

**CT-159**

**Data Structures Algorithms &Applications**

**DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY**

**(SPECIALIZATION IN DATA SCIENCE)**

**NAME: MUHAMMAD KHUZAIMA HASSAN**

**ROLL NO: DT-22302**

**SUBMITTED TO: MISS FAIZA NASEEM**

**SUBMISSION DEADLINE: 17-JAN-2024**

**SEMESTER: FALL-2023**

**BATCH: 2022**

**LAB 1**

**1. Write a C++ code to copy data of a 2D array in a 1D array using Column Major Order.**

**PROGRAM**

#include <iostream>

using namespace std;

int main()

{

int rows,cols;

cout<<”enter number of rows: “;

cin>>rows;

cout<<”enter number of columns: “;

cin>>cols;

int darr[rows][cols];

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

{

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

{

cout<<”enter element at row “<<i<<” and column “<<j<<”: “;

cin>>darr[i][j];

}

}

cout<<”\nprinting 2d array\n”;

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

{

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

{

cout<<” element at row “<<i<<” and column “<<j<<”: “;

cout<<darr[i][j]<<endl;

}

}

// calculate the size of the 1d array

int onedsize = rows \* cols;

int\* onedarray = new int[onedsize];

int index =0;

//int 1d arr[index];

cout<<”copying data from 2d array to 1d array”;

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

{

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

{

onedarray[index]=darr[i][j];

index++;

}

}

cout<<”\nprinting 1d array\n”;

cout << “1d array (column major order): “;

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

{

cout << onedarray[i] << “ “;

}

}

**2. Write a program to calculate the CGPA of students of all subjects of a single semester.**

**Assume all the courses have the same credit hour (let’s assume 3 credit hours).**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Data Structure** | **Programming for AI** | **Digital Logic Design** | **Probability & Statistics** | **Finance & Accounting** |
| **Ali** | **3.66** | **3.33** | **4.0** | **3.0** | **2.66** |
| **Hiba** | **3.33** | **3.0** | **3.66** | **3.0** | **---** |
| **Asma** | **4.0** | **3.66** | **2.66** | **---** | **---** |
| **Zain** | **2.66** | **2.33** | **4.0** | **---** | **---** |
| **Faisal** | **3.33** | **3.66** | **4.0** | **3.0** | **3.33** |

**PROGRAM**

#include <iostream>

#include <string>

#include <vector>

struct student {

std::string name;

std::vector<double> grades;

};

// function to calculate cgpa for a student

double calculatecgpa(const student& student) {

double totalpoints = 0.0;

int totalcourses = 0;

for (const double& grade : student.grades) {

if (grade != -1) { // exclude missing courses

totalpoints += grade;

totalcourses++;

}

}

return (totalcourses > 0) ? (totalpoints / totalcourses) : 0.0;

}

int main() {

// define the courses and credit hours

const int numcourses = 5;

const double credithours = 3.0;

// create student records

std::vector<student> students = {

{“ali”, {3.66, 3.33, 4.0, 3.0, 2.66}},

{“hiba”, {3.33, 3.0, 3.66, 3.0, -1}},

{“asma”, {4.0, 3.66, 2.66, -1, -1}},

{“zain”, {2.66, 2.33, 4.0, -1, -1}},

{“faisal”, {3.33, 3.66, 4.0, 3.0, 3.33}}

};

// calculate and display cgpa for each student

for (const student& student : students) {

double cgpa = calculatecgpa(student);

std::cout << student.name << “’s cgpa: “ << cgpa << std::endl;

}

return 0;

}

**3. Suppose you are planning a picnic. You want to store the names of the students of your**

**class who have submitted their contribution money. Write a program that uses jagged**

**array to store the students’ names and then perform a search for your name.**

**PROGRAM**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

int main() {

// Prompt the user to input the number of students

int num\_students;

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

cin >> num\_students;

// Create a jagged array to store names

vector<vector<string>> students(1);

// Populate the jagged array with user-defined names

for (int i = 0; i < num\_students; ++i) {

string name;

cout << "Enter the name of student " << i + 1 << ": ";

cin >> name;

students[0].push\_back(name);

}

// Prompt the user for the name to search

string search\_name;

cout << "Enter the name to search: ";

cin >> search\_name;

// Perform the search

bool found = false;

for (const auto &name : students[0]) {

if (name == search\_name) {

found = true;

break;

}

}

// Display the result of the search

if (found) {

cout << search\_name << " has submitted the contribution." << endl;

} else {

cout << search\_name << " has not submitted the contribution." << endl;

}

return 0;

}

**4. Using the abstract data Type of a Matrix given below, write a program that**

**1. Input a 4\*3 matrix from user in 2D array**

**2. Map this array in 1D array using Row major order**

**3. Input second matrix of 3\*4 in 2D array**

**4. Map this array in 1D array using Row major order.**

**5. Now perform matrix multiplication on these 1D arrays**

**6. Save the result back in a 2D array.**

**PROGRAM**

#include <iostream>

using namespace std;

class matrix {

int \*\*p;

int r;

int c;

int \*rowmajor;

int \*multiply1D;

public:

matrix(int row, int col);

void input2D();

void disp2D();

void dispRowMajor();

void Multiply\_rowMajor(matrix &x);

void rowMajor\_2D();

~matrix();

};

matrix::matrix(int row, int col) {

r = row;

c = col;

p = new int \*[r];

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

p[i] = new int[c];

}

rowmajor = new int[r \* c];

}

void matrix::input2D() {

cout << "Enter elements for " << r << "x" << c << " matrix:" << endl;

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

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

cin >> p[i][j];

} }}

void matrix::disp2D() {

cout << "2D Matrix:" << endl;

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

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

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

}

cout << endl;

}}

void matrix::dispRowMajor() {

cout << "Row Major Order Matrix:" << endl;

int k = 0;

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

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

rowmajor[k++] = p[i][j];

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

} }

cout << endl;

}

void matrix::Multiply\_rowMajor(matrix &x) {

multiply1D = new int[r \* x.c];

int k = 0;

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

for (int j = 0; j < x.c; j++) {

int sum = 0;

for (int l = 0; l < c; l++) {

sum += rowmajor[i \* c + l] \* x.rowmajor[l \* x.c + j];

}

multiply1D[k++] = sum;

}}}

void matrix::rowMajor\_2D() {

cout << "Resultant 2D Matrix:" << endl;

int \*\*resultant = new int \*[r];

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

resultant[i] = new int[c];

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

resultant[i][j] = multiply1D[i \* c + j];

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

}

cout << endl;

}}

matrix::~matrix() {

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

delete[] p[i];

}

delete[] p;

delete[] rowmajor;

delete[] multiply1D;

}

int main() {

matrix a(4, 3);

a.input2D();

a.disp2D();

a.dispRowMajor();

matrix b(3, 4);

b.input2D();

b.disp2D();

b.dispRowMajor();

a.Multiply\_rowMajor(b);

a.rowMajor\_2D();

return 0;

}

**Implement this question for any number of rows and columns using class “matrix”.**

**PROGRAM**

#include <iostream>

using namespace std;

class matrix {

int \*\*p;

int r;

int c;

int \*rowmajor;

int \*multiply1D;

public:

matrix(int row, int col);

void input2D();

void disp2D();

void dispRowMajor();

void Multiply\_rowMajor(matrix &x);

void rowMajor\_2D();

~matrix();

};

matrix::matrix(int row, int col) {

r = row;

c = col;

p = new int \*[r];

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

p[i] = new int[c]; }

rowmajor = new int[r \* c];

}

void matrix::input2D() {

cout << "Enter elements for " << r << "x" << c << " matrix:" << endl;

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

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

cin >> p[i][j];

} }}

void matrix::disp2D() {

cout << "2D Matrix:" << endl;

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

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

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

}

cout << endl;

}}

void matrix::dispRowMajor() {

cout << "Row Major Order Matrix:" << endl;

int k = 0;

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

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

rowmajor[k++] = p[i][j];

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

} }

cout << endl;

}

void matrix::Multiply\_rowMajor(matrix &x) {

multiply1D = new int[r \* x.c];

int k = 0;

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

for (int j = 0; j < x.c; j++) {

int sum = 0;

for (int l = 0; l < c; l++) {

sum += rowmajor[i \* c + l] \* x.rowmajor[l \* x.c + j];

}

multiply1D[k++] = sum;

} }}

void matrix::rowMajor\_2D() {

cout << "Resultant 2D Matrix:" << endl;

int \*\*resultant = new int \*[r];

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

resultant[i] = new int[c];

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

resultant[i][j] = multiply1D[i \* c + j];

