Data Structures 5

This chapter introduces the most important data structures of the C++ standard library. In competitive programming, it is crucial to know which data structures are available in the standard library and how to use them. This often saves a large amount of time when implementing an algorithm.

Section 5.1 first describes the vector structure which is an efficient dynamic array. After this, we will focus on using iterators and ranges with data structures, and briefly discuss deques, stacks, and queues.

Section 5.2 discusses sets, maps and priority queues. Those data structures are often used as building blocks of efficient algorithms, because they allow us to maintain dynamic structures that support both efficient searches and updates.

Section 5.3 shows some results about the efficiency of data structures in practice. As we will see, there are important performance differences that cannot be detected by only looking at time complexities.

5.1 Dynamic Arrays

In C++, ordinary arrays are fixed-size structures, and it is not possible to change the size of an array after creating it. For example, the following code creates an array which contains n integer values:

```
int array[n];
```

A *dynamic array* is an array whose size can be changed during the execution of the program. The C++ standard library provides several dynamic arrays, most useful of them being the vector structure.

5.1.1 Vectors

A *vector* is a dynamic array that allows us to efficiently add and remove elements at the end of the structure. For example, the following code creates an empty vector and adds three elements to it:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

Then, the elements can be accessed like in an ordinary array:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Another way to create a vector is to give a list of its elements:

```
vector<int> v = {2,4,2,5,1};
```

We can also give the number of elements and their initial values:

```
vector<int> a(8); // size 8, initial value 0
vector<int> b(8,2); // size 8, initial value 2
```

The function `size` returns the number of elements in the vector. For example, the following code iterates through the vector and prints its elements:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

A shorter way to iterate through a vector is as follows:

```
for (auto x : v) {
    cout << x << "\n";
}
```

The function `back` returns the last element of a vector, and the function `pop_back` removes the last element:

```
vector<int> v = {2,4,2,5,1};
cout << v.back() << "\n"; // 1
v.pop_back();
cout << v.back() << "\n"; // 5
```

Vectors are implemented so that the push_back and pop_back operations work in $O(1)$ time on average. In practice, using a vector is almost as fast as using an ordinary array.

5.1.2 Iterators and Ranges

An *iterator* is a variable that points to an element of a data structure. The iterator begin points to the first element of a data structure, and the iterator end points to the position *after* the last element. For example, the situation can look as follows in a vector v that consists of eight elements:

```
[ 5, 2, 3, 1, 2, 5, 7, 1 ]
  ↑                      ↑
  v.begin()              v.end()
```

Note the asymmetry in the iterators: begin() points to an element in the data structure, while end() points *outside* the data structure.

A *range* is a sequence of consecutive elements in a data structure. The usual way to specify a range is to give iterators to its first element and the position after its last element. In particular, the iterators begin() and end() define a range that contains all elements in a data structure.

The C++ standard library functions typically operate with ranges. For example, the following code first sorts a vector, then reverses the order of its elements, and finally shuffles its elements.

```
sort(v.begin(),v.end());
reverse(v.begin(),v.end());
random_shuffle(v.begin(),v.end());
```

The element to which an iterator points can be accessed using the * syntax. For example, the following code prints the first element of a vector:

```
cout << *v.begin() << "\n";
```

To give a more useful example, lower_bound gives an iterator to the first element in a sorted range whose value is *at least* x, and upper_bound gives an iterator to the first element whose value is *larger than* x:

```
vector<int> v = {2,3,3,5,7,8,8,8};
auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);
cout << *a << " " << *b << "\n"; // 5 7
```

Note that the above functions only work correctly when the given range is sorted. The functions use binary search and find the requested element in logarithmic time.

If there is no such element, the functions return an iterator to the element after the last element in the range.

The C++ standard library contains a large number of useful functions that are worth exploring. For example, the following code creates a vector that contains the unique elements of the original vector in a sorted order:

```
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());
```

5.1.3 Other Structures

A *deque* is a dynamic array that can be efficiently manipulated at both ends of the structure. Like a vector, a deque provides the functions `push_back` and `pop_back`, but it also provides the functions `push_front` and `pop_front` which are not available in a vector. A deque can be used as follows:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

The operations of a deque also work in $O(1)$ average time. However, deques have larger constant factors than vectors, so deques should be used only if there is a need to manipulate both ends of the array.

C++ also provides two specialized data structures that are, by default, based on a deque. A *stack* has the functions `push` and `pop` for inserting and removing elements at the end of the structure and the function `top` that retrieves the last element:

```
stack<int> s;
s.push(2); // [2]
s.push(5); // [2,5]
cout << s.top() << "\n"; // 5
s.pop(); // [2]
cout << s.top() << "\n"; // 2
```

Then, in a *queue*, elements are inserted at the end of the structure and removed from the front of the structure. Both the functions `front` and `back` are provided for accessing the first and last element.

```
queue<int> q;
q.push(2); // [2]
q.push(5); // [2,5]
cout << q.front() << "\n"; // 2
q.pop(); // [5]
cout << q.back() << "\n"; // 5
```

5.2 Set Structures

A *set* is a data structure that maintains a collection of elements. The basic operations of sets are element insertion, search, and removal. Sets are implemented so that all the above operations are efficient, which often allows us to improve on running times of algorithms using sets.

5.2.1 Sets and Multisets

The C++ standard library contains two set structures:

- `set` is based on a balanced binary search tree and its operations work in $O(\log n)$ time.
- `unordered_set` is based on a hash table and its operations work, on average,[1] in $O(1)$ time.

Both structures are efficient, and often either of them can be used. Since they are used in the same way, we focus on the `set` structure in the following examples.

The following code creates a set that contains integers and shows some of its operations. The function `insert` adds an element to the set, the function `count` returns the number of occurrences of an element in the set, and the function `erase` removes an element from the set.

[1]The worst-case time complexity of the operations is $O(n)$, but this is very unlikely to occur.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

An important property of sets is that all their elements are *distinct*. Thus, the function count always returns either 0 (the element is not in the set) or 1 (the element is in the set), and the function insert never adds an element to the set if it is already there. The following code illustrates this:

```
set<int> s;
s.insert(3);
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

A set can be used mostly like a vector, but it is not possible to access the elements using the [] notation. The following code prints the number of elements in a set and then iterates through the elements:

```
cout << s.size() << "\n";
for (auto x : s) {
    cout << x << "\n";
}
```

The function find(x) returns an iterator that points to an element whose value is x. However, if the set does not contain x, the iterator will be end().

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

Ordered Sets The main difference between the two C++ set structures is that set is *ordered*, while unordered_set is not. Thus, if we want to maintain the order of the elements, we have to use the set structure.

For example, consider the problem of finding the smallest and largest value in a set. To do this efficiently, we need to use the set structure. Since the elements are sorted, we can find the smallest and largest value as follows:

```
auto first = s.begin();
auto last = s.end(); last--;
cout << *first << " " << *last << "\n";
```

Note that since `end()` points to an element after the last element, we have to decrease the iterator by one.

The `set` structure also provides the functions `lower_bound`(x) and `upper_bound`(x) that return an iterator to the smallest element in a `set` whose value is *at least* or *larger than* x, respectively. In both the functions, if the requested element does not exist, the return value is `end()`.

```
cout << *s.lower_bound(x) << "\n";
cout << *s.upper_bound(x) << "\n";
```

Multisets A *multiset* is a set that can have several copies of the same value. C++ has the structures `multiset` and `unordered_multiset` that resemble `set` and `unordered_set`. For example, the following code adds three copies of the value 5 to a multiset.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

The function `erase` removes all copies of a value from a multiset:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Often, only one value should be removed, which can be done as follows:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

Note that the functions `count` and `erase` have an additional $O(k)$ factor where k is the number of elements counted/removed. In particular, it is *not* efficient to count the number of copies of a value in a multiset using the `count` function.

5.2.2 Maps

A *map* is a set that consists of key-value pairs. A map can also be seen as a generalized array. While the keys in an ordinary array are always consecutive integers $0, 1, \ldots, n-1$, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values.

The C++ standard library contains two map structures that correspond to the set structures: `map` is based on a balanced binary search tree and accessing elements takes $O(\log n)$ time, while `unordered_map` uses hashing and accessing elements takes $O(1)$ time on average.

The following code creates a map whose keys are strings and values are integers:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

If the value of a key is requested but the map does not contain it, the key is automatically added to the map with a default value. For example, in the following code, the key “aybabtu” with value 0 is added to the map.

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

The function `count` checks if a key exists in a map:

```
if (m.count("aybabtu")) {
    // key exists
}
```

Then, the following code prints all keys and values in a map:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

5.2.3 Priority Queues

A *priority queue* is a multiset that supports element insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element. Insertion and removal take $O(\log n)$ time, and retrieval takes $O(1)$ time.

A priority queue is usually based on a heap structure, which is a special binary tree. While a `multiset` provides all the operations of a priority queue and more, the benefit of using a priority queue is that it has smaller constant factors. Thus, if we only need to efficiently find minimum or maximum elements, it is a good idea to use a priority queue instead of a set or multiset.

By default, the elements in a C++ priority queue are sorted in decreasing order, and it is possible to find and remove the largest element in the queue. The following code illustrates this:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

If we want to create a priority queue that supports finding and removing the smallest element, we can do it as follows:

```
priority_queue<int,vector<int>,greater<int>> q;
```

5.2.4 Policy-Based Sets

The g++ compiler also provides some data structures that are not part of the C++ standard library. Such structures are called *policy-based* structures. To use these structures, the following lines must be added to the code:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

After this, we can define a data structure `indexed_set` that is like `set` but can be indexed like an array. The definition for `int` values is as follows:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
             tree_order_statistics_node_update> indexed_set;
```

Then, we can create a set as follows:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

The speciality of this set is that we have access to the indices that the elements would have in a sorted array. The function `find_by_order` returns an iterator to the element at a given position:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

Then, the function `order_of_key` returns the position of a given element:

```
cout << s.order_of_key(7) << "\n"; // 2
```

If the element does not appear in the set, we get the position that the element would have in the set:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

Both the functions work in logarithmic time.

5.3 Experiments

In this section, we present some results concerning the *practical* efficiency of the data structures presented in this chapter. While time complexities are a great tool, they do not always tell the whole truth about the efficiency, so it is worthwhile to also do experiments with real implementations and data sets.

5.3.1 Set Versus Sorting

Many problems can be solved using either sets or sorting. It is important to realize that algorithms that use sorting are usually much faster, even if this is not evident by just looking at the time complexities.

As an example, consider the problem of calculating the number of unique elements in a vector. One way to solve the problem is to add all the elements to a set and return the size of the set. Since it is not needed to maintain the order of the elements, we may use either a `set` or an `unordered_set`. Then, another way to solve the problem is to first sort the vector and then go through its elements. It is easy to count the number of unique elements after sorting the vector.

Table 5.1 shows the results of an experiment where the above algorithms were tested using random vectors of `int` values. It turns out that the `unordered_set`

Table 5.1 The results of an experiment where the number of unique elements in a vector was calculated. The first two algorithms insert the elements to a set structure, while the last algorithm sorts the vector and inspects consecutive elements

Input size n	`set` (s)	`unordered_set` (s)	Sorting (s)
10^6	0.65	0.34	0.11
$2 \cdot 10^6$	1.50	0.76	0.18
$4 \cdot 10^6$	3.38	1.63	0.33
$8 \cdot 10^6$	7.57	3.45	0.68
$16 \cdot 10^6$	17.35	7.18	1.38

Table 5.2 The results of an experiment where the most frequent value in a vector was determined. The two first algorithms use map structures, and the last algorithm uses an ordinary array

Input size n	`map` (s)	`unordered_map` (s)	Array (s)
10^6	0.55	0.23	0.01
$2 \cdot 10^6$	1.14	0.39	0.02
$4 \cdot 10^6$	2.34	0.73	0.03
$8 \cdot 10^6$	4.68	1.46	0.06
$16 \cdot 10^6$	9.57	2.83	0.11

algorithm is about two times faster than the `set` algorithm, and the sorting algorithm is more than ten times faster than the `set` algorithm. Note that both the `set` algorithm and the sorting algorithm work in $O(n \log n)$ time; still the latter is much faster. The reason for this is that sorting is a simple operation, while the balanced binary search tree used in `set` is a complex data structure.

5.3.2 Map Versus Array

Maps are convenient structures compared to arrays, because any indices can be used, but they also have large constant factors. In our next experiment, we created a vector of n random integers between 1 and 10^6 and then determined the most frequent value by counting the number of each element. First we used maps, but since the upper bound 10^6 is quite small, we were also able to use arrays.

Table 5.2 shows the results of the experiment. While `unordered_map` is about three times faster than `map`, an array is almost a hundred times faster. Thus, arrays should be used whenever possible instead of maps. Especially, note that while `unordered_map` provides $O(1)$ time operations, there are large constant factors hidden in the data structure.

Table 5.3 The results of an experiment where elements were added and removed using a multiset and a priority queue

Input size n	`multiset` (s)	`priority_queue` (s)
10^6	1.17	0.19
$2 \cdot 10^6$	2.77	0.41
$4 \cdot 10^6$	6.10	1.05
$8 \cdot 10^6$	13.96	2.52
$16 \cdot 10^6$	30.93	5.95

5.3.3 Priority Queue Versus Multiset

Are priority queues really faster than multisets? To find out this, we conducted another experiment where we created two vectors of n random `int` numbers. First, we added all elements of the first vector to a data structure. Then, we went through the second vector and repeatedly removed the smallest element from the data structure and added the new element to it.

Table 5.3 shows the results of the experiment. It turns out that in this problem a priority queue is about five times faster than a multiset.