Data Structures and Algorithms
Week 3 problem sheet

## A. Trees

  

1. (source: Weiss 18.1) For the tree shown above, state:
	1. Which node is the root?
	2. Which nodes are leaves?
	3. What is the tree’s height?
	4. What are the results of traversing the tree using
		1. preorder
		2. postorder
		3. inorder
		4. level order
* **Solution**
* The answers are:
* a. Node “A” is the root
 b. Nodes G, H, I, L, M, and K are the leaves.
 c. The tree has height 4.
 d. The results if we traverse the tree are as follows:
	+ preorder (NLR): A, B, D, G, H, E, I, J, L, M, C, F, K
	+ postorder (LRN): G, H, D, I, L, M, J, E, B, K, F, C, A
	+ inorder (LNR): G, D, H, B, I, E, L, J, M, A, C, F, K
	+ level order: A, B, C, D, E, F, G, H, I, J, K, L, M
*

## B. Binary trees

1. State whether each of the following statements is TRUE or FALSE, and explain your reasoning.
	1. The height of a binary tree is $O\left(logn\right)$
	2. The proper descendants of a node’s ancestors are also descendants of that node.
	3. A binary tree always has more external nodes than internal nodes.
	4. Every tree has a root node.
*
* **Solution**
* a. The height of a binary tree is $O\left(logn\right)$: FALSE.
This is only true if the tree is a *balanced* binary tree.
b. The proper descendants of a node’s ancestors are also descendants of that node: FALSE.
As an example, consider the tree used in problem A.1 above.
If we look at node J, then J’s ancestors are A, B and E. The proper descendants of node A is the entire tree, with the exception of node A. Now clearly, these proper descendants are *not* the same as J’s descendants (which are the nodes L and M). Hence the statement is false.
c. A binary tree always has more external nodes than internal nodes: FALSE.
Consider a degenerate tree (e.g. the subtree C, F and K from the tree shown in problem A(1) above). It has one external node (K) and two internal nodes (C and F). So the statement is false.
d. Every tree has a root node: TRUE.
This is true (unless we consider trees with no nodes to be trees).
*
1. (Source: Weiss 18.9) Write efficient methods (and give their Big “O” running times) that take a reference to a binary tree root t and compute
	1. The number of leaves in t
	2. The number of nodes in t that contain one non-null child
	3. The number of nodes in t that contain two non-null children
*
* **Solution**
* For problem (a):
* public static int countLeaves(BinaryTreeNode t) {
 if (t.left == null && t.right == null)
 return 1;
 int total = 0;
 if (t.left != null) {
 total += countLeaves(t.left);
 }
 if (t.right != null) {
 total += countLeaves(t.right);
 }
 return total;
 }
* All nodes must be visited, so it has complexity $O\left(n\right)$, where $n$ is the number of nodes in the tree.
* For problem (b):
* // count nodes with 1 child
 public static int countNodes(BinaryTreeNode t) {
 int numChildren = 0;
 if (t.left != null)
 numChildren++;
 if (t.right != null)
 numChildren++;
 int total = 0;
 if (numChildren == 1)
 total += 1;
 if (t.left != null) {
 total += countNodes(t.left);
 }
 if (t.right != null) {
 total += countNodes(t.right);
 }
 return total;
 }
* Again, all nodes must be visited, so complexity is $O\left(n\right)$.
* For problem (c):
* // count nodes with 2 children
 public static int countNodes(BinaryTreeNode t) {
 int numChildren = 0;
 if (t.left != null)
 numChildren++;
 if (t.right != null)
 numChildren++;
 int total = 0;
 if (numChildren == 2)
 total += 1;
 if (t.left != null) {
 total += countNodes(t.left);
 }
 if (t.right != null) {
 total += countNodes(t.right);
 }
 return total;
 }
*
1. (Source: Weiss 18.10) Suppose a binary tree stores ints. Write efficient methods (and give their Big-Oh running times) that take a reference to a binary tree root t and compute
	1. The number of even data items
	2. The sum of all the items in the tree
*
* **Solution**
* For problem (a):
* public static int countEven(BinaryTreeNode t) {
 int total = 0;
 if (this.value % 2 == 0)
 total++;
 if (t.left != null) {
 total += countEven(t.left);
 }
 if (t.right != null) {
 total += countEven(t.right);
 }
 return total;
 }
* All nodes must be visited, so complexity is $O\left(n\right)$.
* For problem (b):
* public static int sumTree(BinaryTreeNode t) {
 int total = this.value;
 if (t.left != null) {
 total += sumTree(t.left);
 }
 if (t.right != null) {
 total += sumTree(t.right);
 }
 return total;
 }
* All nodes must be visited, so complexity is $O\left(n\right)$. (We assume the cost of performing arithmetic here is negligible.)
*

