built on 2022/06/30 at 13:15:04 ***due:*** *fri jul 08 @ 11:59pm*

This assignment contains both programming and written questions. As a result, we are accepting your work as *two separate* Canvas assignments. For the written portion, you will typeset your answer, produce a PDF file called `hw7.pdf`, and upload it to Canvas. No other format will be accepted. For the programming portion, you'll zip all your files as `a7.zip` and upload it Canvas. Only hand in the files were asked—do *not* hand in the whole IntelliJ project.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Task 1: Mathematical Truth (2 points)

Remember binary trees from class? You will prove a proposition about them using a proof method of your choice:

> **Proposition:** Every binary tree on n nodes where each has either zero or two children has precisely $\frac{n+1}{2}$ leaves.

(*Hint:* By induction.)

Task 2: Higher-Order Merge (2 points)

For this task, save your work in `MultiwayMerge.java`

This is a standard problem in data processing. You could find solutions everywhere on the Internet; please resist the urge to look them up :)

Say you have k `LinkedList<Integer>` instances, each of which is ordered from small to large, though there may be duplicates. Your goal is to merge these lists to create a combined sorted list. You will implement a function

```
public static LinkedList<Integer> mergeAll(LinkedList<Integer>[] lists)
```

that takes an array of `LinkedList<Integer>`'s and returns a `LinkedList<Integer>` that is the combined list in sorted order (small to large).

Your algorithm must run in $O(N \log k)$, where N is the sum of the list lengths and $k =$ `lists.length`. Keep in mind, accessing (i.e., reading from or writing to) a linked list is cheap at the front and the back of the list; however, accessing it anywhere else is generally expensive. The cost is proportional to how far it is from the end. *(Hint: Keep a PriorityQueue and store inside it all the lists. Importantly, the size of the PriorityQueue should never exceed k.)*

A Binary Tree Representation

You will be using the provided binary tree class for the rest of the assignment. The class is given in `BinaryTreeNode.java` and is called `BinaryTreeNode`. As with our discussion in class, each binary

tree node is made up of an integer key, a (reference to) left subtree, and a (reference to) right subtree. The value attribute is omitted so we can focus on the key organization. More specifically:

- an empty tree is represented by `null`; and
- a tree node is a class instance of `BinaryTreeNode` (provided with the starter package) with the following attributes:
  - `left` (that is, one can access `T.left`) contains the left child of this node, `null` if nonexistent.
  - `right` (that is, one can access `T.right`) contains the right child of this node, `null` if nonexistent.
  - `key` is the key stored at this node

In addition, there are two ways to construct a binary tree node. For example,

- `new BinaryTreeNode(10001)` creates a lone binary tree node with the key 10001 that has no children (both `left` and `right` will be `null`).
- Assuming `ll` and `rr` are binary tree nodes, `new BinaryTreeNode(ll, 512, rr)` creates a binary tree node with the key 512, where the left subtree of this tree is what `ll` is, and the right subtree of this tree is what `rr` is.

Finally, we note that if `T` is a binary tree node, we can directly assign `T.left` and `T.right` at will.

Task 3: Let's Grow a Tree (2 points)

For this task, save your work in `MakeTree.java`

In this task, you will write a function to construct a "balanced" binary *search* tree from a list of keys—and analyze its running time.

Subtask I: You are to write a function

```
public static BinaryTreeNode buildBST(int[] keys)
```

that takes an array of integer keys and returns a BST represented using the aforementioned format.

If n is the length of `keys`, your algorithm should take at most $O(n \log n)$ time and construct a BST that is no deeper than $1 + \log_2 n$ levels.

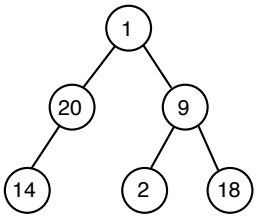
Subtask II: You will also analyze the running time of this algorithm and argue why the tree generated meets the depth requirement.

Task 4: Uprooting (4 points)

For this task, save your work in `Uproot.java`

You may still remember our discussion about tree representations several lectures ago. The *parent mapping* representation, which falls out directly from our definition of a tree, keeps for each node, the parent of that node in a map.

