

Trees Homework: Third Question

Algorithm 1:

1.

```
// Function to determine whether a given binary tree is a BST by keeping a
// valid range (starting from [-INFINITY, INFINITY]) and keep shrinking
// it down for each node as we go down recursively
public static boolean isBST(Node node, int minKey, int maxKey)
{
    // base case
    if (node == null) {
        return true;
    }

    // if the node's value falls outside the valid range
    if (node.data < minKey || node.data > maxKey) {
        return false;
    }

    // recursively check left and right subtrees with an updated range
    return isBST(node.left, minKey, node.data) &&
        isBST(node.right, node.data, maxKey);
}

// Function to determine whether a given binary tree is a BST
public static void isBST(Node root)
{
    if (isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE)) {
        System.out.println("The tree is a BST.");
    }
    else {
        System.out.println("The tree is not a BST!");
    }
}

// Function to determine whether a given binary tree is a BST
public static void isBST(Node root)
{
    if (isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE)) {
        System.out.println("The tree is a BST.");
    }
    else {
        System.out.println("The tree is not a BST!");
    }
}

private static void swap(Node root)
{
    Node left = root.left;
    root.left = root.right;
    root.right = left;
}

public static void main(String[] args)
{
    int[] keys = { 15, 10, 20, 8, 12, 16, 25 };

    Node root = null;
    for (int key: keys) {
        root = insert(root, key);
    }

    // swap left and right nodes
    swap(root);
    isBST(root);
}
```

2.

- I learned that using Integer.MIN_VALUE and Integer.MAX_VALUE as extreme values for comparison is very useful.
- I would implement it exactly the same.

3. `int[] keysI = { 15, 10, 20, 8, 12, 16, 25, 6, 1 }; // h = 4`
`int[] keysII = { 15, 10, 20, 8, 12, 16, 25, 6, 30, 36, 45 }; // h = 5`
`int[] keysIII = { 15, 10, 20, 8, 12, 16, 25, 6, 1, 30, 36, 45, 50 }; // h = 6`

- h = 4; Execution time is: 378529 ns
- h = 5; Execution time is: 400212 ns
- h = 6; Execution time is: 489765 ns

Algorithm 2:

1.

```
// Function to find the k'th largest node in the BST.
// Here, 'i' denotes the total number of nodes processed so far
public static Node kthLargest(Node root, AtomicInteger i, int k)
{
    // base case
    if (root == null) {
        return null;
    }

    // search in the right subtree
    Node left = kthLargest(root.right, i, k);

    // if k'th largest is found in the left subtree, return it
    if (left != null) {
        return left;
    }

    // if the current node is k'th largest, return its value
    if (i.incrementAndGet() == k) {
        return root;
    }

    // otherwise, search in the left subtree
    return kthLargest(root.left, i, k);
}

// Function to find the k'th largest node in the BST
public static Node kthLargest(Node root, int k)
{
    // maintain index to count the total number of nodes processed so far
    AtomicInteger i = new AtomicInteger(0);

    // traverse the tree in an inorder fashion and return k'th node
    return kthLargest(root, i, k);
}

public static void main(String[] args)
{
    int[] keys = { 15, 10, 20, 8, 12, 16, 25 };

    Node root = null;
    for (int key: keys) {
        root = insert(root, key);
    }

    int k = 2;
    Node node = kthLargest(root, k);

    if (node != null) {
        System.out.println(node.data);
    }
    else {
        System.out.println("Invalid Input");
    }
}
```

2.

- I learned that the algorithm for finding the kth largest and kth smallest node is essentially identical with the only difference being that to find the kth largest, the right subtree would be searched first.
- I would implement it by performing an in-order traversal and store the keys into an array in ascending order and using an atomic integer or regular integer to keep track of the number of nodes in the tree. Then I would return array[tracker - k] to return the kth largest node,

3. `int[] keysI = { 15, 10, 20, 8, 12, 16, 25, 6, 1 }; // h = 4`
`int[] keysII = { 15, 10, 20, 8, 12, 16, 25, 6, 30, 36, 45 }; // h = 5`
`int[] keysIII = { 15, 10, 20, 8, 12, 16, 25, 6, 1, 30, 36, 45, 50 }; // h = 6`