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

}

cout << endl; }}

matrix::~matrix() {

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

delete[] p[i]; }

delete[] p;

delete[] rowmajor;

delete[] multiply1D;}

int main() {

int row1, col1, row2, col2;

cout << "Enter the number of rows and columns for the first matrix:" << endl;

cin >> row1 >> col1;

matrix a(row1, col1);

a.input2D();

cout << "Enter the number of rows and columns for the second matrix:" << endl;

cin >> row2 >> col2;

matrix b(row2, col2);

b.input2D();

a.disp2D();

a.dispRowMajor();

b.disp2D();

b.dispRowMajor();

a.Multiply\_rowMajor(b);

a.rowMajor\_2D();

return 0;}

**5. Write a program that creates a 2D array of 5x5 values of type boolean. Suppose indices**

**represent cities and that the value at row i, column j of a 2D array is true just in case i and**

**j are direct neighbors and false otherwise. Use initializer list to instantiate and initialize**

**your array to represent the following configuration: (\* means “neighbors”)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Khi** | **Hyd** | **Jam** | **Mpk** | **Kot** |
| **Khi** |  | **\*** |  |  |  |
| **Hyd** | **\*** |  | **\*** |  |  |
| **Jam** |  | **\*** |  |  | **\*** |
| **Mpk** |  | **\*** | **\*** |  |  |
| **Kot** |  | **\*** | **\*** | **\*** |  |

**PROGRAM**

#include <iostream>

#include <vector>

using namespace std;

bool cities[5][5] = {

{false, true, false, false, false},

{true, false, true, false, false},

{false, true, false, false, true},

{false, true, true, false, false},

{false, true, true, true, false}};

bool checkCommonNeighbor(int city1, int city2) {

for (int neighbor = 0; neighbor < 5; neighbor++) {

if (cities[city1][neighbor] && cities[city2][neighbor]) {

return true;

}

}

return false;

}

int main() {

int city1, city2;

string city[5]={"Khi","Hyd","Jam","Mpk","Kot"};

cout << "City Map(1 means 'neighbors'):" << endl;

cout << "\tKhi\tHyd\tJam\tMpk\tKot" << endl;

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

{

cout<<city[i]<<"\t";

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

{

cout<<cities[i][j]<<"\t";

}

cout<<endl;

}

cout << "Enter city 1: ";

cin >> city1;

cout << "Enter city 2: ";

cin >> city2;

if (checkCommonNeighbor(city1, city2)) {

cout << "Cities " << city1 << " and " << city2 << " have a common neighbor." << endl;

} else {

cout << "Cities " << city1 << " and " << city2 << " do not have a common neighbor." << endl;

}

return 0;

}

**LAB 2**

**1. If the array is already sorted, we don’t want to continue with the comparisons. This can**

**be achieved with modified bubble sort. Update the code in example 02 to have a**

**modified bubble sort function.**

**PROGRAM**

#include <iostream>

using namespace std;

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

int i, j;

bool swapped;

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

swapped = false;

// Last i elements are already in place

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

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

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

swapped = true;

}

}

// If no two elements were swapped in the inner loop, then the array is already sorted

if (!swapped) {

break;

} }}

int main() {

int n;

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

cin >> n;

int arr[n];

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

cout << "Enter the element at " << i + 1 << ": ";

cin >> arr[i];

}

bubbleSort(arr, n);

cout << "PRINTING SORTED ARRAY\n";

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

cout << "Element at " << i + 1 << ": " << arr[i] << endl;

}

return 0;

}

**2. Create a Person class which has following attributes:**

**First Name, Last Name, Birth Year, Birth Month, Birth Date**

**Develop C++ solution such that day month and year are taken as input for N persons and**

**perform Sorting based on year, month and day using Selection Sort.**

**PROGRAM**

#include <iostream>

#include <vector>

using namespace std;

class Person {

private:

string firstName;

string lastName;

int birthYear;

int birthMonth;

int birthDate;

public:

Person(string firstName, string lastName, int birthYear, int birthMonth, int birthDate) {

this->firstName = firstName;

this->lastName = lastName;

this->birthYear = birthYear;

this->birthMonth = birthMonth;

this->birthDate = birthDate;}

string getFirstName() const {

return firstName;}

string getLastName() const {

return lastName; }

int getBirthYear() const {

return birthYear;}

int getBirthMonth() const {

return birthMonth; }

int getBirthDate() const {

return birthDate;}

void print() const {

cout << "\nFirst Name: " << firstName << endl;

cout << "Last Name: " << lastName << endl;

cout << "Birth Year: " << birthYear << endl;

cout << "Birth Month: " << birthMonth << endl;

cout << "Birth Date: " << birthDate << endl;

}};

void selectionSort(vector<Person>& persons) {

int n = persons.size();

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

int minIndex = i;

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

if (persons[j].getBirthYear() < persons[minIndex].getBirthYear()) {

minIndex = j;

} else if (persons[j].getBirthYear() == persons[minIndex].getBirthYear()) {

if (persons[j].getBirthMonth() < persons[minIndex].getBirthMonth()) {

minIndex = j;

} else if (persons[j].getBirthMonth() == persons[minIndex].getBirthMonth()) {

if (persons[j].getBirthDate() < persons[minIndex].getBirthDate()) {

minIndex = j;

}} } }

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

}}

int main() {

int n; // Number of persons

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

cin >> n;

vector<Person> persons;

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

string firstName, lastName;

int birthYear, birthMonth, birthDate;

cout<<" \nPERSON NUMBER "<<i+1<<"\n";

cout<<"Enter first name: ";

cin >> firstName;

cout<<"Enter last name: ";

cin>> lastName;

cout<<"Enter Birth year: ";

cin>> birthYear;

cout<<"Enter Birth month: ";

cin>> birthMonth;

cout<<"Enter Birth date: ";

cin>> birthDate;

Person person(firstName, lastName, birthYear, birthMonth, birthDate);

persons.push\_back(person);

}

selectionSort(persons);

for (Person person : persons) {

person.print();

cout << endl;

}

return 0;

}

**3. Given an array arr[ ] of length N consisting cost of N toys and an integer K the amount**

**with you. The task is to find maximum number of toys you can buy with K amount.**

**Test Case:**

**Input: N = 7, K = 50, arr[] = {1, 12, 5, 111, 200, 1000, 10}**

**Output: 4**

**Explanation: The costs of the toys. You can buy are 1, 12, 5 and 10.**

**PROGRAM**

#include <iostream>

using namespace std;

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

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;} }

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

}}

int maximumToys(int arr[], int n, int k) {

selectionSort(arr, n);

int count = 0;

int totalCost = 0;

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

if (totalCost + arr[i] <= k) {

totalCost += arr[i];

count++;

cout<<"\nYou can buy: "<<arr[i];

} else {

break; } }

return count;}

int main() {

int n;

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

cin >> n;

int arr[n];

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

cout << "Enter the element no "<<i+1<<": ";

cin >> arr[i]; }

int k;

cout << "Enter the budget: ";

cin >> k;

selectionSort(arr, n);

cout<<"\tSorted array of elements is \n";

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

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

int result = maximumToys(arr, n, k);

cout << "\nMaximum number of toys you can buy: " << result << endl;

return 0;

}

**4. Create a single class Sort, which will provide the user the option to choose between all 4**

**sorting techniques. The class should have following capabilities:**

** Take an array and a string (indicating the user choice for sorting technique) as**

**input and perform the desired sorting.**

** Should allow the user to perform analysis on a randomly generated array. The**

**analysis provides number of comparisons and number of swaps performed for**

**each technique.**

** After printing all the results, the class should highlight the best and worst**

**techniques.**

**PROGRAM**

#include <iostream>

#include <string>

#include <cstdlib>

#include <ctime>

using namespace std;

class Sort {

private:

int comparisons;

int swaps;

public:

Sort() : comparisons(0), swaps(0) {}

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

comparisons = 0;

swaps = 0;

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

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

comparisons++;

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

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

swaps++;

}} }}

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

comparisons = 0;

swaps = 0;

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

int minIndex = i;

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

comparisons++;

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

minIndex = j;

}}

if (minIndex != i) {

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

swaps++;

}}}

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

comparisons = 0;

swaps = 0;

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

int key = arr[i];

int j = i - 1;

comparisons++;

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

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

j--;

swaps++;

comparisons++;

}

arr[j + 1] = key;

}}

void performSort(int arr[], int n, string sortType) {

if (sortType == "Bubble") {

bubbleSort(arr, n);

} else if (sortType == "Selection") {

selectionSort(arr, n);

} else if (sortType == "Insertion") {

insertionSort(arr, n);

} }

void printAnalysis(string sortType) {

cout << "Analysis for " << sortType << " Sort:" << endl;

cout << "Number of comparisons: " << comparisons << endl;

cout << "Number of swaps: " << swaps << endl;

cout << endl;

}

};

int main() {

srand(time(0)); // Seed for random number generation

int n;

cout << "Enter the size of the array: ";

cin >> n;

int arr[n];

cout << "Randomly generated array: ";

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

arr[i] = rand() % 100; // Random numbers between 0 and 99

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

}

cout << endl;

string sortType;

cout << "Enter the sorting technique (Bubble/Selection/Insertion): ";

cin >> sortType;

Sort sorter;

sorter.performSort(arr, n, sortType);

sorter.printAnalysis(sortType);

// Highlight the best and worst techniques

cout << "Best technique: Selection Sort" << endl;

cout << "Worst technique: Bubble Sort" << endl;

return 0;

}

**LAB 3**

1. Implement a singly linked list class with the following functions:

a) Insert a node at head

b) Insert a node at tail/end/back

c) Insert a node at any position

d) Delete a node by value

e) Delete head

f) Delete tail

g) Delete a node at any position.

