Test structuri de date

Agusoaei Alexandru 133

Lab 1)

1. Pasul comun este selectarea maximului(sau a minimului) si mutarea elementului la final(sau la inceput). In SelectionSort, selectarea maximului are complexitatea O(n), iar in HeapSort selectarea este mai eficienta, deoarece se foloseste o structura de heap, complexitatea fiind O(logn).
2. In HeapSort, trebuie sa construim max-heapul(sau min-heapul) pentru a putea incepe algoritmul. Dupa acest pas stim sigur ca radacina este cel mai mare (sau mai mic) element, mutam acest element la capatul vectorului si repetam acest algoritm, excluzand elementul deja pus la locul lui in vector. Folosind heapify() in mod repetat, putem crea acest heap dintr-un vector.

Lab 2)

1. if(head == nullptr) return -1;

Nod\* final = head;

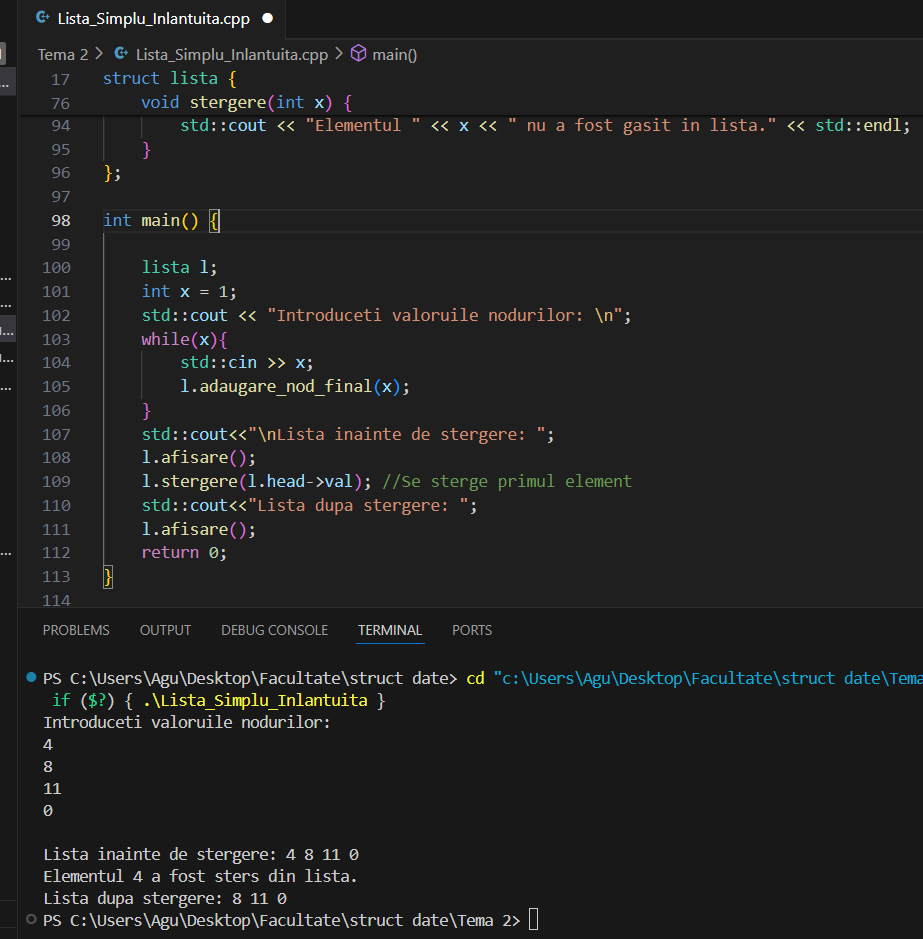
while(aux->final != head)

final = final->next;

