Lecture 23: Binary Search Trees

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki
A binary tree is a binary search tree iff

- it is empty or
- if the value of every node is both greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree.
public class BinarySearchTree> {
    protected BinaryTree root;
    public void add(E value);
    public void contains(E value);
    public void remove(E value);
    protected BinaryTree locate(BinaryTree root, E val);
    protected BinaryTree predecessor(BinaryTree node);
    protected BinaryTree removeTop(BinaryTree topNode);
}
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
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
    E rootValue = root.value();
    BinaryTree<E> child;
    if (rootValue.equals(value)) return root; // found at root
    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue, value) < 0) {
        child = root.right();
    } else {
        child = root.left();
    }
    // no child there: not in tree, return this node,
    // else keep searching
    if (child.isEmpty()) {
        return root;
    } else {
        return locate(child, value);
    }
}
Using locate to add a node

Case One: Locate returns pointer to where node should be added
  • If value less than returned node, create new left child
  • If value greater than returned node, create new right child
Using locate to add a node

Case Two: Locate returns pointer to node with same value
- Duplicates go in left subtree (could have chosen right)
- Where in the left subtree?

3 = locate(root, 3)
8 = locate(root, 8)
14 = locate(root, 14)
Using locate to add a node

Case Two: Locate returns pointer to node with same value

- Duplicates go in left subtree (could have chosen right)
- *Should be the rightmost descendant*
Predecessor

Finds the rightmost descendent in left subtree
- The next-smallest value in the tree
- What’s the big-O runtime?
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    BinaryTree<E> result = root.left();
    while (!result.right().isEmpty()) {
        result = result.right();
    }
    return result;
}

protected BinaryTree<E> successor(BinaryTree<E> root) {
    BinaryTree<E> result = root.right();
    while (!result.left().isEmpty()) {
        result = result.left();
    }
    return result;
}
public void add(E value) {
    BinaryTree<E> newNode = new BinaryTree<E>(value, EMPTY, EMPTY);
    // add value to binary search tree
    // if there's no root, create value at root
    if (root.isEmpty()) {
        root = newNode;
    } else {
        BinaryTree<E> insertLocation = locate(root, value);
        E nodeValue = insertLocation.value();
        // The location returned is the successor or predecessor
        // of the to-be-inserted value
        if (ordering.compare(nodeValue, value) < 0) {
            insertLocation.setRight(newNode);
        } else {
            if (!insertLocation.left().isEmpty()) {
                // if value is in tree, we insert just before
                predecessor(insertLocation).setRight(newNode);
            } else {
                insertLocation.setLeft(newNode);
            }
        }
    }
    count++;
}
Removing nodes - Easy cases

- Node is a leaf
- Node has only one child
Removing nodes - General Case

Left Child has a right subtree:
Removing nodes

- Calling `remove(E val)` removes node with value `val`
- Predecessor of root becomes new root
  - Predecessor is in left subtree
  - Predecessor has no right subtree
- Complexity is $O(h)$ where $h$ is height of tree
  - Worst-case $O(h)$ to locate
  - Worst-case $O(h)$ to find predecessor
Complexity

- `locate, add, contains, remove` are all $O(h)$
- Can we guarantee that $h$ is $O(\log n)$?
  - Only if tree stays balanced!!
- Binary search trees that stay balanced
  - AVL trees
  - Red-black trees
- We’ll do splay tree, which doesn’t guarantee balance
  - but guarantees good average behavior
  - easier to understand than alternatives
  - better than others if likely to go back to recent nodes