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.

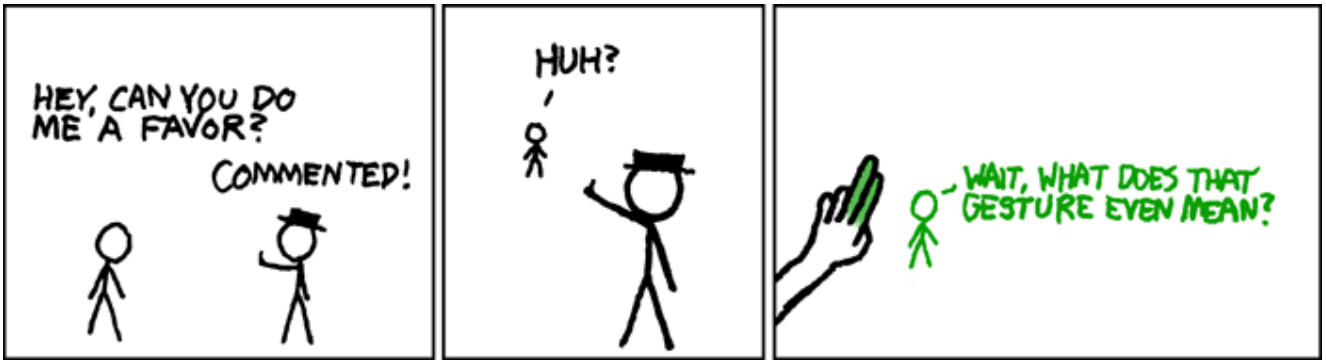
Name (Last, First) Solution

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 following BST member function, which is intended to determine the number of values in the BST that are less than a given value `K`. 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.

```
/** Returns the number of elements in the tree that are < K (according
 *  to compareTo().
 *
 *  Pre:  K is not null
 *  Post: neither (the target of) K nor the BST are modified
 *  Returns:
 *        size of the set {X in BST | X < K}
 */
public int kCount( T K ) {

   return kCountHelper(K, root);

}

private int kCountHelper(T K, BinaryNode sroot) {

   // Nothing here, nothing to count.
   if ( sroot == null ) return 0;

   int kCompare = K.compareTo(sroot.element);

   if ( kCompare <= 0 ) {

      // K <= sroot.element; the only candidates to be smaller than K
      // must lie to the left of sroot
      return kCountHelper(K, sroot.left);
   }
   else {

      // K > sroot.element; so we have one value to count here,
      // and there could be other values smaller than K in both
      // subtrees of sroot.
      return 1 +
             kCountHelper(K, sroot.left) +
             kCountHelper(K, sroot.right);
   }
}
```

Here's my analysis of the problem:

- if I'm at a non-existent node (sroot == null), I have not found anything < K
- if I'm at a node, and K is <= the element there, then everything in the right subtree of the current node is > the element in the current node and therefore > K, and I only need to look to the left
- if I'm at a node, and K is > the element there then I need to count that element, and everything in the left subtree of the current node is also < K, so I have to look there, and there may be elements in the left subtree that are < K, so I also have to look to the left

2. [12 points] Consider the following claim about key values stored in a BST that does not contain any duplicates:

If X is the key stored in any leaf node, and Y is the key stored in that leaf node's parent, then either:

(i) Y is the smallest key in the BST that is larger than X, **or**

(ii) Y is the largest key in the BST smaller than X.

If the statement is true, say so and present an argument proving it. If the statement is false, say so and provide a counter-example.

I came up with three proofs. The first is rigorous and uses induction in a similar manner to the proofs in the course notes:

Proof 1 (induction on the number of levels in the BST):
The statement holds trivially if the number of levels is 0 or 1. Suppose the tree has 2 levels; then Y is in the root node. Therefore, Y is larger than the value in its left child and smaller than the value in its right child (assuming both exist), so the statement holds trivially.

Now, suppose the statement holds for any BST with k or fewer levels, and suppose that T is a BST with k + 1 levels.

