CS 61B Spring 2018

Disjoint Sets, Trees, Hashing

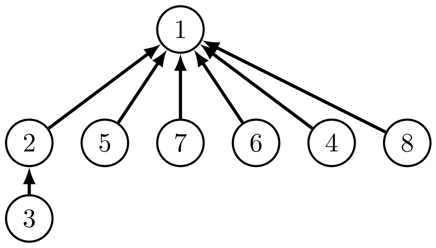
Discussion 9: March 12, 2018

1 WQU and Path Compression

Assume we have eight sets, represented by integers 1 through 8, that start off as completely disjoint sets. Draw the WQU Tree after the series of `union()` and `find()` operations with path compression. Write down the result of `find()` operations. Break ties by choosing the smaller integer to be the root.

```
union(2, 3);
union(1, 6);
union(5, 7);
union(8, 4);
union(7, 2);
find(3);
union(6, 4);
union(6, 3);
find(7);
find(8);
```

`find()` returns 2, 1, 1 respectively



2 Is This a BST?

The following method `isBSTBad` is supposed check if a given binary tree is a BST, though for some binary trees, it is returning the wrong answer. Think about an example of a binary tree for which `isBSTBad` fails. Then, write `isBSTGood` so that it returns the correct answer for any binary tree. The `TreeNode` class is defined as follows:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
}
```

Hint: You will find `Integer.MIN_VALUE` and `Integer.MAX_VALUE` helpful when writing `isBSTGood`.

An example of a binary tree for which the method fails:

```
      10
     /  \
    5    15
   / \
  3   12
```

The method fails for some binary trees that are not BSTs since it only checks that the value at a node is greater than its left child and less than its right child, not that its value is greater than every node in the left subtree and less than every node in the right subtree. Above is one example of a tree for which it fails.

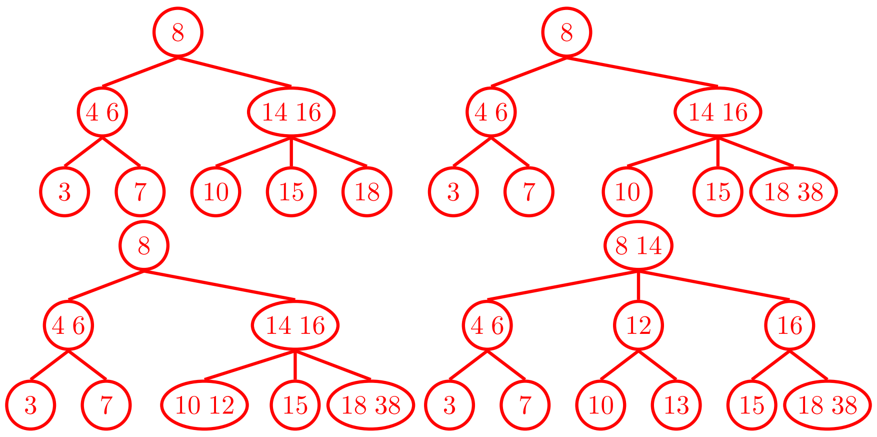
By the way, the method does return true for every binary tree that actually is a BST.

Below is the correct code:

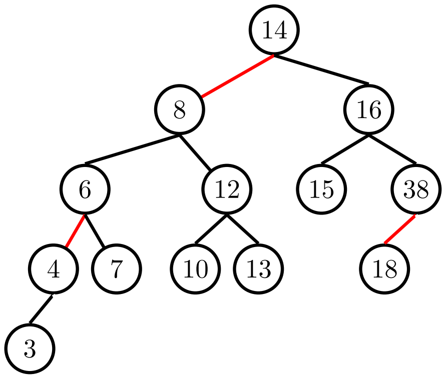
```
public static boolean isBSTGood(TreeNode T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(TreeNode T, int min, int max) {
    if (T == null) {
        return true;
    } else if (T.val < min || T.val > max) {
        return false;
    } else {
        return isBST(T.left, min, T.val) && isBST(T.right, T.val, max);
    }
}
```

3 2-3 Trees and LLRB's

3.1 Draw what the following 2-3 tree would look like after inserting 18, 38, 12, and 13.



3.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



3.3 *Extra:* If a 2-3 tree has depth `H` (that is, the leaves are at distance `H` from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?

2h comparisons. The maximum number of comparisons occur from a root to leaf path with the most nodes. Because the height of the tree is h, we know that there is a path down the leaf-leaning red-black tree that consists of at most h black links, for black links in the left-leaning red-black tree are the links that add to the height of the corresponding 2-3 tree. In the worst case, in the 2-3 tree representation, this path can consist entirely of nodes with two items, meaning in the left-leaning red-black tree representation, each blank link is followed by a red link. This doubles the amount of nodes on this path from the root to the leaf. This example will represent our longest path, which is 2h nodes long, meaning we make at most 2h comparisons in the left-leaning red-black tree.

4 Hashing

4.1 Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage.

```
public int hashCode() {
    return -1;
}
```

Valid. As required, this hash function returns the same `hashCode` for Integers that are `equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent (collisions occur 100% of the time).

```
public int hashCode() {
    return intValue() * intValue();
}
```

Valid. Similar to (a), this hash function returns the same `hashCode` for integers that are `equals()`. However, integers that share the same absolute values will collide (for example, $x = 5$ and $x = -5$ will have the same hash code). A better hash function would be to just return the `intValue()` itself.

```
public int hashCode() {
    return super.hashCode();
}
```

Invalid. This is not a valid hash function because integers that are `equals()` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the integer object's location in memory.

4.2 *Extra, but highly recommended:* For each of the following questions, answer **Always**, **Sometimes**, or **Never**.

(a) When you modify a key that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

Sometimes. If the `hashCode` for the key happens to change as a result of the modification, then we won't be able to reliably retrieve the key.

(b) When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

Always. The bucket index for an entry in a `HashMap` is decided by the key, not the value. Mutating the value does not affect the lookup procedure.