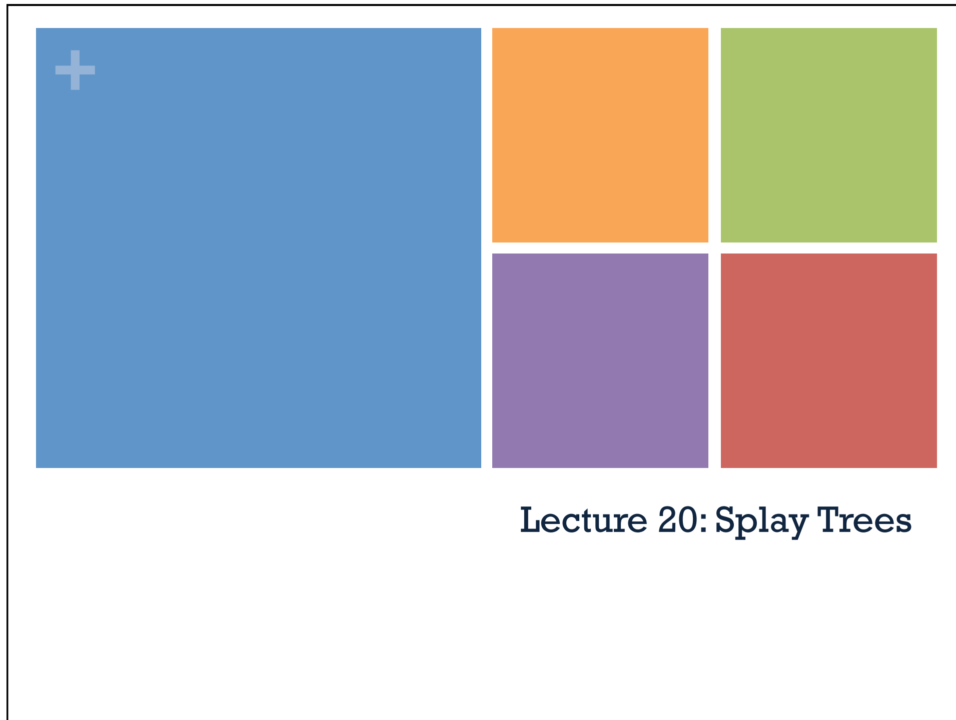
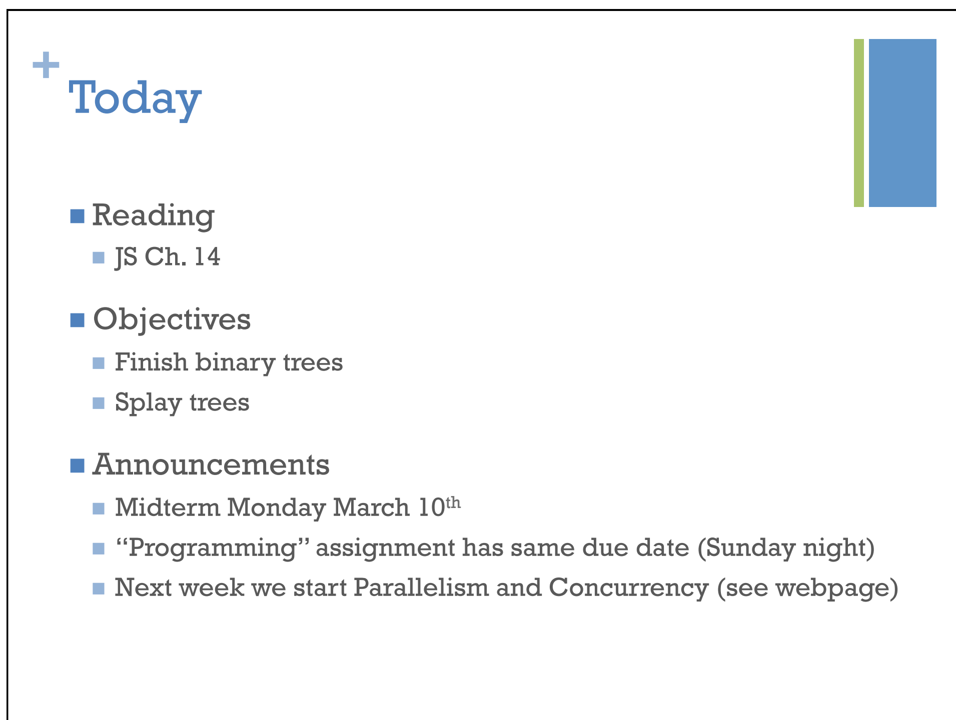

