Chapter 4

Built-in Data Structures

A data structure determines how data is stored. (is it sorted? indexed? what operations does it support?) Each data structure supports some operations efficiently, while other operations are either inefficient or not supported at all. This chapter introduces the data structures in the C++ standard library that are frequently used in competitive programming.

The C++ standard library data structures are designed to store any type of data. We put the desired data type within the <> brackets when declaring the data structure, as follows:

```
vector<string> v;
```

This creates a `vector` structure that only stores objects of type `string`.

For our examples below, we will primarily use the `int` data type, but note that you can use any data type including `string` and user-defined structures.

Essentially every standard library data structure supports the `size()` method, which returns the number of elements in the data structure, and the `empty()` method, which returns `true` if the data structure is empty, and `false` otherwise.

4.1 Iterators

Before the data structures are introduced, you should understand an iterator. An iterator allows you to traverse a container by providing a pointer. For example, `vector.begin()` returns an iterator pointing to the first element of the vector. Apart from the standard way of traversing a vector (by treating it as an array), you can also use iterators:

```
for (vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it) {
    cout << *it; //prints the values in the vector using the pointer
}
```

However, a more generic way to do this is with a for-each loop and `auto` (C++11 and later versions) that automatically infers the type of an object:

```
for(auto element : v) {
    cout << element; //prints the values in the vector
}
```

4.2 Dynamic Arrays

You're probably already familiar with regular (static) arrays. Now, there are also dynamic arrays (`vector` in C++) that support all the functions that a normal array does, and can resize itself to accommodate more elements. In a dynamic array, we can also add and delete elements at the end in $O(1)$ time.

However, we need to be careful that we only add elements to the end of the `vector`; insertion and deletion in the middle of the `vector` is $O(n)$.

```
vector<int> v;
v.push_back(2); // [2]
v.push_back(3); // [2, 3]
v.push_back(7); // [2, 3, 7]
v.push_back(5); // [2, 3, 7, 5]
v[1] = 4; // sets element at index 1 to 4 -> [2, 4, 7, 5]
v.erase(v.begin() + 1); // removes element at index 1 -> [2, 7, 5]
// this remove method is O(n); to be avoided
v.push_back(8); // [2, 7, 5, 8]
v.erase(v.end()-1); // [2, 7, 5]
// here, we remove the element from the end of the list; this is O(1).
v.push_back(4); // [2, 7, 5, 4]
v.push_back(4); // [2, 7, 5, 4, 4]
v.push_back(9); // [2, 7, 5, 4, 4, 9]
cout << v[2]; // 5
v.erase(v.begin(), v.begin()+3); // [4, 4, 9]
// this erases the first three elements; O(n)
```

To iterate through a static or dynamic array, we can use either the regular for loop or the for-each loop.

In order to sort a dynamic array, use `sort(v.begin(), v.end())`, whereas static arrays require `sort(arr, arr + N)` where N is the number of elements to be sorted. The default sort function sorts the array in ascending order.

In array-based contest problems, we'll use one-, two-, and three-dimensional static arrays most of the time. However, we can also have static arrays of dynamic arrays, dynamic arrays of static arrays, and so on. Usually, the choice between a static array and a dynamic array is just personal preference.

4.3 Stacks and the Various Types of Queues

Stacks

A stack is a Last In First Out (LIFO) data structure that supports three operations: `push`, which adds an element to the top of the stack, `pop`, which removes an element from the top of the stack, and `top`, which retrieves the element at the top without removing it, all in $O(1)$ time. Think of it like a real-world stack of papers.

```
stack<int> s;
s.push(1); // [1]
s.push(13); // [1, 13]
cout << s.size() << endl; // 2
s.pop(); // [1]
cout << s.top() << endl; // 1
s.pop(); // []
cout << s.size() << endl; // 0
```

Queues

A queue is a First In First Out (FIFO) data structure that supports three operations of `push`, insertion at the back of the queue, `pop`, deletion from the front of the queue, and `front`, which retrieves the element at the front without removing it, all in $O(1)$ time.

```
queue<int> q;
q.push(1); // [1]
q.push(3); // [3, 1]
q.pop(); // [3]
q.push(4); // [4, 3]
cout << q.size() << endl; // 2
cout << q.front() << endl; // 4
```