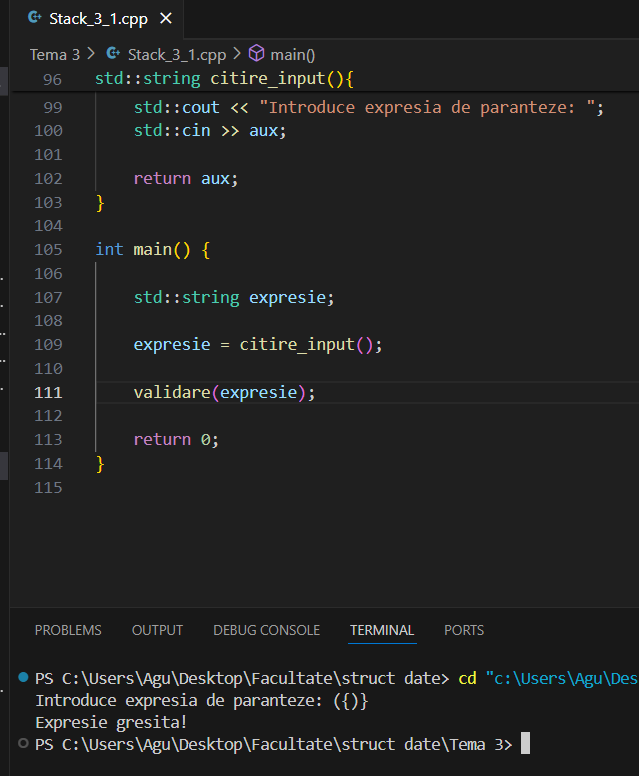
head = head->next;

final->next = head;

//Aflam nodul final al listei, mutam head-ul catre noul head si la final mutam next-ul nodului final catre noul head al listei

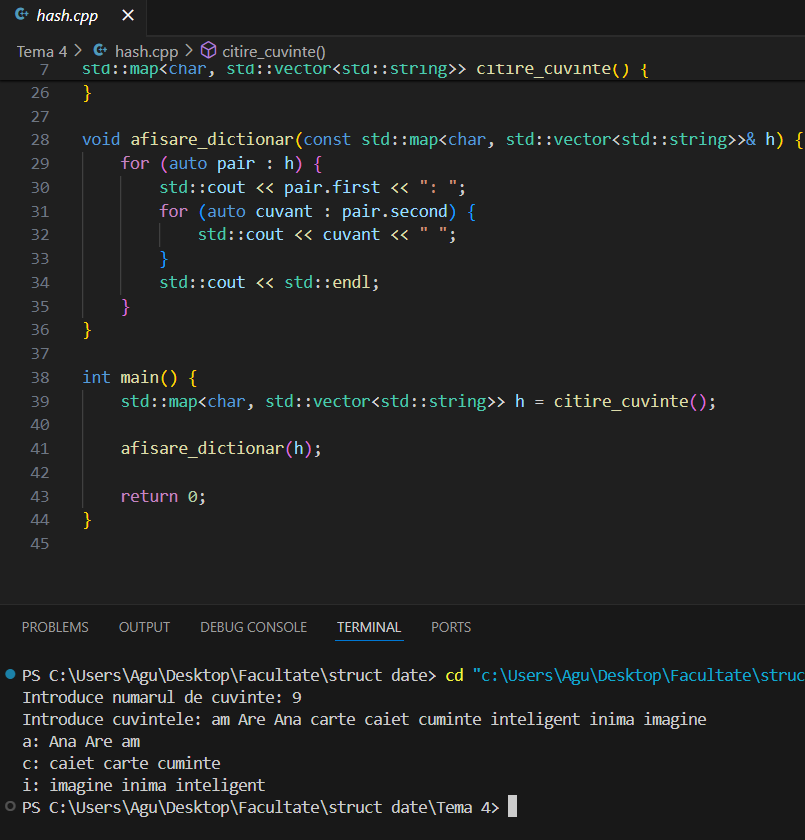


Lab 3)



1. Initial plecam cu un vector pentru rezultate initializat cu -1. Parcurgem vectorul nostrum de la final la inceput si succesiv punem pe stiva elementele pentru a le putea compara pe fiecare cu varful stivei, daca stiva este goala inseamna ca elementul respective nu are un numar mai mare decat el care sa urmeze si ramane -1 in vectorul pentru rezultat, altfel vectorul rezultat va contine varful stivei.

