**Name: Umer Rashid**

**Roll: 22F-3396**

**Task 1:**

1. **O n^2**
2. **O N^5**
3. **On^2**
4. **O (N log 2 N)**
5. **Log 2 N**

**Task 2:**

#include <iostream>

using namespace std;

class List

{

private:

int\* arr;

int capacity;

int size;

public:

// default constructor

List()

{

arr = new int[3];

for (int i = 0; i < 3; i++)

{

arr[i] = i + 1;

}

size = 0;

}

// Constructors

List(int capacity )

{

this->capacity = capacity;

arr = new int[capacity];

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

{

arr[i] = i + 1;

}

size = 0;

}

//copy constructor

List(const List& listObj)

{

capacity = listObj.capacity;

size = listObj.size;

arr = new int[capacity];

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

{

arr[i] = listObj.arr[i];

}

}

//destructor

~List() {

delete[] arr;

}

// display function

void printList(List obj)

{

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

{

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

}

cout << endl;

}

//search funciton

int searchElement(int X)

{

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

{

if (arr[i] == X) {

return i;

}

}

cout << "Element is not in the List" << endl;

}

// insert function

void insertElementAt(int X, int pos)

{

if (isFull())

{

int newCapacity = capacity \* 2;

int\* newArr = new int[newCapacity];

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

{

newArr[i] = arr[i];

}

delete[] arr;

arr = newArr;

capacity = newCapacity;

}

if (pos < 0 || pos > size)

{

cout << "Error: Invalid position for insertion." << endl;

return;

}

for (int i = size; i > pos; --i)

{

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

}

arr[pos] = X;

++size;

}

// delete function

bool deleteElement(int X)

{

int pos = searchElement(X);

if (pos != -1)

{

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

{

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

}

--size;

return true;

}

return false;

}

// function to check if list is full

bool isFull()

{

if (size == capacity)

{

return true;

}

else

return false;

}

// function to check if list is empty

bool isEmpty() const

{

if (size == 0)

{

return true;

}

else

return false;

}

// function to display length of the list

int length()

{

return size;

}

// function to reverse the list

void reverseList()

{

int start = 0;

int end = size - 1;

while (start < end)

{

swap(arr[start], arr[end]);

++start;

--end;

}

}

// function to empty the list

void emptyList()

{

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

{

arr[i] = -1;

}

cout << "List Emptied" << endl;

}

// function to copy the list into anothe list

void copyList(const List& other)

{

if (this != &other)

{

delete[] arr;

capacity = other.capacity;

size = other.size;

arr = new int[capacity];

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

{

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

}

}

}

};

int main()

{

List myList(10);

int choice;

while (true)

{

cout << "\nMenu:\n";

cout << "1. Print List\n";

cout << "2. Search Element\n";

cout << "3. Insert Element\n";

cout << "4. Delete Element\n";

cout << "5. Reverse List\n";

cout << "6. Empty List\n";

cout << "7. Copy List\n";

cout << "8. Quit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

myList.printList(myList);

break;

case 2:

int searchValue;

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

cin >> searchValue;

cout << "Element found at position " << myList.searchElement(searchValue) << endl;

break;

case 3:

int val, pos;

cout << "Enter the value to insert: ";

cin >> val;

cout << "Enter the position to insert: ";

cin >> pos;

myList.insertElementAt(val, pos);

break;

case 4:

int deleteval;

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

cin >> deleteval;

if (myList.deleteElement(deleteval))

{

cout << "Element deleted from the list." << endl;

}

else

{

cout << "Element not found in the list." << endl;

}

break;

case 5:

myList.reverseList();

cout << "List reversed." << endl;

break;

case 6:

myList.emptyList();

cout << "List emptied." <<endl;

break;

case 7:

{

List copyList;

copyList.copyList(myList);

cout << "List copied to another list." << endl;

}

break;

case 8:

return 0;

default:

cout << "The choice entered is wrong" << endl;

break;

}

}

return 0;

}

**Task 3**

#include<iostream>

#include<fstream>

#include<string>

using namespace std;

class Employee

{

public:

string name;

int workingHours;

int wages;

public:

Employee()

{

workingHours = 0;

}

Employee(string name, int workingHours)

{

this->name = name;

this->workingHours = workingHours;

}

~Employee()

{

}

int calculateWage(int workingHours)

{

wages= 50 \* workingHours;

return wages;

}

};

struct Node

{

Employee data;

Node\* next;

Node(Employee value)

{

data = value;

next = nullptr;

}

};

class List {

private:

Node\* head;

public:

