**DATA STRUCTURE**

**Name :** Faizan Tariq

**Roll No:** 22F-3858

**Section:** BS(CS) 3D

Assignment #1

**TASK 1:**

1. The worst case time complexity as function of n is ” times.
2. The worst case time complexity as function of n is ” times
3. The worst case time complexity as function of n is ” times.
4. Outer loop runs n/2 times and inner loop twice its value for each time so its time complexity will be “n\* log(n)\* times.
5. It runs half of its total value of N so its times complexity is log(n)

**TASK 2:**

**CODE :**

#include<iostream>

using namespace std;

class Array {

private:

int \* arr;

int size=0;

int numItems;

public:

Array() { // default constructor

numItems = -1;

cout << "\nEnter Size of an array :";

while (size <= 0)

{

cin >> size;

if (size > 0) {

arr = new int[size];

cout << "\nInitialize Your array :";

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

{

cin >> arr[i];

numItems++;

}

cout << endl;

}

else {

cout << "\nSize must be greater then 0.\n";

}

}

}

Array(int Size) { // parameterized constructor

numItems = -1;

while (Size <= 0)

{

cin >> Size;

if (size > 0) {

arr = new int[Size];

size = Size;

cout << "\nInitialize Your array :";

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

{

cin >> arr[i];

numItems++;

}

cout << endl;

}

else {

cout << "\nSize must be greater then 0.\n";

cin >> size;

}

}

}

Array(const Array& other) // copy constructor

{

size = other.size;

arr = new int[size];

numItems = other.numItems;

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

{

arr[i] = other.arr[i];

}

}

bool isEmpty() {

return (numItems == -1);

}

bool isFull() {

return (numItems == size - 1);

}

int length() {

return size;

}

void printList() {

if (isEmpty()) {

cout << "\nList is empty, please enter some data\n";

return;

}

else {

cout << "\nElements is array :";

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

{

cout << \*(arr+i)<<" ";

}

cout << endl;

}

}

int searchElementLinear(int X) { //Linear Search

bool found = false;

if (isEmpty()) {

cout << "\nList is empty, please enter some data\n";

return -1;

}

else {

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

{

if (arr[i] == X) {

found = true;

return i; // returns element index location of dynamic array

}

}

}

if (!found) {

cout << "\nElement cannot be found please enter other value.\n";

return -1;

}

}

int searchElementBinary(int X) { // binary Search

// first sort array for binary search

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

{

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

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

cout << "After Sorting :";

printList();

// now apply binary search

int start = 0, end = size - 1;

int middle = 0;

while (start <= end) {

middle = (start + end) / 2;

if (arr[middle] == X) {

return middle;

}

else if(arr[middle] > X) {

end = middle - 1;

}

else if (arr[middle] < X) {

start = middle + 1;

}

}

return -1;

}

void insertElementAt(int value, int pos) {

bool found = false;

if (isEmpty()) {

cout << "\nList is empty, please enter some data\n";

return;

}

else {

if (pos < 0 || pos >= size) {

cout << "\nPlease enter valid position";

cout << endl;

return;

}

int\* tempArr = new int[size + 1];

for (int i = 0; i < pos; i++) // copies before pos

{

tempArr[i] = arr[i];

}

tempArr[pos] = value;// stores value at specific pos given by user

for (int i = pos+1; i <= size; i++)

{

tempArr[i] = arr[i-1];

}

delete[] arr;

size++;

numItems++;

arr = tempArr;

}

}

bool deleteElement(int X) {

bool found = false;

if (isEmpty()) {

cout << "\nList is empty, please enter some data\n";

return false;

}

int findIndex = searchElementLinear(X); // search element and return index of that element

if (findIndex >= 0 || findIndex < size) { // deletes element at find index

for (int i = findIndex; i < size-1; i++)

{

arr[i] = arr[i + 1];

}

size--;

numItems--;

}

}

void reverseList() {

int start = 0;

int end = size - 1;

int tempElement; // for swaping

for (int i = 0; i < size/2; i++)

{

tempElement = arr[start]; // swap start with end

arr[start] = arr[end];

arr[end] = tempElement;

start++;

end--;

}

}

void copyList(const Array& other) {

numItems = other.numItems;

size = other.size;

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

{

arr[i] = other.arr[i];

}

}

void emptyList() {

numItems = -1;

size = 0;

}

~Array() {

emptyList();

delete[] arr;

