1. **[10 points]** Suppose that we insert 200 elements into a previously empty hash table, using an array of dimension 500. Suppose that exactly 10 of those records have slot 42 as their home slot. We have no information about the home slots of the other records, except that none of them have slot 42 as their home slot.

Suppose that the hash table resolves collisions by linear probing. If we now perform a search for one of those 10 records whose home was slot 42, what is the maximum number of table slots that might be accessed? Explain.

Answer:

We know that slots 42 - 51 will be filled, because the 10 records that collide in slot 42 would probe through those slots. However, some of the other 190 records might have home slots in the range 43 - 51, or have probed into that range of slots due to collisions of their own.

Suppose that the other 190 records all collided in slot 43, and they were inserted before any of the 10 records that collide in slot 42 were inserted. Then the first of those 10 records would fill slot 42, and the other 9 would probe all the way past the other 190 records, and then fill the next 9 slots after that.

In that case, if we searched for the last record that was inserted, we'd start in slot 42, score a miss, and then probe all the way to the end of a run of 199 slots to find a match.

So, we might have to examine 200 slots.

That's clearly the worst case, because there are only 200 filled slots in the table.

Many answers failed to address just how the elements were placed in the hash table; without that discussion, there's no way to actually explain how it could be necessary to examine 200 slots (and hence to 200 comparisons).

There are many other arrangements, besides the one I described above, that would achieve the worst case.

2. [12 points] Haskell Hoo IV is designing and implementing a hash table, and is pondering how to deal with primary collisions (i.e., cases where two different key values are hashed, the hash value is modded by the table size, and the two different keys get the same home slot).

Being aware of the limitations of the various probing strategies we have discussed in our course, he proposes the following approach to probing. At each probe step, add to the index of the current table slot the index of the key's home slot (and mod by the table size of course).

Assume that the table size is sufficiently large to be of no concern. Describe two different things that could prevent Haskell's idea from working. Be specific. If you describe more than two shortcomings, only the first two will be evaluated.

Answer:

Note: this can be viewed as a variation of either linear probing or double hashing, since we take a fixed-size step when probing, but the step size is determined by the home slot index, not by hashing a key.

The index of the home slot could be 0; in that case Hoo's strategy would just jump up and down on the home slot.

The index of the home slot might be a divisor of the table size. In that case, the probing would cycle back to the home slot after examining only a potentially-small set of table slots.

Hoo's scheme is not guaranteed to examine all the table slots, but neither are a number of other probe schemes. As long as there's a sufficiently long period, the scheme will likely work adequately. OTOH, the second flaw described above could yield an extremely short period.

The key here is that the question asks about flaws that would "prevent Haskell's idea from working". There are certainly other objections that do not actually prevent the idea from working.

Two keys that collide in the same slot will then generate the same probe sequence from that slot. That's also true of linear and quadratic probing, but not of double hashing. However, that shortcoming doesn't prevent the scheme from working any more than linear and quadratic probing cannot work. It just means that there will be clustering. That's undesirable, but not fatal.

3. [14 points] Determine the exact count complexity function T(N) of the following algorithm. Your answer for T(N) should be in simplest form. Show supporting work!

```
for (i = 1; i <= N; i++) {          // Line  1
   x = 0;                           // Line  2
   y = i + N;                       // Line  3
   for (j = 1; j <= 2*i; j++) {     // Line  4
      if ( i % j < 2 ) {            // Line  5
         x = y + j;                 // Line  6
         y--;                       // Line  7
      }
      else {
         x = 2*y - j;               // Line  8
      }
   }
}
```

Answer:

Analyzing each line of the given code, we get these costs:

Line 1: 1 before loop; 2 per pass; 1 to exit outer loop

Line 2: 1 per pass of outer loop

Line 3: 2 per pass of outer loop

Line 4: 1 per pass of outer loop (to init j); 3 per pass of inner loop; 2 to exit inner loop

Line 5: 2 per pass of inner loop

Lines 5-6: 3, if done

Line 8: 3, if done

$$T(N) = 1 + \text{Sum}(i = 1 \text{ to } N)\{ 2 + 1 + 2 + 1 + \text{Sum}(j = 1 \text{ to } 2i)\{ 3 + 2 + 3 \} + 2 \} + 1$$

$$= \text{Sum}(i = 1 \text{ to } N)\{ \text{Sum}(j = 1 \text{ to } 2i)\{8\} + 8\} + 2$$

$$= \text{Sum}(i = 1 \text{ to } N)\{ 16i + 8\} + 2$$

$$= 16N(N+1)/2 + 8N + 2$$

$$= 8N^2 + 16N + 2$$

Many students seemed to assume that, because there was a 2*i in the outer loop, that the number of passes would be logarithmic. That is incorrect.

Instructions for Questions 4 and 5:

Suppose that the same set of 50 integers, without duplicates, were inserted into a BST and an AVL tree, not necessarily in the same order, and no deletions were performed on either tree.

4. **[7 points]** The cost of performing one single rotation or one double rotation after an insertion in an AVL tree is O(1). Explain why this is true. (You do not need to describe the rotations in detail; a one-sentence answer is possible.)

Answer:

Each rotation requires checking/resetting a small, fixed number of pointers (3 for a single rotation, 5 for a double rotation), and resetting some balance factors (2 for a single rotation, 3 for a double rotation). That implies an upper bound that's a small constant number of fixed-cost operations.

So the cost of a rotation is O(1).

The most common issue here was lack of precision about the cost of each type of rotation.

5. **[7 points]** Could the BST have fewer levels than the AVL? Justify your conclusion.

Answer:

This one is a bit tricky. The AVL does restructure itself in a way that reduces the number of levels in the tree, versus a BST… if the elements are inserted in the same order.

But if the order of insertion is not the same, all bets are off.

Here's the BST we'd get if we inserted 40, 20, 50, 10, 30, 60, 70 in that order (3 levels):

On the other hand, this tree has the AVL property, with the same keys as the tree above, but has 4 levels:

This was obtained by inserting the keys in the order 50, 20, 60, 10, 40, 70, 30; BTW, that triggered no rotations.

The common error was to simply assert the AVL cannot have fewer levels than the BST.

A common false statement was that an AVL tree has the minimum number of levels for the number of nodes it contains… that is also false.

An alternate explanation, that was acceptable, was to point out that the AVL is only guaranteed to have no more than 1.44 log(N) levels while a BST can have as few as 1 + log(N) levels. Those bounds, if correct, do imply that a case like the one I constructed is possible.