List() {

head = nullptr;

}

void insertAtBeginning(Employee value) {

Node\* newNode = new Node(value);

newNode->next = head;

head = newNode;

}

void insertAtEnd(Employee value) {

Node\* newNode = new Node(value);

//statement for starting condition

if (head == nullptr) {

head = newNode;

return;

}

Node\* current = head;

while (current->next != nullptr) {

current = current->next;

}

current->next = newNode;

}

void displayList() {

Node\* current = head;

cout << "Name" << "\t" << "Wage" << endl;

while (current != nullptr) {

cout << current->data.name << "\t" << current->data.calculateWage(current->data.workingHours) << "\n";

current = current->next;

}

}

};

int main()

{

List list;

ifstream fin("Employee.txt");

ofstream fout("Wages.txt");

if (!fin)

{

cout << "Failed to open the file." << endl;

return 1;

}

fout << "Name" << "\t" << "Wage" << endl;

while (!fin.eof())

{

Employee employee;

getline(fin, employee.name, ',');

fin >> employee.workingHours;

fin.ignore();

if (!fin.eof())

{

list.insertAtEnd(employee);

fout << employee.name << "\t" << employee.calculateWage(employee.workingHours) << "\n";

}

}

list.displayList();

fin.close();

fout.close();

return 0;

}

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

**Task 4:**

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* next;

Node(int value)

{

data = value;

next = nullptr;

}

};

class CircularLL

{

private:

int N;

int M;

public:

Node\* circularlist(int N)

{

this->N = N;

int j = 1;

Node\* head = new Node(j);

Node\* current = head;

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

{

current->next = new Node(i);

current = current->next;

}

current->next = head;

return head;

}

int Last(int N, int M)

{

this->N = N;

this->M = M;

Node\* head = circularlist(N);

Node\* current = head;

Node\* prev = current;

while (current->next != current)

{

int count = 1;

while (count < M)

{

prev = current;

current = current->next;

count++;

}

prev->next = current->next;

Node\* temp = current;

current = current->next;

delete temp;

}

return current->data;

}

};

int main()

{

int N, M;

CircularLL Object;

cout << "Enter the value ot N: ";

cin >> N;

cout << "Enter the value of M: ";

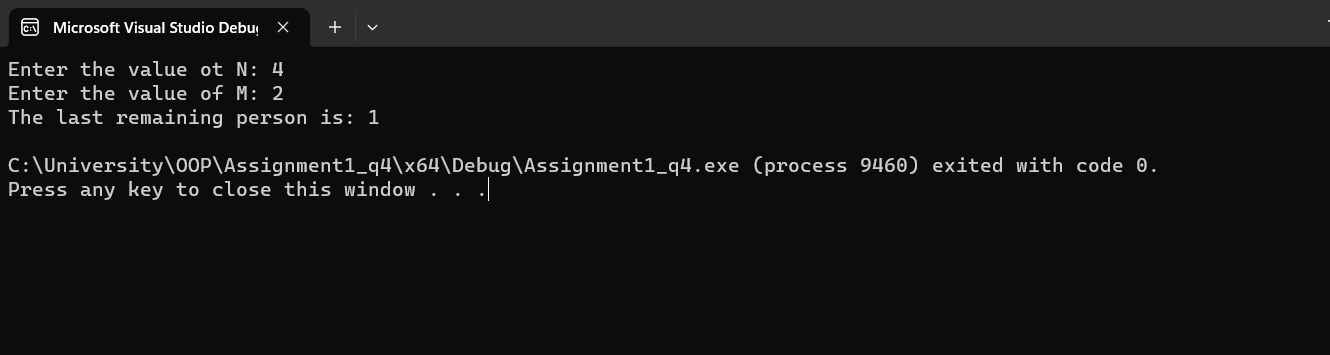
cin >> M;

int last = Object.Last(N, M);

cout << "The last remaining person is: " << last << endl;

return 0;

}



**Task 5:**

#include <iostream>

using namespace std;

struct Node

{

int coefficient;

int exponent;

Node\* next;

};

class List {

public:

List()

{

}

~List()

{

}

void insert(Node\*& head, int c, int e)

{

Node\* newNode = new Node;

newNode->coefficient = c;

newNode->exponent = e;

newNode->next = nullptr;

if (!head)

{

head = newNode;

}

else

{

Node\* current = head;

Node\* prev = nullptr;

while (current && current->exponent > e)

{

prev = current;

current = current->next;

}

if (current && current->exponent == e)

{

current->coefficient += c;

delete newNode;

