Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 120 minutes to earn 120 points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz booklet contains 12 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz at the end of the exam period.
- This quiz is closed book. You may use **one** $8\frac{1}{2}'' \times 11''$ or A4 crib sheet (both sides). No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Parts	Points	Grade	Grader
1	4	10		
2	4	16		
3	4	24		
4	3	20		
5	3	15		
6	1	15		
7	3	20		
Total		120		

Name: _____

Friday Recitation:	Rishabh	Rob	Chieu	Jason	Matthew
	10 AM	**12 PM**	**1 PM**	**2 PM**	**3 PM**

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:

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

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

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

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

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

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

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

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:

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

(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:

(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:

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.

(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.

(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.

(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?

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.

(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?

(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.

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 α), what is the asymptotic expected running time of Ben's algorithm? Explain your answer.

$T(m, n) = \Theta($_____$)$

Explain:

(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 α), what is the asymptotic expected running time of Alyssa's algorithm? Explain your answer.

$T(m, n) = \Theta($_____$)$

Explain:

(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 α vary. He implements the rolling hash like Alyssa.

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

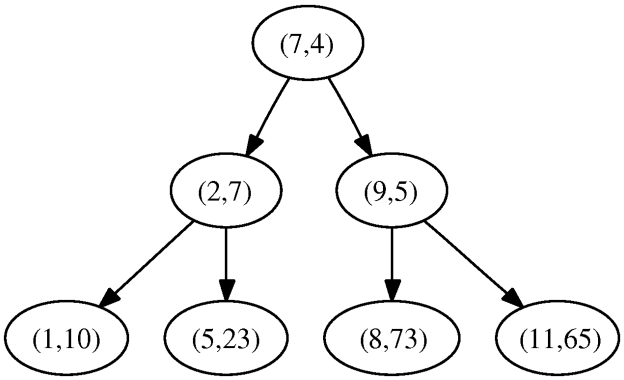
$T(m, n) = \Theta($_____$)$

Explain:

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.

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.

```
1  # gets successor of x in tree rooted at root
2  def get_successor(x, root):
3      if x == None:
4          return None
5      elif x.right != None:
6          return get_minimum(x.right)
7      else:
8          while x != root and x.parent.right == x:
9              x = x.parent
10         return x.parent
11
12 # gets minimum node in tree rooted at x
13 def get_minimum(x):
14     if x == None:
15         return None
16     elif x.left == None:
17         return x
18     else:
19         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.

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.

```
1 # returns ordered list of values in tree rooted at root
2 def spell_out(root):
3     node_list = []
4     current_node = get_minimum(root)
5     while current_node != root.parent:
6         node_list.append(current_node.value)
7         current_node = get_successor(current_node, root)
8     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`?

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

SCRATCH PAPER

SCRATCH PAPER