Self-Balancing Trees

Jan 7, 2016

A lot of computer science problems call for associative data structures, meaning that we want a data structure mapping keys to values. There are a lot of ways to implement associative mappings, and the best data structure for the job depends heavily on what the keys look like, what access operations are most frequent, the size of the collection, and whether the association is on disk or in memory.

That said, there are two commonly used data structures for general purpose in-memory associations:

- Hash Tables
- Self-Balancing Trees (hereafter called a “balanced tree”)

A quick review of the running time for these two data structures:

	Hash Table	Balanced Tree
Space	O(n)	O(n)
Search	O(1)	O(log n)
Insert	O(1)	O(log n)
Delete	O(1)	O(log n)

It’s because of this chart that most people largely only know of hash tables. In fact, many popular high-level languages today like Python and Javascript don’t really give developers a choice in the matter: there is only one associative data structure, the hash table. If you wanted, you could implement a balanced tree in these languages, but even in the cases where a balanced tree would normally be better than a hash table it’s often not so in Python/Javascript because of the advantage that hash tables have as a natively implemented runtime feature.

The matter isn’t actually quite as simple as this though. Balanced treees have a lot going for them.

The Caveats Of Hash Tables

First, we have to consider what the running times above mean. Hash tables don’t really have O(1) time for all of the cases listed above. Usually what you will see in the literature is that a hash table has *amortized* O(1) running time for the above operations. What actually happens is that over time hash tables can become overloaded.

When this happens they suffer from hash collisions. When a hash collision occurs the value must spill over into a list of values who all share the same hash key. This kills performance: if the lists of values becomes too long then all of the operations suffer from linear list scans. In the worst case, where all of the values hash the same, all of the operations are actually O(n).

Typically hash tables have a way to "fix" this problem by growing the internal hash table size. The naive implementation of this algorithm takes O(n) time since all of the keys have to be re-hashed into a new hash table. With a good hashing algorithm this should only happen with frequency $1/n$ so the amortized time is still O(1); that is, with probability (n-1)/n the operation takes O(1) time, and with probability $1/n$ the operation takes O(n) time. However, even though the amortized time is O(1), there will be occasional very long O(n) pauses.

The problem of occasional O(n) re-hashing with a hash table can be solved with yet another trick. When the hash table needs to be resized, a new larger hash table is created. Each subsequent operation checks both tables, and when a value is in the old table it's copied into the new table. When the old table eventually is empty it can be deallocated. This maintains the O(1) behavior in all cases because two O(1) operations is still O(1). There's a cost to this though: the storage cost for the data structure is doubled. Additionally, if we have to rehash before the old table is completely emptied, there could still be a long pause to empty the old table.

All of this assumes, once again, that the hashing algorithm actually fairly hashes keys. If the hashing algorithm is unfair then the running time of things gets much worse due to hashing collisions. In fact, the hashing algorithm could be unfair and we could get particularly unlucky with our keys and have the same probem.

Futhermore, certain data structures can be costly to hash. For instance, there are a number of high performance non-cryptographic string hashing algorithms, e.g. MurmurHash. Typically all of these algorithms have to scan the entire key. If the keys are very large this could be a very expensive operation. If it's known in advance that large keys are expected this can be partially mitigated by just hashing part of the key, but this requires application-specific knowledge to implement effectively since it has to be known what part of the key is most likely to vary (e.g. variation is most likely at the front of the string, or variation is most likely at the end of the string). This same problem applies to equality checks: if for some reason it's costly to check if one key is equal to another then hash collisions become especially expensive since a slow equality comparison needs to happen.

To summarize, hash tables are only effective when the hashing operation is fast and when hash collisions are infrequent. There are two operations required on keys to implement a hash table: a hashing function, and an equality check function (in case of hash collisions). There are a lot of things that one can get wrong implementing a hash table, and these mistakes can be costly.

The Benefits Of Balanced Trees

Self-balancing trees are frequently implemented as red-black trees, although there are a number of other algorithms (e.g. AVL trees, splay trees).

