CSE250 Lecture Notes Weeks 7–8, K-W chs. 4–7

Data Structures—Performance and Objectives

Kenneth W. Regan
University at Buffalo (SUNY)

Weeks 7–8+

Basic Data Structure "Facts of Life"

Any data structure that supports iterators gives "fingered access" in $O(1)$ time. But what if we only have an address, relative directions, a key to match, or a feature to search?

- An array/`vector` provides *indexed access* in $O(1)$ time.
- But `insert` or `erase` of items "in the middle," has a worst-case $\Theta(n)$ time—owing to the need to "Move House(s)."
- A linked list can do `insert/erase` or other "splicing" from an iterator in $O(1)$ time.
- But it has no addressing, and search may require polling all n items.

Can we do both search and splicing in $o(n)$ time?—such as $O(\sqrt{n})$ or $O(\log n)$ time?

Can we "hash out" a way to do both in $O(1)$ time?

Always? Reliably?

Compromises and "Amortized" Performance

[Diagram and discussion of the "Valli" data structure, which is a vector of n/r pointers into a sorted n-item linked list.]
[Name stands for "Vector And Linked List + Iterator"—on a machine named for Justin Timberlake, it commemorates Frankie Valli.]

- *Amortized* $O(g(n))$ performance means that
  - The time for a single call is usually $O(g(n))$, and/or
  - The *average* amount of time over a long series of calls is $O(g(n))$.
  - This is also allowed to depend on whether the distribution of data items and/or calls is "normal" versus "adversarial."
- The `refresh` policy for `Valli` needs to be timed carefully.
- Adversarial: a burst of inputs with the same or similar keys.
- Basic binary search trees have the same problem!

Performance Goals of "`Valli`"

- Promise of `Valli` is $O(\log(n/r) + r)$ time lookup. If $r = \Theta(\log n)$, this becomes $O(\log n)$
- If (say) $r = \sqrt{n}$, then the time is $O(\log(n) + \sqrt{n}) = O(\sqrt{n})$, which isn't as good.
- (Why aren't we saying "Θ" in the last item? Because you can get lucky!—the item you want might be at the beginning of the length-r segment.)
- Also important is that the vector only has about n/r entries. Not only does this reduce the extra space needed for the vector (compared to the list), it also reduces how often one needs to resize/`refresh` the vector compared to keeping one pointer per list node.
- Thus putting r significantly down also degrades the performance. Having $r = \Theta(\log n)$ is a "sweet spot."
- But a fixed value such as $r = 20$ can be "sweet" for a lot of sizes, roughly up to 2^{20} = a million!

How much worse is "Amortized"—?

- Suppose we call `refresh` whenever the number n of items doubles.
- Waiting until $n = 2r$, as hinted on the project spec, means refreshing when the size hits 40, 80, 160,. . .
- Suppose $n = 10 * 2^k$, e.g. $k = 12$, $n = 40,960$ = about 40,000.
- We have spent "Theta-Of" $40 + 80 + 160 + \cdots + 20,480 + 40,960 =$ about $80,000$ on refreshes.
- This is about $2n$, and we inserted n items, so the average cost **of `refresh` per `insert`** was 2 times whatever constant is in the "Theta" for saying `refresh` takes $\Theta(n)$ time.
- That's less on average than the $\Theta(\log n)$ time that binary search takes to find where to insert the next item in the first place.
- Thus although `refresh` seems a pain when it's called, its *amortized* cost is *negligible*, meaning Little-Oh of something else, $2 = o(\log n)$.
- (This means there's slack to refesh a little more often and so maybe improve the performance of `find`. . .)

Binary Search Trees

[Guest lecture. It was review for many, but commences the intent stated in Week 1 that coverage would be sequential once Chapter 8 was hit. I've located it here amid the "amortized" discussion because it sets up the important examples on the next slides.]

Other "Amortized $O(1)$" Examples

- Text has the same doubling strategy for the `capacity` of a `KW::vector` in Chapter 4, and explains similar results.
- Implementations of the standard STL `vector` usually do similarly.
  - When a `vector` changes its capacity it generally migrates to a new block of memory, thus *invalidating* any iterators unwisely left on it—demo `STLinvalids.cpp`.
- Classic example is next-step in a binary-tree tra(ns)versal, which can be coded as `operator++` for an iterator.
  - Main transversals are *preorder*, *inorder*, and *postorder*, but for a binary *search* tree, only *inorder* respects the sorting.
- A binary tree with n nodes has $n - 1$ edges. In any of the three main transversals, each edge is stepped thru twice. Hence the average time per step is proportional to $(2n - 2)/n = 2 - 2/n$, which is about (and no more than) 2.
- Thus although some individual steps can take a long time, the `operator++` takes "amortized $O(1)$" time.