Deques

A deque (usually pronounced "deck") stands for double ended queue and is a combination of a stack and a queue, in that it supports $O(1)$ insertions and deletions from both the front and the back of the deque. The four methods for adding and removing are `push_back`, `pop_back`, `push_front`, and `pop_front`. The methods for retrieving the first and last elements without removing are `front` and `back`.

```
deque<int> d;
d.push_front(1); // [1]
d.push_back(2); // [1, 2]
d.push_front(3); // [3, 1, 2]
```

```
d.push_back(4); // [3, 1, 2, 4]
d.pop_back(); // [3, 1, 2]
d.pop_front(); // [1, 2]
```

Priority Queues

A priority queue supports the following operations: insertion of elements, deletion of the element considered highest priority, and retrieval of the highest priority element, all in $O(\log n)$ time according to the number of elements in the priority queue. Priority is based on a comparator function, and in C++ the highest element is put at the front of the queue. The priority queue is one of the most important data structures in competitive programming, so make sure you understand how and when to use it.

```
priority_queue<int> pq;
pq.push(4); // [4]
pq.push(2); // [2, 4]
pq.push(1); // [1, 2, 4]
pq.push(3); // [1, 2, 3, 4]
cout << pq.top() << endl; // 4
pq.pop(); // [1, 2, 3]
pq.pop(); // [1, 2]
pq.push(5); // [1, 2, 5]
```

4.4 Sets and Maps

A set is a collection of objects that contains no duplicates. There are two types of sets: unordered sets (`unordered_set` in C++), and ordered set (`set` in C++).

Unordered Sets

The unordered set works by hashing, which is assigning a usually-unique code to every variable/object which allows insertions, deletions, and searches in $O(1)$ time, albeit with a high constant factor, as hashing requires a large constant number of operations. However, as the name implies, elements are not ordered in any meaningful way, so traversals of an unordered set will return elements in some arbitrary order. The operations on an unordered set are `insert`, which adds an element to the set if not already present, `erase`, which deletes an element if it exists, and `count`, which returns 1 if the set contains the element and 0 if it doesn't.

```
unordered_set<int> s;
s.insert(1); // [1]
s.insert(4); // [1, 4] in arbitrary order
```

```
s.insert(2); // [1, 4, 2] in arbitrary order
s.insert(1); // [1, 4, 2] in arbitrary order
// the add method did nothing because 1 was already in the set
cout << s.count(1) << endl; // 1
set.erase(1); // [2, 4] in arbitrary order
cout << s.count(5) << endl; // 0
s.erase(0); // [2, 4] in arbitrary order
// if the element to be removed does not exist, nothing happens

for(int element : s){
    cout << element << " ";
}
cout << endl;
// You can iterate through an unordered set, but it will do so in arbitrary
↪  order
```

Ordered Sets

The second type of set data structure is the ordered or sorted set. Insertions, deletions, and searches on the ordered set require $O(\log n)$ time, based on the number of elements in the set. As well as those supported by the unordered set, the ordered set also allows four additional operations: `begin()`, which returns an iterator to the lowest element in the set, `end()`, which returns an iterator to the highest element in the set, `lower_bound`, which returns an iterator to the least element greater than or equal to some element k, and `upper_bound`, which returns an iterator to the least element strictly greater than some element k.

```
set<int> s;
s.insert(1); // [1]
s.insert(14); // [1, 14]
s.insert(9); // [1, 9, 14]
s.insert(2); // [1, 2, 9, 14]
cout << *s.upper_bound(7) << '\n'; // 9
cout << *s.upper_bound(9) << '\n'; // 14
cout << *s.lower_bound(5) << '\n'; // 9
cout << *s.lower_bound(9) << '\n'; // 9
cout << *s.begin() << '\n'; // 1
auto it = s.end();
cout << *(--it) << '\n'; // 14
s.erase(s.upper_bound(6)); // [1, 2, 14]
```

The primary limitation of the ordered set is that we can't efficiently access the k^{th} largest element in the set, or find the number of elements in the set greater than some arbitrary x. These operations can be handled using a data structure called an order statistic tree, but that is beyond the scope of this book.

