

# Level 0 Training

---

## Session 5

## STL, Part 2

Set, multiset, ordered set,  
map (ordered), and unordered map

Find by key. Keep the right information.  
Understand order, duplicates, and cost.

# How to use this reference

This book is a reference for studying **after the session**. It explains what each tool stores, why its operations work, and how to choose between tools. Read the explanation before treating the final tables as a revision sheet.

## A practical study routine

1. **Name the information.** Are you storing a sequence, a collection of distinct values, repeated copies, or a value attached to a key?
2. **Trace a small case.** Write the complete state after each operation. Keep duplicates when the tool keeps them. Record both columns of a map.
3. **Predict and test.** Run a complete example, change one input, and explain the changed output. Include an empty collection, an absent key, a repeated key, and equal values.
4. **Solve independently.** Try the problem before its hint. Read the solution after a genuine attempt; later, close the book and write your own version from the idea.

## How the examples are labelled

A **complete program** includes headers and `main` and runs independently. A **fragment** illustrates a smaller idea, with its placement stated. A **wrong example** is deliberately incorrect, not a template to copy.

**Pause and predict** questions have explained answers nearby; cover the answer first. Sections labelled **Optional extension** are for a second reading. The main ordered-set sections are part of this session; their deeper duplicate-handling extension is optional.

### C++23 and the GNU extension

The ordinary examples use standard C++23. Programs labelled **GNU PBDS** additionally require GNU libstdc++ extension headers. C++23 alone does not guarantee those headers exist. Every complete listing comes from an included `.cpp` source file; compile one file at a time.

## Read in stages

You should be comfortable with input, loops, strings, pairs, vectors, functions, references, and the basic growth rates from Sessions 1–4. Return to the earlier explanation whenever a prerequisite feels unclear.

Study sets first, then multisets, then maps and hashing, and finally GNU ordered sets. You are not expected to learn every operation in one sitting.

# The Level 0 learning path

| Session | Topics                                                                   |
|---------|--------------------------------------------------------------------------|
| 1       | Intro to problem solving: data types, conditions, and loops.             |
| 2       | Arrays and strings.                                                      |
| 3       | Functions and complexity analysis.                                       |
| 4       | STL, part 1: vector, pair, deque, queue, priority queue, and stack.      |
| 5       | STL, part 2: set, multiset, ordered set, ordered map, and unordered map. |
| 6       | Static range queries: prefix sum, partial sum, and frequency array.      |

## What is included in this edition

This booklet contains **Session 5: STL, Part 2**. Section numbers begin with 5 so they match the collected book. The updated collected edition contains Sessions 1–5.

## What Session 5 will help you do

Keep distinct values or repeated copies, search by a key, find a nearby value in sorted order, maintain counts and other associated information, distinguish ordered lookup from hashing, and answer rank and selection queries with a GNU ordered set.

## Two names to understand before you begin

An **ordered map** in this syllabus is `std::map`: it orders entries by key. The **ordered set** used for rank queries is a GNU PBDS tree alias introduced later. Ordinary `std::set` is also sorted, but it does not provide efficient rank/select member functions.

## Conventions

Inputs follow each task's contract. Text tasks use lowercase ASCII words unless specified otherwise. Complexity assumes fixed-width numeric operations; string-key costs and average-case hashing are labelled separately. Unless otherwise stated, reported storage includes the containers created by the program, not only a few loop variables.

# Contents

|                                                                |          |
|----------------------------------------------------------------|----------|
| How to use this reference                                      | i        |
| The Level 0 learning path                                      | ii       |
| <b>5 STL, Part 2</b>                                           | <b>1</b> |
| 5.1 Keys are not positions                                     | 2        |
| 5.2 Headers, declarations, and the naming trap                 | 3        |
| 5.3 Set construction and insertion                             | 4        |
| 5.4 Membership: contains, find, and count                      | 6        |
| 5.5 Traversal, extremes, and safe iterators                    | 8        |
| 5.6 Erasing safely: keys, iterators, and loops                 | 10       |
| 5.7 Bounds: the first key at or after a threshold              | 12       |
| 5.8 Predecessors, successors, and value ranges                 | 14       |
| 5.9 Multiset: keep repeated copies                             | 16       |
| 5.10 Multiset erasure: one copy or every copy?                 | 18       |
| 5.11 Duplicate ranges, counts, and a bag simulation            | 20       |
| 5.12 Map: attach a value to each key                           | 22       |
| 5.13 Map access: lookup may insert                             | 24       |
| 5.14 Inserting, replacing, and keeping the first value         | 26       |
| 5.15 Traversing maps and changing mapped values                | 28       |
| 5.16 Frequency maps and tie-breaking                           | 30       |
| 5.17 Ordered map bounds search the keys                        | 33       |
| 5.18 Grouping with a map of vectors                            | 35       |
| 5.19 Unordered map: exact-key lookup without sorting           | 37       |
| 5.20 Unordered-map queries and deterministic output            | 38       |
| 5.21 Buckets, reserve, rehashing, and validity                 | 40       |
| 5.22 Choose a container and count its real work                | 42       |
| 5.23 References, copies, and fresh test cases                  | 44       |
| 5.24 GNU ordered sets: sorted order plus rank                  | 46       |
| 5.25 Setting up the GNU order-statistics tree                  | 47       |
| 5.26 Counting smaller keys with <code>order_of_key</code>      | 49       |
| 5.27 Selecting a key with <code>find_by_order</code>           | 51       |
| 5.28 Dynamic ranks: update the set, then query it              | 53       |
| 5.29 Optional extension: ranked duplicates with unique IDs     | 55       |
| 5.30 Worked problem: sell one affordable ticket                | 57       |
| 5.31 Worked problem: two different positions with a target sum | 59       |
| 5.32 Optional extension: composite keys and set combinations   | 61       |
| 5.33 Optional extension: keep only positive frequency entries  | 63       |
| 5.34 Debugging keyed containers and ranked queries             | 65       |
| 5.35 Practice problems: choose the representation first        | 67       |
| 5.36 Trace, tool-choice, and complexity drills                 | 79       |
| 5.37 Hints before solutions                                    | 84       |
| 5.38 Explained practice solutions                              | 89       |
| 5.39 Explained trace and complexity drills                     | 112      |
| 5.40 Quick reference, glossary, and readiness checklist        | 118      |
| 5.41 Technical notes and references                            | 123      |

## SESSION 5

## STL, Part 2

**Set, multiset, ordered set, map (ordered), and unordered map**

A vector answers questions about positions: “What is at index 3?” This session introduces tools that answer questions about **keys**: “Have I seen this value?” or “What score belongs to this name?” Later, an ordered set will also answer “How many values are smaller?”

The most important decision is what information must survive. A set forgets repeated copies. A multiset keeps them. A map attaches information to a key. These are different models of the problem, not interchangeable spellings.

**Five tools, five questions**

| Tool                       | What it stores                                      | Typical question                    |
|----------------------------|-----------------------------------------------------|-------------------------------------|
| <code>set</code>           | Unique keys in sorted order.                        | Is this value present?              |
| <code>multiset</code>      | Sorted keys, including repeated copies.             | Which remaining copy can I remove?  |
| <code>map</code>           | One associated value per key, ordered by key.       | What belongs to this key?           |
| <code>unordered_map</code> | One associated value per key, with no sorted order. | Can I look up this exact key?       |
| GNU ordered set            | Unique sorted keys plus rank information.           | Which key has sorted position $k$ ? |

**Your route through the chapter**

**Stage A: keys and copies.** Learn sets, membership, safe iterators, nearby values, multisets, and the difference between erasing one copy and all copies.

**Stage B: associated information.** Build maps for scores, counts, and groups. Understand when a lookup inserts a new key. Compare ordered lookup with hashing.

**Stage C: rank and independent work.** Use a GNU ordered set for rank/select operations. Then solve the practice, explain your invariant, and count the real work.

**A small drawing is often the best debugger**

For a set, write the sorted keys. For a multiset, write every copy. For a map, draw two columns: key and value. An output that looks surprising often comes from tracing the wrong model.

## 5.1 Keys are not positions

### A question that a vector does not answer directly

Suppose participant IDs arrive as 42 7 42 18. You want to know how many *different* participants appeared. A vector stores four arrivals. A set stores the three distinct IDs: 7 18 42.

The repeated 42 is not another participant in this interpretation. But if these values are ticket prices and there are two tickets costing 42, deleting a repeated value would lose a real ticket. That second interpretation needs multiplicity: the number of copies.

A **key** is the information used to identify or locate an entry. In a set of IDs, the ID is the key and the whole stored value. In a map from name to score, the name is the key and the score is the associated value. Associative containers support lookup by keys.<sup>[5-1]</sup>

### Choose what one entry means

| Input meaning     | One entry means                                | Suitable model                      |
|-------------------|------------------------------------------------|-------------------------------------|
| Attendance        | One different participant.                     | <code>set&lt;int&gt;</code>         |
| Ticket stock      | One physical ticket.                           | <code>multiset&lt;int&gt;</code>    |
| Latest score      | One name and its latest score.                 | <code>map&lt;string, int&gt;</code> |
| Occurrence counts | One value and the number of times it appeared. | <code>map&lt;int, int&gt;</code>    |

The short type spellings in this table omit `std::` for space; complete programs use the full names and their headers.

### Sorted order is not arrival order

If the input order matters, preserve it separately in a vector. A map ordered by name cannot tell you who arrived first merely by looking at `begin()`. A set does not remember which occurrence of a repeated integer arrived first.

#### Pause and predict – 1

Values arrive as 5 2 5 2 9. How many entries belong in a set of seen values, a multiset of arrivals, and a map from value to occurrence count?

**Check your prediction.** The set has 3 entries; the multiset has 5. The frequency map has 3 entries: 2 -> 2, 5 -> 2, and 9 -> 1. A map's size counts keys, not the sum of their stored counts.

## 5.2 Headers, declarations, and the naming trap

### Use the header for the tool

| Header                             | Facilities used here                                                           |
|------------------------------------|--------------------------------------------------------------------------------|
| <code>&lt;set&gt;</code>           | <code>std::set</code> , <code>std::multiset</code> .                           |
| <code>&lt;map&gt;</code>           | <code>std::map</code> .                                                        |
| <code>&lt;unordered_map&gt;</code> | <code>std::unordered_map</code> .                                              |
| <code>&lt;string&gt;</code>        | <code>std::string</code> for text keys.                                        |
| <code>&lt;utility&gt;</code>       | <code>std::pair</code> for combined keys or results.                           |
| <code>&lt;functional&gt;</code>    | <code>std::less</code> , <code>std::greater</code> .                           |
| <code>&lt;iterator&gt;</code>      | <code>std::next</code> , <code>std::prev</code> , <code>std::distance</code> . |

The GNU ordered-set headers are introduced in their own section. They are not required for the four standard container types above.<sup>[5-3][5-5][5-9][5-10]</sup>

**Fragments – inside a function, after including the required headers:**

```
std::set<int> ids;
std::multiset<int> prices;
std::map<std::string, int> score;
std::unordered_map<int, int> frequency;
```

For a map, the first type is the **key type** and the second is the **mapped type**. In `score`, a key is a string and its associated value is an integer.

### “Ordered map” means `std::map` here

The syllabus uses *ordered map* to contrast it with `unordered_map`. Write `std::map`, not `std::ordered_map`. The entries are ordered by key, not by the mapped value. A map with name `keys` is not automatically a score ranking.<sup>[5-6]</sup>

### “Ordered set” can mean two different things

Ordinary `std::set` already keeps keys sorted. In competitive-programming discussions, *ordered set* often names an alias for a GNU PBDS tree that adds rank/select operations. This book uses `OrderedSet` for that explicitly configured extension, not for a standard class named `std::ordered_set`.<sup>[5-20]</sup>

#### Pause and predict – 2

In `std::map<int, std::string> label;`, which type is the key? Does `label[100]` mean “the element at index 100”?

**Check your prediction.** The key type is `int`; the associated value is a string. The expression uses the *key* 100, not a sequence position. It can also insert that key if it is missing, as a later section explains.

## 5.3 Set construction and insertion

### One copy of each key

A `std::set<int>` stores distinct integers in ascending order with its default comparison. A duplicate insertion leaves the existing collection unchanged.<sup>[5-2]</sup>

```
// Fragment -- inside a function; include <set>.  
std::set<int> empty;  
std::set<int> values{8, 3, 8, 1, 3};  
values.insert(5);  
values.insert(3); // Already present: no additional entry.
```

After these statements, `values` contains 1 3 5 8. There is no separate “sort the set” step. There is also no `push_back`: the collection chooses the position according to its ordering rule.

### Insertion also tells you whether something changed

The ordinary single-value `set.insert(x)` returns a pair. Its `first` field is an iterator to the stored key; its `second` field is `true` exactly when a new key was inserted.<sup>[5-1]</sup>

```
auto result = seen.insert(x);  
if (result.second) {  
    // x was not previously in the set.  
}
```

This avoids making membership testing and insertion two separate operations when you need both. One line can be short because the library does work, not because the operation is free.

#### Pause and predict – 3

Insert 4, then 4, then 1 into an empty set. What are the three insertion flags, the final size, and the final traversal order?

**Check your prediction.** The flags are `true`, `false`, and `true`. The final size is 2, and traversal visits 1 4. Insertion order did not become traversal order.

**Why it works.** After processing the first  $i$  values, `seen` contains exactly their distinct keys. Inserting the next value either adds a new key or finds the existing one, so the flag and size describe the next prefix correctly.

**Cost.** With  $d$  distinct keys, one ordinary insertion takes  $O(\log(d + 1))$  work. All  $n$  insertions take  $O(n \log(n + 1))$  as a simple upper bound, with  $O(d)$  stored keys.

**Complete program – duplicates disappear from a set**

```
#include <iostream>
#include <set>

int main() {
    std::set<int> values{8, 3, 8, 1, 3};
    std::cout << values.size() << '\n';
    bool first = true;
    for (int x : values) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    std::cout << '\n';
}
```

**Sample input****Sample output**

(no input)

3

1 3 8

The initializer lists five integers, but the set stores only three. The range loop visits each stored key once. The `first` flag prints separators between values, not before the first value.

**Task.** Read  $0 \leq n \leq 200000$ , then  $n$  integers in  $[-10^9, 10^9]$ . After each arrival, print **NEW** if it is the first occurrence, otherwise **OLD**; also print the number of distinct keys seen so far.

**Complete program – identify first occurrences**

```
#include <iostream>
#include <set>

int main() {
    int n = 0;
    std::cin >> n;
    std::set<int> seen;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        auto result = seen.insert(x);
        std::cout << (result.second ? "NEW" : "OLD")
                  << ' ' << seen.size() << '\n';
    }
}
```

**Sample input****Sample output**

5

NEW 1

7 2 7 7 4

NEW 2

OLD 2

OLD 2

NEW 3

## 5.4 Membership: contains, find, and count

### Choose the result you actually need

| Expression                 | Meaning for a set                                                    |
|----------------------------|----------------------------------------------------------------------|
| <code>s.contains(x)</code> | A truth value: is <code>x</code> present?                            |
| <code>s.find(x)</code>     | An iterator to <code>x</code> , or <code>s.end()</code> when absent. |
| <code>s.count(x)</code>    | The number of matching entries: 0 or 1 for a set.                    |

These operations do not insert missing keys. The `contains` spelling is available in the C++23 target used by this book.<sup>[5-3]</sup>

**Fragments – inside a function; `s` is a set:**

```
if (s.contains(7)) {
    std::cout << "YES\n";
}
auto it = s.find(7);
if (it != s.end()) {
    std::cout << *it << '\n';
}
```

`*it` accesses the value at an iterator. You can only do that when the iterator points to an actual element. The `end()` marker means “past all elements,” not “the last element.”

### Do not compare an iterator with an integer

**Wrong fragments – do not copy:**

```
if (s.find(7) == 7) { }    // Iterator is not the key.
if (s.find(7) == -1) { }   // Absence is not encoded as -1.
std::cout << *s.find(99);  // Unsafe if 99 is absent.
```

Use `it == s.end()` to test absence. Use `contains` for a yes/no question. Keeping those two ideas separate prevents a large class of bugs.

#### Pause and predict – 4

A set contains `-1 0 4`. What does `find(-1)` return? How do you distinguish that valid key from a failed lookup?

**Check your prediction.** It returns an iterator to the stored key `-1`. Compare the iterator with `end()` to detect failure. A key’s numerical value has no role in that test.

## A list of membership queries

**Task.** Read  $n$   $q$ , then  $n$  integers and  $q$  query integers. Print YES or NO for each query. Here  $0 \leq n, q \leq 200000$ ; stored keys lie in  $[-10^9, 10^9]$ , and queries may lie in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ .

**Complete program – query without changing the collection**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        std::cout << (values.contains(x) ? "YES" : "NO")
                  << '\n';
    }
}
```

### Sample input

4 3  
9 2 9 5  
9  
3  
2

### Sample output

YES  
NO  
YES

## Trace what changes, and what does not

The input creates the set 2 5 9. Asking about 3 prints NO; it does not add 3. The final query about 2 still sees exactly the original set.

**Cost.** Building by repeated insertion and answering all queries takes  $O((n + q) \log(n + 1))$  as an upper bound. The collection stores  $O(d)$  keys, where  $d \leq n$ . There is no need to store the entire query list.

### Membership is not the same as frequency

A set can tell you that 9 appeared. It cannot tell you whether 9 appeared once or a thousand times: repeated insertions do not create repeated entries. Use a multiset or a map of counts when multiplicity matters.

## 5.5 Traversal, extremes, and safe iterators

### Walk through the sorted keys

Equivalent fragments – inside a function, for a set `s`:

```
for (int x : s) {
    std::cout << x << '\n';
}
for (auto it = s.begin(); it != s.end(); ++it) {
    std::cout << *it << '\n';
}
```

The first form is shorter when you only need the values. The second exposes iterator positions, which are useful for bounds and erasure. Both visit the keys in the order defined by the set's comparison.

### An iterator is not an integer index

Set iterators are bidirectional: you can advance to the next entry and, when a predecessor exists, move to the previous one. They are not random-access vector iterators.<sup>[5-1]</sup>

```
// Wrong fragments for a set:
// std::cout << s[2];
// auto it = s.begin() + 2;
// int index = s.find(7) - s.begin();
```

A set has no `operator[]` and no constant-time “jump to index” interface. `std::next(s.begin(), k)` can walk to an existing rank, but it performs `k` iterator steps. It is not a hidden  $O(1)$  index operation.<sup>[5-15]</sup>

### The smallest and largest stored keys

For a nonempty ascending set, `*s.begin()` is the smallest key and `*s.rbegin()` is the largest. Reverse traversal starts at the last element and moves in the opposite order.<sup>[5-3][5-14]</sup>

For an empty set, `begin() == end()`. Do not dereference either marker. Do not decrement `begin()`; there is no valid entry before the first one.

#### Pause and predict – 5

For a nonempty set 2 6 9, is `*s.end()` equal to 9? Is `std::next(s.begin(), 2)` a constant-time operation?

**Check your prediction.** No: dereferencing `end()` is invalid. Use `*s.rbegin()` for 9. The `next` call performs two iterator increments here; in general its work grows with the requested number of steps.

## Extremes with an explicit empty case

**Task.** Read  $0 \leq n \leq 200000$  integers in  $[-10^9, 10^9]$ . Print the smallest and largest distinct keys, or EMPTY when  $n=0$ .

**Complete program – do not access an empty set**

```
#include <iostream>
#include <set>

int main() {
    int n = 0;
    std::cin >> n;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    if (values.empty()) {
        std::cout << "EMPTY\n";
    } else {
        std::cout << *values.begin() << ' '
                  << *values.rbegin() << '\n';
    }
}
```

**Sample input**

**Sample output**

|                |       |
|----------------|-------|
| 5              | -9 -2 |
| -8 -2 -5 -2 -9 |       |

## A set key cannot be changed in place

An iterator into a set exposes a read-only key. Changing a stored 2 into 100 without repositioning it would break the ordering rules. To replace a key, erase the old one and insert the new one.<sup>[5-2]</sup>

```
// Fragment -- inside a function; s is a set<int>.
auto it = s.find(2);
if (it != s.end()) {
    s.erase(it);
    s.insert(100);
}
```

If 100 already exists, the final insertion does not create another copy. A replacement can therefore reduce the set's size. This is correct set behavior, not a failure of insertion.

**Cost of the example.** Building takes  $O(n \log(n+1))$ , the two endpoint accesses take  $O(1)$ , and stored keys take  $O(d)$  space. If you only need a minimum and maximum once, a streaming scan from Session 1 is simpler and uses constant extra space; the set is used here to practise its interface.

## 5.6 Erasing safely: keys, iterators, and loops

### Erasing by key does not require the key to exist

Fragments – inside a function; `s` is a set:

```
auto removed = s.erase(7); // Number erased: 0 or 1.
s.clear();                // Remove all elements.
bool isEmpty = s.empty(); // Check only; erase nothing.
```

A missing key makes `erase(key)` do nothing and return zero. By contrast, `erase(iterator)` requires an iterator that points to an existing element. Passing `end()` is invalid.<sup>[5-1]</sup>

```
auto it = s.find(7);
if (it != s.end()) {
    s.erase(it);
    // Do not use it again: its element was erased.
}
```

For these ordered standard containers, inserting a key does not invalidate existing element iterators. Erasing an entry invalidates iterators and references to that entry, not to the other surviving entries.<sup>[5-1]</sup>

### The returned iterator tells a loop where to continue

`s.erase(it)` returns an iterator to the next surviving entry, or `end()` when there is none. A removal loop must choose *either* that returned iterator *or* an increment; doing both can skip an entry.

