**BUBBLE SORT:**

#include <iostream>

void swap(int &a, int &b) {

int temp = a;

a = b;

b = temp;

}

void bubbleSort(int arr[], int n) {

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

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

// If the element found is greater than the next element, swap them

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

}

void printArray(int arr[], int size) {

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

std::cout << arr[i] << " ";

}

std::cout << std::endl;

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

std::cout << "Original array: ";

printArray(arr, n);

bubbleSort(arr, n);

std::cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Insertion Sort:**

#include <iostream>

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

--j;

}

arr[j + 1] = key;

}

}

void printArray(int arr[], int size) {

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

std::cout << arr[i] << " ";

}

std::cout << std::endl;

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

std::cout << "Original array: ";

printArray(arr, n);

insertionSort(arr, n);

std::cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Selection Sort:**

#include <iostream>

void swap(int &a, int &b) {

int temp = a;

a = b;

b = temp;

}

void selectionSort(int arr[], int n) {

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

// Find the minimum element in the unsorted part of the array

int minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the found minimum element with the first element

swap(arr[i], arr[minIndex]);

}

}

void printArray(int arr[], int size) {

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

std::cout << arr[i] << " ";

}

std::cout << std::endl;

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

std::cout << "Original array: ";

printArray(arr, n);

selectionSort(arr, n);

std::cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Radix sort :**

#include <iostream>

#include <vector>

using namespace std;

// A utility function to get the maximum value in arr[]

int getMax(int arr[], int n) {

int max = arr[0];

for (int i = 1; i < n; i++)

if (arr[i] > max)

max = arr[i];

return max;

}

// Using counting sort to sort the elements based on significant places

void countingSort(int arr[], int n, int exp) {

const int RADIX = 10;

vector<int> output(n);

vector<int> count(RADIX, 0);

// Count occurrences of elements at each significant place

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

count[(arr[i] / exp) % RADIX]++;

// Cumulative count to determine the positions

for (int i = 1; i < RADIX; i++)

count[i] += count[i - 1];

// Build the output array

for (int i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % RADIX] - 1] = arr[i];

count[(arr[i] / exp) % RADIX]--;

}

// Copy the output array back to arr[]

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

arr[i] = output[i];

}

// Main function to implement radix sort

void radixSort(int arr[], int n) {

// Find the maximum number to know the number of digits

int max = getMax(arr, n);

// Do counting sort for every digit

for (int exp = 1; max / exp > 0; exp \*= 10)

countingSort(arr, n, exp);

}

// A utility function to print an array

void printArray(int arr[], int n) {

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

cout << arr[i] << " ";

cout << endl;

}

// Driver program

int main() {

int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

printArray(arr, n);

radixSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**SHELL SORT:**

#include<iostream>

using namespace std;

int shellSort (int arr[],int n){

for(int gap=n/2;gap>0;gap/=2){

for (int i= gap ;i<n;i+=1){

int temp =arr[i];

int j;

for(j=i;j>=gap&&arr[j-gap]>temp;j-=gap){

arr[j]=arr[j-gap];

}

arr[j]=temp;

}

}

return 0;

}

void printArray(int arr[],int n){

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

cout<<arr[i]<<" ";

}

}

int main(){

int arr[]={12,34,54,2,3},i;

int n=sizeof(arr)/sizeof(arr[0]);

cout<<"Array before sorting: ";

printArray(arr,n);

shellSort(arr,n);

cout<<"\nArray after Shell Sort: ";

printArray(arr,n);

cout<<"\nGap Size: "<<n/2;

return 0;

}

**Quick\_sort:**

#include <iostream>

#include <vector>

template <typename T>

int partition(std::vector<T>& arr, int low, int high) {

T pivot = arr[high], temp;

int i = low - 1;

for (int j = low; j < high; ++j)

if (arr[j] < pivot) std::swap(arr[++i], arr[j]);

std::swap(arr[i + 1], arr[high]);

return i + 1;

}

template <typename T>

void quickSort(std::vector<T>& arr, int low, int high) {

if (low < high) {

int p = partition(arr, low, high);

quickSort(arr, low, p - 1);

quickSort(arr, p + 1, high);

}

}

int main() {

std::vector<int> arr = {12, 4, 5, 6, 7, 3, 1, 15};

std::cout << "Original array: ";

for (int num : arr) std::cout << num << " ";

quickSort(arr, 0, arr.size() - 1);

std::cout << "\nQuick Sorted array: ";

for (int num : arr) std::cout << num << " ";

return 0;

}

**Linear Search:**

#include <iostream>

// Linear search function

int linearSearch(int arr[], int n, int target) {

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

if (arr[i] == target) {

return i; // Return the index of the target element if found

}

}

return -1; // Return -1 if the target element is not found

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 22;

int result = linearSearch(arr, n, target);

if (result != -1) {

std::cout << "Element " << target << " found at index " << result << std::endl;

} else {

std::cout << "Element " << target << " not found in the array" << std::endl;

}

return 0;

}

**Binary Serach:**

#include <iostream>

// Binary search function (iterative approach)

int binarySearch(int arr[], int low, int high, int target) {

while (low <= high) {

int mid = low + (high - low) / 2;

// Check if the target is present at the middle

if (arr[mid] == target) {

return mid;

}

// If the target is greater, ignore the left half

if (arr[mid] < target) {

low = mid + 1;

}

// If the target is smaller, ignore the right half

else {

high = mid - 1;

}

}

// If the target is not present in the array

return -1;

}

int main() {

int arr[] = {11, 12, 22, 25, 64};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 25;

int result = binarySearch(arr, 0, n - 1, target);

if (result != -1) {

std::cout << "Element " << target << " found at index " << result << std::endl;

} else {

std::cout << "Element " << target << " not found in the array" << std::endl;

}

return 0;

}

**Modulo Division:**

#include<iostream>

using namespace std;

const int size = 7;

int arr[size];

void init()

{

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

{

arr[i] = -1;

}

}

void insert(int value)

{

int key = value % size; // use of modulo division

if (arr[key] == -1)

{

arr[key] = value;

cout << value << " inserted at arr[" << key << "]\n";

}

else

{

cout << "Collision : arr[" << key << "] has element " << arr[key] << " already!\n";

cout << "Unable to insert " << value << "\n";

}

}

void search(int value)

{

int key = value % size;

if (arr[key] == value)

