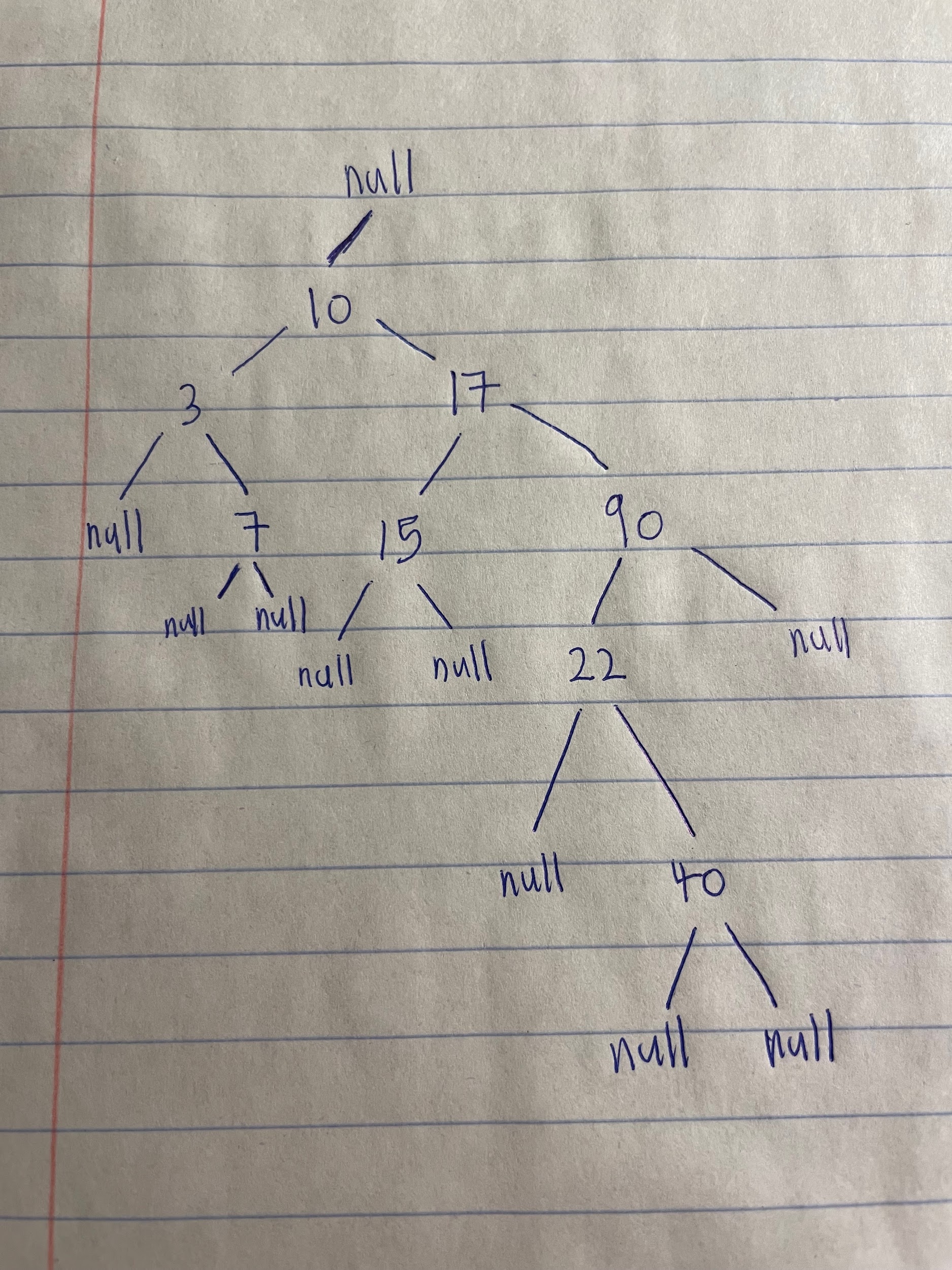
1)



| **Number Inserted** | **# of Comparisons** |
| --- | --- |
| 10 | 1 (branch out from null) |
| 17 | 2 |
| 3 | 2 |
| 90 | 4 |
| 22 | 6 |
| 7 | 4 |
| 40 | 8 |
| 15 | 4 |

**31 comparisons total**

2)

* For the worst case successful search, 9 comparisons is done to get to the number that is deepest in the binary tree, which is **“40.”**
* For the worst case unsuccessful (failed) search, 10 comparisons should be made and **find a number slightly above or below “40”** that will ultimately end at a null node.
* To find the average successful search, find the mean of the values of comparisons for each individual value in the tree. 1+3+3+5+5+5+7+9=38 -> 38/8 = **4.75** comparisons in an average successful search.
* To find 17:
  + Finding the top of the right subtree (“17”) requires **3** units of work.
  + Moving the pointers (previous and current) down through the tree that puts:
    - Previous and current to initially(requires **2** units of work)
    - Move previous and current to 10 & 17 (requires **2** units of work)
* To locate y:
  + Locate smallest value of “22” of right subtree from number “17”:
    - Previous and current moved from 17 to 90, then 90 to 22 (requires **4** units of work)
* Replace 17 with 22 value:
  + No comparison or pointer assignment made (0 unit of work)
* Deletes y of “22”:
  + Pointer assignment made from 90 to 40 (**1** unit of work)

Total units of work:

2 + 2 + 4 + 1 = 9

3 + 9 = **12 units of work**

3)

public void insert(T target) {

if (this.root == null) {

this.root = new BSTNode<T>(target);

} else {

recursiveInsert(this.root, target);

}

}

public void recursiveInsert(BSTNode<T> root, T target) {

int c = 0;

c = target.compareTo(root.data);

if (c == 0) {

throw new IllegalArgumentException("Duplicate key");

} else if (c < 0) {

if (root.left == null) {

BSTNode<T> newNode = new BSTNode<T>(target);

root.left = newNode;

size++;

}

else {

recursiveInsert(root.left,target);

}

}

else {

if (root.right == null) {

BSTNode<T> newNode = new BSTNode<T>(target);

root.right = newNode;

size++;

}

else {

recursiveInsert(root.right,target);

}

}

}

4)

public static <T extends Comparable<T>> void keysInRange(BSTNode<T> root, T min, T max, ArrayList<T> result) {

if (root!=null) {

if (root.data.compareTo(max)<=0 && root.data.compareTo(min)>=0) {

result.add(root.data);

}

keysInRange(root.left,min,max,result);

keysInRange(root.right,min,max,result);

}

}

5)

public static <T extends Comparable<T>> void reverseKeys(BSTNode<T> root) {

if (root == null)

return;

BSTNode<T> temp = root.left;

root.left = root.right;

root.right = temp;

reverseKeys(root.right);

reverseKeys(root.left);

}

6)

public static <T extends Comparable<T>> T kthLargest(BSTNode<T> root, int k) {

if (root.rightSize + 1 == k) {

return root.data;

} else if (root.rightSize + 1 < k) {

return kthLargest(root.left, k - (root.rightSize + 1));

} else {

return kthLargest(root.right, k);

}

}