```
for (auto it = s.begin(); it != s.end(); ) {
    if (*it < 0) it = s.erase(it);
    else ++it;
}
```

The loop has no increment in its header because the body controls progress. After deleting the current entry, the returned iterator already points to the next candidate.

#### Pause and predict – 6

Why is `s.erase(s.find(100));` unsafe when 100 might be absent, even though `s.erase(100);` is safe in that case?

**Check your prediction.** The first expression passes an iterator. A failed `find` returns `end()`, which is not an erasable element. The second passes a key; its interface explicitly allows a missing key and then erases nothing.

## Remove all negative distinct values

**Task.** Read  $0 \leq n \leq 200000$  integers in  $[-10^9, 10^9]$ . Store their distinct values in a set, erase negative keys with an iterator loop, and print the remaining keys in ascending order. Print a blank line when none remain.

**Complete program – a loop that remains valid after erasure**

```
#include <iostream>
#include <set>

int main() {
    int n = 0;
    std::cin >> n;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    for (auto it = values.begin(); it != values.end(); ) {
        if (*it < 0) it = values.erase(it);
        else ++it;
    }
    bool first = true;
    for (int x : values) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    std::cout << '\n';
}
```

### Sample input

6  
-2 4 -1 0 4 8

### Sample output

0 4 8

**Trace.** The constructed set is -2 -1 0 4 8. Erasing -2 returns the iterator to -1; erasing -1 returns the iterator to 0. The loop then increments past 0, 4, and 8 without changing them.

**Cost.** Building dominates at  $O(n \log(n+1))$ . The erasure/traversal pass is linear in the number of stored keys in amortized total work: erasing by an already-known iterator has amortized constant cost in these ordered standard containers. A key-based erase would also need lookup.<sup>[5-1]</sup>

### Do not generalize this erase return type to every extension

The pattern above applies to `std::set`, `std::multiset`, `std::map`, and the corresponding standard iterator-erasure interface. GNU PBDS has its own API. The GNU examples in this chapter erase by key and do not assume identical return types.

## 5.7 Bounds: the first key at or after a threshold

### Two questions that differ at equality

For an ascending set, `lower_bound(x)` finds the first key  $\geq x$ , while `upper_bound(x)` finds the first key  $> x$ . Each returns `end()` when no matching key exists. They find a *position*, not a number of elements.<sup>[5-1]</sup>

| Query | lower bound in {2,5,9} | upper bound        |
|-------|------------------------|--------------------|
| 1     | 2                      | 2                  |
| 2     | 2                      | 5                  |
| 5     | 5                      | 9                  |
| 8     | 9                      | 9                  |
| 9     | 9                      | <code>end()</code> |
| 10    | <code>end()</code>     | <code>end()</code> |

The two results differ only when the queried key exists. For a missing key such as 8, both searches stop at the same next key, 9.

### Use the member operation

```
// Preferred for a set, inside a function:
auto it = values.lower_bound(x);
```

This uses the set's key-search interface. A generic call such as `std::lower_bound(values.begin(), values.end(), x)` can make logarithmically many comparisons but still need linear iterator movement on a set. For an ordered associative container, use its member bound operation.<sup>[5-16]</sup>

### Check before dereferencing

```
auto it = values.lower_bound(x);
if (it == values.end()) std::cout << "NONE\n";
else std::cout << *it << '\n';
```

A query above the maximum is an ordinary input case. It should not produce an invalid access or a made-up numerical answer.

#### Pause and predict – 7

For keys 3 6 10, what do the lower and upper bounds of 6 return? What about 7?

**Check your prediction.** At 6 they return 6 and 10. At 7 both return 10. A lower bound may equal the query; an upper bound must come strictly after it in the ascending order.

## Compare both bounds in one program

**Task.** Read  $n$   $q$ ,  $n$  stored integers, and  $q$  queries. For every query, print the lower-bound key and upper-bound key, using NONE for a missing result. Bounds are as in the membership example.

**Complete program – equality changes the answer**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        auto low = values.lower_bound(x);
        auto high = values.upper_bound(x);
        if (low == values.end()) std::cout << "NONE";
        else std::cout << *low;
        std::cout << ' ';
        if (high == values.end()) std::cout << "NONE";
        else std::cout << *high;
        std::cout << '\n';
    }
}
```

| Sample input | Sample output |
|--------------|---------------|
| 4 4          | 2 2           |
| 2 5 9 5      | 5 9           |
| 1            | 9 9           |
| 5            | 9 NONE        |
| 8            |               |
| 9            |               |

**Why it works.** The set contains every distinct input key. Its lower-bound search returns the smallest acceptable key for the first condition; its upper-bound search returns the smallest acceptable key for the stricter condition. Neither operation changes the set.

**Cost.** Two searches per query are still  $O(\log(d+1))$ , not  $O(\log^2 d)$ : consecutive costs add. The whole program takes  $O((n+q)\log(n+1))$  time and  $O(d)$  stored keys.

## 5.8 Predecessors, successors, and value ranges

### Four useful threshold questions

For ascending keys, the first two are direct bounds. The last two require stepping back from a bound, but only when there is an earlier element.

| Wanted key            | Safe starting idea                                                                  |
|-----------------------|-------------------------------------------------------------------------------------|
| Smallest key $\geq x$ | <code>lower_bound(x)</code> ; check against <code>end()</code> .                    |
| Smallest key $> x$    | <code>upper_bound(x)</code> ; check against <code>end()</code> .                    |
| Largest key $< x$     | Start at <code>lower_bound(x)</code> ; step back only if not <code>begin()</code> . |
| Largest key $\leq x$  | Start at <code>upper_bound(x)</code> ; step back only if not <code>begin()</code> . |

**Task.** Read  $n$   $q$ , stored integers, and queries. Print the greatest stored key not exceeding each query, or `NONE`. Use the same bounds as the membership example.

**Complete program – a guarded step to the previous key**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q-- > 0) {
        int x = 0;
        std::cin >> x;
        auto it = values.upper_bound(x);
        if (it == values.begin()) {
            std::cout << "NONE\n";
        } else {
            --it;
            std::cout << *it << '\n';
        }
    }
}
```

| Sample input | Sample output |
|--------------|---------------|
| 3 4          | NONE          |
| 2 5 9        | 5             |
| 1            | 5             |
| 5            | 9             |
| 8            |               |
| 20           |               |

## Why the guard also handles an empty set

If `upper_bound(x) == begin()`, there is no earlier key to select. This can mean the set is empty, or that every key is larger than `x`. Both situations require `NONE`.

If the bound equals `end()` *and the set is nonempty*, stepping back is valid and reaches the largest key. The relevant guard for a backward step is therefore `it != begin()`, not merely `it != end()`.

### Pause and predict – 8

Does `std::distance(s.begin(), s.lower_bound(x))` count keys smaller than `x`? Is the entire expression logarithmic?

**Check your prediction.** For an ascending set, it counts them correctly. But `distance` walks the set iterators, so the expression can take linear time. Correctness does not imply the desired efficiency. GNU rank queries solve this later.

## List keys inside an inclusive value interval

For  $L \leq R$ , the interval  $[L, R]$  corresponds to the iterator range from `lower_bound(L)` up to, but not including, `upper_bound(R)`. Notice that this is a range of *values*, not indices.

**Task.** Read  $n$   $L$   $R$ , then  $n$  integers. Print the distinct input keys inside  $[L, R]$  in ascending order. Here  $0 \leq n \leq 200000$ ,  $L \leq R$ , and keys and endpoints lie in  $[-10^9, 10^9]$ .

**Complete program – visit a value interval**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, left = 0, right = 0;
    std::cin >> n >> left >> right;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    auto finish = values.upper_bound(right);
    bool first = true;
    for (auto it = values.lower_bound(left);
         it != finish; ++it) {
        if (!first) std::cout << ' ';
        std::cout << *it;
        first = false;
    }
    std::cout << '\n';
}
```

### Sample input

```
7 3 8
9 3 6 3 1 8 10
```

### Sample output

```
3 6 8
```

**Cost after building.** Locating both endpoints costs  $O(\log(d+1))$ ; printing  $k$  results costs  $O(k)$ . The query is  $O(\log(d+1) + k)$ , not merely logarithmic.

## 5.9 Multiset: keep repeated copies

### Equal values can represent different items

If three tickets all cost 5, they are still three tickets. A `std::multiset<int>` retains repeated copies and visits them in sorted order. Each insertion adds an element, even when an equal key is already present.<sup>[5-4]</sup>

```
// Fragment -- inside a function; include <set>.
std::multiset<int> bag{5, 2, 5, 7, 5};
std::cout << bag.size() << '\n';      // 5 entries
std::cout << bag.count(5) << '\n';   // 3 matching copies
```

The sorted sequence is 2 5 5 5 7. The size is not the number of distinct values. The three copies of 5 are separate stored entries even though their integer values are equal.

**Task.** Read  $0 \leq n \leq 200000$  integers in  $[-10^9, 10^9]$ . Print the multiset size and then every stored copy in ascending order on the next line.

**Complete program – preserve multiplicity**

```
#include <iostream>
#include <set>

int main() {
    int n = 0;
    std::cin >> n;
    std::multiset<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    std::cout << values.size() << '\n';
    bool first = true;
    for (int x : values) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    std::cout << '\n';
}
```

#### Sample input

5  
5 2 5 7 5

#### Sample output

5  
2 5 5 5 7

## Operations that look familiar

A multiset provides `insert`, `find`, `contains`, `count`, `begin`, `end`, `lower_bound`, `upper_bound`, `erase`, `clear`, `size`, and `empty`. Its keys are still read-only through iterators.<sup>[5-5]</sup>

`find(x)` gives an iterator to a matching element, or `end()`. It is enough to remove one arbitrary copy of an integer value. Do not use it to assume a particular position within a run of equal keys; use `lower_bound(x)` for the first matching position in sorted traversal.

## Insertion returns a different shape

For a set, single-value insertion returns `pair<iterator, bool>` because a new entry may or may not be created. For a multiset, the same ordinary form returns an *iterator*: every insertion creates a new entry.<sup>[5-1]</sup>

```
// Fragments -- s is set<int>, bag is multiset<int>.
auto uniqueResult = s.insert(5);
bool wasNew = uniqueResult.second;
auto copyPosition = bag.insert(5);
std::cout << *copyPosition << '\n';
```

Do not copy the `.second` insertion test from set code into multiset code. Similar operation names do not guarantee identical return types.

## When a frequency map will be more convenient

A multiset physically represents each copy. A map from value to count stores one record per distinct value. If you only need counts, a frequency map can avoid storing a separate node for every occurrence. If you repeatedly select an individual smallest or nearby copy, a multiset expresses that behavior directly.

### Pause and predict – 9

A multiset starts as 4 4 7. After inserting 4, what are its size and `count(4)`? Does insertion report “not inserted” because 4 existed?

**Check your prediction.** Its size is 4 and the count is 3. Insertion creates another copy and returns an iterator to it. There is no failed-duplicate insertion flag for this ordinary multiset operation.

## 5.10 Multiset erasure: one copy or every copy?

### The argument selects the behavior

`bag.erase(x)` removes **all** copies equivalent to the key `x` and returns how many were erased. `bag.erase(it)` removes **one** element at a valid iterator.<sup>[5-1][5-5]</sup>

```
// Fragment -- erase one copy if it exists.
auto it = bag.find(x);
if (it != bag.end()) bag.erase(it);

// Different task: erase every copy of x.
auto removed = bag.erase(x);
```

For 2 5 5 5 7, erasing one found 5 leaves 2 5 5 7. Erasing by the key 5 leaves 2 7. The distinction changes the answer to stock and ticket problems.

#### Pause and predict – 10

Two tickets cost 4 and one costs 9. After selling one 4, how many tickets should remain? Which erasure form matches that action?

**Check your prediction.** Two tickets remain: one 4 and one 9. Find a 4, check the iterator, and erase that iterator. `erase(4)` would incorrectly sell both 4-priced tickets at once.

### Run both operations, in order

This program takes no input. It removes one 5, prints the remaining count, removes every remaining 5, then prints the number removed and the final count.

#### Complete program – same name, different removal scope

```
#include <iostream>
#include <set>

int main() {
    std::multiset<int> bag{2, 5, 5, 5, 7};
    auto it = bag.find(5);
    if (it != bag.end()) bag.erase(it);
    std::cout << bag.count(5) << '\n';
    auto removed = bag.erase(5);
    std::cout << removed << '\n';
    std::cout << bag.count(5) << '\n';
}
```

#### Sample input

(no input)

#### Sample output

2  
2  
0

## No copy found means nothing to erase

The guarded one-copy pattern is safe for an empty multiset and for an absent key. A call such as `bag.erase(bag.find(x))` omits that guarantee: a missing key produces `end()`, not an erasable element.

## Trace the guard on three states

| Before removing one 5 | What <code>find(5)</code> returns | State after the guarded removal                   |
|-----------------------|-----------------------------------|---------------------------------------------------|
| Empty                 | <code>end()</code>                | Empty; no erasure is attempted.                   |
| 2, 7                  | <code>end()</code>                | 2, 7; a missing key is not an error in this task. |
| 2, 5, 5, 7            | An iterator to one 5              | 2, 5, 7; the other occurrence survives.           |

The branch has two separate jobs. The search determines whether the occurrence exists. The guard determines whether the returned iterator is an element that can legally be erased. The fact that `erase(key)` tolerates an absent key does not make `erase(end())` a valid iterator operation.

## Reuse the rule in a small helper

**Fragment** – define above `main`; include `<set>`:

```
bool eraseOne(std::multiset<int>& bag, int x) {
    auto it = bag.find(x);
    if (it == bag.end()) return false;
    bag.erase(it);
    return true;
}
```

The reference parameter lets the helper modify the caller's bag. Its boolean result reports whether anything was removed; printing is left to the caller. The early return protects the erasure and handles an empty bag without a separate empty-case branch.

With `bag = {5, 5}`, three calls to `eraseOne(bag, 5)` return `true`, `true`, and `false`. The sizes become 1, 0, and 0. Trace that sequence before trying it: a removal request is not a promise that an occurrence exists.

## 5.11 Duplicate ranges, counts, and a bag simulation

### Bounds isolate all equal copies

For 2 5 5 5 9, `lower_bound(5)` points to the first 5; `upper_bound(5)` points to 9. The half-open range between them contains exactly the three 5s. `equal_range(5)` returns those two bounds as a pair.<sup>[5-1]</sup>

```
// Fragment -- inside a function; bag is a multiset<int>.
auto bounds = bag.equal_range(5);
for (auto it = bounds.first; it != bounds.second; ++it) {
    std::cout << *it << '\n';
}
```

For a missing key, the range is empty: the two iterators are equal. Both may be `end()`, and that is safe as long as you do not dereference them.

### Counting duplicates is not always logarithmic

If  $k$  copies match, `bag.count(x)` has  $O(\log(n+1) + k)$  complexity. The matching range may need to be traversed to count its entries. A yes/no query can use `contains` or `find` instead.<sup>[5-1]</sup>

Calling `count(5)` a hundred thousand times on a bag containing a hundred thousand 5s is not a cheap repeated lookup. It can revisit the same large run on every call. A frequency map stores the count directly when count queries are the real task.

### State the command contract before coding

An initially empty multiset supports four commands:

| Command | Action                                              |
|---------|-----------------------------------------------------|
| 1 x     | Add one copy of x.                                  |
| 2 x     | Remove one copy if present; otherwise do nothing.   |
| 3 x     | Print the number of copies of x.                    |
| 4       | Print the smallest key, or EMPTY. Do not remove it. |

For this teaching simulation,  $0 \leq q \leq 5000$  and keys lie in  $[-10^9, 10^9]$ . The smaller command bound is deliberate because repeated multiset counting can be expensive.

**Cost.** Ordinary updates and membership-style search are logarithmic. Across count queries, add the numbers of matching copies actually counted. With at most  $q$  live entries, a simple worst-case upper bound for the whole simulation is  $O(q^2)$ ; do not label it  $O(q \log q)$  without accounting for those counts.

#### Pause and predict – 11

When you only need to know whether a multiset contains 5, why can `contains(5)` be a better choice than `count(5) > 0`?

**Check your prediction.** Both give the same truth value, but counting may traverse every matching copy. Membership only needs to locate a match, not count all matches.

## Complete program – command arguments and duplicate state

```

#include <iostream>
#include <set>

int main() {
    int q = 0;
    std::cin >> q;
    std::multiset<int> bag;
    while (q--> {
        int type = 0;
        std::cin >> type;
        if (type == 4) {
            if (bag.empty()) std::cout << "EMPTY\n";
            else std::cout << *bag.begin() << '\n';
            continue;
        }
        int x = 0;
        std::cin >> x;
        if (type == 1) bag.insert(x);
        else if (type == 2) {
            auto it = bag.find(x);
            if (it != bag.end()) bag.erase(it);
        } else {
            std::cout << bag.count(x) << '\n';
        }
    }
}

```

## Sample input

## Sample output

|     |       |
|-----|-------|
| 8   | EMPTY |
| 4   | 1     |
| 1 5 | 5     |
| 1 5 | 0     |
| 2 5 |       |
| 3 5 |       |
| 2 8 |       |
| 4   |       |
| 3 8 |       |

## Trace the important changes

The first command sees an empty bag. Two insertions create two copies of 5. Removing once leaves one copy, so the count query prints 1. Removing absent 8 changes nothing. The minimum query prints 5 without removing it.

Command 4 has no extra argument. Reading an integer after every command unconditionally would consume part of the next command. The `continue` keeps the input parser aligned with the contract.

## 5.12 Map: attach a value to each key

### A two-column collection

A `std::map<Key, Value>` stores one associated value for each unique key. A name-to-score map might contain these entries:

| Key: name | Value: score |
|-----------|--------------|
| ali       | 0            |
| aya       | 95           |
| nour      | 70           |

The *keys* are unique, but the scores need not be. Two different names may both map to 70. The default map visits entries in ascending key order, which for these lowercase words is lexicographical order. It does not sort by score.<sup>[5-6]</sup>

**Fragments – inside a function; include `<map>` and `<string>`:**

```
std::map<std::string, int> scores;
scores["aya"] = 70;
scores["ali"] = 0;
scores["aya"] = 95;
```

The last assignment changes Aya's existing score. It does not create a second "aya" record. The map has two entries, not three.

### The key selects the record

`scores["aya"]` does not mean a position in an array. It means the value associated with the exact key "aya". Spaces and capitalization are part of a string key, so a problem's input convention matters. These examples use one lowercase ASCII word per name.

### Decide the repeated-key policy

"Store a score" is incomplete unless you know what happens when the same name appears again. A problem might ask for the latest score, the first score, the greatest score, a total, or a list of all scores. The container will not infer that policy for you.

For *latest score*, assignment replaces the associated value. For *total score*, add to it. For *first score*, insert only when absent. We will implement all three patterns in this chapter.

#### Pause and predict – 12

After `scores["ali"] = 40;` and `scores["aya"] = 40;;`, does the map contain one entry because the values are equal?

**Check your prediction.** No. It contains two entries because the keys are different. Map uniqueness applies to the key, not to the associated value.

## A directory with latest-value queries

**Task.** Read  $n$   $q$ , followed by  $n$  `name score` records and  $q$  query names. Keep the last score for each name. Print the score for a known name, or `MISSING` for an unknown one. Here  $0 \leq n, q \leq 200000$ , names have 1–20 lowercase ASCII letters, and scores lie in  $[-10^9, 10^9]$ .