Lab 4)



Lab 5)

1. Stim ca intr-un vector sortat, elementul din mijloc are un numar egal de elemente mai mici si mai mari decat el. Alegem acest element ca si radacina pentru arborele nostru, fiul stang va fi mijlocul subvectorului din stanga, iar fiul drept mijlocul subvectorului din dreapta. In mod repetat, executam acest algoritm pentru fiecare fiu stang si drept in continuare.
2. Trebuie sa convertim cei 2 arbori in 2 vectori, reprezentand arborii in ineordine. Realizam interclasarea intre acesti 2 vectori, rezultand intr-un vector mai mare ce are ca elemente cei 2 arbori.

La final, trebuie sa folosim algoritmul de la cerinta 1 pentru a crea din acest vector sortat un arbore de cautare. Acest arbore de la final reprezinta interclasarea intre cei 2 arbori initiali.

Lab 6)

1. Algoritmul Kruskal se bazeaza pe muchii, sortandu-le crescator si parcurgandu-le. Algoritmul Prim se bazeaza pe noduri, alegand un nod arbitrar, dupa care se alege muchia de cost minim pentru a adauga un nou nod in arborele final de cost minim.
2. Incepem algoritmul prin alegerea arbitrara a unui nod. Consideram acest nod in arborele de cost minim si continuam prin a alege intr-o maniera greedy muchia de cel mai mic cost, adaugand nodul asociat in arborele de cost minim. Repetam procesul de a alege muchia de cel mai mic cost la care avem acces prin nodurile deja adaugate pana cand toate nodurile din arbore au fost adaugate. La finalul algoritmului, vom avea arborele de cost minim.

Lab1

#include <iostream>

#include <vector>

#include <fstream>

using namespace std;

void selectionSort(vector<int>& v) {

    int n = v.size();

    for (int i = 0; i < n - 1; i++) {

        int minIndex = i;

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

            if (v[j] < v[minIndex]) {

                minIndex = j;

            }

        }

        swap(v[i], v[minIndex]);

    }

}

void heapify(vector<int>& v, int len, int i) {

    int maxim = i;

    int left = 2 \* i + 1;

    int right = 2 \* i + 2;

    if (left < len && v[maxim] < v[left]) {

        maxim = left;

    }

    if (right < len && v[maxim] < v[right]) {

        maxim = right;

    }

    if (maxim != i) {

        swap(v[maxim], v[i]);

        heapify(v, len, maxim);

    }

}

void heapSort(vector<int>& v) {

    int len = v.size();

    // Construieste max-heapul (rearanjeaza vectorul)

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

        heapify(v, len, i);

    }

    // Extrage elementele unul cate unul din heap

    len -= 1;

    while (len > 0){

        // Muta radacina curenta la sfarsit

        swap(v[0], v[len]);

        // Apeleaza heapify pe heapul redus

        heapify(v, len, 0);

        len -= 1;

    }

}

vector<int> citire\_input(){

    vector<int> aux;

    int nr, x;

    cout<<"\nLungime vector: ";

    cin>>nr;

    cout<<"\nIntroduce elementele vectorului: \n";

    for(int i=1; i<=nr; i++){

        cin>>x;

        aux.push\_back(x);

    }

    return aux;

}

int main(){

    vector <int> v = citire\_input();

    //selectionSort(v);

    cout << "\nVector nesortat: ";

    for(int a : v){

        cout<<a<<" ";

    }

    cout<<"\nVector sortat: ";

    heapSort(v);

    for(int a : v){

        cout<<a<<" ";

    }

}

Lab 2

Lista Simplu Inlantuita

#include <iostream>

struct Nod {

    int val;

    Nod \*next;

    Nod(int x) {

        val = x;

        next = nullptr;

    }

    ~Nod() {

    }

};

struct lista {

    Nod \*head;

    lista() {

        head = nullptr;

    }

    ~lista() {

        Nod \*aux = head;

        while (aux != nullptr) {

            Nod \*next = aux->next;

            delete aux;

            aux = next;

        }

    }

    void adaugare\_nod\_final(int val) {

        Nod \*auxx = new Nod(val);

        if (head == nullptr) {

            head = auxx;

        } else {

            Nod \*aux = head;

            // Parcurgem lista până la ultimul element

            while (aux->next != nullptr) {

                aux = aux->next;

            }

            aux->next = auxx;

        }

    }

    void adaugare\_nod\_inceput(int val) {

        Nod \*auxx = new Nod(val);

        auxx->next = head;

        head = auxx;