More concretely, for this task, we'll **assume the keys are integers and are unique**. In this context, then, parent mapping keeps a map `Map<Integer, Integer>`, where a node is referred to by its key—and looking up a node in this map would result in the key of its parent node. For example, we show a tree and its corresponding map below:




```
// Representation as a map:
Map<Integer, Integer> p = new ...;
p.put(20, 1);
p.put(9, 1);
p.put(14, 20);
p.put(2, 9);
p.put(18, 9);
```

In this problem, we will write functions to change to and from this representation and the more traditional representation using a `BinaryTreeNode` class.

Subtask I: Implement a function

```
public static HashMap<Integer, Integer> treeToParentMap(BinaryTreeNode T)
```

that takes a binary tree `T`, though not necessarily a BST, and returns a `HashMap` representing the same tree using the parent-mapping representation.

Subtask II: Implement a function

```
public static BinaryTreeNode parentMapToTree(Map<Integer, Integer> map)
```

that takes a parent-mapping map and returns a binary tree encoded as `BinaryTreeNode`. You may notice that the parent-mapping representation has no notion of left vs. right. You are free to choose which is your left node and which is your right node. Moreover, we guarantee that the tree encoded in the map is a legit binary tree, though not necessarily a BST.

Performance Expectations: On input with about 2,000 nodes, we expect your code to return within 2 seconds.

Task 5: Decorative Tree (2 points)

For this task, save your work in `Decor.java`

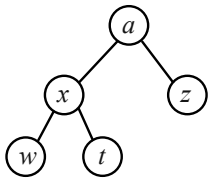
The 1502-er club @ MUIC plans to sell made-to-order embroidery trees and flowers. For a small fee, they'll craft a tree of your design for you. There's a quirk, however. Instead of sending them a drawing, you'll *provide them with the post-order traversal and the in-order traversal of the tree* you intend to build; and they only build binary trees. These are merely binary trees, not necessarily BSTs.

Now **any tree traversal can be written as a sequence of keys we encounter as we traverse the tree.** Specifically, each of the following functions (in a Python-like language) takes a tree and generates a list of keys that one would encounter during a traversal:

```
def postList(T):
  if T==None: return []
  else:
    return postList(T.left) +
           postList(T.right) +
           [T.key]
```

```
def inList(T):
  if T==None: return []
  else:
    return inList(T.left) +
           [T.key] +
           inList(T.right)
```

For example, traversing the tree structure below using post-order and in-order traversals (i.e., the functions `postList` and `inList`, respectively) yield the two lists on its right:



post-order: $[w, t, x, z, a]$
in-order: $[w, x, t, a, z]$

We will assume that the keys are unique—that is to say, no two nodes have the same key. With this assumption, convince yourself that it is possible to come up with a recipe for constructing a tree given the post- and in- order traversal sequences. Your recipe probably looks like this:

(1) determine the root's key from the given sequences;

(2) carve out the post- and in- order traversal sequences for the left subtree;

(3) carve out the post- and in- order traversal sequences for the right subtree; and

(4) solve these recursively.

Your Task: You will implement a function

```
public static BinaryTreeNode mkTree(
    List<Integer> postOrder,
    List<Integer> inOrder
)
```

that takes in two traversal sequences corresponding to postorder or inorder traversals and returns a tree (represented using `BinaryTreeNode`'s and `null` for an empty tree) that matches the traversal sequences. Write as many helper functions as appropriate.

Expectations: For full-credit, your algorithm should run in $O(n)$ time, where n is the length of the traversal sequences.

(*Hint:* A `Map` data structure can be useful in finding where the root of the tree is located. Also, once the location of the root is known in both sequences, can you easily determine in $O(1)$ time the sizes of the left- and right- subtrees?)

Task 6: Optional: HackerRank Problems (0 points)

This part is optional but will be good practice. You do not need to hand it in. There are *four* problems in this set. Work on the following problems on HackerRank:

- `jesse-and-cookies`
- `is-binary-search-tree`
- `queens-attack-2`
- `maximum-subarray-sum`