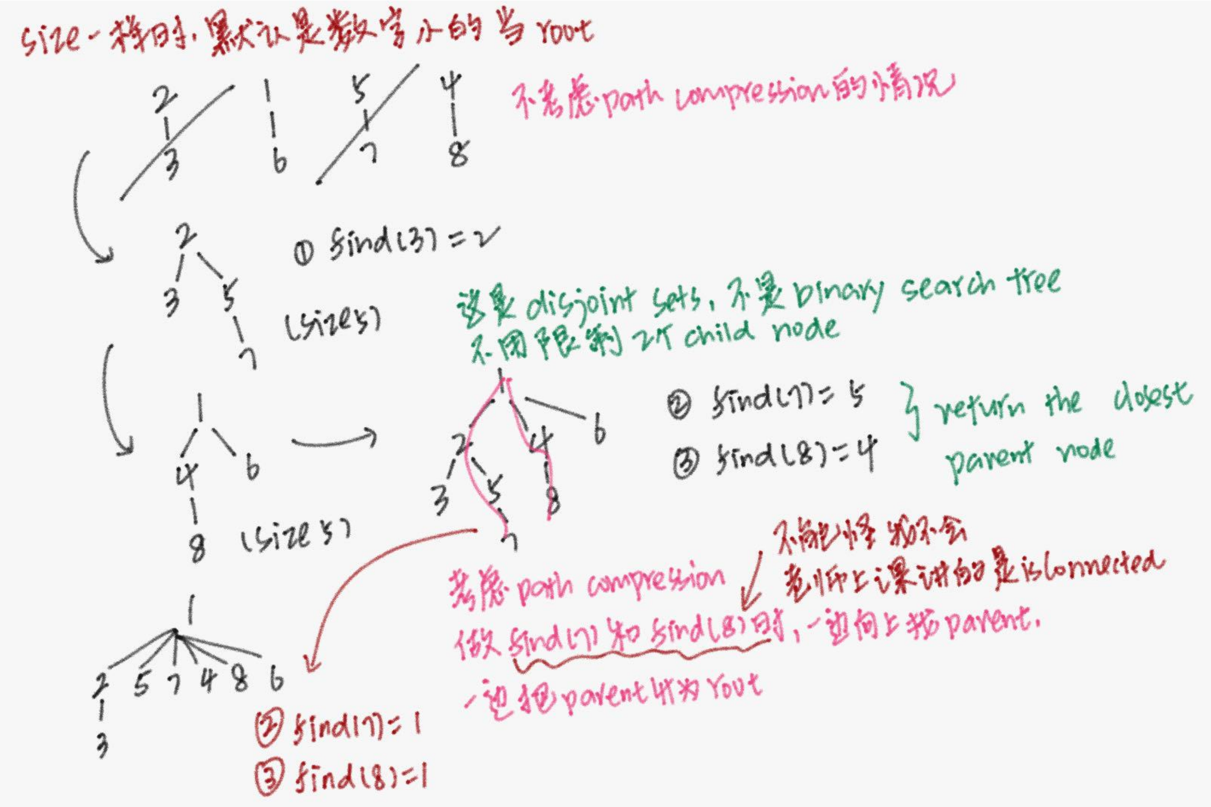
CS 61B Spring 2018 — Disjoint Sets, Trees, Hashing

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);
```



size一样时，默认是数字小的当root

不考虑path compression的情况

① find(3)=2

(size 5)

这是disjoint sets，不是binary search tree
不用限制2个child node

(size 5)

② find(7)=5
③ find(8)=4
} return the closest parent node

不确定对不对
老师上课讲的是isConnected

考虑path compression
做find(7)和find(8)时，一边向上找parent，
一边把parent改为root

② find(7)=1
③ find(8)=1

2 Is This a BST?

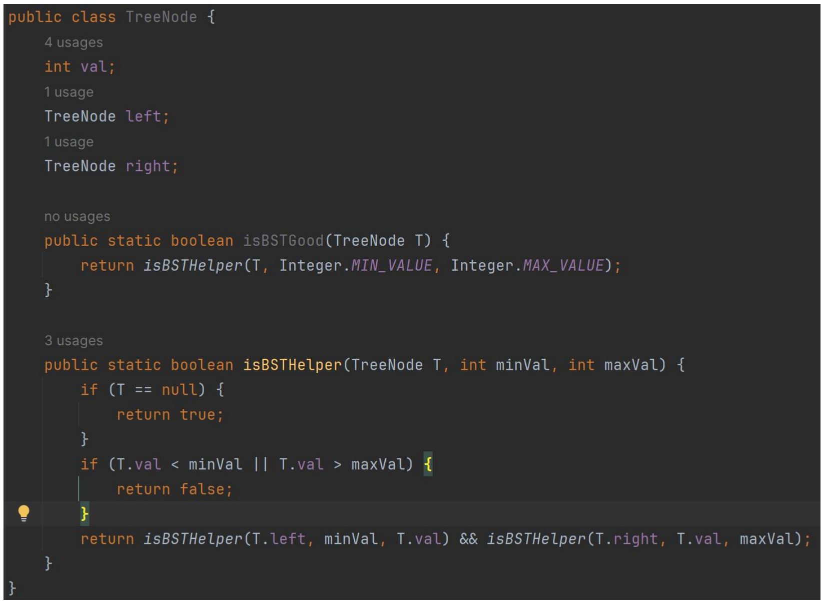
The following method isBSTBad is supposed check if a given binary tree is a BST, though for some binary trees, it is returning the wrong answer. Think about an example of a binary tree for which isBSTBad fails. Then, write isBSTGood so that it returns the correct answer for any binary tree. The TreeNode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

it only checks compares the neighbouring nodes, and cannot guarantee that all nodes in the sub-tree satisfy the requirement