}

};

void menu(int & choice) {

cout << "\n1. Print List"<<endl

<<"2. Linear Search"

<< endl << "3. Binary Search"

<< endl << "4. Insert At index"

<< endl << "5. Delete"

<< endl <<"6. Check Full"

<< endl << "7. Check Empty"

<< endl << "8. Length"

<< endl << "9. Reverse"

<< endl << "10. Empty List"

<< endl <<"11. Copy From Another List"

<< endl << "12. Exit"

<< endl << "\nEnter choice: ";

cin >> choice;

}

int main() {

int choice = 0;

int element = 0; int position = 0;

Array temp;

while (1) {

menu(choice);

switch (choice) {

case 1:

temp.printList();

break;

case 2:

cout<< "Enter element to search: ";

cin >> element;

position = temp.searchElementLinear(element);

if (position == -1) {

cout << "Element not found please enter another element.\n";

}

else cout << "Element found at index " << position << "."<<endl;

break;

case 3:

cout << "Enter element to search (binary): ";

cin >> element;

position = temp.searchElementBinary(element);

if (position == -1) cout << "Element not found please enter another element.\n";

else cout << "Element found at index " << position << ".\n";

break;

case 4:

cout << "\nEnter element: ";

cin >> element;

cout << "\nEnter position: ";

cin >> position;

temp.insertElementAt(element, position);

break;

case 5:

cout << "Enter element to delete: ";

cin >> element;

temp.deleteElement(element);

break;

case 6:

if (temp.isFull()) {

cout << "List is full.\n";

}

else { cout << "List contains some elements full.\n"; }

break;

case 7:

if (temp.isEmpty()) {

cout << "List is empty.\n";

}

else cout << "List constains some elements\n";

break;

case 8:

cout << "Length of list: " << temp.length() << ".\n";

break;

case 9:

temp.reverseList();

cout << "\n List reversed.\n";

break;

case 10:

temp.emptyList();

cout << "\n List is now empty.\n";

break;

case 11:

{

Array otherArr;

cout << "Data in new Arr :";

otherArr.printList();

cout << "\n Copying data of new Array in old array.\n";

temp.copyList(otherArr);

}

break;

case 12:

return 0;

default:

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

}

}

return 0;

}

**Worst Case Time Complexity:**

**Insertation :** Insertion have worst case time complexity because if we want to insert at 1st index I will need to shift other elements to index+1, so its time complexity is (N)

**Deletion:** Deletion have worst case time complexity because if we want to delete at 1st index I will need to shift other elements to index-1 , so its time complexity is (N)

**OUTPUT:  
A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated** **A screenshot of a computer program

Description automatically generated** A screenshot of a computer program

Description automatically generated

**A screenshot of a computer program

Description automatically generated**

**TASK 3:**

**CODE :**

#include<iostream>

#include<fstream>

#include<string>

using namespace std;

struct Employee\_Node {

string name;

int hours;

int wage;

Employee\_Node\* next;

Employee\_Node() {

name = "";

hours = 0;

wage = 0;

next = NULL;

}

Employee\_Node(string Name, int Hours) {

name = Name;

hours = Hours;

wage = 0;

next = NULL;

}

void calculateWage() {

wage = hours \* 50;

}

};

class Company {

Employee\_Node\* head;

public:

Company() {

head = NULL;

}

void createEmployeeNodes() { // this function will get data from employees.txt and create nodes of every employee

ifstream fin("Employees.txt");

string tempname;

int temphour;

if (fin.is\_open()) {

while (fin >> tempname >> temphour) {

Employee\_Node\* newNode = new Employee\_Node;

newNode->hours = temphour;

newNode->name = tempname;

if (head == NULL) { // its mean there is no any existing node in runtime

head = newNode;

}

else {

Employee\_Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

}

else {

cout << "\nFile cannot open.\n";

}

}

void safeEmployeeWages() {

ofstream fout("Wages.txt");

if (fout.is\_open()) {

Employee\_Node\* current = head;

do

{

current->calculateWage(); // calculates wage of every employee

fout << current->name << " " << current->wage<<endl;

current = current->next;

} while (current != NULL);

}

else {

cout << "\nWages.txt Not found."<<endl;

}

}

void displayNodes() {

Employee\_Node\* temp = head;

do{

cout <<"NAME : " << temp->name << " Wage: " << temp->wage<<endl;

temp = temp->next;

} while (temp != NULL);