}

else

{

if (prev)

{

prev->next = newNode;

}

else

{

head = newNode;

}

newNode->next = current;

}

}

}

Node\* addNumbers(Node\* num1, Node\* num2)

{

Node\* result = nullptr;

while (num1 || num2)

{

int c1 = num1->coefficient;

int e1 = num1->exponent;

int c2 = num2->coefficient;

int e2 = num2->exponent;

if (e1 > e2)

{

insert(result, c1, e1);

num1 = num1->next;

}

else if (e2 > e1)

{

insert(result, c2, e2);

num2 = num2->next;

}

else

{

insert(result, c1 + c2, e1);

num1 = num1->next;

num2 = num2->next;

}

}

return result;

}

void display(Node\* head)

{

while (head)

{

cout << head->coefficient;

if (head->exponent >= 0)

{

cout << "x^" << head->exponent;

}

if (head->next)

{

cout << " + ";

}

head = head->next;

}

cout << endl;

}

};

int main()

{

List list;

Node\* num1 = nullptr;

Node\* num2 = nullptr;

//5x^3 + 4x^2 + 2x^0

list.insert(num1, 5, 3);

list.insert(num1, 4, 2);

list.insert(num1, 2, 0);

//5x^1 + 5x^0

list.insert(num2, 5, 1);

list.insert(num2, 5, 0);

cout << "Number 1: ";

list.display(num1);

cout << "Number 2: ";

list.display(num2);

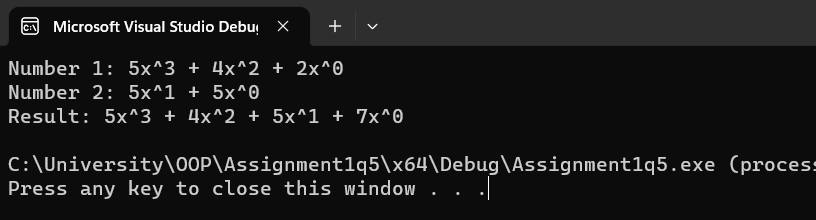
Node\* result = list.addNumbers(num1, num2);

cout << "Result: ";

list.display(result);

return 0;

}



**Task6:**

#include <iostream>

using namespace std;

struct Message

{

int id;

Message\* next;

Message(int id)

{

this->id = id;

next = nullptr;

}

};

class Gmail

{

private:

Message\* unreadHead;

Message\* readHead;

Message\* trashHead;

public:

Gmail()

{

unreadHead = nullptr;

readHead = nullptr;

trashHead = nullptr;

}

void move(int id, Message\*& prevlocation, Message\*& newlocation)

{

if (prevlocation->id == id)

{

Message\* movedMessage = prevlocation;

prevlocation = prevlocation->next;

movedMessage->next = newlocation;

newlocation = movedMessage;

}

else

{

Message\* prev = nullptr;

Message\* current = prevlocation;

while (current && current->id != id)

{

prev = current;

current = current->next;

}

if (current)

{

prev->next = current->next;

current->next = newlocation;

newlocation = current;

}

}

}

void addUnreadMessage(int id)

{

Message\* newMessage = new Message(id);

newMessage->next = unreadHead;

unreadHead = newMessage;

}

void printSection(Message\* section)

{

if (!section)

{

cout << "EMPTY";

}

else

{

while (section)

{

cout << section->id << " ";

section = section->next;

}

}

cout << endl;

}

void handleQuery(int queryType, int id)

{

switch (queryType)

{

case 1:

move(id, unreadHead, readHead);

break;

case 2:

move(id, readHead, trashHead);

break;

case 3:

move(id, unreadHead, trashHead);

break;

case 4:

move(id, trashHead, readHead);

break;

}

}

void printAllSections()

{

printSection(unreadHead);

printSection(readHead);

printSection(trashHead);

}

};

int main()

{

Gmail mailbox;

int N;

cout << "Enter the number of initial messages: ";

cin >> N;

for (int i = N; i >=1; i--)

{

mailbox.addUnreadMessage(i);

}

int Q;

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

cin >> Q;

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

{

cout << "Menu:\n";

cout << "1. Move a message from UNREAD to READ\n";

cout << "2. Move a message from READ to TRASH\n";

cout << "3. Move a message from UNREAD to TRASH\n";

cout << "4. Move a message from TRASH to READ\n";

cout << "Enter query type and message ID : ";

int type, id;

cin >> type >> id;

mailbox.handleQuery(type, id);

cout << "Updated sections:\n";

mailbox.printAllSections();

}

return 0;

}

****