{

cout << "Search Found\n";

}

else

{

cout << "Search Not Found\n";

}

}

void display()

{

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

{

cout << "arr[" << i << "] = " << arr[i] << "\n";

}

}

int main()

{

init();

insert(10);

insert(4);

insert(2);

insert(3);

cout << "Hash table\n";

display();

cout << "\nSearching value 4..\n";

search(4);

return 0;

}

**Digit Extraction:**

#include<iostream>

int digit\_extraction(int key) {

int key\_length = 0;

int first\_digit = 0;

int fourth\_digit = 0;

first\_digit = key % 10000000;

first\_digit = first\_digit / 1000000;

fourth\_digit = key % 1000;

fourth\_digit = fourth\_digit / 100;

std::cout << key << " key would be hashed at location " << first\_digit << fourth\_digit << std::endl;

}

int main() {

digit\_extraction(1347878); // 18

digit\_extraction(1234678); // 16

return 0;

}

**Fold Shift:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

#include<math.h>

#include <iostream>

using namespace std;

int count\_digits(int key){

int count=0;

while(key!=0){

key/=10;

++count;

}

return count;

}

int fold\_shift(int key,int size){

int key\_roll=key;

int key\_sum=0;

int key\_frac=0;

int key\_length=0;

int fraction=size;

key\_length = count\_digits(key\_roll);

while(key\_length>0){

if (key\_length>fraction){

key\_frac=key\_roll/(int)pow(10,(key\_length-fraction));

key\_sum+=key\_frac;

key\_roll=key\_roll%(int)pow(10,(key\_length-fraction));

key\_length=key\_length-fraction;

}

else{

key\_sum+=key\_roll;

break;

}

}

return key\_sum%(int)pow(10,(fraction));

}

int main(){

cout<<"\n"<<fold\_shift(12345678,1);

cout<<"\n"<<fold\_shift(12789,3);

cout<<"\n"<<fold\_shift(5678,2);

return 0;

}

**Linear Collsion:**

#include <iostream>

using namespace std;

const int size = 10;

int ht[size];

void store(int x[], int n);

int modulodivision(int key);

int linearprobe(int address);

int main() {

int i, n, x[10];

char ch;

cout << "Enter the number of elements: ";

cin >> n;

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

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

cin >> x[i];

}

store(x, n);

cout << "Hashtable is as shown:" << endl;

for (i = 0; i < size; i++) {

cout << ht[i] << " ";

}

return 0;

}

void store(int x[], int n) {

int i, key, address;

// Initializing hash table to empty

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

ht[i] = -1;

// Copying elements from original array to hashtable

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

key = x[i];

address = modulodivision(key);

if (ht[address] != -1)

address = linearprobe(address);

ht[address] = key;

}

}

// Hash Function

int modulodivision(int key) {

int address;

address = key % size + 1;

if (address == size) {

return 0;

} else {

return address;

}

}

// Collision Resolution

int linearprobe(int address) {

while (ht[address] != -1) {

address++;

if (address == size)

address = 0;

}

return address;

}

**Stack Array:**

#include <iostream>

const int MAX\_SIZE=100;

class Stack{

private:

int arr[MAX\_SIZE];

int top;

public:

Stack(){

top=-1;

}

void push (int value){

if (top<MAX\_SIZE - 1){

arr[++top]=value;

std::cout<<"Pushed: "<<value<< std::endl;

}

else{

std::cout<<"Stack overflow! Cannot push "<<value<<"."<<std::endl;

}

}

void pop(){

if(top>=0){

std::cout<<"Popped: : "<<arr[top--]<< std::endl;

}

else{

std::cout<<"Stack UnderFlow! Cannot pop from an empty stack. "<<std::endl;

}

}

bool isEmpty(){

return top==-1;

}

int peek(){

if(top>=0){

return arr[top];

}

else{

std::cerr<<"Stack is empty! Cannot peek. "<<std::endl;

return -1;

}

}

};

int main(){

Stack myStack;

myStack.push(10);

myStack.push(20);

myStack.push(30);

std::cout<<"Top Element: "<<myStack.peek() <<std::endl;

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop();

return 0;

}

**Stack Link list:**

#include <iostream>

// Node class to represent individual elements in the linked list

class Node {

public:

int data;

Node\* next;

Node\* nullptr;

// Constructor to initialize the node

Node(int value) : data(value), next(nullptr) {}

};

// Stack class using linked list

class Stack {

private:

Node\* top;

Node\* nullptr;

public:

// Constructor

Stack() : top(nullptr) {}

// Function to push an element onto the stack

void push(int value) {

Node\* newNode = new Node(value);

newNode->next = top;

top = newNode;

std::cout << "Pushed: " << value << std::endl;

}

// Function to pop an element from the stack

void pop() {

if (!isEmpty()) {

Node\* temp = top;

top = top->next;

std::cout << "Popped: " << temp->data << std::endl;

delete temp; // Free the memory of the popped node

} else {

std::cout << "Stack underflow! Cannot pop from an empty stack." << std::endl;

}

}

// Function to check if the stack is empty

bool isEmpty() {

return top == nullptr;

}

// Function to get the top element of the stack without removing it

int peek() {

if (!isEmpty()) {

return top->data;

} else {

std::cerr << "Stack is empty. Cannot peek." << std::endl;

return -1; // Assuming -1 as an invalid value

}

}

};

int main() {

Stack myStack;

myStack.push(10);

myStack.push(20);

myStack.push(30);

std::cout << "Top element: " << myStack.peek() << std::endl;

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop(); // Trying to pop from an empty stack

return 0;

}

**BFS:**

#include<iostream>

#include<queue>

#define NODE 6

using namespace std;

typedef struct node{

int val;

int state;

}node;

int graph[NODE][NODE]={

{0,1,1,1,0,0},

{0,0,0,1,1,0},

{1,0,0,1,0,1},

{1,1,1,0,1,1},

{0,1,0,1,0,1},

{0,0,1,1,1,0},

};

void bfs(node \*vert,node s){

node u;

int i,j;

queue<node> que;

for(i=0;i<NODE;i++){

vert[i].state=0;

}

vert[s.val].state=1;

que.push(s);

while(!que.empty()){

u=que.front();

que.pop();

cout<<char(u.val+'A')<<" ";

for(i=0;i<NODE;i++){

if(graph[i][u.val]){

if(vert[i].state==0){

vert[i].state=1;

que.push(vert[i]);

