inzva ACWC'23 | Data Structures Notes

Prepared with ♥, just for you...

inzva Team

January 24, 2023

Arrays... aren't they enough?

We see the advantages of stronger data structures whenever we actually need them — almost always!
Let's examine some weaknesses of our primitive friends — arrays!

1. The size is fixed once defined:
   - `int a[15]` → can't contain more than 15 elements!
   - `vector`: "I'm here to help!"
2. Element insertion/removal hurts — $\mathcal{O}(n)$
   - `set`: "Use me!"
3. Can't have a large size due to memory limits — indices can't be large!
   - `map`: "OMG hi!"

These notes are majorly based on C++ STL data structures. So, embrace C++ :)

vector

(API reference)

1. Define: `vector<type> v(size, initial_value)`
   - `type = int, long long` etc. → `initial_value` defaults to 0
   - `vector<int> v(5)`: 5 integers initialized to 0
   - `vector<long long> xd(1e6, 314)`: A million 64-bit integers initialized to 314
   - `vector<vector<int>> wow(100)`: 2D vector — a vector containing 100 empty vectors
   - `vector<vector<int>> wow(100, vector<int>(6, 2))`: A vector containing 100 vectors, each containing 6 integers initialized to 2
2. Append a new element (add to the end): `v.push_back(x)`
3. Remove the last element: `v.pop_back()`
4. Initialize with a vector: `vector<int> v2 = v` or `vector<int> v2(v)`
5. Copy: `vector<int> v2(5, 2); v2 = v;`
6. Swap: `v.swap(v2)` or `swap(v, v2)`
   - Works in $O(1)$!
7. Get the last element: `v.back()`

queue

(API reference)

1. Define: `queue<type> q`
2. Add a newcomer: `q.push(5)`
3. Get the "oldest" element: `q.front()`
4. Remove the "oldest" element: `q.pop()`
5. How many elements?: `q.size()`
6. Is it empty? `q.empty()`
   - `q.size() == 0` also does the job!

stack

(API reference)

queue's evil brother — whoever comes last is taken care of first!

1. Define: `stack<type> s`
2. Add a newcomer: `s.push(5)`
3. Get the "newest" element: `s.top()`
4. Remove the "newest" element: `s.pop()`
5. How many elements? `s.size()`
6. Is it empty? `s.empty()`
   - `s.size() == 0` also does the job!

Fun fact!

Under the hood, recursion takes the advantage of **stack**. In fact, everything "temporary" is stored inside **stack** (e.g., variables declared inside functions)!

deque

(API reference)

It combines **queue** and **stack** together: **d**ouble-**e**nded **que**ue!

1. Define: `deque<type> q`
2. Add to the back/front: `q.push_back/front(5)`
3. Get the back/front element: `q.back/front()`
4. Remove an element from back/front: `q.pop_back/front()`
5. How many elements? `q.size()`
6. Is it empty? `q.empty()`
   - `q.size() == 0` also does the job!

set

(API reference)

A set of ordered, distinct elements! What's cool about it is:

- Insert
- Remove
- Find
- Find "closest"

All can be performed in $O(\log n)$!

1. Define: `set<type> s`
2. Insert: `s.insert(5)`
3. Find: `s.find(5)`
   - Returns `s.end()` if doesn't exist!
4. Remove: `s.erase(5)` (Does nothing if doesn't exist)
5. Exists? `s.count(5) == 1`
   - `s.find(5) != s.end()` also works!
6. Min. element: `*s.begin()`
7. Max. element: `*s.rbegin()` or `*prev(s.end())`

set

It's also a binary-searchable data structure with its own **`lower_bound`** and **`upper_bound`** methods!

1. Find the smallest element not smaller than x: **`s.lower_bound(x)`**
2. Find the smallest element larger than x: **`s.upper_bound(x)`**

All methods that find an element return an iterator **`it`**. Iterators are very similar to pointers:

1. Access the element: **`*it`**
2. Go to the previous element's iterator: **`prev(it)`** or **`it--`** or **`--it`**
3. Go to the next element's iterator: **`next(it)`** or **`it++`** or **`++it`**

Sample problem

multiset

(API reference)
Same as **set** but allows multiple elements with the same value! However, there are some differences...