}

};

int main() {

Company temp;

temp.createEmployeeNodes();

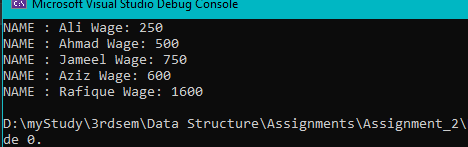
temp.safeEmployeeWages();

temp.displayNodes();

return 0;

}

OUTPUT :

****

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**TASK 4:**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node() {

data = 0;

next = NULL;

}

Node(int Data) {

data = Data;

next = NULL;

}

};

class circularLinkedList {

private:

Node\* head;

public:

circularLinkedList() {

head = NULL;

}

bool isEmpty() {

return head == NULL;

}

void insertData(int data) {

Node\* newNode = new Node(data);

if (isEmpty()) {

head = newNode;

head->next = head;

return;

}

Node\* cur = head;

while (cur->next != head) { // traverse

cur = cur->next;

}

cur->next = newNode; // insert at last

newNode->next = head;

}

int getRemainingNode(int m) {

if (isEmpty()) {

return -1;

}

Node\* cur = head; // for traverse

Node\* initial = new Node; // node before cur node

while (cur->next != cur) {

int count = 0;

while (count != m) {

count++;

initial = cur;

cur = cur->next; // traverse till count != m

}

// now cur node is at m location

// delete m node

initial->next = cur->next;

delete cur;

cur = initial->next;

}

return cur->data; /// returns data of remaining node

}

~circularLinkedList() {

if (isEmpty()) {

return;

}

Node\* cur = head;

Node\* nextNode = new Node;

while (cur->next != head) {

nextNode = cur->next;

delete cur;

cur = nextNode;

}

delete nextNode;

}

};

int main() {

cout << "Enter Length of circle :";

int N;

cin >> N;

circularLinkedList List;

cout << "Enter Data in Circular Linked List :";

int data;

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

{

cin >> data;

if (data > 0) {

List.insertData(data);

}

else {

cout << "\nData cannot be negative.\n";

}

}

cout << "Count to choose Next :";

int M;

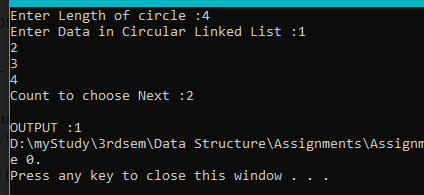
cin >> M;

cout << endl;

cout<<"OUTPUT :"<<List.getRemainingNode(M);

return 0;

}

**OUTPUT:  
**

**TASK 5:**

#include<iostream>

#include<fstream>

using namespace std;

struct PolynomialNode {

int coefficient;

int pwr;

PolynomialNode\* next;

PolynomialNode() {

coefficient = 0;

pwr = 0;

next = NULL;

}

PolynomialNode(int coeff, int PWR ) {

coefficient = coeff;

pwr = PWR;

next = NULL;

}

};

class PolyNomial {

private:

PolynomialNode \* head;

public:

PolyNomial() {

head = NULL;

}

bool isEmpty() {

return (head == NULL);

}

void addNode(int coeff, int pwr) {

PolynomialNode\* newNode = new PolynomialNode(coeff, pwr);

if (isEmpty()) {

head = newNode;

return;

}

PolynomialNode\* cur = head;

while (cur->next != NULL) {

cur = cur->next;

}

cur->next =newNode;

}

void importFromFile(string filename) {

ifstream fin(filename);

if (fin.is\_open()) {

int coeff, pwr;

char base, sign; // only for get text in format

while (fin >> coeff >> base >> sign >> pwr)

{

addNode(coeff, pwr);

}

}

else {

cout << "\n Cannot open file\n";

}

fin.close();

}

PolyNomial operator+(const PolyNomial& other) { // operator overloading

PolyNomial asnwer;

PolynomialNode\* node1 = head;

PolynomialNode\* node2 = other.head;

while (node1 && node2)// true until both are NULL

{

if (node1->pwr > node2->pwr) {

asnwer.addNode(node1->coefficient, node1->pwr);

node1 = node1->next;

}

else if (node1->pwr == node2->pwr) {

asnwer.addNode(node1->coefficient + node2->coefficient, node1->pwr);

// traverse

node1 = node1->next;

node2 = node2->next;

}

else {

asnwer.addNode(node2->coefficient, node2->pwr);

node2 = node2->next;