```
public static boolean isBSTBad(TreeNode T) {
    if (T == null) {
        return true;
    } else if (T.left != null && T.left.val > T.val) {
        return false;
    } else if (T.right != null && T.right.val < T.val) {
        return false;
    } else {
        return isBSTBad(T.left) && isBSTBad(T.right);
    }
}

public static boolean isBSTGood(TreeNode T) {
    return isBSTHelper(                                              );
}

public static boolean isBSTHelper(                                   ) {
```

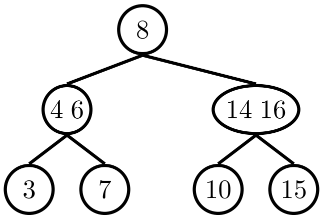


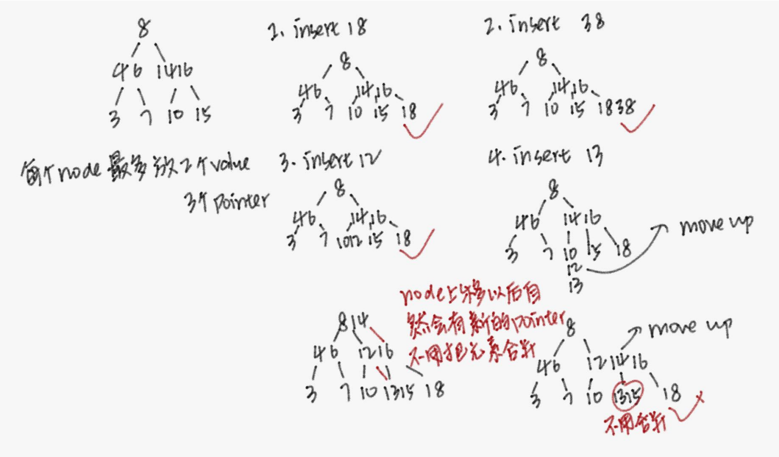
```
public class TreeNode {
    4 usages
    int val;
    1 usage
    TreeNode left;
    1 usage
    TreeNode right;

    no usages
    public static boolean isBSTGood(TreeNode T) {
        return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    3 usages
    public static boolean isBSTHelper(TreeNode T, int minVal, int maxVal) {
        if (T == null) {
            return true;
        }
        if (T.val < minVal || T.val > maxVal) {
            return false;
        }
        return isBSTHelper(T.left, minVal, T.val) && isBSTHelper(T.right, T.val, maxVal);
    }
}
```

```
}
```

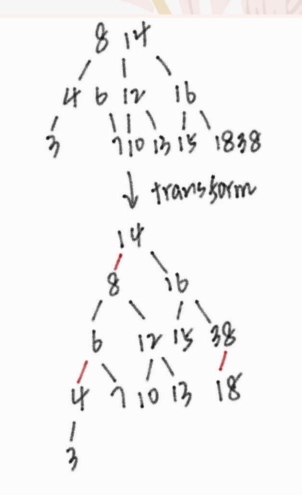
3 2-3 Trees and LLRB's

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

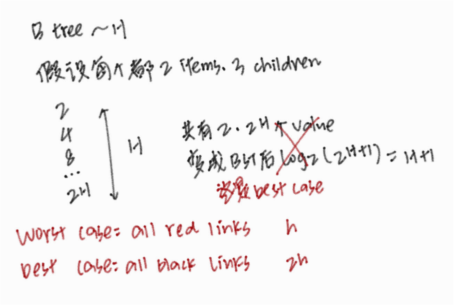




3.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



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?



B tree ~H

假设每个都2 items. 3 children

2
4
8
...
2^H

H

共有$2 \cdot 2^H$~~个value~~

~~变成BST后log2~~$(2H+1)$ = H+1

忽略best case

worst case: all red links h

best case: all black links 2h

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;
}
```

all elements are clustered
at the same block -> valid

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

absolute values -> bad
but valid hashing

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

valid but bad
all parent classes and its child
classes have the same hashcode

this hashcode is not valid, because we are currently talking about Integer class, which has its own overriden . equals() method (which compares the integer value instead of memory address), and its own .hashCode method (which also do computation on its integer value); however, by using super.hashCode, we are ignoring this overriden hashCode, and now it conflicts with .equals

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 hashing of the updated version happens to be the same as before

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

Always, because the key and the hashing code didn't change.