**Complete program – a missing record is not a zero score**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::map<std::string, int> scores;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int score = 0;
        std::cin >> name >> score;
        scores[name] = score;
    }
    while (q--) {
        std::string name;
        std::cin >> name;
        auto it = scores.find(name);
        if (it == scores.end()) std::cout << "MISSING\n";
        else std::cout << it->second << '\n';
    }
}
```

### Sample input

```
3 3
aya 70
ali 0
aya 95
aya
ali
noor
```

### Sample output

```
95
0
MISSING
```

## Read an entry through its iterator

The iterator points to a key-value pair. `it->first` is the key; `it->second` is the associated value. The arrow notation is a convenient way to access a field through an iterator: `it->second` means the same as `(*it).second` here.

The query loop checks `end()` before reading `second`. It correctly distinguishes Ali’s existing score 0 from an absent record. Treating zero as “missing” would lose that distinction.

**Cost.** With bounded-length keys, the program takes  $O((n + q) \log(n + 1))$  time and  $O(d)$  entries, where  $d$  is the number of distinct names. With unbounded text, include the work of comparing and storing key characters.

## 5.13 Map access: lookup may insert

### Square brackets mean access-or-create

For `map<string, int>`, `m[key]` returns the associated integer by reference. If the key is missing, it first creates an entry with an integer value of zero. This is useful for counting, but surprising when you only intended to read.<sup>[5-7][5-8]</sup>

```
// Fragment -- inside a function; scores is map<string, int>.
std::cout << scores["unknown"] << '\n';
// The map now contains "unknown", even though this looked like output.
```

The default associated value depends on its type: zero for `int`, an empty string for `std::string`, or an empty vector for `std::vector<int>` in these standard declarations.

### Compare the access tools

| Expression                   | When present            | When absent                                              |
|------------------------------|-------------------------|----------------------------------------------------------|
| <code>m[key]</code>          | Reference to the value. | Inserts a default value, then returns its reference.     |
| <code>m.at(key)</code>       | Reference to the value. | Throws <code>std::out_of_range</code> ; inserts nothing. |
| <code>m.find(key)</code>     | Iterator to the entry.  | Returns <code>end()</code> .                             |
| <code>m.contains(key)</code> | <code>true</code> .     | <code>false</code> .                                     |
| <code>m.count(key)</code>    | 1.                      | 0.                                                       |

The last four operations do not create missing keys. `count` counts entries matching the key, *not* a frequency integer stored as the associated value.<sup>[5-7][5-9]</sup>

### Use at when presence is already guaranteed

If you just inserted every array value into a frequency map, looking up one of those same values with `at` is safe. When arbitrary query keys may be absent, `find` makes the missing case explicit without requiring exception handling.

#### Pause and predict – 13

A frequency map stores `frequency[7] = 12`. What are `frequency.count(7)` and `frequency.at(7)`?

**Check your prediction.** They are 1 and 12. There is one entry whose key is 7, and its associated value is the count 12. Confusing these two meanings of “count” gives the wrong answer.

## A small program exposes the side effect

This program takes no input. Predict every line before running it.

**Complete program – a read-looking expression grows the map**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> scores;
    std::cout << scores.size() << '\n';
    std::cout << scores["nour"] << '\n';
    std::cout << scores.size() << '\n';
    auto it = scores.find("omar");
    std::cout << (it == scores.end()) << '\n';
    std::cout << scores.size() << '\n';
}
```

| Sample input | Sample output |
|--------------|---------------|
| (no input)   | 0             |
|              | 0             |
|              | 1             |
|              | 1             |
|              | 1             |

## Explain each line of output

The initial size is zero. Reading `scores["nour"]` creates a zero-valued entry and prints zero. The next size is therefore one. Finding `"omar"` returns `end()`, so the comparison prints one for `true`. The map size stays one because `find` did not insert Omar.

## A truth test is not a presence test

**Wrong approach when zero is a legitimate stored value:**

```
if (scores[name]) {
    std::cout << "FOUND\n";
}
```

This checks whether an integer is nonzero, not whether the key exists. It also inserts the key when missing. Use `contains(name)` to ask about existence, or `find(name)` when you need the stored value too.

### Read-only maps do not provide operator[]

A `const` map cannot allow a missing-key lookup to modify its contents. Use `find`, `contains`, `count`, or the `const` overload of `at` when writing read-only helper functions. We will apply this with `const` references later.

## 5.14 Inserting, replacing, and keeping the first value

### The repeated-key rule belongs in the code

Fragments – inside a function; `m` is a map:

```
m[key] = value;           // Insert or replace.
auto result = m.insert({key, value}); // Insert only if absent.
```

The ordinary single-entry `insert` returns a pair: the iterator to the matching entry and a flag indicating whether a new entry was created. If the key already exists, its old associated value is retained.<sup>[5-1]</sup>

### Two useful convenience operations

`insert_or_assign(key, value)` states the insert-or-replace intention directly. `try_emplace(key, value)` creates a new mapped value only when the key is absent; an existing one is retained. They are available in our C++23 target.<sup>[5-8]</sup>

You can solve the core exercises with ordinary assignment, `insert`, and `find`. The two extra spellings are conveniences, not a reason to memorize several versions of the same task at once. When passing a function call as an argument, remember that argument evaluation still occurs; `try_emplace` is not a general promise to avoid evaluating every expression you pass.

#### Pause and predict – 14

A task says “keep the first price for each product.” Is `price[name] = x`; correct for every arrival?

**Check your prediction.** No: it keeps the last price. Use insertion that does not overwrite an existing key, or check absence before assigning. The correct syntax follows the task’s repeated-key rule.

#### Complete program – first value, replacement, and explicit assignment

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> scores;
    scores.insert({"aya", 70});
    auto result = scores.insert({"aya", 90});
    std::cout << result.second << ' ' << scores.at("aya")
              << '\n';
    scores["aya"] = 95;
    scores.try_emplace("aya", 100);
    std::cout << scores.at("aya") << '\n';
    scores.insert_or_assign("aya", 80);
    std::cout << scores.at("aya") << '\n';
}
```

| Sample input | Sample output |
|--------------|---------------|
| (no input)   | 0 70          |
|              | 95            |
|              | 80            |

The attempted insertion of 90 returns a false flag and preserves 70. Assignment then changes the value to 95. `try_emplace` leaves that existing value alone. Finally, `insert_or_assign` replaces it with 80.

### Trace the record after every update

| Operation on the same key <code>aya</code>       | Stored score afterward               |
|--------------------------------------------------|--------------------------------------|
| <code>insert({"aya", 70})</code> on an empty map | 70: the key is new.                  |
| <code>insert({"aya", 90})</code>                 | 70: the existing record is retained. |
| <code>scores["aya"] = 95</code>                  | 95: assignment replaces the value.   |
| <code>try_emplace("aya", 100)</code>             | 95: the existing record is retained. |
| <code>insert_or_assign("aya", 80)</code>         | 80: the value is replaced.           |

The size remains 1 throughout. There are not five different records merely because five updates mention the name. A unique-key map separates the number of keys from the number of updates performed on those keys.

### First, latest, or accumulated?

These are three different problem statements. Keeping the first score needs insertion that does not replace an existing entry. Keeping the latest score needs assignment. Accumulating points needs `scores[name] += amount`, so each new amount contributes to a running total rather than replacing it.

For amounts 10, 4, and 7 under one name, these policies produce 10, 7, and 21 respectively. The container declaration can be the same in all three solutions; the update operation expresses the task's policy. When a program has correct syntax but the wrong answer, check this policy before changing the data structure.

## 5.15 Traversing maps and changing mapped values

### Every entry has first and second fields

The stored entry type for `map<Key, Value>` is a pair with a const key and a mapped value. The key cannot be changed through the iterator, but the mapped value can be changed in a non-const map.<sup>[5-6]</sup>

```
// Fragments -- inside a function; scores is map<string, int>.
for (const auto& entry : scores) {
    std::cout << entry.first << ' ' << entry.second << '\n';
}
for (auto& entry : scores) {
    entry.second += 10;
}
```

The first loop reads without copying the string key. The second loop modifies the actual associated integers. Neither loop changes the keys or the map's key order.

### A copy is still a copy

```
for (auto entry : scores) {
    entry.second += 10; // Changes the local copy, not scores.
}
```

Removing `&` from the loop declaration changes the meaning. This is the same reference distinction from Session 3, now applied to a key-value entry.

### Complete program – compare a copy loop with a reference loop

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> scores{{"nour", 70},
                                     {"ali", 90}};
    for (auto entry : scores) entry.second += 5;
    for (auto& entry : scores) entry.second += 10;
    for (const auto& entry : scores) {
        std::cout << entry.first << ' ' << entry.second
                  << '\n';
    }
}
```

#### Sample input

(no input)

#### Sample output

ali 100  
nour 80

The first loop's increments do not survive. The second adds ten to each stored score. Output is ordered by name, so Ali appears before Nour even though Nour was listed first in the initializer.

## Iterator notation and structured bindings

Equivalent read-only fragments:

```
for (auto it = scores.begin(); it != scores.end(); ++it) {
    std::cout << it->first << ' ' << it->second << '\n';
}

// Optional syntax, introduced in Session 4:
for (const auto& [name, score] : scores) {
    std::cout << name << ' ' << score << '\n';
}
```

The structured binding gives the two fields descriptive names. The `const auto&` still matters: the binding reads the existing entry rather than making an unnecessary copy.

## Erasing records

`scores.erase(name)` removes that key’s entire record, if present. It is not “set the score to zero.” A zero score is still a record. The same guarded iterator-erasure pattern used for sets also works for maps.

```
// Fragment -- remove entries with a negative score.
for (auto it = scores.begin(); it != scores.end(); ) {
    if (it->second < 0) it = scores.erase(it);
    else ++it;
}
```

Do not erase the current record inside an ordinary range loop and then allow the loop to increment its erased iterator. Use an explicit iterator loop when the current entry may be removed.

## Equality compares stored information

Two standard maps with the same type can be compared with `==`. They are equal when they contain the same key-value entries. This can compare two independently built frequency maps, such as character counts for an anagram test. The comparison itself may inspect all entries; it is not a constant-time identity test.<sup>[5-9][5-14]</sup>

### Pause and predict – 15

Does changing `entry.second` from 10 to 100 move an entry to the end of a map? Can `entry.first` be assigned a new key through the map iterator?

**Check your prediction.** No to both. Map order depends on keys, not values. The stored key is read-only; changing it requires a deliberate removal/insertion approach, not assignment through the iterator.

## 5.16 Frequency maps and tie-breaking

### One entry remembers one count

A frequency map attaches an occurrence count to each distinct key. The pattern `++frequency[x]` is useful because a missing integer count starts at zero before being incremented.

```
// Fragment -- inside a function; frequency is map<int, int>.
++frequency[x];
```

After the prefix `4 7 4`, the entries are `4 -> 2` and `7 -> 1`. The map size is 2, while the sum of the counts is 3. The input prefix has three items but only two different keys.

**Task.** Read  $0 \leq n \leq 200000$  lowercase ASCII words, each of length 1–20. Print the number of different words, then each `word count` in lexicographical order.

**Complete program – a count grows with each occurrence**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<std::string, int> frequency;
    for (int i = 0; i < n; ++i) {
        std::string word;
        std::cin >> word;
        ++frequency[word];
    }
    std::cout << frequency.size() << '\n';
    for (const auto& entry : frequency) {
        std::cout << entry.first << ' ' << entry.second
                  << '\n';
    }
}
```

#### Sample input

```
6
red blue red green blue red
```

#### Sample output

```
3
blue 2
green 1
red 3
```

## Prove a frequency table with a prefix invariant

Before reading anything, the empty map correctly describes zero occurrences. When the next word arrives, only that word's count changes, and it increases by exactly one. Therefore, after every prefix, the map stores the exact counts for that prefix. After the final word, it describes the entire input.

With bounded word lengths, building takes  $O(n \log(n+1))$ , and output visits each of the  $d$  distinct keys once. Space is  $O(d)$  entries plus their key text. For arbitrary maximum key length  $L$ , allow up to  $O(L)$  work per string comparison and  $O(dL)$  key storage.

## The most frequent value needs a tie rule

Suppose the input is 8 3 8 3 2 2 9. Values 2, 3, and 8 all appear twice. “Print the most frequent value” is ambiguous without a tie rule. Here the rule is: **choose the smallest value among tied counts**.

An ascending map visits 2 before 3 before 8. Replacing the answer only on a *strictly greater* count preserves the earlier, smaller key when counts tie.

**Task.** Read  $1 \leq n \leq 200000$  integers in  $[-10^9, 10^9]$ . Print the most frequent value and its frequency, choosing the smaller value on ties.

### Complete program – sorted traversal plus strict improvement

```
#include <iostream>
#include <map>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<int, int> frequency;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++frequency[x];
    }
    int bestValue = 0, bestCount = 0;
    for (const auto& entry : frequency) {
        if (entry.second > bestCount) {
            bestValue = entry.first;
            bestCount = entry.second;
        }
    }
    std::cout << bestValue << ' ' << bestCount << '\n';
}
```

### Sample input

### Sample output

7

2 2

8 3 8 3 2 2 9

## Trace the candidate

| Key | Count | Action                                     | Best after the step |
|-----|-------|--------------------------------------------|---------------------|
| 2   | 2     | Greater than the initial count 0: replace. | Value 2, count 2.   |
| 3   | 2     | Equal count: keep the smaller earlier key. | Value 2, count 2.   |
| 8   | 2     | Equal count: keep.                         | Value 2, count 2.   |
| 9   | 1     | Smaller count: keep.                       | Value 2, count 2.   |

The initial `bestValue = 0` does not mean zero is a valid candidate. It is a placeholder until the first positive frequency beats `bestCount = 0`. The task guarantees nonempty input, so at least one candidate is selected.

## Why greater-or-equal would change the answer

Using `>=` instead of `>` would keep replacing the candidate on equal counts. With ascending traversal, the final answer would be the *largest* tied key. Neither comparison is universally correct; its meaning must match the stated tie rule.

For an unordered map, traversal gives no smallest-first guarantee. There you must explicitly compare both count and key when resolving a tie, or sort the entries before scanning them.

### Pause and predict – 16

Input values are `-5 2 -5 2`. With the smaller-value tie rule, what should be printed? Why must you not initialize the best count to a large value?

**Check your prediction.** Print `-5 2`. The first positive count must be able to replace the initial placeholder. An initial count larger than every real count could prevent any actual candidate from being selected.

## 5.17 Ordered map bounds search the keys

### A threshold on keys, not on associated values

For `map<int, int> records`, `records.lower_bound(x)` finds the first entry whose key is at least `x` under the default ascending comparison. Its associated value does not participate in the search.<sup>[5-1][5-9]</sup>

A schedule might map a start time to a room number. Searching for the next start time uses the keys. Searching for a particular room number is a different query and is not made fast merely because the records are in a map.

**Task.** Read `n` `q`, then `n` integer **key value** records and `q` threshold keys. Repeated keys keep their last value. For each threshold, print the first record with key at least that threshold, or **NONE**. Here  $0 \leq n, q \leq 200000$ ; record fields lie in  $[-10^9, 10^9]$ , queries in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ .

### Complete program – a nearby record by key

```
#include <iostream>
#include <map>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::map<int, int> records;
    for (int i = 0; i < n; ++i) {
        int key = 0, value = 0;
        std::cin >> key >> value;
        records[key] = value;
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        auto it = records.lower_bound(x);
        if (it == records.end()) std::cout << "NONE\n";
        else std::cout << it->first << ' ' << it->second
                        << '\n';
    }
}
```

### Sample input

```
3 3
10 100
3 90
7 50
4
10
11
```

### Sample output

```
7 50
10 100
NONE
```

## Explain the sample

The keys are 3, 7, and 10. Threshold 4 selects key 7 and prints its value 50. The fact that key 3 has a larger value, 90, is irrelevant. Threshold 11 has no suitable record.

**Cost.** Each bound is  $O(\log(d + 1))$ . Building and all queries take  $O((n + q) \log(n + 1))$  time and  $O(d)$  entries. Printing  $k$  records in a key interval would add  $O(k)$  output work.

## Trace the key comparison, then read the value

| Threshold | First key at least the threshold | Output entry |
|-----------|----------------------------------|--------------|
| 4         | 7                                | 7 50         |
| 10        | 10                               | 10 100       |
| 11        | None                             | NONE         |

For threshold 4, the map skips key 3 even though its associated value 90 exceeds 4. The bound comparison is between 4 and the keys 3, 7, and 10; the values 90, 50, and 100 do not decide the location. After locating an entry, the iterator lets the program print both fields.

A query asking for the first name alphabetically would need names as ordered keys. A query asking for the largest stored score is different: a map keyed by ID does not maintain score order for free. Decide which field the query searches before deciding which field becomes the key.

For an empty map, every bound returns `end()`. The same checked branch prints `NONE`; no dummy record or special numeric sentinel is needed.

## 5.18 Grouping with a map of vectors

### The associated value need not be a number

A map can attach a vector to a key. Suppose we want the positions where each input value occurs. For 4 2 4 7 2 4, the groups are:

| Value | One-based input positions |
|-------|---------------------------|
| 2     | 2 5                       |
| 4     | 1 3 6                     |
| 7     | 4                         |

```
// Fragment -- inside a function; include <map> and <vector>.
std::map<int, std::vector<int>> positions;
positions[x].push_back(i);
```

If  $x$  is absent, square brackets create an empty vector for that key. Then `push_back` appends the current position. This is intentional access-or-create behavior, not an accidental query insertion.

**Task.** Read  $0 \leq n \leq 200000$  integers in  $[-10^9, 10^9]$ . For each distinct value, print **value count positions...**, ordered by value. Positions are one-based and increasing within a group. Empty input produces no output lines.

**Complete program – one vector per distinct key**

```
#include <iostream>
#include <map>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<int, std::vector<int>> positions;
    for (int i = 1; i <= n; ++i) {
        int x = 0;
        std::cin >> x;
        positions[x].push_back(i);
    }
    for (const auto& entry : positions) {
        std::cout << entry.first << ' ' << entry.second.size();
        for (int index : entry.second) std::cout << ' ' << index;
        std::cout << '\n';
    }
}
```

**Sample input**

6  
4 2 4 7 2 4

**Sample output**

2 2 2 5  
4 3 1 3 6  
7 1 4

## Two different orders are being preserved

The map orders the *groups* by key. Each vector preserves the order in which positions were appended. Reading the input from left to right means those positions are already increasing. No separate sorting step is needed inside a group.

## Nested loops are not automatically quadratic

The output has a loop over groups and an inner loop over each group's positions. However, every original position belongs to exactly one vector. If the group sizes are  $g_1, g_2, \dots, g_d$ , then

$$g_1 + g_2 + \dots + g_d = n.$$

The inner loops print  $n$  positions in total, not  $n$  positions for every group. Total output work is  $O(n + d)$ .

Building performs one map access per input value and one vector append. With amortized vector append cost, total construction takes  $O(n \log(n + 1))$ . The map stores  $d$  keys and all vectors together store  $n$  positions, so storage is  $O(n + d)$ , not  $O(d)$  alone.

## Avoid copying every group

Use `const auto& entry` while printing. A loop written as `for (auto entry : positions)` copies the entire key-value pair, including its vector. The output may remain correct, but you have added extra copying and temporary storage without helping the task.

### Pause and predict – 17

If one group contains 999 positions and another contains 1, how many position visits occur when both groups are printed? Does the map's two-entry size mean the storage is constant?

**Check your prediction.** There are 1000 position visits in total. The associated vectors still store 1000 positions; map size alone does not count the memory inside its values.

## 5.19 Unordered map: exact-key lookup without sorting

### Keep key-value entries, change the lookup strategy

A `std::unordered_map<Key, Value>` still stores one associated value per unique key. Its main difference from `map` is that it organizes lookup using hashing instead of providing sorted key traversal.<sup>[5-10][5-11]</sup>

Think of a hash as a rule that helps choose where to look. The container places entries into groups called **buckets**. A **collision** happens when different keys are directed to the same bucket. The container still checks key equality; a collision does not make different keys become the same entry.

You do not need to implement a hash table to use it. You do need to know which guarantees its interface does and does not provide.

### No promised sorted or insertion order

```
// Fragment -- inside a function; include <unordered_map>.
std::unordered_map<int, int> count;
count[8] = 2;
count[3] = 7;
count[5] = 1;
```

Do not predict a required output order from a range loop over this object. It is not guaranteed to visit 3, 5, 8, and it is not guaranteed to preserve 8, 3, 5. “Unordered” does not mean a new random shuffle on every access; it means the interface gives no usable sorted-order promise.<sup>[5-11]</sup>

### Average cost is not a worst-case promise

For fixed-size keys and ordinary hash/equality costs, exact-key lookup, insertion, and key erasure are average  $O(1)$ , but can take  $O(d)$  in the worst case. Unfavorable collisions can therefore make many operations much slower than the average-case estimate.<sup>[5-11][5-13]</sup>

A map’s logarithmic key lookup is a different guarantee. A hash map is not automatically the fastest or safest choice for every contest task. Choose it when you need exact-key access without ordered queries and can accept its stated cost model.

#### Pause and predict – 18

Two different keys collide in a hash table. Must one overwrite the other? Does `unordered_map.begin()` identify the smallest key?

**Check your prediction.** No to both. Equality checks distinguish different keys despite collisions. The beginning of unordered traversal has no guaranteed minimum-key meaning.

## 5.20 Unordered-map queries and deterministic output

### Familiar operations, the same insertion warning

`find`, `contains`, `count`, `insert`, `erase`, `size`, and `empty` have familiar roles. Square brackets still create a default mapped value for a missing key, and `at` still throws when the key is missing.<sup>[5-12][5-13]</sup>

**Task.** Read  $n$   $q$ , then  $n$  integers and  $q$  query integers. Print each query's frequency, using zero for an absent key. Finally print the number of distinct *input* keys. Queries must not create extra entries. Bounds are as in the membership example.

**Complete program – query counts without inserting missing keys**

```
#include <iostream>
#include <unordered_map>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::unordered_map<int, int> frequency;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++frequency[x];
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        auto it = frequency.find(x);
        std::cout << (it == frequency.end() ? 0 : it->second)
                  << '\n';
    }
    std::cout << frequency.size() << '\n';
}
```

#### Sample input

```
5 3
8 2 8 3 2
8
99
2
```

#### Sample output

```
2
0
2
3
```

**Why the final size is useful.** Missing query 99 prints zero, but the final size stays three. A careless `frequency[x]` query would silently add a zero-valued 99 record and change that final size.

**Cost.** Average  $O(n+q)$  work for integer keys, with a conservative worst-case bound  $O((n+q)(d+1))$  from potentially linear hash operations. The live entries use  $O(d)$  storage plus the bucket array.

## When sorted output is required

You may use an ordered map from the start. Another approach is to build a hash map, copy its entries into a vector, and sort that vector for output. This does not turn the hash map into an ordered map; it creates a separate ordered representation.

**Task.** Read  $0 \leq n \leq 200000$  words of 1–20 lowercase ASCII letters. Count them with an unordered map, then print `word count` lines in lexicographical order. Empty input produces no output lines.

**Complete program – explicitly sort the output records**

```
#include <algorithm>
#include <iostream>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::unordered_map<std::string, int> frequency;
    for (int i = 0; i < n; ++i) {
        std::string word;
        std::cin >> word;
        ++frequency[word];
    }
    std::vector<std::pair<std::string, int>> rows;
    for (const auto& entry : frequency) rows.push_back(entry);
    std::sort(rows.begin(), rows.end());
    for (const auto& row : rows) {
        std::cout << row.first << ' ' << row.second << '\n';
    }
}
```

### Sample input

```
5
tea milk tea water milk
```

### Sample output

```
milk 2
tea 2
water 1
```

The vector contains pairs of word and count. Default pair ordering compares the word first. Keys are unique in this vector, so no count-based tie-break is needed.<sup>[5-18]</sup>

**Cost.** For bounded-length words, expected/average hash-table work plus sorting gives  $O(n + d \log(d + 1))$  under the average hash model. This is not a guaranteed worst-case linear build. Copying records and sorting them also require additional storage; the original hash entries have not disappeared.

### Operations that an unordered map does not provide

There is no ordered `lower_bound`, `upper_bound`, or rank-selection interface. A map with integer keys is still key-based: `m[2]` looks up key 2, not the third entry in iteration order.

## 5.21 Buckets, reserve, rehashing, and validity

### Size and bucket count measure different things

`size()` counts entries. `bucket_count()` counts the buckets used to organize those entries. The **load factor** relates them: entries divided by buckets. A larger bucket count does not mean more stored keys.<sup>[5-11]</sup>

`reserve(n)` requests enough bucket room for approximately `n` entries under the current maximum load-factor setting. It may reorganize the table, but it does not insert `n` default key-value records. `rehash(b)` requests a bucket-count lower bound subject to the container's load requirements; it does not promise exactly `b` buckets.

