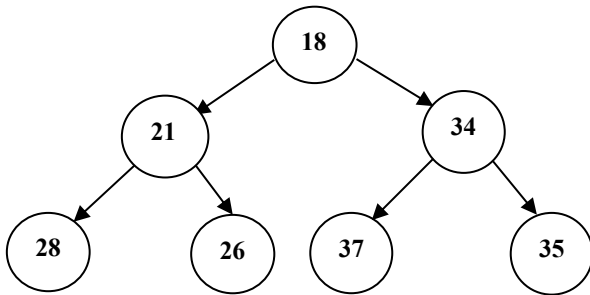


FIRSTNAME, LASTNAME (PRINT IN UPPERCASE): **KEY**

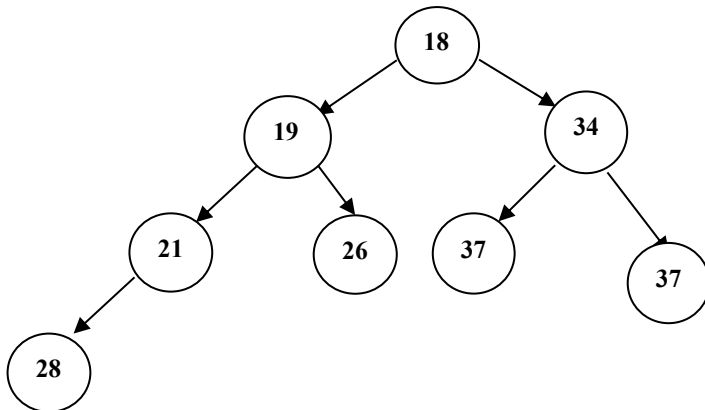
STUDENT ID (e.g. 123456789):

**INSTRUCTIONS:**

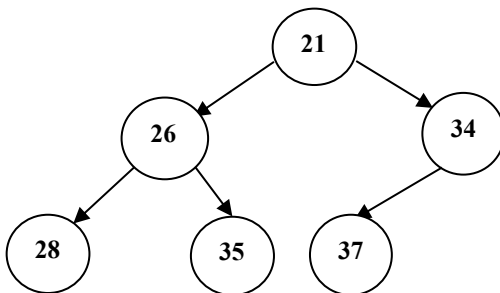
Assume the following min-heap.



1. (4 pts) Draw the heap (as a tree) that would result from inserting **19** in the heap above.



2. (4 pts) Draw the heap (as a tree) that would result from a `removeMin` operation in the heap above (the original, not the one from #1 with **19** in it).



Assume the code below with all necessary import statements.

```
public class BinarySearchTree<K extends Comparable<K>> {
    private class Node {
        private K key;
        private Node left, right;

        private Node(K key) {
            this.key = key;
        }
    }

    private Node root;

    public static void main(String[] args) {
        BinarySearchTree<Integer> tree = new BinarySearchTree<Integer>();

        tree.add(40);
        tree.add(20);
        tree.add(60);
        tree.add(10);
        tree.add(30);
        tree.add(50);
        tree.add(70);
        tree.add(71);

        System.out.println(tree.makeList(20));
        System.out.println(tree.makeList(30));
        System.out.println(tree.makeList(40));
        System.out.println(tree.makeList(35));
    }

    public ArrayList<ArrayList<K>> makeList(K target)
    {
        ArrayList<ArrayList<K>> myList= new ArrayList<ArrayList<K>> ();
        myList.add(new ArrayList<K> ());
        myList.add(new ArrayList<K> ());

        makeListAux(myList, root, target);

        return myList;
    }

    private void makeListAux ( ArrayList<ArrayList<K>> myList, Node rootAux, K target)
    {

        //code this one

    }

    private void //code your 2nd Recursive auxiliary to be called by makeListAux
    {

        //code this one

    }

    /*Assume code for add to add nodes to the BST as seen in class; smaller add to left,
    larger add to right, no duplicate keys allowed*/

}
```

---

### Driver Output

```
[[71, 70, 60, 50, 40, 30, 20, 10], [10, 20, 30]]  
[[71, 70, 60, 50, 40, 30, 20, 10], [30]]  
[[71, 70, 60, 50, 40, 30, 20, 10], [10, 20, 30, 40, 50, 60, 70, 71]]  
[[71, 70, 60, 50, 40, 30, 20, 10], []]
```

---

`makeListAux` will populate the first `ArrayList` in the parameter `myList` with the keys in the tree in descending order (no sort calls allowed, your traversal should be able to achieve this). If the `target` value is in the tree, the second `ArrayList` in the parameter `myList` will be populated by the keys of the subtree rooted at `target` in ascending order. If the `target` value is not there, just leave the second `ArrayList` in the parameter `myList` empty (see last output).

You can have a second recursive auxiliary method of your choice to be called by `makeListAux`. No loops in your code and you may only use the following **library method** calls in all the code you write.

`compareTo` of your comparable

`boolean add(E e)` Appends the specified element to the end of this `ArrayList`.

`E get(int index)` Returns the element at the specified position in this list.

```
private void makeListAux ( ArrayList<ArrayList<K>> myList, Node rootAux, K target)  
{  
    if (rootAux == null)  
        return;  
    else{  
        makeListAux(myList, rootAux.right, target);  
        myList.get(0).add(rootAux.key);  
        if(rootAux.key.compareTo(target)==0) //do in-order  
        {  
            inOrder( myList.get(1), rootAux);  
        }  
        makeListAux(myList, rootAux.left, target);  
    }  
}  
  
private void inOrder( ArrayList<K> myList, Node rootAux)  
{  
    if (rootAux == null)  
        return;  
    else{  
        inOrder(myList, rootAux.left);  
        myList.add(rootAux.key);  
        inOrder(myList, rootAux.right);  
    }  
}
```