Virginia Tech 1872

READ THIS NOW!

- Print your name in the space provided below.
- There are 5 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. However, you may include examples from the course notes.
- No calculators, cell phones, 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

YOU'LL NEVER FIND A PROGRAMMING LANGUAGE THAT FREES YOU FROM THE BURDEN OF CLARIFYING YOUR IDEAS.
BUT I KNOW WHAT I MEAN!

xkcd.com

Answers to questions are in blue.

Commentary about the questions and answers is in green.

1. Suppose we have a BST and that two values, X and Y, have been inserted into the tree so that Y is the left child of X, and Y is in a leaf node. Could there be a value, Z, which was also inserted into the BST such that $Y < Z < X$? Let's consider some (but not all) cases; what can you conclude about each of the following scenarios? (The BST may contain other values as well, but not duplicates.)

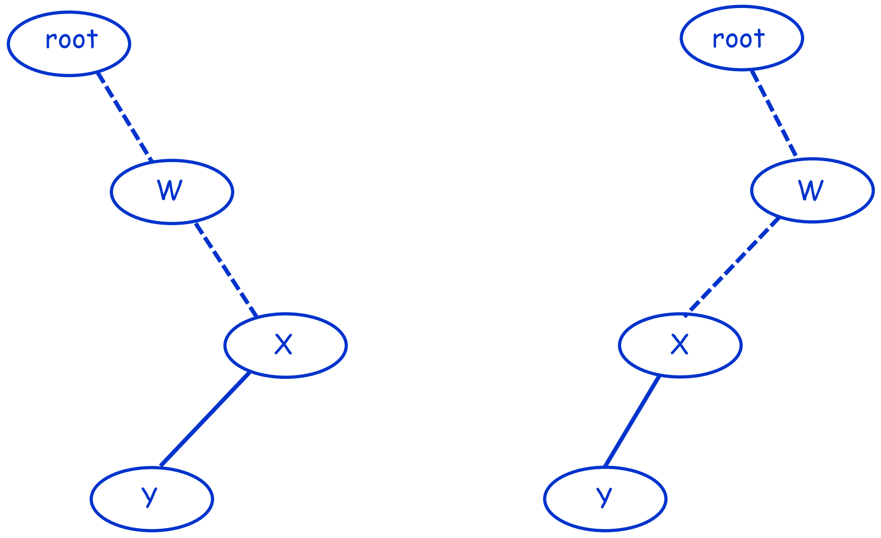
So, what we know is that two values, X and Y, are in the BST, Y is the left child of X, and Y is a leaf. And, since Y is a child of X, X was inserted before Y. Those are absolute facts. The question is: could there be another value Z, that is somewhere in the BST, such that $Y < Z < X$?

a) [10 points] Suppose that Z was inserted sometime after both X and Y were inserted.

IF we know that the insertion of Z must reach X, then Z would go to the left at X, since $Z < X$. Then, reaching Y, Z would go to the right, meaning that Y could not be in a leaf.

That is a contradiction, and therefore there can be no such value Z such that Z was inserted after both X and Y and $Y < Z < X$.

But… must the insertion of Z reach X at all? Could Z have taken a different direction during its insertion, and not be in X's left subtree? Suppose there is some value, say W, along the path from the root to X such that Z went in a different direction at W than X did:



Now, if Y went right at W, then so will Z, since $Z > Y$. And, if X went left at W, then so will Z, since $Z < X$. Therefore, the insertion of Z must reach X.

b) [10 points] Suppose that Z was inserted sometime before either X or Y was inserted.

My intent here was that Z was inserted before both X and Y, but the phrasing allowed the interpretation that Z was inserted after X but before Y. Note though, that X must have been inserted before Y in any case.

This one's easier… if Z was inserted before both X and Y, then X would have gone right at (or above?) Z and Y would have gone left at (or above?) Z, so the arrangement described earlier could not have occurred.

Now, we might ask: could X have gone in a different direction and not have reached Z at all? Interesting question… which I will leave for you to ponder.

If Z was inserted after X but before Y, then Z would have reached X (see argument in part a). And, X could not already have a left child since Y is supposed to wind up being X's left child. Therefore, Z would now become the left child of X, which is a contradiction.

2. Suppose we have a value, X that has been inserted into some data structure. Suppose we delete X from the structure, and then immediately reinsert X into the structure. Could the cost of finding X before it was deleted be different than the cost of finding X after it has been reinserted? If yes, would there always be a difference, or only in certain cases?

