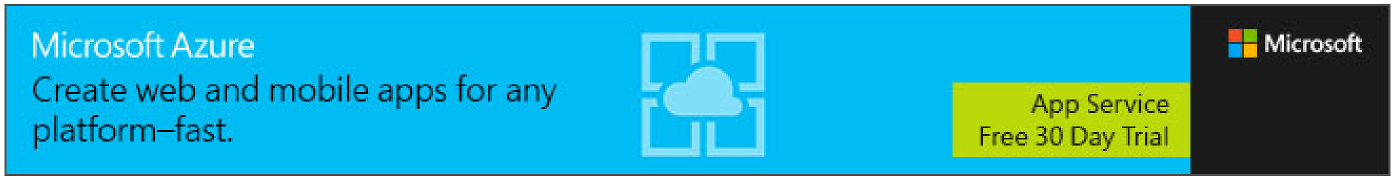
sign up log in tour help stack overflow careers

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free. Take the 2-minute tour ×

what the difference between map and hashmap in STL [duplicate]



This question already has an answer here:
map vs. hash_map in C++ *4 answers*

in C++ STL, there are two map, map and hashmap. Anyone know the main difference of them?

c++ stl

asked Feb 28 '11 at 8:57
user496949
17.2k 58 193 312

marked as duplicate by mgibsonbr, Lazy Ninja, sashoalm, DuckMaestro, Graviton May 21 '13 at 11:38

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

4 Answers

map uses a red-black tree as the data structure, so the elements you put in there are sorted, and insert/delete is O(log(n)). The elements need to be implement at least `operator<` .

hashmap uses a hash, so elements are unsorted, insert/delete is O(1). Elements need to implement at least `operator==` and you need a hash function.

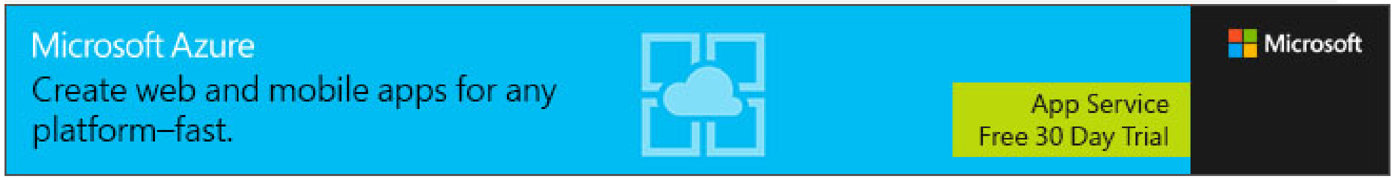
edited Feb 28 '11 at 9:31

answered Feb 28 '11 at 9:01
martinus
8,094 9 44 72

for map, does the element needs to support ==? – user496949 Feb 28 '11 at 9:07

2 @user: No, `std::map` uses *equivalence* based on `operator<` , not *equality* based on `operator==` . – fredoverflow Feb 28 '11 at 9:34

8 To clarify: equivalence means `!(a<b || b<a)` , i.e. "neither is smaller than the other". – MSalters Feb 28 '11 at 10:27



hash_map uses a hash table. This is "constant" time in theory. Most implementations use a "collision" hash table. What happens in reality is:

- It creates a big table
- You have a "hash" function for your object that generates you a random place in the table (random-looking, but the hash function will always return the same value for your object) and usually this is the mod of the actual 32-bit (or 64-bit) hash value with the size of the table.
- The table looks to see if the space is available. If so it places the item in the table. If not it checks if the element there is the one you are trying to insert. If so it is a duplicate so no insert. If not, this is called a "collision" and it uses some formula to find another cell and this continues until it either finds a duplicate or an empty cell.
- When the table gets filled up too much it resizes. An efficient (in time) implementation will store all the original hash values together with the elements so it won't need to recalculate the hashes when it does this. In addition, comparing the hashes is usually faster than comparing the elements, so it can do this whilst searching to eliminate most of the collisions as a pre-step.
- If you never delete anything it is simple. However deleting elements adds an extra

complexity. A cell that had an element in it which has been deleted is in a different state from one that was just empty all along, as you may have had collisions and if you just empty it, those elements won't be found. So there is usually some "mark". Of course now when we want to reuse the cell, we still have to recurse down in case there is a duplicate lower down (in which case we can't insert in this cell), then remember to reuse the deleted cell.

- The usual constraint is that your objects must be implemented to check for equality, but Dinkumware (or was it SGI) implemented theirs with operator< which might be slower but has the advantage of decoupling your elements and the type of associated container they can be stored in, although you still need a hash function to store in a hash.

The theory is that if you have a big enough table, the operations are constant time, i.e. it does not depend on the number of actual elements you have. In practice, of course, the more elements you have the more collisions occur.

std::map uses a binary tree. There is no need to define a hash function for an object, just strictly ordered comparison. On insertion it recurses down the tree to find the insertion point (and whether there are any duplicates) and adds the node, and may need to rebalance the tree so the depth of leaves is never more than 1 apart. Rebalancing time is relative to the depth of the tree too so all these operations are O(log N) where N is the number of elements.

The advantages of hash is the complexity The advantages of the tree is:

- Totally scalable. It only uses what it needs, no need for a huge table or to pre-empt the size of the table, although hash may require less "baggage" per element than a tree.
- No need to hash first, which for a good function can take longer than the comparisons would if the data set is not large.

One other issue with `std::map` is that it uses a single strictly-ordered comparison function whilst a "compare" function that returned -1, 0 or 1 would be a lot more efficient, particularly with the most commonly used key type, std::string, which already has this function implemented (it is `char_traits::compare`). (This inefficiency is based on the premise that to check that `x==y`, you check `x<y` and `y<x` so you do two comparisons. You would do this just once per lookup).

edited Jul 3 '14 at 22:48

answered Feb 28 '11 at 11:21

CashCow
21.8k 3 27 57

On the last issue I brought up, a more efficient compare strategy for associative containers map and set might be std::compare or whatever returns -1, 0 or 1. A default version can be created that uses (a<b) and (b<a) or std::less(a,b) and std::less(b,a) it just seems wasteful to call both the above on the occasiosn when we can optimise with a single call. – CashCow Jul 6 '12 at 10:13

Note that because there were some vendors of STL who provided hash_map and they were not the same, when it was added to the c++ standard it was called unordered_map. – CashCow Jan 15 '14 at 9:39

`map` is a red-black tree, `O(log(n))` access time. `hash_map` (which is not standard, however `unordered_map` will become standard) uses (conceptually) a hash of the key as an index in an array of linked lists, and therefore has a best-case access time of `O(1)` and a worst case of `O(n)` .

See http://en.wikipedia.org/wiki/Red-black_tree

answered Feb 28 '11 at 9:00
Erik
44k 3 114 145

The main difference is the searching time.

for few data is better map

for lots of data is better hashmap

anyway the tecnical answers given previously are correct.

answered Feb 28 '11 at 9:05
Matteo TeoMan Mangano
173 1 7

1 Your performance comments are increasingly true as the number of elements becomes astronomical, but may well be wrong for uses of a few thousand or even million elements... all depends on relative speeds of creating the hash value vs key comparison, # collisions and collision-handling techniques. As always, benchmark your actual usage if you care. – Tony D Feb 28 '11 at 9:40