Then T consists of a root node and two subtrees, SL and SR, each of which have k or fewer levels. Let X be the value in a leaf node in T, and Y be the value in the parent of X. There are two cases: Y is in SL and Y is in SR (Y being the root is handled by the base cases).

If Y is in SL, then Y is smaller than the value in the root, and every value in SR. Now there are two further cases: X is the left child of Y and X is the right child of Y.

If X is the left child of Y, then X < Y and by the inductive assumption, Y is the smallest key in SL that's larger than X. Since Y is smaller than the root and all keys in SR, Y is the smallest key in T that's larger than X.

If X is the right child of Y, then Y < X, and by the inductive assumption, Y is the largest key in SL that's smaller than X. And, since X is in SR, X is smaller than the root and every key in SR. So, Y is the largest key in T that's smaller than X.

The case that Y is in SR is argued similarly.

The second proof uses a property of inorder traversal; this may seem a little sketchy since that property hasn't been proved, but it's easy to prove (by induction) that inorder traversals visit the keys in ascending order.

Proof 2:
Recall that an inorder traversal of a BST visits the keys in ascending order.

Now, X is either the left child or the right child of Y.

Therefore, during an inorder traversal, either X is visited immediately before Y, or X is visited immediately after Y.

Either way, there is no possibility that a value that lies between X and Y could occur in the tree.

The third proof is also rigorous, and is the hardest to argue. It hinges on observations about the way insertion works, what the order of insertion must be for ancestor/descendent keys, and a common mathematical trick: define something to be the smallest (largest) element of a set of things that have some common property, and use that bound to leverage a contradiction.

Proof 3:
Suppose that X is the left child of Y, so that X < Y. Let Z be the smallest element of the tree such that X < Z <= Y, and suppose that Z ≠ Y. Note that Z cannot be on the path from the root to Y, since then Y would have gone to the right at Z but X would have gone to the left at Z.

Then, there must be a "lowest" key, W, that lies along the path from the root of the BST to Y, such that W is an ancestor of both Z and Y (if nothing else, the root has that property).

Note that this implies that X,Y and Z were all inserted after W.

Now, since W is the "lowest" common ancestor of Y and Z, but Z cannot be on the path from the root to Y, it must be true that Y and Z went in different directions at W, and that means that Y > W > Z > X (since Z < Y).

But then, when X was inserted (after Y since Y is the parent of X), X would have gone in a different direction than Y at W, and so Y cannot be the parent of X. Therefore, by contradiction, Z must be Y.

A symmetric argument handles the case that Y < X.

Note:

In grading this, I gave no credit for answers that alleged the statement was false. If such an answer did not include a counterexample (incorrect, of course) then it did not meet the problem statement. And, any alleged counterexample must be incorrect.

I did notice that almost all of the supposed counterexamples violated the fact that one of the nodes referred to in the problem is a leaf.

Finally, a fair number of answers resorted to what I call "proof by sincere assertion". That is, a critically-needed fact was stated with no supporting argument. Those got no more than half credit.

3. A hash table implementation uses linear probing to resolve collisions.

a) [5 points] Suppose the table is initially empty, and that records R1, R2, …, and RN are inserted into the table, in that order, all of those records have different key values, and all of those records map to the same home slot. What is the minimum number of record comparisons that could be performed in a search for the record RN.

N.

Since the table is initially empty and linear probing is used, the records R1 through RN must have been inserted in a consecutive sequence of N cells (possibly wrapping around the end). So, any search for RN must begin in the home slot (holding R1) and proceed slot by slot until RN is found (after N comparisons).

b) [5 points] Assume the same conditions given in part a). What is the maximum number of record comparisons that could be performed in a search for the record RN.

N.

By the logic given in part a), there is only one possible outcome when RN is sought.

c) [5 points] Suppose the table is initially empty, and records S1, S2, …, and SM are inserted into the table, in that order, all of those records have different key values, and all of those records map to different home slots. Then, records R1, R2, …, and RN are inserted into the table, in that order, all of those records have different key values, and all of those records map to the same home slot, which is not the home slot of any of the Sk inserted earlier. What is the minimum number of record comparisons that could be performed in a search for the record RN.

