CSE 241 Class 12

Jeremy Buhler

October 12, 2015

Onwards to trees!

1 Collection Types Revisited

A while back, we introduced the idea of *collections*.

- abstract data types
- support **insert**, **remove**, **find**
- examples: list, array, hash table

Where did we end up?

- could implement three ops in *expected* constant time
- worst-case linear time
- performance was highly sensitive to bad inputs

No self-respecting computer scientist would be satisfied with these properties. Can we make stronger sublinear-time guarantees?

2 Ordered Collections

We will extend the collection ADT with some extra operations that imply a *total ordering* on keys (not assumed for, e.g., hash tables).

- **T.min()** – return the record in T with smallest key
- **T.max()** – return the record in T with largest key
- **T.succ(x)** – if x is a record in T, return the record with the next larger key (or *null* if x is the max)
- **T.pred(x)** – if x is a record in T, return the record with the next smaller key (or *null* if x is the min)

Why the extension?

- ordering information can be practically useful

- **Example**: list of students sorted by GPA
- ordered collections can often be searched faster than unordered collections!
- **Example**: sorted vs unsorted array

the trick: support fast search *and* fast insertion/deletion while maintaining ordering property.

3 Binary Search Trees

Time for new data structure!

- holds collection of records with ordered keys
- one node per key (could hang record off node)
- each node has pointers to **left** and **right** children and a **parent** pointer

Here's an example of a binary search tree:

Where is the ordering?

- Every correct binary search tree maintains the following invariant (the **BST property**):
- Let x and y be nodes in tree.
- If node y is in *left subtree* of x, then

$$y.\text{key} \leq x.\text{key}.$$

- If node y is in *right subtree* of x, then

$$y.\text{key} > x.\text{key}.$$

- (Note that second inequality is strict; could instead have made first strict and second non-strict. *One* must be strict for search to work efficiently in the presence of duplicate keys.)

Colloquially, every element on left side of any subtree is smaller than root, while every item on right side is larger than root.

[**illustrate BST prop and violations on example tree**]

4 Tree Operations

Some tree operations are "obvious."

- **find** (animate on tree)
- **insert**
- **min/max** (where are they?)

If you have any question about how these ops work, look at your text.

- Let's talk about **succ**.
- To start, how would I print all the keys in a tree in order? (a so-called "in-order traversal")
- Pseudocode:

```
InOrder(x)
  if x ≠ null
      InOrder(x.left)
      print x.key
      InOrder(x.right)
```

- **Lemma**: InOrder(T.root) prints elements of T in correct order.

Proof: by induction on tree height h
Base: $h = 1$ (a single node). Trivial!
Ind: Suppose true for $h' < h$. Consider general tree of height h:

By BST property, left subtree contains keys $\leq$ than root, while right subtree contains keys $\geq$ root. By i.h., left and right subtrees printed in correct order, and root lies between them. QED

- What does this have to do with **succ**? Quite a lot.
- Suppose I hand you x in T.
- If x has a right subtree T', its successor is T'.min().
- If x has *no* right subtree?
- What does in-order traversal do?

- There is some subtree $\hat{T}$ of T for which x is the rightmost node in $\hat{T}$'s left subtree.
- Traversal proceeds from x to root of $\hat{T}$.

```
SUCC(x)
  if x.right ≠ null
      return x.right.min()
  else
      y ← x.parent
      while y ≠ null and x = y.right do
          x ← y
          y ← y.parent
      return y
```

What is complexity of **succ**?

- either walks down to bottom of tree (**min**)
- or up, perhaps from leaf to root
- either way, cost is $O(h)$, where h is tree height!
- **pred**? Just like SUCC, but lefts and rights are reversed. You can work it out.

5 Deletion

Insertions always happen at the bottom of the tree. Deletions, alas, can happen **anywhere**.

- Let's consider three cases for deletion of x.
- **Case 1**: x has no children? (obvious: remove it)
- **Case 2**: x has one child?
- remove x; replace parent's ptr to x with pointer to child.
- **Case 3**: x has two children?

```
REMOVE(x)
  if x.left = null and x.right = null
      remove x
  else if x.left = null or x.right = null
      replace appropriate child of x.parent with x's child
  else
      y ← SUCC(x)
      x.key ← y.key                    ▷ also copy rest of record at y into x
      REMOVE(y)
```

Example:

6 Correctness of Deletion

Claim: REMOVE is correct.

- Start by showing that removal *terminates*.
- case 1, case 2 non-recursive; focus on recursive case 3.
- Let $y = \text{succ}(x)$. Two cases...
- If y is $\min(x.\text{right})$, y has no left child.
- Hence, recursive call terminates in case 1 or 2.
- If y is *not* $\min(x.\text{right})$, x must have a null **right** pointer.
- But then x cannot have two children, so we can't be in case 3!

Still must show that if BST property holds originally (for every pair of nodes), then it holds after REMOVE(x).

- Case 1: trivial – deletion but no rearrangement.
- Case 2: if x is right child of its parent z, then then every node below x is $> z$; hence, subtree below x can become z's right subtree. Similar proof if x is left child of z.
- Case 3:
  1. Node y is immediate successor of node x in tree.
  2. Hence, every other node's key is either $\leq x$ or $\geq y$.
  3. Conclude that replacing x's key with y's key cannot violate BST property between x and any other node.
  4. Removing y is now case 1 or 2, so BST property is maintained by previous argument.
- Conclude that REMOVE is correct. QED

7 Speed

All major tree operations – insert, remove, find, pred/succ – require time $O(h)$, where h is tree height.

- Good case: $h = O(\log n)$ ("balanced tree")
- Bad case: $h = n$ (a stick!)
- Does some series of insertions cause $h = n$?
- Yes! Insert keys in sorted order.

Oh dear. How can we avoid linear-time worst case?

- Randomized approach: skip lists (next time).
- Deterministic algos to maintain balance: B-trees (later).