**HashMap in C++**

Table of Contents

[Inserting Values 2](#_Toc48982910)

[Iterators 4](#_Toc48982911)

[Time Complexity Problems 5](#_Toc48982912)

[Extra Features 6](#_Toc48982913)

[Examples of Uses of Hashing 7](#_Toc48982914)

[Finding the Number with the Most Occurrence in a Series 7](#_Toc48982915)

[Determining if an Array is a Subarray of Another Array 8](#_Toc48982916)

[Searching for a Substring 9](#_Toc48982917)

## Inserting Values

#include <iostream>  
#include <map>  
using namespace std;  
  
int main()  
{  
 map<char, int> myMap; *// char -> key; int -> value*  
 myMap['c'] = 99; *// direct access with key*  
 myMap['a'] = 97; *// memory is dynamically allocated*  
 myMap.insert({'d', 100}); *// alternative method*  
 cout<<myMap['a']<<" "<<myMap['c']<<endl;  
 *// prints 97 99*  
}

C++

Notice that the alternative method for inserting a key value pair using the insert() function makes use of a specific syntax. It is actually using another data type called a pair. Using curly braces like this creates a pair and passes is to the insert function, which uses the first value of the pair as the key and the second value as the value. We could also have used the insert() function like myMap.insert(make\_pair('d', 100)) or myMap.insert(pair<char, int> ('d', 100)).

An important note about the two methods of inserting values is how they deal with repeated keys. The basic map we have used above does not allow duplicate keys. If we try to insert a key that already exists using the insert() function, there will not be any errors, but the command will essentially be ignored. However, if we try to do the same thing using direct access, the old value for the key will be replaced with the new value.

myMap.insert({'c', 100}); *// fails*  
myMap['c'] = 100; *//works*

C++

If it is absolutely essential that we be able to use the same key multiple times, then we must use a different type of map called a multimap. Multimaps are the same as normal maps and we can do all of the same things, but it does not allow direct access, since obviously direct access cannot tell the compiler which of the multiple values to return.

multimap<char, int> myMultiMap;

C++

## Iterators

We can loop over a map in the same way that we loop over an array, but the syntax has some differences. With arrays, we can simply loop over the indices one after another from beginning to end. With maps however, the positions in between values that are empty literally do not exist. This is why we are forced to make use of a different type of object, called an iterator. An iterator can be set up to start at the beginning of the map and loop over each object in turn until it reaches the end, even though the hash table does not actually have ‘indices’ in the way that we think of them with arrays.

map<char, int> :: iterator it;  
for (it = myMap.begin(); it != myMap.end(); it++)  
 cout<<it->first<<" "<<it->second<<endl;  
 //prints key value

C++

Notice the way in which we had to declare the iterator. It was declared in reference to the specific type of map that it was meant to work with.

myMap.begin() returns a pointer to the key value pair at the start of the map. myMap.end() returns a pointer to the position just after the last element in the map.

There is an alternative way to use iterators, shown below. In this method, we do not need to separately specify the exact variation of map the iterator will be working with. It is automatically detected.

for (auto it = myMap.begin(); it != myMap.end(); it++)  
 cout<<it->first<<" "<<it->second<<endl;

C++

Please note that you may face problems using this method, since it requires C++14.

## Time Complexity Problems

If you use iterators to print a map that uses integers as the key, an odd property should pop out to you immediately. Every time, the pairs will be printed with the keys in ascending order. This happens regardless of the data type of the keys or values. This is definitely useful in many situations, since we are able to get a sorted list, but it is odd behaviour since the process of hashing does not store keys in any order.

The reason behind this odd behaviour is that we have not actually been dealing with hashing so far. In reality, the two maps we have seen make use of a balanced binary search tree, specifically a red-black tree, which is a variant of an AVL tree. Since it is a balanced binary tree, doing a simple in-order traversal prints the contents in a sorted order, even if this behaviour is not required. Sadly, this also means that every single operation we have looked at so far had a time complexity of O(log n) instead of the O(1) promised.