N.

The insertion of M additional records surely cannot yield a shorter search for RN than if those M records did not occur in the table.

d) [5 points] Assume the same conditions given in part c). What is the maximum number of record comparisons that could be performed in a search for the record RN.

N + M.

In the worst case, the M additional records all fell into slots that would have been accessed during the insertion of R1 through RN if those M records had not existed. In that case, a search for RN would have to proceed past each of the additional M records.

e) [8 points] What would the answer to parts c) and d) be if the table used chaining instead of probing to resolve collisions?

Yes, the answer to part d) would now be N. The answer to part c) would not change, unless you assumed that chaining was implemented so that new elements were added at the front of the chain, in which case the minimum would be 1 comparison. If you assumed the chain was something other than a linked list, that was OK as long as you said that and you correctly determined the minimum/maximum number of comparisons.

If chaining, is used all of the Rk will be stored in a single slot in the table, and none of the Sk will fall into that slot. Therefore, the situation is identical to part a). Note that NO comparison is needed to find the home slot; that's calculated directly via the hash function.

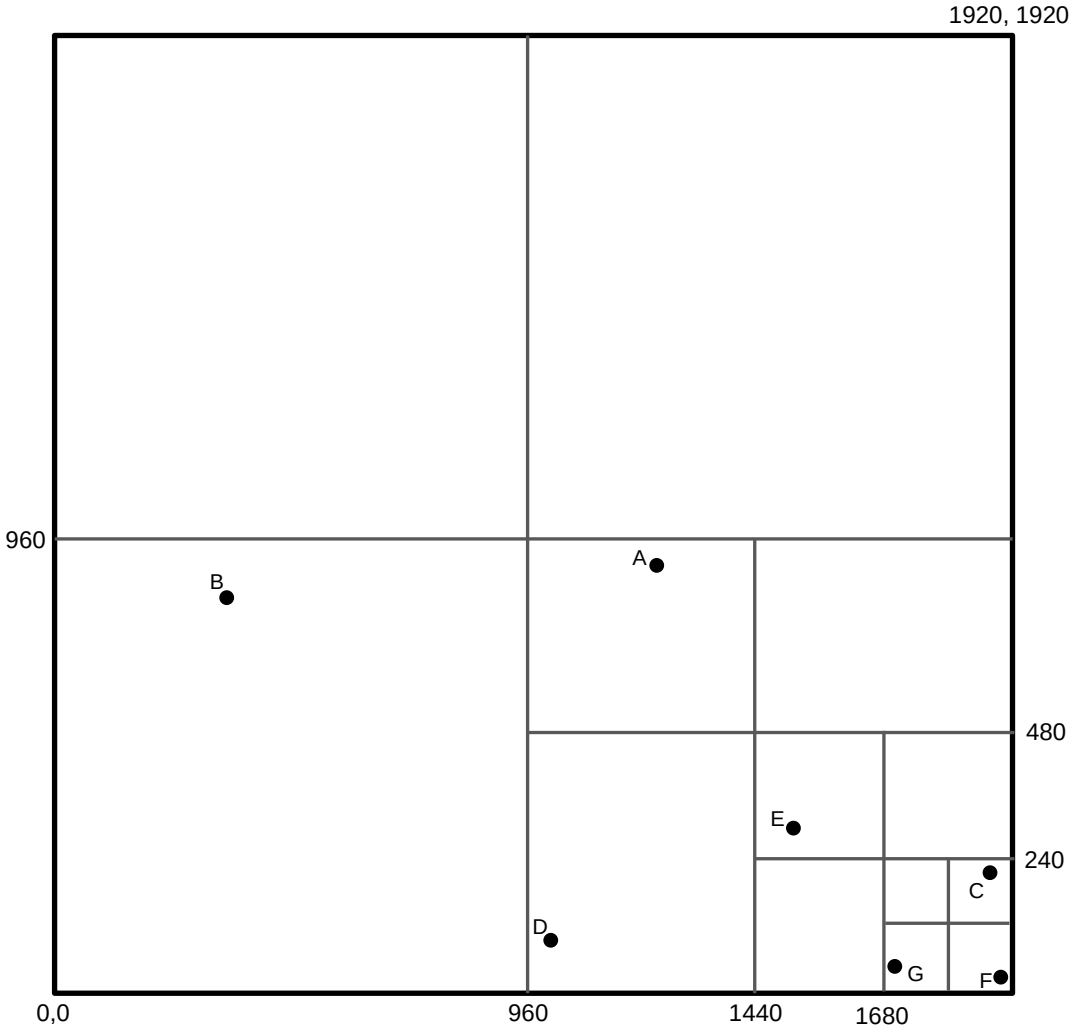
4. [14 points] A full high-definition image has resolution 1920 x 1080 (width x height). In video there is often very little change in an image from one frame to the next. Many video codecs take advantage of this by only storing the regions of an image that has changed between frames.

