# Quiz 1 Solutions

**Problem 1. Asymptotic orders of growth** [10 points] (4 parts)

For each pair of functions, circle the asymptotic relationships that apply. You do not need to give a proof.

**(a)** $f(n) = \sqrt{n}$
$g(n) = \log n$
Circle all that apply:

$f = O(g)$ $\qquad$ $f = \Theta(g)$ $\qquad$ $f = \Omega(g)$

**Solution:** $\log n$ grows more slowly than any polynomial in $n$, so $f = \Omega(g)$.

**(b)** $f(n) = 1$
$g(n) = 2$
Circle all that apply:

$f = O(g)$ $\qquad$ $f = \Theta(g)$ $\qquad$ $f = \Omega(g)$

**Solution:** All constants are related to each other by a constant factor, so they have the same order of growth. $f = \Theta(g)$, and therefore $f = O(g)$ and $f = \Omega(g)$ also.

**(c)** $f(n) = 1000 \cdot 2^n$
$g(n) = 3^n$
Circle all that apply:

$f = O(g)$ $\qquad$ $f = \Theta(g)$ $\qquad$ $f = \Omega(g)$

**Solution:** $2^n/3^n$ approaches 0 as $n \to \infty$. Therefore, the only relation that holds is $f = O(g)$.

**(d)** $f(n) = 5n \log n$
$g(n) = n \log 5n$
Circle all that apply:

$f = O(g)$ $\qquad$ $f = \Theta(g)$ $\qquad$ $f = \Omega(g)$

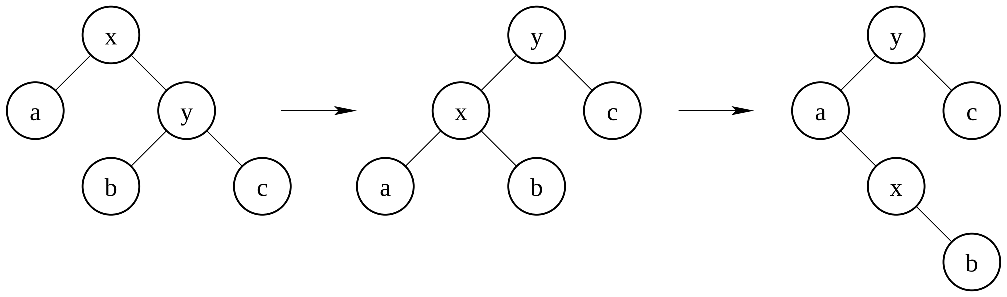
**Solution:** $n \log 5n = n \log 5 + n \log n = \Theta(n \log n)$. Also, $5n \log n = \Theta(n \log n)$. These are the same order of growth, so $f = \Theta(g)$, $f = O(g)$, and $f = \Omega(g)$.

**Problem 2. True or False** [16 points] (4 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

**(a) T F** Performing a left rotation on a node and then a right rotation on the *same* node will not change the underlying tree structure.
*Explain:*

**Solution:** False. The figure below shows what happens in this case. To undo the left rotation on $x$, we must do a right rotation on $y$.



**(b) T F** While inserting an element into a BST, we will pass the element's predecessor and successor (if they exist).
*Explain:*

**Solution:** True. The predecessor of a node is either the maximum element of its left subtree, or one of its ancestors. A newly-inserted node has no descendants, so the predecessor must be one of its ancestors, and hence the predecessor is on the path that was traversed during the insertion procedure. (A similar argument holds for the successor.)

**(c) T F** For a hash table using open addressing, if we maintain $m = \Theta(n)$, then we can expect a good search and insert runtime.
*Explain:*

**Solution:** False. If the hash table is nearly full, for example if $n = m - 1$, then the runtimes will take $\frac{1}{1-\alpha} = O(n)$.

**(d) T F** If we know ahead of time all the keys that will ever be inserted into a hash table, it is possible to design a hash table that *guarantees* $O(1)$ lookup and insertion times, while using $O(n)$ space.
*Explain:*

**Solution:** True. Use perfect hashing.

**Problem 3. Short Answer** [24 points] (4 parts)

**(a)** Describe an efficient method to merge two balanced binary search trees with $n$ elements each into a balanced BST. Give its running time.

