

# Lecture 19: Binary Search & Splay Trees

CS 62  
Spring 2012  
Kim Bruce & Kevin Coogan

## Exam Monday

- In class: 50 minutes
- Covers everything through Splay trees
- Studying essential
  - Do form study groups
  - Do problems from sample exams
  - Do problems from text
  - Review session Friday in office hours 2 - 3 p.m.
- Quiz Friday?

## BST

- 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.

## Implementation

- Focus on trickiest methods:
  - add, get, & remove
  - protected methods: locate, predecessor, and removeTop

```

// @pre root and value are non-null
// @post returned: 1 - existing tree node with the desired value, or
//                2 - the node to which value should be added
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);
    }
}

```

```

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;
}

```

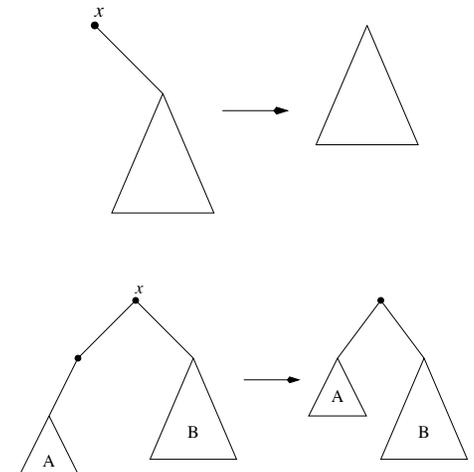
```

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++;
}

```

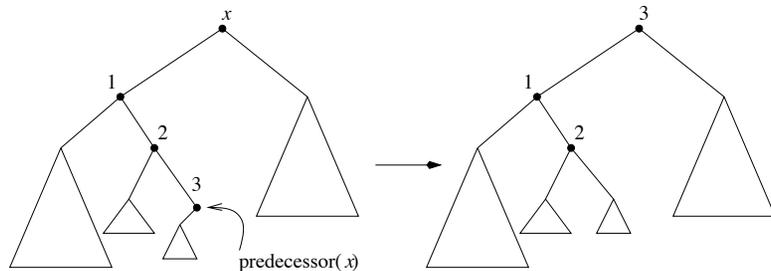
## Remove node

- Remove topmost node.
- Easy cases:
  - no left subtree, or no right subtree -- easy, they are new tree
  - left child has no right subtree



## General Case

- Left Child has a right subtree:



## Remove method

- Locate element to be deleted
- RemoveTop of node rooted at element
- Hook up resulting tree as child of elt's parent.
- $O(h)$ , where  $h$  is height of tree.
  - $O(h)$  to find,
  - Could be another  $O(h)$  to find predecessor
  - Constant to patch back together.

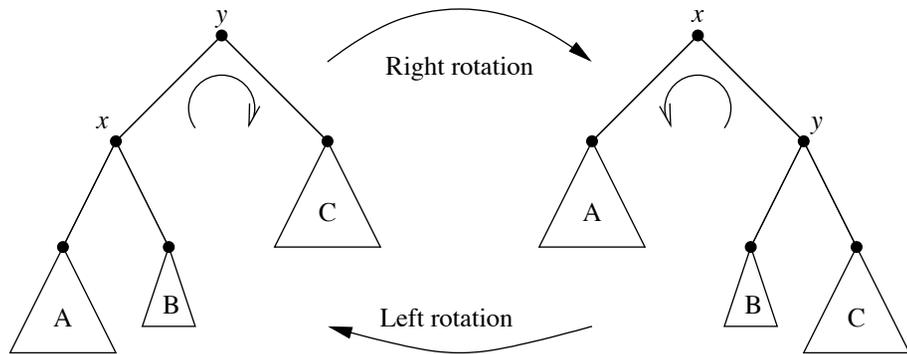
## Complexity

- Add, get, contains, remove
  - Proportional to height of tree
- Can we guarantee  $O(\log n)$ ?
  - Only if we can keep them balanced!!
  - Special 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