- h = 4; Execution time is: 10243 ns
- h = 5; Execution time is: 12363 ns
- h = 6; Execution time is: 11852 ns

Algorithm 3

1.

```
// Recursive function to find the k'th smallest node
// in the BST (using inorder traversal)
public static Node kthSmallest(Node root, AtomicInteger counter, int k)
{
    // base case
    if (root == null) {
        return null;
    }

    // recur for the left subtree
    Node left = kthSmallest(root.left, counter, k);

    // if k'th smallest node is found
    if (left != null) {
        return left;
    }

    // if the root is k'th smallest node
    if (counter.incrementAndGet() == k) {
        return root;
    }

    // Function to find the k'th smallest node in the BST
    public static Node findKthSmallest(Node root, int k)
    {
        // Counter to keep track of the total number of the visited nodes.
        // 'AtomicInteger' is used here since 'Integer' is passed by value in Java
        AtomicInteger counter = new AtomicInteger(0);

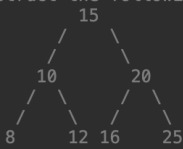
        // Recursively find the k'th smallest node
        return kthSmallest(root, counter, k);
    }

    public static void main(String[] args)
    {
        Node root = new Node(15); // Construct the following BST
        root.left = new Node(10); //
        root.right = new Node(20); //
        root.left.left = new Node(8); //
        root.left.right = new Node(12); //
        root.right.left = new Node(16); //
        root.right.right = new Node(25); //

        int k = 4;

        // find the k'th smallest node
        Node result = findKthSmallest(root, k);

        if (result != null) {
            System.out.printf("%d'th smallest node is %d", k, result.data);
        }
        else {
            System.out.printf("%d'th smallest node does not exist.", k);
        }
    }
}
```



2.

- I learned that the algorithm for finding the kth largest and kth smallest node is essentially identical with the only difference being that to find the kth smallest, the left subtree would be searched first.
- I would implement it the same way as I stated in my answer for algorithm 2, question 2. The only differences would be that I would be searching the left subtree first and I would be returning array[k-1] for the kth smallest node.

3. `int[] keysI = { 15, 10, 20, 8, 12, 16, 25, 6, 1 }; // h = 4`
`int[] keysII = { 15, 10, 20, 8, 12, 16, 25, 6, 30, 36, 45 }; // h = 5`
`int[] keysIII = { 15, 10, 20, 8, 12, 16, 25, 6, 1, 30, 36, 45, 50 }; // h = 6`
- h = 4; Execution time is: 9302 ns
 - h = 5; Execution time is: 11096 ns
 - h = 6; Execution time is: 13382 ns

Algorithm 4

1.

```
// Recursive function to print a complete binary search tree in increasing order
public static void printIncreasingOrder(int[] keys, int low, int high)
{
    if (low > high) {
        return;
    }

    // recur for the left subtree
    printIncreasingOrder(keys, low*2 + 1, high);

    // print the root node
    System.out.print(keys[low] + " ");

    // recur for the right subtree
    printIncreasingOrder(keys, low*2 + 2, high);
}

public static void main(String[] args)
{
    int[] keys = { 15, 10, 20, 8, 12, 18, 25 };
    printIncreasingOrder(keys, 0, keys.length - 1);
}
```

2.

- I learned that in order for this code to properly function, the array passed into the function must represent a binary search tree in level order.
- I would reproduce the solution in a manner where the code did not rely on the array to represent a binary search tree in level order. I would just pass in the binary search tree, perform an in-order traversal, store the values in an array accordingly, and then use a for-loop to print the values.

3. (The code runs for these arrays but does not print in the proper order)
- `int[] keysI = { 15, 10, 20, 8, 12, 16, 25, 6, 1 }; // h = 4`
`int[] keysII = { 15, 10, 20, 8, 12, 16, 25, 6, 30, 36, 45 }; // h = 5`

```
int[] keysIII = { 15, 10, 20, 8, 12, 16, 25, 6, 1, 30, 36, 45, 50 }; // h = 6
```

- h = 4; Execution time is: 548961 ns
- h = 5; Execution time is: 844585 ns
- h = 6; Execution time is: 1273579 ns