}

}

// now if only a or b is not null

while (node1) {

asnwer.addNode(node1->coefficient, node1->pwr);

node1 = node1->next;

}

while (node2) {

asnwer.addNode(node2->coefficient, node2->pwr);

node2 = node2->next;

}

return asnwer;

}

void displayPolynomial() {

if (isEmpty()) {

cout << "\nNode is empty.\n";

return;

}

PolynomialNode\* cur = head;

while (cur != NULL) {

cout << cur->coefficient << "x^" << cur->pwr;

if (cur->next && cur->next->coefficient >= 0) {

cout << "+";

}

cur = cur->next;

}

}

};

int main() {

PolyNomial result, temp1, temp2;

temp1.importFromFile("polynomial1.txt");

cout << "\n1st polynomial imported from file :\n";

temp1.displayPolynomial();

temp2.importFromFile("polynomial2.txt");

cout << "\n2nd polynomial imported from file :\n";

temp2.displayPolynomial();

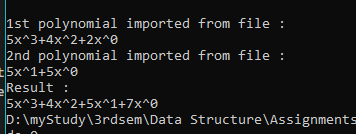
result = temp1 + temp2;

cout << "\nResult :\n";

result.displayPolynomial();

return 0;

}

**OUTPUT :  
**

**A screenshot of a computer

Description automatically generated**

**A black background with white text and purple text

Description automatically generated**

**TASK 6:**

**CODE:**

#include<iostream>

using namespace std;

struct Node {

int id;

Node\* next;

bool read , unread, trash;

Node() {

id = 0;

read = false;

unread = true; // by default unread is true

trash = false;

next = NULL;

}

Node(int Id) {

id = Id;

read = false;

unread = true; // by default unread is true

trash = false;

next = NULL;

}

};

class Linkedlist {

private :

Node\* head;

int id;

public:

Linkedlist() {

id = 0;

head = NULL;

}

bool isEmpty() {

return (head == NULL);

}

void addMessage() {

id++; // assign id to every message greater than 1 to previous

Node\* newNode = new Node(id);

if (isEmpty()) {

head = newNode;

}

else {

Node\* cur = head;

// traverse till end

while (cur->next != NULL) {

cur = cur->next;

}

cur->next = newNode;

}

}

void displayUnreadSection() {

cout << "\nInitial UNREAD Section : ";

if (isEmpty()) {

cout << "\nNo node present.\n";

}

Node\* cur = head;

bool found = false;

while (cur != NULL) {

if (cur->unread) { // if cur->unread == true

cout << cur->id << " ";

found = true;

}

cur = cur->next;

}

if (!found) {

cout << "\"Empty\"\n";

}

}

void displayreadSection() {

cout << "\nREAD Section : ";

if (isEmpty()) {

cout << "\nNo node present.\n";

}

Node\* cur = head;

bool found = false;

while (cur != NULL) {

if (cur->read) { // if cur->read != false

cout << cur->id << " ";

found = true;

}

cur = cur->next;

}

if (!found) {

cout << "\"Empty\"\n";

}

}

void displaytrashSection() {

cout << "\nTrash Section : ";

if (isEmpty()) {

cout << "\nNo node present.\n";

}

Node\* cur = head;

bool found = false;

while (cur != NULL) {

if (cur->trash) { // if cur->trash != false

cout << cur->id<<" ";

found = true;

}

cur = cur->next;

}

if (!found) {

cout << "\"Empty\"\n";

}

}

void messageActions(int action\_Id, int message\_Id) {

// for traverse to 1st check if message exists in list having that message\_Id

Node\* cur = head;

bool found = false;

while (cur != NULL && message\_Id != cur->id) { /// loop until cur have same id as message\_ID

cur = cur->next;

}

if (cur->id == message\_Id) {

found = true;

}

else {

found = false;

}

if (!found) {

cout << "\n Message cannot found having that id :" << message\_Id << ", Please try again with different id.\n";

return;

}

if (action\_Id == 1) { // to convert unread to read

if (cur->unread == true) {

cur->read = true;

cur->unread = false;

cur->trash = false;

}

else if (cur->unread == false) {

cout << "\n Message is already in read OR trash section.\n";

}

}

else if (action\_Id == 2) { // to convert read to trash

if (cur->read == true) {

cur->unread = false;

cur->read = false;

cur->trash = true;

}

else if (cur->read == false) {

cout << "\n Message is already in unread OR trash section.\n";