3. Use the class of SLL created by you during the lab task 1. Do the following:

a) Reverse the linked list

b) Sort the contents of linked list

c) Find the duplicates in the linked list

PROGRAM:

#include<iostream>

using namespace std;

class Node{

    public:

    int data;

    Node\*next;

Node(int data){

    this->data=data;

    this->next=NULL;

}

~Node(){

     int value=this->data;

     if(this->next==NULL){

        delete next;

        this->next=NULL;

     }

     cout<<"Memory is free for "<<value<<endl;

}

};

void InsertAtHead(Node\*&head,int data){

    Node\*temp=new Node(data);

    temp->next=head;

    head=temp;

}

void InsertAtTail(Node\*&tail,int data){

    Node\*temp=new Node(data);

    tail->next=temp;

    tail=temp;

}

void InsertAtposition(Node\*&head,Node\*&tail,int position,int data){

    Node\*temp=head;

    if (position == 1){

        InsertAtHead(head,data);

        return;

    }

   int cnt=1;

   while(cnt<position-1){

    temp=temp->next;

    cnt++;

   }

   if (temp->next==NULL){

    InsertAtTail(tail,data);

    return;

   }

   Node\*Nodetoinsert=new Node(data);

   Nodetoinsert->next=temp->next;

   temp->next=Nodetoinsert;

}

void deletenodebyvalue(Node\*&head,int value){

    Node\*curr=head;

    Node\*prev=NULL;

    if (head != NULL && head->data == value) {

        head = head->next;

        delete curr;

        return;

    }

    int data=head->data;

    while(data!=value && curr!=NULL){

        prev=curr;

        curr=curr->next;

        data=curr->data;

    }

    if (curr==NULL){

        return;

    }

    prev->next=curr->next;

    curr->next=NULL;

    delete curr;}

void deletehead(Node\*&head){

    Node\*temp=head;

        head=head->next;

        temp->next=NULL;

        delete temp;}

void deletetail(Node\*& tail) {

  if (tail == NULL) {

    return;}

Node\* temp = tail;

if (temp->next != NULL) {

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

      temp = temp->next;}

    delete temp->next;

    temp->next = NULL;

    tail = temp;

} else {

    tail = NULL;

    delete temp;

  }}

void deletenode(Node\* &head,int position){

if (position==1){

 Node\*temp=head;

        head=head->next;

        temp->next=NULL;

        delete temp; }

    else{

        Node\*curr=head;

        Node\*prev=NULL;

        int cnt=1;

        while(cnt<position){

            prev=curr;

            curr=curr->next;

            cnt++; }

        prev->next=curr->next;

        curr->next=NULL;

        delete curr;

    }}

void print(Node\*&head){

    Node\*temp=head;

    while(temp!=NULL){

        cout<<temp->data<<" ";

        temp=temp->next;

    }

    cout<<endl;}

Node\*reversenode(Node\* &head){

    if (head==NULL || head->next==NULL){

        return head;}

    Node\*curr=head;

    Node\*prev=NULL;

    Node\*forward=NULL;

    while(curr!=NULL){

    forward=curr->next;

    curr->next=prev;

    prev=curr;

    curr=forward;}

    return prev; }

void SelectionSort(Node\*&head){

    Node\*temp=head;

    Node\*curr;

    Node\*min;

    while(temp!=NULL){

          min=temp;

          curr=temp->next;

          while(curr!=NULL){

            if (min->data > curr->data){

                min=curr;

            }

          curr=curr->next;}

         swap(min->data,temp->data);

          temp=temp->next;

          } }

void duplicates(Node\*&head){

    Node\*temp=head;

   int count = 0;

   while(temp!=NULL){

     Node\*curr=temp->next;

    while(curr!=NULL){

        if (curr->data==temp->data){

          count++; }

       curr=curr->next;}

   temp=temp->next; }

    cout<<"No of Duplicates: "<<count<<endl;

}

int main(){

Node\*Node1=new Node(10);

Node\*head=Node1;

    Node\*tail=Node1;

    print(head);

    InsertAtHead(head,13);

    print(head);

    InsertAtTail(tail,13);

    print(head);

    InsertAtposition(head,tail,2,10);

    print(head);

    SelectionSort(head);

    print (head);

    duplicates(head);

    deletehead(head);

    print(head);

   deletetail(tail);

    print(head);

    deletenodebyvalue(head,3);

    print(head);

   head=reversenode(head);   print(head);

return 0;}

2. Solve the following problem using a Singly Linked List. Given a singly linked list of

characters, write a function to make word out of given letters in the list. Test Case:

Input:C->S->A->R->B->B->E->L->NULL,

Output:S->C->R->A->B->B->L->E->NULL

PROGRAM:

#include <iostream>

class Node {

public:

    char data;

    Node\* next;

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

};

class SinglyLinkedList {

private:

    Node\* head;

public:

    SinglyLinkedList() : head(nullptr) {}

    void insertAtTail(char data) {

        Node\* newNode = new Node(data);

        if (!head) {

            head = newNode;

            return;

        }

        Node\* current = head;

        while (current->next) {

            current = current->next;

        }

        current->next = newNode;

    }

    void rearrangeToWord(const char\* word) {

        Node\* newHead = nullptr;

        for (int i = 0; word[i] != '\0'; ++i) {

            char currentChar = word[i];

            Node\* current = head;

            Node\* prev = nullptr;

            // Find the node with the current character

            while (current && current->data != currentChar) {

                prev = current;

                current = current->next;

            }

            // Move the found node to the new list

            if (current) {

                if (prev) {

                    prev->next = current->next;

                } else {

                    head = current->next;

                }

                current->next = newHead;

                newHead = current;

            }

        }

        // Update the original list with the rearranged nodes

        head = newHead;

    }

    void displayList() const {

        Node\* current = head;

        while (current) {

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

            current = current->next;

        }

        std::cout << "NULL\n";

    }

};

int main() {

    SinglyLinkedList linkedList;

    linkedList.insertAtTail('C');

    linkedList.insertAtTail('S');

    linkedList.insertAtTail('A');

    linkedList.insertAtTail('R');

    linkedList.insertAtTail('B');

    linkedList.insertAtTail('B');

    linkedList.insertAtTail('E');

    linkedList.insertAtTail('L');

    std::cout << "Original List: ";

    linkedList.displayList();

    const char\* targetWord = "SCRABBLE";

    linkedList.rearrangeToWord(targetWord);

    std::cout << "Rearranged List for \"" << targetWord << "\": ";

    linkedList.displayList();

    return 0;}

**LAB 4**

1. Create a doubly link list and perform the mentioned tasks.

a. Insert a new node at the end of the list.

b. Insert a new node at the beginning of list.

c. Insert a new node at given position.

d. Delete any node.

e. Print the complete doubly link list.

PROGRAM:

#include<iostream>

using namespace std;

class Node{

 public:

 int data;

 Node\*prev;

 Node\*next;

 // Constructor

 Node(int data){

    this->data=data;

    this->prev=NULL;

    this->next=NULL;

 }

 ~Node(){

    int value=this->data;

    if(this->next==NULL)

    {

        delete next ;

        this->next = NULL;

    }

    cout<<"Memory is free "<<value<<endl;

}};

void print(Node\*&head){

    Node\*temp=head;

    while(temp!=NULL){

        cout<<temp->data<<" ";

        temp=temp->next;

    }

    cout<<endl;

}

int getlength(Node\* &head){

    int len=0;

    Node\*temp=head;

    while(temp!=NULL){

        len++;

        temp=temp->next;

    }

    return len;

}

void InsertAtHead(Node\*&head,Node\*&tail,int data){

    if (head==NULL){

        Node\*temp=new Node(data);

        head=temp;

        tail=temp;

    }

    else{Node\*temp=new Node(data);

    temp->next=head;

    head->prev=temp;

    head=temp;

    temp->prev=NULL;}

}

void InsertAtTail(Node\*&head,Node\*&tail,int data){

    if(tail==NULL){

        Node\*temp=new Node(data);

        tail=temp;

        head=temp;

    }

    else{Node\*temp=new Node(data);

    tail->next=temp;

    temp->prev=tail;

    tail=temp;

    temp->next=NULL;}

}

void InsertAtPosition(Node\*&head,Node\*&tail,int position,int data){

    Node\*temp=head;

    if (position==1){

        InsertAtHead(head,tail,data);

        return;}

    int cnt=1;

    while(cnt<position-1){

        temp=temp->next;

        cnt++;