}

}

}

u.state=2;

}

}

int main(){

node vertices[NODE];

node start;

char s;

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

vertices[i].val=i;

}

s='B';

start.val=s-'A';

cout<<"BFS Traversal: ";

bfs(vertices,start);

cout<<endl;

}

**DFS:**

#include<iostream>

#include<stack>

#define NODE 6

using namespace std;

typedef struct node{

int val;

int state;

}node;

int graph[NODE][NODE]={

{0,1,1,1,0,0},

{0,0,0,1,1,0},

{1,0,0,1,0,1},

{1,1,1,0,1,1},

{0,1,0,1,0,1},

{0,0,1,1,1,0},

};

void dfs(node \*vertex,node start){

node u;

stack<node> myStack;

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

vertex[i].state=0;

}

myStack.push(start);

while(!myStack.empty()){

u=myStack.top();

myStack.pop();

cout<<char(u.val+'A')<<" ";

if(u.state!=1){

u.state=1;

vertex[u.val].state=1;

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

if(vertex[i].state==0){

myStack.push(vertex[i]);

vertex[i].state=1;

}

}

}

}

}

int main(){

node vertices[NODE];

node start;

char s;

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

vertices[i].val=i;

}

s='C';

start.val=s-'A';

cout<<"DFS Traversal: ";

dfs(vertices,start);

cout<<endl;

}

**KRUSKAL:**

#include<iostream>

#include<algorithm>

using namespace std;

const int MAX=1e4+5;

int id[MAX],nodes,edges;

pair<long long,pair<int,int> >p[MAX];

void init (){

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

id[i]=i;

}

int root(int x){

while(id[x]!=x){

id[x]=id[id[x]];

x=id[x];

}

return x;

}

void union1(int x,int y){

int p=root(x);

int q=root(y);

id [p]=id[q];

}

long long kruskal(pair<long long ,pair<int,int> >p[]){

int x,y;

long long cost,minimumCost=0;

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

x=p[i].second.first;

y=p[i].second.second;

cost=p[i].first;

if(root(x)!=root(y)){

minimumCost+=cost;

union1(x,y);

}

}

return minimumCost;

}

int main(){

int x,y;

long long weight,cost,minimumCost;

init();

cout<<"Enter nodes and Edges: ";

cin>>nodes>>edges;

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

cout<<"Enter the value of X,Y and edges:";

cin>>x>>y>>weight;

p[i]=make\_pair(weight,make\_pair(x,y));

}

sort (p,p+edges);

minimumCost=kruskal(p);

cout<<"Minimum cost is: "<<minimumCost<<endl;

return 0;

}

**PRISM:**

#include<iostream>

using namespace std;

const int V = 6;

int minKey(int key[], bool visited[]) {

int min = 999, min\_index;

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

if (visited[v] == false && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void printMST(int parent[], int cost[V][V]) {

int minCost = 0;

cout << "Edge \tWeight\n";

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

cout << parent[i] << " - " << i << "\t" << cost[i][parent[i]] << "\n";

minCost += cost[i][parent[i]];

}

cout << "Total cost is : " << minCost;

}

void findMST(int cost[V][V]) {

int parent[V], key[V];

bool visited[V];

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

key[i] = 999;

visited[i] = false;

parent[i] = -1;

}

key[0] = 0;

parent[0] = -1;

for (int x = 0; x < V - 1; x++) {

int u = minKey(key, visited);

visited[u] = true;

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

if (cost[u][v] != 0 && visited[v] == false && cost[u][v] < key[v]) {

parent[v] = u;

key[v] = cost[u][v];

}

}

}

printMST(parent, cost);

}

int main() {

int cost[V][V];

cout << "Enter the weights for a graph with " << V << " vertices:\n";

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

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

cin >> cost[i][j];

}

}

findMST(cost);

return 0;

}

**Adjancey Graph:**

#include<iostream>

using namespace std;

int vertArr[20][20];

int count=0;

void displayMatrix(int v){

int i,j;

for (i=0;i<v;i++){

for(j=0;j<v;j++){

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

}

cout<<endl;

}

}

void add\_edge(int u,int v){

vertArr[u][v]=1;

vertArr[v][u]=1;

}

main(int argc,char argv[]){

int v=6;

add\_edge(0,4);

add\_edge(0,3);

add\_edge(1,2);

add\_edge(1,2);

add\_edge(1,5);

add\_edge(2,3);

add\_edge(5,4);

displayMatrix(v);

}

**Postfix evaluation:**

#include <iostream>

#include <stack>

using namespace std;

int isOperator(char c) {

  return (c == '+' || c == '-' || c == '\*' || c == '/');

}

int getPrecedence(char c) {

  if (c == '+' || c == '-') {

    return 1;

  } else if (c == '\*' || c == '/') {

    return 2;

  }

  return 0;

}

string infixToPostfix(string infixExpression) {

  stack<char> operators;

  string postfixExpression = "";

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

    char ch = infixExpression[i];

    if (isalnum(ch)) {

      postfixExpression += ch;

    } else if (isOperator(ch)) {

      while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(ch)) {

        postfixExpression += operators.top();

        operators.pop();

      }

      operators.push(ch);

    } else if (ch == '(') {

      operators.push(ch);

    } else if (ch == ')') {

      while (!operators.empty() && operators.top() != '(') {

        postfixExpression += operators.top();

        operators.pop();

      }

      operators.pop();  // Pop '(' from stack

    }

  }

  while (!operators.empty()) {

    postfixExpression += operators.top();

    operators.pop();

  }

  return postfixExpression;

}

int main() {

  string infixExpression;

  cout << "Enter infix expression: ";

  cin >> infixExpression;

  string postfixExpression = infixToPostfix(infixExpression);

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

  return 0;

}

**Balancing of parantheses:**

#include<iostream>

#include<stack>

using namespace std;

bool isBalanced(string expr) {

   stack<char> s;

   char ch;

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

      if (expr[i]=='('||expr[i]=='['||expr[i]=='{') {

         s.push(expr[i]);

         continue;

      }

      if (s.empty())

         return false;

         switch (expr[i]) {

            case ')':

               ch = s.top();

               s.pop();

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

                  return false;

                  break;

            case '}':

               ch = s.top();

               s.pop();