}

}

else if (action\_Id == 3) { // to convert unread to trash

if (cur->unread == true) {

cur->read = false;

cur->unread = false;

cur->trash = true;

}

else if (cur->unread == false) {

cout << "\n Message is already in read OR trash section.\n";

}

}

else if (action\_Id == 4) { // to convert trash to read

if (cur->trash == true) {

cur->read = true;

cur->unread = false;

cur->trash = false;

}

else if (cur->trash == false) {

cout << "\n Message is already in read OR unread section.\n";

}

}

else {

cout << "\n Invalid Action id!\n";

}

}

void displaySections() {

displayUnreadSection();

displayreadSection();

displaytrashSection();

}

};

void actionMenu() {

cout << "\n1. Move the message with ID X from UNREAD to READ.";

cout << "\n2. Move the message with ID X from READ to TRASH.";

cout << "\n3. Move the message with ID X from UNREAD to TRASH.";

cout << "\n4. Move the message with ID X from TRASH to READ.";

cout << "\n Choose the action to perfrom :";

}

int main() {

Linkedlist emailSystem;

int n;

cout << "How many messsages you want to add in you inbox :";

cin >> n;

while (n > 0) {

emailSystem.addMessage(); // id will automatically be generated and incremented by 1

n--;

}

int actionId = 0 , messageId = 0;

while (actionId >= 0 && messageId >= 0) {

cout << "\nEnter Your queries (-1 to exit) :\n";

emailSystem.displaySections();

cout << endl;

actionMenu();

cin >> actionId;

cout << "\nEnter Id of message on which you want to perform an action :";

cin >> messageId;

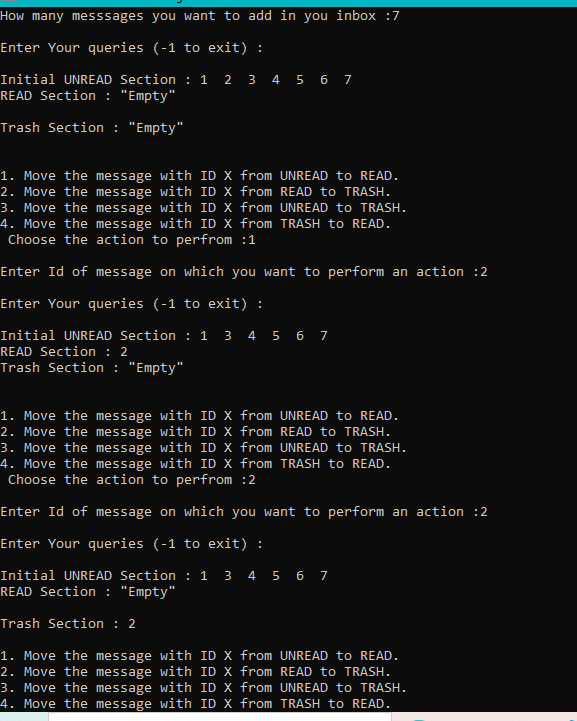
emailSystem.messageActions(actionId, messageId);

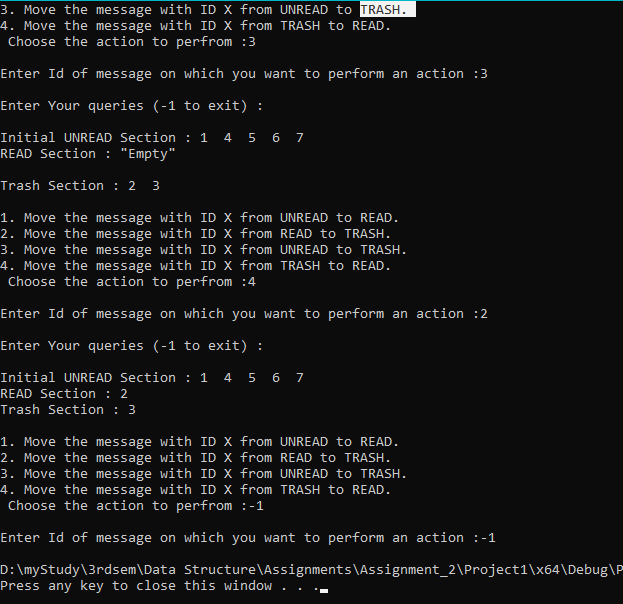
}

return 0;

}

**OUTPUT:**

****

****