    }

    if (temp->next==NULL){

        InsertAtTail(head,tail,data);

        return;

    }

    Node\*Nodetoinsert=new Node(data);

    Nodetoinsert->next=temp->next;

    temp->next->prev=Nodetoinsert;

    temp->next=Nodetoinsert;

    Nodetoinsert->prev=temp;

}

void deletenode(Node\*&head,int position){

 if (position==1)

 {

   Node\*temp=head;

   temp->next->prev=NULL;

   head=temp->next;

   temp->next=NULL;

   delete temp;

 }

 else{

   Node\*curr=head;

   Node\*prev=NULL;

   int cnt = 1;

   while (cnt<position)

   {

    prev= curr;

    curr=curr->next;

    cnt++;

   }

   curr->prev=NULL;

   prev->next=curr->next;

   curr->next=NULL;

   delete curr;

     }

}

int main()

{    Node\*head=NULL;

    Node\*tail=NULL;

    print(head);

    InsertAtHead(head,tail,23);

    InsertAtHead(head,tail,3);

    print(head);

    cout<<getlength(head)<<endl;

    cout<<head->prev<<endl;

    InsertAtTail(head,tail,5);

    InsertAtTail(head,tail,1);

    print(head);

    cout<<tail->next<<endl;

   InsertAtPosition(head,tail,3,4);

   print(head);

   InsertAtPosition(head,tail,1,4);

   print(head);

   InsertAtPosition(head,tail,7,4);

   print(head);

   cout<<"Head: "<<head->data<<endl;

     cout<<"Tail: "<<tail->data<<endl;

   deletenode(head,1);

   print(head);

   deletenode(head,6);

   print(head);

   cout<<"LENGTH : "<<getlength(head)<<endl;

    return 0;

}

2. Create two doubly link lists, say L and M . List L should contain all even elements from 2 to

10 and list M should contain all odd elements from 1 to 9. Create a new list N by

concatenating list L and M.

3. Using the above created list N, sort the contents of list N is descending order.

PROGRAM:

#include<iostream>

using namespace std;

class Node{

    public:

    int data;

    Node\*next;

    Node\*prev;

    Node(int data) {

        this->data=data;

        this->next=NULL;

        this->prev=NULL;

    }

~Node(){

    int value=this->data;

    if (this->next==NULL){

        delete next;

        this->next=NULL;

    }

    cout<<"Memory is free for"<<value<<endl;

}

};

void insertathead(Node\*&head,Node\*&tail,int data){

    if (head==NULL){

        Node\*temp=new Node (data);

        head=temp;

        tail=temp;

    }

    else{

        Node\*temp=new Node(data);

        temp->next=head;

        head->prev=temp;

        head=temp;

        temp->prev=NULL;

    }

}

void insertattail(Node\*&head,Node\*&tail,int data){

    if (tail==NULL){

        Node\*temp=new Node(data);

        tail=temp;

        head=temp;

    }

    else{

        Node\*temp=new Node (data);

        tail->next=temp;

        temp->prev=tail;

        tail=temp;

        temp->next=NULL;

    }

}

void insertatposition(Node\*&head,Node\*&tail,int position,int data){

    if (position==1){

        insertathead(head,tail,data);

        return;

    }

  Node\*temp=head;

   int cnt =1;

    while(cnt<position-1){

        temp=temp->next;

    cnt++;

    }

    if (temp->next==NULL){

        insertattail(head,tail,data);

        return;

    }

   Node\* Nodetoinsert=new Node(data);

   Nodetoinsert->next=temp->next;

   temp->next->prev=Nodetoinsert;

   temp->next=Nodetoinsert;

   Nodetoinsert->prev=NULL;

}

void print (Node\*&head){

    Node\*temp=head;

    while (temp!=NULL){

        cout<<temp->data<<" ";

        temp=temp->next;

    }

    cout<<endl;

}

Node\* combineLists(Node\* head1, Node\* head2) {

    if (head1 == NULL) {

        return head2;

    }

    if (head2 == NULL) {

        return head1;

    }

    Node\* temp = head1;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = head2;

    head2->prev = temp;

    return head1;

}

void SelectionSort(Node\*&head){

    Node\*temp=head;

    Node\*curr;

    Node\*max;

    while(temp!=NULL){

          max=temp;

          curr=temp->next;

          while(curr!=NULL){

            if (max->data < curr->data){

                max=curr;

            }

          curr=curr->next;

          }

         if (max->data!=temp->data){

            swap(max->data,temp->data);

         }

           temp=temp->next;

          }

    }

int main()

{

    Node\*head1=NULL;

    Node\*tail1=NULL;

    cout<<"Printing List L : ";

    insertathead(head1,tail1,2);

    insertattail(head1,tail1,4);

    insertatposition(head1,tail1,3,6);

    insertatposition(head1,tail1,4,8);

    insertatposition(head1,tail1,5,10);

    print(head1);

    Node\*head2=NULL;

    Node\*tail2=NULL;

    cout<<"Printing List M : ";

    insertathead(head2,tail2,1);

    insertattail(head2,tail2,3);

    insertatposition(head2,tail2,3,5);

    insertatposition(head2,tail2,4,7);

    insertatposition(head2,tail2,5,9);

    print (head2);

    cout<<"Printing Combined Lists : ";

    Node\* combinedHead = combineLists(head1, head2);

    print(combinedHead);

    cout<<"Printing List in desending order : ";

    SelectionSort(combinedHead);

    print(combinedHead);

    return 0;}

4. Create a circular link list and perform the mentioned tasks.

a. Insert a new node at the end of the list.

b. Insert a new node at the beginning of list.

c. Insert a new node at given position.

d. Delete any node.

e. Print the complete doubly link list.

PROGRAM:

#include<iostream>

using namespace std;

class Node{

 public:

 int data;

 Node\*next;

Node(int data ){

    this->data=data;

    this->next=NULL;

 }

 ~Node(){

    int value=this->data;

    if(this->next==NULL){

        delete next;

        this->next=NULL;}

 cout<<"Memory is free for "<<value<<endl;

 }};

void InsertNode(Node\* &tail,int element,int data){

    if (tail==NULL){

        Node\*NewNode=new Node(data);

        tail=NewNode;

        NewNode->next=NewNode;

    }

    else{

        Node\*temp=tail;

        while(temp->data!=element){

            temp=temp->next;

        }

        Node\*New=new Node(data);

        New->next=temp->next;

        temp->next=New;

    }}

void print(Node\* &tail){

    Node\*temp=tail;

    if (tail == NULL){

        cout<<"List is empty"<<endl;

        return;}

    int cnt=0;

    do{

        cout<<tail->data<<" ";

        tail=tail->next;

        cnt++;

    }

    while(tail!=temp);

   cout<<endl<<"length: "<<cnt<<endl;

}

void DeleteNode(Node\*&tail,int value){

    if (tail == NULL){

        cout<<"List is empty"<<endl;

        return;}

    else{

        Node\*prev=tail;

        Node\*curr=prev->next;

        while(curr->data != value){

            prev=curr;

            curr=curr->next;

        }

        prev->next=curr->next;

        if (curr==prev){

            tail=NULL; }

      else if (tail==curr){

            tail=prev; }

        curr->next=NULL;

        delete curr;

}}

int main(){

Node\*tail=NULL;

InsertNode(tail,3,4);

    print(tail);

    InsertNode(tail,4,5);

    print(tail);

    InsertNode(tail,5,6);

    print(tail);

    InsertNode(tail,6,7);

    print(tail);

    InsertNode(tail,7,8);

    print(tail);

    InsertNode(tail,8,9);

    print(tail);

    DeleteNode(tail,4);

    print(tail);

    DeleteNode(tail,9);

    print(tail);

    DeleteNode(tail,7);

    print(tail);

    DeleteNode(tail,6);

    print(tail);

return 0;

}

5. Break the above-created circular linked list into two halves.

#include<iostream>

#include<utility>  // for std::pair

using namespace std;

class Node {

public:

    int data;

    Node\* next;

    Node(int data) {

        this->data = data;

        this->next = NULL;

    }

    ~Node() {

        int value = this->data;

        if (this->next == NULL) {

            delete next;

            this->next = NULL;

        }

        cout << "Memory is freed for " << value << endl;

    }

};

void InsertNode(Node\* &tail, int element, int data) {

    if (tail == NULL) {

        Node\* NewNode = new Node(data);

        tail = NewNode;

        NewNode->next = NewNode;

    }

    else {

        Node\* temp = tail;

        while (temp->data != element) {

            temp = temp->next;

        }

        Node\* New = new Node(data);

        New->next = temp->next;

        temp->next = New;

    }

}

void DeleteNode(Node\*& tail, int value) {

    if (tail == NULL) {

        cout << "List is empty" << endl;

        return;

    }

    else {

        Node\* prev = tail;

        Node\* curr = prev->next;

        while (curr->data != value) {

            prev = curr;

            curr = curr->next;