    }

    void afisare() {

        Nod \*aux = head;

        while (aux != nullptr) {

            std::cout << aux->val << " ";

            aux = aux->next;

        }

        std::cout << std::endl;

    }

    void cautare(int x) {

        Nod \*aux = head;

        while (aux != nullptr) {

            if (aux->val == x) {

                std::cout << "Elementul " << x << " este in lista." << std::endl;

                return;

            }

            aux = aux->next;

        }

        std::cout << "Elementul " << x << " nu este in lista." << std::endl;

    }

    void stergere(int x) {

        Nod \*aux = head;

        Nod \*prev = nullptr;

        while (aux != nullptr) {

            if (aux->val == x) {

                if (prev != nullptr) {

                    prev->next = aux->next;

                } else {

                    head = aux->next;

                }

                delete aux;

                std::cout << "Elementul " << x << " a fost sters din lista." << std::endl;

                return;

            }

            prev = aux;

            aux = aux->next;

        }

        std::cout << "Elementul " << x << " nu a fost gasit in lista." << std::endl;

    }

};

int main() {

    lista l;

    int x = 1;

    std::cout << "Introduceti valoruile nodurilor: \n";

    while(x){

        std::cin >> x;

        l.adaugare\_nod\_final(x);

    }

    std::cout<<"\nLista inainte de stergere: ";

    l.afisare();

    l.stergere(l.head->val); //Se sterge primul element

    std::cout<<"Lista dupa stergere: ";

    l.afisare();

    return 0;

}

Lista Dublu Inlantuita

#include <iostream>

struct Nod{

    int val;

    Nod \*next;

    Nod \*prev;

    Nod(int x){

        val = x;

        next = nullptr;

        prev = nullptr;

    }

    ~Nod(){

        delete next;

        delete prev;

    }

};

struct lista\_dublu{

    Nod \*head;

    lista\_dublu(){

        head = nullptr;

    }

    ~lista\_dublu(){

        Nod \*aux = head;

        while(aux != nullptr){

            Nod \*next = aux->next;

            delete aux;

            aux = next;

        }

    }

    void adaugare\_nod\_final(int val){

        //Se creeaza nodul ce trebuie adaugat

        Nod \*newNod = new Nod(val);

        if(head == nullptr){

            head = newNod;

        }

        else {

            Nod \*aux = head;

            //Parcurgem lista pana la ultimul element

            while(aux->next != nullptr){

                aux = aux->next;

            }

            aux->next = newNod;

            newNod->prev = aux;

        }

    }

    void adaugare\_nod\_inceput(int val){

        Nod \*newNod = new Nod(val);

        newNod->next = head;

        head = newNod;

    }

    void afisare(){

        Nod \*aux = head;

        while(aux != nullptr){

            std::cout<<aux->val<<" ";

            aux = aux->next;

        }

        std::cout << std::endl;

    }

    void cautare(int x) {

        Nod \*aux = head;

        while (aux != nullptr) {

            if (aux->val == x) {

                std::cout << "Elementul " << x << " este in lista." << std::endl;

                return;

            }

            aux = aux->next;

        }

        delete aux;

        std::cout << "Elementul " << x << " nu este in lista." << std::endl;

    }

    void stergere(int x) {

        Nod \*aux = head;

        while (aux != nullptr) {

            if (aux->val == x) {

                if (aux->prev != nullptr) {

                    aux->prev->next = aux->next;

                } else {

                    head = aux->next;

                }

                if (aux->next != nullptr) {

                    aux->next->prev = aux->prev;

                }

                return;

            }

            aux = aux->next;

        }

        delete aux;

        std::cout << "Elementul " << x << " nu a fost gasit in lista." << std::endl;

    }

};

int main(){

    lista\_dublu lista;

    lista.adaugare\_nod\_final(4);

    lista.adaugare\_nod\_final(22);

    lista.adaugare\_nod\_final(14);

    lista.afisare();

    lista.adaugare\_nod\_inceput(3);

    lista.afisare();

    lista.cautare(4);

    lista.cautare(11);

    lista.stergere(22);

    lista.afisare();

}

Lista Circulara

#include <iostream>

struct Nod {

    int val;

    Nod \*next;