Lecture 20: Splay Trees

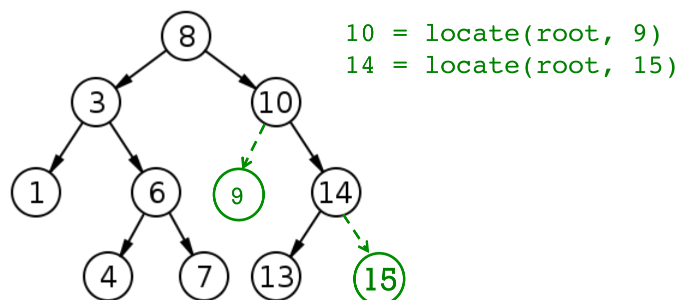

Today
Reading
JS Ch. 14
Objectives
Finish binary trees
Splay trees
Announcements
Midterm Monday March 10th
"Programming" assignment has same due date (Sunday night)
Next week we start Parallelism and Concurrency (see webpage)

Recap: Locating a value in a BST

```
protected BinaryTree<E> locate(BinaryTree<E> node, E value) {
    E nodeValue = node.value();
    BinaryTree<E> child;

    // If node's value equals value, we're done
    if (nodeValue.equals(value))
        return node;

    // Look left if less than, right if greater
    if (ordering.compare(nodeValue,value) < 0) {
        child = node.right();
    } else {
        child = node.left();
    }

    // If no child, return node. Else keep searching
    if (child.isEmpty()) {
        return node;
    } else {
        return locate(child, value);
    }
}
```

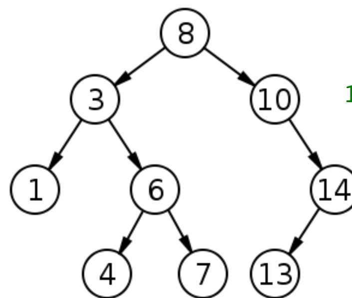
Recap: Using locate to add a node

- 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



+ Recap: Using locate to add a node

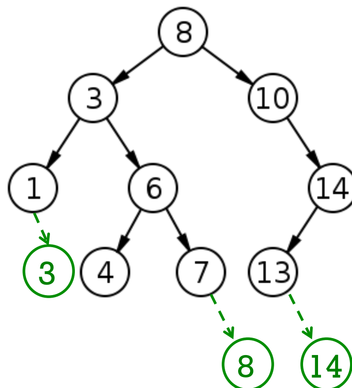
- Case Two: Locate returns pointer to node with same value
  - Duplicates go in left subtree (could have chosen right)
  - Where in the left subtree?




```
3 = locate(root, 3)
8 = locate(root, 8)
14 = locate(root, 14)
```

+ Recap: 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 descendent*



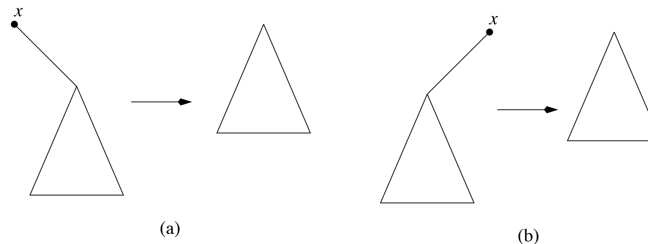

```
myBST.add(3);
myBST.add(8);
myBST.add(14);
```

