READ THIS NOW!

- Print your name in the space provided below.
- There are 6 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. You may include examples from the course notes.
- 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.

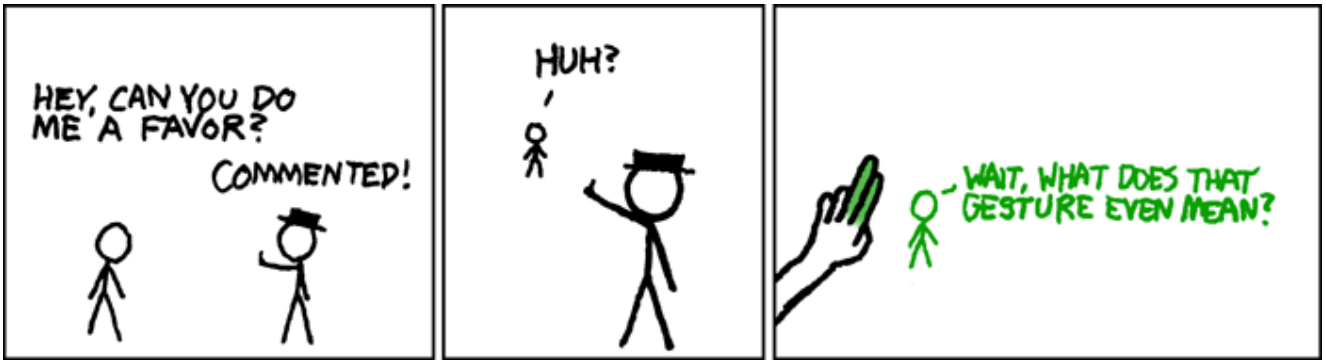
Name (Last, First) Solution

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. [16 points] Complete the implementation of the private helper function for the following BST member function, which is intended to delete from the BST all the nodes that were leaves when the function was called. The only relevant declarations from the BST implementation are that it has a member named `root` and an inner `BinaryNode` class. You may not invoke other tree methods. The function should call a private helper function, for which you should show an implementation. Assume the tree does not store duplicate values.

```
// Indicate any changes to the BST class here:
// Add the following member to the BST class:
int nLeavesDeleted;


/** Deletes all (current) leaf nodes from the BST.
 *
 *  Returns:
 *        the number of leaves that were deleted
 */
public int rmLeaves( ) {

   nLeavesDeleted = 0;

   root = rmLeavesHelper( root );

}

private BinaryNode rmLeavesHelper( BinaryNode sRoot ) {

   if ( sRoot == NULL ) return 0;

   // if current node is a leaf, count it and remove it
   if ( sRoot.left == NULL && sRoot.right == NULL ) {

      nLeavesDeleted++;     // count this leaf
      return null;          // tell parent to lose this child node
   }

   // deal with possible leaves in the subtrees of the current node
   sRoot.left  = rmLeavesHelper( sRoot.left );
   sRoot.right = rmLeavesHelper( sRoot.right );

   return sRoot;
}
```

2. The design discussion for a PR quadtree implemention resulted in the decision to create an inheritance hierarchy of nodes.

a) [8 points] What was (allegedly) gained by using a hierarchy of node types?

In a PR quadtree, internal nodes do not store data objects and leaf nodes never acquire children.

By having a hierarchy of node types, we can save the cost of a data object reference in each internal node and the cost of four child references in each leaf node.

b) [8 points] Exactly why did the design not derive the leaf type from the internal node type, or vice versa?

Leaf nodes have a data member (the data reference) that internal nodes do not need, and internal nodes have four data members (the child references) that leaf nodes do not need.

If we derived either type from the other, the derived type would inherit one or more data members that it does not need, and that would increase the memory cost for no reason.

---

3. Assume that you have a collection of data objects that correspond to points with integer coordinates in the square region of the xy-plane bounded by x = 0, y = 0, x = 1024, y = 1024. Assume that you will organize the data objects in a PR-quadtree (with bucket size 1).

a) [8 points] If the tree contains N data objects, what is the minimum number of levels the tree could have (as a function of N)? Justify your answer.