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

                  return false;

                  break;

            case ']':

               ch = s.top();

               s.pop();

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

                  return false;

                  break;

         }

      }

      return (s.empty());

}

main() {

   string expr = "[()\*()]";

   cout<<"expression is"<<expr<<endl;

   if (isBalanced(expr))

      cout << "Balanced";

   else

      cout << "Not Balanced";

}

**Sparse\_matrix:**

#include <iostream>

#include <vector>

using namespace std;

struct SNode

{

    int data;

    int col;

    int row;

    SNode\* next;

};

struct MatrixNode

{

    vector<vector<int>> matrix;

    SNode\* SNPTR;

};

SNode\* createNode(int row, int col, int data)

{

    SNode\* newNode = new SNode();

    newNode->data = data;

    newNode->col = col;

    newNode->row = row;

    newNode->next = nullptr;

    return newNode;

}

MatrixNode\* addToList(MatrixNode\* MNhead, int row, int col, int data)

{

    SNode\* newNode = createNode(row, col, data);

    if (MNhead->SNPTR == nullptr)

    {

        MNhead->SNPTR = newNode;

        return MNhead;

    }

    newNode->next = MNhead->SNPTR;

    MNhead->SNPTR = newNode;

    return MNhead;

}

MatrixNode\* constructSparseMatrix(MatrixNode\* MNhead, vector<vector<int>> Matrix)

{

    MNhead = new MatrixNode();

    MNhead->matrix = Matrix;

    for (int i = 0; i < MNhead->matrix.size(); i++) {

        for (int j = 0; j < MNhead->matrix[i].size(); j++) {

            if (Matrix[i][j] != 0)

                MNhead = addToList(MNhead, i, j, Matrix[i][j]);

        }

    }

    return MNhead;

}

void displaySparseMatrix(MatrixNode\* MNhead)

{

    cout << "Sparse Matrix:\n";

    for (int i = 0; i < MNhead->matrix.size(); ++i) {

        for (int j = 0; j < MNhead->matrix[i].size(); ++j) {

            cout << MNhead->matrix[i][j] << " ";

        }

        cout << endl;

    }

}

void displayLinkedList(MatrixNode\* MNhead)

{

    SNode\* current = MNhead->SNPTR;

    cout << "Linked List:\n";

    while (current != nullptr) {

        cout << "Row: " << current->row << " Col: " << current->col << " Data: " << current->data << endl;

        current = current->next;

    }

}

int main()

{

    MatrixNode\* MNhead = nullptr;

    vector<vector<int>> Matrix = {{0, 1, 0, 0, 0},

                                  {0, 1, 0, 0, 0},

                                  {0, 0, 2, 0, 0},

                                  {0, 3, 0, 4, 0},

                                  {0, 0, 5, 0, 0}};

    MNhead = constructSparseMatrix(MNhead, Matrix);

    displaySparseMatrix(MNhead);

    displayLinkedList(MNhead);

    cin.get();

    return 0;

}

**Polynomial\_addition:**

#include <bits/stdc++.h>

using namespace std;

struct Node

{

int coeff;

int pow;

struct Node\* next;

};

void create\_node(int x, int y, struct Node\*\* temp)

{

struct Node \*r, \*z;

z = \*temp;

if (z==NULL)

{

r = (struct Node\*)malloc(sizeof(struct Node));

r->coeff = x;

r->pow = y;

\*temp = r;

r->next = (struct Node\*)malloc(sizeof(struct Node));

r= r->next;

r->next = NULL;

}

else

{

r->coeff = x;

r->pow = y;

r->next = (struct Node\*)malloc(sizeof(struct Node));

r= r->next;

r->next = NULL;

}

}

void polyadd(struct Node\* poly1, struct Node\* poly2, struct Node\* poly)

{

while(poly1->next && poly2->next)

{

if (poly1->pow > poly2->pow)

{

poly->pow = poly1->pow;

poly->coeff = poly1->coeff;

poly1 = poly1->next;

}

else if (poly->pow < poly2->pow)

{

poly->pow = poly2->pow;

poly->coeff = poly2->coeff;

poly2 = poly2->next;

}

else

{

poly->pow = poly1->pow;

poly->coeff = poly1->coeff + poly2->coeff;

poly1 = poly1->next;

poly2 = poly2->next;

}

poly->next= (struct Node\*)malloc(sizeof(struct Node));

poly= poly->next;

poly->next = NULL;

}

while (poly1->next || poly2->next)

{

if(poly1->next)

{

poly->pow = poly1->pow;

poly->coeff = poly1->coeff;

poly1 = poly1->next;

}

if(poly2->next)

{

poly->pow = poly2->pow;

poly->coeff = poly2->coeff;

poly2 = poly2->next;

}

poly->next=(struct Node\*)malloc(sizeof(struct Node));

poly = poly->next;

poly->next = NULL;

}

}

void show(struct Node\* node)

{

while(node->next !=NULL)

{

printf("%dx^%d",node->coeff,node->pow);

node = node->next;

if (node->coeff>=0)

{

if(node->next !=NULL)

        printf("+");

}

}

}

int main()

{

struct Node \*poly1=NULL, \*poly2 = NULL, \*poly = NULL;

create\_node(5,2, &poly1);

create\_node(4,1, &poly1);

create\_node(2,0, &poly1);

create\_node(-5,1, &poly2);

create\_node(-5,0, &poly2);

cout<<"1st Number:";

show(poly1);

cout<<"\n2nd Number:";

show(poly2);

poly = (struct Node\*)malloc(sizeof(struct Node));

polyadd(poly1, poly2, poly);

cout<<"\nAdded polynomial:";

show(poly);

return 0;

}

**Singlylinkedlist\_with\_count&reverse:**

#include<bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    struct Node\* next;

    Node(int data) {

        this->data = data;

        next = NULL;

    }

};

struct LinkedList {

    Node\* head;

    LinkedList() {

        head = NULL;

    }

    void reverse() {

        auto curr = head; // current pointer

        Node\* prev = NULL; // previous pointer

        while (curr) {

            auto temp = curr->next;

            curr->next = prev;

            prev = curr;

            head = prev;

            curr = temp;

        }

    }

    void print() {

        struct Node\* temp = head;

        while (temp != NULL) {

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

            temp = temp->next;

