Advanced algorithms

School of computing

By:ali mostafa Mahmoud
202101194

Table of Contents

Introduction

The report will cover different algorithms like quick sort, heap sort, bubble sort, selection sort, insertion sort, matrices, tree graphs, and normal graphs. The coursework will challenge my knowledge and my understanding of different algorithms by solving tasks. Sorting algorithms are a collection of instructions that are used to organize objects into a specific order using an array or list as input. Sorts can be done in ascending (A, 0-9) or descending (Z-A, 9-0) order, and are most frequently done in numerical or alphabetical (or lexicographical) order. Sorting algorithms are critical to computer science because they may frequently lessen the complexity of the code. These algorithms directly apply to data structure algorithms, divide and conquer strategies, database algorithms, and searching algorithms, among many other areas. A tree data structure is an edge-connected hierarchical data structure made up of nodes. There can be more than one child node for every node, but only one parent node. The root node is the highest in the tree and It is simple to perform traversal in linear time. A graph data structure is made up of nodes, also known as vertices and the edges that join them. Edges show the connections between the many entities that nodes, such as people, and locations. Graphs are used for Social networks, road networks, and computer networks. A matrix is a rectangular arrangement of characters, numbers, symbols, or points, arranged in specific row and column orders, with each element's placement determined by its row and column.

The tasks

Task 2

Code explanation

The goal of this task is to sort a double-linked list using a quick sort. QuickSort is a sorting method that uses the Divide and Conquer algorithm as its foundation. It selects an element to act as a pivot and divides the array around it by positioning the pivot correctly within the sorted array. The first step is to find the pointer of the last node. In the second step, if the pointer of the last node is available the code will recursively sort the linked list using pointers to the first and last nodes of a linked list. Like how partition functions for arrays, the partition function functions similarly with linked lists. In the last step the, code returns the pointer of the pivot element instead index of the pivot element. QuickSort() is only a wrapper function in the code that follows; the primary recursive method, _quickSort(), is comparable to quicksort() for array implementation. For the code implementation, I did the following

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Node
{
    public:
    int data;
    Node *next;
    Node *prev;
};
```

Importing the needed libraries and then creating a class node. then created three variables data(integer), next(node) and prev(node).

```
void swap ( int* a, int* b )
{ int t = *a; *a = *b; *b = t; }



Node *lastNode(Node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}
```

The swap function has two integer pointers as parameters. the swap function is used for swapping two elements by initializing variables. The last node function has one parameter which is the pointer. The last node function is used for finding the last element by looping through the node root and then ends by returning the root.

```
Node* partition(Node *l, Node *h)
{
    int x = h->data;

    Node *i = l->prev;

    for (Node *j = l; j != h; j = j->next)
    {
```

```
        if (j->data <= x)
        {
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next;
    swap(&(i->data), &(h->data));
    return i;
}
```

The partition function has two node parameters. partition function identifies the last element as the pivot, positions the pivot element correctly in the sorted array, and then moves all smaller (less than pivot) items to the left and all larger elements to the right of the pivot.

```
void _quickSort(Node* l, Node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        Node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}
void quickSort(Node *head)
{

    _quickSort(head, h);
}
```

The _quicksort function has two node parameters, and this function uses a quick sort algorithm recursively. The quick sort uses an if statement to check certain conditions to use the partition and run itself again with new numbers. The other quicksort function has one node parameter. other quicksort function that finds the last node and calls the recursive quicksort.

```
void printList(Node *head)
{
```

```
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the
beginning of the Doubly Linked List */
void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the
    beginning, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list of the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Print list function that has one node parameter pointer that prints the contents of the array by a while loop. A push function that has one node parameter and an integer parameter's function is used for inserting a node at the beginning of a double linked.

```
int main()
{
    Node *a = NULL;
```

```
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

The main function pushes the number inside the list and then prints the list before the quick sort and after.

Task 3:

Code explanation

The goal is to write four different sort algorithms: heap sort, bubble sort, selection sort, and insertion sort. for the four methods to return the total number of comparisons made to the sorted array (arr). The minimal heap sorting method serves as the foundation for the heap_sort implementation. Create a function named test_comparisons() that uses 30 random arrays to execute the preceding 4 functions. Thirty arrays sorted thirty arrays arranged inversely and thirty random arrays. Code have to Determine how many comparisons are made and how long it takes to complete each n in seconds. Then In the console, print the findings as tables. Store the outcome in an Excel spreadsheet, with separate sheets for the number of comparisons and execution time. then Plot the outcomes as a chart (two independent charts) in Excel.

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>
using namespace std;
```

first, import the needed libraries for the code.

```
int bubblcount(int ar[], int n)
{
   int numcomp = 0;
   bool swapcurs;
   for (int i = 0; i < n - 1; i++)
   {
      swapcurs = false;
      for (int x = 0; x < n - i - 1; x++)
      {
         if (ar[x] > ar[x+1])
         {
            swap(ar[x], ar[x+1]);
            swapcurs = true;
         }
         numcomp++;
      }
      if (!swapcurs)
      {
         break;
      }
   }
   return numcomp;
}
```

