CS 61B Discussion 9 Spring 2015

1 2-3-4 Tree Insertion and Deletion

Given the following 2-3 tree, draw what the tree would look like after inserting 18.

```
    Original tree:            After inserting 18:
       [8]                         [8]
     /     \                      /     \
   [6]     [14]                [6]    [14, 16]
  /   \   /     \             /  \    /    |    \
 [3] [7] [10]  [15, 16]     [3]  [7] [10] [15] [18]
```

2 BSTs and Balance

Given the following binary trees, determine if each is a BST, and whether it has minimum-BST-height (circle the correct answer). By minimum-BST-height, we mean that the height of the tree is the same as the height of the optimal binary search tree containing the given elements.

```
         6                        10                    9
       /   \                    /    \                /   \
      5     9                  6      12             7       11
    /     /   \              /       /             /   \       \
   2     7     10           3       11            5      8       13
     \                    / \                   / \
       3                 1   7                 3   8
         \
          4

    Valid:  TRUE             Valid:  FALSE         Valid:  FALSE
 Balanced:  FALSE         Balanced:  FALSE      Balanced:  TRUE
```

Suppose we know the height H and number of nodes N of a BST. Can we determine whether or not this BST is minimum-BST-height without having to check the values of each node? If so, how? If not, why not?

Check that `h = floor(log(n))` where `h` is the height of the tree and `n` is the number of nodes.

3 Binary Tree Creation

Implement a function that, given a **sorted** array of integers, creates and returns a maximally balanced Binary Search Tree. You can assume you have a method slice that takes in an integer array and two indices to slice between (inclusive of the first index): [needs updating]

```
slice([1, 2, 3], 0, 1) -> [1]
slice([1, 2, 3], 1, 3) -> [2, 3]
```

Use the following definition of a Binary Search Tree Node (BSTNode):

```
public class BSTNode {
    public BSTNode left, right;
    public int value;

    public BSTNode(int n) {
        value = n;
    }
}

public BSTNode makeBST(int[] nums) {
    if (nums.length == 0) return null;

    int mid = nums.length / 2;
    BSTNode result = new BSTNode(nums[mid]);

    result.left = makeBST(slice(nums, 0, mid));
    result.right = makeBST(slice(nums, mid + 1, nums.length));

    return result;
}
```

Runtime recap: What is the runtime of `makeBST()`?

O(n)

4 Common Ancestor

Challenge Problem: Implement a function that, given a valid BST and two integers, returns the `BSTNode` X that is the deepest common ancestor of the two integers. By deepest, we mean that its distance from the root is maximized. By common ancestor, we mean that `n1 <= X.val` and `n2 >= X.val`. You may assume that `n1 < n2`. If no such node exists, return null.

```
public BSTNode commonAncestor(BSTNode root, int n1, int n2) {
        if (n1 > n2)
            return null;
        if ((n1 <= root.val) && (n2 >= root.val))
            return root;
        if (n2 < root.val)
            return commonAncestor(root.left, n1, n2);
        if (n1 > root.val)
            return commonAncestor(root.right, n1, n2);
        return null;
}
```

Runtime recap: What is the runtime of `commonAncestor()`?

O(log n) on a balanced tree, but **O(n)** in general.