    Nod(int x) {

        val = x;

        next = nullptr;

    }

};

struct lista {

    Nod \*head;

    lista() {

        head = nullptr;

    }

    ~lista() {

        if (head != nullptr) {

            Nod \*aux = head;

            Nod \*primul = head;

            do {

                Nod \*next = aux->next;

                delete aux;

                aux = next;

            } while (aux != primul);

        }

    }

    void adaugare\_nod\_final(int val) {

        Nod \*nod = new Nod(val);

        if (head == nullptr) {

            head = nod;

            nod->next = head;

        } else {

            Nod \*aux = head;

            while (aux->next != head) {

                aux = aux->next;

            }

            aux->next = nod;

            nod->next = head;

        }

    }

    void adaugare\_nod\_inceput(int val) {

        Nod \*nod = new Nod(val);

        if (head == nullptr) {

            head = nod;

            nod->next = head;

        } else {

            Nod \*aux = head;

            while (aux->next != head) {

                aux = aux->next;

            }

            nod->next = head;

            head = nod;

            aux->next = head;

        }

    }

    void afisare() {

        if (head != nullptr) {

            Nod \*aux = head;

            do {

                std::cout << aux->val << " ";

                aux = aux->next;

            } while (aux != head);

            std::cout << std::endl;

        }

    }

    void cautare(int x) {

        if (head != nullptr) {

            Nod \*aux = head;

            do {

                if (aux->val == x) {

                    std::cout << "Elementul " << x << " este in lista." << std::endl;

                    return;

                }

                aux = aux->next;

            } while (aux != head);

        }

        std::cout << "Elementul " << x << " nu este in lista." << std::endl;

    }

    void stergere(int x) {

        if (head == nullptr) {

            std::cout << "Elementul " << x << " nu a fost gasit in lista." << std::endl;

            return;

        }

        Nod \*aux = head;

        Nod \*prev = nullptr;

        // gasim nodul de sters

        do {

            if (aux->val == x) {

                if (prev != nullptr) {

                    prev->next = aux->next;

                } else {

                    Nod \*last = head;

                    while (last->next != head) {

                        last = last->next;

                    }

                    head = aux->next;

                    last->next = head;

                }

                if (aux == head && aux->next == head) {

                    // Lista are un singur element care trebuie șters

                    head = nullptr;

                }

                delete aux;

                std::cout << "Elementul " << x << " a fost sters din lista." << std::endl;

                return;

            }

            prev = aux;

            aux = aux->next;

        } while (aux != head);

        std::cout << "Elementul " << x << " nu a fost gasit in lista." << std::endl;

    }

};

int main() {

    lista lista;

    lista.adaugare\_nod\_final(4);

    lista.adaugare\_nod\_final(22);

    lista.adaugare\_nod\_final(14);

    lista.afisare();

    lista.adaugare\_nod\_inceput(3);

    lista.afisare();

    lista.cautare(4);

    lista.cautare(11);

    lista.stergere(22);

    lista.afisare();

    return 0;

}

Lab 3

Cerinta 1

#include <iostream>

#include <string>

struct Nod {

    char val;

    Nod \*next;

    Nod(char x) {

        val = x;

        next = nullptr;

    }

    // ~Nod(){

    //     delete next;

    // }

};

struct Stack{

    Nod \*top;

    Stack() {

        top = nullptr;

    }

    ~Stack() {

        while (top != nullptr) {

            Nod \*aux = top;

            top = top->next;

            delete aux;

        }

    }

    void push(char val) {

        Nod \*nod = new Nod(val);

        nod->next = top;

        top = nod;

    }

    char pop() {

        char ch = top->val;

        if (top == nullptr) {

            std::cout << "Stiva este goala." << std::endl;

        }

        Nod \*aux = top;

        top = top->next;

        delete aux;

        return ch;

    }

    void afisare() {

        Nod \*aux = top;

        while (aux != nullptr) {

            std::cout << aux->val << " ";

            aux = aux->next;

        }

        std::cout << std::endl;

    }

    bool empty(){

        if (top == nullptr)

            return true;

        return false;

    }

};

void validare(std::string s) {

    bool ok;

