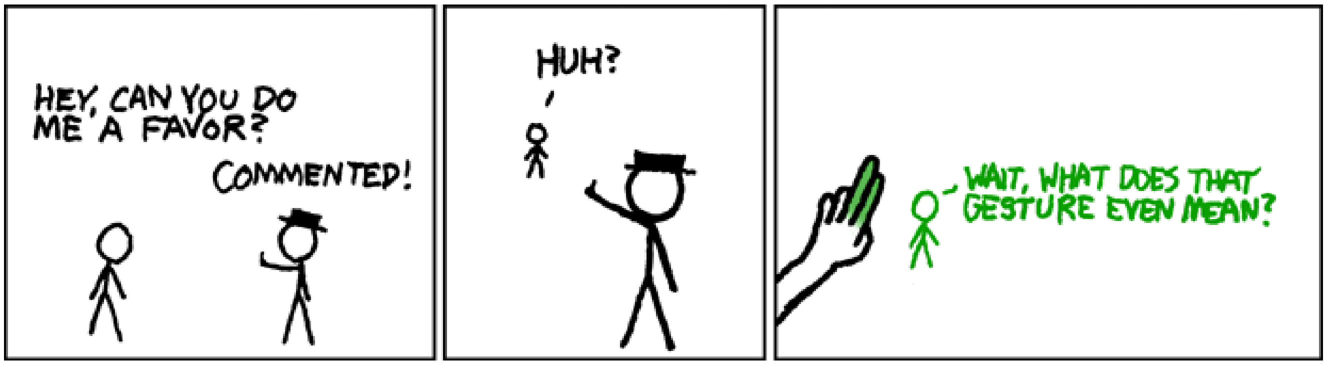
READ THIS NOW!

- Print your name in the space provided below.
- There are 7 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. **[18 points]** Complete the implementations of the public function and private helper function described below, which are intended to return the number of left child nodes in the tree; i.e., the number of nodes that are left children of their parent node.

The only relevant declarations from the BST implementation are that it has a member named `root` and an inner `BinaryNode` class, with `BinaryNode` (reference) members `left` and `right`, and a member `element` of generic type. You may not invoke other tree methods.

You may not modify the given function interfaces in any way!

```
// Indicate any data members you'd add to the BST class here:

private int nLeftChildren;
```

Since the helper function is void, the problem cannot be solved without adding a data member to use as the counter. Of course, the counter should be private.

```
/** Reports the number of left child nodes in the BST.
 *
 *  Returns:
 *        the number of left child nodes in the BST
 */
public int numLefties( ) {

   // Write body of the public function here:

  nLeftChildren = 0;

  numLeftiesHelper( root );

  return nLeftChildren;
}
```

The counter needs to be reset to 0 for each call.

```
private void numLeftiesHelper( BinaryNode sRoot ) {

  if ( sRoot == null ) return;     // nothing of interest here, or below

  if ( sRoot.left != null ) {          // if have a left child,

     nLeftChildren++;                 //    count it

     numLeftiesHelper( sRoot.left );  //    examine its subtree
  }

  numLeftiesHelper( sRoot.right );    // examine right subtree, if any
}
```

In the helper function, there are a few points of interest:

- it's absolutely necessary to have a return in the null test (if it's at the beginning), or to otherwise skip the rest of the body
- since you have to check for a non-null left pointer, the recursive call to the left side should be avoided if the left pointer is null (without a repeated check for that)

2. The design discussion for a PR quadtree implemention resulted in the suggestion that making leaf nodes buckets, capable of holding more than one data object might improve the performance of some operations on the tree.

a) **[6 points]** What would (allegedly) be gained by using a bucketed leaf nodes? Be complete and precise.

Using buckets will reduce the length of some branches in the tree, since a leaf will not split until we find enough data points in its region to overflow the bucket.

This will help with search cost, and therefore with insertion and deletion as well.

I expected an answer to mention that some branches would be shorter, AND that therefore insert/search/delete would all be more efficient (on those branches).