```
// Fragment -- inside a function.
std::unordered_map<int, int> frequency;
frequency.max_load_factor(0.7F);
frequency.reserve(1000);
// frequency.size() is still 0.
```

The `F` marks a `float` literal. Set a chosen load-factor policy before reserving so the reservation uses that policy. The default policy is fine for learning; this tuning is optional.

### Rehashing invalidates iterators, not surviving references

A rehash invalidates the table's iterators and can change traversal order. References and pointers to surviving elements remain valid; erasing the referenced element still invalidates them.<sup>[5-11]</sup>

#### Wrong fragment – do not reuse an iterator across possible rehashing:

```
auto it = frequency.find(7);
frequency.reserve(100000);
// std::cout << it->second; // Reacquire the iterator instead.
```

Insertion may also trigger rehashing. The safe beginner habit is to obtain a fresh iterator after an operation that may rehash, unless you have deliberately established a guarantee that it cannot happen.

#### Pause and predict – 19

After reserving room in an empty unordered map, can you say there are 1000 entries? Does reserving make all future lookups worst-case constant time?

**Check your prediction.** No. Reserving creates no entries, and it does not eliminate collisions or change the worst-case lookup guarantee. It is a storage-planning operation, not a proof of constant worst-case work.

## A stable value reference and a fresh iterator

This program takes no input and prints only guaranteed results. It deliberately does not print an implementation-specific exact bucket count.

### Complete program – distinguish an element reference from an iterator

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, int> values;
    values.max_load_factor(0.7F);
    values.reserve(100);
    std::cout << values.size() << '\n';
    values[7] = 30;
    int& savedValue = values.at(7);
    values.rehash(1000);
    savedValue += 2;
    auto fresh = values.find(7);
    std::cout << fresh->second << ' ' << values.size() << '\n';
}
```

#### Sample input

(no input)

#### Sample output

0  
32 1

## Explain what survives

Reserving leaves the map empty. Assigning key 7 creates one entry with value 30. `at(7)` returns a reference to that existing value. Rehashing reorganizes buckets without erasing it. The reference can therefore add two to the value, and a newly obtained iterator reads 32.

The example does not retain and then dereference an old iterator. An element reference and an iterator have different validity rules even when both were originally obtained from the same entry.

## Reserve deliberately, not mechanically

Reserving a very large number of buckets can use substantial memory even with very few keys. The useful storage model is  $O(d + b)$  for  $d$  entries and  $b$  buckets, plus dynamic memory in keys and values. Do not describe an empty, heavily reserved hash table as storing nothing at all.

Repeatedly requesting more reserve space inside a tight loop can also force unnecessary reorganizations. When a sensible bound is known, reserve once before the build rather than treating reserve as an operation to perform after every insertion.

### No hash implementation details are prerequisites

The core practice requires knowing the interface and its guarantees, not writing a custom hasher. A fixed custom-hash template copied from elsewhere is not a substitute for understanding collisions, portability, or a worst-case bound.

## 5.22 Choose a container and count its real work

### Start with the required operations

| Need                                                      | A suitable starting point                                       |
|-----------------------------------------------------------|-----------------------------------------------------------------|
| Keep distinct keys and answer nearby-key queries.         | <code>set</code> .                                              |
| Keep repeated items and select/remove a nearby copy.      | <code>multiset</code> .                                         |
| Store information per key and traverse keys in order.     | <code>map</code> .                                              |
| Store information per key; only exact-key lookup matters. | <code>unordered_map</code> , with average/worst-case tradeoffs. |
| Keep the original arrival order and re-visit positions.   | <code>vector</code> , possibly alongside a set or map.          |
| Answer changing rank/select queries efficiently.          | GNU ordered set, introduced next.                               |

There may be several correct solutions. For static data that only needs one sorted pass, a sorted vector can be simpler and more compact than maintaining a separate ordered container. A data structure earns its place by supporting the operations the algorithm needs.

### Ordered-container costs

Let  $n$  be the current number of stored entries and  $k$  the number of matching or visited entries. Treat integer comparison and destruction as constant-cost operations.

| Operation                                                                              | Cost                                              |
|----------------------------------------------------------------------------------------|---------------------------------------------------|
| <code>find</code> , <code>contains</code> , ordinary insertion, member bounds          | $O(\log(n + 1))$ .                                |
| <code>set.count</code> , <code>map.count</code>                                        | $O(\log(n + 1))$ , since at most one key matches. |
| <code>multiset.count(x)</code> , <code>multiset.erase(x)</code>                        | $O(\log(n + 1) + k)$ .                            |
| Erase one already-known valid iterator                                                 | Amortized $O(1)$ .                                |
| <code>size()</code> , <code>empty()</code> , <code>begin()</code> , <code>end()</code> | $O(1)$ .                                          |
| Traverse or clear all entries                                                          | $O(n)$ .                                          |
| Walk $k$ set/map iterator steps                                                        | $O(k)$ , not $O(1)$ .                             |

These are the costs of `std::set`, `std::multiset`, and `std::map` operations under their ordinary preconditions. Erasing by iterator does not make *finding that iterator* free.<sup>[5-1][5-14][5-15]</sup>

## Average hash costs versus worst-case hash costs

For integer-key unordered-map lookup, ordinary insertion, and key erasure, use average  $O(1)$  and worst-case  $O(n)$ . Rehashing is a separate whole-table operation; do not put it in a constant-time column merely because its name is short. The standard specifies average linear and worst-case quadratic complexity for `rehash`.<sup>[5-11][5-13]</sup>

If a solution makes  $q$  queries against at most  $n$  keys, an average  $O(q)$  lookup estimate is not a worst-case  $O(q)$  guarantee. With an ordered map, the analogous key-search estimate is  $O(q \log(n+1))$  without assuming favorable hash distribution.

## Text keys add character work

Comparing two strings can inspect many characters before finding a difference. A typical string hash also processes the key's characters. Therefore, treating arbitrary-length strings as fixed-cost integers can hide important work.<sup>[5-19]</sup>

For a maximum key length  $L$ , a conservative ordered-map lookup estimate is  $O(L \log(d+1))$ . Under an ordinary linear-time string-hash model and suitable average bucket behavior, allow  $O(L)$  average character work per exact-key hash lookup, not automatically  $O(1)$  independent of text length. The C++ interface does not promise a particular hashing algorithm.

The chapter's 20-character word bound lets many teaching estimates treat key comparison as bounded work. That is a stated simplification, not a general claim about all strings.

## Repeated counting can dominate everything else

```
// Fragment -- bag has n copies of the same integer.
long long total = 0;
for (int x : bag) {
    total += bag.count(x);
}
```

There are  $n$  loop iterations and each count may inspect  $n$  matching copies. Total work can be quadratic. With  $n=100000$ , that means reasoning about roughly ten billion visits, not merely one hundred thousand logarithmic lookups.

### Pause and predict – 20

You find a lower bound in a map, then print 5000 entries from there. Is the entire operation logarithmic? If you copy the map into a helper on every query, does that add work?

**Check your prediction.** Printing adds work proportional to the number of entries printed. Passing a whole map by value also creates a copy; repeated queries can then spend linear copying work before the lookup even starts. Count both.

## 5.23 References, copies, and fresh test cases

### Read a map without copying or changing it

A helper that only queries should normally take a `const` reference. This keeps the original collection, avoids copying it, and makes accidental subscript insertion unavailable through that parameter.

**Task.** Read  $n$   $q$ , build an integer frequency map, and print  $q$  requested frequencies. Missing keys have frequency zero. Bounds are as in the membership example.

**Complete program – a read-only lookup helper**

```
#include <iostream>
#include <map>

int countOf(const std::map<int, int>& frequency, int key) {
    auto it = frequency.find(key);
    if (it == frequency.end()) return 0;
    return it->second;
}

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::map<int, int> frequency;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++frequency[x];
    }
    while (q--) {
        int key = 0;
        std::cin >> key;
        std::cout << countOf(frequency, key) << '\n';
    }
}
```

#### Sample input

```
4 3
5 5 2 9
5
7
2
```

#### Sample output

```
2
0
1
```

### Read the signature as a contract

`const std::map<int, int>&` means the helper receives a read-only reference to the existing map. `int key` is a small copied input. The helper returns a count and does not print, insert, or erase anything.

Removing the `&` would copy all entries on every call. With  $q$  queries and  $d$  distinct keys, that could add  $O(qd)$  copy work. A correctly selected container cannot rescue an unnecessarily copied interface.

**Pause and predict – 21**

Two cases contain 1 2 and 2 3. What should the two distinct counts be? What might happen if a shared set is never cleared?

**Check your prediction.** Both answers should be 2. An uncleared set can produce 2 and then 3 because it wrongly retains key 1 from the first case.

**Each test case starts with its own state**

**Task.** Read  $1 \leq T \leq 200000$ . For each case, read  $n$  and its integers, then print the distinct count. Each  $0 \leq n \leq 200000$ , the sum of all lengths is at most 200000, and keys lie in  $[-10^9, 10^9]$ .

**Complete program – local state inside solve**

```
#include <iostream>
#include <set>

void solve() {
    int n = 0;
    std::cin >> n;
    std::set<int> seen;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        seen.insert(x);
    }
    std::cout << seen.size() << '\n';
}

int main() {
    int tests = 0;
    std::cin >> tests;
    while (tests--) solve();
}
```

**Sample input****Sample output**

|         |   |
|---------|---|
| 3       | 2 |
| 4       | 0 |
| 1 2 2 1 | 1 |
| 0       |   |
| 3       |   |
| 9 9 9   |   |

The set is created inside `solve()`, so every call starts empty. A global set without clearing would mix participant IDs from different cases. A previous case's empty or nonempty state must not affect the next answer.

**Add the work across cases**

For case lengths  $n_1, \dots, n_T$ , add each case's  $O(n_i \log(n_i + 1))$  insertion work and the constant input/output overhead per case. With total length  $N$ , a simple overall bound is  $O(T + N \log(N + 1))$ . Peak set storage follows the largest live case, not the sum of every case processed.

## 5.24 GNU ordered sets: sorted order plus rank

### Two questions an ordinary set does not answer quickly

For the distinct keys 2, 5, 8, 10, the third smallest key is 8, and three keys are strictly smaller than 9. These are **order-statistics** questions: finding an item by rank and finding a rank by value.

|                 |   |   |   |    |
|-----------------|---|---|---|----|
| Zero-based rank | 0 | 1 | 2 | 3  |
| Key             | 2 | 5 | 8 | 10 |

A `std::set` already stores keys in order, but its iterators do not support direct indexing. Walking from `begin()` to the third item is possible; walking to an arbitrary rank  $k$  costs  $O(k)$ . Repeating long walks for many queries can be too slow.

A **GNU PBDS ordered set** adds rank information to a tree. With the red-black-tree configuration used here, insertion, deletion by key, rank queries, and selection by rank take  $O(\log(n+1))$  time for fixed-cost keys. The tree stores additional counts so it can skip whole groups of keys rather than walking past them one by one. You do not need to implement those counts yourself. <sup>[5-20][5-22][5-23][5-24]</sup>

### Choose it for the operations, not for the name

| Tool                  | Find a rank            | Read item at rank      | Insert/delete in the middle |
|-----------------------|------------------------|------------------------|-----------------------------|
| Sorted vector         | Logarithmic search     | Constant indexing      | Linear shifting             |
| <code>std::set</code> | Linear walk in general | Linear walk in general | Logarithmic key operation   |
| GNU ordered set       | Logarithmic            | Logarithmic            | Logarithmic                 |

For a vector, the rank search assumes the vector is already sorted; building or maintaining it is additional work. A fixed dataset may need only a sorted vector. Dynamic updates plus rank queries are where the GNU tool becomes useful.

#### This is an extension, not a standard container name

C++23 has no `std::ordered_set` or `std::ordered_map`. The ordered map in this course is `std::map`. The order-statistics tree below belongs to GNU's policy-based data structures, in namespace `__gnu_pbds`. A C++23 compiler setting alone does not promise that these headers exist. <sup>[5-20][5-21]</sup>

#### Pause and predict – 22

A sorted set contains 2, 5, 8, 10. How many stored keys are strictly smaller than 8? What is the key at zero-based rank 2?

**Check your prediction.** The count is 2, because only 2 and 5 are smaller. The key at rank 2 is 8. A rank is a position in the current sorted collection, not the numerical value of the key.

## 5.25 Setting up the GNU order-statistics tree

### Give the long type a short name

The following complete program uses GNU PBDS. Save it as a separate file and compile it with a GNU C++23 environment that provides the PBDS headers. All later GNU examples repeat the required setup; there is no hidden header to find.

#### Complete program – GNU PBDS extension in C++23 mode

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    OrderedSet values;
    values.insert(7);
    values.insert(2);
    values.insert(9);
    values.insert(7);
    std::cout << values.size() << '\n';
    std::cout << values.order_of_key(7) << '\n';
    auto it = values.find_by_order(1);
    if (it != values.end()) std::cout << *it << '\n';
    std::cout << (values.find_by_order(3) == values.end())
        << '\n';
}
```

#### Sample input

(no input)

#### Sample output

3  
1  
7  
1

### Read the declaration in small pieces

`using OrderedSet = ...;` gives a long type the shorter name `OrderedSet`. This is a type alias, not an object. The line `OrderedSet values;` creates an object of that type.

Inside the tree configuration, `int` is the key type, `null_type` means there is no separate mapped value, and `std::less<int>` requests strict ascending order. `rb_tree_tag` chooses a red-black tree. Finally, `tree_order_statistics_node_update` maintains the extra information used for rank and selection. The two `ext/pb_ds` headers provide the tree and its order-statistics policy.<sup>[5-21][5-22][5-23]</sup>

## Explain all four output lines

After inserting 7, 2, 9, and 7, the distinct keys are 2, 7, 9. Inserting 7 twice does not create two copies. The size is therefore 3. Only one key is strictly below 7, so `order_of_key(7)` is 1. The key at rank 1 is 7. Rank 3 lies past the last key, so `find_by_order(3)` returns `end()`; the final comparison prints 1 for true.<sup>[5-23][5-24]</sup>

```
g++ -std=c++23 -Wall -Wextra -O2 \
    examples/session_05/25_pbds_basic.cpp -o demo
./demo
```

Run that command from the source-package root. An error saying an `ext/pb_ds` header is missing is an environment/availability issue, not a signal to replace the comparison with a non-strict one. Use a supported GNU toolchain when the judge permits it, or choose a different algorithm with the tools the environment provides.

## What you need to remember now

Memorize the meaning of the two extra operations before memorizing the type declaration. Keep the verified declaration as a reference. You do not need to learn how to write your own templates or implement red-black-tree rotations in this session.

### Strict means that a key is not less than itself

The comparator must describe a valid strict ordering. `std::less<int>` satisfies that requirement for integers. `std::less_equal<int>` does not: it reports that `5 <= 5` is true. It is not a legitimate way to make this tree accept duplicates. The optional duplicate section uses unique pair keys instead.<sup>[5-21][5-22]</sup>

## 5.26 Counting smaller keys with `order_of_key`

**The argument is a value; the answer is a count**

`values.order_of_key(x)` counts stored keys strictly smaller than `x` under our ascending comparator. The queried key does not have to be present, and the query does not insert it. With 2, 5, 8, 10, queries 1, 5, 9, and 20 return 0, 1, 3, and 4.<sup>[5-23][5-24]</sup>

**Task.** Read  $n$   $q$ , then  $n$  integers and  $q$  thresholds. For each threshold, print the number of *distinct* input values strictly smaller than it. Here  $0 \leq n, q \leq 200000$ ; values are in  $[-10^9, 10^9]$ , and thresholds are in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ .

**Complete program – count distinct smaller values, GNU PBDS**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    OrderedSet values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        std::cout << values.order_of_key(x) << '\n';
    }
}
```

**Sample input**

**Sample output**

|            |   |
|------------|---|
| 5 4        | 0 |
| 8 2 5 2 10 | 1 |
| 2          | 2 |
| 5          | 4 |
| 7          |   |
| 20         |   |

## Why duplicates do not contribute twice

If the input is 4 4 7, the tree contains only 4, 7. A query for 8 returns 2, not 3. That is correct for this task because it explicitly asks for *distinct* values. The word “distinct” changes the required representation.

Construction and queries together take  $O((n + q) \log(n + 1))$  time, with  $O(d)$  tree storage for  $d$  distinct keys. When  $n=0$ , every answer is zero; account separately for the  $O(q)$  input/output work. A convenient bound covering empty inputs is  $O((n + q)(1 + \log(n + 1)))$ .

### Pause and predict – 23

The input values are 3 3 8. What are `order_of_key(3)`, `order_of_key(4)`, and `order_of_key(9)`? Does the middle query change the tree?

**Check your prediction.** The answers are 0, 1, and 2. The tree contains the distinct keys 3 and 8 throughout. Querying 4 only counts keys before it; it does not insert 4.

## 5.27 Selecting a key with `find_by_order`

**The argument is a rank; the answer is an iterator**

`find_by_order(k)` returns an iterator to zero-based rank `k`. Dereferencing that iterator gives the key. A rank at least the size produces `end()`, not the greatest key. Never dereference that result.<sup>[5-23][5-24]</sup>

**Task.** Read `n` `q`, `n` integers, then `q` signed zero-based ranks. Print the selected distinct value or `INVALID`. Bounds are  $0 \leq n, q \leq 200000$ , keys in  $[-10^9, 10^9]$ , and requested ranks in  $[-200000, 200001]$ .