One of the simplest sorting methods, bubble sort, involves frequently switching nearby components that are out of order. Due to its relatively high average and worst-case time complexity, this approach is not appropriate for huge data sets. the Bubble Sort algorithm, Move left, compare the items that are next to it, and insert the higher element on the right. The largest element is initially shifted to the rightmost end in this manner. Once the data is sorted, this procedure is repeated to identify the second largest place, and so on.

```
int selectioncount(int ar[], int n)
{
   int numcomp = 0;
   int min;
```

```
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (ar[j] < ar[min])
            {
                min = j;
            }
            numcomp++;
        }
        swap(ar[i], ar[min]);
    }
    return numcomp;
}
```

Selection sort is a straightforward and effective sorting algorithm that functions by repeatedly choosing the largest (or smallest) element from the list's unsorted section and shifting it to the sorted section. The algorithm alternates the initial member of the unsorted section of the list with the smallest (or largest) element it repeatedly selects from the unsorted segment. To sort the complete list, this step is repeated for the remaining unsorted portion.

```
nt heap(int arr[], int n, int i)
{
    int numcomp = 0;
    int minvl = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n)
    {
        numcomp++;
        if (arr[l] < arr[minvl])
        {
            minvl = l;
        }
    }
    if (r < n)
    {
```

```
        numcomp++;
        if (arr[r] < arr[minvl])
        {
            minvl = r;
        }
    }
    if (minvl != i)
    {
        swap(arr[i], arr[minvl]);
        numcomp += heap(arr, n, minvl);
    }
    return numcomp;
}

int heapsocount(int arr[], int n)
{
    int numcomp = 0;
    for (int x = (n - 2) / 2; x >= 0; x--)
    {
        numcomp += heap(arr, n, x);
    }
    for (int x = n - 1; x > 0; x--)
    {
        swap(arr[0], arr[x]);
        numcomp += heap(arr, x, 0);
    }
    return numcomp;
}
```

Heap sort is a sorting method based on comparison and the Binary Heap data structure. It is comparable to the selection sort in which the minimum element is located first and positioned first. For the remaining components, follow the same procedure. After utilizing the heapify(heap) function to turn the array into a heap data structure, remove each Max-heap root node one at a time and replace it with the heap's last node before heapifying the heap's root. Until the heap's size is more than 1, keep doing this. Create a heap using the supplied input array. Until the heap has just one element, keep doing the following steps: Place the final member in the heap in place of the largest element, the root element. Eliminate the heap's final part, which is now in the proper place. Add to the heap the remaining components. The elements in the input array are rearranged to produce the sorted array.

```
int insertsocount(int ar[], int n)
{
   int s = 0;
   int numcomp = 0;

   for (int i = 1; i < n; i++)
   {
      s = ar[i];
      for (int j = i - 1; j > -1; j--)
      {
         numcomp++;
         if (s < ar[j])
         {
            swap(ar[j], ar[j+1]);
         }
         else
         {
            break;
         }
      }
   }
   return numcomp;
}
```

A straightforward sorting method called insertion sort creates a sorted array one element at a time. Because it doesn't need any more memory space than the initial array, it is referred to as a "in-place" sorting method. Since the first member in the array is presumed to be sorted, we must begin with the second element. If the second element is smaller than the first after comparing the two, then switch them. To place the third element among the first three in the proper order, move it to the second element, compare it with that element, and then move back to the first element. Proceed with the sorting process, comparing each element with the ones that came before it and switching as necessary to arrange it correctly in the hierarchy of sorted elements. Continue sorting until the full array is done.

```
void reset(int te[], int ar[], int z )
{
   for (int x = 0; x < z; x++)
```

```
    {
        te[x] = ar[x];
    }
}
```

The function reet uses three parameters: two arrays and one integer. The reset function is used for reshuffling the sorted array using a for loop that stores all the numbers of ar array in te array.

```
void exsorts(int te[], int ar[], int x, string artype, string soalg)
{
    reset(te, ar, x);
    auto start = chrono::high_resolution_clock::now();
    int st;
    if (soalg == "Heap")
    {
        st =  heapsocount(te, x);
    }
    else if (soalg == "Bubble")
    {
        st = bubblcount(te, x);
    }
    else if (soalg == "Selection")
    {
        st =  selectioncount(te, x);
    }
    else
    {
        st = insertsocount(te, x);
    }
    auto end = chrono::high_resolution_clock::now();
    double exec_time = chrono::duration_cast<chrono::nanoseconds>(end - start).count() * 1e-9;
    dataFile << x << "," << artype << "," << soalg << "," << st << "," << exec_time << "\n";
    pTable(x, artype, soalg, st, exec_time);
}
```