Maps

A map is a set of ordered pairs, each containing a key and a value. In a map, all keys are required to be unique, but values can be repeated. Maps have three primary methods: one to add a specified key-value pairing, one to retrieve the value for a given key, and one to remove a key-value pairing from the map. Like sets, maps can be unordered (`unordered_map` in C++) or ordered (`map` in C++). In an unordered map, hashing is used to support $O(1)$ operations. In an ordered map, the entries are sorted in order of key. Operations are $O(\log n)$, but accessing or removing the next key higher or lower than some input `k` is also supported.

Unordered Maps

In an unordered map `m`, the `m[key] = value` operator assigns a value to a key and places the key and value pair into the map. The operator `m[key]` returns the value associated with the key. The `count(key)` method returns the number of times the key is in the map (which is either one or zero), and therefore checks whether a key exists in the map. Lastly, `erase(key)` and `erase(it)` removes the map entry associated with the specified key or iterator. All of these operations are $O(1)$, but again, due to the hashing, this has a high constant factor.

```
unordered_map<int, int> m;
m[1] = 5; // [(1, 5)]
m[3] = 14; // [(1, 5); (3, 14)]
m[2] = 7; // [(1, 5); (3, 14); (2, 7)]
m.erase(2); // [(1, 5); (3, 14)]
cout << m[1] << '\n'; // 5
cout << m.count(7) << '\n' ; // 0
cout << m.count(1) << '\n' ; // 1
```

Ordered Maps

The ordered map supports all of the operations that an unordered map supports, and additionally supports `lower_bound` and `upper_bound`, returning the iterator pointing to the lowest entry not less than the specified key, and the iterator pointing to the lowest entry strictly greater than the specified key respectively.

```
map<int, int> m;
m[3] = 5; // [(3, 5)]
m[11] = 4; // [(3, 5); (11, 4)]
m[10] = 491; // [(3, 5); (10, 491); (11, 4)]
cout << m.lower_bound(10)->first << " " << m.lower_bound(10)->second << '\n'; //
↪  10 491
cout << m.upper_bound(10)->first << " " << m.upper_bound(10)->second << '\n'; //
↪  11 4
m.erase(11); // [(3, 5); (10, 491)]
if (m.upper_bound(10) == m.end())
```

```
{
    cout << "end" << endl; // Prints end
}
```

A note on unordered sets and maps: In USACO contests, they're generally fine, but in CodeForces contests, you should always use sorted sets and maps. This is because the built-in hashing algorithm is vulnerable to pathological data sets causing abnormally slow runtimes, in turn causing failures on some test cases.

Multisets

Lastly, there is the multiset, which is essentially a sorted set that allows multiple copies of the same element. In addition to all of the regular set operations, the multiset `count()` method returns the number of times an element is present in the multiset. The time complexity of this operation is $O(\log n + f)$ where f is the number of occurrences of the specified element in the multiset. This is because the $\log n$ factor searches for the element, and the f factor iterates linearly through the sorted set to find the number of occurrences.

```
multiset<int> ms;
ms.insert(1); // [1]
ms.insert(14); // [1, 14]
ms.insert(9); // [1, 9, 14]
ms.insert(2); // [1, 2, 9, 14]
ms.insert(9); // [1, 2, 9, 9, 14]
ms.insert(9); // [1, 2, 9, 9, 9, 14]
cout << ms.count(4) << '\n'; // 0
cout << ms.count(9) << '\n'; // 3
cout << ms.count(14) << '\n'; // 1
```

The `begin()`, `end()`, `lower_bound()`, and `upper_bound()` operations work the same way they do in the normal sorted set.

4.5 Problems

Again, note that CSES's grader is very slow, so don't worry if you encounter a Time Limit Exceeded verdict; as long as you pass the majority of test cases within the time limit, you can consider the problem solved, and move on.

1. CSES Problem Set Task 1621: Distinct Numbers
   `https://cses.fi/problemset/task/1621`
2. CSES Problem Set Task 1084: Apartments
   `https://cses.fi/problemset/task/1084`
3. CSES Problem Set Task 1091: Concert Tickets
   `https://cses.fi/problemset/task/1091`

4. CSES Problem Set Task 1163: Traffic Lights
   `https://cses.fi/problemset/task/1163`
5. CSES Problem Set Task 1164: Room Allocation
   `https://cses.fi/problemset/task/1164`