If sorting is not important to us, then we should not use the normal map. Instead, we should use the unordered map. This actually uses hashing, which means that all the operations we perform will have a time complexity of O(1). Note that the syntax of the operations themselves remains the same. Even unordered multimaps exist.

#include <unordered\_map>

unordered\_map<char, int> myMap;  
unordered\_multimap<char, int> myMultiMap;

C++

## Extra Features

There are a wide range of functions available for use with maps. One example is shown below:

myMap.erase(myMap.begin(), myMap.find('c'));

C++

Here, we are erasing all the values within a range beginning at the beginning of the map. We stop at the point where we find a specific key. Note that the key we stop at is not erased.

The find() method can also be used to check if a key is valid. If an invalid key is used, a pointer to the end of the table will be returned.

if (myMap.find('c') != myMap.end())  
 cout<<"Key is valid"<<endl;

C++

There are many other functions, such as to find the size of the map, check if it is empty, copy it to another map and more.

## Examples of Uses of Hashing

### Finding the Number with the Most Occurrence in a Series

Say we have an array of numbers, int arr[5] = {10, 20, 30, 20, 15}, and we have to find the number that occurs the greatest number of times. In the worst case, we would have a counter for each of the numbers and loop over the array repeatedly, each time checking how many times a different number occurs. This would give us a time complexity of O(n2). If we sort the array, then we know that when we are looking at one number, say 10, we will find all the occurrences of 10 together. If we find a different number, we know that there are no further occurrences of 10 in the array. This allows us to do the actually loop over the array just once, but along with the time needed to sort the array, the overall time complexity becomes O(nlog n).

The best solution is to create a second array, with a size of the maximum element in the first array, and use the values of the first array as indices to keep a count in the second array. Thus, when we encounter 10, we go to the 10th index of the second array and increment its value. Since we only have to loop over the original array once, and we are not having to sort anything, this gives us a time complexity of O(n). However, there is a problem here. If the values are very large numbers, the second array we take will be very large as well. This is where hashing becomes the proper solution. Instead of directly using the values as indices, we can take them to be keys and pass them to hash functions, using the results to store their count in a hash table.

### Determining if an Array is a Subarray of Another Array

Say we have two arrays, int arr1[6] = {11, 1, 13, 21, 3, 7} and int arr2[4] = {1, 3, 11, 7}. The worst possible solution is to loop over the first array repeatedly, each time search for one of the values in the second array. This would result in a time complexity of O(n2).

Hashing gives us a far better solution. We can take the first array and put its values in a hash map, using the values themselves as keys and storing the number of times they occur with the keys. In our case, the occurrence count for each number would be 1, but this is a good way to ensure we can also use this same method with arrays with repeated values. With this hash map set up, we can simply search for each of the values from the second array. If the first array is of length m and the second array is of length n, setting up the map would take O(m) time and searching for all the values would take O(n) time, giving us a total time complexity of O(m + n), or O(n).

### Searching for a Substring

Say we have two strings, string str1 = "baaaaabcde" and string str2 = "aab". At first glance, it may seem like using the method we used in the previous example would work here as well. However, that plan would be flawed. The previous method only checks if each of the characters in the second string occurs at all in the first string, but to accurately determine a substring, we need to ensure the order of the characters is the same as well.

The way to go about this, is to take ‘windows’ of characters from the first string, each of the same length as the length of the second string. In this example, the windows would be "baa", "aaa" and so on. The worst solution is to check each of these windows against the second string to determine if they are the same, meaning 3 characters need to be checked for each window in our case. If there are m windows and n characters in the second string, we will have a total time complexity of O(mn).

However, this can be improved upon by using a hashing function. We can pass each of these into a hashing function and calculate their total ASCII values. If the result of any window matches the total ASCII value of the second string, only then do we check each of the characters. Thus, we essentially only have to loop over the first array, meaning the time complexity is reduced to O(n). Even this we can improve upon. In the current algorithm, "baa" and "aab" would both appear to be matches, since they have the same characters in different orders. If we use a more complex hashing function which takes the position of the characters into account, these false positives will not occur.