## Splay Trees

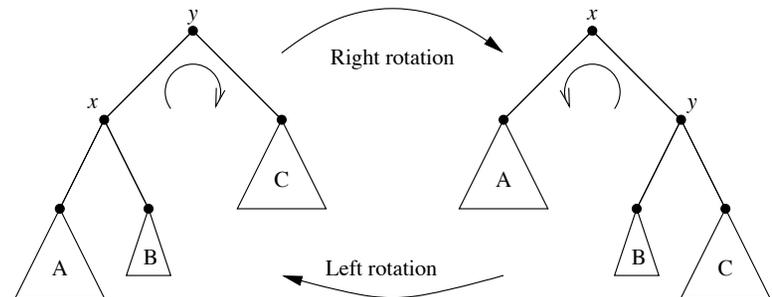
# Rotating Trees

- Key idea: Rotate node higher in tree while keeping it in order.



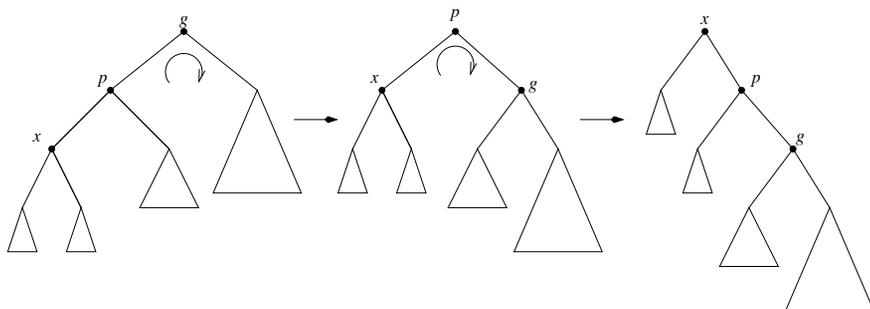
# Rotating Trees

- Rotate x to root, while maintain BST structure
  - All nodes in subtree A go up one level, all in C go down one level, all in B stay same.
  - See code in BinaryTree



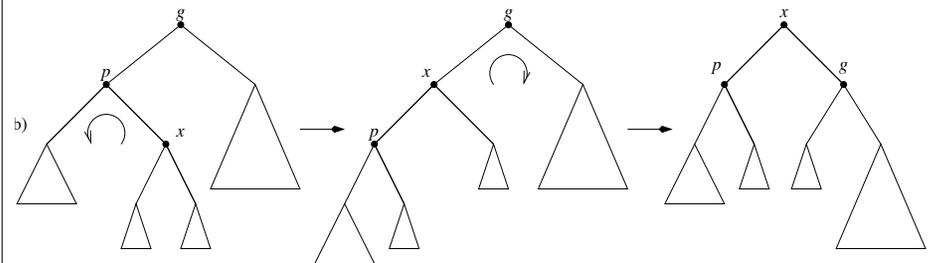
# Shifting elements toward root

- Move x up two levels w/ two rotations
- If x is left child of a left child ...



# Shifting elements toward root

- If x is a right child of a left child.



*Symmetric if interchange left and right.*

## Splay Tree

- Idea behind splay tree.
  - Every time find, get, add: or remove an element x, move it to the root by a series of rotations.
  - Other elements rotate out of way while maintaining order.
- Splay means to spread outwards

## How to Splay in Words

- if x is root, done.
- if x is left (or right) child of root,
  - rotate it to the root
- if x is left child of p, which is left child of g,
  - do right rotation about g and then about p to get x to grandparent position. Continue splaying until at root.
- if x is right child of p, which is left child of g,
  - rotate left about p and then right about g. Continue splaying until at root.

*Results in moving node to root!*

## Splay Tree

- Modify tree operations:
  - When do add, contains, or get, splay the elt.
  - When remove an elt, splay its parent.
- Average depth of nodes on path to root cut in half on average!
- If repeatedly look for same elements, then rise to top -- and found faster!
- Splay code is ugly -- but follows ideas given

## Example of modified operation

```
public boolean contains(E val) {
    if (root.isEmpty()) return false;

    BinaryTree<E> possibleLocation = locate(root,val);
    if (val.equals(possibleLocation.value())) {
        splay(root = possibleLocation);
        return true;
    } else {
        return false;
    }
}
```

# General Trees

## Representing general trees

- If fixed number, not a problem.
- What if variable?
  - Keep list of children