Chapter 4

Data structures

A **data structure** is a way to store data in the memory of a computer. It is important to choose an appropriate data structure for a problem, because each data structure has its own advantages and disadvantages. The crucial question is: which operations are efficient in the chosen data structure?

This chapter introduces the most important data structures in the C++ standard library. It is a good idea to use the standard library whenever possible, because it will save a lot of time. Later in the book we will learn about more sophisticated data structures that are not available in the standard library.

Dynamic arrays

A **dynamic array** is an array whose size can be changed during the execution of the program. The most popular dynamic array in C++ is the `vector` structure, which can be used almost like an ordinary array.

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]
```

After this, 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
```

The function `size` returns the number of elements in the vector. The following code iterates through the vector and prints all elements in it:

```
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 in the vector, and the function pop_back removes the last element:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

The following code creates a vector with five elements:

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

Another way to create a vector is to give the number of elements and the initial value for each element:

```
// size 10, initial value 0
vector<int> v(10);
```

```
// size 10, initial value 5
vector<int> v(10, 5);
```

The internal implementation of a vector uses an ordinary array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are moved to the new array. However, this does not happen often and the average time complexity of push_back is $O(1)$.

The `string` structure is also a dynamic array that can be used almost like a vector. In addition, there is special syntax for strings that is not available in other data structures. Strings can be combined using the + symbol. The function `substr`(k,x) returns the substring that begins at position k and has length x, and the function `find(t)` finds the position of the first occurrence of a substring `t`.

The following code presents some string operations:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

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.

The C++ standard library contains two set implementations: The structure `set` is based on a balanced binary tree and its operations work in $O(\log n)$ time. The structure `unordered_set` uses hashing, and its operations work in $O(1)$ time on average.

The choice of which set implementation to use is often a matter of taste. The benefit of the `set` structure is that it maintains the order of the elements and provides functions that are not available in `unordered_set`. On the other hand, `unordered_set` can be more efficient.

The following code creates a set that contains integers, and shows some of the 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.

```
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
```

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

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

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(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ also contains the structures `multiset` and `unordered_multiset` that otherwise work like `set` and `unordered_set` but they can contain multiple instances of an element. For example, in the following code all three instances of the number 5 are added 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 instances of an element from a multiset:

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

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

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

Map structures

A **map** is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the 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 implementations that correspond to the set implementations: the structure `map` is based on a balanced binary tree and accessing elements takes $O(\log n)$ time, while the structure `unordered_map` uses hashing and accessing elements takes $O(1)$ time on average.

The following code creates a map where the keys are strings and the 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
}
```

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

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

Iterators and ranges

Many functions in the C++ standard library operate with iterators. An **iterator** is a variable that points to an element in a data structure.

The often used iterators `begin` and `end` define a range that contains all elements in a data structure. The iterator `begin` points to the first element in the data structure, and the iterator `end` points to the position *after* the last element. The situation looks as follows:

```
{  3,  4,  6,  8,  12,  13,  14,  17  }
   ↑                                   ↑
   s.begin()                           s.end()
```

Note the asymmetry in the iterators: `s.begin()` points to an element in the data structure, while `s.end()` points outside the data structure. Thus, the range defined by the iterators is *half-open*.

Working with ranges

Iterators are used in C++ standard library functions that are given a range of elements in a data structure. Usually, we want to process all elements in a data structure, so the iterators `begin` and `end` are given for the function.

For example, the following code sorts a vector using the function `sort`, then reverses the order of the elements using the function `reverse`, and finally shuffles the order of the elements using the function `random_shuffle`.

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

These functions can also be used with an ordinary array. In this case, the functions are given pointers to the array instead of iterators:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Set iterators

Iterators are often used to access elements of a set. The following code creates an iterator `it` that points to the smallest element in a set:

```
set<int>::iterator it = s.begin();
```

A shorter way to write the code is as follows:

```
auto it = s.begin();
```

The element to which an iterator points can be accessed using the $*$ symbol. For example, the following code prints the first element in the set:

```
auto it = s.begin();
cout << *it << "\n";
```

Iterators can be moved using the operators ++ (forward) and -- (backward), meaning that the iterator moves to the next or previous element in the set.

The following code prints all the elements in increasing order:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

The following code prints the largest element in the set:

```
auto it = s.end(); it--;
cout << *it << "\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
}
```

The function `lower_bound`(x) returns an iterator to the smallest element in the set whose value is *at least* x, and the function `upper_bound`(x) returns an iterator to the smallest element in the set whose value is *larger than* x. In both functions, if such an element does not exist, the return value is `end`. These functions are not supported by the `unordered_set` structure which does not maintain the order of the elements.