        }

        prev->next = curr->next;

        if (curr == prev) {

            tail = NULL;

        }

        else if (tail == curr) {

            tail = prev;

        }

        curr->next = NULL;

        delete curr;

    }

}

void print(Node\* &tail) {

    Node\* temp = tail;

    if (tail == NULL) {

        cout << "List is empty" << endl;

        return;

    }

    int cnt = 0;

    do {

        cout << tail->data << " ";

        tail = tail->next;

        cnt++;

    } while (tail != temp);

    cout << endl << "length: " << cnt << endl;

}

pair<Node\*, Node\*> breakCircularList(Node\* tail) {

    pair<Node\*, Node\*> result;

    if (tail == NULL || tail->next == tail) {

        // The list is empty or has only one element, cannot be split

        result.first = tail;

        result.second = NULL;

        return result;

    }

    // Use the tortoise and hare approach to find the midpoint

    Node\* slow = tail->next;

    Node\* fast = tail->next->next;

    while (fast != tail && fast->next != tail) {

        slow = slow->next;

        fast = fast->next->next;

    }

    // Set the heads and tails for the two halves

    result.first = tail->next;

    result.second = slow->next;

    // Break the circular link between the two halves

    slow->next = tail->next;

    tail->next = result.second;

    return result;

}

int main() {

    Node\* tail = NULL;

    InsertNode(tail, 3, 4);

    print(tail);

    InsertNode(tail, 4, 5);

    print(tail);

    InsertNode(tail, 5, 6);

    print(tail);

    InsertNode(tail, 6, 7);

    print(tail);

    InsertNode(tail, 7, 8);

    print(tail);

    InsertNode(tail, 8, 9);

    print(tail);

    pair<Node\*, Node\*> halves = breakCircularList(tail);

    cout << "First Half: ";

    print(halves.first);

    cout << "Second Half: ";

    print(halves.second);

    return 0;

}

**LAB 5**

1. Please write a program which performs the following tasks:

a. Make a left to right scan of the postfix expression

b. If the element is an operand push it on Stack

c. If the element is operator, evaluate it using as operands the correct number from stack

and pushing the result onto the stack.

**PROGRAM:**

#include<iostream>

#include<stack>

using namespace std;

int prec(char c){

    if(c=='^'){

        return 3;

    }

    else if(c=='\*' || c=='/'){

        return 2;

    }

    else if (c=='+' || c=='-'){

        return 1;

    }

    else{

        return -1;

    }

}

string infixtopostfix(string s){

    stack<char>st;

    string res;

    for(int i=0;i<s.length();i++){

        if ((s[i]>='a'&&s[i]<='z' ) || (s[i]>='A'&&s[i]<='Z') ){

            res+=s[i];

        }

        else if (s[i]=='('){

            st.push(s[i]);

        }

        else if (s[i]==')'){

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

              res+=st.top();

              st.pop();

             }

             if(!st.empty()){

                st.pop();

             }

        }

        else{

            while(!st.empty()&&prec(st.top())>prec(s[i])){

                res+=st.top();

                st.pop();

            }

            st.push(s[i]);

        }

    }

     while (!st.empty()){

        res+=st.top();

        st.pop();

     }

     return res;

}

int main(){

   cout<<infixtopostfix("(a-b/c)\*(a/k-l)")<<endl;

return 0;

}

2. A palindrome is a word, phrase, number, or another sequence of characters that reads the same

backward and forwards. Can you determine if a given string, s, is a palindrome? Write a

Program using stack for checking whether a string is palindrome or not.

**PROGRAM:**

#include <iostream>

#include <stack>

using namespace std;

void palindrome\_check(const string& str) {

    stack<char> st;

    int len = str.length();

    int mid = len / 2;

    for (int i = 0; i < mid; i++) {

        st.push(str[i]);

    }

    for (int i = mid + len % 2; i < len; i++) {

        if (str[i] != st.top()) {

            cout << "Not a Palindrome";

            return;

        }

        st.pop();

    }

    cout << "Palindrome";

}

int main() {

    string str;

    cout << "Enter the string: ";

    cin >> str;

    palindrome\_check(str);

    return 0;}

3. Write a program using stacks which takes an expression as input and determines whether the

delimiters are matched or not.

**PROGRAM:**

#include <iostream>

#include <stack>

using namespace std;

bool areDelimitersMatched(const string& expression) {

    stack<char> delimiterStack;

    for (char c : expression) {

        switch (c) {

            case '(':

            case '[':

            case '{':

                delimiterStack.push(c);

                break;

            case ')':

                if (delimiterStack.empty() || delimiterStack.top() != '(') {

                    return false;

                }

                delimiterStack.pop();

                break;

            case ']':

                if (delimiterStack.empty() || delimiterStack.top() != '[') {

                    return false;

                }

                delimiterStack.pop();

                break;

            case '}':

                if (delimiterStack.empty() || delimiterStack.top() != '{') {

                    return false;

                }

                delimiterStack.pop();

                break;

            // Ignore other characters

            default:

                break;

        }}

return delimiterStack.empty(); // Check if there are any unmatched opening delimiters}

int main() {

    string expression;

    cout << "Enter an expression: ";

    getline(cin, expression);

    if (areDelimitersMatched(expression)) {

        cout << "The delimiters are matched." << endl;

    } else {

        cout << "The delimiters are not matched." << endl;

    }

    return 0;}

**LAB 6**

1. Write the definition of all the functions listed in Example 01.

#include <iostream>

using namespace std;

const int ERROR\_INDICATOR = -1;

class Queue {

private:

    int front, rear, size;

    unsigned capacity;

    int\* array;

public:

    Queue(int siz) {

        capacity = siz;

        array = new int[capacity];

        front = size = 0;

        rear = capacity - 1;

    }

void insert(int j) {

        if (isFull()) {

            cout << "Queue is full. Cannot insert " << j << endl;

            return;

        }

        rear = (rear + 1) % capacity;

        array[rear] = j;

        size++;

        cout << j << " enqueued to queue\n";

    }

    int remove() {

        if (isEmpty()) {

            cout << "Queue is empty. Cannot dequeue\n";

            return ERROR\_INDICATOR;

        }

        int item = array[front];

        front = (front + 1) % capacity;

        size--;

        return item; }

int peek() {

        if (isEmpty()) {

            cout << "Queue is empty. No element to peek\n";

            return ERROR\_INDICATOR;

        }

        return array[front];

    }

bool isEmpty() {

        return (size == 0);

    }

    bool isFull() {

        return (size == capacity);

    }

    int getSize() {

        return size;}

  void display() {

        cout << "Queue: ";

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

            int index = (front + i) % capacity;

            cout << array[index] << " ";

        }

        cout << endl;}};

int main() {

    Queue myQueue(5);

    myQueue.insert(10);

    myQueue.insert(20);

    myQueue.insert(30);

    myQueue.display();

    cout << "Front element is: " << myQueue.peek() << endl;

    cout << myQueue.remove() << " dequeued from queue\n";

    myQueue.display();

    cout << "Front element is: " << myQueue.peek() << endl;

    cout << "Queue size is " << myQueue.getSize() << endl;

    return 0;

}

2. Please implement the Generic Queue definition using singly linked list (you may use the

Singly Linked List that you already developed in Lab # 3), you may also add any

functions needed in the Singly Linked List definition given in Lab # 3. Your

implementation should work for the main function given below.

**PROGRAM:**

#ifndef QUEUE\_H

#define QUEUE\_H

#include <iostream>

using namespace std;

// Singly Linked List Node

template <typename T>

struct Node {

    T data;

    Node\* next;

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

};

// Singly Linked List

template <typename T>

class LinkedList {

private:

    Node<T>\* head;

public:

    LinkedList() : head(nullptr) {}

    ~LinkedList() {

  while (head) {

  Node<T>\* temp = head;

            head = head->next;

            delete temp;

        } }

void insertFront(T value) {

        Node<T>\* newNode = new Node<T>(value);

        newNode->next = head;

        head = newNode;

    }

T removeFront() {

        if (isEmpty()) {

cerr << "Error: List is empty.\n";

return T(); // Assuming T() as a default value for the type T

        }

        Node<T>\* temp = head;

        head = head->next;

        T value = temp->data;

        delete temp;

  return value;

    }

bool isEmpty() const {

        return (head == nullptr);

    }

    T peekFront() const {

        if (isEmpty()) {

            cerr << "Error: List is empty.\n";

  return T(); // Assuming T() as a default value for the type T

}

return head->data;

}

};

// Generic Queue using Singly Linked List

template <typename T>

class Queue {

private:

    LinkedList<T> list;

public:

    void enqueue(T value) {

        list.insertFront(value);

    }

    T dequeue() {

        return list.removeFront();

    }

    T peek() const {

        return list.peekFront();

    }

  bool isEmpty() const {

        return list.isEmpty();

    }

};

#endif

#include "Queue.h"

#include "Queue.cpp"

#include <iostream>

using namespace std;

int main(){

Queue<int> \*q =new Queue<int>();

if(q->IsEmpty())

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

q->Put(1);

q->Put(2);

q->Put(3);

while (!q->IsEmpty()){

int value=q->Get(); cout<<value<<endl;

}

return 0;

}

3. Give the C++ code to implement the Generic Queue using array. Please change the

private data members as you are now using an array.