**Complete program – guarded zero-based selection, GNU PBDS**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    OrderedSet values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int k = 0;
        std::cin >> k;
        if (k < 0 || k >= static_cast<int>(values.size())) {
            std::cout << "INVALID\n";
        } else {
            auto it = values.find_by_order(
                static_cast<OrderedSet::size_type>(k));
            std::cout << *it << '\n';
        }
    }
}
```

**Sample input**

```
4 4
8 2 8 5
0
2
3
-1
```

**Sample output**

```
2
8
INVALID
INVALID
```

## Validate first, convert second

The function's rank type is unsigned. Our input variable is signed so it can represent a negative invalid request. We first test whether  $k < 0$  or  $k \geq \text{size}$ . Only a valid rank is converted to the tree's size type.

Converting a negative number to an unsigned type is not a useful way to validate it. Likewise, `static_cast<int>(values.size())` is appropriate here because the stated 200000-key bound fits the tested `int` type; it is not a rule for arbitrary huge collections.

## One-based statements need one conversion

For a task asking for the first, second, or  $k$ -th smallest key, require  $1 \leq k \leq \text{size}$ , then request `find_by_order(k - 1)`. Do not subtract one twice. Later practice problems deliberately state their indexing convention so you must read it before coding.

### Pause and predict – 24

A tree contains 4, 9, 15. Which call selects the third smallest key? Is zero-based rank 3 valid?

**Check your prediction.** Use `find_by_order(2)` after validating the one-based request 3. Zero-based rank 3 is invalid because the valid ranks are 0, 1, and 2.

## 5.28 Dynamic ranks: update the set, then query it

### A rank belongs to the current state

Insertions and deletions change ranks. If 5, 12, 20 becomes 5, 8, 12, 20, key 12 moves from zero-based rank 1 to rank 2. The key itself did not change; its position among the stored keys did.

**Task.** Start empty and process  $0 \leq q \leq 200000$  commands. Every command contains two integers. 1 *x* inserts a key; 2 *x* erases that key if present; 3 *x* prints the number of keys strictly less than *x*; 4 *k* prints the key at *zero-based* rank *k*, or INVALID. Keys and thresholds fit  $[-2 \cdot 10^9, 2 \cdot 10^9]$ ; ranks fit the signed range used in the selection example.

**Complete program – online updates and queries, GNU PBDS**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int q = 0;
    std::cin >> q;
    OrderedSet values;
    while (q--) {
        int type = 0, x = 0;
        std::cin >> type >> x;
        if (type == 1) values.insert(x);
        else if (type == 2) values.erase(x);
        else if (type == 3) {
            std::cout << values.order_of_key(x) << '\n';
        } else if (x < 0 || x >= static_cast<int>(values.size())) {
            std::cout << "INVALID\n";
        } else {
            auto it = values.find_by_order(
                static_cast<OrderedSet::size_type>(x));
            std::cout << *it << '\n';
        }
    }
}
```

| Sample input | Sample output |
|--------------|---------------|
| 9            | 1             |
| 1 8          | 8             |
| 1 3          | 0             |
| 1 8          | INVALID       |
| 3 8          |               |
| 4 1          |               |
| 2 3          |               |
| 3 8          |               |
| 4 1          |               |
| 2 99         |               |

### A short state trace

| Command | Distinct keys after command | Output  |
|---------|-----------------------------|---------|
| 1 8     | 8                           | —       |
| 1 3     | 3, 8                        | —       |
| 1 8     | 3, 8                        | —       |
| 3 8     | 3, 8                        | 1       |
| 4 0     | 3, 8                        | 3       |
| 2 3     | 8                           | —       |
| 4 1     | 8                           | INVALID |

The invariant is: the tree contains exactly the currently active distinct keys. Insertion preserves uniqueness; erasure removes the requested key or changes nothing. Therefore, each order-statistics query sees exactly the state its command requires.

### Do not save a numerical rank as an identity

A rank can refer to a different value after an update. Store an actual key when you need to remember which item you mean. An ID and a rank solve different problems, just as a value and a vector index do.

For  $q$  commands, time is  $O(q(1 + \log(q + 1)))$  and storage is  $O(d_{\max})$ , where  $d_{\max}$  is the largest number of active distinct keys. Queries do not create new keys.

## 5.29 Optional extension: ranked duplicates with unique IDs

### Make the full keys distinct, not the comparison non-strict

Suppose the values are 5, 2, 5, 8, and both copies of 5 must count. Store (value, id) pairs with IDs 0, 1, 2, and 3. The full keys become (5,0), (2,1), (5,2), (8,3). They are different even when their value fields match.

Pair comparison orders by the first field, then by the second field to break ties. Therefore, equal values appear together, and each occurrence has its own position.<sup>[5-18]</sup>

### Complete program – duplicate values, unique pair keys, GNU PBDS

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <utility>

using Item = std::pair<int, int>;
using OrderedBag = __gnu_pbds::tree<
    Item, __gnu_pbds::null_type, std::less<Item>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    OrderedBag values;
    values.insert({5, 0});
    values.insert({2, 1});
    values.insert({5, 2});
    values.insert({8, 3});
    std::cout << values.order_of_key({5, -1}) << '\n';
    std::cout << values.order_of_key({5, 4}) << '\n';
    auto it = values.find_by_order(2);
    if (it != values.end()) std::cout << it->first << '\n';
    auto one = values.lower_bound({5, -1});
    if (one != values.end() && one->first == 5) {
        Item key = *one;
        values.erase(key);
    }
    std::cout << values.size() << '\n';
}
```

| Sample input | Sample output |
|--------------|---------------|
| (no input)   | 1             |
|              | 3             |
|              | 5             |
|              | 3             |

### Use sentinels justified by the ID range

In this example the only IDs are 0 through 3. Therefore:

| Query                              | Meaning                                            |
|------------------------------------|----------------------------------------------------|
| <code>order_of_key({5, -1})</code> | Number of occurrences with value strictly below 5. |
| <code>order_of_key({5, 4})</code>  | Number of occurrences with value at most 5.        |

No pair with first field 5 is smaller than  $(5, -1)$ , because every valid ID is nonnegative. Every stored pair with first field 5 is smaller than  $(5, 4)$ , because every valid ID is below 4. This explains the output counts 1 and 3.

For IDs  $0, \dots, n-1$ , use  $-1$  and  $n$  as the corresponding boundaries. These are *ID sentinels*, not arbitrary magic numbers. With dynamic insertions, keep IDs unique and choose a valid bound for every ID you have issued. Do not recycle an ID while its old pair is still stored.

### Select and remove the intended occurrence

Zero-based rank 2 is the pair  $(5, 2)$ ; printing its first field gives 5. To erase one 5, the program finds `lower_bound({5, -1})`, checks that the resulting pair really has value 5, copies that full key, and erases it. The other 5 remains.

There is no need to calculate  $x+1$  to count values at most  $x$ . Avoiding that expression also avoids overflow at a maximum representable key. The pair-key technique preserves strict comparison throughout.

#### Pause and predict – 25

With stored pairs  $(4, 0)$ ,  $(4, 1)$ ,  $(9, 2)$ , what is `order_of_key({4, -1})`? What is the value at zero-based rank 1? Why not use `less_equal` instead?

**Check your prediction.** The count is 0, and the value at rank 1 is 4. Both occurrences have their own distinct full key. `less_equal` violates the strict-order requirement; unique pair IDs solve the duplicate problem without breaking that requirement.

## 5.30 Worked problem: sell one affordable ticket

### Translate the selection rule before choosing operations

A ticket office stores individual tickets, including tickets with equal prices. Each customer has a budget. Serve customers in input order and sell the most expensive remaining ticket whose price does not exceed that budget. Print -1 when none is affordable. This is the specified rule for each customer; we are not claiming an optimization over all possible customer orders.

**Input.** Read  $n$ ,  $q$ ,  $n$  prices, and  $q$  budgets. Here  $0 \leq n, q \leq 200000$ , prices lie in  $[1, 10^9]$ , and budgets in  $[0, 10^9]$ .

**Complete program – upper bound, predecessor, erase one**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::multiset<int> prices;
    for (int i = 0; i < n; ++i) {
        int price = 0;
        std::cin >> price;
        prices.insert(price);
    }
    while (q-- > 0) {
        int budget = 0;
        std::cin >> budget;
        auto it = prices.upper_bound(budget);
        if (it == prices.begin()) {
            std::cout << -1 << '\n';
        } else {
            --it;
            std::cout << *it << '\n';
            prices.erase(it);
        }
    }
}
```

#### Sample input

```
4 5
5 3 5 9
5
4
5
8
10
```

#### Sample output

```
5
3
5
-1
9
```

## Trace a budget of 5

With prices 3, 5, 5, 9, `upper_bound(5)` points to 9: the first price that is too expensive. Moving one step back reaches an affordable 5. Erasing that *iterator* sells exactly one ticket. The remaining prices are 3, 5, 9.

If the budget is 2, the upper bound equals `begin()`. There is no previous item, so no ticket is affordable. If the budget is 20, the upper bound is `end()`; decrementing is valid because the bag is nonempty and we already checked it is not `begin()`.

## Why the solution works

Before every customer, the multiset contains exactly the unsold tickets. The upper bound separates affordable prices from prices greater than the budget. If no affordable price exists, it equals `begin()`. Otherwise its predecessor is the greatest affordable price. Erasing precisely that occurrence preserves the invariant for the next customer.

## Cost and common wrong turns

There are  $n$  insertions and at most one bound search and one iterator erasure per customer. Time is  $O((n + q)(1 + \log(n + 1)))$ , and stored ticket space is  $O(n)$ . Repeatedly scanning from the largest ticket down to the budget can be much slower.

A `set` is wrong here because equal-price tickets are still separate tickets. `tickets.erase(price)` is wrong because it would sell every equal-price ticket at once. `lower_bound(budget)` is not the same boundary as the first price *strictly greater* than the budget.

### Pause and predict – 26

Tickets cost 4, 4, 9, and three customers each have budget 4. What outputs should appear? What mistake could make the second answer wrong?

**Check your prediction.** The outputs are 4, 4, and -1. Erasing by key after the first sale would remove both copies of 4, incorrectly making the second answer -1.

## 5.31 Worked problem: two different positions with a target sum

### Ask about the past before adding the present

Given an array and a target, decide whether two *different indices* hold values whose sum is the target. The values may be equal: two different occurrences of 5 form a valid sum of 10, but one occurrence cannot be used twice.

**Input.** Read  $n$  **target**, followed by  $n$  values. Here  $0 \leq n \leq 200000$ , values lie in  $[-10^9, 10^9]$ , and the target lies in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ . Print YES or NO.

**Complete program – look up the complement before insertion**

```
#include <iostream>
#include <unordered_map>

int main() {
    int n = 0;
    long long target = 0;
    std::cin >> n >> target;
    std::unordered_map<long long, int> seen;
    bool possible = false;
    for (int i = 0; i < n; ++i) {
        long long x = 0;
        std::cin >> x;
        if (seen.contains(target - x)) possible = true;
        ++seen[x];
    }
    std::cout << (possible ? "YES" : "NO") << '\n';
}
```

#### Sample input

4 10  
5 1 5 8

#### Sample output

YES

### Why the order of these two lines matters

For the current value  $x$ , look for  $\text{target} - x$  among *previous* values, then record  $x$ . The map therefore never contains the current position during its own query. This makes the two chosen positions distinct without storing their indices.

### Trace the two smallest decisive cases

| Input values, target 10 | Query before insertion                                | Result |
|-------------------------|-------------------------------------------------------|--------|
| 5                       | Complement 5 is absent; then store this 5.            | NO     |
| 5 5                     | First query misses; second query finds the earlier 5. | YES    |

The map's invariant is that its keys are exactly values from already processed positions, with counts of their occurrences. When a complement is found, it came from an earlier index and makes a

valid pair. Conversely, for any valid pair, the later position finds the earlier position's value when processed. Thus the algorithm finds a pair exactly when one exists.

The program keeps reading after a match and remembers `found=true`. This makes the input contract explicit and lets the same logic be used inside a multi-case solution. All target and complement arithmetic uses `long long`; the difference between two allowed input values may exceed a 32-bit signed integer.

### Average versus worst-case analysis

There are  $n$  lookups and insertions. Average work under the hash-container model is  $O(n)$ ; worst-case work can be  $O(n^2)$ . Storage is  $O(d + b)$  for  $d$  distinct keys and  $b$  hash buckets. An ordered map or set gives an alternative  $O(n \log(n + 1))$  key-operation bound. Outputting only existence does not require storing the number of pairs.

#### Pause and predict – 27

Why does inserting `x` before querying `target-x` incorrectly accept the single value 5 for target 10?

**Check your prediction.** The query would find the very same occurrence just inserted. It would use one position twice. Querying the previously seen values first prevents that false match.

## 5.32 Optional extension: composite keys and set combinations

### A key can contain more than one field

Two grid points are identical only when both coordinates match. A `set<pair<int,int>>` naturally stores unique coordinate pairs in lexicographic order. Sorting by the first coordinate does not discard points that share only that coordinate.<sup>[5-18]</sup>

### A different comparator changes the meaning of before

An optional descending declaration is `std::set<int, std::greater<int>>`, with `<functional>` included. For that ordering, `begin()` gives the greatest key. Bound operations still follow the comparator: `lower_bound(x)` is the first element not ordered before `x`; the familiar ascending “at least `x`” wording no longer applies unchanged. Use the ascending form until you can trace the chosen comparison carefully.<sup>[5-1]</sup>

**Task.** Read  $0 \leq n \leq 200000$  points, coordinates in  $[-10^9, 10^9]$ . Print the number of distinct points, then each point in increasing first-coordinate order, using the second coordinate to break ties.

#### Complete program – distinct two-field keys

```
#include <iostream>
#include <set>
#include <utility>

int main() {
    int n = 0;
    std::cin >> n;
    std::set<std::pair<int, int>> points;
    for (int i = 0; i < n; ++i) {
        int x = 0, y = 0;
        std::cin >> x >> y;
        points.insert({x, y});
    }
    std::cout << points.size() << '\n';
    for (const auto& point : points) {
        std::cout << point.first << ' ' << point.second << '\n';
    }
}
```

#### Sample input

```
4
2 1
1 9
2 1
2 0
```

#### Sample output

```
3
1 9
2 0
2 1
```

## Do not change a key while it is stored

The ordering depends on the full pair. Changing a stored key in place would break that ordering; set iterators therefore do not let you assign to its key fields. Erase the old key and insert a new one when a point's coordinates change. A map's mapped value is different: it is not the field used to locate the entry.<sup>[5-1]</sup>

## Distinct intersection: present in both collections

**Task.** Read  $n$   $m$ , then two lists of lengths  $n$  and  $m$ , both at most 200000. Values lie in  $[-10^9, 10^9]$ . Print each value appearing in both lists once, in ascending order. Print an empty line if there are no such values.

**Complete program – membership plus a distinct result set**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, m = 0;
    std::cin >> n >> m;
    std::set<int> first, common;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        first.insert(x);
    }
    for (int i = 0; i < m; ++i) {
        int x = 0;
        std::cin >> x;
        if (first.contains(x)) common.insert(x);
    }
    bool start = true;
    for (int x : common) {
        if (!start) std::cout << ' ';
        std::cout << x;
        start = false;
    }
    std::cout << '\n';
}
```

### Sample input

```
4 5
2 8 2 5
8 8 1 2 9
```

### Sample output

```
2 8
```

The first set records membership in the first list. Each value from the second list enters the result set only if it belongs to the first. Therefore, no unrelated value enters the result, every common value is considered, and duplicate occurrences collapse automatically.

Time is  $O((n+m)(1+\log(n+m+1)))$ ; stored space is bounded by the distinct keys in the first list plus the distinct common keys. A union would insert both lists into one set. A difference would print first-list keys that are absent from the second set. In all three cases, decide whether the problem means distinct values or individual occurrences before selecting a representation.

## 5.33 Optional extension: keep only positive frequency entries

### Distinguish an absent key from a zero-valued record

A frequency map often represents an active bag. For that representation, it is useful to store a key exactly when its frequency is positive. Then `frequency.size()` is the number of currently distinct values.

**Task.** Start empty and process  $0 \leq q \leq 200000$  commands: 1 `x` adds one occurrence; 2 `x` removes one occurrence if present; 3 prints the number of currently distinct values. Keys are in  $[-10^9, 10^9]$ . A removal of an absent value changes nothing.

**Complete program – erase entries whose count becomes zero**

```
#include <iostream>
#include <map>

int main() {
    int q = 0;
    std::cin >> q;
    std::map<int, int> frequency;
    while (q--) {
        int type = 0;
        std::cin >> type;
        if (type == 3) {
            std::cout << frequency.size() << '\n';
            continue;
        }
        int x = 0;
        std::cin >> x;
        if (type == 1) ++frequency[x];
        else {
            auto it = frequency.find(x);
            if (it != frequency.end()) {
                --it->second;
                if (it->second == 0) frequency.erase(it);
            }
        }
    }
}
```

#### Sample input

```
8
1 4
1 4
2 4
3
2 4
3
2 9
3
```

#### Sample output

```
1
0
0
```

## Explain the invariant and the boundary cases

Insertion uses subscript deliberately: a missing count starts at zero and is increased. Removal uses `find`: a missing value must not create an entry. A present count decreases by one, and a zero count causes its whole entry to be erased. Consequently, every stored entry has a positive count after each command.

Not every map should erase zeros. A financial total of zero can be a real record that the output must retain. The correct rule follows the meaning of the data, not a universal preference for nonzero values.

### Pause and predict – 28

A bag represented by frequencies performs add 7, add 7, remove 7, remove 7. What are the distinct counts after each step? Why is leaving 7  $\rightarrow$  0 inconsistent with using map size as the last answer?

**Check your prediction.** The counts are 1, 1, 1, and 0. A stored zero-valued entry still contributes one to `size()`; erase it when this representation means that only positive frequencies are active.

## 5.34 Debugging keyed containers and ranked queries

### Check the representation before changing the code

A correctly written operation can still solve the wrong problem. A set does not count repeated occurrences. A map is sorted by key, not by its mapped score. A hash container has no minimum at `begin()`. An order-statistics tree with distinct keys does not rank every duplicate occurrence.

| Symptom                             | First question to ask                                                    |
|-------------------------------------|--------------------------------------------------------------------------|
| Too few items remain after deletion | Did key erasure remove all equal multiset values?                        |
| Size grows after read-only queries  | Did a missing-key subscript insert an entry?                             |
| Crash on empty/small inputs         | Was <code>end()</code> dereferenced or <code>begin()</code> decremented? |
| Correct sample, slow large input    | Is a count, iterator walk, copy, or nested scan hiding linear work?      |
| Rank answer is off by one           | Is the statement zero-based or one-based? Is the bound strict?           |
| Equal values disappear              | Does the representation store distinct keys rather than occurrences?     |
| Hash output order differs           | Did the statement require sorting that the code never performed?         |
| Later test cases include old values | Is every case given a fresh local container?                             |

### Three wrong fragments to repair

```
// WRONG: end() is not an element.
auto it = values.find(x);
std::cout << *it;

// WRONG: might remove every copy, not one.
bag.erase(x);

