Solution

Q1. [12 points] Suppose we have a BST containing numerous values, including two values, X and Y, so that Y is the left child of X, and Y is in a leaf node.

Could there be a value, Z, which was inserted into the BST after X was inserted but before Y was inserted, such that Y < Z < X?

If yes, explain where the node containing Z might be, in relation to the node containing X. If not, explain why not.

Answer 1:

We have to consider what could have happened when Z is inserted, given that X is in the tree, Y has not been inserted yet, but Y will become the left child of X, and Y will be a leaf, and that Z was inserted after X but before Y.

We know at that when Z was inserted, X had no left child (otherwise there is no way that X could have had a leaf as its left child, holding Y).

So, if the descent for the insertion of Z reached X, Z would have become X's left child; but that's impossible since there would then be no way for Y to become X's left child (unless you want to hypothesize that Z was deleted before Y was inserted, which is sneaky).

Therefore, Z could not have reached X… which means Z must have gone left (since Z < X) before reaching X. For that to occur, Z must have reached a value, call it W, along the path from the root to X such that Z < W. But, if so, then Y would have reached the same value and also have gone left before reaching X, another contradiction of the given facts.

So, aside from the hypothesis of an unmentioned deletion, no such value Z could exist in the tree.

Answer 2:

Suppose such a Z exists in the tree. Then, in an inorder traversal of the tree, we must encounter the three values in the order Y . . . Z . . . X, possibly with intervening values between Y and Z and/or between Z and X.

But, Y is the immediate left child of X, so the only way we could encounter Z after Y but before X would be if Z lies in the right subtree of Y.

However, Y is a leaf, so the right subtree of Y must be empty.

Therefore, there can be no such value Z in the tree.

For Q2 and Q3, consider the following Java code; assume there is a full implementation of the BinaryTree class, and that the implementation does not allow the insertion of a duplicate value.

```
public class BinaryTree<T extends Comparable<? super T> > {

   private BinaryNode {

      T          element;
      BinaryNode left;
      Binarynode right;
      . . .
   }

   private BinaryNode root;

   . . .

   public boolean isBST( ) {

      return isBSTHelper( root );
   }

   private boolean isBSTHelper( 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) );
   }
}
```

First of all, there were two typos in the given code (my fault for not compiling). The two recursive calls should have been to isBSTHelper(), not to isBST(). If you noticed that, and answered accordingly, that's fully acceptable.

Q2. [12 points] Could the given function ever return true for a `BinaryTree` object that did not have the BST property? Justify your answer.

Answer:
Yes. The flaw in the function is that it only compares parent to children. The function would say the following tree is a BST:

Mark Twain once said, in an entirely different context, "few things are more annoying than a good example". One simple example was all that was required here. Some answers consisted of unclear descriptions of the flaw, when a simple example would have nailed the question.

Q3. [12 points] Could the given function ever return false for a `BinaryTree` object that did have the BST property? Justify your answer.

Answer:

No. The function will only return false if there's a case where a parent has a value that's smaller than its right child or larger than its left child, which means no BST.

Many answers focused on the question of duplicate values in the tree. Since the instructions for these two questions clearly said the function was intended for a BST implementation that did not allow the insertion of duplicate values, the issue of duplicates was irrelevant.

A lot of answers talked about when the function would return true, without actually addressing the question at hand: could the function every say "false" incorrectly?

Q4. [12 points] Consider the implementation of a hash table that uses linear probing to resolve collisions. 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).

We would like to improve the cost of searches for the element that's being inserted.

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_3), and leaving slot s_4 empty?

Answer:

This could break subsequent searches: if the element that's in s3 is in its home slot, a search won't find that element unless it probes all the way around the table.

Q5. [12 points] Suppose that, in a hash table that uses linear probing to resolve collisions, exactly four elements have slot 100 as their home slot. Suppose that 1000 other elements have also been inserted to the hash table (but none of those have slot 100 as their home slot).