**PROGRAM:**

#include <iostream>

#include <stdexcept>

using namespace std;

template <typename T>

class Queue {

private:

    int front, rear, size;

    unsigned capacity;

    T\* array;

public:

    Queue(unsigned cap) : front(0), rear(cap - 1), size(0), capacity(cap) {

        array = new T[cap];

    }

    ~Queue() {

        delete[] array;

    }

    void enqueue(T item) {

  if (isFull()) {

            cerr << "Error: Queue is full. Cannot enqueue.\n";

            return;

        }

        rear = (rear + 1) % capacity;

        array[rear] = item;

        size++;

        cout << item << " enqueued to queue\n";

    }

    T dequeue() {

        if (isEmpty()) {

      cerr << "Error: Queue is empty. Cannot dequeue.\n";

            return T();

        }

        T item = array[front];

        front = (front + 1) % capacity;

        size--;

        return item;

    }

T peek() const {

if (isEmpty()) {

            cerr << "Error: Queue is empty. Cannot peek.\n";

            return T();

        }

        return array[front];

    }

    bool isEmpty() const {

return (size == 0); }

bool isFull() const {

return (size == capacity);

}

int getSize() const {

return size;

}

};

int main() {

  Queue<int> intQueue(5);

    intQueue.enqueue(10);

    intQueue.enqueue(20);

    intQueue.enqueue(30);

    cout << "Front element is: " << intQueue.peek() << endl;

    cout << intQueue.dequeue() << " dequeued from queue\n";

cout << "Front element is: " << intQueue.peek() << endl;

cout << "Is the queue empty? " << (intQueue.isEmpty() ? "Yes" : "No") << endl;

Queue<string> stringQueue(3);

stringQueue.enqueue("Hello");

stringQueue.enqueue("World");

    cout << "Front element is: " << stringQueue.peek() << endl;

return 0; }

**LAB 7**

1.Write a program and recurrence relation to find the Fibonacci series of n where n>2.

**PROGRAM:**

#include <iostream>

using namespace std;

int fibonacci(int n)

{

    if (n <= 1)

    {

        return n;

    }

return fibonacci(n - 1) + fibonacci(n - 2);

}

int main()

{

    int n;

    cout << "Enter a number for Fibonacci Series: ";

    cin >> n;

    for (int i = 1; i <= n; ++i)

    {

        cout << fibonacci(i) << " ";

    }

}

2. Write a program and recurrence relation to find the Factorial of n where n>2.

**PROGRAM:**

#include <iostream>

using namespace std;

int factorial(int n)

{

    if (n == 0)

    {

        return 1;

    }

    return n \* factorial(n - 1);

}

int main()

{

    int n;

    cout << "Enter a number for factorial :";

    cin >> n;

    int ans = factorial(n);

    cout << endl;

    cout << "Factorial: " << ans << endl;

    return 0;}

3. Write a recursive function which will take input from the user until a special character

(also selected by the user) is not entered. Then print all the input in reverse.

Sample Input:

Enter Special Character: !

Enter Character: A

Enter Character: B

Enter Character: C

Enter Character: !

Sample Output: C B A

**PROGRAM:**

#include <iostream>

using namespace std;

void reverseOutput(char specialChar, char input[], int &size)

{

    char currentChar;

    cout << "Enter characters (terminate with '" << specialChar << "'): ";

    if (cin >> currentChar && currentChar == specialChar)

    {

        return;

    }

    input[size++] = currentChar;

    reverseOutput(specialChar, input, size);

    cout << "PRINTING IN REVERSE: ";

    cout << input[--size] << " " << endl;

}

int main()

{

    char specialChar;

    const int maxSize = 100;

    char input[maxSize];

    int size = 0;

    cout << "Enter special character: ";

    cin >> specialChar;

    reverseOutput(specialChar, input, size);

    return 0;

}

4. Write a recursive function which will raise a number (double) to a non-negative integer

power n. The function receives the double value and integer as arguments.

**PROGRAM:**

#include <iostream>

using namespace std;

double power(double base, int exponent)

{

    if (exponent == 0)

    {

        return 1.0;

    }

    else

    {

        return base \* power(base, exponent - 1);

    }

}

int main()

{

    double base;

    int exponent;

    cout << "Enter base(double): ";

    cin >> base;

    cout << "Enter Exponent: ";

    cin >> exponent;

    if (exponent < 0)

    {

        cout << "The exponent should be greater than zero.";

    }

    else

    {

        cout << base << " raised to the power " << exponent << " is : " << power(base, exponent);

    }

    return 0;

}

5. Write a recursive method that for a positive integer n prints odd numbers

a. between 1 and n

b. between n and 1

**PROGRAM:**

#include <iostream>

using namespace std;

void printOddNumbersUpToN(int n) {

    if (n >= 1) {

        printOddNumbersUpToN(n - 1);

        if (n % 2 != 0) {

            cout << n << " ";

        }

    }

}

void printOddNumbersDownTo1(int n) {

    if (n >= 1) {

        if (n % 2 != 0) {

            cout << n << " ";

        }

        printOddNumbersDownTo1(n - 1);

    }

}

int main() {

    int n;

    do {

        cout << "Enter a positive integer (n): ";

        cin >> n;

    } while (n <= 0);

    cout << "a. Odd numbers between 1 and " << n << ": ";

    printOddNumbersUpToN(n);

    cout << "\nb. Odd numbers between " << n << " and 1: ";

    printOddNumbersDownTo1(n);

    cout << endl;

    return 0;

}

**LAB 8**

1. Write a program to implement a recursive version of merge-sort. Run it for some sample

data.

**PROGRAM:**

#include <iostream>

using namespace std;

void Merge(int arr[], int s, int e)

{

    int mid = (s + e) / 2;

    int len1 = mid - s + 1;

    int len2 = e - mid;

    int \*first = new int[len1];

    int \*second = new int[len2];

    int mainarrayindex = s;

    for (int i = 0; i < len1; i++)

    {

        first[i] = arr[mainarrayindex++];

    }

    mainarrayindex = mid + 1;

    for (int i = 0; i < len2; i++)

    {

        second[i] = arr[mainarrayindex++];

    }

    int index1 = 0;

    int index2 = 0;

    mainarrayindex = s;

    while (index1 < len1 && index2 < len2)

    {

        if (first[index1] < second[index2])

        {

            arr[mainarrayindex++] = first[index1++];

        }

        else

        {

            arr[mainarrayindex++] = second[index2++];

        }

    }

    while (index1 < len1)

    {

        arr[mainarrayindex++] = first[index1++];

    }

    while (index2 < len2)

    {

        arr[mainarrayindex++] = second[index2++];

    }

    delete[] first;

    delete[] second;

}

void Mergesort(int arr[], int s, int e)

{

    if (s >= e)

    {

        return;

    }

    int mid = (s + e) / 2;

    Mergesort(arr, s, mid);

    Mergesort(arr, mid + 1, e);

    Merge(arr, s, e);

}

int main()

{

    int arr[10] = {10, 9, 8, 9, 6, 5, 9, 3, 6, 1};

    int n = 10;

    Mergesort(arr, 0, n - 1);

    for (int i = 0; i < n; i++)

    {

        cout << arr[i] << " ";

    }

    cout << endl;

}

2. Write a program to implement a recursive version of quicksort. Run it for some sample

data.