For example, the following code finds the element nearest to x:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}
```

The code assumes that the set is not empty, and goes through all possible cases using an iterator `it`. First, the iterator points to the smallest element whose value is at least x. If `it` equals `begin`, the corresponding element is nearest to x. If `it` equals `end`, the largest element in the set is nearest to x. If none of the previous cases hold, the element nearest to x is either the element that corresponds to `it` or the previous element.

Other structures

Bitset

A **bitset** is an array whose each value is either 0 or 1. For example, the following code creates a bitset that contains 10 elements:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

The benefit of using bitsets is that they require less memory than ordinary arrays, because each element in a bitset only uses one bit of memory. For example, if n bits are stored in an `int` array, $32n$ bits of memory will be used, but a corresponding bitset only requires n bits of memory. In addition, the values of a bitset can be efficiently manipulated using bit operators, which makes it possible to optimize algorithms using bit sets.

The following code shows another way to create the above bitset:

```
bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

The function count returns the number of ones in the bitset:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

The following code shows examples of using bit operations:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Deque

A **deque** is a dynamic array whose size can be efficiently changed at both ends of the array. Like a vector, a deque provides the functions push_back and pop_back, but it also includes 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 internal implementation of a deque is more complex than that of a vector, and for this reason, a deque is slower than a vector. Still, both adding and removing elements take $O(1)$ time on average at both ends.

Stack

A **stack** is a data structure that provides two $O(1)$ time operations: adding an element to the top, and removing an element from the top. It is only possible to access the top element of a stack.

The following code shows how a stack can be used:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Queue

A **queue** also provides two $O(1)$ time operations: adding an element to the end of the queue, and removing the first element in the queue. It is only possible to access the first and last element of a queue.

The following code shows how a queue can be used:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Priority queue

A **priority queue** maintains a set of elements. The supported operations are 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.

While an ordered set efficiently supports all the operations of a priority queue, the benefit of using a priority queue is that it has smaller constant factors. A priority queue is usually implemented using a heap structure that is much simpler than a balanced binary tree used in an ordered set.

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;
```

Policy-based data structures

The g++ compiler also supports some data structures that are not part of the C++ standard library. Such structures are called *policy-based* data 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;
```

Now 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
```

And 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.

Comparison to sorting

It is often possible to solve a problem using either data structures or sorting. Sometimes there are remarkable differences in the actual efficiency of these approaches, which may be hidden in their time complexities.

Let us consider a problem where we are given two lists A and B that both contain n elements. Our task is to calculate the number of elements that belong

to both of the lists. For example, for the lists

$$A = [5,2,8,9,4] \quad \text{and} \quad B = [3,2,9,5],$$

the answer is 3 because the numbers 2, 5 and 9 belong to both of the lists.

A straightforward solution to the problem is to go through all pairs of elements in $O(n^2)$ time, but next we will focus on more efficient algorithms.

Algorithm 1

We construct a set of the elements that appear in A, and after this, we iterate through the elements of B and check for each elements if it also belongs to A. This is efficient because the elements of A are in a set. Using the `set` structure, the time complexity of the algorithm is $O(n \log n)$.

Algorithm 2

It is not necessary to maintain an ordered set, so instead of the `set` structure we can also use the `unordered_set` structure. This is an easy way to make the algorithm more efficient, because we only have to change the underlying data structure. The time complexity of the new algorithm is $O(n)$.

Algorithm 3

Instead of data structures, we can use sorting. First, we sort both lists A and B. After this, we iterate through both the lists at the same time and find the common elements. The time complexity of sorting is $O(n \log n)$, and the rest of the algorithm works in $O(n)$ time, so the total time complexity is $O(n \log n)$.

Efficiency comparison

The following table shows how efficient the above algorithms are when n varies and the elements of the lists are random integers between $1 \ldots 10^9$:

n	Algorithm 1	Algorithm 2	Algorithm 3
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

Algorithms 1 and 2 are equal except that they use different set structures. In this problem, this choice has an important effect on the running time, because Algorithm 2 is 4–5 times faster than Algorithm 1.

However, the most efficient algorithm is Algorithm 3 which uses sorting. It only uses half the time compared to Algorithm 2. Interestingly, the time complexity of both Algorithm 1 and Algorithm 3 is $O(n \log n)$, but despite this, Algorithm 3 is ten times faster. This can be explained by the fact that sorting is a

simple procedure and it is done only once at the beginning of Algorithm 3, and the rest of the algorithm works in linear time. On the other hand, Algorithm 1 maintains a complex balanced binary tree during the whole algorithm.