Suppose we now search the hash table for one of the elements whose home slot is 100. What is the maximum number of element comparisons that might be required during that search? Justify your conclusion.

Answer:

Suppose one element, with home slot 100, is inserted.

Suppose each of the other 1000 elements have slot 101 as their home slot (unlikely, but possible), and suppose those 1000 elements were inserted before the other 3 with home slot 100. Then those elements would fill slots 101 – 1100.

Then, when the other 3 were inserted, they would fill slots 1101 – 1103.

Then, a search for the last of those elements would probe every slot from 100 to 1103 before scoring a hit. That's 1004 slots to be examined, and each requires comparing two elements.

For Q6, Q7, and Q8, assume that you have a collection of data objects that correspond to points with real coordinates (like 3.1459), in the square region of the xy-plane bounded by x = 0.0, y = 0.0, x = 1024.0, y = 1024.0. Assume that you will organize the data objects in a PR-quadtree (with bucket size 1).

In Q6 and Q7, the term "distance" refers to the actual Cartesian distance, not the taxicab distance.

Q6. [12 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, Y, is inserted into the tree, and does cause a region to split. What is the maximum distance between Y and the closest data object that was already in the tree? Justify your conclusion.

Answer:
To force a split, the new element must fall in the same leaf as an existing element. Each leaf represents a square that is 8x8 (smaller).

The maximum separation for two points that fall in an 8x8 square is achieved if the points are at opposite corners of the square, and the diagonal of an 8x8 square is sqrt(8^2 + 8^2) units.

Simplifying (not necessary for this question): 8 sqrt(2).

Technically, this is the least upper bound for the maximum distance. Depending on how region boundaries are handled, two opposite corners may not actually belong to the same region. In that case the maximum separation is strictly less than 8 sqrt(2).

Q7. [12 points] Suppose we have inserted 42 data points into the PR-quadtree. What is the minimum number of levels the tree might have? (An empty tree contains 0 levels.) Justify your conclusion.

Answer:
A PR quadtree node has no more than 4 children, so the maximum number of nodes in level k is 4^k (remember, the root is in level 0).

We need 42 leaves to hold the data point. With 3 levels, the bottom level would only contain 4^2 nodes; with 4 levels, the bottom level would contain 4^3 nodes.

So, there would have to be at least 4 levels in the tree.

Q8. [12 points] After inserting all the data objects into the tree described above, we discover that they all actually lie within the square bounded by x = 256.0, y = 256.0, x = 512.0, y = 512.0. This region is much smaller than the one described originally. Describe the root node and its children.

Answer:

The square in question is the NE quadrant of the SW quadrant of the given world.

So the NW, NE, and SE quadrants of the world are empty; so the root node has only one child (an internal node to the SW).

(And, the SW node has only one child, to the NE.)

Q9. [4 points] In the optimal case, each of the following data structures would have the same average cost for performing a search for a specific value:

- an array that stores elements sorted into ascending order
- a binary search tree that has the minimum number of levels (for the number of nodes it contains)

A sorted array is simpler to implement than a binary search tree. So, why would we ever want a binary search tree in preference to a sorted array?

Answer:

If we never do insertions or deletions, there's no disadvantage to using a sorted array.

But insertion or deletion, except at the tail of the array, would require shifting elements in the array. We could argue an insertion or deletion could, on average, require shifting half the elements in the array. (It could be worse or better, depending on the pattern of insertions and deletions.)

For the minimum-height BST, insertion would require descending through about log(N) levels and hanging a new leaf.

And deletion could be more expensive, but would only touch nodes in the branch containing the node to be deleted.

So, for situations where insertions and deletions play a significant role, the BST would be more efficient (assuming it is minimum-height).

Some answers raised the cost of storage. The conclusion there is less clear. If using an array, and having to allow for a uncertain number of entries, you'd either make the array large, hoping to guess right, or you'd eat the cost of making a larger array and transferring data into it. But the BST uses 8 bytes for each pointer. That adds up, and is especially bothersome if the user's data objects are small.