        }

    }

    void push(int data) {

        Node\* temp = new Node(data);

        temp->next = head;

        head = temp;

    }

    int countNodes() {

        int count = 0;

        Node\* temp = head;

        while (temp != NULL) {

            count++;

            temp = temp->next;

        }

        return count;

    }

};

int main() {

    LinkedList list;

    list.push(20);

    list.push(90);

    list.push(39);

    list.push(54);

    cout << "The linked list Before Reversing: ";

    list.print();

    list.reverse();

    cout << "\nAfter Reversing: ";

    list.print();

    cout << "\nNumber of nodes in the list: " << list.countNodes() << endl;

    return 0;

}

**DoublyLinkedlist\_with\_delete:**

#include<iostream>

using namespace std;

struct Node

{

    int data;

    struct Node \*prev;

    struct Node \*next;

};

struct Node\* head = NULL;

void insert(int newdata)

{

    struct Node\* newnode = (struct Node\*) malloc(sizeof(struct Node));

    newnode->data = newdata;

    newnode->prev = NULL;

    newnode->next = head;

    if(head != NULL)

        head->prev = newnode;

    head = newnode;

}

int countNodes()

{

    Node\* temp = head;

    int i = 0;

    while(temp != NULL)

    {

        i++;

        temp = temp->next;

    }

    return i;

}

void display()

{

    struct Node\* ptr;

    ptr = head;

    while(ptr !=NULL)

    {

        cout << ptr->data << " ";

        ptr = ptr->next;

    }

}

void deleteNode(int key)

{

    struct Node\* temp = head;

    while (temp != NULL && temp->data != key)

    {

        temp = temp->next;

    }

    if (temp == NULL)

    {

        cout << "Node with value " << key << " not found in the list." << endl;

        return;

    }

    if (temp->prev != NULL)

    {

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

    }

    else

    {

        head = temp->next;

    }

    if (temp->next != NULL)

    {

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

    }

    free(temp);

}

int main(){

    insert(10);

    insert(20);

    insert(70);

    insert(30);

    cout << "The doubly linked list is: ";

    display();

    cout << "\nNumber of nodes: " << countNodes() << endl;

    // Delete a node with value 20

    deleteNode(20);

    cout << "After deleting a node, the doubly linked list is: ";

    display();

    cout << "\nNumber of nodes: " << countNodes() << endl;

    return 0;

}

**DoublyCircular\_Linkedlist**

#include <iostream>

using namespace std;

class node

{

public:

int data;

int key;

node \*next;

node \*prev;

};

node \*head = NULL;

node \*last = NULL;

node \*current;

bool isEmpty()

{

return head == NULL;

}

int length()

{

int length = 0;

node \*current;

for(current = head; current != NULL; current = current->next)

{

length++;

}

return length;

}

void displayForward()

{

node \*ptr = head;

cout<<"\n[ ";

while(ptr != NULL)

{

cout<<ptr->key,ptr->data;

ptr = ptr ->next;

}

cout<<" ]";

}

void displayBackward()

{

node \*ptr = last;

cout<<"\n[ ";

while(ptr != NULL)

{

cout<<ptr->key,ptr->data;

ptr = ptr ->prev;

}

cout<<" ]";

}

void insertFirst(int key, int data){

node \*link = new node();

link -> key = key;

link -> data = data;

if(isEmpty()){

last = link;

}

else{

head->prev = link;

}

link->next = head;

head = link;

}

void insertLast(int key, int data){

node \*link = new node();

link -> key = key;

link -> data = data;

if(isEmpty()){

last = link;

}

else{

last->next = link;

}

link->prev = last;

last = link;

}

node\* deleteFirst(){

node \*tempLink = head;

if(head->next == NULL){

last = NULL;

}

else{

head->next->prev = NULL;

}

head = head -> next;

return tempLink;

}

node\* deleteLast(){

node \*tempLink = last;

if(head->next == NULL){

head = NULL;

}

else{

last->prev->next = NULL;

}

last = last -> prev;

return tempLink;

}

node\* del(int key)

{

node\* current = head;

node\* previous = NULL;

if(head == NULL)

{

return NULL;

}

while(current ->key != key)

{

if(current->next == NULL)

{

return NULL;

}

else

{

previous = current;

current = current->next;

}

}

if(current ==  head)

{

head = head->next;

}

else

{

current ->prev->next = current->next;

}

if(current == last)

{

last = current ->prev;

}

else

{

current ->next->prev = current->prev;

}

return current;

}

bool insertAfter(int key, int newKey, int data)

{

node \*current = head;

if(head == NULL)

{

return false;

}

while(current->key != key)

{

if(current->next == NULL)

{

return false;

}

else

{

current =  current->next;

}

}

node \*newLink = new node();

newLink->key = newKey;

newLink->data = data;

if(current==last)

{

newLink->next = NULL;

last = newLink;

}

else

{

newLink->next = current->next;

current->next = newLink;

}

newLink->prev = current;

current->next = newLink;

return true;

}

main()

{

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

cout<<"\nList(First to last): ";

displayForward();

cout<<"\nList(Last to first): ";

displayBackward();

cout<<"\nList, after deleting first record: ";

deleteFirst();

displayForward();

cout<<"\nList, after deleting last record:: ";

deleteLast();

displayForward();

cout<<"\nList, insert after key(4): ";

insertAfter(4,7,13);

displayForward();

cout<<"\nList, insert delete key(4): ";

del(4);

displayForward();

}

**CircularLinked\_list:**

#include<iostream>

using namespace std;

struct Node

{

int data;

struct Node \*next;

};

struct Node\* head = NULL;

void insert (int newdata)

{

struct Node \*newnode = (struct Node \*)malloc(sizeof(struct Node));

struct Node \*ptr = head;

newnode->data = newdata;

newnode->next = head;

if (head!= NULL)

{

while(ptr->next != head)

ptr = ptr->next;

ptr->next = newnode;

}

else

newnode->next = newnode;

head = newnode;

}

void display()

{

struct Node\* ptr;

ptr = head;

do

{

cout<<ptr->data<<" ";

ptr = ptr->next;

}

while(ptr !=head);

}

int count\_node() {

    struct Node\* temp = head;

    int count = 0;

    if (head != NULL) {

        do {

            count++;

            temp = temp->next;

        } while (temp != head);

    }

    return count;

}

void delete\_node(int key) {

    if (head == NULL) {

        cout << "List is empty. Cannot delete.\n";

        return;

    }

    struct Node\* temp = head;

    struct Node\* prev = NULL;

    // Find the node with the given key

    while (temp->data != key) {

        if (temp->next == head) {

            cout << "Node with key " << key << " not found.\n";

            return;

        }

        prev = temp;

        temp = temp->next;

    }

    if (temp->next == head) {

        head = NULL;

        free(temp);

    } else if (temp == head) {

        prev = head;

        while (prev->next != head)

            prev = prev->next;

        head = temp->next;

        prev->next = head;

        free(temp);

    } else {

        prev->next = temp->next;

        free(temp);

    }

}

int main()

{

insert(3);

insert(1);

insert(7);

insert(2);

insert(9);

cout<<"\nthe ciruclar linked list is: ";

display();

cout<<"\n Number of nodes:"<< count\_node();

int key\_to\_delete = 7;

    delete\_node(key\_to\_delete);

    cout << "\nAfter deleting node with key " << key\_to\_delete << ": ";

display();

cout<<"\n Number of nodes:"<< count\_node();

return 0;

}

**Fold\_boundary:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

#include<math.h>

#include <iostream>

using namespace std;

int count\_digits(int key){

int count=0;

while(key!=0){

key/=10;

++count;

}

return count;

}

int reversDigits(int num) {

int rev\_num = 0;

while (num > 0)  {

rev\_num = rev\_num \* 10 + num % 10;

num = num / 10;

}

return rev\_num;

}

int fold\_boundary(int key,int size){

int key\_roll=key;

int key\_sum=0;

int key\_frac=0;

int middle=0;

int left=0;

int right=0;

int digits=0;

int key\_length=0;

int fraction = size;

key\_length = count\_digits(key\_roll);

key\_frac = key\_roll / (int)pow(10, (key\_length - fraction));

left=reversDigits(key\_frac);

key\_roll = key\_roll % (int)pow(10,3);

right=reversDigits(key\_roll);

digits = (int)log10(key) + 1;

middle= (int)(key / pow(10, digits/ 2)) % 10;

key\_sum = left +middle+ right;

return key\_sum % (int)pow(10, (fraction));

}

int main() {

cout<<"\n"<<fold\_boundary(3347878, 3);

cout<<"\n"<<fold\_boundary(1234678, 3);

return 0;

}

**Heap:**

#include <iostream>

using namespace std;

class Node {

public:

    int data;

    Node\* next;

    Node(int value) : data(value), next(nullptr) {}

};

class Heap {

private:

    Node\* root;

    Node\* insert(Node\* root, int value, bool isMaxHeap) {

        if (root == nullptr) {

            return new Node(value);

        }

        if ((isMaxHeap && value > root->data) || (!isMaxHeap && value < root->data)) {

            Node\* newNode = new Node(value);

            newNode->next = root;

            return newNode;

        }

        root->next = insert(root->next, value, isMaxHeap);

        if (root->next != nullptr && ((isMaxHeap && root->next->data > root->data) || (!isMaxHeap && root->next->data < root->data))) {

            swap(root->data, root->next->data);

        }

        return root;

    }

    void reheap(Node\* currentNode, bool isMaxHeap) {

        if (currentNode->next != nullptr && ((isMaxHeap && currentNode->next->data > currentNode->data) || (!isMaxHeap && currentNode->next->data < currentNode->data))) {

            swap(currentNode->data, currentNode->next->data);

            reheap(currentNode->next, isMaxHeap);

        }

    }

    Node\* deleteNode(Node\* root, int value) {

        if (root == nullptr) {

            return nullptr;

        }

        if (root->data == value) {

            Node\* temp = root->next;

            delete root;

            return temp;

        } else {

            root->next = deleteNode(root->next, value);

        }

        return root;

    }

public:

    Heap() : root(nullptr) {}

    void insert(int value, bool isMaxHeap) {

        root = insert(root, value, isMaxHeap);

    }

    void printHeap(Node\* root) {

        if (root != nullptr) {

            printHeap(root->next);

            cout << root->data << " ";

        }

    }

    void printHeap() {

        printHeap(root);

        cout << endl;

    }

    void performReheap(bool isMaxHeap) {

        if (root != nullptr) {

            reheap(root, isMaxHeap);

        }

    }

    void deleteElement(int value) {

        root = deleteNode(root, value);

    }

};

int main() {

    Heap maxHeap, minHeap;

    int numElements;

    cout << "Enter the number of elements: ";

    cin >> numElements;

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

        int element;

        cout << "Enter element " << i + 1 << " for Max Heap and Min Heap: ";

        cin >> element;

        maxHeap.insert(element, true); // Insert into Max Heap

        minHeap.insert(element, false); // Insert into Min Heap

    }

    cout << "Max Heap: ";

    maxHeap.printHeap();

    cout << "Min Heap: ";

    minHeap.printHeap();

   int newElementMax;

    cout << "Enter a new element to insert into the max heap: ";

    cin >> newElementMax;

    maxHeap.insert(newElementMax, true);

    cout << "Max Heap after inserting " << newElementMax << ": ";

    maxHeap.printHeap();

    maxHeap.performReheap(true);

    cout << "Max Heap after reheapup: ";

    maxHeap.printHeap();

    int newElementMin;

    cout << "Enter a new element to insert into the min heap: ";

    cin >> newElementMin;

    minHeap.insert(newElementMin, false);

    cout << "Min Heap after inserting " << newElementMin << ": ";

    minHeap.printHeap();

    minHeap.performReheap(false);

    cout << "Min Heap after reheapdown: ";

    minHeap.printHeap();

    int elementToDeleteMax;

    cout << "Enter the element to delete from the max heap: ";

    cin >> elementToDeleteMax;

     maxHeap.deleteElement(elementToDeleteMax);

    cout << "Max Heap after deleting " << elementToDeleteMax << ": ";

    maxHeap.printHeap();

    int elementToDeleteMin;

    cout << "Enter the element to delete from the min heap: ";

    cin >> elementToDeleteMin;

    minHeap.deleteElement(elementToDeleteMin);

    cout << "Min Heap after deleting " << elementToDeleteMin << ": ";

    minHeap.printHeap();

    return 0;

}

**CIRCULAR QUEUE using LINKED LIST**

#include <iostream>

struct Node {

    int data;