Balanced trees are a lot simpler than hash tables. Usually only one operation is needed to implement a balanced tree: a comparator function that takes two keys *k1* and *k2* and returns a value LT if *k1 < k2*, EQ if *k1 = k2*, and GT if *k1 > k2*. This is the same operator that is required to sort a list of values. An example of this operator that you might be familiar with is the classic strcmp() of `<string.h>` fame. Because there is only one operation required, and because it is well defined (unlike hashing), there are a lot fewer gotchas with balanced trees than with hash tables.

At first glance the slower O(log *n*) time for balanced tree operations seems bad. In fact, all of the operations have expected O(log *n*) time; but they also have worst case O(log *n*) time. There aren't any weird edge cases where something could take O(*n*) time. Things are much more predictable with a balanced tree.

Scanning

There is one place where balanced trees really shine compared to hash tables: scanning. First, allow me to explain exactly what this means.

To an end user, a balanced tree actually looks like a sorted list of key/value tuples. By way of analogy, searching for a value in a sorted list is O(log *n*), and thus searching for a given key in the sort list of key/value tuples is also O(log *n*).

Sorted list data structures can do a couple of interesting things. For one thing, we can traverse the list quickly once we find an element, since each element has a pointer to the next and previous elements in the list. A balanced tree usually also supports finding the next or previous node for any key/value pair stored in the tree in amortized O(1) time. This means that we can scan the list in sorted order in O(*n*) time. The same operation on a hash table would take O(*n*) time to scan the table and then another O(*n* log *n*) time to sort the list of keys. And that's not all: because balanced trees internally store nodes as key/value pairs, for each tuple scanned in a balanced tree we can immediately access the value without hashing. Besides the overhead of hashing, frequently the value (or at least the value pointer) will be stored in the same cache line as the key (or key pointer), which makes this data structure very CPU-friendly.

Another interesting property of balanced trees is that we can find values "in between" nodes. For instance, suppose that our list of keys is: [1, 3, 5, 6, 7, 9, 10]. A tree will support a function where given the key 8, it will either return the pointer to 7 or the pointer to 9 (depending on if you want the smaller element or the larger element). There are a few reasons this is useful.

One reason that the "in between" search function is useful is because it lets one implement certain data structures like inteval trees trivially. It is not possible to implement this data structure efficiently only using a hash table.

Another nice property of the "in between" search function is that it makes the common case of checking for a key, and then inserting a value if it's not present, more efficient. Consider the following example from Python:

```
d = {}
...  # some things happen to fill d
try:
    actual_val = d[k]
    print "the val already existed and is " + str(actual_val)
except KeyError:
    d[k] = v
    print "the val did not already exist and is now" + str(v)
```

In a tight loop this can be a problem because we hash the key k twice: once to check if it's in the dictionary, and then again to insert it into the dictionary. Actually, things are worse because if the key is usually missing, as catching an exception is somewhat expensive. Python has a special setdefault() method to handle this use case, but it's not intuitive and is frequently overlooked.

By comparison, here's how this would look in C++ using the `std::map` type:

```
map<int, string> m;
...  // some things happen to fill m
auto it = m.lower_bound(k);
if (it != m.end() && it->first == k) {
  cout << "key already existed, value is " << it->second << "\n";
} else {
```

```
  m.insert(it, make_pair(k, v));
  cout << "inserted new value " << v << "\n";
}
```

I'm not going to try to defend the extra parentheses and semicolons, but you can see here how the C++ API actually works. The `lower_bound()` method returns an "iterator" to the in-between place between two values. If the iterator returned doesn't equal the input key then we can ask the map to insert the new key/value pair, and we can use the result from `lower_bound()` as a hint to the insert function to tell it where to go.

Conclusion

If you're writing in a high level language like Python, Javascript, Ruby, or PHP you probably should go with the weird language-specific optimized hash table implementtion that the runtime gives you. In most cases the built in associative structure for each of these languages will be faster than anything implemented in the interpreted runtime.

If you're fortunate enough to be writing code in a language like C, C++, or Java where it's easy to use specific high-performance data structures for different use cases, you should definitely consider the pros and cons of using a balanced tree v.s. a hash table when writing code. Particularly if you find yourself in situations where you need to do something build build a hash table and then iterate over the keys in sorted order, you may find that using a balanced tree data structure (usually called a "map" in these languages) is both simpler (i.e. fewer lines of code to write) and faster.