Be careful when erasing!

s.erase(5) erases **all** occurrences of 5 in **multiset**. Therefore, you'd better use **s.erase(s.find(5))** if you are to erase just one. **s.find(5)** returns an iterator of a single element, and thus, you would erase just one.

Don't use **count** to check the existence!

s.count(5) returns the number of elements with value 5. Its complexity grows linearly with the element count. In the worst case, it works in $O(n)$. Therefore, just use **s.find(5)** $\neq$ **s.end()**.

unordered_set

(API reference)
It's literally the **unordered** version of `set`.
However, it doesn't support `lower/upper_bound` functions.
It uses hashing logic to achieve an amortized complexity of $\mathcal{O}(1)$ per operation. So, it's also called "hash set". It's not used that much, though.

map

(API reference)
It's an indexed version of `set`: use the key to access the value!
Every method for `set` works for `map`'s keys!

1. Define: `map<key_type, value_type> mp`
2. Assign a new value to a key: `mp[3] = 5`
   - If the key doesn't exist, it will be created and its value is initialized to 5!

Tip

To check whether a key exists, use `mp.count(3)` or `mp.find(3) ≠ mp.end()` instead of using the value `mp[3]`. Be aware that the `[]` operator creates the key, which could cause performance issues because increasing the size also increases the methods' run times.

Sample problem

unordered_map

(API reference)
Similar to **unordered_set**, it's the unordered version of **map**! It's also called "hash map".
Unlike **unordered_set**, **unordered_map** is often used because it supports (key, value) pairs and the methods have an amortized complexity of $O(1)$. So, it's pretty much like an array that supports large indexing.

Iterating over `vector/set/map`

Let **`container`** be one of **`vector/set/map`**. C++ grants us this beautiful syntax for iterating over it:

- **`for(auto& x: container)`** (use elements themselves, being able to mutate)
- **`for(auto x: container)`** (use the copies of elements)

Particularly for **`map`**, we may also use this (requires C++17):

- **`for(auto& [key, value]: map_container)`**

This is called **structured binding**.

priority_queue

(API reference)

A **queue** that **prioritizes** its elements. In C++, it acts as a max heap — the largest element is on the top. All methods work in $O(\log n)$. You can also mimic it using `set` but since `priority_queue` is specialized for the max element, it's a bit faster.

1. Define: `priority_queue<type> q`
2. Add a new element: `q.push(5)`
3. Get the topmost (largest) element: `q.top()`
4. Remove the topmost element: `q.pop()`

How to make it work for min?

For primitive integer data types, you can simply negate the element before adding it to the queue. Figure out why this works :)

Another option is to use `priority_queue<int, vector<int>, greater<int>>` (for `int`).

Using custom structs as element types for ordered data structures

Order-based data structures, and also ordering functions like **`sort`** uses **`<`** operator of the elements' type. For primitive types, nothing special is required. However, for custom structs, we may need to overload the operator:

```
struct Edge {
    int u;
    int v;
    int weight;
    bool operator<(const Edge &o) const {
        return weight < o.weight;
    }
};
```

Helper data structures

1. `pair<first_type, second_type>`: Literally stores two elements of desired types. Use `.first` to get the first element and `.second` to get the second element.

Maps!

In fact, `map` uses `pair` to store (key, value) pairs!

2. `array<type, size>`: STL array for using the plain array inside other structures. For instance, you may use `set<array<int, 3>>`. I also prefer `array` over `pair` for fast coding with its random access operator :D

Bonus: sorting based on custom comparison

We use the **sort** function for sorting arrays/vectors:

- `sort(a, a + n)`
- `sort(v.begin(), v.end())`

This function in fact has a third parameter: **comparator function**. By default, it uses the default comparator of the element type: the < operator. However, we can write our own custom comparator, e.g. for sorting in descending order.

```cpp
bool comparator(int a, int b) {
    return a > b;
}

int main() {
    vector<int> v = {5, 4, 2, 1, 3};
    // This...
    sort(v.begin(), v.end(), comparator);
    // or this (w/ lambda function)
    sort(v.begin(), v.end(), [](int a, int b) {
        return a > b;
    });
}
```

Tip

You can also use `sort(v.rbegin(), v.rend())` to sort in descending order!

Be careful!

The comparator function must not return true for both element orderings (a, b) and (b, a). Otherwise, it would cause lots of pain. It's because the sorting mechanism uses the function to decide whether to swap two elements — it could cause infinitely many swaps!