**PROGRAM:**

#include <iostream>

using namespace std;

int partition(int arr[], int s, int e)

{

  int pivot = arr[s];

  int cnt = 0;

  for (int i = s + 1; i <= e; i++)

  {

    if (arr[i] <= pivot)

    {

      cnt++;

    }

  }

  int pivotIndex = s + cnt;

  swap(arr[pivotIndex], arr[s]);

  int i = s, j = e;

  while (i < pivotIndex && j > pivotIndex)

  {

    while (arr[i] < pivot)

    {

      i++;

    }

    while (arr[j] > pivot)

    {

      j--;

    }

    if (i < pivotIndex && j > pivotIndex)

    {

      swap(arr[i++], arr[j++]);

    }

  }

  return pivotIndex;

}

void Quicksort(int arr[], int s, int e)

{

  if (s >= e)

  {

    return;

  }

  int p = partition(arr, s, e);

  Quicksort(arr, s, p - 1);

  Quicksort(arr, p + 1, e);

}

int main()

{

  int arr[10] = {10, 9, 8, 6, 9, 3, 4, 5, 8, 1};

  int n = 10;

  Quicksort(arr, 0, n - 1);

  for (int i = 0; i < n; i++)

  {

    cout << arr[i] << " ";

  }

  cout << endl;

  return 0;

}

**LAB 9**

1. Implement pre-order traversal using the above code.

2. Implement post-order traversal using the above code.

3. Modify the above program to create and traverse the binary tree based on user input.

**PROGRAM:**

#include <iostream>

#include <queue>

using namespace std;

// A binary tree node has data, a pointer to the left child, and a pointer to the right child

struct Node

{

    int data;

    struct Node \*left, \*right;

};

// Utility function to create a new tree node

Node \*newNode(int data)

{

    Node \*temp = new Node;

    temp->data = data;

    temp->left = temp->right = NULL;

    return temp;

}

// Given a binary tree, print its nodes in inorder

void printInorder(Node \*node)

{

    if (node == NULL)

        return;

    // first recur on the left child

    printInorder(node->left);

    // then print the data of the node

    cout << node->data << " ";

    // now recur on the right child

    printInorder(node->right);

}

// Implement pre-order traversal

void printPreorder(Node \*node)

{

    if (node == NULL)

        return;

    // first print the data of the node

    cout << node->data << " ";

    // then recur on the left child

    printPreorder(node->left);

    // now recur on the right child

    printPreorder(node->right);

}

// Implement post-order traversal

void printPostorder(Node \*node)

{

    if (node == NULL)

        return;

    // first recur on the left child

    printPostorder(node->left);

    // then recur on the right child

    printPostorder(node->right);

    // now print the data of the node

    cout << node->data << " ";

}

// Function to create a binary tree based on user input

Node \*createUserTree()

{

    char stopChar;

    int data;

    cout << "Enter the root value: ";

    cin >> data;

    Node \*root = newNode(data);

    // Use a queue to keep track of nodes to be processed

    queue<Node \*> nodeQueue;

    nodeQueue.push(root);

    while (!nodeQueue.empty())

    {

        Node \*currentNode = nodeQueue.front();

        nodeQueue.pop();

        // Left child

        cout << "Enter the left child value for node " << currentNode->data << " ('q' to stop): ";

        cin >> stopChar;

        if (stopChar == 'q')

            break;

        cin >> data;

        currentNode->left = newNode(data);

        nodeQueue.push(currentNode->left);

        // Right child

        cout << "Enter the right child value for node " << currentNode->data << " ('q' to stop): ";

        cin >> stopChar;

        if (stopChar == 'q')

            break;

        cin >> data;

        currentNode->right = newNode(data);

        nodeQueue.push(currentNode->right);

    }

    return root;

}

int main()

{

    // Create a binary tree based on user input

    Node \*root = createUserTree();

    // Function calls for different traversals

    cout << "\nInorder traversal of the binary tree is \n";

    printInorder(root);

    cout << "\nPreorder traversal of the binary tree is \n";

    printPreorder(root);

    cout << "\nPostorder traversal of the binary tree is \n";

    printPostorder(root);

    return 0;

}

**LAB 10**

1. Analyze the given code to find the difference between “find” and “search” methods.

2. Merge “find” and “search” methods into a single method by necessary modifications in the

given code and execute it.

**ANALYSIS:**

* find method is responsible for locating a node with a specific value and providing its location (loc) and its parent (par).
* search method is used to search for a specific value in the binary search tree and print the depth at which the value is found.

Merging "find" and "search" methods:

We can merge these two methods into a single method called locate that returns the depth if the element is found and -1 if the element is not found. This way, we keep the flexibility to use the method for both locating and searching.

Here is the modified code:

#include <iostream>

#include <cstdlib>

using namespace std;

struct nod {

    int info;

    struct nod \*l;

    struct nod \*r;

} \*r;

class BST {

public:

    void search(int data);

    void insert(nod \*tree, nod \*newnode);

    void del(int data);

    void casea(nod \*par, nod \*loc);

    void caseb(nod \*par, nod \*loc);

    void casec(nod \*par, nod \*loc);

    void show(nod \*ptr, int level);

    BST() {

        r = NULL;

    }

private:

    void find(int data, nod \*\*par, nod \*\*loc);

    int locate(nod \*root, int data, int depth);

};

void BST::find(int data, nod \*\*par, nod \*\*loc) {

    nod \*ptr, \*ptrsave;

    if (r == NULL) {

        \*loc = NULL;

        \*par = NULL;

        return;

    }

    if (data == r->info) {

        \*loc = r;

        \*par = NULL;

        return;

    }

    if (data < r->info)

        ptr = r->l;

    else

        ptr = r->r;

    ptrsave = r;

    while (ptr != NULL) {

        if (data == ptr->info) {

            \*loc = ptr;

            \*par = ptrsave;

            return;

        }

        ptrsave = ptr;

        if (data < ptr->info)

            ptr = ptr->l;

        else

            ptr = ptr->r;

    }

    \*loc = NULL;

    \*par = ptrsave;

}

void BST::search(int data) {

    int depth = locate(r, data, 1);

    if (depth == -1) {

        cout << "Data not found in the tree" << endl;

    }

}

int BST::locate(nod \*root, int data, int depth) {

    if (root == NULL)

        return -1; // Element not found

    if (data == root->info) {

        cout << "\nData found at depth: " << depth << endl;

        return depth;

    } else if (data < root->info)

        return locate(root->l, data, depth + 1);

    else

        return locate(root->r, data, depth + 1);

}

void BST::insert(nod \*tree, nod \*newnode) {

    if (r == NULL) {

        r = new nod;

        r->info = newnode->info;

        r->l = NULL;

        r->r = NULL;

        cout << "Root Node is Added" << endl;

        return;

    }

    if (tree->info == newnode->info) {

        cout << "Element already in the tree" << endl;

        return;

    }

    if (tree->info > newnode->info) {

        if (tree->l != NULL)

            insert(tree->l, newnode);

        else {

            tree->l = newnode;

            tree->l->l = NULL;

            tree->l->r = NULL;

            cout << "Node Added To Left" << endl;

            return;

        }

    } else {

        if (tree->r != NULL)

            insert(tree->r, newnode);

        else {

            tree->r = newnode;

            tree->r->l = NULL;

            tree->r->r = NULL;

            cout << "Node Added To Right" << endl;

            return;

        }

    }

}

void BST::del(int data) {

    nod \*par = NULL, \*loc = r;

    if (r == NULL) {

        cout << "Tree empty" << endl;

        return;

    }

    int depth = locate(r, data, 1);

    if (depth == -1) {

        cout << "Item not present in tree" << endl;

        return;

    }

    find(data, &par, &loc);

    if (loc->l == NULL && loc->r == NULL) {

        casea(par, loc);

        cout << "Item deleted" << endl;

    }

    if (loc->l != NULL && loc->r == NULL) {

        caseb(par, loc);

        cout << "Item deleted" << endl;

    }

    if (loc->l == NULL && loc->r != NULL) {

        caseb(par, loc);

        cout << "Item deleted" << endl;

    }

    if (loc->l != NULL && loc->r != NULL) {

        casec(par, loc);

        cout << "Item deleted" << endl;

    }

    free(loc);

}

void BST::casea(nod \*par, nod \*loc) {

    if (par == NULL) {

        r = NULL;

    } else {

        if (loc == par->l)

            par->l = NULL;

        else

            par->r = NULL;

    }

}

void BST::caseb(nod \*par, nod \*loc) {

    nod \*child;

    if (loc->l != NULL)

        child = loc->l;

    else

        child = loc->r;

    if (par == NULL)

        r = child;

    else {

        if (loc == par->l)

            par->l = child;

        else

            par->r = child;

    }

}

void BST::casec(nod \*par, nod \*loc) {

    nod \*ptr, \*ptrsave, \*suc, \*parsuc;

    ptrsave = loc;

    ptr = loc->r;

    while (ptr->l != NULL) {

        ptrsave = ptr;

        ptr = ptr->l;

    }

    suc = ptr;

    parsuc = ptrsave;

    if (suc->l == NULL && suc->r == NULL)

        casea(parsuc, suc);

    else

        caseb(parsuc, suc);

    if (par == NULL)

        r = suc;

    else {

        if (loc == par->l)

            par->l = suc;

        else

            par->r = suc;

    }

    suc->l = loc->l;

    suc->r = loc->r;

}

void BST::show(nod \*ptr, int level) {

    int i;

    if (ptr != NULL) {

        show(ptr->r, level + 1);

        cout << endl;

        if (ptr == r)

            cout << "Root->: ";

        else {

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

                cout << " ";

        }

        cout << ptr->info;

        show(ptr->l, level + 1);

    }

}

int main() {

    int c, item;

    BST bst;

    nod \*t;

    while (1) {

        cout << "1.Insert Element " << endl;

        cout << "2.Delete Element " << endl;

        cout << "3.Search/Locate Element" << endl;

        cout << "4.Display the tree" << endl;

        cout << "5.Quit" << endl;

        cout << "Enter your choice : ";

        cin >> c;

        switch (c) {

            case 1:

                t = new nod;

                cout << "Enter the number to be inserted: ";

                cin >> t->info;

                bst.insert(r, t);

                break;

            case 2:

                if (r == NULL) {

                    cout << "Tree is empty, nothing to delete" << endl;

                    continue;