Recap: Using locate to add a node

```
public void add(E value) {
    BinaryTree<E> newNode = new BinaryTree<E>(value);

    // If no root, make new node root
    if (root.isEmpty()) {
        root = newNode;
    } else {

        // Find where new node should go
        BinaryTree<E> insertLocation = locate(root,value);
        E nodeValue = insertLocation.value();

        if (ordering.compare(nodeValue,value) < 0) {
               insertLocation.setRight(newNode);   // case one
           }
        else {
            if (!insertLocation.left().isEmpty()) { // case two
                predecessor(insertLocation).setRight(newNode);
            } else {
                insertLocation.setLeft(newNode); // case one
            }
        }
    }
    count++;
}
```

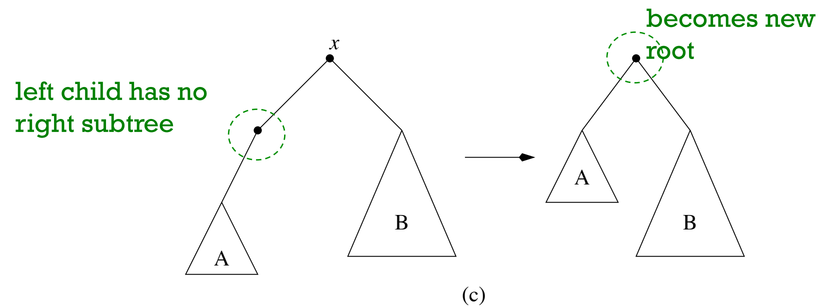
Remove a node

- Calling `remove(E val)` removes node with value val
- Case One:
  - Node to be removed has no left subtree or no right subtree



Remove a node

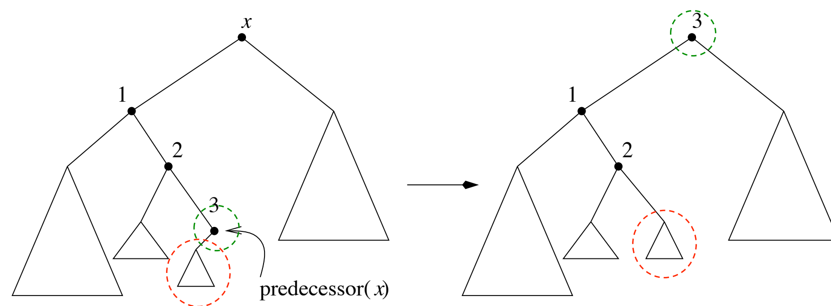
- Calling `remove(E val)` removes node with value val
- Case Two:
  - Node to be removed has both left and right subtree but its left child has no right subtree



(c)

Remove a node

- Calling `remove(E val)` removes node with value val
- Case Three:
  - Predecessor of root node becomes new root



+ Remove a node

- To remove a node
  - Locate the node to be removed
  - Remove node
  - Depending on case, reset pointers (may require finding predecessor)
- Complexity is O(h) where h is height of tree
  - Worst-case O(h) to locate
  - Worst-case O(h) to find predecessor
- Recall that $\log_2 N \leq h \leq N$