The exsorts function has five parameters. exsorts first, runs the reshuffle function. Second, the if statements check if soalg is equal to the needed sort algorithm. If so the it will run the sort algorithm needed. Third , print table function takes five parameters and it is used for printing the table with the required things.

```
void testcompar()
{
    srand(time(0));

    dataFile << "Size of Array,Type of Array,Sorting Algorithm,Number of Comparisons,Execution Time\n";
    cout << left << setw(20) << "Size of Array" << left << setw(20) << "Type of Array"
    << left << setw(20) << "Sorting Algorithm" << left << setw(25) << "Number of Comparisons"
    << left << setw(20) << "Execution Time" << "\n";

    for (int m = 1;m < 31; m++)
    {
        int* randarr = new int[m];
        int* sortedarr = new int[m];
        int* invarr = new int[m];
        int number1 = 0;
        int number2 = 0;
        for (int x = 0; x < m; x++)
        {
            randarr[x] = rand() % (m * 20);
            number1 += rand() % (m * 20);
            sortedarr[x] = number1;
            number2 += rand() % (m * 20);
            invarr[m-x-1] = number2;
        }
        rsorts(randarr, "Random", m);
        rsorts(sortedarr, "Sorted", m);
        rsorts(invarr, "Inversely-sorted", m);
        delete[] randarr;
        delete[] sortedarr;
        delete[] invarr;
    }
}
```

The testcompar function is used for creating thirty random, sorted and inversely sorted array. then used the run_sorts function 3 times for every type of every to sort the arrays. after that, the arrays get deleted to create new arrays, and so on.

Task4:

Code explanation

The goal of this task is to Provide a recursive program that sorts an unsorted array (A) of numbers so that all elements smaller than or equal to k appear before any elements larger than k. The program should take one argument, an integer k, and an array of integers A. Regarding an array of n values, how complex is your program? Recursive and iterative algorithms differ in terms of complexity.

```
class Node
{
   public:
      string data;
      Node* left;
      Node* right;

      Node(string d)
      {
         data = d;
         left = NULL;
         right = NULL;
      }

      ~Node()
      {
         delete left;
         delete right;
      }
};
```

First, create the class node that has three variables: one string and two node pointers. Second, create the constructor that has one parameter that will be initialized with the string variable and the rest will be initialized as null. third, create the destructor that destroys the node pointers.

```
lass Tree
{
   public:
      Node* root;
```

```
Tree()
{
    root = NULL;
}
```

First, create the class tree that has one variable: the root node variable. Second, create the constructor and have the node pointer initialized as null. third, create the destructor that deletes the root.

```
double rCalcRes()
    {
        return calcR(root);
    }

    double calcR(Node* current)
    {
        if (current->l == NULL && current->r == NULL)
        {
            return stod(current->d);
        }
        else
        {
            string m = current->d;
            if (m == "+")
            {
                return calcR(current->l) + calcR(current->r);
            }
            if (m == "-")
            {
                return calcR(current->l) - calcR(current->r);
            }
            if (m == "*")
            {
                return calcR(current->l) * calcR(current->r);
            }
```

```
            else
            {
                return calcR(current->l) / calcR(current->r);
            }
        }
    }
```

The rcalcres function is used to return the root of the tree. The calcr function has one node pointer as a parameter. The function calc result is used for calculating the total answer for the different expressions.

```
void rnExTre(string sx)
    {
        root = epTre(sx);
    }
```

The rnExTre has one parameter which is a string. rnExTre function is used for storing the string after using eptre function in the root variable.

```
string cleanS(string s)
    {
        stack<int> stacko;
        vector<int> vicko;
        int n = s.length();
        for (int i = 0; i < n - 1; i++)
        {
            char currElement = s[i];
            if (currElement == '(')
            {
                stacko.push(i);
            }
            else if (currElement == ')')
            {
                stacko.pop();
            }
```

```
        }
        if (stacko.size() == 1 && stacko.top() == 0)
        {
            return s.substr(1, n-2);
        }
        return s;
    }
```

The function cleans that has one parameter which is a string. Cleans function is used for checking if there is a balanced parentheses

```
int prece(string s, vector<int> vicko)
    {
        for (char ope : {'-', '+', '*', '/'})
        {
            for (int i = 0; i < vicko.size(); i++)
            {
                if (s[vicko.at(i)] == ope)
                {
                    return i;
                }
            }
        }
    }
```