Adversarial Data

- A more-important variable in data-structure performance is the sequence in which data items are presented.
- Example: `Valli` will "degrade" if a burst of consecutive inserts have keys that are equal or near each other in sorted order.
  - A burst of toys tagged "McDonald's..." could blow up the segment between mileposts on "Ma..." and "Me..." Then a search for (e.g.) "McFarlane Models" would have to wade thru all the "McD" stuff, just like on a singly-linked list.
  - A `refresh` would cure that, but with the global "doubling" strategy mentioned above, it might take a long time.
  - Hence the idea of refreshing whenever the # of items between any two mileposts doubles...—but it's harder to code.
- With basic binary search trees (as in Ch. 8, as opposed to *balanced* trees to come in Ch. 11), things can be as bad or even worse!

"Scraggly" Trees

- To generate an extreme example, insert 20 then 19,18,17,...,1.
- A basic BST created in that order will be a line angling down left.
- No lookup-time advantage over a (sorted) linked list.
- Note also that `iterator begin() const` will take $n - 1$ steps walking from the root until it finds the node with 1.
    - The "$2 - 2/n$" amortized performance is not upset by this, because the final $n - 1$ steps will take 1 hop each.
    - If the inserts were in order 1,2,3,...,20 then `begin()` would be immediate, but the last application of `operator++` before the iterator hits `end()` would take $n - 1$ upward steps.
- A "random permutation" of any data will give you a pretty balanced tree (this is proved as a theorem in some texts), BUT—
- *Real-world data often isn't random!* In particular, it's natural to copy into one data structure from another in sorted order. If you copy into a basic BST that way, you're...scraggly!
    - `Valli` won't be affected as badly, as refreshes will fix things.

What Do We Try to Guarantee?

- The basic tradeoff: going for the best *long-run* performance may require tolerating some bad short-runs.
- Or worse, it may expose us to rare(?) cases that could give data structures like BSTs "permanent bad shape."
- Alternative: smooth out the bumps so no instance is too bad—but you need a more-complicated data structure that does overall more work, even for "good data cases"!
- Example: the Red-Black Tree (Ch. 11) actually used by the C++ STL for *all of* `set`, `multiset`, `map`, `multimap`. It *guarantees $O(\log n)$ time performance* for `insert`, `erase`, `find`,...—but with a higher *principal constant* "hiding under the O"!
  - Hence, one can often compete with it!
  - According to the article cited at the end of the Project 1 spec, a simple sorted vector is better for some situations.
  - `Valli` is almost as good, for almost all situations!
- Hash Tables can Beat The Tree, but, not with the guarantees. . .

More on Binary Search Trees

- [Coverage from official K-W text notes, and code.]
- [Code examples, including `iterator` class which the text doesn't give, and same STL-conformant interface—also largely to illustrate issues with set-vs.-multiset in Chapter 9.]
- Text's code for `insert` is recursive, likewise `erase` and `find`.
- Actual STL code is iterative, and uses extra `parent` links.
- Either way, and as-usual, `erase` is the most difficult to code. When a *non*-leaf is deleted, one needs to find another node to put in its place.
    - Can be either the inorder successor or the inorder predecessor—text does the latter.
- Despite recursion being "high-level", text uses an important "low-level" detail: *passing pointers by reference*. Needs a slide to itself...

*&

The main public `insert(const I& item)` method *delegates* to a private "helper method"

```
void insert(const I& item, Node*& local_root) {
   ...
   if (local_root == NULL) {  //standard uses dummy NIL node instead
      local_root = new Node(item,NULL,NULL);
   }
   ...
}
```

Note that if you had a `current` pointer as parameter, you would hit `if (current == NULL) { ...` at the same place. If you then did `current = new Node(...);` you would link the new node only to `current`. You wouldn't link it to any parent node!

Passing the link by reference ensures that you modify the link itself.

(Hence I would prefer naming it `link` or `link2localRoot`.)

Alternative Implementations [FYI, not in text]

- If you maintain a `prev` pointer, you can assign the new node to `prev->left` or to `prev->right` accordingly, with no `*&` needed.
- But you need to test `if (prev->left == NULL)` and/or `if (prev->right == NULL)` along with `lessThan`, which is slower.
- Having a `parent` link doesn't let you rest easy with a `current` pointer, the same way a `prev` link does in a doubly-linked list—because when you hit `current == NULL`, it has no `parent`!
- Or even if you use a dummy `NIL` node, unlike a dummy end-node in a list, it doesn't have a unique parent!
- Pointer-to-pointer also works, aka. *double indirection*:

```
void insert(const I& item, Node** ptr2link) {
   ...
   if (*ptr2link == NULL) {
      *ptr2link = new Node(item,NULL,NULL);
   }
   ...  //[the best singly-linked lists are similar!]
```