cout << "3.Search" << endl;
        cout << "4.Display" << endl;
        cout << "5.Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice){
            case 1:
                cout << "Enter the value for insertion: ";
                cin >> value;
                obj.insert(value);
                break;
            case 2:
                cout << "Enter the value to delete: ";
                cin >> value;
                obj.remove(value);
                break;
            case 3:
                cout << "Enter the value to search: ";
                cin >> value;
                obj.search(value);
                break;
            case 4:
                obj.display();
                break;
            case 5:
                cout << "Exiting..." << endl;
                return 0;
            default:
                cout << "Selected choice cease to Exist\nPlease Try Again" << endl;
        }
    }
}



void hashtable::insert(int key){
    int index = hashfunction(key);
    table[index].v.push_back(key);
    cout << "Inserted element " << key << endl;
}


void hashtable::display(){
    for(int i = 0;i < SIZE;i++){
        cout << i;
        for(int each : table[i].v){
            cout << " ---> " << each;
        }
        cout << endl;
    }
}


void hashtable::search(int key){
    int index = hashfunction(key);
    for(int i = 0;i < table[index].v.size() - 1;i++){
        if(table[index].v[i] == key){
            cout << "Element found at chain " << index << " at index " << i + 1 << endl;
            return;
        }
    }

    cout << "Element not found in any chains !!" << endl;
}


void hashtable::remove(int key){
    int chain = hashfunction(key);
    int chainsize = table[chain].v.size();
    int keyindex = -1;

    for(int i = 0;i < chainsize;i++){
        if(table[chain].v[i] == key){
            keyindex = i;
            break;
        }
    }

    if(keyindex == -1){
        cout << "The Element is not in the hashtable" << endl;
        return;
    }else{
        for(int i = keyindex;i < chainsize - 1;i++){
            table[chain].v[i] = table[chain].v[i + 1];
        }

        table[chain].v.erase(table[chain].v.end() - 1);
    }
}
```

12 Graph ADT

12.1 Adjmatrix

12.1.1 Question

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

12.1.2 Algorithm

Algorithm 1 - InsertEdge

Input

1. u - first vertex
2. v - second vertex
3. adj[][] - adjacency matrix of graph
4. n - number of vertices

Output

- Updated adjacency matrix with edge (u,v) inserted

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print “Invalid vertex”
    2. Return
2. $adj[u][v] \leftarrow 1$
3. $adj[v][u] \leftarrow 1$
```

Algorithm 2 - DeleteEdge

Input

1. u - first vertex
2. v - second vertex
3. adj[][] - adjacency matrix of graph
4. n - number of vertices

Output

- Updated adjacency matrix with edge (u,v) removed

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print “Invalid vertex”
    2. Return
2. $adj[u][v] \leftarrow 0$
3. $adj[v][u] \leftarrow 0$
```

Algorithm 3 - SearchEdge

Input

1. u - first vertex
2. v - second vertex
3. adj[][] - adjacency matrix of graph
4. n - number of vertices

Output

- Status of edge existence between vertices u and v

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print “Invalid vertex”
    2. Return
2. If $adj[u][v] = 1$:
    1. Print “Edge exists between $u$ and $v$”
3. Else:
    1. Print “No edge exists between $u$ and $v$”
```

Algorithm 4 - DisplayGraph

Input

1. adj[][] - adjacency matrix of graph
2. n - number of vertices

Output

- Visual representation of the adjacency matrix

Steps

```
1. Print "Adjacency Matrix:"
2. For $i$ from 0 to $n-1$:
    1. For $j$ from 0 to $n-1$:
        1. Print $adj[i][j]$ followed by space
    2. Print newline
```