### C. Binary search trees

1. (Source: Weiss 19.1) Show the result of inserting 3, 1, 4, 6, 9, 2, 5, and 7 in an initially empty binary search tree. Then show the result of deleting the root.
* **Solution**
* After the numbers have been inserted, the tree will look like this:
* 
* If the root is deleted, then it must be replaced (either by the largest item in its left-hand sub-tree, or the smallest item in its right-hand sub-tree).
* If we use the largest item in the left sub-tree, the result is as follows:
* 
1. (Source: Weiss 19.2) Draw all binary search trees that can result from inserting permutations of 1, 2, 3, and 4. How many trees are there? What are the probabilities of each tree’s occurring if all permutations are equally likely?
* **Solution**
* There are 24 permutations of these numbers (we can calculate this by observing that $4×3×2×1=24$).
* Each permutation gives rise to a particular tree, but sometimes the same tree occurs for multiple different permutations. For example, inserting [3, 1, 2, 4] and [3, 2, 4, 1] and [3, 2, 1, 4] all make the same tree.
* The following permutations give rise to unique trees; in parentheses are the number of permutations that give an identical tree.
* 1234 (1), 1243 (1), 1324 (2), 1423 (1), 1432 (1),
* 2143 (3), 2134 (3),
* 3214 (3), 3124 (3),
* 4321 (1), 4312 (1), 4213 (2), 4123 (1), 4132 (1).
* So 14 trees are possible. We can calculate the probability of any tree occurring by looking at the number in parentheses; if we call that number $f$, then the probability of the tree occurring is $\frac{f}{24}$.
1. (Source: Weiss 19.15) Implement the BinaryTree methods find, findMin, and findMax recursively.
* **Solution**
* Note we are asked to implement the methods for a Binary Tree, *not* a Binary Search Tree. Therefore, we can’t assume that the nodes are in any particular order; in order to find a node, we must search the whole tree.
* Below, we show code for the findMin method, which makes use of a helper method minPair. The findMax and find methods are very similar and are not shown.
* /\*\*
 \* Compare two tree nodes and return the one with the min value
 \* @param n1 binary tree node 1
 \* @param n2 binary tree node 2
 \* @return the node with the minimum value
 \*/
 public static BinaryNode minPair(BinaryNode n1, BinaryNode n2) {
 if (n1==null && n2==null) {
 throw new IllegalArgumentException("at least one node for minPair must be non null");
 }

 if (n1 == null) {
 return n2;
 } else if (n2 == null) {
 return n1;
 } else if (n1.value <= n2.value) {
 return n1;
 } else {
 return n2;
 }
 }
* /\*\*
 \* Find the node in a tree with the minimum value for that tree
 \* @param t
 \* @return node in t with the minimum value
 \*/
 public static BinaryNode findMin(BinaryNode t) {
 if (t == null)
 return null;

 BinaryNode leftMin = minPair(t, findMin(t.left));
 BinaryNode rightMin = minPair(t, findMin(t.right));
 return minPair(leftMin, rightMin);
 }

## D. Maps

Use the MapDemo.java class from the code bundle, the Oracle online Java documentation (at [**https://docs.oracle.com/javase/8/docs/api/java/util/Map.html**](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)) and your preferred IDE (e.g. Eclipse) to work out answers to the following questions about maps.

1. How do you enter a new pair into a map object?
2. Maps have at most one value per key. What happens if you enter a pair with a key that already exists in the map?
3. How do you find all the keys in a map (the map’s domain)?
4. How do you find all the values in a map (the map’s range)?
5. Describe two ways to remove a pair from a map.

Questions 1–5 are intended to give you practice writing your own code in Eclipse, so full sample answers are not provided.

