READ THIS NOW!

- Print your name in the space provided below.
- There are 8 short-answer questions, priced as marked. The maximum score is 100.
- This examination is closed book and closed notes, aside from the permitted one-page fact sheet. Your fact sheet may contain definitions and examples, but it may not contain questions and/or answers taken from old tests or homework.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Until solutions are posted, you may not discuss this examination with any student who has not taken it.
- Failure to adhere to any of these restrictions is an Honor Code violation.
- When you have finished, sign the pledge at the bottom of this page and turn in the test and your signed fact sheet.

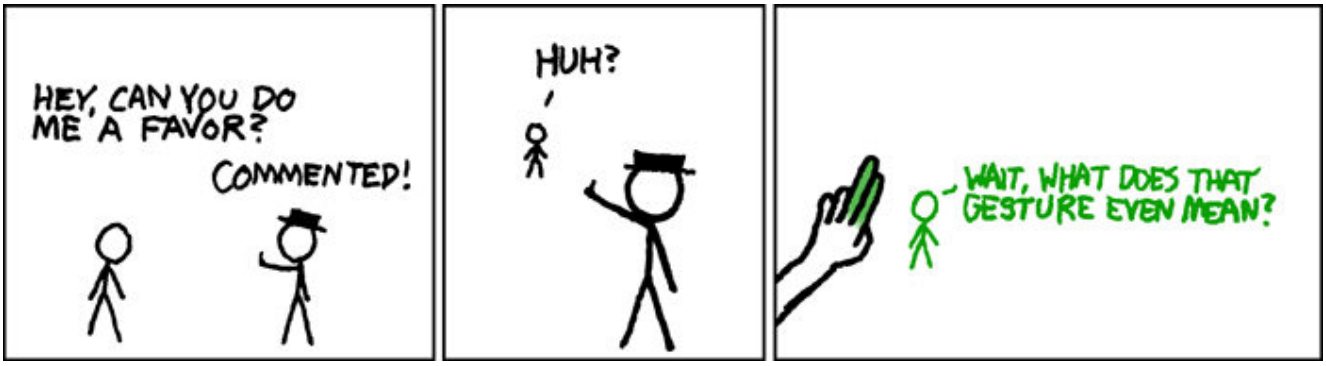
Name (Last, First) Solution

printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____

signed


HEY, CAN YOU DO ME A FAVOR?
COMMENTED!
HUH?
WAIT, WHAT DOES THAT GESTURE EVEN MEAN?

xkcd.com

1. [12 points] The following Java function is intended to take the root of a binary tree and determine whether the data elements stored in the tree obey the ordering property for a BST (assuming duplicate values go to the right):

```
boolean isBST( BinaryNode sRoot ) {

   if ( sRoot == null ) return true;

   if ( (sRoot.left != null) &&
        (sRoot.element.compareTo(sRoot.left.element) <= 0 )
      return false;

   if ( (sRoot.right != null) &&
        (sRoot.element.compareTo(sRoot.right.element) > 0 )
      return false;

   return ( isBST(sRoot.left) && isBST(sRoot.right) );
}
```

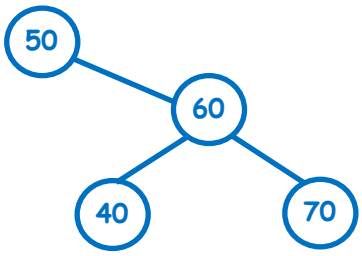
The function is syntactically correct. Is the function logically correct or not? If it's not correct, is the problem that it will sometimes say a tree has the BST property when it does not, or that it will sometimes say a tree does not have the BST property when it does, or both?

Justify your conclusion carefully. Example trees would be useful.

The logic is incorrect; the fundamental error is that the tests are only "local", that is, the value in each node is compared only to values in the children of that node, to see if the value in the left child is actually smaller and the value in the right child is actually larger or equal to the value in the parent.

That means the function will always be correct if it returns false.

But it would return true if given the following tree, which violates the BST property:



2. a) [12 points] Prove: for all $\lambda \geq 1$, a full binary tree with λ levels must have at least $2\lambda - 1$ nodes.

If $\lambda = 1$ then we have a tree with one node, and so the number of nodes, N, equals $2\lambda - 1$.

Suppose that for some $\lambda \geq 1$, if T is a full binary tree with λ levels then T has at least $2\lambda - 1$ nodes.

Let T be a full binary tree with $\lambda + 1$ levels. Then the root of T is not a leaf, since there are at least 2 levels, and hence the root has two nonempty subtrees, one of which must have λ levels.

That subtree must, by the inductive assumption above, have at least $2\lambda - 1$ nodes.

Since the other subtree must contain at least one node, and we also have the root node, the number of nodes in the entire tree must be at least $(2\lambda - 1) + 1 + 1 = 2(\lambda + 1) - 1$ nodes.

QED

b) [2 points] Where, in your proof, did you use the fact that the tree was full?

That's explicitly indicated in my proof above, but the answer is you need to know it's a FBT in order to argue the root has two nonempty subtrees when making the inductive step.

3. [10 points] Consider the implementation of a hash table. Suppose that when an insertion operation is performed, the following hash table slots are examined during the insertion:

$$s_0, s_1, s_2, s_3, s_4$$

Suppose that slot s_4 is found to be empty, and that there is a tombstone in slot s_2 (and no other tombstones occur in this sequence of slots).