+ Complexity

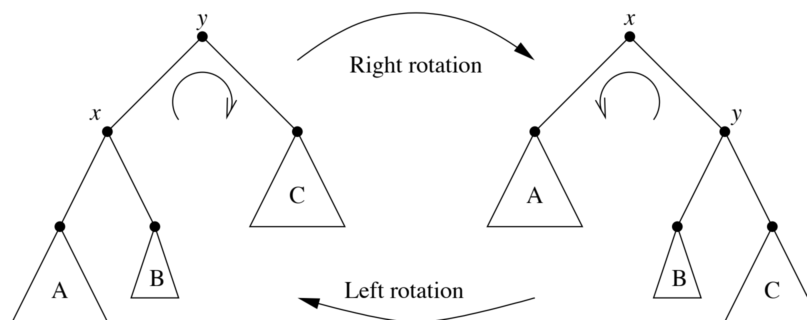
- locate, predecessor, add, contains, remove are all O(h)
- Can we guarantee that h is $O(\log_2 n)$?
  - Only if tree stays balanced!!
- Binary search trees that stay balanced
  - AVL trees (1962)
  - Red-black trees (1972)
- Splay trees (1985) don't necessarily stay balanced but provide fast access for repeated calls

Splay Trees

- To splay (v.): to spread out and apart
- A splay tree is a binary search tree that rearranges its nodes via *splaying* in order to provide faster access to recently accessed elements
- Splaying moves a node to the root of the tree
- Splaying uses two fundamental operations: right and left rotations

Right and Left Rotations

- Key idea: Rotate a node higher in tree while maintaining binary search tree property

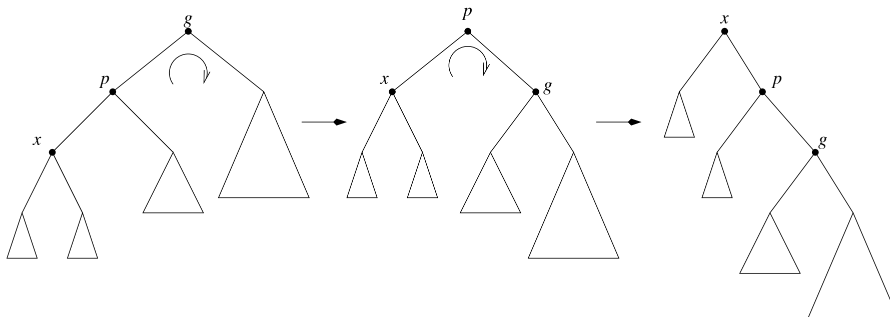


Splaying

- Case One: x is the root
  - Done!
- Case Two: x is the left or right child of the root
  - Left or right rotate. Done!
- Case Three: x is the left child of a left child (or right of right)
  - Right rotation of grandparent. Right rotation of parent. Continue splaying
- Case Four: x is the right child of a left child (or left of right)
  - Left rotation of parent. Right rotation of grandparent. Continue splaying

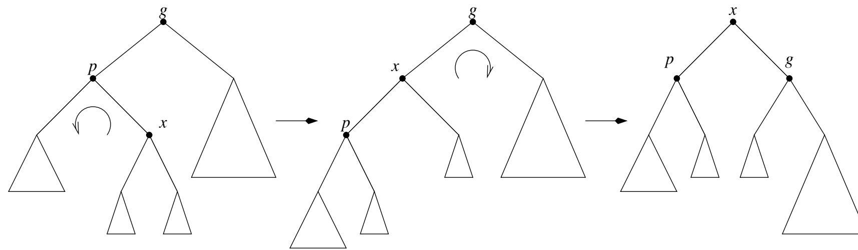
Case Three

- x is the left child of a left child (or right of right)



Case Four

- x is the right child of a left child (or left of right)



Splay Trees

- When do you splay?
  - When call add, contains, or get method – splay on element
  - When call remove – splay on element's parent
- Depth of nodes on original path from x to root is halved on average
- If repeatedly look for same elements, then rise to top -- and found faster!
- Splay code is non-trivial but follows ideas given

Example of modified operation

```
public boolean contains(E val)  {
  // If empty tree return false
  if(root.isEmpty()) { return false; }

  // Locate node
  BinaryTree<E> possibleLocation = locate(root,val);

  // If the value is in the tree, take the
  // opportunity to splay
  if (val.equals(possibleLocation.value())){
      splay(possibleLocation);
      root = possibleLocation;
      return true;
  } else {
      return false;
}
```

contains changes the tree…