**Solution:** We can start by doing an in-order walk of both trees concurrently. At each step, we compare the two tree elements and add the smaller one into a list, L, before calling that element's successor. When we finish walking both trees, L will contain a sorted list of elements from both trees. This takes $O(n + n) = O(n)$ total time.

Now, from the sorted list, we want to create a *balanced* binary tree, which is the same problem as described in problem set 2. We can do this by setting the root as the middle element of the list, and letting the first half of the list be its left subtree and the second half be its right subtree (recursively creating the balanced subtrees as well). This also takes $O(n + n) = O(n)$ time.

The total time for this algorithm is therefore $O(n)$.

**(b)** Suppose you are given a list of $n$ elements such that the location of each element is at most $\lg(\lg n)$ elements away from the location it would be in if the list were sorted. Describe an $o(n \lg n)$-time method to sort the list and give its asymptotic running time.

**Solution:** Insertion sort. With insertion sort, at iteration $i$, we attempt to move the $i - th$ element into its correct place within the first (0..i) slots. Since an element can be at most $\lg(\lg n)$ positions away from its correct position, the element will be moved at most $\lg(\lg n)$ times. There are $n - 1$ iterations in insertion sort, so the asymptotic runtime is bounded by $O(n \lg(\lg n))$, which is $o(n \lg n)$. Note that the little-oh notation discounted all the $n \lg n$ algorithms.

**(c)** Suppose we have a hash table that resolves collisions using open addressing with linear probing. Slots with no keys contain either an EMPTY marker or a DELETED marker. Alyssa P. Hacker tries to reduce the number of DELETED markers; she proposes to use the following rules in the delete method:

i. If the object in the next slot is EMPTY, then a DELETED marker is not necessary.

ii. If the object in the next slot has a different initial probe value, then a DELETED marker is not necessary.

Determine whether each of the above rules guarantees that searches return a correct result. Explain.

**Solution:**

i. True. If the next slot is EMPTY, then any search would stop at that slot and return false. If we didn't place a DELETED marker, search would stop one slot earlier and still return false.

ii. False. Consider a counterexample. Let $h(k_1, 0) == h(k_3, 0)$ and $h(k_1, 0) + 1 == h(k_2, 0)$. If keys $k_1$, $k_2$, and $k_3$ are inserted into a hash table in that order, then they would hash to consecutive slots in that same order. Now if $k_1$ is deleted from the hash table, but is not marked with a DELETED marker, then search for $k_3$ would incorrectly return false, because the delete method would check slot $h(k_3, 0)$ first and see that it is EMPTY.

**(d)** An open-addressing hash table that resolves collisions using linear probing is initially empty. Key $k_1$ is inserted into the table first, followed by $k_2$, and then $k_3$ (the keys themselves are drawn randomly from a universal set of keys).

i. Suppose $k_2$ is deleted from the hash table and replaced by a DELETED marker. What is the probability that searching for $k_3$ requires exactly three probes?

ii. What is the probability that searching for $k_1$ takes exactly two probes?

**Solution:**

i. $P(k_3 \text{ requires exactly 3 probes}) = 3/m^2$. We treat a DELETED marker like an inserted element during a search or insert operation. The probability we have three probes during a search for $k_3$ is the same as the probability we have three probes while inserting $k_3$. Inserting $k_3$ in the table would require three probes iff the first two probes hit occupied slots. Since we have two keys in the table $k_1$ and $k_2$, they need to be consecutive and $k_3$ should hash to the one which is above of the two. So we have two cases:

   (a) $k_2$ is above of $k_1$ : $k_1$ is free to hash to any slot and $k_2$ should hash to the slot above $k_1$. So the probability for this case is $1/m$.

   (b) $k_2$ is below $k_1$: $k_1$ is free to hash to any slot. Now for $k_2$ to hash to the next slot, it can either hash to the same slot occupied by $k_1$ or it can hash directly to the next slot. So $k_2$ has 2 places to hash to, therefore the probability of this case is $2/m$.

   Now for $k_3$ to require 3 probes for insertion, it should hash to the key above of the two $k_1$ and $k_2$, so the total probability $k_3$ hashing to the above slot is: $(1/m + 2/m) \times 1/m = 3/m^2$.

ii. $P(k_1 \text{ requires exactly 2 probes}) = 0$. Because $k_1$ was inserted first, it must have been inserted into its initial probe slot. Therefore, searching for $k_1$ will only require the initial probe.