The big-O results for the data structures are not relevant here. The question is really simpler than that. You need to consider whether, when X is reinserted, it must wind up in the same spot it was before, or it might wind up in a "better" or "worse" spot.

The question was not, at all, about the cost of the reinsertion of X.

a) **[10 points]** What conclusion follows if the data structure in question was a PR quadtree? Explain clearly.

There will be no difference:

The position of X in the quadtree is determined entirely by the splitting that must be done to separate X from any nearby points (even if buckets are used).

So, X will be reinserted at precisely the same position it was deleted from.

I was picky about this. You needed to offer an explanation of why the insertion process for a PR quadtree will place X back in the same (node) position. I didn't give full credit if you just said "PR quadtree structure is independent of the order of insertion". The question did not specify bucket size. If you took that into account, your conclusion should have been that the search cost might be a little higher since X might move to the end of the bucket.

b) **[10 points]** What conclusion follows if the data structure in question was a BST? Explain clearly.

There may or may not be a difference:

X will be reinserted in a leaf node.

If X was in an internal node of the BST, then X will require a longer search.

On the other hand, if X was in a leaf node, reinserting X will put X in a leaf node in precisely the same position, so the search cost will be the same.

c) **[10 points]** What conclusion follows if the data structure in question was a hash table that uses probing to resolve collisions? Explain clearly.

There may or may not be a difference:

When X is deleted, its position will become a tombstone.

When X is reinserted, the probing logic could put X into an earlier slot in the probe sequence (if tombstones are recycled, and there is an earlier tombstone in the probe sequence), or into the same slot it occupied before (if tombstones are recycled, but there's no earlier tombstone in the probe sequence), or into a later slot in the probe sequence (if tombstones are not recycled, although that's unlikely).

Tombstones MUST be used on deletion operations if the hash table uses probing, no matter what form of probing is used. And, tombstones will always be recycled because doing so costs you an insignificant amount of extra work, but also makes some future searches more efficient.

3. **[10 points]** Suppose we have a PR quadtree that is storing data objects with coordinates in the range [0, 128], and that subtree currently looks like Figure 3A below. The node labelled Parent is an internal node; it may or may not be the root of the tree. Now, suppose that the data object C is deleted from the tree, and the implementation results in a branch contraction, making the subtree look like Figure 3B below.

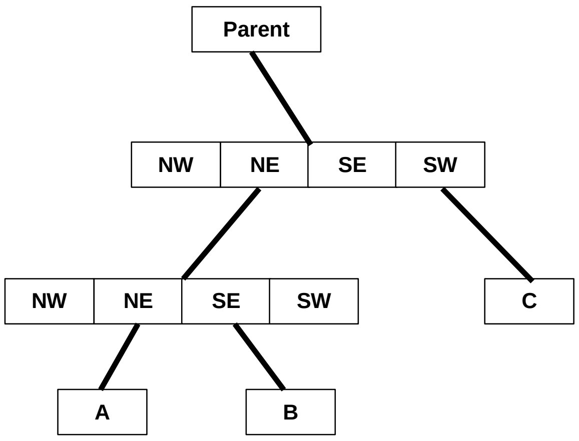


Figure 3A

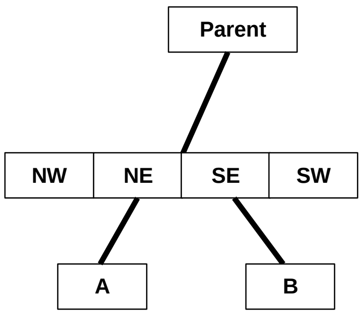
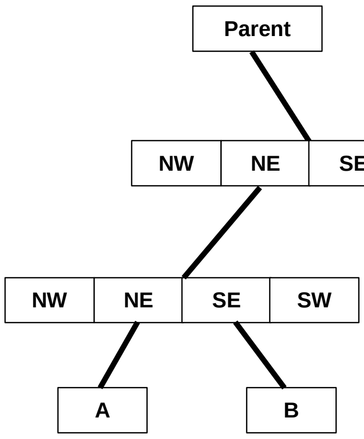


Figure 3B

Is this correct? If not, explain why it is wrong, and draw the subtree (below) as it should look after the deletion. If this is correct, carefully explain why. Either way, your explanation should refer to regions and partitioning.