Haskell Hoo V decided to follow this suggestion. In doing so, he decided that he would have each leaf node contain its own PR quadtree, using bucket size 1. Those PR quadtrees would then be used as the buckets.

b) **[8 points]** How should Haskell Hoo V have set the world boundaries for each of these PR quadtree buckets?

Since the PR quadtree bucket represents the region corresponding to the leaf node that contains it, the "world" for these PR quadtrees will be determined by the region boundaries of the leaf nodes that contain them.

c) **[6 points]** Describe the effect of Haskell Hoo V's approach on the performance of operations in his PR quadtree (versus simply having each leaf node store a single data object).

Well, each of these PR quadtrees will mimic the subtree that we would have had if we'd simply used a bucket size of 1 in the original tree.

There's no advantage whatsoever. In fact, it seems there's a slight disadvantage since we have to absorb a little more storage (root pointer for each of these trees), and a little more coding (to switch from our normal traversal pattern to accessing the root of the new quadtree).

However... consider how the tree would be built by a sequence of insertions. When the first point is inserted, we would create a leaf node, containing a PR quadtree that contained the point... when a second point is inserted, it will go into that PR quadtree in the usual manner (for a PR quadtree with a bucket size of 1)... and from there onwards we will just be modifying that PR quadtree... so Haskell seems to have missed the point entirely.

In short, this is a profoundly stupid idea... missing the point of having buckets entirely. I expected an answer to address the actual effect on the tree with some precision.

3. **[12 points]** According to Samet's paper on spatial data structures, the maximum number of levels in a PR quadtree can be as high as

$$\left\lceil \log\left(\frac{\sqrt{2}S}{D}\right) \right\rceil$$

where S is the length of the side of the "world" and D is the smallest distance between any pair of points stored in the quadtree. So, for example, if the world is 1024 x 1024 and there's a pair of points in the tree that are a distance of 4 apart, the number of levels in the tree could be as large as 9.

Explain why, in the situation described above, it is possible that there could be much fewer than 9 levels in the tree. Be very precise. An answer like "well, that's just an upper bound" will not receive any credit. (A diagram might be very useful in answering this question.)

Simply put, we get the worst case only if the two closest data points are not naturally separated by a region boundary that resulted from an earlier split.

For example, if we have two points in the NE and SW quadrants of the world, they can be arbitrarily close to the world center and never require more than one splitting to separate them:

```
+--------------------+
|         |          |
|         |          |
|         |          |
|         |B         |
|---------|----------|
|        A|          |
|         |          |
|         |          |
|         |          |
+--------------------+
```

An answer needed to explain (or show) how it's possible to have a very small value for D (i.e., a pair of data points that are very close together) and still have a relatively shallow quadtree.

There were some interesting, false assertions in the answers:

- there do not have to be a lot of insertions to (potentially) create a very stalky branch; two insertions will achieve that with the right coordinates
- it is, therefore, irrelevant how many leaf nodes are in the tree, or how many data points have been inserted (aside from needing 2)

4. **[12 points]** Suppose a hash table, using linear probing to resolve collisions, is initially empty, and records S1, S2, …, and SM are inserted into the table, in that order, that all of those records have different key values, and that all of those records map to different home slots.

After 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, which is not the home slot of any of the Sk inserted earlier. The size of the hash table is much larger than M + N. What is the maximum number of record comparisons that could be performed in a search for the record RN? Justify your answer.

The S-records all map to different home slots, so they are inserted without any collisions, and wind up filling M slots in the table.

When R1 is inserted, it goes into its home slot, without any collisions.

When R2 is inserted, it collides with R1, so we probe. Now, this could result in R2 colliding with every one of the S-records, but we could not probe linearly and reach R1 again. So, the worst case for R2 is that we get M + 1 collisions when we insert it, and look at M + 2 slots.

If that happens, then inserting R3 will require looking at M + 3 slots (R1, every S-record, and R2)....

... and so forth...

So, the worst case for RN would be colliding with R1, every S-record, and R2 through RN-1, which would amount to M + N – 1 collisions, and looking at M + N slots.

