# A concrete illustration of practical running time vs big-O notation

devblogs.microsoft.com/oldnewthing/20090612-00

June 12, 2009

Raymond Chen

One of the five things every Win32 programmer needs to know is that memory latency can throw your big-*O* computations out the window. Back in 2003, I ran into a concrete example of this.

Somebody started with the algorithm presented in *Fast Algorithms for Sorting and Searching Strings* by Jon L. Bentley and Robert Sedgewick, implemented it in C#, and compared the performance against a `HashTable`, `TernaryTree` and `SortedList`. Surprisingly, the hash table won on insertion and retrieval of tens of thousands of randomly-generated strings. Why?

Remember, big-*O* notation hides the constants, and those constants can get pretty big. What's more, the impact of those constants is critical for normal-sized workloads. The big-*O* notation allows you to compare algorithms when the data sets become extremely large, but you have to keep the constants in mind to see when the balance tips.

The central point of my presentation at the PDC was that complexity analysis typically ignores memory bandwidth effects and assumes that all memory accesses perform equally. This is rarely true in practice. As we saw, leaving L2 is a big hit on performance, and accessing the disk is an even greater hit.

The tree doesn't rebalance, so inserting strings in alphabetical order will result in a bad search tree. (They do call this out in their paper.) To locate a *k*-character string, Bentley-Sedgewick traverses at least *k* pointers, usually more. ("How much more" depends on how many prefixes are shared. More shared prefixes = more pointers traversed.) It also requires $k(4p)$ bytes of memory to store that string, where $p$ is the size of a pointer. Remember those pesky constants. High constant factor overhead starts to kill you when you have large datasets.

More on those constants: Complexity analysis assumes that an add instruction executes in the same amount of time as a memory access. This is not true in practice, but the difference is a bounded constant factor, so it can be ignored for big-*O* purposes. Note, however, that that

constant often exceeds one million if you take a page fault. One million is a big constant.

Going back to memory bandwidth effects: At each node, you access one character and one pointer. So you use only 6 bytes out of a 64-byte cache line. You're wasting 90% of your bus bandwidth and you will certainly fall out of L2.

Bentley-Sedgewick says that this is beaten out by not traversing the entire string being searched for in the case of a miss. *I.e., their algorithm is tuned for misses*. If you expect most of your probes to be misses, this can be a win. (The entire string is traversed on a hit, of course, so there is no gain for hits.)

Note also that this "cost" for traversing the string on a miss is overstated due to memory bandwidth effects. The characters in a string are contiguous, so traversing the string costs you only $L/64$ cache lines, where $L$ is the length of the string, and one potential page fault, assuming your string is less than 4KB. Traversing the tree structure costs you at least $L$ cache lines and probably more depending on your branching factor, as well as $L$ potential page faults.

Let's look at the testing scenario again. The testing was only on hits, so the improved performance on misses was overlooked entirely. What's more, the algorithm takes advantage of strings with common prefixes, but the testing scenario used randomly-generated strings, which generates a data set opposite from the one the algorithm was designed for, since randomly-generated strings are spread out across the problem space instead of being clustered with common prefixes.

Those are some general remarks; here are some performance notes specific to the CLR.

I don't know whether it does or not, but I would not be surprised if `System.String.GetHashCode` caches the hash value in the string, which would mean that the cost of computing the hash is shared by everybody who uses it in a hashing operation. (Therefore, if you count the cost incurred only by the algorithm, hashing is free.)

Note also that Bentley-Sedgewick's `insert()` function stores the object back into the tree in the recursive case. Most of the time, the value being stored is the same as the value that was already there. This dirties the cache line for no reason (forcing memory bandwidth to be wasted on a flush) and—particularly painful for the CLR—you hit the write barrier and end up dirting a whole boatload of cards. A very small change avoids this problem: Change

```
p->eqkid = insert(p->eqkid, ++s);
```

to

```
Tptr newkid = insert(p->eqkid, ++s);
if (p->eqkid != newkid) p->eqkid = newkid;
```

(and similarly in the other branches). “This code is short but subtle, and worth careful study.” How very true.

Note also that if you use their “sleazy hack” of coercing a string to a `Tptr`, you had to have changed the type of `eqkid` from `Tptr` to `object`. This introduces a CLR type-check into the inner loop. Congratulations, you just tubed the inner loop performance.

Now go to the summary at the end of the article.

1. “Ternary trees do not incur extra overhead for insertion or successful searches.” I’m not sure what “extra” means here, but hash tables have the same behavior.
2. “Ternary trees are usually substantially faster than hashing for unsuccessful searches.” Notice that they are optimized for misses.
3. “Ternary trees gracefully grow and shrink; hash tables need to be rebuilt after large size changes.” True, but the CLR hashtable does this so you don’t have to. Somebody wrote it for you.
4. “Ternary trees support advanced searches such as partial-match and near-neighbor search.” These operations weren’t tested.
5. “Ternary trees support many other operations, such as traversal to report items in sorted order.” These operations weren’t tested either.

Notice that the article doesn’t claim that ternary trees are faster than hashing for successful searches. So if that’s all you’re testing, you’re testing the wrong thing. One of the big benefits of ternary trees is the new operations available (4 and 5), but if you’re not going to perform those operations, then you ended up paying for something you don’t use.

There are several morals of the story.

1. Constants are important.
2. Memory bandwith is important.
3. Performance optimizations for unmanged code do not necessarily translate to managed code.
4. What are you really testing?

Mind you, Bentley and Sedgewick are not morons. They know all this.

[Typo fixed 11am, thanks Nathan_works and Jachym Kouba.]

Raymond Chen

**Follow**