// WRONG for read-only membership: creates absent keys.
if (frequency[x] > 0) { /* ... */ }
```

For the first, compare with `end()` before access. For the second, find one occurrence and erase only a valid iterator. For the third, use `find` or `contains`; if the program needs the count, read `it->second` only after a successful find.

### A boundary-testing plan

For each program, predict the answer for an empty collection where allowed; one element; all equal values; all distinct values; absent keys below the minimum, between two keys, and above the maximum; and deletion of the last element. Test multiple identical commands, not just different commands.

For map programs, include a mapped value of zero and repeated assignments to the same key. For rank programs, test a negative request, the first valid rank, the last valid rank, and exactly one

position after it. For hash programs, do not assert one implementation's iteration order unless the code explicitly sorts the output.

### Trace state, then count work

Write the collection after each command before running the program. If the state is correct but an output is wrong, inspect the query and indexing convention. If the state itself is wrong, inspect insertion, erasure, duplicate handling, and resets. Then count how many operations of each kind can execute on a maximum-size input.

#### Core readiness before practice

You should be able to choose between distinct membership, repeated occurrences, key-to-value records, and rank queries. You should also be able to explain why `find` is read-only, why `operator[]` may insert, why the multiset erase overload matters, and why an average hash bound is not a worst-case guarantee.

## 5.35 Practice problems: choose the representation first

Try each task without looking at its solution. Write the input/output contract, choose what each stored key and value mean, trace one boundary case, and estimate the operations before coding.

### Shared constraints and conventions

Unless a task narrows them, lengths  $n$ ,  $m$ , and command/query counts  $q$  are between 0 and 200000. Stored integer values and IDs used as data keys lie in  $[-10^9, 10^9]$ . Query values and thresholds may lie in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ . Requested signed ranks lie in  $[-200000, 200001]$ . Input follows the stated format. Empty input collections are allowed unless explicitly excluded.

For multi-line output, print only the requested answers, not prompts. Words and names use lowercase ASCII English letters. A displayed sample is one example, not the full set of valid inputs.

### A suggested route

Problems 1–8 practise sets and multisets. Problems 9–18 practise maps, hashing, and combined patterns. Problems 19–21 use the GNU rank/select extension. Problem 22 is an optional duplicate-ranking challenge. Every task has a separate hint and an explained solution later in the chapter.

#### Explain your container choice

Before using a container, complete this sentence: “I need to keep *these items*, because the next operation asks for *this information*.” A declaration alone does not explain an algorithm.

## 1. Distinct values in order

Read  $n$ , then  $n$  integers. Print the number of distinct values on the first line. On the second line, print those values in ascending order, separated by spaces. For an empty input list, print 0 and an empty second line.

| Sample input  | Sample output |
|---------------|---------------|
| 6             | 4             |
| 4 -1 4 7 -1 0 | -1 0 4 7      |

*Hint: page 84. Explained solution: page 90.*

## 2. First repeated occurrence

Read  $n$  and  $n$  integers in input order. Print the value whose repeated occurrence is encountered first while scanning left to right. Print **NONE** if no value repeats. This is not necessarily the smallest repeated value or the value with the earliest first occurrence.

| Sample input | Sample output |
|--------------|---------------|
| 6            | 3             |
| 8 3 5 3 8 5  |               |

*Hint: page 84. Explained solution: page 91.*

### 3. How many distinct values are shared?

Read  $n$   $m$ , the first list of  $n$  integers, then the second list of  $m$  integers. Print the number of distinct values that appear in both lists. Repeated copies in either list contribute only once.

---

| Sample input | Sample output |
|--------------|---------------|
|--------------|---------------|

---

|           |   |
|-----------|---|
| 4 5       | 2 |
| 1 2 2 7   |   |
| 2 2 7 7 9 |   |

---

*Hint: page 84. Explained solution: page 92.*

### 4. Next available value

Read  $n$   $q$ , then  $n$  integers and  $q$  query thresholds. For each threshold  $x$ , print the smallest stored value greater than or equal to  $x$ , or **NONE** if it does not exist. Queries do not remove or insert values.

---

| Sample input | Sample output |
|--------------|---------------|
|--------------|---------------|

---

|          |      |
|----------|------|
| 4 3      | 7    |
| 3 7 7 10 | 10   |
| 7        | NONE |
| 8        |      |
| 11       |      |

---

*Hint: page 84. Explained solution: page 93.*

## 5. Active participant IDs

Start with no active IDs. Read  $q$  commands:  $1\ x$  activates ID  $x$ ;  $2\ x$  deactivates it if present;  $3\ x$  prints YES or NO for whether it is active;  $4$  prints the active distinct count. Repeated activation does not add a second copy. Only types 3 and 4 produce output.

| Sample input | Sample output |
|--------------|---------------|
| 8            | 1             |
| 1 7          | YES           |
| 1 7          | NO            |
| 4            | 0             |
| 3 7          |               |
| 2 7          |               |
| 3 7          |               |
| 2 9          |               |
| 4            |               |

*Hint: page 84. Explained solution: page 94.*

## 6. Remove one occurrence per request

Read  $n\ q$ , then  $n$  integers and  $q$  removal requests. For each request  $x$ , remove one occurrence of  $x$  if present, then print the total number of remaining occurrences. A missing request changes nothing.

| Sample input | Sample output |
|--------------|---------------|
| 4 4          | 3             |
| 5 5 2 9      | 3             |
| 5            | 2             |
| 7            | 2             |
| 5            |               |
| 5            |               |

*Hint: page 84. Explained solution: page 95.*

## 7. Affordable ticket sales

Read  $n$   $q$ , then  $n$  ticket prices and  $q$  customer budgets. Prices lie in  $[1, 10^9]$ , budgets in  $[0, 10^9]$ . Serve customers in order: sell one remaining ticket at the greatest price not exceeding the budget and print that price. Print  $-1$  if none is affordable. Equal-price tickets remain separate tickets.

| Sample input | Sample output |
|--------------|---------------|
| 3 4          | 4             |
| 4 4 9        | 4             |
| 4            | -1            |
| 4            | 9             |
| 4            |               |
| 10           |               |

*Hint: page 85. Explained solution: page 96.*

## 8. Remove every matching occurrence

Read  $n$   $x$ , then  $n$  integers. Remove every occurrence equal to  $x$ . Print the remaining occurrence count on the first line and all remaining values in ascending order on the second line, keeping duplicates. Print an empty second line if none remain.

| Sample input | Sample output |
|--------------|---------------|
| 6 5          | 3             |
| 5 1 5 9 1 5  | 1 1 9         |

*Hint: page 85. Explained solution: page 97.*

## 9. A word-frequency report

Read  $n$ , then  $n$  lowercase words. Print one line per distinct word, containing the word and its occurrence count. Lines must be in lexicographic word order. Each word has 1–20 characters. When  $n=0$ , print no lines.

| Sample input               | Sample output |
|----------------------------|---------------|
| 5                          | apple 2       |
| pear apple pear apple plum | pear 2        |
|                            | plum 1        |

*Hint: page 85. Explained solution: page 98.*

## 10. Keep the latest score

Read  $n$   $q$ . Next read  $n$  records, each a lowercase name followed by an integer score. A later record replaces the score for the same name. Then read  $q$  names; print the latest score for each, or **MISSING**. Names have 1–20 letters, and scores lie in  $[-10^9, 10^9]$ . Queries must not create new records.

| Sample input | Sample output |
|--------------|---------------|
| 3 3          | 7             |
| a 0          | 4             |
| b 4          | MISSING       |
| a 7          |               |
| a            |               |
| b            |               |
| c            |               |

*Hint: page 85. Explained solution: page 99.*

### 11. Most frequent, smallest on a tie

Read  $n$  and  $n$  integers, with  $1 \leq n \leq 200000$ . Print the most frequent value and its frequency. If several values share the greatest frequency, choose the smallest value.

---

| Sample input       | Sample output |
|--------------------|---------------|
| 6<br>-3 7 7 -3 9 9 | -3 2          |

---

*Hint: page 85. Explained solution: page 100.*

### 12. Total amount per name

Read  $n$ , then  $n$  records of a lowercase name and a signed amount. Names have 1–20 letters; each amount lies in  $[-10^9, 10^9]$ . Sum amounts for each name. Print every recorded name and its total in lexicographic order. Keep names whose total is zero.

---

| Sample input                              | Sample output    |
|-------------------------------------------|------------------|
| 4<br>aya 100<br>ali -5<br>aya 25<br>ali 5 | ali 0<br>aya 125 |

---

*Hint: page 85. Explained solution: page 101.*

### 13. First value occurring once

Read  $n$  and  $n$  integers. Print the first value in input order whose total frequency is exactly one. Print NONE if no such value exists. The answer is not necessarily the smallest unique value.

| Sample input | Sample output |
|--------------|---------------|
| 6            | 7             |
| 8 2 8 7 2 3  |               |

*Hint: page 86. Explained solution: page 102.*

### 14. Count without inserting

Read  $n$   $q$ ,  $n$  integers, then  $q$  query values. Use an `unordered_map` to print the frequency of each queried value, using zero for missing values. After all queries, print the number of distinct values in the original input. Queries must not add entries.

| Sample input | Sample output |
|--------------|---------------|
| 4 3          | 0             |
| 6 6 1 9      | 2             |
| 7            | 0             |
| 6            | 3             |
| 8            |               |

*Hint: page 86. Explained solution: page 103.*

### 15. Same letters, same counts

Read two nonempty lowercase English words, each of length at most 200000. Print **YES** if one can be rearranged into the other, otherwise **NO**. Repeated letters must appear equally often. Use maps for counts; no frequency array is required.

---

**Sample input****Sample output**

---

silent listenYES

---

*Hint: page 86. Explained solution: page 104.*

### 16. Group original positions by value

Read  $n$  and  $n$  integers. For each distinct value in ascending value order, print the value, its occurrence count, and all of its one-based positions in increasing position order. Put one group on each line. For  $n=0$ , print no lines.

---

**Sample input****Sample output**

---

5

1 2 2 4

3 1 3 1 3

3 3 1 3 5

---

*Hint: page 86. Explained solution: page 105.*

## 17. Fresh distinct counts

Read  $1 \leq T \leq 200000$ . For each test case, read  $n$  and  $n$  integers and print its distinct count. Each length is at most 200000, and the sum of all lengths is at most 200000. Cases are independent.

| Sample input | Sample output |
|--------------|---------------|
| 3            | 2             |
| 3            | 1             |
| 5 5 8        | 0             |
| 1            |               |
| 5            |               |
| 0            |               |

*Hint: page 86. Explained solution: page 106.*

## 18. A target sum from different indices

Read  $n$  **target**, followed by  $n$  integers. Values lie in  $[-10^9, 10^9]$ , and **target** in  $[-2 \cdot 10^9, 2 \cdot 10^9]$ . Print **YES** exactly when two different input positions have values summing to **target**. Otherwise print **NO**. Equal values at different positions are allowed.

| Sample input | Sample output |
|--------------|---------------|
| 3 12         | YES           |
| 6 1 6        |               |

*Hint: page 86. Explained solution: page 107.*

## 19. Rank of a threshold (GNU PBDS)

Use the GNU order-statistics set. Read  $n$   $q$ , then  $n$  integers and  $q$  thresholds. For each threshold, print the number of distinct input values strictly smaller than it. Repeated input values count once, and queries do not insert keys.

| Sample input | Sample output |
|--------------|---------------|
| 5 3          | 1             |
| 4 4 9 1 7    | 2             |
| 4            | 4             |
| 5            |               |
| 10           |               |

*Hint: page 87. Explained solution: page 108.*

## 20. The k-th distinct value (GNU PBDS)

Use the GNU order-statistics set. Read  $n$   $q$ , then  $n$  integers and  $q$  signed requests  $k$ . Print the *one-based*  $k$ -th smallest distinct value, or INVALID when  $k < 1$  or  $k$  exceeds the distinct count.

| Sample input | Sample output |
|--------------|---------------|
| 5 4          | 1             |
| 8 3 8 1 3    | 8             |
| 1            | INVALID       |
| 3            | INVALID       |
| 4            |               |
| 0            |               |

*Hint: page 87. Explained solution: page 109.*

## 21. An active ranked set (GNU PBDS)

Start with an empty GNU order-statistics set. Read  $q$  two-integer commands: 1  $x$  inserts  $x$ ; 2  $x$  erases it if present; 3  $x$  prints the number of active keys strictly less than  $x$ ; 4  $k$  prints the *one-based*  $k$ -th smallest active key or `INVALID`. Duplicates are not stored.

| Sample input | Sample output |
|--------------|---------------|
| 8            | 6             |
| 1 6          | 0             |
| 1 2          | INVALID       |
| 1 6          |               |
| 4 2          |               |
| 2 2          |               |
| 3 6          |               |
| 4 2          |               |
| 2 99         |               |

*Hint: page 87. Explained solution: page 110.*

## 22. Rank individual occurrences (optional GNU challenge)

Read  $n$   $q$ , then  $n$  integers, keeping duplicates. Next process two-integer commands: 1  $x$  prints the number of input occurrences strictly smaller than  $x$ ; 2  $k$  prints the *one-based*  $k$ -th smallest occurrence or `INVALID`. There are no updates. Use unique pair IDs in a GNU order-statistics tree.

| Sample input | Sample output |
|--------------|---------------|
| 4 4          | 1             |
| 5 2 5 8      | 5             |
| 1 5          | INVALID       |
| 2 3          | 3             |
| 2 5          |               |
| 1 8          |               |

*Hint: page 87. Explained solution: page 111.*

## 5.36 Trace, tool-choice, and complexity drills

Do these on paper first. All fragments belong inside `main` with the relevant headers. Assume the ordinary ascending comparator unless a drill explicitly says otherwise. A fragment labelled wrong is not a complete working program.

### Drill 1 – A duplicate insertion is still an operation

```
std::set<int> s = {4, 1, 4};
auto result = s.insert(1);
s.insert(7);
s.erase(4);
std::cout << result.second << ' ' << s.size() << '\n';
for (int x : s) std::cout << x << ' ';
```

Write the two output lines, ignoring the final trailing space. Does erasing key 4 invalidate the iterator stored in `result.first`?

### Drill 2 – Two erase overloads, two different effects

```
std::multiset<int> bag = {2, 2, 2, 5};
auto it = bag.find(2);
bag.erase(it);
std::cout << bag.count(2) << ' ' << bag.size() << '\n';
auto removed = bag.erase(2);
std::cout << removed << ' ' << bag.size() << '\n';
```

Write the output and the bag contents after each erasure. Explain why neither erasure removes the 5.

### Drill 3 – A query that secretly writes

```
std::map<int, int> count;
count[6] = 0;
std::cout << count.contains(6) << '\n';
std::cout << count[8] << '\n';
std::cout << count.count(8) << '\n';
std::cout << count.size() << '\n';
```

Predict all four lines. Explain the difference between `count.count(8)` and `count[8]`. Which expression creates an entry?

### Drill 4 – Key order is not score order

```
std::map<std::string, int> score;
score.insert({"nour", 80});
score.insert({"nour", 95});
score["ali"] = 100;
score["nour"] = 90;
for (const auto& entry : score) {
    std::cout << entry.first << ' ' << entry.second << '\n';
}
```

Write the output. Did the second `insert` replace 80? Would the traversal become score-descending if the name `nour` were assigned 200?

### Drill 5 – Three kinds of value boundary

A multiset contains 2, 5, 5, 9. For each threshold 1, 5, 6, 9, 10, write the value at `lower_bound`, the value at `upper_bound`, and the greatest value not exceeding the threshold. Use NONE when no answer exists. How many occurrences are in `equal_range(5)`?

### Drill 6 – Erase safely while traversing

```
// WRONG: an erased iterator is used again.
for (auto it = s.begin(); it != s.end(); ++it) {
    if (*it < 0) s.erase(it);
}
```

Rewrite the loop so it removes every negative key from a `std::set<int>`. Explain what the returned erase iterator means and why `++it` must not happen unconditionally.

### Drill 7 – Nested output does not imply a quadratic bound

A map groups `n` input indices by value. There are `d` map entries, and all stored vectors together contain exactly `n` indices.

```
for (const auto& entry : positions) {
    for (int index : entry.second) {
        std::cout << index << ' ';
    }
}
```

How many times does the inner output statement execute? What copying would occur if the outer declaration were `auto entry` rather than `const auto& entry`?

## Drill 8 – Hash counting and an ordered report

You count  $n$  words in an `unordered_map<string,int>`, obtaining  $d$  entries. The report must list words in lexicographic order. Explain why directly printing the map is insufficient. Give two valid approaches, and estimate the work of hash counting followed by sorting copied entries. Assume bounded word length.

## Drill 9 – A rehash changes iterator validity

```
std::unordered_map<int, int> score;
score[7] = 10;
auto it = score.find(7);
int& value = score.at(7);
score.rehash(1000);
value += 5;
```

Can the program safely use `value` after the rehash? Can it safely continue using `it`? Write a safe statement to print key 7's mapped value after the rehash.

## Drill 10 – The hidden cost of repeatedly counting duplicates

A multiset contains  $n$  copies of 4. The following code runs once:

```
long long total = 0;
for (int x : bag) {
    total += bag.count(x);
}
```

Find the final total and worst-case time growth. A frequency map stores one entry  $4 \rightarrow n$ ; explain how that representation can produce this total without visiting each duplicate group repeatedly. Assume  $1 \leq n \leq 200000$ .

### Drill 11 – Rank, selection, and distinctness

A GNU integer order-statistics tree receives insertions 8, 3, 8, 12. Find the answers to `order_of_key(8)`, `order_of_key(9)`, and `find_by_order(1)` (report the key). Then erase 3 and answer the same questions again. Does one-based request `k=3` remain valid after deletion? Would a pair-ID tree keeping both copies of 8 have the same size as this tree?

### Drill 12 – Choose a tool and name one pitfall

For each requirement, choose a suitable chapter tool and state one common mistake to avoid.

- a) Keep active distinct IDs, print them in increasing order, and find the first ID at least a threshold.
- b) Sell one cheapest available ticket at a time, keeping equal-price tickets as separate items.
- c) Associate each name with its latest score and print names in alphabetical order.
- d) Answer exact-key numeric lookups when an average-time hash bound is acceptable and no ordered queries are needed.
- e) Maintain active distinct numbers under insertions and deletions while answering k-th-smallest queries.
- f) Rank all occurrences of numbers, including equal values, with the GNU tree extension.

Would the choice in part (d) automatically satisfy a requirement for a logarithmic *worst-case* lookup guarantee?

#### After each answer

State whether a query mutates the collection. For complexity, count every visited element, copied value, and output item. Do not replace an explanation with a container name alone.

## 5.37 Hints before solutions

Use only enough of a hint to restart your own attempt. Once the idea is clear, return to the problem and write the program before reading its full solution.

### Problem 1: Distinct values in order

Which container automatically removes repeated keys while preserving ascending traversal?

### Problem 2: First repeated occurrence

Remember previously seen values. Stop changing the answer after the first insertion that reports an existing key.

### Problem 3: How many distinct values are shared?

Use one set for membership in the first list and another to collect common values without repetitions.

### Problem 4: Next available value

The requested boundary includes equality. Which member function finds the first value that is not smaller?

### Problem 5: Active participant IDs

A participant is either active or inactive; multiplicity is not part of the state.

### Problem 6: Remove one occurrence per request

Find a single occurrence before choosing the erase overload.

**Problem 7: Affordable ticket sales**

Find the first price strictly above the budget; the answer, when one exists, is immediately before it.

**Problem 8: Remove every matching occurrence**

This task intentionally wants the key-erasure behavior that was wrong for the previous removal task.

**Problem 9: A word-frequency report**

A word is a key; its count is a mapped value. An ordered map already provides the output order.

**Problem 10: Keep the latest score**

Use assignment while reading records and a read-only search while answering queries.

**Problem 11: Most frequent, smallest on a tie**

Build frequencies in ascending key order and replace the current answer only for a strictly larger frequency.

**Problem 12: Total amount per name**

The map stores accumulated totals, not the latest individual amount. Choose a wide enough mapped type.

**Problem 13: First value occurring once**

You need total frequencies and the original order. Keep the input sequence for a second pass.

**Problem 14: Count without inserting**

Use subscript while counting, but `find` while querying. A missing lookup should leave size unchanged.

**Problem 15: Same letters, same counts**

Compare how many times each character occurs, not merely whether both words contain the same distinct letters.

**Problem 16: Group original positions by value**

A key can map to a vector. Append the current position when that key is read.

**Problem 17: Fresh distinct counts**

Put the set inside `solve()` so that every case begins empty.

**Problem 18: A target sum from different indices**

For the current value, search for the complement among previous values before recording the current one.

**Problem 19: Rank of a threshold (GNU PBDS)**

The threshold is a key value, not a requested rank. Use the query that returns a count.

**Problem 20: The k-th distinct value (GNU PBDS)**

Validate the one-based request against the number of distinct keys, then convert it to zero-based rank once.

**Problem 21: An active ranked set (GNU PBDS)**

Keep update commands separate from query commands, and do not confuse the lesson's zero-based simulator with this one-based task.

**Problem 22: Rank individual occurrences (optional GNU challenge)**

Use the input index as a unique second field. All IDs are nonnegative, so  $(x, -1)$  precedes every pair with value  $x$ .

## Hints for the trace and complexity drills

1. Separate the returned insertion flag from the resulting set size.
2. Iterator erasure removes one occurrence; key erasure removes the matching group.
3. Track the set of keys after every expression, including expressions used for reading.
4. A map's iteration order follows its keys. Assignment and insertion are different updates.
5. Draw the first value at least the threshold and the first value strictly greater than it.
6. Never advance an iterator after erasing the element it referred to.
7. Sum all group lengths rather than multiplying the number of groups by the full input length.
8. Unordered traversal provides no ascending-order guarantee; separate counting from formatting.
9. Rehashing invalidates iterators, not references to surviving elements.
10. How many equal elements does one count call inspect, and how many times is it repeated?
11. Draw the sorted distinct keys, label ranks from zero, and count only keys strictly before the query.
12. Choose based on duplicates, key/value data, sorted boundaries, rank queries, and guarantees.

## 5.38 Explained practice solutions

These solutions are references, not templates to memorize. Compare the representation, update rule, boundary checks, and complexity with your own attempt. Afterwards, close the solution and reconstruct it from the idea.

For ordered-container estimates,  $d$  denotes the number of distinct keys. The extra  $1 + \log(\cdot + 1)$  form keeps bounds meaningful for empty collections. For hash maps,  $b$  denotes the number of buckets; average-time claims are labelled rather than presented as worst-case guarantees.

## Solution 1 – Distinct values in order

**Idea.** Insert every value into a set, then print its size and traverse it.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0;
    std::cin >> n;
    std::set<int> distinct;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        distinct.insert(x);
    }
    std::cout << distinct.size() << '\n';
    bool first = true;
    for (int x : distinct) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    std::cout << '\n';
}
```

**Why it works.** Each input value enters the set exactly once as a distinct key. Repeated insertions do not create extra entries, and traversal produces the required ascending order.

**Cost.**  $O(n(1 + \log(n + 1)))$  time and  $O(d)$  stored space for  $d$  distinct keys.

**Check carefully.** Test no values, one value, all equal values, and negative values. The second line still exists when the set is empty.