                }

                cout << "Enter the number to be deleted: ";

                cin >> item;

                bst.del(item);

                break;

            case 3:

                cout << "Search/Locate:" << endl;

                cin >> item;

                bst.search(item);

                break;

            case 4:

                cout << "Display BST:" << endl;

                bst.show(r, 1);

                cout << endl;

                break;

            case 5:

                exit(1);

            default:

                cout << "Wrong choice!" << endl;

        }

    }

    return 0;

}

**LAB 11**

1. The above code is for Max Heap. Implement a Min Heap and then insert and delete elements.

2. Implement peek and extract operations for both max and min values.

**PROGRAM**

#include <iostream>

#include <vector>

using namespace std;

void swap(int \*a, int \*b)

{

    int temp = \*b;

    \*b = \*a;

    \*a = temp;

}

void heapifyMax(vector<int> &hT, int i, int size)

{

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

    if (l < size && hT[l] > hT[largest])

        largest = l;

    if (r < size && hT[r] > hT[largest])

        largest = r;

    if (largest != i)

    {

        swap(&hT[i], &hT[largest]);

        heapifyMax(hT, largest, size);

    }

}

void insertMax(vector<int> &hT, int newNum)

{

    int size = hT.size();

    if (size == 0)

        hT.push\_back(newNum);

    else

    {

        hT.push\_back(newNum);

        for (int i = size / 2 - 1; i >= 0; i--)

            heapifyMax(hT, i, size);

    }

}

void deleteNodeMax(vector<int> &hT, int num)

{

    int size = hT.size();

    int i;

    for (i = 0; i < size; i++)

    {

        if (num == hT[i])

            break;

    }

    swap(&hT[i], &hT[size - 1]);

    hT.pop\_back();

    for (int i = size / 2 - 1; i >= 0; i--)

        heapifyMax(hT, i, size - 1);

}

int extractMax(vector<int> &hT)

{

    if (hT.size() == 0)

    {

        cout << "Heap is empty\n";

        return -1;

    }

    int maxVal = hT[0];

    deleteNodeMax(hT, hT[0]);

    return maxVal;

}

int peekMax(const vector<int> &hT)

{

    if (hT.size() == 0)

    {

        cout << "Heap is empty\n";

        return -1;

    }

    return hT[0];

}

void heapifyMin(vector<int> &hT, int i, int size)

{

    int smallest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

    if (l < size && hT[l] < hT[smallest])

        smallest = l;

    if (r < size && hT[r] < hT[smallest])

        smallest = r;

    if (smallest != i)

    {

        swap(&hT[i], &hT[smallest]);

        heapifyMin(hT, smallest, size);

    }

}

void insertMin(vector<int> &hT, int newNum) {

    int size = hT.size();

    if (size == 0)

        hT.push\_back(newNum);

    else {

        hT.push\_back(newNum);

        int i = size;

        while (i > 0 && hT[(i - 1) / 2] > hT[i]) {

            swap(&hT[i], &hT[(i - 1) / 2]);

            i = (i - 1) / 2;

        }

    }

}

void deleteNodeMin(vector<int> &hT, int num)

{

    int size = hT.size();

    int i;

    for (i = 0; i < size; i++)

    {

        if (num == hT[i])

            break;

    }

    swap(&hT[i], &hT[size - 1]);

    hT.pop\_back();

    for (int i = size / 2 - 1; i >= 0; i--)

        heapifyMin(hT, i, size - 1);

}

int extractMin(vector<int> &hT)

{

    if (hT.size() == 0)

    {

        cout << "Heap is empty\n";

        return -1;

    }

    int minVal = hT[0];

    deleteNodeMin(hT, hT[0]);

    return minVal;

}

int peekMin(const vector<int> &hT)

{

    if (hT.size() == 0)

    {

        cout << "Heap is empty\n";

        return -1;

    }

    return hT[0];

}

void printArray(const vector<int> &hT)

{

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

        cout << hT[i] << " ";

    cout << "\n";

}

int main()

{

    vector<int> maxHeap;

    insertMax(maxHeap, 3);

    insertMax(maxHeap, 4);

    insertMax(maxHeap, 9);

    insertMax(maxHeap, 5);

    insertMax(maxHeap, 2);

    cout << "Max-Heap array: ";

    printArray(maxHeap);

    deleteNodeMax(maxHeap, 4);

    cout << "After deleting an element in Max-Heap: ";

    printArray(maxHeap);

    cout << "Max-Heap peek: " << peekMax(maxHeap) << endl;

    cout << "Extract Max value: " << extractMax(maxHeap) << endl;

    cout << "Max-Heap array after extraction: ";

    printArray(maxHeap);

    vector<int> minHeap;

    insertMin(minHeap, 3);

    insertMin(minHeap, 4);

    insertMin(minHeap, 9);

    insertMin(minHeap, 5);

    insertMin(minHeap, 2);

    cout << "\nMin-Heap array: ";

    printArray(minHeap);

    deleteNodeMin(minHeap, 4);

    cout << "After deleting an element in Min-Heap: ";

    printArray(minHeap);

    cout << "Min-Heap peek: " << peekMin(minHeap) << endl;

    cout << "Extract Min value: " << extractMin(minHeap) << endl;

    cout << "Min-Heap array after extraction: ";

    printArray(minHeap);

    return 0;

}

**LAB 12**

1.Implement a hashing algorithm using Linear Probing.

2. Implement a hashing algorithm using Quadratic Probing.

3. Implement a hashing algorithm using Double Hashing.

**PROGRAM**

#include <iostream>

#include <list>

#include <algorithm>

using namespace std;

class HashMapTable

{

public:

    int table\_size;

    list<int> \*table;

    HashMapTable(int key)

    {

        this->table\_size = key;

        table = new list<int>[table\_size];

    }

    int hashFunction(int key)

    {

        return key % table\_size;

    }

    void insertElement(int key)

    {

        int index = hashFunction(key);

        table[index].push\_back(key);

    }

    void deleteElement(int key)

    {

        int index = hashFunction(key);

        list<int>::iterator i = find(table[index].begin(), table[index].end(), key);

        if (i != table[index].end())

        {

            table[index].erase(i);

        }

    }

    void displayHashTable()

    {

        for (int i = 0; i < table\_size; i++)

        {

            cout << i;

            for (auto j : table[i])

            {

                cout << " ==> " << j;

            }

            cout << endl;

        }

    }

};

class LinearProbingHashTable : public HashMapTable

{

public:

    LinearProbingHashTable(int key) : HashMapTable(key) {}

    void insertElementLinearProbing(int key)

    {

        int index = hashFunction(key);

        while (!table[index].empty())

        {

            index = (index + 1) % table\_size;

        }

        table[index].push\_back(key);

    }

};

class QuadraticProbingHashTable : public HashMapTable

{

public:

    QuadraticProbingHashTable(int key) : HashMapTable(key) {}

    void insertElementQuadraticProbing(int key)

    {

        int index = hashFunction(key);

        int i = 1;

        while (!table[index].empty())

        {

            index = (index + i \* i) % table\_size;

            i++;

        }

        table[index].push\_back(key);

    }

};

class DoubleHashingHashTable : public HashMapTable

{

public:

    DoubleHashingHashTable(int key) : HashMapTable(key) {}

    int hashFunction2(int key)

    {

        return 1 + (key % (table\_size - 2)); // Ensure step size is relatively prime to table\_size

    }

    void insertElementDoubleHashing(int key)

    {

        int index = hashFunction(key);

        int step = hashFunction2(key);

        // Iterate until an empty slot is found or a full cycle is completed

        while (!table[index].empty() && table[index].front() != key)

        { // Check for existing key

            index = (index + step) % table\_size;

            if (index == hashFunction(key))

            { // Full cycle check

                cout << "Hash table is full\n";

                return;

            }

        }

        table[index].push\_back(key);

    }

};

int main()

{

    int arr[] = {1,2,3,4,5,6};

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

    // Linear Probing

    LinearProbingHashTable linearTable(6);

    for (int i = 0; i < n; i++)

        linearTable.insertElementLinearProbing(arr[i]);

    cout << "Linear Probing Hash Table:" << endl;

    linearTable.displayHashTable();

    cout << endl;

    // Quadratic Probing

    QuadraticProbingHashTable quadraticTable(6);

    for (int i = 0; i < n; i++)

        quadraticTable.insertElementQuadraticProbing(arr[i]);

    cout << "Quadratic Probing Hash Table:" << endl;

    quadraticTable.displayHashTable();

    cout << endl;

    // Double Hashing

    DoubleHashingHashTable doubleHashingTable(6);

    for (int i = 0; i < n; i++)

        doubleHashingTable.insertElementDoubleHashing(arr[i]);

    cout << "Double Hashing Hash Table:" << endl;

    doubleHashingTable.displayHashTable();

    return 0;

}