12.1.3 Code

main.cpp

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

class graph
{
private:
    int numVertices;
    vector<vector<int>> adjMatrix;
public:
    graph(int vertices ){
        numVertices = vertices;
        adjMatrix.resize(vertices,vector<int>(vertices,0));
    };
    bool isValid(int);
    void insertEdge(int,int);
    void deleteEdge(int,int);
    void searchEdge(int,int);
    void display();
};



int main(){
    int num;
    cout<<"Enter The Number of Vetices : ";
    cin>>num;
    graph g(num);
    int u, v;
    int choice;

    while (true) {
        cout << "\n--- Graph Menu ---\n";
        cout << "1. Insert Edge\n";
        cout << "2. Delete Edge\n";
        cout << "3. Search Edge\n";
        cout << "4. Display\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.insertEdge(u, v);
                break;
            case 2:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.deleteEdge(u, v);
                break;
            case 3:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.searchEdge(u, v);
                break;
            case 4:
                g.display();
                break;
            case 5:
                cout << "Exiting...\n";
                return 0;
            default:
                cout << "Invalid choice. Try again!\n";
        }
    }
    return 0;
}

bool graph::isValid(int v){
    return (v>=0 && v<numVertices);
}
void graph::insertEdge(int u,int v){
    if(isValid(u) && isValid(v)){
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1;
        cout<<"Edge Inserted between("<<u<<","<<v<<") and ("<<v<<","<<u<<")"<<endl;
    }
    else{
        cout<<"Invalid"<<endl;
    }
}
void graph::deleteEdge(int u,int v){
    if(isValid(u) && isValid(v)){
        adjMatrix[u][v] = 0;
        adjMatrix[v][u] = 0;
        cout<<"Edge Deleted between("<<u<<","<<v<<") and ("<<v<<","<<u<<")"<<endl;
    }
    else{
        cout<<"Invalid"<<endl;
    }
}
void graph::searchEdge(int u,int v){
    if (isValid(u) && isValid(v))
    {
        if(adjMatrix[u][v] == 1){
            cout<<"Vertex is present at position ("<<u<<","<<v<<")"<<endl;
        }
        else{
            cout<<"Vertex is NOT present at position ("<<u<<","<<v<<")"<<endl;
        }
    }
    else{
        cout<<"Invalid Index"<<endl;
    }
}

void graph::display(){
    for (int i = 0; i < numVertices; i++)
    {
        for (int j = 0; j < numVertices; j++)
        {
            cout<<adjMatrix[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;

}
```

12.2 Adjlist

12.2.1 Question

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

12.2.2 Algorithm

Algorithm 1 - InsertEdge

Input

1. u - first vertex
2. v - second vertex
3. adjList[] - adjacency list representation of graph
4. n - number of vertices

Output

- Updated adjacency list with edge (u,v) inserted

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print "Invalid vertex"
    2. Return
2. Create new node $newNode$ with $data = v$ and $next = adjList[u]$
3. $adjList[u] \leftarrow newNode$
4. Create new node $newNode$ with $data = u$ and $next = adjList[v]$
5. $adjList[v] \leftarrow newNode$
```

Algorithm 2 - DeleteEdge

Input

1. u - first vertex
2. v - second vertex
3. adjList[] - adjacency list representation of graph
4. n - number of vertices

Output

- Updated adjacency list with edge (u,v) removed

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print "Invalid vertex"
    2. Return
2. $temp \leftarrow adjList[u]$
3. $prev \leftarrow null$
4. While $temp \neq null$ and $temp.data \neq v$:
    1. $prev \leftarrow temp$
    2. $temp \leftarrow temp.next$
5. If $temp \neq null$:
    1. If $prev = null$:
        1. $adjList[u] \leftarrow temp.next$
    2. Else:
        1. $prev.next \leftarrow temp.next$
    3. Delete $temp$
6. $temp \leftarrow adjList[v]$
7. $prev \leftarrow null$
8. While $temp \neq null$ and $temp.data \neq u$:
    1. $prev \leftarrow temp$
    2. $temp \leftarrow temp.next$
9. If $temp \neq null$:
    1. If $prev = null$:
        1. $adjList[v] \leftarrow temp.next$
    2. Else:
        1. $prev.next \leftarrow temp.next$
    3. Delete $temp$
```

Algorithm 3 - SearchEdge

Input

1. u - first vertex
2. v - second vertex
3. adjList[] - adjacency list representation of graph
4. n - number of vertices

Output

- True if edge (u,v) exists, False otherwise

Steps