    Stack stack;

    for (char c : s) {

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

            stack.push(c);

        else {

            if (stack.empty() ||

                (c == ')' && stack.top->val != '(') ||

                (c == '}' && stack.top->val != '{') ||

                (c == ']' && stack.top->val != '[')) {

                ok = false;

                break;

            }

            stack.pop();

        }

    }

    //Varificare daca au ramas paranteze neinchise

    ok = stack.empty();

    if(ok == true)

        std::cout<<"Expresie corecta!"<<std::endl;

    else

        std::cout<<"Expresie gresita!"<<std::endl;

    return;

}

std::string citire\_input(){

    std::string aux;

    std::cout << "Introduce expresia de paranteze: ";

    std::cin >> aux;

    return aux;

}

int main() {

    std::string expresie;

    expresie = citire\_input();

    validare(expresie);

    return 0;

}

Cerinta 2

#include <iostream>

#include <vector>

using namespace std;

struct Nod {

    int val;

    Nod \*next;

    Nod(int x) {

        val = x;

        next = nullptr;

    }

};

struct Stack{

    Nod \*top;

    Stack() {

        top = nullptr;

    }

    ~Stack() {

        while (top != nullptr) {

            Nod \*aux = top;

            top = top->next;

            delete aux;

        }

    }

    void push(int val) {

        Nod \*nod = new Nod(val);

        nod->next = top;

        top = nod;

    }

    int pop() {

        if (top == nullptr) {

            std::cout << "Stiva este goala." << std::endl;

            return -1; // Returnăm -1 pentru a indica o eroare

        }

        int ch = top->val;

        Nod \*aux = top;

        top = top->next;

        delete aux;

        return ch;

    }

    int acces\_top() {

        if (top == nullptr) {

            std::cout << "Stiva este goala." << std::endl;

            return -1;

        }

        return top->val;

    }

    bool empty() {

        return top == nullptr;

    }

};

std::vector<int> gasesteElemente(const std::vector<int>& v) {

    int n = v.size();

    std::vector<int> rez(n, -1);

    Stack st;

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

        while (!st.empty() && v[st.acces\_top()] <= v[i]) {

            st.pop();

        }

        if (!st.empty()) {

            rez[i] = v[st.acces\_top()];

        }

        st.push(i);

    }

    return rez;

}

vector<int> citire\_input(){

    vector<int> aux;

    int nr, x;

    cout<<"\nLungime vector: ";

    cin>>nr;

    cout<<"\nIntroduce elementele vectorului: \n";

    for(int i=1; i<=nr; i++){

        cin>>x;

        aux.push\_back(x);

    }

    return aux;

}

int main() {

    std::vector<int> v = citire\_input();

    v = gasesteElemente(v);

    std::cout << "Elementele urmatoare mai mari sunt: \n";

    for (int val : v) {

        std::cout << val << " ";

    }

    std::cout << std::endl;

    return 0;

}

Lab 4

#include <iostream>

#include <fstream>

#include <map>

#include <vector>

#include <algorithm>

std::map<char, std::vector<std::string>> citire\_cuvinte() {

    std::map<char, std::vector<std::string>> h;

    std::string cuvant;

    int nr;

    std::cout<<"Introduce numarul de cuvinte: ";

    std::cin>>nr;

    std::cout<<"Introduce cuvintele: ";

    for(int i=1; i<=nr; i++){

        std::cin>>cuvant;

        char prima\_litera = std::tolower(cuvant[0]);

        h[prima\_litera].push\_back(cuvant);

    }

    for (auto& pair : h) {

        std::sort(pair.second.begin(), pair.second.end());

    }

    return h;

}

void afisare\_dictionar(const std::map<char, std::vector<std::string>>& h) {

    for (auto pair : h) {

        std::cout << pair.first << ": ";

        for (auto cuvant : pair.second) {

            std::cout << cuvant << " ";

        }

        std::cout << std::endl;

    }

}

int main() {

    std::map<char, std::vector<std::string>> h = citire\_cuvinte();

    afisare\_dictionar(h);

    return 0;

}

Lab 5

Cerinta 1

#include <iostream>

#include <vector>

struct Node {

    int value;

    Node\* left;

    Node\* right;

    Node(int x) : value(x), left(nullptr), right(nullptr) {}

};

Node\* sortedArrayToBST(std::vector<int> v, int left, int right)

