

1.1.

Passul intermediar între selection sort și heap sort este **interschimbarea**, în sensul că după ce se face interschimbarea vor exista o **parte sortată** (la selection sort prin selecția minimului va fi plasată în partea de început, în timp ce la heap sort prin max-heap nu va mai fi în heap, însemnând că va fi sortată descreșcător partea sortată). Pentru selection sort la fiecare pas se determină minimul curent, iar o dată ce se face o interschimbare este plasat pe poziția corespunzătoare de la început, în timp ce pentru heap sort, se va interschimba la un moment dat elementul curent cu primul element $v[1]$, însemnând că partea sortată este scoasă din heap și vor pune acele elemente de la dreapta spre stânga, de la mare la mic ($v[1]$ este maximul).

1.2.

Primul pas este de a construi un max-heap (min-heap), care va însemna parcurgerea de la $n/2$ la 1 a tuturor val din vector iar pentru fiecare se aplică heapify (de la $n/2+1$ până la n sunt frunze). Heapify va face "float down" în momentul în care rădăcina nu este maximul (când se compară cu copiii săi) și se aplică recursiv, în adâncime pt a asigura că este respectată condiția $v[\text{parinte}] > v[\text{copil}]$. După ce s-a făcut build heap, se parcurg nodurile în ordine inversă din vector și se face o interschimbare dintre $v[1]$ și el curent despre care se știe că este maximul de fiecare dată și se aplică încă o dată heapify pt a reface structura de heap dacă este stricată.

2.1.

```
void delete_head()
{
    node *aux = head;
    head = head->next; // noul cap e urm elem din lista
    aux->prev->next = head; // ultimul element va pointa catre elem aflat dupa
    cap
    head->prev = aux->prev; // conectez campul de prev de la noul head la ultimul
    element al listei
    delete aux;
}
```

2.2. rezolvare:

```
void delete_head()
{
    node *aux = head;
    if(head != nullptr)
    {
        head = head->next;
    }
    delete aux;
}
```

```
,
void delete_head()
{
    node *aux = head;
    if(head != nullptr)
    {
        head = head->next;
    }
    delete aux;
}
};

> int main() {
    SimplyLinkedList list;
    int data, x, i;

    while(data!=0)
    {
        cin>>data;
        list.insert_node_end(data);
    }
    list.delete_head();
    list.display();
}
```

Project
sll C:\Users\x\CLionProjects\sll
cmake-build-debug
CMakeLists.txt
main.cpp
External Libraries
Scratches and Consoles

CMakeLists.txt

main.cpp

258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
main

main.cpp

```
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
main
```

Run sll x

↑
↓
≡

```
C:\Users\x\CLionProjects\sll\cmake-build-debug\sll.exe  
2 5 8 34 23 67 2 0  
5 -> 8 -> 34 -> 23 -> 67 -> 2 -> 0 -> NULL  
  
Process finished with exit code 0
```

The screenshot shows the CLion IDE interface. The Project view on the left shows the project 'sll' with files 'CMakeLists.txt' and 'main.cpp'. The main.cpp file is open, showing a stack implementation using a vector. The code includes comments in Romanian. The Run window at the bottom shows the output of the program, indicating that the expression '()' is correct and the stack is empty.

```

102 stack<int> current_stack, aux_stack; // pe current_stack se a
103 string expr;
104 cout<<"Expresia matematica care foloseste paranteze (), {}, [] este:
105 cin>>expr;
106 /*for(unsigned long long i=0; i<expr.length(); i++)
107 {
108     current_stack.push(expr[i]);
109 }
110 current_stack.display();*/
111 bool ok = true; // presupun ca expresia e scrisa corect
112 for(char c: expr)
113 {
114     if(c=='(' || c=='[' || c=='{')// introduc pe stiva (
115     {
116         current_stack.push( data: c);
117     } else if (c==')' || c==']' || c=='}')
118     {
119         if(current_stack.empty())
120             ok = false;
121         else
122             current_stack.pop();
123     }
124 }
125 if(ok)
126     cout<<"Expresia este corecta";
127 else
128     cout<<"Expresia nu este corecta";
129 }
```

Run window output:

```

C:\Users\x\CLionProjects\sll\cmake-build-debug\sll.exe
Expresia matematica care foloseste paranteze (), {}, [] este:
()
Expresia nu are corect imbricate parantezele: ()
Valoarea elementului care a fost scos de pe stiva: {
Valoarea elementului care a fost scos de pe stiva: {
Ce a ramas pe stiva current_stack:
| ( -> Ce a ramas pe stiva aux_stack:
| ) -> Expresia nu e scrisa corect
Process finished with exit code 0

```

3.1.

3.2. Ideea de baza este sa folosesc o stiva pentru a retine elementele ale caror prim mai mare element inca nu a fost gasit si cat timp acea stiva nu este goala si valoarea din vf stivei e una mai mare decat ce a elemntului curent, inseamna ca se va retine o pereche de elemente de tipul (el_curent, el_mai_mare) (am ales sa nu retin pozitiile, ci valorile in sine (pentru usurinta procesarii de la final), in doua cozi). Cum functioneaza stiva: fiecare element din vector trebuie adaugat pe stiva, iar in momentul in care un element curent din vector e mai mare decat valoarea din vf stivei, se realizeaza pop si se continua verificarea conditiei. Daca raman elemente pe stack, inseamna ca nu le-a fost gasit primul cel mai mare element, pentru ca se afla la sf vectorului.

4.1.

6. Iesire meniu

Tastati varianta aleasa:5

Cuvintele curente ale dictionarului sunt:

Litera A: ani -> Are -> Ana ->

Nu exista niciun cuvânt cu litera B

Litera C: carte -> caiet -> cuminte ->

Nu exista niciun cuvânt cu litera D

Nu exista niciun cuvânt cu litera E

Nu exista niciun cuvânt cu litera F

Nu exista niciun cuvânt cu litera G

Nu exista niciun cuvânt cu litera H

Litera I: inteligent -> inima -> imagine ->

Nu exista niciun cuvânt cu litera J

Nu exista niciun cuvânt cu litera K

Nu exista niciun cuvânt cu litera L

Nu exista niciun cuvânt cu litera M

Nu exista niciun cuvânt cu litera N

Nu exista niciun cuvânt cu litera O

Nu exista niciun cuvânt cu litera P

Nu exista niciun cuvânt cu litera Q

Nu exista niciun cuvânt cu litera R

Nu exista niciun cuvânt cu litera S

Nu exista niciun cuvânt cu litera T

Nu exista niciun cuvânt cu litera U

Nu exista niciun cuvânt cu litera V

Nu exista niciun cuvânt cu litera W

Nu exista niciun cuvânt cu litera X

Nu exista niciun cuvânt cu litera Y

Nu exista niciun cuvânt cu litera Z

5.1.

Ideea este sa existe o diferenta maxima de inaltime intre subarborii din stanga si dreapta care sa nu depaseasca 1. Pentru ca vectorul este sortat crescator deja, se va crea o functie recursive de divide et impera care va construi in sine arborele bst, in care array-ul se injumatateste, radacina arborelui fiind elementul din mijloc, apoi se parcurge subarray-urile din stanga si dreapta recursiv. Pentru verificare se parcurge in inordine, despre care stim ca va afisa subarborele cu valorile sortate.

5.2.

Interclasarea a doi arbori echilibrati va insemna sa se parcurga in inordine fiecare arbore si sa se retina elementele intr-un array comun, apoi construirea unui arbore balansat, folosind o interclasarea vectorilor, care e cunoscuta. Se va reveni in final la structura de arbore folosind alg de la problema anterioara.

6.1.

Diferenta se refera la faptul ca un mst e construit pe baza unui drum parcurs la Prim, celalalt e construit in functie de muchii sortate crescator dupa cost la Kruskal. La Prim se construiesc o padure, un drum posibil prin care s-ar obtine costul minim. Mai exact, se extrage nodul cu un cost minim de la pozitia curenta si se adauga in setul de minimum spanning tree, verificandu-se vecinii lui dpdv al costului, daca au mai fost parcursi sau nu (nu trebuie sa se repete) si sa nu existe cicluri. Kruskal sorteaza toate muchiile grafului in ordine crescatoare a greutatilor. Se fac mai multi arbori partiali, evitandu-se ciclurile, si se tine evidenta printr-un dsu. se iau muchiile in ordine sortata si se adauga intr-un arbore partial corespunzator/ curent (daca nu creaza ciclu).

6.2. Prim se porneste de la un nod de start, apoi se verifica ce muchii poate forma (cele care nu au fost declarate drept infinity, int_max), se alege cea de cost minim si care nu va forma niciun ciclu. Se continua parcurgerea de la nodul adiacent celui anterior si se realizeaza aceeasi pasi. Un nod "valid" se adauga in setul de la mst si se verifica vecinii.

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <ctime>

using namespace std;

void display_array(vector<int> v)
{
    for(int i = 0; i < v.size(); i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;
}

void display_subarray(const vector<int> &v, int start, int end)
{
    for(int i = start; i <= end; i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;
}

// bubble sort
void bubble_sort(vector<int> &v)
{
    int length = v.size();
    bool swapped;
    int pass = 0;
    do{
        swapped = false;
        pass++; // nr de parcurgeri complete ale vectorului
        cout<<"Parcurgerea "<<pass<<":"<<endl;
        for(int i = 0; i < length-1; i++)
        {
            if(v[i] > v[i+1])
            {
                swap(v[i], v[i+1]);
                swapped = true;
            }
        }
    }
}
```

```

        }
        display_array(v);
    }
    length--; // se reduce lungimea pt ca fiecare maxim corespunzator
fiecarei treceri complete prin vector va fi plasat la final (e iredundant sa
parcurgem acea secventa deja sortata)
    } while(swapped);
}

// selection sort
void selection_sort_min(vector<int> &v) // prin selectia minimului
{
    int pass = 0;
    for(int i = 0; i < v.size() - 1; i++)
    {
        pass++;
        cout<<"Parcurerea "<<pass<<":"<<endl;
        int p = i; // pivotul e initializat cu indexul curent deoarece elementele
din vector de un index mai mic decat cel curent au fost deja sortate
        for(int j = i+1; j < v.size(); j++)
        {
            if(v[j] < v[p])
            {
                p = j; // pivotul e de fapt indexul celui mai mic element din
vector, care se plaseaza pe prima pozitie
            }
        }
        if(p != i) // daca am gasit minimul curent din vectorul ramas nesortat,
interschimbam val curenta cu cea a minimului
        {
            swap(v[i], v[p]);
        }
        display_array(v);
    }
}

// insertion sort
void insertion_sort(vector<int> &v)
{
    int pass = 0;
    for(int i = 1; i < v.size(); i++)
    {
        pass++;
        cout<<"Parcurerea "<<pass<<":"<<endl;
        int poz = i;
        while(poz > 0 && v[poz-1] > v[poz]) // v[poz] ar putea fi inlocuita cu o
variabila currentvalue pt readability
        {
            swap(v[poz], v[poz-1]);
            poz--;
            display_array(v);
        }
    }
}

// merge sort
void merge(vector<int> &v, int start, int mid, int end)

```

```

{
    vector<int> temp(end - start + 1);
    // i e indexul pentru subvectorul din stanga, iar j pentru cel din dreapta
    int i = start, j = mid+1;
    int k = 0; // index pentru temp (temporary vector)

    // strict pt afisare (cum arata subvectorii inainte de merging (dupa ce sunt
    sortati, dar inca "separati"))
    cout<<"Merging/ Impera: "<<endl;
    display_subarray(v, start, mid);
    display_subarray(v, mid+1, end);

    // imbinare vectori pana cand se ajunge la sfarsitul oricaruia
    while(i <= mid && j <= end)
    {
        // plasare element cel mai mic dintre cei doi subvectori in temp[]
        if(v[i] < v[j])
        {
            temp[k++] = v[i++];
            // incrementare indecsi pt testarea urmatoarelor valori (indexul care
            ramane neincrementat coresp val care
            // nu a fost pusa in temp[] la pasul curent, dar care va fi pusa la
            un mom dat)

            // SAU
            // temp[k] = v[i];
            // k++; i++;
        } else
        {
            temp[k++] = v[j++];
        }
    }

    // daca ne aflam la sfarsitul subvectorului din stanga, inseamna ca vom
    completa temp cu val ramase in subvectorul din dreapta
    if(i > mid)
    {
        while(j <= end)
        {
            temp[k++] = v[j++];
        }
    } else // sf subvectorului din dreapta => completare val din subvectorul din
    stanga
    {
        while(i <= mid)
        {
            temp[k++] = v[i++];
        }
    }

    // suprascrierea vectorului v cu elem din temp
    for(int index = 0; index <= end - start; index++)
    {
        v[index + start] = temp[index];
    }

    // vectorul merged

```



```

    display_subarray(v, start, end);
}

void merge_sort(vector<int> &v, int start, int end) // sortarea in sine a
subvectorilor - divide et impera
{
    if(start >= end) // caz de baza
    {
        return;
    } else // if(start < end)
    {
        int mid = (start+end) / 2;
        // partea de splitting/ divide, apeluri recursive pt formarea de
subvectori

        // strict pt afisare:
        cout<<"Splitting/ Divide: "<<endl;
        display_subarray(v, start, mid);
        display_subarray(v, mid+1, end);

        merge_sort(v, start, mid); // sortare prima jumatate a vectorului, vazut
ca subvectorul din stanga
        merge_sort(v, mid + 1, end); // sortare a doua jumatate, subvector
dreapta

        // partea de merging/ interclasare a subvectorilor deja sortati
        merge(v, start, mid, end);
    }
}

// quick sort
// elementele mai mari sau egale cu pivotul vor fi mutate in dreapta vectorului,
in timp ce celelalte vor fi mutate in stanga
int partition_middle_pivot(vector<int> &v, int start, int end) // stabilim ca val
din mijl vectorului este pozitia initiala a pivotului (se alege aleatoriu in
general SAU ultima val SAU prima val)
{
    int pivot = v[(start+end)/2], i = start, j = end;

    // strict pt afisare
    cout<<"Inainte de partitionare:"<<endl;
    display_subarray(v, start, end);
    cout<<"Pivot:"<<endl<<pivot<<endl;

    while(i <= j) // se opreste cand i > j
    {
        while(v[j] > pivot) // gasire element din dreapta care ar trebui sa fie
in partea stanga
        {
            j--;
            // prima aparitie a unei val mai mici decat pivotul
        }
        while(v[i] < pivot) // gasire element din stanga care ar trebui sa fie in
partea dreapta
        {
            i++; // indicele i este mutat spre dreapta pana cand se gaseste un
element cu o valoare mai mare decat cea a pivotului

```

```

        // prima aparitie a unei val mai mari decat pivotul
    }
    if(i <= j)
    {
        swap(v[i], v[j]);
        // urm pozitii de verificat
        i++;
        j--;

        // strict pt afisare
        cout<<"Dupa partitionare:"<<endl;
        display_subarray(v, start, end);
    }
}
return i;
}

void quick_sort(vector<int> &v, int start, int end)
{
    if(start < end)
    {
        int pivot_index = partition_middle_pivot(v, start, end); // pivot_index e
        // indexul de partitionare
        // apeluri recursive pentru a testa valorile aflate pe poz de dinaintea
        // pivotului si dupa el
        quick_sort(v, start, pivot_index - 1); // sortare subvector din stanga
        quick_sort(v, pivot_index, end); // sortare subvector din dreapta
    }
}

// heap sort
// length - lungime vector v
void display_heap(vector<int> v, int length)
{
    for(int i = 0; i < length/2; i++)
    {
        cout<<"Nod parinte: "<<v[i]<<"\t";
        if(2*i+1 < length)
        {
            cout<<"Copil stanga nod"<<i<<">: " <<v[2*i+1]<<"\t";
        }
        if(2*i+2 < length)
        {
            cout<<"Copil dreapta nod"<<i<<">: " <<v[2*i+2]<<"\t";
        }
        cout<<endl;
    }
}

void heapify(vector<int> &v, int length, int i) // max_heapify()
{
    // gasire elem maxim intre radacina, copil stang si drept
    int largest = i; // presupunem ca rad e maxima
    int left = 2*i + 1; // stanga
    int right = 2*i + 2; // dreapta

    if(left < length && v[left] > v[largest])

```

```

    {
        largest = left;
    }

    if(right < length && v[right] > v[largest])
    {
        largest = right;
    }

    // swap + heapify, daca radacina nu e cea mai mare val
    if(largest != i)
    {
        swap(v[i], v[largest]); // val din radacina se interschimba cu copilul de
        valoare maxima (stang sau drept)
        heapify(v, length, largest); // recursiv pt fiecare nivel (in adancime)
    }

    // strict pt afisare
    cout<<"Parcursiere:"<<endl;
    display_heap(v, length);
}

void build_heap(vector<int> &v, int length) // curs - buildMaxHeap()
{
    for(int i = length/2-1; i >= 0; i--)
    {
        heapify(v, length, i); // aplicam heapify pt fiecare nod incepand cu
        penultimul nivel (care in practica are copii (aflati pe ultimul nivel))
    }
}

void heap_sort(vector<int> &v, int length)
{
    build_heap(v, length);
    for(int i = length-1; i >= 0; i--)
    {
        swap(v[0], v[i]); // interschimbarea elementului de la pasul curent cu
        primul (radacina heap-ului format), insemnand ca acum maximul curent se afla la
        final

        // v[i] e asadar un element sortat, plasat acum pe poz lui
        corespunzatoare, eliminat din heap

        // se aplica heapify pt radacina (care acum e v[i]) pt a obtine din nou
        cel mai mare element positionat la radacina heap-ului
        heapify(v, i, 0); // se reduce pt fiecare pas lungimea heap-ului din
        moment ce unele elemente sunt scoase din heap, ele fiind sortate
    }
}

int main()
{
    vector<int> a;
    /*const int vec_length = 20; // nr maxim de elemente din vector
    const int val_max = 100; // elem din vector sa aiba valori cuprinse intre 0
    si 99 (ne vom folosi de restul impartirii la 100)
    srand(time(nullptr)); // un seed pt un generator de numere
    for(int i = 0; i <= vec_length; i++)

```

```

{
    a.push_back(rand() % val_max);
}*/
int vec_length;
cout<<"Introduceti nr de elemente pentru vector:"<<endl; cin>>vec_length;
for(int i = 0; i < vec_length; i++)
{
    int val;
    cin>>val;
    a.push_back(val);
}

cout<<"Inainte de sortare:"<<endl;
display_array(a);

int choice;
cout<<"1. Selection Sort\n2. Bubble Sort\n3. Insertion Sort\n4. Merge
Sort\n5. Quick Sort\n6. Heap Sort\nTastati varianta de sortare aleasa: ";
cin>>choice;

switch(choice)
{
    case 1:
        cout<<"Selection Sort:"<<endl;
        selection_sort_min(a);
        break;
    case 2:
        cout<<"Bubble Sort:"<<endl;
        bubble_sort(a);
        break;
    case 3:
        cout<<"Insertion Sort:"<<endl;
        insertion_sort(a);
        break;
    case 4:
        cout<<"Merge Sort:"<<endl;
        merge_sort(a, 0, a.size() - 1);
        break;
    case 5:
        cout<<"Quick Sort:"<<endl;
        quick_sort(a, 0, a.size() - 1);
        break;
    case 6:
        cout<<"Heap Sort:"<<endl;
        heap_sort(a, a.size());
        break;
    default:
        cout<<"Invalid"<<endl;
        return 1;
}
cout<<endl<<"Final:"<<endl;
display_array(a);

return 0;
}
#include <iostream>

```

```

using namespace std;

struct node{
    // o lista simplu inlantuita nu va avea campul prev, deci nu declar node
    *prev
    int val; // value
    node *next;
};

class SimplyLinkedList{ // in general
private:
    node *head; // capul
public:
    SimplyLinkedList() // constructor fara parametri
    {
        head = nullptr;
    }
    node *get_head()
    {
        return head;
    }
    node *get_nth_node(int i)
    {
        node *curr = head;
        int count = 0;
        while (curr != nullptr)
        {
            if(count == i)
            {
                return curr;
            }
            count++;
            curr = curr->next;
        }
    }
    node *create_node(int data)
    {
        // se cunoaste numai val curenta si presupunem ca o conexiune cu un nod
        anterior exista deja
        node *aux = new node; // declararea unui nod si alocarea memoriei
        aux->val = data;
        aux->next = nullptr; // adresa urmatorului camp este inca necunoscuta,
        din moment ce se insereaza un nod la finalul listei
        return aux; // returnare nod aux creat
    }
    void create_list(int n)
    {
        int x;
        for(int i = 0; i < n; i++)
        {
            cout<<"Valoarea nodului curent: "; cin>>x;
            node *aux = create_node(x);
            if(head == nullptr) // daca lista e vida, capul va reprezenta primul
            element introdus
            {
                head = aux;
            } else // daca exista cel putin un element in lista, inseamna ca se

```

modifica campul next al penultimului element din lista, elementul curent devenind ultimul la fiecare iteratie

```
{
    node *curr = head;
    while(curr->next != nullptr) // parcurgerea listei de la inceput
pana la ultimul element din lista
    {
        curr = curr->next;
    }
    curr->next = aux;
}

}

void display()
{
    if(head == nullptr)
    {
        cout<<"Lista e vida"<<endl;
    } else
    {
        node *curr = head;
        while(curr != nullptr)
        {
            cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
            curr = curr->next;
            if(curr == nullptr) // partea de tail a listei
            {
                cout<<"NULL"<<endl;
            }
        }
    }
}

void display_reversed(node *curr)
{
    if(curr == nullptr)
    {
        return;
    } else
    {
        display_reversed(curr->next);
        cout<<curr->val<<" -> ";
    }
}

void insert_node_end(int data) // similar cu implementarea pt create_list()
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
    } else
    {
        node *curr = head;
        while(curr->next != nullptr)
        {
            curr = curr->next;
        }
        curr->next = aux;
    }
}
```

```

    }
}
void insert_node_front(int data)
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
    } else
    {
        aux->next = head; // adresa campului urmator e head-ul anterior
        head = aux; // noul head e nodul curent
    }
}
void insert_node_after(node *prev_node, int data) // inserarea unui nod dupa
nodul prev_node
{
    if(prev_node == nullptr) // trebuie sa existe ca sa fie inserat un alt
nod dupa el
    {
        return;
    }
    node *aux = new node;
    aux->val = data;
    aux->next = prev_node->next; // campul next al lui aux e acelasi cu cel
al lui prev_node
    prev_node->next = aux; // noul nod e inserat dupa prev_node
}
node *search_node(int data)
{
    node *curr = head; // cautarea iterativa incepe din capul listei
    while(curr != nullptr)
    {
        if(curr->val == data)
        {
            return curr;
        }
        curr = curr->next; // urmatorul nod
    }
    return nullptr; // nodul cu valoarea data nu e gasit
}
void remove_node(int data) // ar fi mers si daca stiam adresa lui del_node
(minus linia 135); rescriere sub forma void remove_node(node *del_node)
{
    node *del_node = search_node(data); // cautare nod bazata pe gasirea
pozitiei in care se afla in lista
    node *curr = head; // lista va fi parcursa incepand cu head-ul
    if(del_node == head) // daca nodul care trebuie sters e capul listei,
noul cap al listei e elementul care urmeaza dupa head
    {
        head = head->next; // se muta doar capul listei, restul elementelor
nu sunt afectate
    } else
    {
        if(del_node != nullptr) // daca nodul care trebuie sters exista in
lista, adica functia search node nu returneaza nullptr, atunci se vor deplasa
restul elementelor

```

```

    {
        while(curr != nullptr) // parcurgere linked list pana la tail
        {
            if(curr->next == del_node)
            {
                // partea de unlinking
                curr->next = del_node->next; // in mom in care campul
                next al nodului curent pointeaza catre nodul care trebuie sters, se reconstituie
                legatura pentru a sari peste del_node
            }
            curr = curr->next; // deplasare
        }
    }
    delete del_node; // stergerea efectiva pt eliberarea zonei de memorie
}
node *max()
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *max_node = head;
    // de data asta curr e nodul care urmeaza dupa head, deoarece se parcurge
    lista folosind doua variabile care mereu isi incrementeaza pozitiile succesiv
    node *curr = head->next; // daca max_node e primul element din lista, vor
    fi verificate restul elementelor din lista incepand cu al doilea, adica head-
    >next
    while(curr != nullptr) // parcurgere lista pana la tail
    {
        if(max_node->val < curr->val)
        {
            max_node = curr;
        }
        curr = curr->next;
    }
    return max_node;
}
node *min() // similar cu functia max
{
    if(head == nullptr)
    {
        return nullptr;
    }
    node* min_node = head;
    node* curr = head->next;
    while(curr != nullptr)
    {
        if(min_node->val > curr->val)
        {
            min_node = curr;
        }
        curr = curr->next;
    }
    return min_node;
}

```



```

node *successor(int data)
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *curr = head;
    while(curr != nullptr)
    {
        if(curr->val == data)
        {
            if(curr->next == nullptr) // elementul cautat este tail-ul, deci
succesorul e nullptr
            {
                return nullptr;
            } else
            {
                return curr->next;
            }
        }
        curr = curr->next; // deplasare
    }
    return nullptr; // nodul cu val data nu a fost gasit
}

node *predecessor(int data)
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    if(head->val == data)
    {
        return nullptr;
    }
    node *curr = head;
    while(curr != nullptr)
    {
        if(curr->next->val == data)
        {
            return curr;
        }
        curr = curr->next; // deplasare
    }
    return nullptr; // nodul cu val data nu a fost gasit
}

};

int main() {
    SimplyLinkedList list;
    int choice, x, i;

    do{
        cout<<endl;
        cout<<"1. Crearea unei liste simplu inlantuite\n2. Inserarea unui nod la
sfarsitul listei\n3. Inserarea unui nod la inceputul listei\n4. Inserarea unui
nod dupa un alt nod existent din lista\n5. Cautarea unui nod in functie de
valoare\n6. Stergerea unui nod in functie de valoare\n7. Cautarea minimului si a

```

```

maximului din lista\n8. Afisarea elementelor curente din lista\n 9. Afisarea
inversa a elementelor din lista\n 10. Cautarea succesorului unui element\n 11.
Cautarea predecesorului unui element\n 12. Iesire meniu\n Tastati varianta
aleasa: ";
    cin>>choice;
    switch(choice)
    {
        case 1:
            int n;
            if(list.get_head() == nullptr) // daca lista nu a fost
initializata inca
            {
                cout<<"Nr. de noduri ale listei simplu inlantuite: "; cin>>n;
                list.create_list(n);
            } else
            {
                cout<<"Lista exista deja"<<endl;
            }
            break;
        case 2:
            cout<<"Valoarea nodului: "; cin>>x;
            list.insert_node_end(x);
            break;
        case 3:
            cout<<"Valoarea nodului: "; cin>>x;
            list.insert_node_front(x);
            break;
        case 4:
            cout<<"Valoarea nodului: "; cin>>x;
            cout<<endl<<"Pozitia nodului dupa care sa fie inserat: "; cin>>i;
            list.insert_node_after(list.get_nth_node(i), x);
            break;
        case 5:
            cout<<"Valoarea de cautat in lista: "; cin>>x;
            if(list.search_node(x) == nullptr)
            {
                cout<<"Valoarea nu a fost gasita in lista"<<endl;
            } else
            {
                cout<<"Valoarea a fost gasita in lista"<<endl;
            }
            break;
        case 6:
            cout<<"Valoarea de sters din lista: "; cin>>x;
            list.remove_node(x);
            break;
        case 7:
            cout<<"Valoarea minima din lista este: "<<list.min()->val<<endl;
            cout<<"Valoarea maxima din lista este: "<<list.max()->val<<endl;
            break;
        case 8:
            cout<<"Elementele curente ale listei simplu inlantuite
sunt:"<<endl;
            list.display();
            break;
        case 9:
            cout<<"Elementele curente ale listei simplu inlantuite in ordine

```

```

inversa sunt:"<<endl;
    list.display_reversed(list.get_head());
    break;
case 10:
    cout<<"Valoarea elementului al carui succesori trebuie returnat:
"; cin>>x;
    if(list.successor(x) != nullptr)
    {
        cout<<"Succesorul are valoarea "<<list.successor(x)-
>val<<endl;
    } else
    {
        cout<<"Elementul selectat nu are succesori"<<endl;
    }
    break;
case 11:
    cout<<"Valoarea elementului al carui predecesori trebuie returnat:
"; cin>>x;
    if(list.predecessor(x) != nullptr)
    {
        cout<<"Predecesorul are valoarea "<<list.predecessor(x)-
>val<<endl;
    } else
    {
        cout<<"Elementul selectat nu are predecesori"<<endl;
    }
    break;
case 12:
    break;
default:
    cout<<"Invalid"<<endl;
    return 1;
}
} while(choice != 12);

return 0;
}
#include <iostream>

using namespace std;

struct node{
    node *prev;
    int val;
    node *next;
};

class DoublyLinkedList{
private:
    node *head;
public:
    DoublyLinkedList()
    {
        head = nullptr;
    }
    node *get_head()
    {

```

```

        return head;
    }
    node *get_nth_node(int i)
    {
        node *curr = head;
        int count = 0;
        while (curr != nullptr)
        {
            if(count == i)
            {
                return curr;
            }
            count++;
            curr = curr->next;
        }
    }
    node *create_node(int data)
    {
        node *aux = new node; // declararea unui nod si alocarea memoriei
        aux->val = data;
        aux->prev = nullptr; // adresa campului prev e necunoscuta
        aux->next = nullptr; // adresa urmatorului camp e necunoscuta
        return aux; // returnare nod aux creat
    }
    void create_list(int n)
    {
        int x;
        for(int i = 0; i < n; i++)
        {
            cout<<"Valoarea nodului curent: "; cin>>x;
            node *aux = create_node(x);
            if(head == nullptr) // daca lista e vida, capul va reprezenta primul
                element introdus
            {
                head = aux;
            } else // daca exista cel putin un element in lista, inseamna ca se
                modifica campul next al penultimului element din lista, elementul curent devenind
                ultimul la fiecare iteratie
            {
                node *curr = head;
                while(curr->next != nullptr) // parcurgerea listei de la inceput
                    pana la ultimul element din lista
                {
                    curr = curr->next;
                }
                curr->next = aux;
                aux->prev = curr; // conexiune stanga
            }
        }
    }
    void display()
    {
        node *curr = head;
        if(head == nullptr)
        {
            cout<<"Lista e vida"<<endl;
        } else
    
```

```

    {
        while(curr != nullptr)
        {
            cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
            curr = curr->next;
            if(curr == nullptr) // partea de tail a listei
            {
                cout<<"NULL"<<endl;
            }
        }
    }
}

void display_reversed() // iterativ; implementarea recursiva de la simply
linked list a fost necesara din cauza lipsei campului prev
{
    node *curr = head;
    if(head == nullptr)
    {
        cout<<"Lista e vida"<<endl;
    } else
    {
        while(curr->next != nullptr) // deplasarea catre dreapta pana la tail
        {
            curr = curr->next;
        }
        while(curr != nullptr) // deplasare catre stanga pana la head
        {
            cout<<curr->val<<" -> ";
            curr = curr->prev;
        }
    }
}

void insert_node_end(int data) // similar cu implementarea pt create_list()
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
    } else
    {
        node *curr = head;
        while(curr->next != nullptr)
        {
            curr = curr->next;
        }
        curr->next = aux;
        aux->prev = curr; // conexiune stanga
    }
}

void insert_node_front(int data)
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
    } else
    {

```

```

        aux->next = head; // adresa campului urmator e head-ul anterior
        head->prev = aux; // conexiune stanga
        head = aux; // noul head e nodul curent
    }
}

void insert_node_after(node *prev_node, int data) // inserarea unui nod dupa
nodul prev_node
{
    if(prev_node == nullptr) // trebuie sa existe ca sa fie inserat un alt
nod dupa el
    {
        return;
    }
    node *aux = new node;
    aux->val = data;
    aux->prev = prev_node;
    aux->next = prev_node->next; // campul next al lui aux e acelasi cu cel
al lui prev_node
    prev_node->next = aux; // noul nod e inserat dupa prev_node
    if(aux->next != nullptr)
    {
        aux->next->prev = aux;
    }
}

node *search_node(int data) // nimic schimbat fata de simply linked list
{
    node *curr = head; // cautarea iterativa incepe din capul listei
    while(curr != nullptr)
    {
        if(curr->val == data)
        {
            return curr;
        }
        curr = curr->next;
    }
    return nullptr; // nodul cu valoarea data nu e gasit
}

void remove_node(int data)
{
    node *del_node = search_node(data); // cautare nod bazata pe gasirea
pozitiei in care se afla in lista
    node *curr = head; // lista va fi parcursa incepand cu head-ul
    if(del_node == head) // daca nodul care trebuie sters e capul listei,
noul cap al listei e elementul care urmeaza dupa head
    {
        head = head->next; // se muta doar capul listei, restul elementelor
nu sunt afectate
        if(head != nullptr)
        {
            head->prev = nullptr;
        }
    } else
    {
        if(del_node != nullptr) // daca nodul care trebuie sters exista in
lista, adica functia search node nu returneaza nullptr, atunci se vor deplasa
restul elementelor
        {

```