```
1. If $u \geq n$ or $v \geq n$:
    1. Print “Invalid vertex”
    2. Return
2. $temp \leftarrow adjList[u]$
3. While $temp \neq null$:
    1. If $temp.data = v$:
        1. Print “Edge exists between $u$ and $v$”
        2. Return
    2. $temp \leftarrow temp.next$
4. Print “No edge exists between $u$ and $v$”
```

12.2.3 Code

main.cpp

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

// --- Graph Class ---
class Graph {
private:
    int numVertices;
    vector<list<int>> adjList;

    bool valid(int u, int v); // Private helper

public:
    Graph(int V); // Constructor
    void insertEdge(int u, int v);
    void deleteEdge(int u, int v);
    void searchEdge(int u, int v);
    void display();
};

// Main Function
int main() {
    int V;
    cout << "Enter number of vertices: ";
    cin >> V;

    Graph g(V);
    int u, v;
    int choice;
    while (true) {
        cout << "\n--- Graph Menu ---\n";
        cout << "1. Insert Edge\n";
        cout << "2. Delete Edge\n";
        cout << "3. Search Edge\n";
        cout << "4. Display\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.insertEdge(u, v);
                break;
            case 2:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.deleteEdge(u, v);
                break;
            case 3:
                cout << "Enter edge (u v): ";
                cin >> u >> v;
                g.searchEdge(u, v);
                break;
            case 4:
                g.display();
                break;
            case 5:
                cout << "Exiting...\n";
                return 0;
            default:
                cout << "Invalid choice. Try again!\n";
        }
    }
}

// --- Function Definitions ---
Graph::Graph(int V) {
    numVertices = V;
    adjList.resize(V);
}

bool Graph::valid(int u, int v) {
```

```cpp
    return u >= 0 && v >= 0 && u < numVertices && v < numVertices;
}

void Graph::insertEdge(int u, int v) {
    if (valid(u, v)) {
        adjList[u].push_back(v);
        cout << "Edge inserted from " << u << " to " << v << endl;
    } else {
        cout << "Invalid vertices!\n";
    }
}

void Graph::deleteEdge(int u, int v) {
    if (valid(u, v)) {
        adjList[u].remove(v);
        cout << "Edge deleted from " << u << " to " << v << endl;
    } else {
        cout << "Invalid vertices!\n";
    }
}

void Graph::searchEdge(int u, int v) {
    if (valid(u, v)) {
        for (int neighbor : adjList[u]) {
            if (neighbor == v) {
                cout << "Edge exists from " << u << " to " << v << endl;
                return;
            }
        }
        cout << "Edge does not exist.\n";
    } else {
        cout << "Invalid vertices!\n";
    }
}

void Graph::display() {
    cout << "\nAdjacency List:\n";
    for (int i = 0; i < numVertices; ++i) {
        cout << i << " -> ";
        for (int neighbor : adjList[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}
```

12.3 Graph Algorithms

12.3.1 Question

Write a separate C++ menu-driven program to implement Graph ADT with the implementation for Prim's algorithm, Kruskal's algorithm, and Dijkstra's algorithm. Maintain proper boundary conditions and follow good coding practices. ### Algorithm

Algorithm 1 - Kruskal's Algorithm

Input

1. G(V,E) - Graph with vertices V and edges E
2. w(u,v) - Weight of edge between vertices u and v

Output

- MST - Minimum Spanning Tree of G
- mstWeight - Total weight of the MST

Steps

```
1. Initialize $priorityQueue \leftarrow$ empty MinHeap
2. Initialize $mstWeight \leftarrow 0$
3. Create DisjointSet $ds$ with $vertices$ elements
4. For each vertex $i$ from 0 to $vertices - 1$:
    1. For each vertex $j$ from 0 to $vertices - 1$:
        1. $weight \leftarrow$ weight of edge $(i, j)$
        2. If $weight > 0$:
            1. Add edge $(i, j, weight)$ to $priorityQueue$
5. While $priorityQueue$ is not empty:
    1. Extract min edge $(u, v, weight)$ from $priorityQueue$
    2. If $ds.find(u) \neq ds.find(v)$:
        1. $ds.unite(u, v)$
        2. Add edge $(u, v, weight)$ to MST
        3. $mstWeight \leftarrow mstWeight + weight$
6. Return MST and $mstWeight$
```