Since PRQuadTrees require a square region you must set the world coordinate boundaries for the image data below to be (0, 0) .. (1920, 1920).

Draw the PRQuadTree, labeled appropriately, that would result from storing the following pixel coordinates. Assume that each leaf stores a single element. You may draw a tree (with nodes and pointers) or a 2D diagram showing the partitions and the data point labels. We suggest doing the latter.

Insert values in this order:

A(1244, 880), B(440, 810), C(1880, 220), D(1000, 100), E(1500, 300), F(1900, 30), G(1700, 50)



5. [14 points] Write a recursive member method for the prQuadTree class from project 3 to return the number of levels in the tree. The only relevant things from the prQuadTree implementation are that it has a member named `root`, and three node types (`prQuadNode`, `prQuadInternal` and `prQuadLeaf`).

```
public int levels() {

   return levelsHelper( root );
}

private int levelsHelper( prQuadNode t) {

   if (t == null)
         return 0;

   if (t.getClass().getName().equals("prQuadTree$prQuadLeaf"))
         return 1;

   prQuadInternal p = (prQuadInternal) t;
   int nwHT, neHT, seHT, swHT;

   nwHT = levelsHelper( p.NW );
   neHT = levelsHelper( p.NE );
   seHT = levelsHelper( p.SE );
   swHT = levelsHelper( p.SW );

   return (1 + Math.max( Math.max(nwHT, neHT), Math.max(seHT, swHT) ) );
}
```

Problem analysis:

- this problem is a simple extension of the BST levels() method from program 2
- the main difference is that there are twice as many subtree levels to compare

6. Consider the implementation and performance issues related to Pugh's probabilistic skiplist.

a) [8 points] Suppose that a skiplist is built, storing 1024 data elements, and that the ideal proportions of node levels is achieved. That is, about half the nodes are level-0, about one-fourth are level-1, about one-eighth are level-2, and so forth.

Despite this, it turns out that there is a data element in the skiplist for which searches require 400 comparisons. Explain how that could have occurred.

Although the ideal proportions may have been achieved there is no guarantee that they have been distributed ideally. The 512 nodes of the first level could potentially have ended up in the first half of the list. Likewise the 256 nodes of the second level could also potentially have ended up in the first quarter of the first half of the list. Continuing with this unfortunate randomness, the 128 nodes of the third level could also potentially have ended up in the first eighth of the first quarter of the first half of the list and so on.

Thus starting at the top level the search would traverse only slightly down the list, (1 or 2 nodes perhaps), before dropping down. The next to the top level would likewise only traverse slightly father down the list. By the time the lowest level is reached the search would have degraded into almost a single-linked list's linear performance.

b) [8 points] Compare the search performance of the skiplist to a BST storing the same data elements.

Average Cost:

In terms of searching, the average cost for a skip list will be the same as a balanced binary search tree, i.e., O(log N).

Worst Cost:

For a BST ordered insertion would force the tree into a degenerate, skewed left or right stalked single linked list with O(n) performance for search.

For a skip list encountering the unfortunate random assignment of levels described previously it would also degenerate into essentially single linked list O(n) performance for search.