{

    if (left > right)

        return nullptr;

    int mid = (left + right) / 2;

    Node\* root = new Node(v[mid]);

    root->left = sortedArrayToBST(v, left, mid - 1);

    root->right = sortedArrayToBST(v, mid + 1, right);

    return root;

}

void preOrderTraversal(Node\* node) {

    if (node == nullptr) {

        return;

    }

    std::cout << node->value << " ";

    preOrderTraversal(node->left);

    preOrderTraversal(node->right);

}

void deleteTree(Node\* node) {

    if (node == nullptr)

        return;

    deleteTree(node->left);

    deleteTree(node->right);

    delete node;

}

int main() {

    std::vector<int> sortedArray = {1, 2, 3, 4, 5, 6, 7};

    int n = sortedArray.size();

    if (n == 0)

        std::cout<<"Vector invalid.";

    else{

        Node\* root = sortedArrayToBST(sortedArray, 0, n-1);

        std::cout << "Arborele in pre-ordine: " << std::endl;

        preOrderTraversal(root);

        std::cout << std::endl;

        deleteTree(root);

    }

    return 0;

}

Cerinta 2

#include <iostream>

#include <vector>

struct Node {

    int value;

    Node\* left;

    Node\* right;

    Node(int x) : value(x), left(nullptr), right(nullptr) {}

};

void preorderBst(Node\* root, std::vector<int>& v) {

    if (root == nullptr) {

        return;

    }

    v.push\_back(root->value);

    preorderBst(root->left, v);

    preorderBst(root->right, v);

}

void preorderPrintBst(Node\* root){

    if (root == nullptr) {

        return;

    }

    std::cout << root->value << " ";

    preorderPrintBst(root->left);

    preorderPrintBst(root->right);

}

std::vector<int> mergeArrays(std::vector<int> v1, std::vector<int> v2) {

    std::vector<int> rez;

    int i = 0, j = 0;

    while (i < v1.size() && j < v2.size()) {

        if (v1[i] < v2[j]) {

            rez.push\_back(v1[i]);

            i++;

        } else {

            rez.push\_back(v2[j]);

            j++;

        }

    }

    while (i < v1.size()) {

        rez.push\_back(v1[i]);

        i++;

    }

    while (j < v2.size()) {

        rez.push\_back(v2[j]);

        j++;

    }

    return rez;

}

Node\* sortedArrayToBST(std::vector<int> v, int left, int right)

{

    if (left > right)

        return nullptr;

    int mid = (left + right) / 2;

    Node\* root = new Node(v[mid]);

    root->left = sortedArrayToBST(v, left, mid - 1);

    root->right = sortedArrayToBST(v, mid + 1, right);

    return root;

}

Node\* mergeTwoBSTs(Node\* root1, Node\* root2) {

    std::vector<int> v1, v2;

    preorderBst(root1, v1);

    preorderBst(root2, v2);

    std::vector<int> mergedVector = mergeArrays(v1, v2);

    return sortedArrayToBST(mergedVector, 0, mergedVector.size()-1);

}

void deleteTree(Node\* node) {

    if (node == nullptr)

        return;

    deleteTree(node->left);

    deleteTree(node->right);

    delete node;

}

int main() {

    Node\* root1 = new Node(1);

    root1->left = new Node(0);

    root1->right = new Node(2);

    Node\* root2 = new Node(4);

    root2->left = new Node(3);

    root2->right = new Node(5);

    Node\* merged = mergeTwoBSTs(root1, root2);

    preorderPrintBst(merged);

    delete root1->left;

    delete root1->right;

    delete root1;

    delete root2->left;

    delete root2->right;

    delete root2;

    deleteTree(merged);

    return 0;

}

Lab 6

Kruskal

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

using namespace std;

struct Muchie{

    int x, y, cost;

};

struct Graph{

    int nr\_noduri, nr\_muchii;

    vector<Muchie> muchii;

};

Graph citire\_input() {

    ifstream in("input\_kruskal.txt");

    Graph graph;

    in >> graph.nr\_noduri >> graph.nr\_muchii;

    for (int i = 0; i < graph.nr\_muchii; i++) {

            Muchie muchie;

            in >> muchie.x >> muchie.y >> muchie.cost;

            graph.muchii.push\_back(muchie);

