Homework 6

Data Structures & Algorithms (Term III/2018–19)

built on 2019/06/18 at 23:32:35 ***due:*** *tue jun 25 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to provide you with more experience solving algorithmic problems in Java and reasoning about code. This assignment requires a starter package, which can be downloaded from the course website.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a6.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw6.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Collaboration Policy: To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

Exercise 1: Facts About Graphs (4 points)

Let $G = (V, E)$ be a simple, undirected graph. A simple graph is one where it has no parallel edges nor self loops. Prove the following statements about graphs using your favorite proof method:

(1) **Proposition A:** The sum of the vertex degrees is exactly $2|E|$. More precisely,

$$\sum_{v \in V} \deg_G(v) = 2|E|,$$

where we remember that $\deg_G(v)$ denotes the degree of v in G. *(Hint: counting two ways)*

(2) **Proposition B:** If every vertex of G has degree at least 2, then G contains a cycle.

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.

Exercise 2: Let's Grow a Tree (2 points)

For this task, save your code 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.

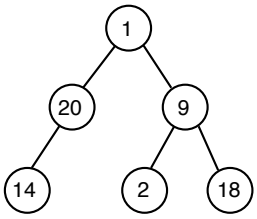
Subtask II: You will also analyze the running time of this algorithm and argue why the tree generated meets the depth requirement.

Exercise 3: Uprooting (4 points)

For this task, save your code 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.

Exercise 4: Decorative Tree (2 points)

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

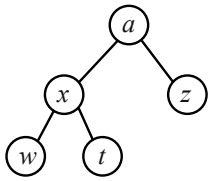
The Programming Club @ MUIC plans to sell made-to-order embroidery trees and flowers. For a small fee, they'll craft a tree of your design for you. There's a quirk, however. Instead of sending them a drawing, you'll *provide them with a post-order traversal and an in-order traversal of the tree* you 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: $[w, x, t, a, z]$

Let's 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. For full-credit, your algorithm should be reasonably fast (i.e., faster than $O(n^3)$).

Exercise 5: HackerRank Problems (8 points)

For this task, save your code in `hackerrank.txt`

There are *four* problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website. Note: If you're planning on using a late token, please finish these HackerRank problems before the due date. The HackerRank "contest" ends at 11:59pm on the day the assignment is due.

Important: You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

https://www.hackerrank.com/muic-data-structures-t-318-assignment-6