    Node\* next;

};

class CircularQueue {

private:

    Node\* front;

    Node\* rear;

    int size;

    int count; // Added a count variable to keep track of the number of elements

public:

    // Constructor to initialize the circular queue with a given capacity

    CircularQueue(int capacity) {

        front = NULL;

        rear = NULL;

        size = capacity;

        count = 0;

    }

    // Destructor to release the memory allocated for nodes

    ~CircularQueue() {

        while (!isEmpty()) {

            dequeue();

        }

    }

    bool isEmpty() {

        return front == NULL;

    }

    bool isFull() {

        // Compare the count of elements with the size

        return count == size;

    }

    void enqueue(int data) {

        if (isFull()) {

            std::cout << "Queue is full" << std::endl;

            return;

        }

        Node\* newNode = new Node{data, NULL};

        if (isEmpty()) {

            front = newNode;

            rear = newNode;

        } else {

            rear->next = newNode;

            rear = newNode;

        }

        count++; // Increment the count

    }

    int dequeue() {

        if (isEmpty()) {

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

            return -1;

        }

        int data = front->data;

        Node\* temp = front;

        if (front == rear) {

            front = NULL;

            rear = NULL;

        } else {

            front = front->next;

        }

        delete temp;

        count--; // Decrement the count

        return data;

    }

    int frontElement() {

        if (isEmpty()) {

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

            return -1;

        }

        return front->data;

    }

    void display() {

        if (isEmpty()) {

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

            return;

        }

 int main() {

    // Example usage of CircularQueue

    CircularQueue queue(5);

     Node\* temp = front;

        while (temp != NULL) {

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

            temp = temp->next;

        }

        std::cout << std::endl;

    }

};

e.enqueue(1);

    queue.enqueue(2);

    queue.enqueue(3);

    std::cout << "Front element: " << queue.frontElement() << std::endl;

    queue.dequeue();

    queue.display();

    queue.enqueue(4);

    queue.enqueue(5);

    queue.enqueue(6); // This will print "Queue is full" since the capacity is 5

        return 0;

}

**PRIORITY QUEUE USING LINKED LIST**

#include <iostream>

class Node {

public:

    int data;

    int priority;

    Node\* next;

    Node(int data, int priority) {

        this->data = data;

        this->priority = priority;

        next = nullptr;

    }

};

class PriorityQueue {

private:

    Node\* head;

public:

    PriorityQueue() {

        head = nullptr;

    }

    bool isEmpty() {

        return head == nullptr;

    }

    void enqueue(int data, int priority) {

        Node\* newNode = new Node(data, priority);

        if (isEmpty()) {

            head = newNode;

            return;

        }

        if (newNode->priority > head->priority) {

            newNode->next = head;

            head = newNode;

            return;

        }

        Node\* temp = head;

        while (temp->next != nullptr && temp->next->priority >= newNode->priority) {

            temp = temp->next;

        }

        newNode->next = temp->next;

        temp->next = newNode;

    }

    int dequeue() {

        if (isEmpty()) {

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

            return -1;

        }

        int data = head->data;

        Node\* temp = head;

        head = head->next;

        delete temp;

        return data;

    }

    int frontElement() {

        if (isEmpty()) {

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

            return -1;

        }

        return head->data;

    }

    void display() {

        if (isEmpty()) {

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

            return;

        }

        Node\* temp = head;

        while (temp != nullptr) {

            std::cout << temp->data << "(" << temp->priority << ") ";

            temp = temp->next;

        }

        std::cout << std::endl;

    }

};

int main() {

    PriorityQueue pq;

    pq.enqueue(10, 2);

    pq.enqueue(20, 1);

    pq.enqueue(30, 3);

    pq.display(); // Output: 30(3) 10(2) 20(1)

    std::cout << "Front element: " << pq.frontElement() << std::endl; // Output: Front element: 30

    pq.dequeue();

    pq.display(); // Output: 10(2) 20(1)

    return 0;

}

**SIMPLE QUEUE USING LINKED LIST**

#include <iostream>  // Add this line for cout and endl

class Node {

public:

    int data;

    Node \*next;

    Node(int data) {

        this->data = data;

        next = nullptr;

    }

};

class LinkedListQueue {

private:

    Node \*front;

    Node \*rear;

public:

    LinkedListQueue() {

        front = nullptr;

        rear = nullptr;

    }

    bool isEmpty() {

        return front == nullptr;

    }

    void enqueue(int data) {

        Node \*newNode = new Node(data);

        if (isEmpty()) {

            front = newNode;

            rear = newNode;

            return;

        }

        rear->next = newNode;

        rear = newNode;

    }

    int dequeue() {

        if (isEmpty()) {

            std::cout << "Queue is empty" << std::endl;  // Use std:: for cout and endl

            return -1;

        }

        int data = front->data;

        Node \*temp = front;

        front = front->next;

        delete temp;

        if (front == nullptr) {

            rear = nullptr;

        }

        return data;

    }

    int frontElement() {

        if (isEmpty()) {

            std::cout << "Queue is empty" << std::endl;  // Use std:: for cout and endl

            return -1;

        }

        return front->data;

    }

    void display() {

        if (isEmpty()) {

            std::cout << "Queue is empty" << std::endl;  // Use std:: for cout and endl

            return;

        }

        Node \*temp = front;

        while (temp != nullptr) {

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

            temp = temp->next;

        }

        std::cout << std::endl;

    }

};

int main() {

    LinkedListQueue queue;

    queue.enqueue(10);

    queue.enqueue(20);

    queue.enqueue(30);

    queue.display(); // Output: 10 20 30

    std::cout << "Front element: " << queue.frontElement() << std::endl; // Output: Front element: 10

    queue.dequeue();

    queue.display(); // Output: 20 30

    return 0;

}

**DOUBLE ENDED QUEUE USING LINKED LIST**

#include <iostream>

struct Node {

    int data;

    Node\* prev;

    Node\* next;

};

class Deque {

private:

    Node\* front;

    Node\* rear;

    int size;

public:

    Deque(int capacity) {

        front = nullptr;

        rear = nullptr;

        size = capacity;

    }

    bool isEmpty() {

        return front == nullptr;

    }

    bool isFull() {

        return size == 0;

    }

    void pushFront(int data) {

        if (isFull()) {

            std::cout << "Deque is full" << std::endl;

            return;

