Last Time

• Binary search trees (Ch 14)
  • Implement the OrderStructure interface
  • The `locate(root, value)` method
    • returns either: node that stores value, or parent where value would be inserted
  • Many methods use `locate`
    • `contains`
    • `add`
    • `remove`
    • ...
Today’s Outline

• Binary search trees (Ch 14)
  • Finish OrderStructure API
    • add() / remove()
  • Tree balancing to maintain small height
    • rotate()
  • Partial taxonomy of balanced tree species
    • AVL Trees
    • Splay Trees
    • Red-Black Trees
Binary Search Trees

• A binary tree is a binary search tree if it is:
  • Empty, or
  • All nodes in the left subtree are less than or equal to the root, all nodes in the right subtree are greater than or equal to the root, and the left and right subtrees are binary search trees.

• In our implementation, right subtrees only hold values that are strictly greater than the root.
  • Why?
Add: Duplicate Values

Where would a new K be added?
A new V?
First (Bad) Attempt: `add(E value)`

```java
public void add(E value) {
    BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
    if (root.isEmpty()) {
        root = newNode;
    } else {
        BinaryTree<E> insertLocation = locate(root,value);
        E nodeValue = insertLocation.value();
        if (ordering.compare(value,nodeValue) > 0) {
            insertLocation.setRight(newNode); // value > nodeValue
        } else {
            insertLocation.setLeft(newNode); // value <= nodeValue
        }
    }
    count++;
}
```

Problem: If duplicate values are allowed in the BST, the left subtree might not be empty when setLeft is called
How to Add Duplicate Values

How to perform: bst.add("v")
locate("v").setLeft(new BinaryTree("v"));
Strategy: Add Duplicates to Predecessor

- If insertLocation has a left child:
  - Find insertLocation’s predecessor, then
  - Add duplicate node as right child of predecessor
    - Why?
      - Relationships among root, pred(root), and node?

- Make duplicate values the successor of their predecessor!
public void add(E value) {
    BinaryTree<E> newNode = new BinaryTree<E>(value, EMPTY, EMPTY);
    if (root.isEmpty()) {
        root = newNode;
    } else {
        BinaryTree<E> insertLocation = locate(root, value);
        E nodeValue = insertLocation.value();
        if (ordering.compare(value, nodeValue) > 0) {
            // value > nodeValue
            insertLocation.setRight(newNode);
        } else {
            // value <= nodeValue
            if (insertLocation.left().isEmpty())
                insertLocation.setLeft(newNode);
            else
                predecessor(insertLocation).setRight(newNode);
        }
    }
    count++;
}
How to Find Predecessor?

predecessor(root) is the rightmost node in root’s left subtree
// return node with largest value in root’s left subtree
// pre: root is not empty, root’s left child is not empty
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    BinaryTree<E> result = root.left();

    while (!result.right().isEmpty())
        result = result.right();

    return result;
}
Removal

• If we can remove the root, we can remove any element in a BST in the same way
  • Why?

• We need to implement:
  • public E remove(E item)

• We can benefit from a helper:
  • protected BT removeTop(BT top)
    • removeTop(BT top) removes top, and returns the root node of the resulting tree

• Assuming removeTop works, let’s implement remove
public E remove(E value) {
    // base case 1: empty tree
    if (isEmpty()) return null;

    // base case 2: root contains value
    if (value.equals(root.value())) {
        E result = root.value();
        count--;
        root = removeTop(root);
        return result;
    }

    ...
}
BST remove()

// general case: find node that holds value, remove node, and re-attach resulting tree at node’s old location
BinaryTree<E> location = locate(root, value);
if (value.equals(location.value())) {
    // found node with value
    count--;  
    BinaryTree<E> parent = location.parent();
    if (parent.right() == location) {
        // removing right child
        parent.setRight(removeTop(location));
    } else {
        // removing left child
        parent.setLeft(removeTop(location));
    }
    return location.value();
}

// value not found in tree, nothing to do
return null;
RemoveTop(topNode)

Detach left and right sub-trees from root (i.e. topNode)
If either left or right is empty, return the other one
If left has no right child

make right the right child of left then return left

Otherwise find largest node C in left

// C is the right child of its own parent P
// C is the predecessor of right (ignoring topNode)

Detach C from P; make C’s left child the right child of P
Make C new root with left and right as its sub-trees
Case 1: No left binary tree

\[
x \rightarrow x.\text{right}
\]

\[
\text{return}
\]

\[
x.\text{right}
\]
Case 2: No right binary tree
Case 3: Left has no right subtree

```
      x
     /   \
   x.left x.right
  a.root       B
         /
        A
```

return

```
      x.left
     /       \
   a.root        x.right
       / \
      A   B
```
Case 4: General Case (HARD!)

- Consider BST requirements:
  - Left subtree must be <= root
  - Right subtree must be >= root

- Strategy: replace the root with the largest value that is less than or equal to it
  - predecessor(root) : rightmost left descendant

- This may require reattaching the predecessor’s left subtree!
Case 4: General Case (HARD!)

Replace root with predecessor(root), then patch up the remaining tree.
Case 4: General Case (HARD!)

Replace root with predecessor(root), then patch up the remaining tree
Let’s Write Some Code

- BinarySearchTree.java
But What About Height?

• Operations’ performance all depend on $h$
• Can we design a binary search tree that is always “shallow” (minimizes $h$)?
• Yes! In many ways.
• AVL trees are one example
  • Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"
• The *balance factor* of a node is the height of its right subtree minus the height of its left subtree.

• A node with balance factor 1, 0, or -1 is considered *balanced*.