This is incorrect. B lies in the SE quadrant of the NE quadrant of the parent of C. B does not lie in SE quadrant of the parent of C. The resulting tree should look like this:



4. Suppose you have a large collection of data values. You are considering storing the data values in a data structure, and are trying to decide between a hash table and a minimum-height BST (which you could build since you have all the data values). You are convinced that no matter which data structure you choose, you can achieve its theoretically optimal performance. You can also implement any additional public methods that seem helpful, no matter which data structure you choose.

There were a couple of common misconceptions in the answers:

Some answers said things like "hash tables are not good for strings" or "hash tables are not good for integers". I have no idea where such ideas came from.

Many answers assumed that the data for the search you were performing was available before the relevant data structure was constructed. This was often implicit, as in "let's use a hash table of dimension N" in part a). That doesn't make sense. If you had that information in advance, and you were only going to perform one such search, there would be no reason to use a data structure at all. You'd just traverse the data elements linearly, O(M) cost, which would be cheaper than building either data structure.

Another issue was to only consider the theoretical optimum costs for search in the data structure, and not take into account the nature of the search problem. If the big-O bounds were always sufficient to make the best choice, life would be much simpler. And, considering the specific problem we have can lead to clever ways of employing the structure.

a) **[10 points]** Suppose your data values are integers, and that your primary concern is to support determining which factors of a given integer, N, are in the collection. Which data structure would be the better choice? Explain why.

Suppose there are M integers in the collection.

Given N, we can determine what the factors are; and, the number of factors will be much smaller than M (since M is "large"). Having done that, we can search for each factor in turn. Assuming optimal costs, the cost of each search would be $\Theta(1)$ in the hash table and $\Theta(\log M)$ in the minimum-height BST.

So, the search cost would be better if we used a hash table.

b) **[10 points]** Now suppose your data values are strings, and your primary concern is to support determining whether, for a given string S, there are any strings in the collection that have S as a suffix. For example, “ball” is a suffix of “football” and “tarball”. Which data structure would be the better choice? Explain why.

Suppose there are M strings in the collection.

The difference between this and part a) is that we don’t have any useful way of creating all the possible strings that have S as a suffix.

We could use the inorder traversal pattern on the BST; that would be $\Theta(M)$.

We could also traverse the array holding the hash table; that would be at least Θ(M), and larger unless the table was of minimum size.

That seems to favor the BST, slightly.

OTOH, we could store the strings in the BST after reversing them. In that case, we just have to search the BST for an occurrence of S (which is probably not there), and if there is a string with the suffix, we will find that as well. The cost of this would be Θ(log M).

5. Haskell Hoo IV is attempting to implement a BST generic that conforms to the specification for Project 2. Being concerned that his logic may be untrustworthy, he has written the private member function shown below. He plans to call it at the end of each insertion and deletion operation while he is testing his BST.

```
// Pre:      sroot is null or points to the root node of a BST
// Post:     the BST is unchanged
// Returns:  true iff the BST ordering property is NOT violated somewhere
//              in the tree
//
private boolean hasBSTProperty( 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 hasBSTProperty( sroot.left ) && hasBSTProperty( sroot.right );
}
```

One strange error was to say the function is wrong because it would classify an empty tree as a BST; an empty tree is a BST, so is a one-node tree.

Another common error was to worry about duplicate entries; the BST from Project 2 did not allow duplicate entries.

Another common error was to simply misinterpret the Java code. For instance, the test in the first if makes sure there IS a left child, and then checks whether the element in the parent node is less than or equal to the element in its left child. If so, you don't have a BST. The logic there is correct. And, there are appropriate NULL tests everywhere one is needed. And, there are no syntax errors in the code.

a) **[10 points]** Could Hoo's function ever return false even though his tree did have the BST ordering property? That is, could Hoo's function ever report a false negative? Justify your answer; if yes, use a specific example of a BST to illustrate; if no, explain exactly why.

The only way that the function will ever return false would be to find a node holding a value, X, such that the value in the left child was larger than X or the value in the right child was smaller than X. Either way, we would not have a valid BST.

Therefore, no, the function will never report a false negative.

b) **[10 points]** Could Hoo's function ever return true even though his tree did not have the BST ordering property? That is, could Hoo's function ever report a false positive? Justify your answer; if yes, use a specific example of a BST to illustrate; if no, explain exactly why.

The flaw in the function is that the checks are entirely "local"; that is, each node is only checked against its immediate children. The function would return true for the following tree, even though the BST property is violated: