Ghid practic: Priority Queue, Map, Unordered Map (C++ & Java)

Fara diacritice. Explicatii detaliate, exemple de cod si bune practici pentru examen.

Structura	Implementare uzuala	Ordine	Operatii tipice (timp)
Priority Queue (min-heap)	Binary heap (array/vector)	Minimul la varf	top O(1), push O(log n), pop O(log n), build O
Priority Queue (max-heap)	Binary heap (array/vector)	Maximul la varf	top O(1), push O(log n), pop O(log n), build O
Map (ordonat)	Red-Black Tree (BST echilibrat)	Chei sortate	insert/find/erase O(log n)
Unordered Map	Hash table	Fara ordine	insert/find/erase O(1) amortizat; worst O(n)

1) Priority Queue (min vs. max)

Concept: o coada cu prioritate care extrage mereu elementul cu prioritatea cea mai buna (min sau max). Implementare uzuala: binary heap intr-un vector.

Complexitati: top O(1), push/pop O(log n); construire heap din n elemente O(n).

Limitari uzuale: nu ai iterare in ordine globala, iar in C++ std::priority_queue nu ofera decrease-key sau actualizare directa.

C++: max-heap (implicit) si min-heap (pe dos)

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

int main() {
    // Max-heap (implicit)
    priority_queue<int> maxpq;
    maxpq.push(10);
    maxpq.push(3);
    maxpq.push(7);
    cout << "Max top = " << maxpq.top() << "\n"; // 10
    maxpq.pop(); // elimina maximul

    // Min-heap ("pe dos") cu comparator greater<>
    priority_queue<int, vector<int>, greater<int>> minpq;
    minpq.push(10);
    minpq.push(3);
    minpq.push(7);
    cout << "Min top = " << minpq.top() << "\n"; // 3
    minpq.pop(); // elimina minimul

    // Truc: pentru prioritati custom, foloseste (prioritate, valoare)
    using Item = pair<int,int>; // (priority, value)
    priority_queue<Item, vector<Item>, greater<Item>> tasks; // min-heap dupa priority
    tasks.push({2, 100});
    tasks.push({1, 42});
    cout << "Task cu prioritate minima: " << tasks.top().second << "\n"; // 42
    return 0;
}
```

Java: min-heap (implicit) si max-heap (pe dos)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Min-heap implicit
        PriorityQueue<Integer> minPQ = new PriorityQueue<>();
        minPQ.add(10);
        minPQ.add(3);
        minPQ.add(7);
        System.out.println("Min top = " + minPQ.peek()); // 3
        minPQ.poll(); // elimina minimul

        // Max-heap ("pe dos") cu reverseOrder()
        PriorityQueue<Integer> maxPQ = new PriorityQueue<>(Comparator.reverseOrder());
        maxPQ.add(10);
        maxPQ.add(3);
        maxPQ.add(7);
        System.out.println("Max top = " + maxPQ.peek()); // 10
        maxPQ.poll(); // elimina maximul

        // Prioritati custom: (priority, value)
        class Task {
            int priority, value;
            Task(int p, int v) { priority = p; value = v; }
        }
        PriorityQueue<Task> tasks = new PriorityQueue<>(Comparator.comparingInt(t -> t.priority
        tasks.add(new Task(2, 100));
        tasks.add(new Task(1, 42));
        System.out.println("Task min priority value = " + tasks.peek().value); // 42
    }
}
```

Cand folosesti Priority Queue:

Vrei mereu minimul sau maximul rapid (e.g., Dijkstra cu min-heap pentru nodul cu distanta minima).

Ai nevoie de top-k elemente (pastrezi un heap de dimensiune k).

Scheduling simplu dupa prioritate (task-uri, evenimente).

2) Map (ordonat) - RB Tree

std::map (C++) si TreeMap (Java) sunt implementate cu Red-Black Tree (BST echilibrat). Cheile sunt sortate. Operatii O(log n).

C++: std::map - insert/find/erase, iterare ordonata, range queries

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

int main() {
    map<string, int> freq;

    // insert/update
    freq["apple"]++;
```

```
    freq["banana"] += 2;
    freq.insert({"cherry", 5});

    // find
    auto it = freq.find("banana");
    if (it != freq.end()) cout << "banana -> " << it->second << "\n";

    // iterare in ordine lexicografica a cheilor
    for (auto &kv : freq) cout << kv.first << " " << kv.second << "\n";

    // range query: toate cheile in [b, d)
    auto lo = freq.lower_bound("b");
    auto hi = freq.lower_bound("d");
    for (auto it = lo; it != hi; ++it) cout << it->first << "\n";

    // erase
    freq.erase("apple");

    // comparator custom (ordine descrescatoare)
    map<int, string, greater<int>> mdesc;
    mdesc[10] = "ten"; mdesc[3] = "three"; mdesc[7] = "seven";
    for (auto &kv : mdesc) cout << kv.first << " "; // 10 7 3
    return 0;
}
```

Java: TreeMap - ordine, subMap/headMap/tailMap

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        TreeMap<String, Integer> freq = new TreeMap<>();

        freq.put("apple", 1);
        freq.put("banana", 2);
        freq.put("cherry", 5);

        System.out.println(freq.getOrDefault("banana", 0)); // 2

        // iterare in ordine
        for (Map.Entry<String, Integer> e : freq.entrySet())
            System.out.println(e.getKey() + " " + e.getValue());

        // intervale
        SortedMap<String, Integer> sub = freq.subMap("b", "d"); // [b, d)
        System.out.println(sub.keySet()); // [banana, cherry]

        // comparator custom (descrescator)
        TreeMap<Integer, String> desc = new TreeMap<>(Comparator.reverseOrder());
        desc.put(10, "ten"); desc.put(3, "three"); desc.put(7, "seven");
        System.out.println(desc.keySet()); // [10, 7, 3]
    }
}
```

Cand folosesti Map (ordonat):

Ai nevoie de ordine pe chei (rapoarte sortate, leaderboard, index ordonat).

Ai nevoie de range queries (lower_bound/upper_bound, subMap).

Vrei garantii de O(log n) stabile indiferent de caz.

3) Unordered Map - Hash Table

std::unordered_map (C++) si HashMap (Java) folosesc hash table. Operatii amortizate O(1); fara ordine; pot aparea rehash-uri.

C++: std::unordered_map - insert/find/erase, iterare, hash custom

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

// Hash custom pentru o structura
struct Point { int x, y; };
struct PointHash {
    size_t operator()(const Point& p) const {
        // combinare simpla de hash (x, y)
        return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
    }
};
struct PointEq {
    bool operator()(const Point& a, const Point& b) const {
        return a.x == b.x && a.y == b.y;
    }
};

int main() {
    unordered_map<string, int> freq;
    freq["apple"]++;
    freq["banana"] += 2;
    cout << freq["banana"] << "\n"; // 2 (atentie: operator[] insereaza daca nu exista)

    // find
    auto it = freq.find("cherry");
    if (it == freq.end()) cout << "nu exista 'cherry'\n";

    // erase
    freq.erase("apple");

    // iterare (fara ordine garantata)
    for (auto &kv : freq) cout << kv.first << " " << kv.second << "\n";

    // hash custom
    unordered_map<Point, int, PointHash, PointEq> mp;
    mp[{1,2}] = 10;
    cout << mp[{1,2}] << "\n";

    // control load factor / buckets
    freq.reserve(1000); // reduce rehash-uri pentru ~1000 elemente
    return 0;
}
```

Java: HashMap - operatii de baza si bune practici

```
import java.util.*;

class Point {
    int x, y;
```

```
    Point(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Point)) return false;
        Point p = (Point)o;
        return x == p.x && y == p.y;
    }
    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
}

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> freq = new HashMap<>();
        freq.put("apple", 1);
        freq.put("banana", 2);
        freq.putIfAbsent("cherry", 0);
        freq.compute("banana", (k, v) -> v == null ? 1 : v + 1); // 3

        System.out.println(freq.get("banana")); // 3
        System.out.println(freq.containsKey("cherry")); // true

        // iterare (fara ordine garantata)
        for (Map.Entry<String, Integer> e : freq.entrySet())
            System.out.println(e.getKey() + " " + e.getValue());

        // tip: setati capacitatea initiala daca stiti marimea
        HashMap<Point, Integer> mp = new HashMap<>(1024, 0.75f);
        mp.put(new Point(1,2), 10);
        System.out.println(mp.get(new Point(1,2))); // 10 (datorita equals/hashCode corecte)
    }
}
```

Cand folosesti Unordered Map:

Ai nevoie de acces foarte rapid mediu la chei, fara cerinte de ordine.

Numaratoare de frecvente, dictionare, mapari ID -> obiect.

Evita cand ai nevoie de intervale sau iterare sortata.

Tips & Pitfalls (pentru examen)

• Priority Queue: in C++ implicit este max-heap; pentru min-heap foloseste greater<>. In Java, implicit este min-heap; pentru max-heap foloseste Comparator.reverseOrder().

• Map (ordonat): potrivit pentru intervale si ordine; foloseste lower_bound/upper_bound (C++) sau subMap/headMap (Java).

• Unordered Map: media este O(1), dar pot aparea cazuri O(n) (coliziuni multe). Alege hash/equals corect (Java) sau functie de hash custom (C++).

• Evita operator[] in unordered_map daca doar verifici existenta; foloseste find/contains pentru a nu insera accidental.

• Rezerva capacitatea (reserve in C++, capacitate initiala in Java) pentru a reduce rehash-urile.

• Pentru chei duplicate: foloseste multimap/unordered_multimap (C++) sau mapeaza catre o lista in Java (ex: Map>).

Complexity Cheat Sheet

Structura	top/peek	push/add	pop/poll	insert/find/erase
Priority Queue	O(1)	O(log n)	O(log n)	-
Map (ordonat)	-	-	-	O(log n)
Unordered Map	-	-	-	O(1) amortizat; O(n) worst