Lecture 18: Binary Search Trees

Fall 2016

Kim Bruce & Peter Mawhorter

This Week

- Quiz on Friday:
  - Binary Trees; Heaps; Binary Search Trees
- Lab today: Iterators
  - Write an iterator for `CompressedTable`
- Assignment: Darwin
  - 2-week assignment
  - Program creatures that compete and infect each other

Assignment 3

- Grades are back
  - As usual, if your grade is not what you expect, talk to me or one of the TAs

Binary Search Tree

- A binary tree is a binary search tree if:
  - It is empty, or
  - The root value is greater than or equal to every node in the left subtree and less than or equal to every node in the right subtree, and both subtrees are binary search trees

Interface

```
public class BinarySearchTree<E extends Comparable<E>> {
  protected BinaryTree<E> root;

  public void add(E value) { ... }
  public void contains(E value) { ... }
  public void remove(E value) { ... }

  protected BinaryTree<E> locate(BinaryTree<E> root, E val){ ... }
  protected BinaryTree<E> predecessor(BinaryTree<E> node){ ... }
  protected BinaryTree<E> removeTop(BinaryTree<E> topNode){ ... }
}
```

Implementation

- Public methods:
  - `add`
  - `contains`
  - `remove`
- Protected methods:
  - `locate`
  - `predecessor`
  - `removeTop`

Locating a Value

- Useful for `add`, `contains`, and `remove`
- Returns a pointer to the node with a given value
  - …or to a node where that exact value could be added
- Recursive implementation (could be iterative)

Locating a Value

- Check current value vs. the search value
  - If equal, return this node
  - If smaller, `locate` within left subtree
  - Else within right subtree
  - If the appropriate subtree is empty, return this node

BinarySearchTree.java

Using Locate to Add

- Case one: locate returns closest node
  - Add a new left/right child if value is smaller/larger than result
- Case two: locate returns exact node
  - Duplicates go in left subtree (arbitrary)
  - Insert right of rightmost descendent

Predecessor

- Finds the rightmost descendent in left subtree
  - The next-smallest value in the tree
  - What's the big-O runtime?

Removing Nodes

- Easy cases:
  - Node is a leaf
  - Node has only one child
- Hard case:
  - Node has two children

Can assume that we're removing the root

Succession Wars

- Where can we find a node which:
  - Is ≥ all left children and ≤ all right children
  - Has at most one child

Hint: we've already got a function for this!

Succession Wars

- The *predecessor*, or rightmost child of left subtree, has both properties
  - Make that value the new root
  - Replace it with its left subtree (it didn't have a right subtree)
  - Result is still a binary search tree

What's the big-O run time?

Keeping Trees Trimmed

- Lots of operations are O(h)
- But our guarantee is just ~ log $n \leq h \leq n$
- Can we do better?

Keeping Trees Trimmed

- We can rotate a left child upwards:
  1. Give our right subtree to our parent as a left subtree
  2. Set our parent as our own right subtree
  3. Take our parent's old position
- All of our left descendants move up
- All of our parent's right descendants move down
- Our right descendants don't change height

Keeping Trees Trimmed

- The symmetric operation can rotate to the right
- A sequence of rotations can move a node to the root
- Bystander nodes end up more-balanced

Splay Trees

- Every time we find, get or add, rotate up to the root
  - Side-effects of rotation give average-case O(log n) tree height
  - Worst case is still O(n)
  - But all O(h) operations are now average-case O(log n)

Splay Trees

- Theory vs. practice
  - All that rotation is expensive
  - Great theoretical properties
  - Simple idea
  - Worse performance than other balancing schemes