• A node with any other balance factor is considered *unbalanced* and requires rebalancing the tree.
Single Rotation

Unbalanced trees can be rotated to achieve balance.
Single Right Rotation

\[
\begin{align*}
G & \quad \text{setRight} \quad -1 \\
\text{setLeft} & \quad \text{root} \\
N & \quad \text{setRight} \\
\text{setLeft} & \quad -2 \\
T1 & \quad \text{height k+1} \\
T2 & \quad \text{height k} \\
T3 & \quad \text{height k-2} \\
T4 & \quad \text{height k-1}
\end{align*}
\]
BinaryTree rotateRight()

// pre: this has a left subtree
// post: rotates local portion of tree so left child is root
protected void rotateRight() {
    // establish pointers/relationships before mucking with the tree
    BinaryTree<E> parent = parent;
    BinaryTree<E> newRoot = left();
    boolean wasChild = parent != null;
    boolean wasLeftChild = isLeftChild();

    // rotate!
    setLeft(newRoot.right()); // hook in new root
    newRoot.setRight(this); // make old root right child of new root
    if (wasChild) {
        // update parent pointers to rotated subtree
        if (wasLeftChild) parent.setLeft(newRoot);
        else parent.setRight(newRoot);
    }
}
Double Rotation
AVL Tree Facts

• A tree that is AVL except at root, where root balance factor equals ±2 can be rebalanced with at most 2 rotations

• add(v) requires at most $O(\log n)$ balance factor changes and one (single or double) rotation to restore AVL structure

• remove(v) requires at most $O(\log n)$ balance factor changes and $O(\log n)$ (single or double) rotations to restore AVL structure

• An AVL tree on $n$ nodes has height $O(\log n)$
AVL Trees have $O(\log n)$ Height

An AVL tree on $n$ nodes has height $O(\log n)$

Proof idea

- Show that an AVL tree of height $h$ has at least $\text{fib}(h)$ nodes (easy induction proof---try it!)
- Recall (HW): $\text{fib}(h) \geq \left(\frac{3}{2}\right)^h$ if $h \geq 10$
- So $n \geq \left(\frac{3}{2}\right)^h$ and thus $\log_{\frac{3}{2}} n \geq h$
  - Recall that for any $a, b > 0$, $\log_a n = \frac{\log_b n}{\log_b a}$
  - So $\log_a n$ and $\log_b n$ are Big-O of one another
- So $h$ is $O(\log n)$
AVL Trees: One of Many

• There are many strategies for tree balancing to preserve $O(\log n)$ height, including
  • AVL Trees: guaranteed $O(\log n)$ height
  • Red-black trees: guaranteed $O(\log n)$ height
  • B-trees (not binary): guaranteed $O(\log n)$ height
    • 2-3 trees, 2-3-4 trees, red-black 2-3-4 trees, ...
  • Splay trees: Amortized $O(\log n)$ time operations
  • Randomized trees: $O(\log n)$ expected height
Splay Trees

Splay trees are self-adjusting binary trees
• Each time a node is accessed, it is moved to root position via rotations
• No guarantee of balance (or shallow height)
• But good *amortized* performance

Theorem: Any set of $m$ operations (add, remove, contains, get) on an $n$-node splay tree take at most $O(m \log n)$ time.
Splay Tree Rotations

Right Zig Rotation (left version too)

Right Zig-Zig Rotation (left version too)

Right Zig-Zag Rotation (left version too)
Splay Tree Iterator

• Even contains method changes splay tree shape
• This breaks the standard in-order iterator!
  • Because the stack is based on the shape of the tree
• Solution: Remove the stack from the iterator
• Observation: Given location of current node (node whose value is next to be returned), we can compute it’s (in-order)successor in next()
  • It’s either left-most leaf of right child of current, or
  • It’s closest ”left-ancestor” of current
    • Ancestor whose left child is also an ancestor of current
• Also, reset must “re-find” root
  • Idea: Hold a single “reference” node, use it to find root
Red-Black Trees

Red-Black trees, like AVL, guarantee shallowness

• Each node is colored *red* or *black*

• Coloring satisfies these rules
  • All empty trees are black
    • We consider them to be the leaves of the tree
  • Children of red nodes are black
  • All paths from a given node to its descendent leaves have the *same number* of black nodes
    • This is called the *black height* of the tree
A Red-Black Tree
(from Wikipedia.org)
Red-Black Trees

The coloring rules lead to the following result

Proposition: No leaf has depth more than twice that of any other leaf.

This in turn can be used to show

Theorem: A Red-Black tree with \( n \) internal nodes has height satisfying

\[
 h \leq 2 \log(n + 1)
\]

Note: The tree will have exactly \( n+1 \) (empty) leaves
Red-Black Trees

Theorem: A Red-Black tree with n internal nodes has height satisfying $h \leq 2 \log(n + 1)$

Proof sketch: Note: we count empty tree nodes!

- If root is red, recolor it black.
- Now merge red children into (black) parents
  - Now $n' \leq n$ nodes and height $h' \geq h/2$
- New tree has all children with degree 2, 3, or 4
  - All leaves have depth exactly $h'$ and there are $n+1$ leaves
    - So $n + 1 \geq 2^{h'}$, so $\log_2(n + 1) \geq h' \geq \frac{h}{2}$
  - Thus $2 \log_2(n + 1) \geq h$
Red-Black Tree Insertion

Black empty leaves not drawn. 7 just added Black-height still 2.
Red-Black Tree Insertion

Black height still 2, color violation moved up
Red-Black Tree Insertion

Left rotation at 6, about to do right rotation
Red-Black Tree Insertion

Right rotation at 20, black height broken, need to recolor
Red-Black Tree Insertion

Color conditions restored, black-height restored.