The quadtree would need N leaves, since only leaves store data. The maximum number of nodes in level k is 4^k, since the each internal node can have up to 4 children.

Therefore, we need for the bottom level, k, to be such that $4^k >= N$, which means that k must satisfy $k >= \log_4(N)$.

So there must be at least $1 + \log_4(N)$ levels.

b) [8 points] Suppose that the PR-quadtree is full enough that every leaf node represents a region that is 8 x 8 or smaller. A new data object, X, is inserted into the tree, and the distance between X and the closest existing data object in the tree is 2. Will the insertion of X necessarily cause a node to split? Either way, justify your conclusion.

X could be on the opposite side of a region boundary from the closest other element, and X is in a region that is currently empty, then there would be no need to split a region.

4. [12 points] Suppose that a hash table uses linear probing to resolve collisions, with a table that has 1021 slots. If 1021 records are hashed into the table, and 1020 of them have different home slots, what is the minimum number of records that could wind up being stored into their home slots? Justify your conclusion.

It's possible that only 1 record falls into its own home slot.

Suppose the two records that share a home slot, say X1 and X2, are inserted first; then X2 will be bumped into a slot that is actually the home slot of a record, say X3, that hasn't been inserted yet.

Now, if X3 is inserted next, it will be bumped from its home slot, into the home slot of some other record that has not been inserted yet.

If this pattern continues, only X1 will be in its home slot.

---

5. [20 points] Two hash tables are implemented, and will be tested with the same set of key values. Both tables use the same hash function, and use an array of the same prime size. The difference is that

- Table A uses quadratic probing to resolve collisions, implemented so that it is guaranteed to find an empty slot if the table contains one.
- Table B uses chaining to resolve collisions, using a probabilistic skip list for the chains.

Aside from simplicity of implementation (assuming we have a canned implementation of the skiplist), identify and explain two different reasons that Table B is likely to exhibit better search performance than Table A. Be specific.

Reason 1:

With Table A, it's possible for two records that have different home slots to collide with each other during probing.

That would increase the cost of searching for that element vs Table B, since there only elements that fall in the same home slot ever collide.

Reason 2:

Even if records with different home slots never collide, Table A will have to do what amounts to a linear search through the records that collide in the same home slot.

Table B will likely get logarithmic search cost when it goes through the chain in a slot.

6. Consider the following BST insertion code:

```
public void insert(T elem) {

   if ( elem == null ) return;

   root = insertHelper(elem, root);
   if ( root != null ) {
      root.element = elem;
   }
}

private BinaryNode insertHelper(T elem, BinaryNode sRoot) {

   if ( sRoot == null ) {
      return getNode();      // retrieves node from node pool,
                             // or makes new one, if pool is empty
   }

   int compare = elem.compareTo(sRoot.element);
   if ( compare < 0 )
      sRoot.left = insertHelper(sRoot.element, sRoot.left);
   else if ( compare > 0 )
      sRoot.right = insertHelper(sRoot.element, sRoot.right);

   return sRoot;
}
```

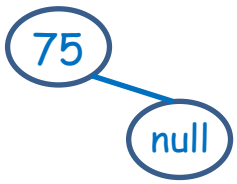
If `insert(50)` is called on an empty tree this will result in a BST consisting of a root node holding the value 50.

a) [10 points] Now suppose `insert(75)` is called on the tree created by the call described above. Describe the state of the resulting tree.

There are two fundamental errors in the given code:

- the public function will always put the data value elem into the current root node, replacing whatever should have been there
- the getNode() function has no way to put the data value elem into the new node, so once the root is inserted, no other nodes will ever contain a data element

Therefore, after inserting 75, the tree will look like this:



b) [10 points] Finally, suppose `insert(100)` is called on the tree created by the call described in part a). Explain what will happen, and why.

Now, the insertion will go to the right from the root node, based on the comparison of 75 and 100 at the root.

When the insertion reaches the right child of the root, the code will call compareTo() and pass it null... the result will most likely be a null pointer exception thrown from compareTo().

However, it is possible compareTo() will return a value in this situation. In that case we would wind up with another empty node below the existing empty node, and the value 100 in the root node.