In that case, a search for RN would require M + N comparisons.

A complete answer will explain HOW it's possible to achieve the alleged maximum, not just state what the maximum is.

5. **[10 points]** A developer is designing a generic hash table, planning to use chaining rather than probing to resolve collisions, and to implement her design in Java. Her supervisor suggests that she should use a BST in each table slot (instead of a simple linked list), because that would be likely to improve the cost of searches that reached a slot holding more than one data element.

She considers the suggestion, and responds by saying that using BSTs in this manner may not be possible.

Is she correct? Explain.

Yes. To insert an object into the hash table, that object must supply a hash method. Since every class in Java has an inherited hash method (even if it may not be very good), every Java object can be inserted into a hash table.

But, to insert an object into a BST, the object must implement compareTo(), or something equivalent, and that is not something that every Java object will do.

Pretty straightforward... the need for Comparable is the ONLY issue here.

6. Double hashing resolves collisions by probing, using a second hash function to choose the probe slots. Specifically, if the key *K* experiences a collision in its home slot *h*, we use the following formula to compute indices for probe slots:

$$Slot(i) = (h + i \cdot G(K))\%N$$

where *G()* is the second hash function and *N* is the size of the table. Assume that *G()* is actually 1-1 and nonzero on the set of possible key values.

a) **[8 points]** Explain why double hashing, when performing a single insertion, can be viewed as a variation of linear probing.

With double hashing, we take a step of fixed length, G(K), at each probing step.

With linear probing, we do exactly that, except the step length is always 1. With double hashing, the step length is likely to be different for different keys.

A lot of answers basically said that all probing strategies have certain things in common, which is true but not relevant to this specific question.

b) **[8 points]** Linear probing is guaranteed to find an empty table slot, if there is one. Explain why, even given the assumptions above about *G()*, this scheme may not find an empty slot, even if there is one.

Naturally, if G(K) == 0, we'd just hop up and down in the home slot... but the given assumptions rule this out.

But, if we have a key such that G(K) > 1 and G(K) divides the size of the table, N, then we might cycle back to the home slot after examining only a few of the table slots.

In fact, it's worse than that. G(K) could be larger than N, but G(K) % N could be a divisor of N. Or, G(K) could even be a multiple of N, which would yield the "hopping in place" behavior described above.

So, it's important that the table size be prime, but even then we may have problems.

A good answer would address one or more specific ways that this scheme could fail, not just assert it was possible.

7. **[12 points]** Consider the BST deletion code below. There is no requirement for a node pool in this implementation, and there are no duplicate values in the tree.

```
public void delete(T elem) {

   if ( elem == null ) return;
   root = deleteHelper(elem, root);
}

private BinaryNode deleteHelper(T x, BinaryNode sroot) {

   if ( sroot == null ) return null;

   int compare = x.compareTo(sroot.element);

   if ( compare < 0 ) {
      sroot.left = deleteHelper(x, sroot.left);
   }
   else if ( compare > 0 ) {
      sroot.right = deleteHelper(x, sroot.right);
   }
   // found the node to delete
   else  {
      // handle leaf case
      if ( sroot.left == null && sroot.right == null ) ) {
         return null;
      }
      // handle single-child case
      else if (sroot.left == null) {
         return sroot.right;
      }
      else if (sroot.right == null) {
         return sroot.left;
      }
      // handle two-child case
      else {
         BinaryNode rMin = findSubTreeMin(sroot.right);
         deleteHelper( rMin.element, sroot);
         rMin.left = sroot.left;
         rMin.right = sroot.right;
         return rMin;
      }
   }
   return node;
}
```

The deletion implementation works correctly, except for one logic error in the handling of the two-child case. The helper function `findSubTreeMin()` is implemented correctly, so it is not the problem. Describe the logic error in the given code, and draw a tree that would trigger that error.

This would fail when rMin points to the right child of sroot, if I had remembered to put sroot.right here. In that case, the right child pointer in rMin actually ends by pointing to rMin.