**Problem 4. Piles** [20 points] (3 parts)

The heaps that we discussed in class are binary trees that are stored in arrays; there are no explicit pointers to children, because the index of children in the array can be computed given the index of the parent.

This problem explores heaps (priority queues) that are represented like search trees, in which each tree node is allocated separately and in which parent nodes have explicit pointers to their children. We will call this data structure a *Pile*. A node in a pile can have zero to two children, and the value stored in the node must be at least as large as the value stored at its children.

**(a)** What is the asymptotic cost of INSERT and EXTRACT-MAX in a pile of height $h$ with $n$ nodes? Explain.

**Solution:** Both costs are $O(h)$. These algorithms don't differ significantly from the corresponding heap operations, except that the height of a pile is not guaranteed to be $O(\lg n)$, because the pile might be an unbalanced tree.

**(b)** An *AVL Pile* is a pile in which every node also stores the height of the subtree rooted at the node, and in which the height of the children of a node can differ by at most one. (The height of a missing child is defined to be $-1$.) What is the maximum height of an AVL pile with $n$ nodes?

**Solution:** The same analysis as for an AVL tree works here, too; the relationship between the nodes doesn't play into the height analysis at all. The height of an AVL pile is $O(\lg n)$.

**(c)** Describe a simple algorithm to insert a value into an AVL Pile while maintaining the AVL property (ensuring that the height difference between siblings is at most one). Argue that your algorithm indeed maintains the AVL property.

**Solution:** The idea is to add the new node into the side that has the smaller height, thus ensuring that the insertion doesn't violate the AVL property. This eliminates the need for rotations, which are difficult to implement correctly in a pile.

Start at the root of the pile. Compare the value being inserted to the value at the root node. Store the larger of the two at the root node, then take the smaller of the two and recursively insert it into the subtree with the smaller height. (If the subtrees have equal height, pick one arbitrarily.) Because the subtree height will increase by at most 1 as a result of the insertion, the AVL invariant is preserved.

Now climb back up from the new leaf to the root, keeping track of the height you climbed up. At each node along the way, if the depth is larger than the height stored at the node then increase the stored height by 1. We cannot increase the heights on the way down because we do not know whether the new leaf will increase the height of a node or not.

**Problem 5. Rolling hashes** [15 points] (3 parts)

Ben Bitdiddle, Louis Reasoner, and Alyssa P. Hacker are trying to solve the problem of searching for a given string of length $m$ in a text of length $n$.

They implement different algorithms that all follow the same general outline, and all make use of a hash function $h(x, p)$ which maps a string $x$ to a number in the range $0 \ldots p - 1$.

The values returned by $h(x, p)$ satisfy uniform hashing.

```
 1  def string_search(text, substring):
 2      m = len(substring)
 3      n = len(text)
 4      if m > n: return None
 5      p = ...   # differs based on implementation
 6      target = h(substring, p)
 7      for start in xrange(0, n-m + 1):
 8          # the +1 includes the endpoint n-m
 9          hvalue = ...    # calculate h(x, p)
10          if hvalue == target:
11              if text[start:start+m] == substring:
12                  return start
13      return None
```

**(a)** Ben Bitdiddle chooses $p$ so that $1/2 \leq \alpha \leq 1$, where $\alpha = n/p$. He implements the hash operation on line 9 using a non-rolling hash:

```
hvalue = h(text[start:start+m], p)
```

In terms of $m$ and $n$ (not $\alpha$), what is the asymptotic expected running time of Ben's algorithm? Explain your answer.

$T(m, n) = \Theta(\underline{\quad m \cdot n \qquad\qquad\qquad})$

*Explain:*

**Solution:** Ben has to slice and hash a substring of length $m$ every time through the loop. Both of these operations take $\Theta(m)$ time, dominating the running time of the loop. The loop runs for $\Theta(n)$ iterations ($n - m + 1$ is still $\Theta(n)$). These multiply to give an overall running time of $\Theta(m \cdot n)$.

**(b)** Alyssa P. Hacker recognizes that this is a good case to use a rolling hash. She defines $h$ so that she can calculate each hash value `h(text[start:start+m], p)` from the previous value `h(text[start-1:start-1+m], p)` using a constant number of arithmetic operations on values no larger than $p$.

Instead of calling the $h$ function on the substring every time like Ben Bitdiddle does, then, on line 9 she simply updates `hvalue` based on its previous value using these arithmetic operations. She chooses $p$ in the same manner as Ben.

In terms of $m$ and $n$ (not $\alpha$), what is the asymptotic expected running time of Alyssa's algorithm? Explain your answer.

$T(m, n) = \Theta($ $n \lg n$ or $n$ $)$

*Explain:*

**Solution:** There are two operations in the inner loop whose runtime we should analyze: calculating the hash, and checking for a substring match. Because of uniform hashing, we know that the substring match is not checked every time, so we will analyze them separately.

After an initialization which takes $\Theta(m)$ time, the hash is performed using a constant number of arithmetic operations on numbers no larger than $p$. As $n$ gets very large, $p$ must as well, making the arithmetic operations take longer. Addition and multiplication by constants both take $\Theta(lgp) = \Theta(lgn)$, so the total cost of the rolling hash over all iterations of the loop is $\Theta(m + n \lg n) = \Theta(n \lg n)$.

It is often acceptable to make the simplifying assumption that arithmetic can be done in constant time. Therefore, you could also say that the rolling hash takes $\Theta(m+n) = \Theta(n)$ time and get full credit.

Each time a matching hash is found, the substrings have to be compared, which takes time proportional to the length of the strings: $\Theta(m)$. However, there will not be a hash collision every time. By uniform hashing we know to expect $\alpha$ hash collisions in total. So the total cost of this step is $\Theta(\alpha \cdot m)$, which is $\Theta(m)$ because $\alpha$ is bounded by a constant.

The total of these two costs is $\Theta(n \lg n + m) = \Theta(n \lg n)$, or $\Theta(n + m) = \Theta(n)$ if you made the simplifying assumption about arithmetic.

**(c)** Louis Reasoner doesn't want to worry about searching for a good prime number, so on line 5 he simply sets `p = 1009` and lets $\alpha$ vary. He implements the rolling hash like Alyssa.

In terms of $m$ and $n$ (not $\alpha$), what is the asymptotic expected running time of Louis's algorithm? Explain your answer.

$$T(m, n) = \Theta(\underline{\ m \cdot n \qquad\qquad\qquad})$$

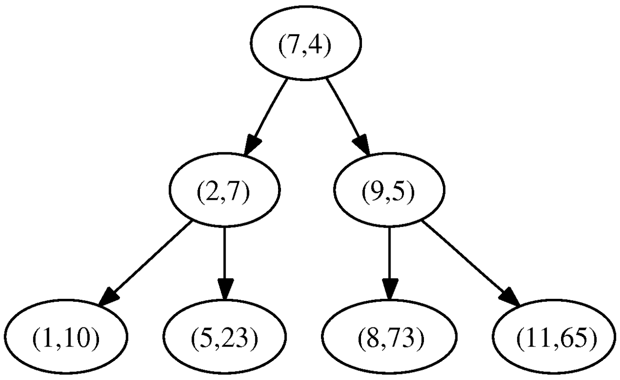
*Explain:*

**Solution:** Analyze this code in the same way as Alyssa's. The crucial difference is that $\alpha$ is no longer bounded by a constant; it grows proportionally to $n$. The term that dominates is $\Theta(n \cdot m)$.

**Problem 6. Treaps** [15 points] (1 parts)

Ben Bitdiddle recently learned about heaps and binary search trees in his algorithms class. He was so excited about getting to know about them, he thought of an interesting way to combine them to create a new hybrid binary tree called a *treap*.

The treap $T$ has a tuple value stored at each node $(x, y)$ where he calls $x$ the $key$, and $y$ the $priority$ of the node. The keys follow the BST property (maintain the BST invariant) while the priorities maintain the min-heap property. An example treap is shown below:



Describe an efficient algorithm for INSERT$((x, y), T)$ that takes a new node with key value $x$ and priority $y$ and inserts it into the treap $T$. Analyze the running time of your algorithm on a treap with $n$ nodes, and height $h$.

**Solution:** Insert the new node according to the standard BST INSERT procedure. This preserves the BST property, but may result in a violation of the heap property. Now, we need to fix the heap property by moving the new node up in the treap until it reaches the correct place. Do this by repeatedly rotating on the node's parent, moving the node up one level while preserving the BST invariant.

The BST insertion requires $O(h)$ time. Each rotation takes $O(1)$, and $O(h)$ rotations are required, so the total insertion time is $O(h)$.

**Problem 7. Amortized Successor** [20 points] (3 parts)

The *successor* of a node $x$ in a binary search tree $T$ is the node in $T$ with the smallest value that is larger than the value of $x$. We assume in this question that all the values in the tree are distinct.

Consider the following code for determining the successor of some node $x$. A node is an object with a value and pointers to its parent and left and right children.

```
# gets successor of x in tree rooted at root
def get_successor(x, root):
    if x == None:
        return None
    elif x.right != None:
        return get_minimum(x.right)
    else:
        while x != root and x.parent.right == x:
            x = x.parent
        return x.parent

# gets minimum node in tree rooted at x
def get_minimum(x):
    if x == None:
        return None
    elif x.left == None:
        return x
    else:
        return get_minimum(x.left)
```

**(a)** Show that the worst-case running time of `get_successor` is $O(\lg n)$ on a balanced BST and $O(n)$ on a general BST.

**Solution:** There are two worst cases where `get_successor` has to traverse the entire height of the tree. In the first case, `get_successor` is called on the root of the tree and its successor is at the bottom of the tree. Here, the call to `get_minimum` must visit every node in between, so that the running time of `get_successor` is $O(h)$, where $h$ is the height of the tree. In the other case, `get_successor` is called on a node at the bottom of the tree and its successor is the root. Here, the loop that checks successive parents to determine the first ancestor to the right of the current node must visit every node in between, for a running time of $O(h)$. Since each of the two cases has a running of $O(h)$, the worst-case running time of `get_successor` is $O(h)$. For a balanced BST, $h = O(\lg n)$, and for a general BST, $h = O(n)$.

The following procedure returns a list of all the values in a binary search tree by finding the minimum and then locating the successor of each node in turn.

```
# returns ordered list of values in tree rooted at root
def spell_out(root):
    node_list = []
    current_node = get_minimum(root)
    while current_node != root.parent:
        node_list.append(current_node.value)
        current_node = get_successor(current_node, root)
    return node_list
```

**(b)** Since `get_successor` is called once for each node in the tree, an upper bound for the worst-case running time of `spell_out` is $O(n \lg n)$ on a balanced BST and $O(n^2)$ on a general BST. This bound, however, is not asymptotically tight.

Using amortized analysis, show that `spell_out` runs in worst-case $O(n)$ time on *any* BST.

*Hint:* How many times is each edge encountered over the course of `spell_out`?

**Solution:** Each edge is encountered exactly twice over the course of `spell_out`, once while descending into a subtree and once while ascending from a subtree. An edge is not passed during more than one descent because once we have visited all nodes in a subtree, subsequent calls to `get_successor` do not check a current node's left child. An edge is also not passed during more than one ascent because each ascent requires a preceding descent.

Each edge traversal has an $O(1)$ cost because the operations within each recursive call to `get_minimum` and each iteration of the `while` loop in `get_successor` has a constant cost. All remaining operations have an $O(1)$ cost and are performed once for each node. Therefore the total cost of `spell_out` consists of local node operations for $n$ nodes (for a total of $O(n)$) and accumulated edge-dependent operations on $n-1$ edges (for a total of $O(n)$). Altogether, the entire procedure runs in $O(n)$ time.

**(c)** Give an asymptotically tight bound on the amortized cost of `get_successor` over the course of one call to `spell_out`.

**Solution:** Over the course of one call to `spell_out`, `get_successor` is called $n$ times, once for each node. The total cost of these `get_successor` calls is the cost of `spell_out` minus the cost of the initial call to `get_minimum`. We can ignore this initial cost since we are computing an upper bound on the total cost of the $n$ `get_successor` calls. The amortized cost of a single call is then $T(n)/n$, there $T(n) = O(n)$ is the overall cost of the $n$ `get_successor` calls. This is $O(1)$.