Fisa de examen - Structuri de date (fara diacritice)

Rezumat rapid al complexitatilor si exemple de cod C++ pentru fiecare structura de baza.

Structura / Operatie	Acces	Cautare	Inserare (mijloc)	Stergere (mijloc)
Vector / Array	O(1)	O(n)	O(n)	O(n)
Lista inlantuita	O(n)	O(n)	O(1)*	O(1)*
Stiva (Stack)	-	-	O(1) push	O(1) pop
Coada (Queue)	-	-	O(1) enqueue	O(1) dequeue
BST echilibrat	O(log n)	O(log n)	O(log n)	O(log n)
BST degenerat	O(n)	O(n)	O(n)	O(n)
BFS / DFS	-	O(V+E)	-	-

* O(1) doar daca avem deja adresa nodului unde inseram/stergem.

1) Vector / Array (acces direct, memorie contigua)

Idei cheie: elemente contigue, dimensiune fixa; acces O(1); inserarile/stergerile la mijloc sunt O(n).

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

int main() {
    int v[5] = {1,2,3,4,5};      // array static
    cout << v[2] << "\n";        // acces O(1)
    v[2] = 10;                   // update O(1)

    // Daca vrem dimensiune variabila folosim vector
    vector<int> a = {1,2,3,4,5};
    a.push_back(6);              // inserare la final amortizat O(1)
    a.erase(a.begin()+2);        // stergere la mijloc O(n)
    cout << a.size() << "\n";
    return 0;
}
```

2) Lista simplu inlantuita (singly linked list)

Idei cheie: noduri legate prin pointeri; acces secvential O(n); inserare/sterge O(1) daca ai pointer la pozitie.

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

struct Node {
    int data;
    Node* next;
    Node(int d, Node* n=nullptr): data(d), next(n) {}
};

void push_front(Node*& head, int x) {
    head = new Node(x, head);    // O(1)
}

bool erase_after(Node* prev) {
    if (!prev || !prev->next) return false;
    Node* del = prev->next;
    prev->next = del->next;
    delete del;                  // O(1)
    return true;
}

void print_list(Node* head) {
```

```
    for (Node* p=head; p; p=p->next) cout << p->data << " ";
    cout << "\n";
}

int main() {
    Node* head = nullptr;
    push_front(head, 30);
    push_front(head, 20);
    push_front(head, 10);        // lista: 10 -> 20 -> 30
    print_list(head);
    erase_after(head);           // sterge 20
    print_list(head);            // 10 -> 30
    // TODO: eliberare rest (delete) in practica
    return 0;
}
```

3) Stiva (Stack) - LIFO

Operatii: push, pop, top - toate O(1). Putem folosi std::stack.

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

int main() {
    stack<int> st;
    st.push(5);
    st.push(10);
    st.push(15);
    cout << st.top() << "\n"; // 15
    st.pop();                 // elimina 15
    cout << st.top() << "\n"; // 10
    cout << st.size() << "\n";
    return 0;
}
```

4) Coada (Queue) - FIFO

Operatii: enqueue (push), dequeue (pop), front - O(1).

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

int main() {
    queue<int> q;
    q.push(7);
    q.push(8);
    q.push(9);
    cout << q.front() << "\n"; // 7
    q.pop();                   // elimina 7
    cout << q.front() << "\n"; // 8
    cout << q.size() << "\n";
    return 0;
}
```

5) Arbore binar de cautare (BST)

Proprietate: left < root < right. Insert/cautare O(log n) in medie; O(n) in cel mai rau caz.

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

struct Node {
    int key;
    Node *left, *right;
    Node(int k): key(k), left(nullptr), right(nullptr) {}
};
Node* insert(Node* root, int k) {
```

```
    if (!root) return new Node(k);
    if (k < root->key) root->left = insert(root->left, k);
    else if (k > root->key) root->right = insert(root->right, k);
    return root;
}

bool search(Node* root, int k) {
    if (!root) return false;
    if (root->key == k) return true;
    if (k < root->key) return search(root->left, k);
    return search(root->right, k);
}

void inorder(Node* r) {
    if (!r) return;
    inorder(r->left);
    cout << r->key << " ";
    inorder(r->right);
}

int main() {
    Node* root = nullptr;
    for (int x: {8,3,10,1,6,14}) root = insert(root, x);
    cout << boolalpha << search(root, 14) << "\n"; // true
    inorder(root); // 1 3 6 8 10 14
    cout << "\n";
    return 0;
}
```

6) Heap (min-heap) cu priority_queue

priority_queue default este max-heap; pentru min-heap folosim greater<>.

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

int main() {
    priority_queue<int, vector<int>, greater<int>> pq; // min-heap
    for (int x: {5,1,7,2,9}) pq.push(x);
    while (!pq.empty()) {
        cout << pq.top() << " "; // 1 2 5 7 9
        pq.pop();
    }
    cout << "\n";
    return 0;
}
```

7) Graf (lista de adiacenta) + BFS/DFS

Reprezentare prin liste de adiacent a; BFS viziteaza pe niveluri, DFS pe adancime.

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

void bfs(int start, const vector<vector<int>>& adj) {
    vector<bool> vis(adj.size(), false);
    queue<int> q;
    vis[start] = true; q.push(start);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        cout << u << " ";
        for (int v : adj[u]) if (!vis[v]) {
            vis[v] = true;
            q.push(v);
        }
    }
    cout << "\n";
```

```
}

void dfs_rec(int u, const vector<vector<int>>& adj, vector<bool>& vis) {
    vis[u] = true;
    cout << u << " ";
    for (int v : adj[u]) if (!vis[v]) dfs_rec(v, adj, vis);
}

int main() {
    int n = 5;
    vector<vector<int>> adj(n);
    auto add_edge = [&](int a, int b) {
        adj[a].push_back(b);
        adj[b].push_back(a);
    };
    add_edge(1,2); add_edge(2,3); add_edge(1,4); add_edge(4,3);

    cout << "BFS from 1: ";
    bfs(1, adj);

    cout << "DFS from 1: ";
    vector<bool> vis(n,false);
    dfs_rec(1, adj, vis);
    cout << "\n";
    return 0;
}
```

Tip: exerseaza implementarea de la zero, apoi compara cu STL. In examen, explica clar conceptele si complexitatile.