## Solution 2 – First repeated occurrence

**Idea.** Track distinct seen values and record the first input value whose insertion fails.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, answer = 0;
    bool found = false;
    std::cin >> n;
    std::set<int> seen;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        auto result = seen.insert(x);
        if (!result.second && !found) {
            answer = x;
            found = true;
        }
    }
    if (found) std::cout << answer << '\n';
    else std::cout << "NONE\n";
}
```

**Why it works.** An unsuccessful insertion means the same value occurred earlier. Processing left to right makes the first such event the earliest repeated occurrence. A separate boolean prevents later repeats from replacing it.

**Cost.**  $O(n(1 + \log(n + 1)))$  time and  $O(d)$  stored space.

**Check carefully.** Compare 8 3 5 3 8 with 8 8 3 3. The answers are 3 and 8. Do not use a numeric sentinel that could also be a valid input value.

### Solution 3 – How many distinct values are shared?

**Idea.** Insert first-list values into one set, then insert a second-list value into the common set only when the first set contains it.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, m = 0;
    std::cin >> n >> m;
    std::set<int> first, common;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        first.insert(x);
    }
    for (int i = 0; i < m; ++i) {
        int x = 0;
        std::cin >> x;
        if (first.contains(x)) common.insert(x);
    }
    std::cout << common.size() << '\n';
}
```

**Why it works.** A value enters the common set precisely when it appears in both lists. Set uniqueness ensures each common value contributes one to its size.

**Cost.**  $O((n + m)(1 + \log(n + m + 1)))$  time and  $O(n + m)$  stored space as a safe upper bound.

**Check carefully.** Either list may be empty. 5 5 5 shared with 5 5 contributes 1, not 2 or 3.

## Solution 4 – Next available value

**Idea.** Build a set and use its member `lower_bound` for every threshold.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        auto it = values.lower_bound(x);
        if (it == values.end()) std::cout << "NONE\n";
        else std::cout << *it << '\n';
    }
}
```

**Why it works.** In ascending order, the returned iterator identifies the first key at least the threshold. If it is `end()`, no such key is stored. Repetitions do not affect the requested value.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(d)$  stored space.

**Check carefully.** Test an exact match, a threshold between keys, one below the minimum, one above the maximum, and an empty set.

## Solution 5 – Active participant IDs

**Idea.** Use a set as the active-state record. Map each command to insertion, key erasure, membership, or size.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int q = 0;
    std::cin >> q;
    std::set<int> active;
    while (q-- > 0) {
        int type = 0;
        std::cin >> type;
        if (type == 4) {
            std::cout << active.size() << '\n';
            continue;
        }
        int id = 0;
        std::cin >> id;
        if (type == 1) active.insert(id);
        else if (type == 2) active.erase(id);
        else std::cout << (active.contains(id) ? "YES" : "NO")
            << '\n';
    }
}
```

**Why it works.** The set starts empty. Each update adds or removes exactly the specified ID while maintaining uniqueness, so membership and size queries directly describe the required state.

**Cost.**  $O(q(1 + \log(q + 1)))$  time and  $O(d_{\max})$  space for the largest active set.

**Check carefully.** Type 4 has no second input. Test repeated activation, repeated deactivation, and a query before any activation.

## Solution 6 – Remove one occurrence per request

**Idea.** Store all occurrences in a multiset. Find the requested value and erase only that iterator when found.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::multiset<int> bag;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        bag.insert(x);
    }
    while (q-- > 0) {
        int x = 0;
        std::cin >> x;
        auto it = bag.find(x);
        if (it != bag.end()) bag.erase(it);
        std::cout << bag.size() << '\n';
    }
}
```

**Why it works.** The multiset initially contains one element per input occurrence. Every valid request removes exactly one matching element; a missing request removes none. Its size is therefore the required remaining count.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(n)$  stored space.

**Check carefully.** Test more removal requests for a value than there are copies. Key erasure would remove too many elements at the first request.

## Solution 7 – Affordable ticket sales

**Idea.** Use a multiset and the predecessor of `upper_bound(budget)`, guarded against `begin()`.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::multiset<int> available;
    for (int i = 0; i < n; ++i) {
        int price = 0;
        std::cin >> price;
        available.insert(price);
    }
    while (q--) {
        int budget = 0;
        std::cin >> budget;
        auto it = available.upper_bound(budget);
        if (it == available.begin()) std::cout << -1 << '\n';
        else {
            --it;
            std::cout << *it << '\n';
            available.erase(it);
        }
    }
}
```

**Why it works.** All affordable prices lie before the upper bound. Its predecessor is therefore the greatest affordable price. Iterator erasure removes only the sold ticket, preserving the remaining inventory.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(n)$  stored space.

**Check carefully.** Test zero tickets, budgets below every price, equal prices, and an affordable ticket at the exact budget.

## Solution 8 – Remove every matching occurrence

**Idea.** Build a multiset, erase by the requested key, then print its size and remaining elements.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

int main() {
    int n = 0, target = 0;
    std::cin >> n >> target;
    std::multiset<int> bag;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        bag.insert(x);
    }
    bag.erase(target);
    std::cout << bag.size() << '\n';
    bool first = true;
    for (int x : bag) {
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    std::cout << '\n';
}
```

**Why it works.** Key erasure removes exactly the full equal-key group. All other occurrences remain, and ordered traversal prints them including repetitions.

**Cost.**  $O(n(1+\log(n+1)))$  total time and  $O(n)$  stored space. The erasure itself costs  $O(\log(n+1)+k)$  for  $k$  removed copies.

**Check carefully.** Test no matches, all values matching, and several equal nonmatching values that must remain repeated.

## Solution 9 – A word-frequency report

**Idea.** Increase a map entry for each word, then traverse the map.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<std::string, int> counts;
    for (int i = 0; i < n; ++i) {
        std::string word;
        std::cin >> word;
        ++counts[word];
    }
    for (const auto& entry : counts) {
        std::cout << entry.first << ' ' << entry.second << '\n';
    }
}
```

**Why it works.** Each occurrence increases only its own word count. Missing words begin at zero by intentional subscript insertion. Traversal lists each distinct word once in key order.

**Cost.** With the 20-character key bound,  $O(n(1 + \log(n+1)))$  time and  $O(d)$  entries. Arbitrary-length words require accounting for their characters.

**Check carefully.** Test **a**, **aa**, repeated words, and no words. Lexicographic order is not order by word length or by frequency.

## Solution 10 – Keep the latest score

**Idea.** Assign each name's score through the map, and answer queries with `find`.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::map<std::string, int> records;
    for (int i = 0; i < n; ++i) {
        std::string key;
        int value = 0;
        std::cin >> key >> value;
        records[key] = value;
    }
    while (q--) {
        std::string key;
        std::cin >> key;
        auto it = records.find(key);
        if (it == records.end()) std::cout << "MISSING\n";
        else std::cout << it->second << '\n';
    }
}
```

**Why it works.** Every assignment makes the stored score the latest seen score for that name. A successful find therefore returns the required record, while a missing find correctly reports absence without inventing a score.

**Cost.** With bounded name lengths,  $O((n + q)(1 + \log(n + 1)))$  time and  $O(d)$  stored entries.

**Check carefully.** A real score of zero is not missing. Test repeated names, negative scores, and repeated queries for an absent name.

## Solution 11 – Most frequent, smallest on a tie

**Idea.** Count with an ordered map, then scan entries in ascending key order, keeping the first maximum-frequency key.

**Complete solution – C++23**

```
#include <iostream>
#include <map>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<int, int> counts;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++counts[x];
    }
    int value = 0, best = 0;
    for (const auto& entry : counts) {
        if (entry.second > best) {
            value = entry.first;
            best = entry.second;
        }
    }
    std::cout << value << ' ' << best << '\n';
}
```

**Why it works.** Every mapped count is exact. Ascending traversal encounters the smallest key in a tied group first. Using a strict frequency comparison prevents a later larger tied key from replacing it.

**Cost.**  $O(n(1 + \log(n + 1)))$  time and  $O(d)$  stored entries.

**Check carefully.** The nonempty-input constraint guarantees a winner. Test all distinct values, equal winning frequencies, and only negative values.

## Solution 12 – Total amount per name

**Idea.** Use `map<string, long long>` and add each amount to its name's running total.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<std::string, long long> totals;
    for (int i = 0; i < n; ++i) {
        std::string name;
        long long amount = 0;
        std::cin >> name >> amount;
        totals[name] += amount;
    }
    for (const auto& entry : totals) {
        std::cout << entry.first << ' ' << entry.second << '\n';
    }
}
```

**Why it works.** After every record, each stored total is the sum of all processed amounts for that name. Traversal prints every recorded name once; a zero total remains a valid record.

**Cost.** With bounded names,  $O(n(1+\log(n+1)))$  time and  $O(d)$  entries. A total can reach magnitude  $2 \cdot 10^{14}$ , so use `long long`.

**Check carefully.** Test cancellation to zero, repeated large positive amounts, and negative totals. Assignment would discard earlier amounts.

## Solution 13 – First value occurring once

**Idea.** Store the values in a vector while building a frequency map, then scan the vector for the first count of one.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    std::map<int, int> counts;
    for (int& x : values) {
        std::cin >> x;
        ++counts[x];
    }
    for (int x : values) {
        if (counts.at(x) == 1) {
            std::cout << x << '\n';
            return 0;
        }
    }
    std::cout << "NONE\n";
}
```

**Why it works.** The first pass knows the complete frequency of every value. The second pass checks that property in input order, so the first accepted value is exactly the requested one.

**Cost.**  $O(n(1 + \log(n + 1)))$  time and  $O(n + d)$  stored space, including the input vector.

**Check carefully.** Test all repeated values, an answer at the final position, and 9 2 2 4, whose answer is 9 rather than 4.

## Solution 14 – Count without inserting

**Idea.** Build hash-map frequencies and answer each query through a checked iterator.

**Complete solution – C++23**

```
#include <iostream>
#include <unordered_map>

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::unordered_map<int, int> counts;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++counts[x];
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        auto it = counts.find(x);
        if (it == counts.end()) std::cout << 0 << '\n';
        else std::cout << it->second << '\n';
    }
    std::cout << counts.size() << '\n';
}
```

**Why it works.** Construction counts every occurrence. Successful queries read those counts; unsuccessful queries print zero without mutation. Therefore, the final size still counts only original distinct keys.

**Cost.** Average  $O(n + q)$  time; worst-case  $O((n + q)(n + 1))$  as a safe bound. Stored space is  $O(d + b)$  for entries and buckets.

**Check carefully.** An empty input followed by missing queries should print zeros and a final distinct count of zero.

## Solution 15 – Same letters, same counts

**Idea.** Build two `map<char,int>` frequency maps and compare them for equality.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string first, second;
    std::cin >> first >> second;
    std::map<char, int> a, b;
    for (char ch : first) ++a[ch];
    for (char ch : second) ++b[ch];
    std::cout << (a == b ? "YES" : "NO") << '\n';
}
```

**Why it works.** Rearrangement changes positions but not character multiplicities. Equal maps mean every character has the same count in both words, which is exactly the condition for such a rearrangement.

**Cost.**  $O((a+b)(1+\log(26))) = O(a+b)$  counting time for word lengths  $a, b$ , and at most 26 entries per map. The supplied program also stores both input strings, using  $O(a+b)$  space.

**Check carefully.** `aab` and `abb` have the same distinct letters but are not anagrams. Different word lengths cannot be accepted.

## Solution 16 – Group original positions by value

**Idea.** Use `map<int, vector<int>>`; append positions during the left-to-right input pass.

**Complete solution – C++23**

```
#include <iostream>
#include <map>
#include <vector>

int main() {
    int n = 0;
    std::cin >> n;
    std::map<int, std::vector<int>> groups;
    for (int position = 1; position <= n; ++position) {
        int x = 0;
        std::cin >> x;
        groups[x].push_back(position);
    }
    for (const auto& entry : groups) {
        std::cout << entry.first << ' ' << entry.second.size();
        for (int p : entry.second) std::cout << ' ' << p;
        std::cout << '\n';
    }
}
```

**Why it works.** Each position is appended to exactly its own value's vector. Input order makes every vector already ordered by position, and map traversal provides ordered value groups.

**Cost.**  $O(n(1 + \log(n + 1)))$  time, including amortized vector appends and  $O(n)$  output. Space is  $O(n + d)$ , not  $O(d)$  alone.

**Check carefully.** The positions in output start at 1. The nested output loops print a total of **n** positions, not **n** per distinct key.

## Solution 17 – Fresh distinct counts

**Idea.** Create a local set for each case and print its size after inserting that case's values.

**Complete solution – C++23**

```
#include <iostream>
#include <set>

void solve() {
    int n = 0;
    std::cin >> n;
    std::set<int> values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    std::cout << values.size() << '\n';
}

int main() {
    int tests = 0;
    std::cin >> tests;
    while (tests--) solve();
}
```

**Why it works.** Only the current case's input can enter its local set. Uniqueness gives that case's distinct count, and destruction after the call prevents state leaking to later cases.

**Cost.** For total length  $N$ , time is  $O(T + N \log(N + 1))$ . Peak set storage is bounded by the largest case, not the sum of all cases.

**Check carefully.** Test several empty cases and overlapping keys across cases. An earlier case must never change a later answer.

## Solution 18 – A target sum from different indices

**Idea.** Maintain a hash map of previously seen values; query `target-x`, then insert `x`.

**Complete solution – C++23**

```
#include <iostream>
#include <unordered_map>

int main() {
    int n = 0;
    long long target = 0;
    std::cin >> n >> target;
    std::unordered_map<long long, int> counts;
    bool found = false;
    for (int i = 0; i < n; ++i) {
        long long x = 0;
        std::cin >> x;
        if (counts.contains(target - x)) found = true;
        ++counts[x];
    }
    std::cout << (found ? "YES" : "NO") << '\n';
}
```

**Why it works.** Every successful lookup uses an earlier position, so it cannot pair an element with itself. Any valid pair is detected when its later position is processed.

**Cost.** Average  $O(n)$  time and worst-case  $O(n^2)$ ; space  $O(d + b)$ . Use `long long` for complement arithmetic.

**Check carefully.** The input 5 with target 10 is NO, while 5 5 is YES. Test zero elements and negative complements.

## Solution 19 – Rank of a threshold (GNU PBDS)

**Idea.** Insert the input into the configured GNU tree and answer each query with `order_of_key`.

**Environment:** GNU PBDS extension; not a portable standard C++23 container.

**Complete solution – C++23**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    OrderedSet values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int x = 0;
        std::cin >> x;
        std::cout << values.order_of_key(x) << '\n';
    }
}
```

**Why it works.** The tree represents exactly the distinct input keys in ascending order. Its rank query counts precisely the represented keys before the threshold, whether or not that threshold is present.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(d)$  tree storage.

**Check carefully.** GNU PBDS headers are required. An empty tree returns zero for every threshold; equality is excluded from the count.

## Solution 20 – The k-th distinct value (GNU PBDS)

**Idea.** Check  $1 \leq k \leq d$ , then dereference `find_by_order(k-1)`.

**Environment:** GNU PBDS extension; not a portable standard C++23 container.

**Complete solution – C++23**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    OrderedSet values;
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        values.insert(x);
    }
    while (q--) {
        int k = 0;
        std::cin >> k;
        if (k < 1 || k > static_cast<int>(values.size())) {
            std::cout << "INVALID\n";
        } else {
            auto it = values.find_by_order(
                static_cast<OrderedSet::size_type>(k - 1));
            std::cout << *it << '\n';
        }
    }
}
```

**Why it works.** The tree omits duplicate keys and ranks remaining keys from zero. Subtracting one maps a valid one-based request to the same sorted position. Invalid requests are rejected before dereferencing.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(d)$  tree storage.

**Check carefully.** GNU PBDS headers are required. Test  $k=0$ ,  $k=1$ ,  $k=d$ , and  $k=d+1$ . Original length is not the distinct count.

## Solution 21 – An active ranked set (GNU PBDS)

**Idea.** Use the tree as the active distinct set. Query rank directly and convert only the one-based selection command.

**Environment:** GNU PBDS extension; not a portable standard C++23 container.

**Complete solution – C++23**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using OrderedSet = __gnu_pbds::tree<
    int, __gnu_pbds::null_type, std::less<int>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int q = 0;
    std::cin >> q;
    OrderedSet values;
    while (q-- > 0) {
        int type = 0, x = 0;
        std::cin >> type >> x;
        if (type == 1) values.insert(x);
        else if (type == 2) values.erase(x);
        else if (type == 3) {
            std::cout << values.order_of_key(x) << '\n';
        } else if (x < 1 || x > static_cast<int>(values.size())) {
            std::cout << "INVALID\n";
        } else {
            auto it = values.find_by_order(
                static_cast<OrderedSet::size_type>(x - 1));
            std::cout << *it << '\n';
        }
    }
}
```

**Why it works.** Updates preserve exactly the active keys. Every query sees their current ordering; validation ensures a selection names an existing position before converting it to zero-based rank.

**Cost.**  $O(q(1 + \log(q + 1)))$  time and  $O(d_{\max})$  storage.

**Check carefully.** GNU PBDS headers are required. Test repeated insertions, absent erasures, selections after deletion, and queries on the empty initial state.

## Solution 22 – Rank individual occurrences (optional GNU challenge)

**Idea.** Store (value,id) pairs with IDs 0 through  $n-1$ . Rank by  $(x,-1)$ ; select rank  $k-1$  and print only the value field.

**Environment:** GNU PBDS extension; not a portable standard C++23 container.

**Complete solution – C++23**

```
#include <functional>
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <utility>

using Item = std::pair<int, int>;
using OrderedBag = __gnu_pbds::tree<
    Item, __gnu_pbds::null_type, std::less<Item>,
    __gnu_pbds::rb_tree_tag,
    __gnu_pbds::tree_order_statistics_node_update>;

int main() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    OrderedBag values;
    for (int id = 0; id < n; ++id) {
        int x = 0;
        std::cin >> x;
        values.insert({x, id});
    }
    while (q--) {
        int type = 0, x = 0;
        std::cin >> type >> x;
        if (type == 1) {
            std::cout << values.order_of_key({x, -1}) << '\n';
        } else if (x < 1 || x > n) {
            std::cout << "INVALID\n";
        } else {
            auto it = values.find_by_order(
                static_cast<OrderedBag::size_type>(x - 1));
            std::cout << it->first << '\n';
        }
    }
}
```

**Why it works.** Every occurrence has a distinct full key. Lexicographic ordering groups equal values but preserves their separate ranks. The sentinel excludes all pairs whose value equals the threshold, giving the required strict occurrence count.

**Cost.**  $O((n + q)(1 + \log(n + 1)))$  time and  $O(n)$  tree storage.

**Check carefully.** GNU PBDS headers are required. With 4 4 9, the second smallest occurrence is 4. Never substitute a non-strict comparator for unique IDs.

## 5.39 Explained trace and complexity drills

### Drill 1 – A duplicate insertion is still an operation

The first line is 0 2; the second is 1 7. The initial set is 1, 4; inserting 1 returns an existing iterator with `second=false`. Inserting 7 yields 1, 4, 7; erasing 4 leaves 1, 7.

`result.first` points to key 1, not to key 4. Ordinary set insertion does not invalidate it, and erasing a different element does not invalidate it either. The iterator remains valid. An iterator directly to the erased 4 would not remain valid.<sup>[5-1]</sup>

### Drill 2 – Two erase overloads, two different effects

The output is:

```
2 3
2 1
```

Iterator erasure removes one 2, leaving 2, 2, 5. Thus the count of 2 is 2 and the total size is 3. Key erasure then removes both remaining 2s, returning the removed count 2 and leaving just 5. Neither operation targets 5.

A common wrong analysis claims the first erasure removes every matching value merely because it came from `find(2)`. The parameter type selects the overload: an iterator identifies one element, while a key identifies the equal-key group.

### Drill 3 – A query that secretly writes

The four lines are 1, 0, 1, and 2. Key 6 exists even though its mapped value is zero. Subscript access to absent key 8 creates the entry 8 -> 0 and returns that zero. Afterward, `count(8)` reports one matching *entry*, not its mapped numeric value. The map contains keys 6 and 8.

The expression that creates key 8 is `count[8]`. Neither `contains` nor the member `count` inserts an absent key. “The stored value is zero” and “the key is missing” are separate states.

### Drill 4 – Key order is not score order

The output is:

```
ali 100
nour 90
```

The second insertion of `nour` does not replace the existing score 80. The later assignment changes that score to 90. Iteration sorts keys, so `ali` appears before `nour`. Changing `nour`’s mapped score to 200 would not move it before `ali`.

To report entries by score, you need a representation or sorting step whose comparison uses scores. A map keyed by name cannot infer a score-order requirement from the mapped values.

**Drill 5 – Three kinds of value boundary**

| Threshold | Lower bound | Upper bound | Greatest $\leq$ threshold |
|-----------|-------------|-------------|---------------------------|
| 1         | 2           | 2           | NONE                      |
| 5         | 5           | 9           | 5                         |
| 6         | 9           | 9           | 5                         |
| 9         | 9           | NONE        | 9                         |
| 10        | NONE        | NONE        | 9                         |

`equal_range(5)` contains two occurrences. Its first iterator points to the first 5 and its second iterator points to 9; the second boundary is excluded. A predecessor search uses `upper_bound`, tests against `begin()`, and only then decrements.

**Drill 6 – Erase safely while traversing**

