CS 61B Spring 2026

BSTs, 2-3 Trees, LLRBs

Exam-Level Discussion 08: March 9, 2026

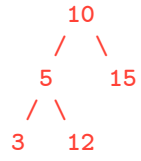
1 Is this a BST?

In this setup, assume a **BST** (Binary Search Tree) has a **key** (the value of the tree root represented as an **int**) and pointers to two other child BSTs, **left** and **right**. Additionally, assume that **key** is between **Integer.MIN_VALUE** and **Integer.MAX_VALUE** non-inclusive.

(a) The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which **brokenIsBST** fails.

```
public static boolean brokenIsBST(BST tree) {
    if (tree == null) {
        return true;
    } else if (tree.left != null && tree.left.key >= tree.key) {
        return false;
    } else if (tree.right != null && tree.right.key <= tree.key) {
        return false;
    } else {
        return brokenIsBST(tree.left) && brokenIsBST(tree.right);
    }
}
```

Here is an example of a binary tree for which **brokenIsBST** fails:



The method fails for some binary trees that are not BSTs because 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 an example of a tree for which it fails.

It is important to note that the method does indeed return true for every binary tree that actually is a BST (it correctly identifies proper BSTs).

(b) Now, write **isBST** that fixes the error encountered in part (a).

Hint: You will find **Integer.MIN_VALUE** and **Integer.MAX_VALUE** helpful.

Hint 2: You want to somehow store information about the keys from previous layers, not just the direct parent and children. How do you use the parameters given to do this?

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

public static boolean isBSTHelper(BST T, int min, int max) {

    if (T == null) {
        return true;
    } else if (T.key <= min || T.key >= max) {
        return false;
    } else {
        isBSTHelper(T.left, min, T.key) && isBSTHelper(T.right, T.key, max);
    }
}
```

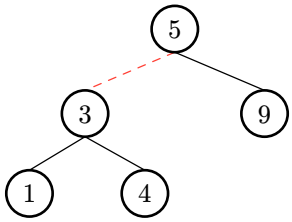
Solution: A BST is a naturally recursive structure, so it makes sense to use a recursive helper to go through the BST and ensure it is valid. Specifically, our recursive helper will traverse the BST while tracking the minimum and maximum valid values for subsequent nodes along our current path. We can get these minimum and maximum values by remembering the key property of BSTs: nodes to the left of our current node are always less than the current value, and nodes to the right of our current node are always greater than our current value. So for example, if we encounter a node with value 5, anything to the left must be < 5.

In our base case, an empty BST is always valid. Otherwise, we can check the current node. If it doesn't fall within our precomputed min/max, we know this is invalid, and return immediately.

Otherwise, we use the properties of BSTs to bound our subsequent min and max values. If we traverse to the left, everything must be less than or equal to the current value, so the value of our current node becomes the new `max` for the tree at `T.left`. Similar logic applies to the right.

2 LLRB Insertions

Given the LLRB below, perform the following insertions and draw the final state of the LLRB. In addition, for each insertion, write the balancing operations needed in the correct order (**`rotateRight`**, **`rotateLeft`**, or **`colorFlip`**). If no balancing operations are needed, write "Nothing". Assume that the link between 5 and 3 is red and all other links are black at the start.



(a) 1. Insert 7

2. Insert 6

3. Insert 2

4. Insert 8

5. Insert 8.5

6. Final state

Solution: For a visualization of the process, see here

1. Insert 7:
   - Nothing

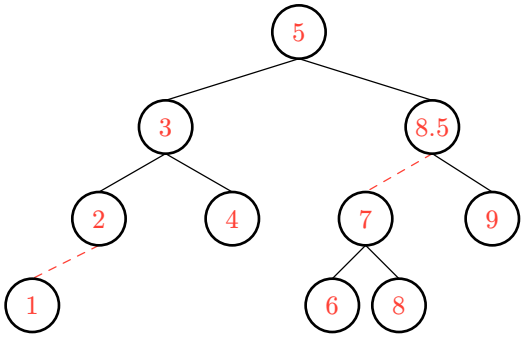
2. Insert 6:
   - rotateRight(9)
   - colorFlip(7)
   - colorFlip(5)

3. Insert 2:
   - rotateLeft(1)

4. Insert 8:
   - Nothing

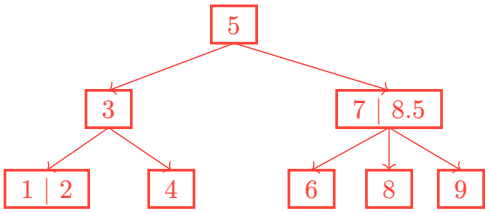
5. Insert 8.5:
   - rotateLeft(8)
   - rotateRight(9)
   - colorFlip(8.5)
   - rotateLeft(7)

6. Final state



(b) Convert the final LLRB to its corresponding 2-3 Tree.

Solution:

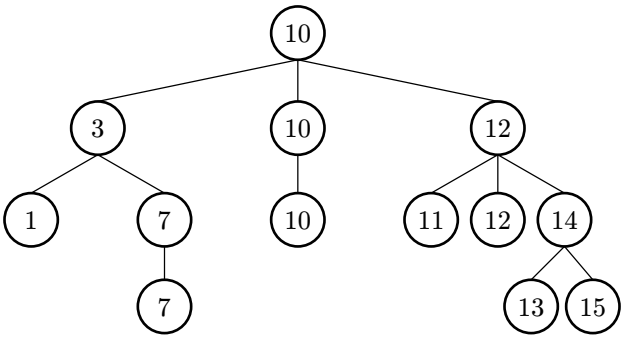


3 Tri-nary Search Tree

We'd like a data structure that acts like a BST (Binary Search Tree) in terms of operation runtimes but allows duplicate values. Therefore, we decide to create a new data structure called a TST (Trinary Search Tree), which can have up to three children, which we'll refer to as `left`, `middle`, and `right`. In this setup, we have the following invariants, which are very similar to the BST invariants:

1. Each node in a TST is a root of a smaller TST
2. Every node to the `left` of a root has a value "lesser than" that of the root
3. Every node to the `right` of a root has a value "greater than" that of the root
4. **Every node to the `middle` of a root has a value equal to that of the root**

Below is an example TST to help with visualization.



Describe an algorithm that will print the elements in a TST in **descending** order. *(Hint: recall that an in-order traversal for a BST gives elements in increasing order.)*

Solution:

Inorder traversal on a BST yields the sorted elements in the BST in ascending order. Therefore, the core of the algorithm we'd like here is going to be quite similar to inorder traversal, but reversed (visit the right child before the left child) and with the added caveat that we also must traverse through the middle children.

In essence, given the root of some TST, we reverse onto the right child subtree, then print the root's value, then reverse onto the middle child subtree, then finally reverse onto the left subtree. The print root value and reverse onto the middle child steps can be swapped, because overall the order of the printed values should be the same.

Pseudocode:

```
reverse(tst):
    if tst is null:
        return
    reverse(tst.right)
    print(tst.value)
    reverse(tst.middle)
    reverse(tst.left)
```