Some brief hints, however, on what methods of the Map class you should be looking at:

* for question 1, check out the put() method
* for question 2: you should try this and find out. (Or check the documentation.)
* for question 3, check out the keySet() method
* for question 4, check out the values() method
* for question 5, check out the remove() and clear() methods (but note that the latter removes all other keys, as well

If you are not sure how to check whether your answers are correct, feel free to contact the lecturer.

## E. Collections API

1. Consider the four core interfaces of the Collections API: Set, List, Queue, Map. For each of the four assignments below, specify which of the four core interfaces is best-suited to the problem, and explain how to use an implementation of it to implement the assignment. You can complete the code for this in CollectionsDemo.java.
	1. Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
	2. WTI has decided that each new product will be named after an employee but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
	3. WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.
	4. WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.
*
* A brief guide to approaching this sort of question:
* a. A List would be fine here. Every item in a list has a specific position in the list. So to choose a random employee from a list of (say) 25 people, you just need to generate a random number from 0 to 24, and look at the name in that position.
* It is *possible* to use a Set instead, but it would require use of the Iterable class, and we won’t examine this in detail.
* b. Whenever we see a requirement to record “unique” items of some sort, it is worth considering whether a Set might be suitable, because Sets only store unique values.
* And in this case, a Set is indeed the best-suited structure.
* c. Whenever we have a collection of things, and want to store some property “about” them, it is worth considering whether a Map would be a good fit.
* In this case, a Map is indeed the best-suited interface. We could use a map of type Map<String,Integer> to store each first name (a String), and associated with that first name, the number of employees who have that name,
* d. Whenever we are asked to maintain some sort of “waiting list” or “queue” or “prioritized list” of values, it is worth considering whether a Queue might be the best data structure - and it is indeed the best-suited one, in this case. We could use a Queue to implement a waiting list maintained on a “first come, first served” basis, OR, if there were some employees who had specifl priority, we could use a Priority Queue.
* In more detail:
* For a.:
* In order to select an employee at random, we need to be able to: (i) find out how many employees there are; (ii) generate a random number from 0 to (number of employees - 1); and (iii) get a reference to a particular employee’s name.
* Any of the interfaces will let us perform task (i), and task (ii) is independent of what interface we choose. But task (iii) will be much more straightforward if we use a List interface.
* Our code would be something like (assuming our list of names is called employeeNames, and is of type List):
* Random r = new Random();
 int i = r.nextInt(employeeNames.size());
 System.out.println( employeeNames.get(i));
* (Assuming we just print the employee name to standard output.)
* For b.:
* If we need to get a unique list of anything – e.g. a list of unique first names – then Set is the most appropriate interface, since sets do not permit duplicate values.
* We assume here that each name in the variable employeeNames contains two names, a first name and a last name, with the first name appearing first, and with a space between the two names. (However, any reasonable assumptions about what the variable contains are okay.)
* Code to get a list of unique first names would then be:
* Set<String> uniqueFirstNames = new TreeSet<>();
 for (String employeeName : employeeNames) {
 uniqueFirstNames.add( employeeName.split(" ")[0] );
 }
* (See the documentation of the [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class for details of the .split method.)
* For c.:
* In this example, we need to record a value – namely, the total count, or frequency of the name – for each first name. This means that we need to maintain a map from names to name frequencies, and a Map is the most appropriate interface.
* Code to do this would be:
* Map<String, Integer> nameFrequency = new TreeMap<>();
 for (String employeeName : employeeNames) {
 String firstName = employeeName.split(" ")[0];
 if (nameFrequency.containsKey(firstName)) {
 int currFreq = nameFrequency.get(firstName);
 nameFrequency.put(firstName, currFreq + 1);
 } else {
 nameFrequency.put(firstName, 1);
 }
 }
* Here we use a TreeMap, but any Map implementation should work.
* For d.:
* Any time we need to maintain a waiting list or queue of some sort, that is a strong hint that we need to use a Queue interface.
* From the wording of the question, it seems we only need to declare and initialize the queue – not populate it.
* We make the assumption that the waiting list should be “first in, first out”, and thus a normal queue is sufficient. If some employees had special priority, we would instead need to use a PriorityQueue.
* So to declare and initialize the queue, we can write:
* Queue<String> waitingList = new LinkedList<String>();
* (because LinkedList implements the Queue interface).
*