+

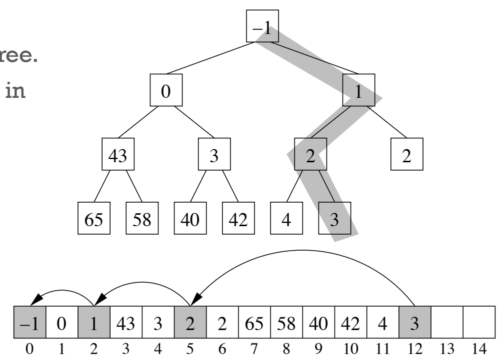
Lecture 19:
Binary Search Trees

+ Today

- Reading
  - JS Ch. 14 (Binary search trees and Splay trees)
- Objectives
  - Binary search trees
- Announcements
  - Midterm is Monday March 10th
  - No quiz on Friday

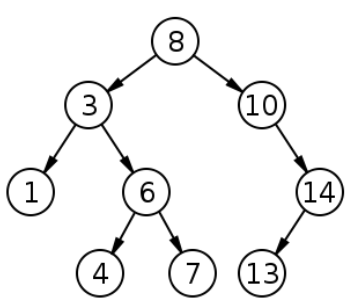
Recap: Adding to a Heap

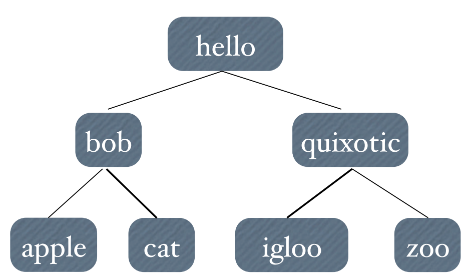
- Pre-condition
  - Heap is a complete binary tree.
  - Values along every path are in ascending order
- Adding
  - Add value to end of `data[]`
  - Percolate upward
- Complexity?



Binary Search Tree (BST)

- A binary search tree is a binary tree such that for every node n:
  - n is greater than or equal to each value in its left subtree
  - n is less than or equal to each value in its right subtree





BST Implementation

```
public class BinarySearchTree<E extends Comparable<E>> {
     protected BinaryTree<E> root;       // root of tree
     protected Comparator<E> ordering;   // comparator

     // public methods
     public void add(E value){...}
     public E contains(E value){...}
     public E remove(E value){...}

     // helper methods
     protected BinaryTree<E> locate(BinaryTree<E> node, E val)
     protected BinaryTree<E> predecessor(BinaryTree<E> node)
     protected BinaryTree<E> removeTop(BinaryTree<E> topNode)
}
```

Locating a value in a BST

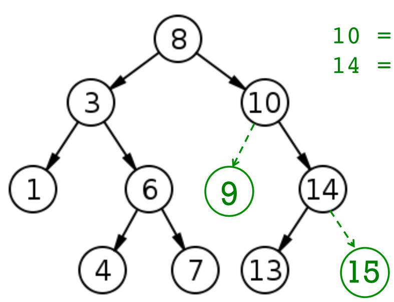
- Useful helper method for `add`, `contains`, and `remove`
- Returns pointer to the node or pointer to where the node should be added
- Recursive implementation (could have done iterative implementation)

Locating a value in a BST

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

      // If node 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);
      }
}
```

Using locate to add a node

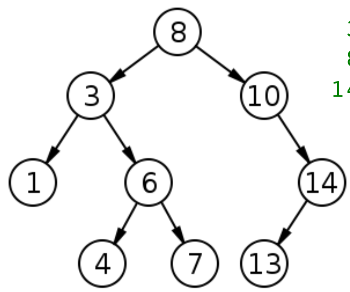
- Case One: Locate returns pointer to where node should be added
  - If value less than returned node, create new left child
  - If value greater than returned node, create new right child




```
10 = locate(root, 9)
14 = locate(root, 15)
```

Using locate to add a node

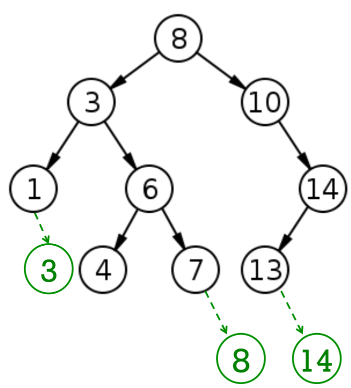
- Case Two: Locate returns pointer to node with same value
  - Duplicates go in left subtree (could have chosen right)
  - Where in the left subtree?




```
 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 descendent*

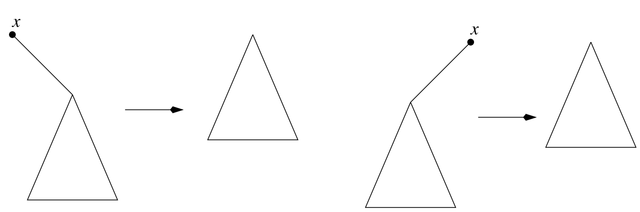


Using locate to add a node

```
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

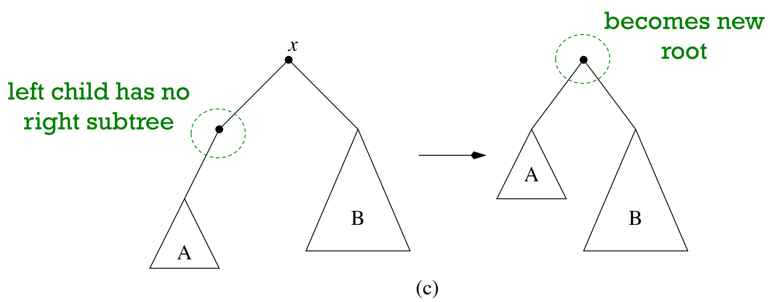
- Calling `remove(E val)` removes node with value val
- Case One:
  - Node to be removed has no left subtree or no right subtree



(a) (b)

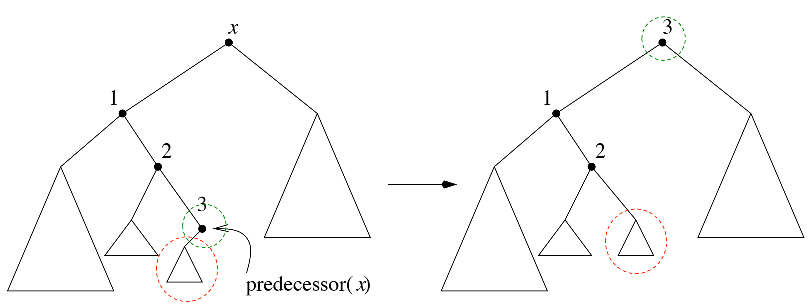
Remove a node

- 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



Remove a node

- Calling `remove(E val)` removes node with value val
- Case Three:
  - Left subtree has right child
  - 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, 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
  - Red-black trees
  - Splay trees