CS 310: BST Removal

Chris Kauffman

Week 12-1
Easy to remove leafs
Easy to remove node with single child
Harder to remove node with two children
class BST\<T\> {  
    private Node\<T\> root;

    // Public facing method, find minimum element and return it
    public T findMin(){ return this.findMin(this.root); }

    // Private helper method return the smallest element in the
    // tree rooted at t
    private T findMin(Node\<T\> t){
        // DEFINE ME
    }

    // Public facing method, eliminate the smallest data in tree
    public void removeMin(){ this.root = removeMin(this.root); }

    // Recursive helper; remove the node with the smallest data
    // in it in the tree rooted at t. The node returned is used
    // to alter the structure of the tree.
    private Node\<T\> removeMin(Node\<T\> t){
        // DEFINE ME
    }
}
BST General \texttt{remove}(x): Cases

**Cases for** \texttt{t.remove}(x)

1. \textit{x} not in tree
   - Leave tree as is or raise an exception
2. \textit{x} at a node with no children
   - Get rid of node containing \textit{x}
3. \textit{x} at a node with 1 child
   - "Pass over" node containing \textit{x}
4. \textit{x} at a node with 2 children
   - Find a \textbf{next} node in sorting order
   - Replace \textit{x} with next nodes data
   - Remove next node
   - Next is minimum of right subtree
Children Cases for `remove(t, x)`

**One Child: Remove 5**

1. Find node \( t \) with data \( x \)
2. Replace with only child

![Diagram](a)

**Two Children: Remove 2**

1. Find node \( t \) with data \( x \)
2. Find min node of \( t.\text{right} \):
   - min must have 0/1 child
3. Replace \( t.\text{data} \) with min.data
4. Remove min

![Diagram](a)
Recursive Implementation: Think Locally

Lesson from insert()
Recall in insert(x,t), did stuff like

\[ t.\text{right} = \text{insert}(x, t.\text{right}); \]
// a new/existing node is returned by insert()

Take same approach for remove(x,t)

Assume these helpers are Available

\[
T \text{ findMin}(\text{Node}<T>\ t); \quad \text{Node}<T> \text{ removeMin}(\text{Node}<T>\ t)
\]

Implement Recursive remove(x,t)

- What to check first?
- How to know if \( t \) is the node?
- What to do if \( t \) isn’t the node?
- If \( t \) is the node, are there separate cases for action?
Cases for recursive remove()

1. ⊥ t is null
   Throw an exception
   throw new ItemNotFoundException();
   Or do nothing to the tree
   return null;

2. ⊥ x less than t.data (recurse left)
   t.left = remove(t.left, x);

3. ⊥ x greater than t.data (recurse right)
   t.right = remove(t.right, x);

4. □ x equals t.data (remove t)
   ▶ t has 0 children, get rid of t
   ▶ t has 1 child, pass over t
   ▶ t has 2 children, replace with next/prev
Case 4: $x$ equals $t.data$ (remove $t$)

Helper methods defined elsewhere

```java
T findMin(Node<T> t);  // Helper method
Node<T> removeMin(Node<T> t)
```

- $t$ has 0 children, get rid of $t$
  ```java
  return null;
  ```
- $t$ has 1 child, pass over $t$
  ```java
  (t.left!=null) ? return t.left : return t.right;
  ```
- $t$ has 2 children, replace with next or prev
  ```java
  t.data = findMin(t.right);
  t.right = removeMin(t.right);
  return t;
  ```
- How are `findMin(t)` and `removeMin(t)` implemented?
  - Where is the minimum node in a tree?
  - How many children does it have?
private static Node<T> remove(T x, Node<T> t) {
    if (t == null)
        throw new ItemNotFoundException(x.toString());
    if (x.compareTo(t.data) < 0)
        t.left = remove(x, t.left);
    else if (x.compareTo(t.data) > 0)
        t.right = remove(x, t.right);
    // Found at this node
    else if (t.left != null && t.right != null) {
        // Two children
        t.data = findMin(t.right);
        t.right = removeMin(t.right);
    }
    else
        // One child or no children
        t = (t.left != null) ? t.left : t.right;
    return t;
}
So Far