The prece function has two parameters: a string and a vector. The prece function is used to apply the bidmas . the BIDMAS rule serves as a memory aid for the sequence of operations used in computations. Simply put, operations in mathematics are the many manipulations we can apply to numbers. Its acronym stands for Addition, Subtraction, Division, Multiplication, and Brackets.

```
int hei(Node* root)
    {
        if (root == nullptr)
        {
            return 0;
        }
        return 1 + max(hei(root->l), hei(root->r));
    }
```

```
int getcolmun(int h)
{
    if (h == 1)
    {
        return 1;
    }
    return getcolmun(h - 1) + getcolmun(h - 1) + 1;
}
```

The function hei has one parameter which is a node root. The hei function Is used for returning the max height for the root left and right. The function column has one parameter which is an integer. The get column function Is used for returning the column needed by using the get column function recursively.

```
void printTree(string **M, Node *root, int col, int row, int height)
    {
        if (root == NULL)
        {
            return;
        }
        M[row][col] = root->d;
        printTree(M, root->l, col - pow(2, height - 2), row + 1, height - 1);
        printTree(M, root->r, col + pow(2, height - 2), row + 1, height - 1);
    }

    void pExTre()
    {
        int ho = hei(root);
        int coll = getcolmun(ho);
        string **t = new string*[ho];
        for (int i = 0; i < ho; i++)
        {
            t[i] = new string[coll];
        }
        printTree(t, root, coll / 2, 0, ho);
        for (int i = 0; i < ho; i++)
        {
            for (int j = 0; j < coll; j++)
```

```
            {
                if (t[i][j] == "")
                {
                    cout << " ";
                }
                else
                {
                    cout << t[i][j];
                }
            }
            cout << endl;
        }
    }
};
```

The print tree and pextre functions are used for printing the final tree.

Task 6:

Code explanation

The goal of task 6 is to create the editor print words on a new line based on the line width. The input consists of a list of words and line width. The requirement is to write a program that places line breaks into Words so that the lines are printed neatly, minimizing gaps. Assuming that the length of each word is smaller than LW.

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

void ttWr( vector<string> wrds, int lWidth) {
```

```
    int x = 0;
    int linm = 1;
    int ln = 0;
    cout << "line 1: ";
    while (x < wrds.size()) {
        ln += wrds[x].length();
        if (ln > lWidth) {
            linm++;
            cout << "\nline " << linm << ": " << wrds[x] << " ";
            ln = wrds[x].length() + 1;
        }
        else {
            cout << wrds[x] << " ";
            ln++;
        }
        x++;
    }
}
```

First, Import the needed libraries and create a function text warp. the .Text wrap function that takes two parameters a vector string and an integer. The second three variables x , line, and ln, are initialized with a specfic numbers for the code implementation. Third, create a while loop that loops through word size, and the Ln variable adds the word's length for every word. Fourth, there is an if statement is used for creating a new line for the new word after a space. Else it will print the rest of the words

```
int main() {
    string s;
    int linewid;
    cout << "text : ";
    getline(cin, s);
    cout << "line width : ";
    cin >> linewid;
    string ter;
    istringstream iss(s);
    vector<string> x;
    while (iss >> ter) {
        x.push_back(ter);
```

```
    }
    ttWr(x, linewid);
    return 0;
}
```

The main function takes the input from the user and then applies the text-wrap function

Task 7

Social media advertising often focuses on interest-based organizations. The goal is to create an algorithm that takes the relationships between various social media users (followers) and returns the groups with the most connections. G=(V,E), where V is the set of nodes and E is the set of edges, is the directed graph that connects the users. When there is a path from a to b and from b to a, the two nodes $a, b \in U$, should be satisfied by the group $U \subseteq V$.my plan for the code implementation is to create a class graph that has several vertices and adds edges to the graph. The class also has function depth-first search is an algorithm for navigating or searching through tree or graph data structures. Starting at the root node (assuming, in the case of a graph, an arbitrary node), the method proceeds as far as it can along each branch before backtracking.

```
##include <iostream>
#include <list>

using namespace std;

class Graph {
    int nVertices;
    list<int>* adjLis;
    bool* vis;
    list<int> fin;
```

First, Import the needed libraries for the code then create the class graph that has four variables the number of vertices, a list of adjacent nodes, vis, and a list of fin nodes. The variables are in the private section.