```

        del_node->prev->next = del_node->next; // unlinking pt campul
next al elementului aflat inaintea lui del_node
        if(del_node->next != nullptr) // stergerea unui nod din mijloc
        {
            del_node->next->prev = del_node->prev; // unlinking pt campul
prev al elementului curent care e inaintea val care trebuie stearsa
        }
    }
    delete del_node; // stergerea efectiva pt eliberarea zonei de memorie
}
/** inutila acea parcurgere folosind curr, dar functioneaza cel putin */
/*void remove_node(int data)
{
    node *del_node = search_node(data); // cautare nod bazata pe gasirea
pozitiei in care se afla in lista
    node *curr = head; // lista va fi parcursa incepand cu head-ul
    if(del_node == head) // daca nodul care trebuie sters e capul listei,
noul cap al listei e elementul care urmeaza dupa head
    {
        head = head->next; // se muta doar capul listei, restul elementelor
nu sunt afectate
        if(head != nullptr)
        {
            head->prev = nullptr;
        }
    } else
    {
        if(del_node != nullptr) // daca nodul care trebuie sters exista in
lista, adica functia search_node nu returneaza nullptr, atunci se vor deplasa
restul elementelor
        {
            while(curr != nullptr) // parcurgere linked list pana la tail
pentru a gasi nodul aflat imediat inaintea nodului care trebuie sters
            {
                if(curr->next == del_node)
                {
                    curr->next = del_node->next; // unlinking pt campul next
al elementului aflat inaintea lui del_node
                    if(del_node->next != nullptr) // stergerea unui nod din
mijloc
                    {
                        del_node->next->prev = curr; // unlinking pt campul
prev al elementului curent care e inaintea val care trebuie stearsa
                    }
                }
                curr = curr->next; // deplasare
            }
            if(del_node->next == nullptr) // stergerea tail-ului
            {
                del_node->prev->next = nullptr;
            }
        }
    }
    delete del_node; // stergerea efectiva pt eliberarea zonei de memorie
}*/
node *max()

```

```

{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *max_node = head;
    // de data asta curr e nodul care urmeaza dupa head, deoarece se parcurge
    lista folosind doua variabile care mereu isi incrementeaza pozitiile succesiv
    node *curr = head->next; // daca max_node e primul element din lista, vor
    fi verificate restul elementelor din lista incepand cu al doilea, adica head-
    >next
    while(curr != nullptr) // parcurgere lista pana la tail
    {
        if(max_node->val < curr->val)
        {
            max_node = curr;
        }
        curr = curr->next;
    }
    return max_node;
}
node *min() // similar cu functia max
{
    if(head == nullptr)
    {
        return nullptr;
    }
    node* min_node = head;
    node* curr = head->next;
    while(curr != nullptr)
    {
        if(min_node->val > curr->val)
        {
            min_node = curr;
        }
        curr = curr->next;
    }
    return min_node;
}
node *successor(int data)
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *curr = head;
    while(curr != nullptr)
    {
        if(curr->val == data)
        {
            if(curr->next == nullptr) // elementul cautat este tail-ul, deci
            succesorul e nullptr
            {
                return nullptr;
            } else
            {
                return curr->next;
            }
        }
    }
}

```



```

        }
    }
    curr = curr->next; // deplasare
}
return nullptr; // nodul cu val data nu a fost gasit
}
node *predecessor(int data)
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    if(head->val == data)
    {
        return nullptr;
    }
    node *curr = head;
    while(curr != nullptr)
    {
        if(curr->val == data)
        {
            return curr->prev;
        }
        curr = curr->next; // deplasare
    }
    return nullptr; // nodul cu val data nu a fost gasit
}
};

int main() {
    DoublyLinkedList list;
    int choice, x, i;

    do{
        cout<<endl;
        cout<<"1. Crearea unei liste dublu inlantuite\n2. Inserarea unui nod la sfarsitul listei\n3. Inserarea unui nod la inceputul listei\n4. Inserarea unui nod dupa un alt nod existent din lista\n5. Cautarea unui nod in functie de valoare\n6. Stergerea unui nod in functie de valoare\n7. Cautarea minimului si a maximului din lista\n8. Afisarea elementelor curente din lista\n 9. Afisarea inversa a elementelor din lista\n 10. Cautarea succesorului unui element\n 11. Cautarea predecesorului unui element\n 12. Iesire meniu\n Tastati varianta aleasa: ";
        cin>>choice;
        switch(choice)
        {
            case 1:
                int n;
                if(list.get_head() == nullptr) // daca lista nu a fost
initializata inca
                {
                    cout<<"Nr. de noduri ale listei dublu inlantuite: "; cin>>n;
                    list.create_list(n);
                } else
                {
                    cout<<"Lista exista deja"<<endl;
                }
            }
        }
    } while(choice != 12);
}

```

```

        break;
    case 2:
        cout<<"Valoarea nodului: "; cin>>x;
        list.insert_node_end(x);
        break;
    case 3:
        cout<<"Valoarea nodului: "; cin>>x;
        list.insert_node_front(x);
        break;
    case 4:
        cout<<"Valoarea nodului: "; cin>>x;
        cout<<endl<<"Pozitia nodului dupa care sa fie inserat: "; cin>>i;
        list.insert_node_after(list.get_nth_node(i), x);
        break;
    case 5:
        cout<<"Valoarea de cautat in lista: "; cin>>x;
        if(list.search_node(x) == nullptr)
        {
            cout<<"Valoarea nu a fost gasita in lista"<<endl;
        } else
        {
            cout<<"Valoarea a fost gasita in lista"<<endl;
        }
        break;
    case 6:
        cout<<"Valoarea de sters din lista: "; cin>>x;
        list.remove_node(x);
        break;
    case 7:
        cout<<"Valoarea minima din lista este: "<<list.min()->val<<endl;
        cout<<"Valoarea maxima din lista este: "<<list.max()->val<<endl;
        break;
    case 8:
        cout<<"Elementele curente ale listei dublu inlantuite
sunt:"<<endl;
        list.display();
        break;
    case 9:
        cout<<"Elementele curente ale listei dublu inlantuite in ordine
inversa sunt:"<<endl;
        list.display_reversed();
        break;
    case 10:
        cout<<"Valoarea elementului al carui succesori trebuie returnat:
"; cin>>x;
        if(list.successor(x) != nullptr)
        {
            cout<<"Succesorul are valoarea "<<list.successor(x)-
>val<<endl;
        } else
        {
            cout<<"Elementul selectat nu are succesori"<<endl;
        }
        break;
    case 11:
        cout<<"Valoarea elementului al carui predecesor trebuie returnat:
"; cin>>x;

```

```

        if(list.predecessor(x) != nullptr)
        {
            cout<<"Predecesorul are valoarea "<<list.predecessor(x)-
>val<<endl;
        } else
        {
            cout<<"Elementul selectat nu are predecesor"<<endl;
        }
        break;
    case 12:
        break;
    default:
        cout<<"Invalid"<<endl;
        return 1;
    }
} while(choice != 12);

return 0;
}
#include <iostream>

using namespace std;

struct node{
    node *prev;
    int val;
    node *next;
};

// ultimul nod din lista are pointerul next care indica spre primul nod, iar
// primul nod din lista are pointerul prev care indica spre ultimul nod
class DoublyCircularLinkedList{
private:
    node *head;
public:
    DoublyCircularLinkedList()
    {
        head = nullptr;
    }
    node *get_head()
    {
        return head;
    }
    node *get_nth_node(int i)
    {
        node *curr = head;
        int count = 0;
        do{
            if(count == i)
            {
                return curr;
            }
            count++;
            curr = curr->next;
        } while(curr != head);
    }
    node *create_node(int data)

```

```

{
    node *aux = new node; // declararea unui nod si alocarea memoriei
    aux->val = data;
    aux->prev = nullptr; // adresa campului prev e necunoscuta
    aux->next = nullptr; // adresa urmatorului camp e necunoscuta
    return aux; // returnare nod aux creat
}

void create_list(int n)
{
    int x;
    for(int i = 0; i < n; i++)
    {
        cout<<"Valoarea nodului curent: "; cin>>x;
        node *aux = create_node(x);
        if(head == nullptr) // daca lista e vida, capul va reprezenta primul
            element introdus
            {
                head = aux;
                // evidentierea circularitatii
                aux->prev = head;
                aux->next = head;
            }
        else // daca exista cel putin un element in lista, inseamna ca se
            modifica campul next al penultimului element din lista, elementul curent devenind
            ultimul la fiecare iteratie
            {
                // nu mai trebuie parcursa lista folosind acel nod curr, deoarece
                se creeaza usor conexiuni circulare de data asta
                aux->prev = head->prev; // elementul curent e mereu legat de
                head; echivalent cu aux->prev = curr
                aux->next = head; // reconstituirea conexiunii circulare pt nodul
                curent
                head->prev->next = aux; // head->prev ajunge la ultimul anterior
                element din lista ce va crea o legatura cu noul ultim element din lista aux, cel
                adaugat
                head->prev = aux;
            }
    }
}

void display()
{
    node *curr = head;
    if(head == nullptr)
    {
        cout<<"Lista e vida"<<endl;
    } else
    {
        cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
        curr = curr->next;
        while(curr != head)
        {
            cout<<curr->val<<" -> ";
            curr = curr->next;
        }
        if(curr == head)
        {
            cout<<"NULL"<<endl;
        }
    }
}

```

```

    }
}
}
void display_reversed() // iterativ; implementarea recursiva de la simply
linked list a fost necesara din cauza lipsei campului prev
{
    node *curr = head;
    if(head == nullptr)
    {
        cout<<"Lista e vida"<<endl;
    } else
    {
        do{
            curr = curr->prev; // head->prev e tail-ul, deci afisarea se face
de la ultimul element la primul datorita buclei while conditionate posterior
            cout<<curr->val<<" -> ";
        } while(curr != head); // continua afisarea pana se revine la capul
listei
    }
}
void insert_node_end(int data) // similar cu implementarea pt create_list()
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
        aux->prev = head;
        aux->next = head;
    } else
    {
        aux->prev = head->prev; // elementul curent e mereu legat de head;
echivalent cu aux->prev = curr
        aux->next = head; // reconstituirea conexiunii circulare pt nodul
curent
        head->prev->next = aux; // head->prev ajunge la ultimul anterior
element din lista ce va crea o legatura cu noul ultim element din lista aux, cel
adaugat
        head->prev = aux;
    }
}
void insert_node_front(int data)
{
    node *aux = create_node(data);
    if(head == nullptr)
    {
        head = aux;
        aux->prev = head;
        aux->next = head;
    } else
    {
        aux->prev = head->prev; // elementul curent e mereu legat de head;
echivalent cu aux->prev = curr
        aux->next = head; // reconstituirea conexiunii circulare pt nodul
curent
        head->prev->next = aux; // head->prev ajunge la ultimul anterior
element din lista ce va crea o legatura cu noul ultim element din lista aux, cel
adaugat
    }
}

```