        }

        Node\* newNode = new Node{ data, nullptr, front };

        if (isEmpty()) {

            front = newNode;

            rear = newNode;

        } else {

            front->prev = newNode;

            front = newNode;

        }

        size--;

    }

    void pushBack(int data) {

        if (isFull()) {

            std::cout << "Deque is full" << std::endl;

            return;

        }

        Node\* newNode = new Node{ data, rear, nullptr };

        if (isEmpty()) {

            front = newNode;

            rear = newNode;

        } else {

            rear->next = newNode;

            rear = newNode;

        }

        size--;

    }

    int popFront() {

        if (isEmpty()) {

            std::cout << "Deque is empty" << std::endl;

            return -1;

        }

        int data = front->data;

        Node\* temp = front;

        if (front == rear) {

            front = nullptr;

            rear = nullptr;

        } else {

            front = front->next;

            front->prev = nullptr;

        }

        delete temp;

        size++;

        return data;

    }

    int popBack() {

        if (isEmpty()) {

            std::cout << "Deque is empty" << std::endl;

            return -1;

        }

        int data = rear->data;

        Node\* temp = rear;

        if (front == rear) {

            front = nullptr;

            rear = nullptr;

        } else {

            rear = rear->prev;

            rear->next = nullptr;

        }

        delete temp;

        size++;

        return data;

    }

    int frontElement() {

        if (isEmpty()) {

            std::cout << "Deque is empty" << std::endl;

            return -1;

        }

        return front->data;

    }

    int rearElement() {

        if (isEmpty()) {

            std::cout << "Deque is empty" << std::endl;

            return -1;

        }

        return rear->data;

    }

    void display() {

        if (isEmpty()) {

            std::cout << "Deque is empty" << std::endl;

            return;

        }

        Node\* temp = front;

        while (temp != nullptr) {

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

            temp = temp->next;

        }

        std::cout << std::endl;

    }

};

int main() {

    Deque deque(5);

    deque.pushFront(1);

    deque.pushBack(2);

    deque.pushFront(3);

    deque.display(); // Output: 3 1 2

    std::cout << "Front element: " << deque.frontElement() << std::endl; // Output: Front element: 3

    std::cout << "Rear element: " << deque.rearElement() << std::endl;   // Output: Rear element: 2

    deque.popFront();

    deque.display(); // Output: 1 2

    return 0;

}

**SIMPLE QUEUE USING ARRAY LIST**

#include <iostream>

class ArrayQueue {

private:

    int\* arr;

    int front;

    int rear;

    int capacity;

public:

    ArrayQueue(int capacity) {

        this->capacity = capacity;

        arr = new int[capacity];

        front = 0;

        rear = -1;

    }

    bool isEmpty() {

        return front > rear;

    }

    bool isFull() {

        return rear == capacity - 1;

    }

    void enqueue(int data) {

        if (isFull()) {

            std::cout << "Queue is full" << std::endl;

            return;

        }

        rear++;

        arr[rear] = data;

    }

    int dequeue() {

        if (isEmpty()) {

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

            return -1;

        }

        int data = arr[front];

        front++;

        return data;

    }

    int frontElement() {

        if (isEmpty()) {

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

            return -1;

        }

        return arr[front];

    }

    void display() {

        if (isEmpty()) {

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

            return;

        }

        for (int i = front; i <= rear; i++) {

            std::cout << arr[i] << " ";

        }

        std::cout << std::endl;

    }

};

int main() {

    // Example usage of ArrayQueue

    ArrayQueue myQueue(5);

    myQueue.enqueue(1);

    myQueue.enqueue(2);

    myQueue.enqueue(3);

    myQueue.display();  // Output: 1 2 3

    std::cout << "Front element: " << myQueue.frontElement() << std::endl;  // Output: Front element: 1

    myQueue.dequeue();

    myQueue.display();  // Output: 2 3

        return 0;

}

**Insertion\_BST**

#include <bits/stdc++.h>

using namespace std;

struct node

{

int key;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

struct node\* temp = (struct node\*)malloc(

sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

{

node->left = insert(node->left, key);

}

else if (key > node->key)

{

node->right = insert(node->right, key);

}

return node;

}

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

cout << root->key << " ";

inorder(root->right);

}

}

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

inorder(root);

return 0;

}

**Deletion\_bst**

#include <iostream>

struct Node {

int key;

Node\* left;

Node\* right;

};

Node\* newNode(int item) {

Node\* temp = new Node;

temp->key = item;

temp->left = temp->right = nullptr;

return temp;

}

void inorder(Node\* root) {

if (root != nullptr) {

inorder(root->left);

std::cout << root->key << " ";

inorder(root->right);

}

}

Node\* insert(Node\* node, int key) {

if (node == nullptr)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

Node\* deleteNode(Node\* root, int k) {

if (root == nullptr)

return root;

if (root->key > k) {

root->left = deleteNode(root->left, k);

} else if (root->key < k) {

root->right = deleteNode(root->right, k);

} else {

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

} else {

Node\* succParent = root;

Node\* succ = root->right;

while (succ->left != nullptr) {

succParent = succ;

succ = succ->left;

}

if (succParent != root)

succParent->left = succ->right;

else

succParent->right = succ->right;

root->key = succ->key;

delete succ;

}

}

return root;

}

int main() {

Node\* root = nullptr;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

std::cout << "Original BST: ";

inorder(root);

std::cout << "\n\nDelete a Leaf Node: 40\n";

root = deleteNode(root, 40);

std::cout << "Modified BST tree after deleting Leaf Node:\n";

inorder(root);

return 0;

}

**Inorder\_BST**

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

struct node\* temp

= new struct node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

cout << root->key << " ";

inorder(root->right);

}

}

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

inorder(root);

return 0;

}

**PostOrder\_BST**

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

struct node\* temp

= new struct node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

void postorder(struct node \*root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

cout<<root->key<<" ";

}

}

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

postorder(root);

return 0;

}

**Preorder\_BST**

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

struct node\* newNode(int item)

{

struct node\* temp

= new struct node;

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

struct node\* insert(struct node\* node, int key)

{

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

void preorder(struct node \*root) {

if (root != NULL) {

cout<<root->key<<" ";

preorder(root->left);

preorder(root->right);

}

}

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

preorder(root);

return 0;

}