```
Graph(int siOfGrph) {
```

```
        nVertices = siOfGrph;
        adjLis = new list<int>[siOfGrph];
        vis = new bool[siOfGrph];
    }

    void adEd(int vx1, int vx2) {
        adjLis[vx1].push_back(vx2);
    }
```

In the public section create a constructor that will be able to use the variables in the private section, the constructor number of vertices, adjacent nodes, and visited bool. Add an edge function that has two parameters that are vertices and connect the two vertices.

```
void execDF() {
        for (int m = 0; m < nVertices; m++) {
            if (!vis[m]) {
                depthFirstSearch(m);
            }
        }
        tranGrph();
        for (int x = 0; x < nVertices; x++) {
            vis[x] = false;
        }
        int c = 1;
        list<int>::reverse_iterator y;
        for (y = fin.rbegin(); y != fin.rend(); y++) {
            if (!vis[*y]) {
                cout << "Group #" << c << ": {";
                transDF(*y);
                cout << "}\n";
                c++;
            }
        }
    }
```

Execdfs function is used for looping through the vertices and implementing the depth-first search on the vertices. then running the transpose graph function and there is a loop that prints the different groups. Lastly, it runs the transdf function and creates a new line.

```
void depthFirstSearch(int verx) {
        vis[verx] = true;
        list<int> adjList = adjLis[verx];
        list<int>::iterator abjver;
        for (abjver = adjList.begin(); abjver != adjList.end(); ++abjver) {
            if (!vis[*abjver]) {
                depthFirstSearch(*abjver);
            }
        }
        fin.push_back(verx);
    }
```

The depth-first search is an algorithm for navigating or searching through tree or graph data structures. Starting at the root node (assuming, in the case of a graph, an arbitrary node), the method proceeds as far as it can along each branch before backtracking. Depth-first search has certain steps that must go through. The first step is to visit node 0 and the other unvisited nodes go to the stack. Currently, the first Node at the top of the stack will be popped from the stack, and keep the rest of the unvisited neighboring nodes in the stack. the algorithm will keep on doing these steps until all the nodes are in the visited list.

```
void transDF(int ve) {
        vis[ve] = true;
        list<int> adjList = adjLis[ve];
        cout << ve << " ";
        list<int>::iterator c;
        for (c = adjList.begin(); c != adjList.end(); ++c) {
            if (!vis[*c]) {
                transDF(*c);
            }
        }
    }
```

The transdfs function has one parameter. transpose dfs function is used for transposing by looping Through adjacent nodes list and checking if the node was not visited to transpose it

```
void tranGrph() {
        list<int>* nAjLis = new list<int>[nVertices];
        for (int m = 0; m < nVertices; m++) {
            list<int>::iterator s;
            for (s = adjLis[m].begin(); s != adjLis[m].end(); ++s) {
                nAjLis[*s].push_front(m);
            }
        }
        for (int x = 0; x < nVertices; x++) {
            adjLis[x].clear();
            list<int>::iterator s;
            for (s = nAjLis[x].begin(); s != nAjLis[x].end(); ++s) {
                adjLis[x].push_front(*s);
            }
        }
        delete[] nAjLis;
    }
};
```

The transpose graph function is The initial stage of transposing a graph and involves going through the adjacency list. Once we locate a vertex 1 in the adjacency list of vertex 2 that represents an edge from 2 to 1 in the original graph, there will be an edge that will be added from 1 to 2 in the transposed graph, that is, add u in the adjacency list of vertex v of the created graph. So, we may obtain the transpose graph by iterating through lists containing all of the main graph's vertices. With 1 vertex representing the number of graph vertices and 3 vertex representing the number of graph edges.

```
int main() {
    Graph m(11);
    m.adEd(0, 1);
    m.adEd(1, 2);
    m.adEd(1, 3);
    m.adEd(3, 0);
    m.adEd(3, 4);
    m.adEd(4, 5);
    m.adEd(5, 3);
    m.adEd(6, 5);
    m.adEd(6, 7);
    m.adEd(6, 8);
```

```
    m.adEd(4, 5);
    m.adEd(9, 3);
    m.adEd(4, 5);
    m.execDF();
    return 0;
}
```

The main is used for creating an object for the graph, adding the edges, and executing the first depth search algorithm.

Conclusion

In conclusion, solving the tasks helped me to understand different algorithms like quick sort, heap sort, bubble sort, selection sort, insertion sort, matrices, tree graphs, and normal graphs. The coursework did challenge me on those different topics. That took time for critical thinking and solving the errors that were faced during the solving of the tasks. The knowledge gained from the coursework is crucial for me because it will help me go through my internship in the summer.

Appendices(the code)

Task 2:

```
#include <bits/stdc++.h>
using namespace std;

class Node
{
    public:
    int data;
    Node *next;
    Node *prev;
```

```
};

void swap ( int* a, int* b )
{ int t = *a; *a = *b; *b = t; }


Node *lastNode(Node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}


Node* partition(Node *l, Node *h)
{
    int x = h->data;

    Node *i = l->prev;

    for (Node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/
void _quickSort(Node* l, Node *h)
{
```

```
    if (h != NULL && l != h && l != h->next)
    {
        Node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}


void quickSort(Node *head)
{
    Node *h = lastNode(head);

    _quickSort(head, h);
}

void printList(Node *head)
{
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}


void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node;
    new_node->data = new_data;


    new_node->prev = NULL;

    new_node->next = (*head_ref);
```

```
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node ;

    (*head_ref) = new_node;
}

int main()
{
    Node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Task 3:

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>
using namespace std;

ofstream dataFile("sort_results.csv");

int heap(int arr[], int n, int i)
```

```
{
    int numcomp = 0;
    int minvl = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n)
    {
        numcomp++;
        if (arr[l] < arr[minvl])
        {
            minvl = l;
        }
    }
    if (r < n)
    {
        numcomp++;
        if (arr[r] < arr[minvl])
        {
            minvl = r;
        }
    }
    if (minvl != i)
    {
        swap(arr[i], arr[minvl]);
        numcomp += heap(arr, n, minvl);
    }
    return numcomp;
}

int heapsocount(int arr[], int n)
{
    int numcomp = 0;
    for (int x = (n - 2) / 2; x >= 0; x--)
    {
        numcomp += heap(arr, n, x);
    }
    for (int x = n - 1; x > 0; x--)
```

```
    {
        swap(arr[0], arr[x]);
        numcomp += heap(arr, x, 0);
    }
    return numcomp;
}

int bubblcount(int ar[], int n)
{
    int numcomp = 0;
    bool swapcurs;
    for (int i = 0; i < n - 1; i++)
    {
        swapcurs = false;
        for (int x = 0; x < n - i - 1; x++)
        {
            if (ar[x] > ar[x+1])
            {
                swap(ar[x], ar[x+1]);
                swapcurs = true;
            }
            numcomp++;
        }
        if (!swapcurs)
        {
            break;
        }
    }
    return numcomp;
}

int selectioncount(int ar[], int n)
{
    int numcomp = 0;
    int min;
    for (int i = 0; i < n - 1; i++)
    {
```

```
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (ar[j] < ar[min])
            {
                min = j;
            }
            numcomp++;
        }
        swap(ar[i], ar[min]);
    }
    return numcomp;
}

int insertsocount(int ar[], int n)
{
    int s = 0;
    int numcomp = 0;

    for (int i = 1; i < n; i++)
    {
        s = ar[i];
        for (int j = i - 1; j > -1; j--)
        {
            numcomp++;
            if (s < ar[j])
            {
                swap(ar[j], ar[j+1]);
            }
            else
            {
                break;
            }
        }
    }
    return numcomp;
}
```

```
void reset(int te[], int ar[], int z )
{
    for (int x = 0; x < z; x++)
    {
        te[x] = ar[x];
    }
}

void pTable(int n, string artype, string sotype, int rt, double extime)
{
    cout << left << setw(20) << n << left << setw(20) << artype << left << setw(20) << sotype << left << setw(25) << rt
<< left << setw(20) << extime << "\n";
}

void exsorts(int te[], int ar[], int x, string artype, string soalg)
{
    reset(te, ar, x);
    auto start = chrono::high_resolution_clock::now();
    int st;
    if (soalg == "Heap")
    {
        st =  heapsocount(te, x);
    }
    else if (soalg == "Bubble")
    {
        st = bubblcount(te, x);
    }
    else if (soalg == "Selection")
    {
        st =  selectioncount(te, x);
    }
    else
    {
        st = insertsocount(te, x);
    }
    auto end = chrono::high_resolution_clock::now();
```

```
    double exec_time = chrono::duration_cast<chrono::nanoseconds>(end - start).count() * 1e-9;
    dataFile << x << "," << artype << "," << soalg << "," << st << "," << exec_time << "\n";
    pTable(x, artype, soalg, st, exec_time);
}

void rsorts(int ar[], string artype, int n)
{
    int* temp = new int[n];
    exsorts(temp, ar, n, artype, "Heap");
    exsorts(temp, ar, n, artype, "Bubble");
    exsorts(temp, ar, n, artype, "Selection");
    exsorts(temp, ar, n, artype, "Insertion");
}

void testcompar()
{
    srand(time(0));

    dataFile << "Size of Array,Type of Array,Sorting Algorithm,Number of Comparisons,Execution Time\n";
    cout << left << setw(20) << "Size of Array" << left << setw(20) << "Type of Array"
    << left << setw(20) << "Sorting Algorithm" << left << setw(25) << "Number of Comparisons"
    << left << setw(20) << "Execution Time" << "\n";

    for (int m = 1;m < 31; m++)
    {
        int* randarr = new int[m];
        int* sortedarr = new int[m];
        int* invarr = new int[m];
        int number1 = 0;
        int number2 = 0;
        for (int x = 0; x < m; x++)
        {
            randarr[x] = rand() % (m * 20);
            number1 += rand() % (m * 20);
            sortedarr[x] = number1;
            number2 += rand() % (m * 20);
            invarr[m-x-1] = number2;
```