```

        head->prev = aux;
        head = aux; // noul head e nodul curent
    }
}

void insert_node_after(node *prev_node, int data) // inserarea unui nod dupa
nodul prev_node
{
    if(prev_node == nullptr) // trebuie sa existe ca sa fie inserat un alt
nod dupa el
    {
        return;
    }
    node *aux = new node;
    aux->val = data;
    aux->prev = prev_node;
    aux->next = prev_node->next; // campul next al lui aux e acelasi cu cel
al lui prev node
    prev_node->next->prev = aux; // actualizare camp prev pentru nodul
urmator
    prev_node->next = aux; // actualizare camp next pentru nodul anterior
}

node *search_node(int data) // nimic schimbat fata de simply linked list
{
    node *curr = head; // cautarea iterativa incepe din capul listei
    do{ // parcurgere de la stanga la dreapta
        if(curr->val == data)
        {
            return curr;
        }
        curr = curr->next;
    } while(curr != head); // conditionat posterior pentru ca nu ar intra in
bucloa deloc daca cautarea incepe de la head; primul pas al cautarii incepe de la
head
    return nullptr; // nodul cu valoarea data nu e gasit
}

void remove_node(int data)
{
    node *del_node = search_node(data); // cautare nod bazata pe gasirea
pozitiei in care se afla in lista
    if(del_node == head) // daca nodul care trebuie sters e capul listei,
noul cap al listei e elementul care urmeaza dupa head
    {
        head = head->next; // se muta doar capul listei, restul elementelor
nu sunt afectate
    }
    if(del_node != nullptr) // trebuie sa fie doar gasit in lista, partile de
unlinking functioneaza pentru stergerea unui element aflat in orice pozitie, fara
sa mai existe restrictii
    {
        del_node->prev->next = del_node->next; // unlinking pt campul next al
elementului aflat inaintea lui del node
        del_node->next->prev = del_node->prev; // unlinking pt campul prev al
elementului curent care e inaintea val care trebuie stearsa
    }
    delete del_node; // stergerea efectiva pt eliberarea zonei de memorie
}

node *max()
```

```

{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *max_node = head;
    // de data asta curr e nodul care urmeaza dupa head, deoarece se parcurge
    lista folosind doua variabile care mereu isi incrementeaza pozitiile succesiv
    node *curr = head->next; // daca max_node e primul element din lista, vor
    fi verificate restul elementelor din lista incepand cu al doilea, adica head-
    >next
    do{
        if(max_node->val < curr->val)
        {
            max_node = curr;
        }
        curr = curr->next;
    } while(curr != head);
    return max_node;
}
node *min() // similar cu functia max
{
    if(head == nullptr)
    {
        return nullptr;
    }
    node* min_node = head;
    node* curr = head->next;
    do{
        if(min_node->val > curr->val)
        {
            min_node = curr;
        }
        curr = curr->next;
    } while(curr != head);
    return min_node;
}
node *successor(int data)
{
    if(head == nullptr) // lista e vida
    {
        return nullptr;
    }
    node *curr = head;
    do{
        if(curr->val == data)
        {
            return curr->next;
        }
        curr = curr->next; // deplasare
    } while(curr != head);
    return nullptr; // nodul cu val data nu a fost gasit
}
node *predecessor(int data)
{
    if(head == nullptr) // lista e vida
    {

```

```

        return nullptr;
    }
    node *curr = head;
    do{
        if(curr->val == data)
        {
            return curr->prev;
        }
        curr = curr->next; // deplasare
    } while(curr != head);
    return nullptr; // nodul cu val data nu a fost gasit
}

};

int main() {
    DoublyCircularLinkedList list;
    int choice, x, i;

    do{
        cout<<endl;
        cout<<"1. Crearea unei liste dublu circulare inlantuite\n2. Inserarea
unui nod la sfarsitul listei\n3. Inserarea unui nod la inceputul listei\n4.
Inserarea unui nod dupa un alt nod existent din lista\n5. Cautarea unui nod in
functie de valoare\n6. Stergerea unui nod in functie de valoare\n7. Cautarea
minimului si a maximului din lista\n8. Afisarea elementelor curente din lista\n
9. Afisarea inversa a elementelor din lista\n 10. Cautarea succesorului unui
element\n 11. Cautarea predecesorului unui element\n 12. Iesire meniu\n Tastati
varianta aleasa: ";
        cin>>choice;
        switch(choice)
        {
            case 1:
                int n;
                if(list.get_head() == nullptr) // daca lista nu a fost
initializata inca
                {
                    cout<<"Nr. de noduri ale listei dublu circulare inlantuite:
"; cin>>n;

                    list.create_list(n);
                } else
                {
                    cout<<"Lista exista deja"<<endl;
                }
                break;
            case 2:
                cout<<"Valoarea nodului: "; cin>>x;
                list.insert_node_end(x);
                break;
            case 3:
                cout<<"Valoarea nodului: "; cin>>x;
                list.insert_node_front(x);
                break;
            case 4:
                cout<<"Valoarea nodului: "; cin>>x;
                cout<<endl<<"Pozitia nodului dupa care sa fie inserat: "; cin>>i;
                list.insert_node_after(list.get_nth_node(i), x);
                break;
        }
    } while(choice != 12);
}

```



```

case 5:
    cout<<"Valoarea de cautat in lista: "; cin>>x;
    if(list.search_node(x) == nullptr)
    {
        cout<<"Valoarea nu a fost gasita in lista"<<endl;
    } else
    {
        cout<<"Valoarea a fost gasita in lista"<<endl;
    }
    break;
case 6:
    cout<<"Valoarea de sters din lista: "; cin>>x;
    list.remove_node(x);
    break;
case 7:
    cout<<"Valoarea minima din lista este: "<<list.min()->val<<endl;
    cout<<"Valoarea maxima din lista este: "<<list.max()->val<<endl;
    break;
case 8:
    cout<<"Elementele curente ale listei dublu circulare inlantuite
sunt:"<<endl;
    list.display();
    break;
case 9:
    cout<<"Elementele curente ale listei dublu circulare inlantuite
in ordine inversa sunt:"<<endl;
    list.display_reversed();
    break;
case 10:
    cout<<"Valoarea elementului al carui succesori trebuie returnat:
"; cin>>x;
    if(list.successor(x) != nullptr)
    {
        cout<<"Succesorul are valoarea "<<list.successor(x)-
>val<<endl;
    } else
    {
        cout<<"Elementul selectat nu are succesori"<<endl;
    }
    break;
case 11:
    cout<<"Valoarea elementului al carui predecesor trebuie returnat:
"; cin>>x;
    if(list.predecessor(x) != nullptr)
    {
        cout<<"Predecesorul are valoarea "<<list.predecessor(x)-
>val<<endl;
    } else
    {
        cout<<"Elementul selectat nu are predecesor"<<endl;
    }
    break;
case 12:
    break;
default:
    cout<<"Invalid"<<endl;
    return 1;

```

```

    }
    } while(choice != 12);

    return 0;
}
#include <iostream>
#include <string>

using namespace std;

struct node
{
    char val;
    node *next; // fiind o stiva nu ar avea campul prev
};

class Stack{ // LIFO
private:
    node *top; // avem acces numai la varful stivei, top reprezinta de fapt tail-
ul listei simplu inlantuite
    node *head;
public:
    // constructor pt o stiva vida, pt care pointer-ul top nu va pointa catre
    nimic
    Stack()
    {
        head = nullptr;
        top = nullptr;
    }
    node *get_top()
    {
        return top;
    }
    node *create_node(char data)
    {
        node *aux = new node;
        aux->val = data;
        aux->next = nullptr;
        return aux;
    }
    void push(char data) // cout<<"Valoarea de introdus pe stiva: "; cin>>aux-
>val; // dereferentiere // nu functioneaza pt ca prelucrez caracter cu caracter o
expresie introdusa de la tastatura
    {
        node *aux = create_node(data);
        if(head == nullptr)
        {
            head = aux;
            top = aux; // vf stivei e chiar head-ul listei formate dintr-un
singur element
        } else
        {
            node *curr = head;
            while(curr->next != nullptr)
            {
                curr = curr->next;
            }
        }
    }

```

```

        curr->next = aux;
        top = aux; // vf stivei
    }
}
void pop()
{
    if(top == nullptr)
    {
        cout<<"A avut loc underflow"<<endl;
        return;
    }
    if(head->next == nullptr) // daca exista un singur element care urmeaza a
    fi scos de pe stiva
    {
        head = nullptr;
        cout<<"Valoarea elementului care a fost scos de pe stiva: "<<top-
>val<<endl;
        delete top;
        top = nullptr;
    } else
    {
        node *curr = head;
        while(curr->next != top) // parcurgere linked list pana la penultimul
        element pentru a putea exprima, dupa ce e sters ultimul element din stiva, top in
        functie de curr (nu exista campul prev)
        {
            curr = curr->next; // deplasare
        }
        curr->next = nullptr;
        cout<<"Valoarea elementului care a fost scos de pe stiva: "<<top-
>val<<endl;
        delete top;
        top = curr;
    }
}
void display()
{
    if(head == nullptr)
    {
        cout<<"Stiva e goala"<<endl;
    } else
    {
        node *curr = head;
        cout<<"| "; // baza stivei
        while(curr != top->next) // sau while(curr != nullptr)
        {
            cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
            curr = curr->next;
        }
    }
}
bool empty() const
{
    return top == nullptr; // true daca e vida, false altfel
}
};

```

```

int main()
{
    Stack current_stack, aux_stack; // pe curent_stack se adauga numai
    parantezele deschise, aux_stack il folosesc doar pentru a afisa mesaje exacte
    legate de ce ramane pe stiva, altfel nu are nicio functionalitate reala
    string expr;
    cout<<"Expresia matematica care foloseste paranteze (), {}, [] este: "<<endl;
    cin>>expr;
    /*for(unsigned long long i=0; i<expr.length(); i++)
    {
        current_stack.push(expr[i]);
    }
    current_stack.display();*/
    bool ok = true; // presupun ca expresia e scrisa corect
    for(char c: expr)
    {
        if(c=='(' || c=='[' || c=='{' )// introduc pe stiva doar caracterul
        curent, care e o paranteza deschisa
        {
            current_stack.push(c);
        } else if (c==')' || c==']' || c=='}')
        {
            aux_stack.push(c); // doar pt a trata cazul in care parantezele
            inchise nu au corespondente paranteze deschise
            char top;
            node *top_stack = current_stack.get_top();
            if(top_stack != nullptr) // daca nu are loc underflow, adica lista
            inlantuita nu e vida
            {
                top = top_stack->val;
            }
            // la fiecare pas scot ce am pe stiva current_stack in momentul in
            care intalnesc o paranteza corespondenta inchisa
            if((c==')' && top=='(') || (c==']' && top=='[') || (c=='}') &&
            top=='{'))
            {
                current_stack.pop(); // imbricare corecta, pop de pe stiva la
                parantezele deschise
                aux_stack.pop(); // pop la parantezele inchise
            } else
            {
                ok = false; // de retinut ca despre caracterul curent din
                expresie stim ca e o paranteza inchisa, insa nu corespunde tipul parantezei
                referitor la paranteza deschisa din vf stivei current_stack
                cout<<"Expresia nu are corect imbricate parantezele: ";
                if(c==')')
                {
                    cout<<"()"<<endl;
                } else if (c==']')
                {
                    cout<<"[]"<<endl;
                } else
                {
                    cout<<"{}"<<endl;
                }
            }
        }
    }
}

```