Would there be any logical issues if the insertion was handled by moving the element that is currently in slot s_3 to slot s_2 (replacing the tombstone), and inserting the new element into slot s_3 (replacing the element that just got moved to slot s_2), and leaving slot s_4 empty?

The proposed strategy is unsafe.

For example, the element that was originally in slot s_3 may actually be in its home slot. If we move that element to slot s_2, searches for that element will either fail altogether or succeed only after one or more probe steps that would not have been needed before.

4. [10 points] What are the precise conditions under which the design of a hash table must incorporate the use of tombstones?

Tombstones are necessary if the table uses some form of probing to resolve collisions.

5. a) [8 points] Quadratic probing suffers the problem that it may not reach all the slots in the table. Could the same difficulty occur if double hashing, with a secondary hash function that never returns 0, was used for the probing strategy? Justify your conclusion precisely.

Yes.

Suppose that we have a key that the second hash function (in double hashing) maps to the integer H. Suppose that H is a divisor of the table size N, say that N = Hq.

Then, if we take q probe steps for this key, we will wind up at the original home slot for the element, so we will not have examined all the slots in the table

b) [8 points] What is the primary advantage of double hashing over quadratic probing?

With double hashing, records that collide in the same home slot will be likely to follow different probe sequences since the secondary hash function will be likely to map the keys for those records to different values.

6. Two developers are told to implement a hashing scheme for the same set of data records, and to use the same hash function and table size. One developer decides to use linear probing to resolve collisions. The other developer decides to use chaining with singly-linked lists in each table slot, appending new records to the lists.

Both developers test their implementations by inserting the same set of data records, in the same order, into their tables, and studying the results.

When testing his implementation, the first developer discovers that records with the keys 83, 17, 52 and 61 collide in the same slot in the table, in that order.

a) [9 points] Given the information above, the second developer expects that her implementation (using chaining) will require exactly four element comparisons to find the record with key 61. Is she correct? Explain.

She is possibly correct. Those records will all go into a chain in the same hash table slot.

But, it is not clear from the given information that those are the ONLY records that collide in that slot (although that is a reasonable supposition).

However, we do not know whether those records will be inserted at the end or the front of the chain as they arrive (or possibly in some other order).

So, depending on the answers to those questions, she may be right or she may be wrong.

But, since she presumably knows those answers, I suspect she is correct since otherwise she would not have reached that conclusion in the first place.

b) [9 points] What can the first developer conclude about the number of element comparisons his implementation (using linear probing) will require to find the record with key 61? Explain.

He can conclude that searches for the record with key 61 will require AT LEAST 4 comparisons.

Even if no other records map to that same table slot, since he has used linear probing it's likely that there will be clustering in his table. If so, other records that hash to nearby slots may "interleave" with these four records and that will cause 61 to be stored further from its home slot.

7. [10 points] Consider implementing Pugh's probabilistic skip list in Java. Would it make sense to use an inheritance hierarchy of node types? If yes, describe the hierarchy. If no, explain why using inheritance would not make sense, and describe how you would implement the nodes. (Hint: consider the in-class discussion of the PR quadtree.)

Different nodes in Pugh's probabilistic skip list may store different numbers of pointers, but otherwise they are identical.

So, if we consider using inheritance for the nodes, the different types would correspond to different levels: Level0Node, Level1Node, etc. We would also need an abstract base type, say Node, from which each of these types would be derived.

But this will not work cleanly. The actual nodes would have to store Node pointers (references), since the targets of those pointers could be of any Level*Node type. And therefore, we'd have to perform type checks on the targets of those pointers during list traversals.

Or, we could have a "chain" of node types, where Level1Node is derived from Level0Node and so forth; then each subtype could add a new data member, a pointer of the type corresponding to its level...

But what does all this buy us? Not much, if anything.

A single node type will do:

```
class skipNode {
    T element;              // generic "socket" for user data object
    int level;              // level of node and dimension of array forward
    skipNode[] forward;     // array of pointers to following nodes
}
```

This allows us to allocate custom-sized pointer arrays for each node, so no space is wasted, an we have no need for type checking of the pointer targets.

Conceptually, the notion of a node's level makes more sense as an attribute of the node, not as specifying a type of node.

8. [10 points] Under what circumstances would it make more sense to use a sorted array to organize a collection of records, rather than an AVL tree? Why?

If we know all the data elements in advance, we can build a sorted array in less time than building an AVL tree, by putting the elements into an array and running an N log N sort algorithm on the array.

If the application does not require deleting any elements after the structure is built, then we do not need to worry about the cost of shifting elements in the sorted array.

The sorted array will also have no storage "overhead" since it does not require node pointers and we can make the array to be exactly the required size.

So, if we have a fixed collection of data records, known in advance, a sorted array will be more efficient.

On the other hand, if we need to add (or remove) data elements on the fly, the AVL tree wins on performance cost since search traversals will be O(log N), and the rebalancing rotations will cost much less than shifting a nontrivial subset of the elements in an array.

Another, less likely scenario is that we are in a situation where time is much less precious than memory utilization. The AVL tree will definitely consume more memory than a sorted array, IF we make the array of minimal size. If we don't know the full set of elements in advance, we might wind up resizing the array multiple times, which is time-consuming, but if space is the primary issue, that would be preferable.