```
        }
        rsorts(randarr, "Random", m);
        rsorts(sortedarr, "Sorted", m);
        rsorts(invarr, "Inversely-sorted", m);
        delete[] randarr;
        delete[] sortedarr;
        delete[] invarr;
    }
}

int main()
{
    testcompar();
    dataFile.close();
    return 0;
}
```

Task 4

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

class Node
{
    public:
        string d;
        Node* l;
        Node* r;

        Node(string data)
        {
            d = data;
            l = NULL;
            r = NULL;
```

```
    }



};

class Tree
{
  public:
    Node* root;

    Tree()
    {
        root = NULL;
    }




    double rCalcRes()
    {
        return calcR(root);
    }

    double calcR(Node* current)
    {
        if (current->l == NULL && current->r == NULL)
        {
            return stod(current->d);
        }
        else
        {
            string m = current->d;
            if (m == "+")
            {
                return calcR(current->l) + calcR(current->r);
            }
            if (m == "-")
            {
```

```
            return calcR(current->l) - calcR(current->r);
        }
        if (m == "*")
        {
            return calcR(current->l) * calcR(current->r);
        }
        else
        {
            return calcR(current->l) / calcR(current->r);
        }
    }
}

void rnExTre(string sx)
{
    root = epTre(sx);
}

Node* epTre(string stt)
{
    stack<int> stacko;
    vector<int> vicko;
    string newStr = cleanS(stt);
    int n = newStr.length();
    for (int i = 0; i < n; i++)
    {
        char cuElent = newStr[i];
        if (cuElent == '(')
        {
            stacko.push(i);
        }
        else if (cuElent == ')')
        {
             stacko.pop();
        }
        else if (stacko.size() == 0 && (cuElent == '+' || cuElent == '-' || cuElent == '*' || cuElent == '/'))
        {
```

```
            vicko.push_back(i);
        }
    }
    int m;
    if (vicko.size() == 0)
    {
        return new Node(newStr);
    }
    else if (vicko.size() == 1)
    {
        m = vicko.at(0);
    }
    else
    {
        int operSymbolIndex = prece(newStr, vicko);
        m= vicko.at(operSymbolIndex);
    }
    Node* ne1 = new Node(newStr.substr(m, 1));
    ne1->l = epTre(newStr.substr(0, m));
    ne1->r = epTre(newStr.substr(m+1, n-m-1));
    return ne1;
}

string cleanS(string s)
{
    stack<int> stacko;
    vector<int> vicko;
    int n = s.length();
    for (int i = 0; i < n - 1; i++)
    {
        char currElement = s[i];
        if (currElement == '(')
        {
            stacko.push(i);
        }
        else if (currElement == ')')
        {
```

```
            stacko.pop();
        }
    }
    if (stacko.size() == 1 && stacko.top() == 0)
    {
        return s.substr(1, n-2);
    }
    return s;
}

int prece(string s, vector<int> vicko)
{
    for (char ope : {'-', '+', '*', '/'})
    {
        for (int i = 0; i < vicko.size(); i++)
        {
            if (s[vicko.at(i)] == ope)
            {
                return i;
            }
        }
    }
}

void rExTreTra()
{
    exTreTra(root);
}

void exTreTra(Node* root)
{
    if (!root)
    {
        return;
    }
    cout << "(";
    exTreTra(root->l);
```

```
    cout << root->d;
    exTreTra(root->r);
    cout << ")";
}

int hei(Node* root)
{
    if (root == nullptr)
    {
        return 0;
    }
    return 1 + max(hei(root->l), hei(root->r));
}

int getcolmun(int h)
{
    if (h == 1)
    {
        return 1;
    }
    return getcolmun(h - 1) + getcolmun(h - 1) + 1;
}

void printTree(string **M, Node *root, int col, int row, int height)
{
    if (root == NULL)
    {
        return;
    }
    M[row][col] = root->d;
    printTree(M, root->l, col - pow(2, height - 2), row + 1, height - 1);
    printTree(M, root->r, col + pow(2, height - 2), row + 1, height - 1);
}

void pExTre()
{
    int ho = hei(root);
```

```
        int coll = getcolmun(ho);
        string **t = new string*[ho];
        for (int i = 0; i < ho; i++)
        {
            t[i] = new string[coll];
        }
        printTree(t, root, coll / 2, 0, ho);
        for (int i = 0; i < ho; i++)
        {
            for (int j = 0; j < coll; j++)
            {
                if (t[i][j] == "")
                {
                    cout << " ";
                }
                else
                {
                    cout << t[i][j];
                }
            }
            cout << endl;
        }
    }
};


int main()
{
    Tree* tr = new Tree();
    string x = "(((4+1)*(1-0))/((2+2)+((4-2)-2))*6)";
    tr->rnExTre(x);
    cout << "Expression tree:" << endl;
    tr->pExTre();
    cout << endl;
    cout << "Value: " << tr->rCalcRes() << endl;
    cout << endl;
    return 0;
```

```
}
```

Task 6:

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

void ttWr( vector<string> wrds, int lWidth) {
    int x = 0;
    int linm = 1;
    int ln = 0;
    cout << "line 1: ";
    while (x < wrds.size()) {
        ln += wrds[x].length();
        if (ln > lWidth) {
            linm++;
            cout << "\nline " << linm << ": " << wrds[x] << " ";
            ln = wrds[x].length() + 1;
        }
        else {
            cout << wrds[x] << " ";
            ln++;
        }
        x++;
    }
}

int main() {
    string s;
    int linewid;
    cout << "text : ";
    getline(cin, s);
    cout << "line width : ";
    cin >> linewid;
```

```
    string ter;
    istringstream iss(s);
    vector<string> x;
    while (iss >> ter) {
        x.push_back(ter);
    }
    ttWr(x, linewid);
    return 0;
}
```

Task 7:

```
#include <iostream>
#include <list>

using namespace std;

class Graph {
    int nVertices;
    list<int>* adjLis;
    bool* vis;
    list<int> fin;

    public:

        Graph(int siOfGrph) {
            nVertices = siOfGrph;
            adjLis = new list<int>[siOfGrph];
            vis = new bool[siOfGrph];
        }

        void adEd(int vx1, int vx2) {
            adjLis[vx1].push_back(vx2);
        }

        void execDF() {
```

```
    for (int m = 0; m < nVertices; m++) {
        if (!vis[m]) {
            depthFirstSearch(m);
        }
    }
    tranGrph();
    for (int x = 0; x < nVertices; x++) {
        vis[x] = false;
    }
    int c = 1;
    list<int>::reverse_iterator y;
    for (y = fin.rbegin(); y != fin.rend(); y++) {
        if (!vis[*y]) {
            cout << "Group #" << c << ": {";
            transDF(*y);
            cout << "}\n";
            c++;
        }
    }
}

void depthFirstSearch(int verx) {
    vis[verx] = true;
    list<int> adjList = adjLis[verx];
    list<int>::iterator abjver;
    for (abjver = adjList.begin(); abjver != adjList.end(); ++abjver) {
        if (!vis[*abjver]) {
            depthFirstSearch(*abjver);
        }
    }
    fin.push_back(verx);
}

void transDF(int ve) {
    vis[ve] = true;
    list<int> adjList = adjLis[ve];
    cout << ve << " ";
```

```
        list<int>::iterator c;
        for (c = adjList.begin(); c != adjList.end(); ++c) {
            if (!vis[*c]) {
                transDF(*c);
            }
        }
    }

    void tranGrph() {
        list<int>* nAjLis = new list<int>[nVertices];
        for (int m = 0; m < nVertices; m++) {
            list<int>::iterator s;
            for (s = adjLis[m].begin(); s != adjLis[m].end(); ++s) {
                nAjLis[*s].push_front(m);
            }
        }
        for (int x = 0; x < nVertices; x++) {
            adjLis[x].clear();
            list<int>::iterator s;
            for (s = nAjLis[x].begin(); s != nAjLis[x].end(); ++s) {
                adjLis[x].push_front(*s);
            }
        }
        delete[] nAjLis;
    }
};

int main() {
    Graph m(11);
    m.adEd(0, 1);
    m.adEd(1, 2);
    m.adEd(1, 3);
    m.adEd(3, 0);
    m.adEd(3, 4);
    m.adEd(4, 5);
    m.adEd(5, 3);
    m.adEd(6, 5);
```

```
    m.adEd(6, 7);
    m.adEd(6, 8);
    m.adEd(4, 5);
    m.adEd(9, 3);
    m.adEd(4, 5);
    m.execDF();
    return 0;
}
```