```

    }
    if(!current_stack.empty()) // raman paranteze deschise pe stiva, carora nu
le-am dat pop
    {
        ok = false;
        cout<<"Ce a ramas pe stiva current_stack:"<<endl;
        current_stack.display();
    }
    if(!aux_stack.empty())
    {
        ok = false;
        cout<<"Ce a ramas pe stiva aux_stack:"<<endl;
        aux_stack.display();
    }

    if(ok)
    {
        cout<<"Expresia e scrisa corect";
    } else
    {
        cout<<"Expresia nu e scrisa corect";
    }

    return 0;
}
#include <iostream>
#include <vector>

using namespace std;

struct node
{
    int val;
    node *next;
};

class Stack{ // LIFO
private:
    node *top; // avem acces numai la varful stivei, top reprezinta de fapt tail-
ul listei simplu inlantuite
    node *head;
public:
    // constructor pt o stiva vida, pt care pointer-ul top nu va pointa catre
nimic
    Stack()
    {
        head = nullptr;
        top = nullptr;
    }
    node *get_top()
    {
        return top;
    }
    node *create_node(int data)
    {
        node *aux = new node;
        aux->val = data;
    }

```

```

        aux->next = nullptr;
        return aux;
    }
    void push(int data) // cout<<"Valoarea de introdus pe stiva: "; cin>>aux-
>val; // dereferentiere // nu functioneaza pt ca prelucrez caracter cu caracter o
expresie introdusa de la tastatura
    {
        cout<<"Valoarea elementului care a fost introdus la finalul stivei:
"<<data<<endl;
        node *aux = create_node(data);
        if(head == nullptr)
        {
            head = aux;
            top = aux; // vf stivei e chiar head-ul listei formate dintr-un
singur element
        } else
        {
            node *curr = head;
            while(curr->next != nullptr)
            {
                curr = curr->next;
            }
            curr->next = aux;
            top = aux; // vf stivei
        }
    }
    void pop()
    {
        if(top == nullptr)
        {
            cout<<"A avut loc underflow"<<endl;
            return;
        }
        if(head->next == nullptr) // daca exista un singur element care urmeaza a
fi scos de pe stiva
        {
            head = nullptr;
            cout<<"Valoarea elementului care a fost scos de pe stiva: "<<top-
>val<<endl;
            delete top;
            top = nullptr;
        } else
        {
            node *curr = head;
            while(curr->next != top) // parcurgere linked list pana la penultimul
element pentru a putea exprima, dupa ce e sters ultimul element din stiva, top in
functie de curr (nu exista campul prev)
            {
                curr = curr->next; // deplasare
            }
            curr->next = nullptr;
            cout<<"Valoarea elementului care a fost scos de pe stiva: "<<top-
>val<<endl;
            delete top;
            top = curr;
        }
    }
}

```

```

void display()
{
    if(head == nullptr)
    {
        cout<<"Stiva e goala"<<endl;
    } else
    {
        node *curr = head;
        cout<<"| "; // baza stivei
        while(curr != top->next) // sau while(curr != nullptr)
        {
            cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
            curr = curr->next;
        }
    }
}

bool empty() const
{
    return top == nullptr; // true daca e vida, false altfel
}

};

class Queue { // FIFO
private:
    node *tail; // operatia de enqueue, adaugarea elementelor unei cozi se face
    la sfarsitul acesteia
    node *head; // operatia de dequeue, eliminarea elementelor unei cozi se face
    la inceputul acesteia
public:
    Queue()
    {
        head = nullptr;
        tail = nullptr;
    }
    node *get_head()
    {
        return head;
    }
    node *create_node(int data)
    {
        node *aux = new node;
        aux->val = data;
        aux->next = nullptr;
        return aux;
    }
    void enqueue(int data)
    {
        cout<<"Valoarea elementului care a fost introdus la finalul cozii:
"<<data<<endl;
        node *aux = create_node(data);
        if(head == nullptr)
        {
            head = aux;
        } else
        {
            tail->next = aux; // se adauga la sfarsitul cozii
        }
    }
}

```

```

        tail = aux;
    }
    void dequeue()
    {
        if(head == nullptr)
        {
            cout<<"Coadă e goală"<<endl;
        }
        node *aux = head;
        cout<<"Valoarea elementului care a fost scos de la începutul cozii:
"<<aux->val<<endl;
        head = head->next;
        delete aux;
    }
    void display()
    {
        if(head == nullptr)
        {
            cout<<"Coadă e goală"<<endl;
        } else
        {
            node *curr = head;
            while(curr != nullptr)
            {
                cout<<curr->val<<" -> "; // se afișează valoarea din nodul curent
                curr = curr->next;
            }
        }
    }
    bool empty()
    {
        return head == nullptr;
    }
};

```

```

int main() {
    vector<int> v;
    vector<int> aux;
    Stack stack;
    Queue queue_max;
    Queue queue_prev;
    int n;

    cout<<"Nr. de elemente pe care le conține vectorul: "; cin>>n;
    v.resize(n);
    for(int i = 0; i < n; i++)
    {
        cout<<"v["<<i<<"]=""; cin>>v[i];
    }

    // fiecare element din vector trebuie adăugat pe stivă, iar în momentul în
    // care un element curent din vector e mai mare decât valoarea din vf stivei, se
    // realizează pop și se continuă verificarea condiției până când stiva rămâne goală
    // sau noul varf e mai mare decât val din vector, iar valoarea imediat mai mare e
    // adăugată în queue_max
    for(int i = 0; i < n; i++)
    {
        cout<<endl<<"Pasul "<<i<<":"<<endl;
    }
}

```



```

        // functionalitatea while-ului se aplica numai daca exista mai multe
        valori la rand egale pe stiva ale caror prim mai mare element nu a fost gasit
        while(!stack.empty() && v[i] > stack.get_top()->val) // top = val din vf
        stivei care trebuie comparata cu fiecare element pana e gasit primul mai mare
        decat el
        {
            queue_max.enqueue(v[i]); // in coada se afla toate solutiile,
            deoarece afisarea valorilor elementelor din vector va fi usoara datorita
            operatiei de dequeue, deoarece se vor potrivi atribuirile
            queue_prev.enqueue(stack.get_top()->val);
            stack.pop(); // o data ce e gasit primul cel mai mare element pt o
            valoare anterioara aflata pe stiva, aceasta va fi scoasa de pe stiva, deoarece nu
            mai trebuie comparata cu nimic
            cout<<endl<<"Afisarea stivei: ";
            stack.display();
            cout<<endl<<"Afisarea cozii: "<<endl;
            queue_max.display();
            cout<<endl;
        }
        stack.push(v[i]);
    }
    // daca raman elemente pe stack, inseamna ca nu le-a fost gasit primul cel
    mai mare element, pentru ca se afla la sfarsitul vectorului
    while(!stack.empty())
    {
        queue_max.enqueue(-1);
        queue_prev.enqueue(stack.get_top()->val);
        stack.pop();
    }
    cout<<endl<<endl<<endl;
    while(!queue_prev.empty() && !queue_max.empty())
    {
        cout<<"Primul cel mai mare element al lui "<<queue_prev.get_head()-
        >val<<" este "<<queue_max.get_head()->val<<endl;
        queue_max.dequeue();
        queue_prev.dequeue();
    }

    return 0;
}
/*10
9 2 3 10 11 4 6 23 23 3
11
9 8 7 8 5 4 3 6 10 10 10*/
#include <iostream>
#include <vector>
#include <string>

using namespace std;

// un dictionar e o colectie de perechi (key, value) deci un nod va avea doua
campuri care contin informatii si un pointer next
struct node { // pt a trata coliziunile, folosim simply linked list pt usurinta
    char key; // una dintre cele 26 de litere din alfabet
    string val; // orice cuvant
    node *next;
};

```

```

// un hash table e un array de simply linked lists, iar fiecare lista contine
elementele cu aceeasi cheie
// functia de hashing produce acelasi index/ aceeasi cheie pentru fiecare cuvant,
referitor la litera cu care incepe fiecare
// aceasta coliziune e rezolvata folosind o lista simplu inlantuita pentru
cuvintele care incep cu aceeasi litera, adica fiecarei chei ii corespunde o lista
// indexul din array e indicat de valoarea functiei hash function(value)

```

```

class HashTable {
private:
    // table[index] e de fapt un pointer la un nod (nullptr sau nu), acel nod
    putand avea campul next care sa pointeze catre un alt nod etc.
    vector<node*> table; // o tabela hash interpretata drept vector care
    pointeaza la elemente de tip node, separate chaining
    int table_size;
public:
    HashTable(int size) : table_size(size), table(size)
    {
        for(int i = 0; i < table_size; i++)
        {
            table[i] = nullptr;
        }
    }
    ~HashTable()
    {
        for (int i = 0; i < table_size; i++)
        {
            node* curr = table[i];
            while(curr != nullptr)
            {
                node* aux = curr->next;
                delete curr;
                curr = aux;
            }
        }
    }
    int hash_function(char letter)
    {
        return toupper(letter) - 'A';
    }
    char convert_to_letter(int index) // doar pt afisare
    {
        if (index >= 0 && index < 26)
        {
            return static_cast<char>(index + 'A');
        }
    }
    node *create_node(string data) // caz general: create_node(char key, string
data)
    {
        node *aux = new node;
        aux->key = toupper(data[0]);
        aux->val = data;
        aux->next = nullptr;
        return aux;
    }
}

```

```

void display_list(char letter)
{
    int index = hash_function(letter);
    node *curr = table[index];
    if(curr == nullptr) // nu exista niciun cuvant inserat in dictionar care
sa inceapa cu litera specificata
    {
        cout<<"Nu exista niciun cuvant cu litera
"<<convert_to_letter(index)<<endl;
    } else
    {
        cout<<"Litera "<<curr->key<<": ";
        while(curr != nullptr)
        {
            cout<<curr->val<<" -> "; // se afiseaza valoarea din nodul curent
            curr = curr->next;
        }
        cout<<endl;
    }
}

void display()
{
    for(char letter = 'a'; letter <= 'z'; letter++)
    {
        display_list(letter);
    }
}

void insert_node(string data) // insert_node_end() din lab2
{
    int index = hash_function(data[0]); // gasirea indexului in functie de
hash_function pt a introduce nodul in table
    node *aux = create_node(data); // crearea nodului, in care este
specificata perechea (key, value)
    // verificarea coliziunilor
    if(table[index] == nullptr) // daca pe pozitia index din tabel nu exista
niciun nod, table[index] va fi initializat cu datele necesare
    {
        table[index] = aux;
    } else // partea de chaining intre noduri, adaugarea lor intr-o lista
    {
        node *curr = table[index];
        while(curr->next != nullptr) // parcurgerea listei de la inceput pana
la ultimul element din lista
        {
            curr = curr->next;
        }
        curr->next = aux;
    }
}

node *search_node(string data) // avand un tabel de noduri, cautarea se face
dupa cuvant si vor returna toate valorile asociate
{
    // trebuie gasit indexul corespunzator cuvantului introdus pt a putea
localiza lista prin care trebuie iterat
    int index = hash_function(data[0]);
    node *curr = table[index]; // cautarea iterativa incepe din capul listei,
adica nodul head corespunzator intrarii de pe pozitia index din tabel

```