```
for (auto it = s.begin(); it != s.end(); ) {
    if (*it < 0) {
        it = s.erase(it);
    } else {
        ++it;
    }
}
```

After erasing the current element, the returned iterator names the following element, or `end()` when there is no following element. Assigning it avoids using the invalidated old iterator. Increment only when the current element survives; otherwise you could skip the next key or advance an invalid/end iterator. The loop also works for an empty set.<sup>[5-1]</sup>

### Drill 7 – Nested output does not imply a quadratic bound

The output statement executes exactly  $n$  times: the inner-loop lengths sum to  $n$ . The outer loop visits  $d$  groups, so the traversal work is  $O(d + n)$ , which is  $O(n)$  for a nonempty grouping with  $d \leq n$ . This analysis concerns traversal, not the earlier map construction.

With `auto entry`, each iteration copies a key/vector entry. Every group's vector would be copied once, adding  $O(n)$  total copied indices, allocations, and temporary storage up to the largest group. This does not necessarily change this particular traversal's asymptotic order, but it is unnecessary work. Repeatedly calling a helper that copies the entire map could have a much larger cumulative cost.

### Drill 8 – Hash counting and an ordered report

An unordered map's iteration order is not specified as sorted order. A valid report can use an ordered map from the start, or copy hash-map entries into a vector and sort that vector by word before printing.

With bounded key lengths, hash counting has average  $O(n)$  work. Copying  $d$  entries adds  $O(d)$ , sorting adds  $O(d \log(d + 1))$ , and output adds  $O(d)$ . The average combined bound is therefore  $O(n + d \log(d + 1))$ . Hash construction can have quadratic worst-case work; adding a sorting step does not remove that qualification. The copied vector also needs additional  $O(d)$  entry storage.

## Drill 9 – A rehash changes iterator validity

The reference `value` remains valid because key 7's element was not erased. Updating it changes the stored mapped value from 10 to 15. The old iterator `it` is invalidated by the rehash and must not be used afterward. A new lookup is safe:

```
std::cout << score.at(7) << '\n'; // Prints 15.
```

Alternatively, obtain a fresh iterator from `find(7)`. The distinction is between an iterator used to traverse the hash table and a reference to a surviving stored object; they do not have the same invalidation rule.<sup>[5-11]</sup>

## Drill 10 – The hidden cost of repeatedly counting duplicates

Each `count(4)` returns `n`. The loop performs it `n` times, so the final total is  $n^2$ . Counting a matching group in a multiset has a term proportional to that group's size, making the repeated work  $O(n^2)$  in this case.

A frequency map stores one key with count `n`. Its contribution to the same total is `1LL * n * n`: each of the `n` occurrences contributes the same frequency `n`. For several keys, sum the square of each positive frequency. This avoids repeatedly enumerating equal-key groups. Use `long long`;  $200000^2 = 40000000000$  exceeds a 32-bit signed integer.

## Drill 11 – Rank, selection, and distinctness

After the insertions, the keys are 3, 8, 12. The two rank counts are 1 and 2, and the selected key at zero-based rank 1 is 8. After erasing 3, the keys are 8, 12; the rank counts become 0 and 1, and rank 1 selects 12.

One-based request  $k=3$  is no longer valid because only two distinct keys remain. A pair-ID tree that kept both original 8 occurrences would start with four items, not three. Do not switch between distinct-key size and occurrence size while interpreting a query.

## Drill 12 – Choose a tool and name one pitfall

- a) `set<int>`: sorted unique membership and member `lower_bound`. Do not index it with brackets or treat a bound iterator as a value without checking it.
- b) `multiset<int>`: the cheapest item is at `begin()` when nonempty. Erase that iterator, not every equal-price key.
- c) `map<string,int>`: assignment updates the latest score and key traversal sorts names. A missing read-only query should not use subscript.
- d) `unordered_map<int,V>` for an appropriate mapped type `V`. Do not claim a constant or logarithmic worst-case lookup bound from its average-time behavior.
- e) A GNU order-statistics set with the shown red-black-tree policy, when the environment permits it. Ordinary `std::set` does not provide logarithmic selection by numerical rank.
- f) A GNU tree of `(value,unique_id)` pairs. Do not use `less_equal` or reuse a live ID to force duplicates.

The hash-map choice in (d) does not automatically meet a logarithmic worst-case requirement. An ordered map provides a different guarantee and may be the correct tradeoff when worst-case key-search growth matters.

## 5.40 Quick reference, glossary, and readiness checklist

### Choose what a stored item means

| Declaration                                       | Meaning                                                             |
|---------------------------------------------------|---------------------------------------------------------------------|
| <code>std::set&lt;int&gt; s;</code>               | Distinct integer keys, ordered increasingly.                        |
| <code>std::multiset&lt;int&gt; bag;</code>        | Ordered integer occurrences; duplicates remain.                     |
| <code>std::map&lt;int,int&gt; m;</code>           | One mapped integer per distinct ordered key.                        |
| <code>std::map&lt;std::string,int&gt; m;</code>   | One mapped integer per word/name key.                               |
| <code>std::unordered_map&lt;int,int&gt; m;</code> | One mapped integer per hashed key; no sorted traversal.             |
| <code>OrderedSet s;</code>                        | The GNU type alias shown in Section 5.24, with rank/select support. |

Use `<set>` for set and multiset, `<map>` for map, `<unordered_map>` for the hash map, `<string>` for string keys, and the two GNU PBDS headers for the extension. A name such as `OrderedSet` exists only after you declare its alias.

### Queries that do not create missing keys

```
// Inside main, after declaring suitable containers.
bool present = s.contains(x);
auto it = m.find(key);
if (it != m.end()) {
    std::cout << it->second << '\n';
}
std::cout << m.count(key) << '\n'; // 0 or 1 entries
```

`find` gives an iterator, `contains` gives a boolean, and `count` gives a number of matching entries. A multiset can have a count greater than one; an ordinary map cannot have multiple entries with the same key.

### Insertion and intended mutation

```
s.insert(x);           // Unique membership.
bag.insert(x);         // One more occurrence.
m[key] = value;        // Insert or replace mapped value.
++frequency[key];     // Intentional zero-then-increment.
```

`m[key]` may insert a missing key even when used in an expression that looks like a read. Use `at` only when a key is guaranteed to exist, or when you intentionally handle its missing-key exception.

## A boundary and erasure recipe card

| Needed result, ascending order  | Recipe                                                                          |
|---------------------------------|---------------------------------------------------------------------------------|
| First value $\geq x$            | Member <code>lower_bound(x)</code> ; check <code>end()</code> .                 |
| First value $> x$               | Member <code>upper_bound(x)</code> ; check <code>end()</code> .                 |
| Greatest value $\leq x$         | <code>upper_bound(x)</code> ; if not <code>begin()</code> , decrement.          |
| Greatest value $< x$            | <code>lower_bound(x)</code> ; if not <code>begin()</code> , decrement.          |
| All values equal to $x$         | <code>equal_range(x)</code> gives a half-open iterator range.                   |
| Minimum / maximum               | Nonempty: <code>*begin()</code> / <code>*rbegin()</code> for a set or multiset. |
| Erase all copies in a multiset  | <code>bag.erase(x)</code> .                                                     |
| Erase one copy if present       | <code>find(x)</code> , check, then <code>erase(it)</code> .                     |
| Continue after iterator erasure | For these standard containers, assign <code>it = c.erase(it)</code> .           |

An empty collection has `begin()==end()`. A past-the-end iterator is a boundary, not a stored element. Never dereference it. Never decrement `begin()`. A map iterator names a key/value entry, so use `it->first` and `it->second` rather than pretending the entry is only a number.

## Compact complexity summary

For fixed-cost keys and values, with  $n$  stored elements and  $k$  matches:

| Operation                                            | Cost                            |
|------------------------------------------------------|---------------------------------|
| Ordered key search / bounds / insertion              | $O(1 + \log(n + 1))$ .          |
| Multiset count or erase by key                       | $O(1 + \log(n + 1) + k)$ .      |
| Standard ordered-container erase at a known iterator | Amortized $O(1)$ .              |
| Walk $k$ set/map iterator steps                      | $O(k)$ .                        |
| Whole ordered-container copy / traversal             | $O(n)$ .                        |
| Hash-map exact-key lookup                            | Average $O(1)$ , worst $O(n)$ . |
| GNU configured tree rank / select / key update       | $O(1 + \log(n + 1))$ .          |

These entries do not include unrelated input/output, long-string comparisons, constructing complex mapped values, or repeated copies made by a caller. A bound lookup followed by printing  $k$  results costs at least the search work plus  $O(k)$  output.

## Map traversal and hash-table reminders

```
// Read entries without copying them.
for (const auto& entry : m) {
    std::cout << entry.first << ' ' << entry.second << '\n';
}

// Change mapped values, not keys.
for (auto& entry : m) {
    entry.second += 1;
}
```

For `std::map`, traversal follows the key comparator. For `unordered_map`, do not assume ascending, insertion, or stable printed order. Sort a copied list of entries when the report needs a particular order.

Insertion into a standard set or map preserves existing iterators and references. Erasure invalidates those referring to erased elements. A hash-table rehash invalidates iterators but preserves references and pointers to surviving elements. Refresh iterators after any operation that may have rehashed the table.<sup>[5-1][5-11]</sup>

## GNU rank/select reminders

```
// After declaring the GNU OrderedSet alias and object.
auto count = values.order_of_key(x); // Strictly smaller.
auto it = values.find_by_order(k);    // Zero-based rank.
if (it != values.end()) {
    std::cout << *it << '\n';
}
```

For signed input, validate before converting to the tree's unsigned rank type. For a one-based request, require  $1 \leq k \leq \text{size}$  and query  $k-1$ . Repeated integer keys do not create repeated ranks.

For duplicate occurrences, store `(value, unique_id)` pairs with a strict comparator. With IDs from 0 to  $n-1$ , `order_of_key({x, -1})` counts occurrences below  $x$ . Selection returns a pair; print its first field when the requested answer is the value.

### The most important contrasts

Key versus position. Distinct value versus occurrence. Missing entry versus mapped zero. Read-only lookup versus subscript insertion. Erase-one iterator versus erase-all key. Ordered bound versus numerical rank. Average hash cost versus a worst-case guarantee.

## Glossary

**Associative container**

A container in which keys are used to find stored entries.

**Key**

The field used to identify and locate an entry; not a numerical position in the container.

**Mapped value**

Data associated with a map key, such as a score or a frequency.

**Distinct value**

A value counted only once even when several input occurrences match it.

**Multiplicity**

The number of occurrences of a value.

**Comparator**

The rule that orders keys. The chapter normally uses strict ascending order.

**Equivalent keys**

For an ordered comparator, neither key is ordered before the other. For ordinary integers with `less`, this matches equality.

**Iterator**

An object that refers to a position in a container and can support specified traversal operations.

**Past-the-end iterator**

A boundary after the last element; it must not be dereferenced.

**Lower bound / upper bound**

In ascending order, the first key at least a threshold / strictly greater than it.

**Half-open range**

A range including its first boundary and excluding its second boundary.

**Hash function**

A function that helps place and locate keys in a hash table.

**Collision**

Different keys mapping to the same hash result or bucket; key equality still distinguishes entries.

**Bucket**

A group of hash-table entries associated with a bucket index.

**Load factor**

The number of stored elements divided by the number of buckets.

**Rehash**

Reorganization of a hash table into a bucket structure that meets the requested constraints.

**Order statistic**

A query involving sorted rank, such as the number of keys before a value or the key at a particular rank.

**PBDS**

GNU policy-based data structures, an implementation extension rather than an ISO C++23 container family.

## Readiness checklist

You are ready to move on when you can do the following without copying a full solution:

1. Explain when duplicates must disappear and when every occurrence must remain.
2. Build a set, test membership, and traverse it in sorted order.
3. Check an iterator against `end()` before dereferencing it.
4. Remove exactly one or all occurrences from a multiset intentionally.
5. Use lower and upper bounds, including a safe predecessor query.
6. Explain why a mapped zero is not the same as a missing key.
7. Count or accumulate with a map, and answer missing-key queries without insertion.
8. Distinguish a map's key order from an order based on mapped values.
9. State average and worst-case hash lookup costs separately.
10. Recognize a linear iterator walk, repeated count, or copied container hidden inside a loop.
11. Use the GNU rank/select API, distinguish its zero-based ranks from one-based requests, and check invalid ranks.
12. Explain why PBDS requires a compatible environment and why ranked duplicates need unique keys rather than `less_equal`.

## A short independent revision task

On blank paper, design a command processor that adds one number, removes one occurrence, asks whether a number exists, and reports the distinct count. Explain whether you will use a multiset plus extra bookkeeping or a frequency map with zero-count cleanup. Then change the request to ask for the *k*-th distinct key after every update. Explain which requirement changed your choice.

## What Session 6 will add

Maps can count values that are large, negative, or named by strings. The next session will introduce array-based frequency techniques when indices lie in a suitable bounded range, then prefix sums and partial-sum ideas for repeated range work. Those are different tools with different input assumptions, not replacements for every keyed container.

This handout contains Session 5. Its chapter and section numbering match the collected edition containing Sessions 1–5.

## 5.41 Technical notes and references

### Language, implementation, and scope

The standard-container examples target **C++23**. Standard wording references use the fixed **N4950** C++23 working draft, publicly rendered in HTML, rather than a moving draft that may contain later changes. These are language/library references, not beginner tutorials. The examples and exercises in this chapter are original teaching material.

**GNU PBDS is separate.** Nine complete programs use GNU extension headers. They compile in the tested GNU environment with `-std=c++23`, but that does not make them portable ISO C++23 library programs. The type alias uses `rb_tree_tag` and the order-statistics node-update policy; the stated tree costs refer to that configuration, not every possible PBDS policy.<sup>[5-20][5-21][5-22][5-25]</sup>

The GNU programs are examples 25–29 and practice solutions 19–22. Standard-container programs do not depend on these headers. You can study and compile the standard part independently.

### Integer ranges, text, and input contracts

The verification environment uses a 32-bit `int` and a 64-bit `long long`. C++ does not universally require `int` to be exactly 32 bits. The supplied numeric constraints assume an implementation wide enough for the stated 200000 counts and billion-scale keys. Large totals and target-complement arithmetic use `long long`.

Examples are contest solutions: input follows the stated format and allowed command types. Missing keys, empty collections, and invalid rank requests are handled wherever the task permits them. Input validation beyond that contract is not the topic of the chapter.

Words and names in the bounded-word exercises use 1–20 lowercase ASCII English letters; anagram input has its own larger bound. Lexicographic string order here is not locale-aware dictionary sorting for arbitrary human languages. A map's counts operate on the exact input keys it receives.

### How to read the complexity claims

Numeric-key analyses use fixed-cost comparisons, hashing operations, and scalar arithmetic. String-key analyses state their bounded length or explicitly account for character work. Standard unordered-container costs are average and worst-case bounds, not a claim that one run will take a particular number of milliseconds.

## Memory and boundary conventions

Ordered-set storage follows the number of distinct keys; multiset storage follows occurrences. A grouped `map<int, vector<int>>` stores both map entries and every appended index. Hash-map storage includes entries and buckets; it is written  $O(d + b)$  where those counts are independently relevant. An input vector or string explicitly kept by a solution is also counted.

We often write  $1 + \log(n + 1)$  to cover an empty collection's constant overhead. Shorter logarithmic statements refer to the usual nonempty-input asymptotic convention. Amortized iterator erasure costs in standard ordered containers do not include the search needed to obtain the iterator.

## What was checked for this edition

There are **34 complete teaching examples** and **22 complete practice solutions**. All 56 programs compiled with **GCC 14.2.0** in C++23 mode. The verification suite passed **7230 input/output checks**: 63 fixed sample checks, 7153 generated or exhaustive-small-case checks, and 14 large boundary checks.

The smaller checks use strict diagnostics, undefined-behavior checks, library assertions, and GNU debug iterators:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -D_GLIBCXX_DEBUG  
-fsanitize=undefined -fno-sanitize-recover=all
```

Large 200000-element cases use the same flags except `-D_GLIBCXX_DEBUG`, because debug-container validation can substantially change the cost of tree operations. Undefined-behavior instrumentation and standard-library assertions remain enabled. These flags describe verification builds, not a requirement for every student submission.<sup>[5-26][5-27][5-28]</sup>

## What the tests do and do not establish

Tests cover absent queries, duplicate groups, erase-one versus erase-all, negative and extreme keys, empty states, repeated commands, zero-valued records, wide totals, both rank conventions, duplicate-ranked pairs, repeated cases, and maximum stated lengths. Generated expected answers use independent Python representations such as sets, counters, dictionaries, and sorted lists.

Passing finite tests is not a proof over all allowed inputs. The written invariants explain why the algorithms work; tests help catch implementation, sample, and boundary mistakes. Deliberately wrong fragments are labelled and excluded from the working-program manifest.

## Rebuild or reproduce the checks

From the source-package root, run `python3 tests/verify_session_05.py`. The JSON verification report records compiler flags, source hashes, per-program check counts, and the result. Every complete printed listing is loaded from the exact distributed `.cpp` file, so students are not given a separate untested transcription.

Compile `session_05.tex` for this handout and `book.tex` for the collected Sessions 1–5 edition. The Makefile and README include commands for all five standalone handouts. The common preamble and shared chapter sources keep typography and edits synchronized between editions.

## Reading the source markers

A superscript such as <sup>[5-7]</sup> points to the corresponding source below. The leading 5 identifies this session so references do not collide when chapters are collected. The GNU API references are pinned to the GCC 14.2.0 documentation used for checking this edition.

## Sources: standard containers and access

### [5-1] C++23 N4950: associative-container requirements

<https://timsong-cpp.github.io/cppwp/n4950/associative.reqmts>

### [5-2] C++23 N4950: set overview

<https://timsong-cpp.github.io/cppwp/n4950/set.overview>

### [5-3] C++23 N4950: set declarations and operations

<https://timsong-cpp.github.io/cppwp/n4950/set>

### [5-4] C++23 N4950: multiset overview

<https://timsong-cpp.github.io/cppwp/n4950/multiset.overview>

### [5-5] C++23 N4950: multiset declarations and operations

<https://timsong-cpp.github.io/cppwp/n4950/multiset>

### [5-6] C++23 N4950: map overview

<https://timsong-cpp.github.io/cppwp/n4950/map.overview>

### [5-7] C++23 N4950: map element access

<https://timsong-cpp.github.io/cppwp/n4950/map.access>

### [5-8] C++23 N4950: map modifiers

<https://timsong-cpp.github.io/cppwp/n4950/map.modifiers>

### [5-9] C++23 N4950: map declarations and operations

<https://timsong-cpp.github.io/cppwp/n4950/map>

### [5-10] C++23 N4950: unordered\_map overview

<https://timsong-cpp.github.io/cppwp/n4950/unord.map.overview>

### [5-11] C++23 N4950: unordered-container requirements

<https://timsong-cpp.github.io/cppwp/n4950/unord.req>

### [5-12] C++23 N4950: unordered\_map element access

<https://timsong-cpp.github.io/cppwp/n4950/unord.map.elem>

### [5-13] C++23 N4950: unordered\_map modifiers

<https://timsong-cpp.github.io/cppwp/n4950/unord.map.modifiers>

### [5-14] C++23 N4950: general container requirements

<https://timsong-cpp.github.io/cppwp/n4950/container.reqmts>

## Sources: ordering, GNU extension, and verification

### [5-15] C++23 N4950: iterator operations

<https://timsong-cpp.github.io/cppwp/n4950/iterator.operations>

### [5-16] C++23 N4950: binary-search algorithms

<https://timsong-cpp.github.io/cppwp/n4950/alg.binary.search>

### [5-17] C++23 N4950: sorting and strict weak ordering

<https://timsong-cpp.github.io/cppwp/n4950/alg.sorting>

### [5-18] C++23 N4950: pair comparisons

<https://timsong-cpp.github.io/cppwp/n4950/pairs.spec>

### [5-19] C++23 N4950: string comparison

<https://timsong-cpp.github.io/cppwp/n4950/string.compare>

## GNU extension and verification

### [5-20] GNU libstdc++: policy-based data structures overview

[https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy\\_data\\_structures.html](https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures.html)

### [5-21] GNU libstdc++: using policy-based data structures

[https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy\\_data\\_structures\\_using.html](https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures_using.html)

### [5-22] GNU libstdc++: PBDS design and tree policies

[https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy\\_data\\_structures\\_design.html](https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures_design.html)

### [5-23] GCC 14.2.0: tree order-statistics node-update API

<https://gcc.gnu.org/onlinedocs/gcc-14.2.0/libstdc++/api/a11918.html>

### [5-24] GCC 14.2.0: order-statistics implementation source

[https://gcc.gnu.org/onlinedocs/gcc-14.2.0/libstdc++/api/a18515\\_source.html](https://gcc.gnu.org/onlinedocs/gcc-14.2.0/libstdc++/api/a18515_source.html)

### [5-25] GCC: selecting a language standard

<https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

### [5-26] GCC: warnings and strict diagnostics

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

### [5-27] GCC: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

### [5-28] GNU libstdc++: using debug mode

[https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug\\_mode\\_using.html](https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html)