Algorithm 2 - Prim's Algorithm

Input

1. G(V,E) - Graph with vertices V and edges E
2. w(u,v) - Weight of edge between vertices u and v
3. start - Starting vertex

Output

- MST - Minimum Spanning Tree of G
- mstWeight - Total weight of the MST

Steps

```
1. Initialize $visited$ map with all vertices marked as false
2. Initialize $priorityQueue \leftarrow$ empty MinHeap
3. Set $visited[start] \leftarrow true$
4. For each vertex $i$ from 0 to $vertices - 1$:
    1. If $i \neq start$:
        1. $weight \leftarrow$ weight of edge $(start, i)$
        2. If $weight > 0$:
            1. Add edge $(start, i, weight)$ to $priorityQueue$
5. Initialize $mstWeight \leftarrow 0$
6. While $priorityQueue$ not empty and $|visited| < vertices$:
    1. Extract min edge $(u, v, weight)$ from $priorityQueue$
    2. If $visited[v]$, continue
    3. Add edge $(u, v, weight)$ to MST
    4. $mstWeight \leftarrow mstWeight + weight$
    5. $visited[v] \leftarrow true$
    6. For each vertex $i$ from 0 to $vertices - 1$:
        1. If $!visited[i]$:
            1. $edgeWeight \leftarrow$ weight of edge $(v, i)$
            2. If $edgeWeight > 0$:
                1. Add edge $(v, i, edgeWeight)$ to $priorityQueue$
7. Return MST and $mstWeight$
```

Algorithm 3 - Dijkstra's Algorithm

Input

1. G(V,E) - Graph with vertices V and edges E
2. w(u,v) - Weight of edge between vertices u and v
3. start - Starting vertex

Output

- distance[] - Array of shortest distances from start to all vertices

Steps

```
1. Initialize $visited$ map with all vertices marked as false
2. Initialize $distance[i] \leftarrow \infty$ for all vertices $i$
3. Set $distance[start] \leftarrow 0$
4. While $!visited[start]$:
    1. For each vertex $i$ from 0 to $vertices - 1$:
        1. If $i \neq start$:
            1. $weight \leftarrow$ weight of edge $(start, i)$
            2. If $weight > 0$ and $distance[start] + weight < distance[i]$:
                1. $distance[i] \leftarrow distance[start] + weight$
    2. $visited[start] \leftarrow true$
    3. $minim \leftarrow \infty$
    4. For each vertex $i$ from 0 to $vertices - 1$:
        1. If $!visited[i]$ and $distance[i] < minim$:
            1. $minim \leftarrow distance[i]$
            2. $start \leftarrow i$
5. Return $distance[]$
```

12.3.2 Code

main.cpp

```
/*
            To implement the following three algorithms using a menu driven program
                                1.Prim's Algorithm
                                2.Kruskal's Algorithm
                                3.Dijikstra's Algorithm
*/

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <limits>

using namespace std;

#define INF INT_MAX
#define MAX_DIST 1000000  // Define a large number instead of INF

class Edge {
public:
    int src, dest, weight;
    Edge(int s, int d, int w);
};

class Graph {
private:
    int V;
    vector<vector<pair<int, int>>> adjList;
    vector<Edge> edges;

public:
    Graph(int vertices);
    void addEdge(int u, int v, int w);
    void display();
    void prims();
    int findParent(int u, vector<int> &parent);
    void kruskal();
    void dijkstra(int src);
};

int main() {
    int V, choice;
    cin >> V;

    Graph g(V);
    while (true) {
        cout << "\nMenu:\n"
             << "1. Add Edge\n"
             << "2. Display Graph\n"
             << "3. Prim's Algorithm\n"
             << "4. Kruskal's Algorithm\n"
             << "5. Dijkstra's Algorithm\n"
             << "6. Exit\n"
             << "Enter your choice: ";
        cin >> choice;

        if (choice == 1) {
            int u, v, w;
            cin >> u >> v >> w;
            g.addEdge(u, v, w);
        } else if (choice == 2) {
            g.display();
        } else if (choice == 3) {
            g.prims();
        } else if (choice == 4) {
            g.kruskal();
        } else if (choice == 5) {
            int src;
            cin >> src;
            g.dijkstra(src);
        } else if (choice == 6) {
            break;
        }
    }
    return 0;
}

Edge::Edge(int s, int d, int w) : src(s), dest(d), weight(w) {}

Graph::Graph(int vertices) {
    V = vertices;
    adjList.resize(V);
}

void Graph::addEdge(int u, int v, int w) {
    adjList[u].emplace_back(v, w);
    adjList[v].emplace_back(u, w);
    edges.emplace_back(u, v, w);
}

void Graph::display() {
    for (int i = 0; i < V; ++i) {
        cout << i << " -> ";
        for (auto& pair : adjList[i]) {  // Iterate over the adjacency list
            int v = pair.first;    // Destination vertex
            int w = pair.second;  // Edge weight
            cout << "(" << v << ", " << w << ") ";
        }
        cout << "\n";
    }
}

void Graph::prims() {
    vector<int> key(V, MAX_DIST), parent(V, -1);
    vector<bool> inMST(V, false);
    key[0] = 0;

    for (int count = 0; count < V - 1; ++count) {
        int u = -1;
        // Find the vertex with the minimum key value that is not yet included in MST
        for (int i = 0; i < V; ++i)
            if (!inMST[i] && (u == -1 || key[i] < key[u]))
                u = i;

        if (u == -1) {
            cout << "The graph is disconnected!" << endl;
            return;
        }
        inMST[u] = true;

        for (auto& pair : adjList[u]) {
            int v = pair.first;   // Destination vertex
            int w = pair.second;  // Edge weight
            if (!inMST[v] && w < key[v]) {
                key[v] = w;
                parent[v] = u;
            }
        }
    }

    cout << "Minimum Spanning Tree (Prim's):\n";
    for (int i = 1; i < V; ++i)
        cout << parent[i] << " - " << i << " (Weight: " << key[i] << ")\n";
}

int Graph::findParent(int u, vector<int> &parent) {
    if (parent[u] != u)
        parent[u] = findParent(parent[u], parent);
    return parent[u];
}

void Graph::kruskal() {
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.weight < b.weight;
    });

    vector<int> parent(V);
    for (int i = 0; i < V; ++i)
        parent[i] = i;

    cout << "Minimum Spanning Tree (Kruskal's):\n";
    for (Edge &e : edges) {
        int pu = findParent(e.src, parent);
        int pv = findParent(e.dest, parent);
        if (pu != pv) {
            cout << e.src << " - " << e.dest << " (Weight: " << e.weight << ")\n";
            parent[pu] = pv;
        }
    }
}

void Graph::dijkstra(int src) {
    vector<int> dist(V, MAX_DIST);
    dist[src] = 0;
```

```cpp
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.emplace(0, src);

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (auto& pair : adjList[u]) {
            int v = pair.first;
            int w = pair.second;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.emplace(dist[v], v);
            }
        }
    }

    cout << "Shortest paths from source " << src << ":\n";
    for (int i = 0; i < V; ++i) {
        if (dist[i] == MAX_DIST)
            cout << "No path to " << i << "\n";
        else
            cout << "Distance to " << i << " = " << dist[i] << "\n";
    }
}
```

13 Lab Record Generator

13.1 assets

13.1.1 Code

main.cpp

13.2 templates

13.2.1 Algorithm

Algorithm 1 - AlgorithmName1

Input

1. input 1
2. input 2

Output

- output 1
- output 2

Steps

1. step 1
2. step 2
3. step 3
    1. step31
    2. step32
    3. step33
        1. step331
        2. step332
4. step 4
5. step 5
    1. step51
    2. step52

Algorithm 2 - AlgorithmName2

Input

1. input 1
2. input 2

Output

- output 1
- output 2

Steps

1. step 1
2. step 2
    1. step 21

13.2.2 Code

main.cpp