```

        // cautarea e asemenea intr-o lista simplu inlantuita pnetru a gasi daca
un cuvânt apartine hash table-ului sau nu
        while(curr != nullptr)
        {
            if(curr->val == data)
            {
                return curr;
            }
            curr = curr->next; // urmatorul nod
        }
        return nullptr; // nodul cu valoarea data nu e gasit
    }
    void remove_node(string data)
    {
        int index = hash_function(data[0]);
        node *del_node = search_node(data);
        node *curr = table[index];
        if(del_node == table[index]) // daca nodul care trebuie sters e capul
listei, noul cap al listei e elementul care urmeaza dupa head
        {
            table[index] = table[index]->next; // se muta doar capul listei,
restul elementelor nu sunt afectate
        } else
        {
            if(del_node != nullptr) // daca nodul care trebuie sters exista in
lista, adica functia search_node nu returneaza nullptr, atunci se vor deplasa
restul elementelor
            {
                while(curr != nullptr) // parcurgere linked list pana la tail
pentru a gasi nodul aflat imediat inaintea nodului care trebuie sters
                {
                    if(curr->next == del_node)
                    {
                        // partea de unlinking
                        curr->next = del_node->next; // in mom in care campul
next al nodului curent pointeaza catre nodul care trebuie sters, se reconstituie
legatura pentru a sari peste del node
                    }
                    curr = curr->next; // deplasare
                }
            }
        }
        delete del_node; // stergerea efectiva pt eliberarea zonei de memorie
    }
};

int main() {
    HashTable dict(26);

    int choice;
    string x;
    char ch;

    do{
        cout<<endl;
        cout<<"1. Inserarea unui cuvânt in dictionar\n2. Cautarea unui cuvânt\n3.
Stergerea unui cuvânt\n4. Afisarea cuvintelor curente de la o sectiune\n5.

```

```

Afisarea dictionarului curent\n6. Iesire meniu\n Tastati varianta aleasa: ";
    cin>>choice;
    switch(choice)
    {
        case 1:
            cout<<"Cuvantul de introdus: "; cin>>x;
            dict.insert_node(x);
            break;
        case 2:
            cout<<"Cuvantul de cautat in dictionar: "<<endl; cin>>x;
            if(dict.search_node(x) == nullptr)
            {
                cout<<"Valoarea nu a fost gasita in lista"<<endl;
            } else
            {
                cout<<"Valoarea a fost gasita in lista"<<endl;
            }
            break;
        case 3:
            cout<<"Cuvantul de sters din dictionar: "; cin>>x;
            dict.remove_node(x);
            break;
        case 4:
            cout<<"Litera de inceput a cuvintelor: "; cin>>ch;
            cout<<endl<<"Cuvintele curente aflate la sectiunea cu litera
"<<ch<<" sunt:"<<endl;
            dict.display_list(ch);
            break;
        case 5:
            cout<<"Cuvintele curente ale dictionarului sunt:"<<endl;
            dict.display();
            break;
        case 6:
            break;
        default:
            cout<<"Invalid"<<endl;
            return 1;
    }
} while(choice != 6);

    return 0;
}

```

```

#include <bits/stdc++.h>
#include <iostream>
#include <vector>

```

```
using namespace std;
```

```

struct node{
    int val;
    // un arbore binar are maxim 2 copii
    node *l_child; // copil stanga -> subarbore stanga
    node *r_child; // copil dreapta -> subarbore dreapta
};

```

// pentru ca un arbore binar sa fie balansat, ar trebui sa existe un numar egal de noduri pentru fiecare subarbore, de aceea ar trebui gasita o radacina care sa

respecte conditia
// valorile din subarborele din stanga sunt mai mici decat cea a nodului curent,
iar valorile subarborilor din dreapta sunt mai mari
// diferenta de inaltime dintre subarborii din stanga si dreapta sa nu depaseasca
1

```
node *create_node(int data)
{
    node *aux = new node; // declararea unui nod si alocarea memoriei
    aux->val = data;
    aux->l_child = nullptr; // adresa campului prev e necunoscuta
    aux->r_child = nullptr; // adresa urmatorului camp e necunoscuta
    return aux; // returnare nod aux creat
}

// functia de construire a bst va returna nodul radacina convenabil folosind
divide et impera
// array-ul se injumatateste, radacina arborelui fiind elementul din mijloc,
apoi se parcurge subarray-urile din stanga si dreapta recursiv
node *dei_build_balanced_bst_from_sorted_arr(int v[], int start, int end) //
limitele vectorului; inferioara, superioara
{
    if(start > end) // caz de baza recursivitate
    {
        return nullptr;
    }
    int middle = (start + end) / 2;
    cout<<v[middle]<<" ";
    node *subtree_root = create_node(v[middle]);
    subtree_root -> l_child = dei_build_balanced_bst_from_sorted_arr(v,
start, middle-1);
    subtree_root -> r_child = dei_build_balanced_bst_from_sorted_arr(v,
middle+1, end);
    return subtree_root;
}

// parcurgere in-ordine, insemnand ca radacina fiecarui subarbore e visitata
dupa ce subarborele sau stang a fost parcurs, dar inainte de inceperea
parcurgerii subarborului drept
void display_inorder(node *root)
{
    if(root != nullptr)
    {
        display_inorder(root->l_child);
        cout<<root->val<<" ";
        display_inorder(root->r_child);
    }
}

void display_preorder(node *root)
{
    if(root != nullptr)
    {
        cout<<root->val<<" ";
        display_preorder(root->l_child);
        display_preorder(root->r_child);
    }
}

void display_postorder(node *root)
{

```

```

        if(root != nullptr)
        {
            display_postorder(root->l_child);
            display_postorder(root->r_child);
            cout<<root->val<<" ";
        }
    }

int main()
{
    int v[] = {2, 2, 4, 5, 6, 7, 9, 11};
    /*
                5
            2   7
        2  4  6  9
                11
    */
    int n = sizeof(v)/sizeof(v[0]);
    node *root = dei_build_balanced_bst_from_sorted_arr(v, 0, n-1);
    cout<<endl<<"Parcurea in-ordine:"<<endl;
    display_inorder(root);
    cout<<endl<<"Parcurea pre-ordine:"<<endl;
    display_preorder(root);
    cout<<endl<<"Parcurea post-ordine:"<<endl;
    display_postorder(root);
    return 0;
}
#include <iostream>
#include <vector>
#define COUNT 10

using namespace std;

struct node{
    int val;
    // un arbore binar are maxim 2 copii
    node *l_child; // copil stanga -> subarbore stanga
    node *r_child; // copil dreapta -> subarbore dreapta
};

// interclasarea a doi arbori echilibrati presupune parcurgerea in-ordine a
// fiecarui arbore si retinerea elementelor intr-un array comun, apoi construirea
// unui arbore balansat
node *create_node(int data)
{
    node *aux = new node; // declararea unui nod si alocarea memoriei
    aux->val = data;
    aux->l_child = nullptr; // adresa campului prev e necunoscuta
    aux->r_child = nullptr; // adresa urmatorului camp e necunoscuta
    return aux; // returnare nod aux creat
}

// functia de construire a bst va returna nodul radacina convenabil folosind
// divide et impera
// array-ul se injumatateste, radacina arborelui fiind elementul din mijloc,
// apoi se parcurge subarray-urile din stanga si dreapta recursiv
node *dei_build_balanced_bst_from_sorted_arr(vector<int> v, int start, int
end) // limitele vectorului; inferioara, superioara

```

```

{
    if(start > end) // caz de baza recursivitate
    {
        return nullptr;
    }
    int middle = (start + end) / 2;
    //cout<<v[middle]<<" ";
    node *subtree_root = create_node(v[middle]);
    subtree_root -> l_child = dei_build_balanced_bst_from_sorted_arr(v,
start, middle-1);
    subtree_root -> r_child = dei_build_balanced_bst_from_sorted_arr(v,
middle+1, end);
    return subtree_root;
}
// parcurgere in-ordine, insemnand ca radacina fiecarui subarbore e visitata
dupa ce subarboarele sau stang a fost parcurs, dar inainte de inceperea
parcurgerii subarboarelui drept
vector<int> create_array_inorder(node *root, vector<int> &aux)
{
    if(root != nullptr)
    {
        create_array_inorder(root->l_child, aux);
        aux.push_back(root->val);
        create_array_inorder(root->r_child, aux);
    }
    return aux;
}
void display_inorder(node *root)
{
    if(root != nullptr)
    {
        display_inorder(root->l_child);
        cout<<root->val<<" ";
        display_inorder(root->r_child);
    }
}
vector<int> merge(vector<int> &a, vector<int> &b)
{
    int i = 0, j = 0, k = 0;
    vector<int> c;
    c.resize(a.size() + b.size());
    while(i < a.size() && j < b.size())
    {
        if(a[i] < b[j])
        {
            c[k] = a[i];
            k++; i++;
        } else
        {
            c[k] = b[j];
            k++; j++;
        }
    }
    while(i < a.size())
    {
        c[k] = a[i];
        k++; i++;
    }
}

```



```

    }
    while(j < b.size())
    {
        c[k] = b[j];
        k++; j++;
    }
    return c;
}

// fac interclasarea a doi arbori cu toate ca as fi putut sa interclasez
vectorii cu ajutorul carora sunt creati arborii si sa evit cativa pasi
void merge_balanced_bsts(node *root_t1, node *root_t2)
{
    vector<int> aux1, aux2, merged;
    // reordonarea elementelor din ambii arbori
    create_array_inorder(root_t1, aux1);
    create_array_inorder(root_t2, aux2);
    merged.resize(aux1.size() + aux2.size()); // din cauza accesarii prin
index, as fi putut sa implementez cu push_back()
    merged = merge(aux1, aux2);
    /*cout<<"Interclasarea celor doi vectori rezultati din arborii
echilibrati binari de cautare:"<<endl;
    for(int i = 0; i < merged.size(); i++)
    {
        cout<<merged[i]<<" ";
    }*/
    // crearea unui nou arbore in functie de vectorul care are valorile
comune sortate crescator; parcurgerea in-ordine a unui arbore binar respecta
conditia
    node *root_merged = dei_build_balanced_bst_from_sorted_arr(merged, 0,
merged.size()-1);
    cout<<endl<<"Arborele rezultat parcurs in in-ordine:"<<endl;
    display_inorder(root_merged);
}

int main()
{
    vector<int> a = {2, 2, 4, 5, 6, 7, 9, 11};
    /*
        5
      2   7
     2  4  6  9
          11
    */
    vector<int> b = {2, 5, 6, 8, 9, 10, 12, 12, 15, 17, 18};
    /*
        10
       5   15
      2   9   12  18
        6       17
         8
    */
    // conexiuni radacini-noduri vectori pt fiecare arbore construit, desi puteam
sa ii construiesc manual
    node *root_t1 = dei_build_balanced_bst_from_sorted_arr(a, 0, a.size()-1);
    node *root_t2 = dei_build_balanced_bst_from_sorted_arr(b, 0, b.size()-1);
    merge_balanced_bsts(root_t1, root_t2);
    /*

```

```

      8
    5      12
  2      6      9      15
2 2      5 6      9 10      12 17
  4      7      11      18

```

```

*/
return 0;
}
#include <iostream>

using namespace std;

// https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-
bst/?ref=lbp
// gasirea succesorului si predecesorului inorder al unei valori din bst
// caz particular: cheia nu e gasita, se intorc valorile intre care se afla cheia
// predecesor: copilul cel mai din dreapta al subarborelui stang, pt ca intr-un
bst ar fi valoarea maxima care e mai mica decat val din nodul al carui predecesor
e cautat
// succesor: copilul cel mai din stanga al subarborelui drept, pt ca ar fi
valoarea minima care e mai mare decat valoarea nodului curent
// cheie == radacina curenta, atunci se poate realiza algoritmul de parcurgere
// cheie < radacina curenta, atunci inseamna ca o pt a gasi valori apropiate se
cauta in subarborele din stanga care va avea elemente mai mici decat radacina
// de fapt succesorul e chiar radacina (copilul cel mai din stanga al
subarborelui drept), algoritmul operand in subarborele stang (pt a gasi
predecesorul)
// cheie > radacina curenta, atunci inseamna ca o pt a gasi valori apropiate se
cauta in subarborele din dreapta care va avea elemente mai mari decat radacina
// de fapt predecesorul e chiar radacina (copilul cel mai din dreapta al
subarborelui stang), algoritmul operand in subarborele drept (pt a gasi
succesorul)

struct node{
    int val;
    // un arbore binar are maxim 2 copii
    node *l_child; // copil stanga -> subarbore stanga
    node *r_child; // copil dreapta -> subarbore dreapta
};

node *find_predecessor(node *subtree_root, int data) // caz general predecesor
{
    node *curr = subtree_root;
    node *pre = new node;
    pre->l_child = nullptr;
    pre->r_child = nullptr;
    // curr va parcurge subarborele stang curent, prioritate avand pozitia din
dreapta
    while(curr != nullptr)
    {
        if(curr->val <= data)
        {
            pre = curr;
            curr = curr->r_child; // dreapta
        } else
        {
            curr = curr->l_child; // stanga
        }
    }
}

```

```

    }
}
return pre;
}

node *find_successor(node *subtree_root, int data) // caz general succesori
{
    node *curr = subtree_root;
    node *suc = new node;
    suc->l_child = nullptr;
    suc->r_child = nullptr;
    // curr va parcurge subarborele drept curent, prioritate avand pozitia din
    stanga
    while(curr != nullptr)
    {
        if(curr->val >= data)
        {
            suc = curr;
            curr = curr->l_child; // stanga
        } else
        {
            curr = curr->r_child; // dreapta
        }
    }
    return suc;
}

// cautare recursiva a nodului cu valoarea data
void search(node *root, node *&suc, node *&pre, int data)
{
    if(root == nullptr) // atentie!!!
    {
        return;
    }
    if(root->val == data) // a fost gasit nodul cu valoarea data ale carui
    predecesori si succesori urmeaza a fi cautati
    {
        suc = find_successor(root, data);
        pre = find_predecessor(root, data);
    } else if(root->val > data) // cautare stanga
    {
        suc = root;
        search(root->l_child, suc, pre, data);
    } else // cautare dreapta
    {
        pre = root;
        search(root->r_child, suc, pre, data);
    }
}

node *create_node(int data)
{
    node *aux = new node; // declararea unui nod si alocarea memoriei
    aux->val = data;
    aux->l_child = nullptr; // adresa campului prev e necunoscuta
    aux->r_child = nullptr; // adresa urmatorului camp e necunoscuta
    return aux; // returnare nod aux creat
}

```

```

}

node *insert(node *aux, int data) // inserare arbore binar
{
    if(aux == nullptr) // daca nu exista e creat pt a adauga noduri copii la
    stanga sau dreapta nodului curent
    {
        return create_node(data);
    }
    if(data < aux->val) // valorile mai mici sunt in stanga
    {
        aux->l_child = insert(aux->l_child, data);
    } else // valorile mai mari sunt in dreapta
    {
        aux->r_child = insert(aux->r_child, data);
    }
    return aux;
}

int main() {
    int key = 65;    //Key to be searched in BST

    /* Let us create following BST
          50
         /  \
        30   70
       /  \  /  \
      20  40 60  80 */
    node *root = create_node(50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    node *predecessor = nullptr, *successor = nullptr;
    search(root, successor, predecessor, key);
    if(successor != nullptr)
    {
        cout<<"Succesorul este: "<<successor->val<<endl;
    }
    if(predecessor != nullptr)
    {
        cout<<"Predecesorul este: "<<predecessor->val<<endl;
    }

    return 0;
}
#include <iostream>

using namespace std;

// https://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-
bst-is-not-allowed/?ref=lbp
// sa se gaseasca al k-lea cel mai mare element
// din moment ce o parcurgere in-ordine afiseaza valorile nodurilor in ordine

```

crescatoare, trebuie realizata o parcurgere in-ordine inversa pentru a gasi cele mai mari elemente
// cand contorul devine k, a fost gasita nodul respectiv

```
struct node{
    int val;
    // un arbore binar are maxim 2 copii
    node *l_child; // copil stanga -> subarbore stanga
    node *r_child; // copil dreapta -> subarbore dreapta
};

// cautare recursiva a nodului cu valoarea data
void search_reverse_inorder(node *root, int k, int &count)
{
    if(root == nullptr || count > k)
    {
        return;
    }
    // parcurgere inversa a parcurgerii in-ordine
    search_reverse_inorder(root->r_child, k, count); // e vizitat cel mai mare element
    // e vizitat elementul curent deci se incrementeaza count
    count++;
    if(count == k)
    {
        cout<<"Al k-lea cel mai mare element este: "<<root->val;
        return;
    }
    search_reverse_inorder(root->l_child, k, count); // e vizitat cel mai mic element
}

node *create_node(int data)
{
    node *aux = new node; // declararea unui nod si alocarea memoriei
    aux->val = data;
    aux->l_child = nullptr; // adresa campului prev e necunoscuta
    aux->r_child = nullptr; // adresa urmatorului camp e necunoscuta
    return aux; // returnare nod aux creat
}

node *insert(node *aux, int data) // inserare arbore binar
{
    if(aux == nullptr) // daca nu exista e creat pt a adauga noduri copii la stanga sau dreapta nodului curent
    {
        return create_node(data);
    }
    if(data < aux->val) // valorile mai mici sunt in stanga
    {
        aux->l_child = insert(aux->l_child, data);
    } else // valorile mai mari sunt in dreapta
    {
        aux->r_child = insert(aux->r_child, data);
    }
    return aux;
}
```

```

int main() {
    /* Let us create following BST
        50
       /  \
      30   70
     /  \  /  \
    20  40 60  80 */
    node *root = create_node(50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    int k, count = 0;
    cout<<"Valoarea lui k: "; cin>>k;
    search_reverse_inorder(root, k, count);

    return 0;
}
#include <iostream>
#include <vector>
#include <algorithm> // pt sortare

using namespace std;

// DSU (disjoint set union) cu o cale care indica parintele si rangul curent
class DSU {
private:
    int *parent;
    int *rank;
public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = -1; // fiecare element e propriul sau părinte la inceput
            rank[i] = 1; // rang init e 1
        }
    }
    int find(int i) // gasirea caii parcurse recursiv
    {
        if(parent[i] == -1)
        {
            return i;
        }
        return parent[i] = find(parent[i]);
    }
    void unite(int x, int y) // reuniune 2 set-uri
    {
        int s1 = find(x);
        int s2 = find(y);
        if(s1 != s2)
        {

```

```

        // se reunesc pe baza rangului
        if(rank[s1] < rank[s2])
        {
            parent[s1] = s2;
        } else if(rank[s1] > rank[s2])
        {
            parent[s2] = s1;
        } else
        {
            parent[s2] = s1;
            rank[s1] += 1;
        }
    }
}

};

class Graph {
    vector<vector<int>> edgelist; // Listă de muchii
    int V; // Numărul de vârfuri

public:
    Graph(int V){ this->V = V;}
    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y }); // greutatea w intre vf x si y
    }
    void kruskals_mst()
    {
        // sortare dupa weight
        sort(edgelist.begin(), edgelist.end());
        DSU s(V);
        int cost = 0; // cost total al arborelui de acoperire minima
        cout << "Muchiile in MST sunt:" << endl;
        for (auto edge : edgelist)
        {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            if (s.find(x) != s.find(y)) // nu are voie sa formeze cicluri
            {
                s.unite(x, y); // reuniune set-uri cu vf x si y
                cost += w;
                cout<<x<<" -- "<<y<<" == "<<w<<endl;
            }
        }
        cout<<"Costul minim al MST-ului: "<<cost<<endl;
    }
};

int main() {
    Graph g(4); // 4 vf
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);
    g.kruskals_mst();
}

```

```

        return 0;
    }
#include <iostream>
#include <climits> // pt INT_MAX

using namespace std;

#define V 5 // nr vf graf

int minKey(int key[], bool mstSet[]) // gaseste vf cu cea mica cheie/ cost din
multimea de vf care inca nu sunt incluse in parborele minim partial curent
construit
{
    // se init variabilele pt minim
    int min = INT_MAX, min_index;
    for(int v = 0; v < V; v++)
    {
        if(mstSet[v] == false && key[v] < min) // vf cu cel mai mic cost si
negasit
        {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

void printMST(int parent[], int graph[V][V])
{
    cout << "Muchie \tGreutate\n";
    for(int i = 1; i < V; i++)
    {
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<endl;
    }
}

void primMST(int graph[V][V])
{
    int parent[V]; // pt a stoca MST-ul construit
    int key[V]; // val cheilor folosite pentru a alege muchia cu greutatea minima
    bool mstSet[V]; // multimea de vf incluse in MST
    // conventie: init toate cheile cu infinity
    for(int i = 0; i < V; i++)
    {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    key[0] = 0; // se cheia 0 astfel incat acest vf sa fie selectat primul
(start_node)
    parent[0] = -1; // nu are parinte start_node
    for(int count = 0; count < V - 1; count++) // V-1 vf pt MST care va rezulta
    {
        int u = minKey(key, mstSet); // se alege vf cu cheia minima
        mstSet[u] = true; // se adauga in multimea de vf din MST-ul curent
        for(int v = 0; v < V; v++)
        {
            // graph[u][v] este nenul doar pt vf adiacente

```



```

        // mstSet[v] este fals pentru vf care nu sunt inca incluse in MST
        // se actualizeaza cheia doar daca val de la graph[u][v] este mai
        mica decat key[v]
        if(graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
        {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }
    printMST(parent, graph);
}

int main()
{
    // matricea de adiacenta
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    primMST(graph);
    return 0;
}

```