Binary Search Trees

- Defined find() / insert() / remove()
- Helpers: findMin() / findMax() / removeMin() / removeMax()
- All ops runtime complexity $O(Height)$
- Discuss balancing trees to ensure that $Height \approx \log(Size)$

Next Time

- Balanced BSTs: AVL and Red-Black Trees
- Reading: Weiss Ch. 19.4-5
Why Worry?

Q: Why worry about insert/remove messing with the tree? What affect can it have on the performance of future ops on the tree?

Q: What property of a tree dictates the runtime complexity of its operations?
Balancing Trees

- add/remove/find complexity $O(\text{height}(t))$
- Degenerate tree has height $N$: a linked list
- Prevent this by re-balancing on insert/remove
- Several kinds of trees do this
  - AVL left/right subtree height differ by max 1
  - Red-black preserve 4 red/black node properties
  - AA red-black tree + all left nodes black
  - Splay amortized bound on ops, very different
Rotations

Re-balancing usually involves

- Drill down during insert/remove
- Follow path back up to make adjustments
- Adjustments even out height of subtrees
- Adjustments are usually rotations
- Rotation changes structure of tree without affecting ordering
Single Rotation Basics

**Right Rotation**
Rotation node becomes the right subtree

![Diagram](image1.png)

(a) Before rotation  (b) After rotation

**Left Rotation**
Rotation node becomes the left subtree

![Diagram](image2.png)

(a) After rotation  (b) Before rotation
Fixing an Insertion with a Single Rotation

Insert 1, perform rotation to balance heights
- Right rotation at 8

(a) Before rotation

(b) After rotation
Single Rotation Practice

Problem 1
- 40 was just inserted
- Rebalance the tree rooted at 16
- Use a single rotation on 16

Problem 2
- 56 is being removed
- Rebalance the tree rooted at 74
- Use a single rotation on 74
Single Rotations Aren’t Enough

Can we fix the following with a single rotation?

(a) Before rotation

(b) After rotation
Double Rotation Overview

**Left-Right**
- Left Rotate at $k_1$
- Right Rotate at $k_3$

**Right-Left**
- Right Rotate at $k_3$
- Left Rotate at $k_2$
Fixing an Insertion with a Double Rotation

Insert 5, perform rotation to balance heights

(a) Before rotation

(b) After rotation
Problem 3

- 35 was just inserted
- Rebalance the tree rooted at 36
- Use two rotations, at 33 and 36
- 36 should move
Practice/Demo Sites

- jGrasp is so-so for seeing tree operations
- Play with Blanced Binary Search Trees online using the following applets (titles hyperlinked)

Qmatica: Pretty AVL
- Standard BSTs
  - Manual Rotation
  - Great Practice
- AVL Trees
- Undo/Redo to rewatch
- Step by step logging

Adjustable Demo
- Standard BSTs
- All three Balanced
  - AVL, Red black, Splay
- Slow down, pause, show balance factors

Scaling AVL
- AVL Tree only
- Scaling view for large trees

Multi Demo
- AVL, Red-Black, B-tree, ...
Code

weiss/nonstandard/Rotations.java contains

1. static <T> BinaryNode<T> rotateWithLeftChild( BinaryNode<T> k2 )
2. static <T> BinaryNode<T> rotateWithRightChild( BinaryNode<T> k1 )
3. static <T> BinaryNode<T> doubleRotateWithLeftChild( BinaryNode<T> k3 )
4. static <T> BinaryNode<T> doubleRotateWithRightChild( BinaryNode<T> k1 )

- 1 and 2 are 4 lines
- 3 and 4 are 2 lines (call 1 and 2)
- Write them
- What is the runtime complexity of each rotation?