    }

    in.close();

    return graph;

}

//Functia gaseste parintele nodului i

int findParent(vector<int>& parent, int i) {

    if (parent[i] != i) {

        parent[i] = findParent(parent, parent[i]);

    }

    return parent[i];

}

void reuniune(vector<int>& parent, vector<int>& rang, int x, int y) {

    int rootX = findParent(parent, x);

    int rootY = findParent(parent, y);

    if (rang[rootX] < rang[rootY]) {

        parent[rootX] = rootY;

    } else if (rang[rootX] > rang[rootY]) {

        parent[rootY] = rootX;

    } else {

        parent[rootY] = rootX;

        rang[rootX]++;

    }

}

vector<Muchie> kruskal(Graph& graph) {

    vector<Muchie> rez;

    //Sortare muchii crescator dupa cost

    sort(graph.muchii.begin(), graph.muchii.end(), [](Muchie a, Muchie b) {

        return a.cost < b.cost;

    });

    vector<int> parent(graph.nr\_noduri + 1);

    vector<int> rang(graph.nr\_noduri + 1, 0);

    //Initializare vector de parinti

    for (int i = 1; i <= graph.nr\_noduri; i++) {

        parent[i] = i;

    }

    for (auto muchie : graph.muchii) {

        int x = findParent(parent, muchie.x);

        int y = findParent(parent, muchie.y);

        if (x != y) {

            rez.push\_back(muchie);

            reuniune(parent, rang, x, y);

        }

    }

    return rez;

}

int main(){

    Graph graph;

    graph = citire\_input();

    int cost = 0;

    vector<Muchie> rez = kruskal(graph);

    for (auto m : rez){

        cost += m.cost;

        cout<< m.x << " " << m.y << endl;

    }

    cout<<cost;

    return 0;

}

Prim

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <climits>

using namespace std;

struct Muchie {

    int x, y, cost;

    Muchie(int \_x, int \_y, int \_cost) : x(\_x), y(\_y), cost(\_cost){}

};

struct Graph {

    int nr\_noduri, nr\_muchii;

    unordered\_map<int, vector<pair<int, int>>> adiacenta;

    Graph(int n = 0, int m = 0) : nr\_noduri(n), nr\_muchii(m) {}

};

Graph citire\_input() {

    ifstream in("input\_kruskal.txt");

    Graph graph;

    in >> graph.nr\_noduri >> graph.nr\_muchii;

    for (int i = 0; i < graph.nr\_muchii; i++) {

        int x, y, cost;

        in >> x >> y >> cost;

        graph.adiacenta[x].push\_back({y, cost});

        graph.adiacenta[y].push\_back({x, cost});

    }

    in.close();

    return graph;

}

vector<Muchie> prim(Graph& graph) {

    vector<int> cost(graph.nr\_noduri + 1, INT\_MAX);

    vector<int> parent(graph.nr\_noduri + 1, -1);

    vector<bool> inMST(graph.nr\_noduri + 1, false);

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    int startNode = 1;

    pq.push({0, startNode});

    cost[startNode] = 0;

    while (!pq.empty()) {

        int nod = pq.top().second;

        pq.pop();

        if (inMST[nod]) continue;

        inMST[nod] = true;

        for (auto vecin : graph.adiacenta[nod]) {

            int y = vecin.first;

            int cost\_value = vecin.second;

            if (!inMST[y] && cost\_value < cost[y]) {

                cost[y] = cost\_value;

                pq.push({cost[y], y});

                parent[y] = nod;

            }

        }

    }

    vector<Muchie> rezultat;

    for (int i = 2; i <= graph.nr\_noduri; ++i) {

        if (parent[i] != -1) {

            rezultat.push\_back(Muchie(parent[i], i, cost[i]));

        }

    }

    return rezultat;

}

int main() {

    Graph graph = citire\_input();

    vector<Muchie> rez = prim(graph);

    int costTotal = 0;

    for (auto muchie : rez) {

        costTotal += muchie.cost;

        cout << muchie.x << " " << muchie.y << " " << muchie.cost << endl;

    }

    cout << "Cost total: " << costTotal << endl;

    return 0;

}