CSE 373 18su: Midterm

Name:

UW ID and UW email address:

Instructions

- **Do not start the exam until told to do so**.
- You have 60 minutes to complete the exam.
- This exam is closed book and closed notes.
- You may not use a graphing calculator, cell phone, or any other electronic devices except a simple 4-function or scientific calculator, although you should not need one.
- If you need extra space, use the back of the page.
- If you have a question, raise your hand to ask the course staff for clarification.
- **Unless otherwise stated runtime analysis is assumed to be worst-case.**

Question	Max points	Earned
Question 1	15	
Question 2	16	
Question 3	10	
Question 4	15	
Question 5	15	
Question 6	15	
Question 7	14	
Total	**100**	

1. True or False (15 points)

(a) An in-order traversal of a BST always will give the elements in sorted order.
T

(b) The runtime of inserting an element into a BST is $\Theta(\log n)$.
F

(c) $f(n) = 4 \cdot n^2 + 20000 \cdot n + 3$ dominates $g(n) = 20 \cdot n^2 - 20 \cdot n + 20$.
T

(d) $f(n) = 8 \cdot n \cdot \log n + 10^8$ dominates $g(n) = 10^{20} \cdot n + 1$.
T

(e) $\log(n2^n) \in \Theta(n \cdot \log n)$
F

(f) Running Floyd's build-heap and building a heap through repeated insertion (in the same order as the array) will always produce different heaps.
F

(g) AVL insert is faster (asymptotically) than linked list insert in all cases (best, worst, and average).
F

(h) A sorted array (min to max) is also a valid min-heap.
T

(i) A complete binary tree is always balanced.
T

(j) A balanced binary tree is always complete.
F

2. Recurrences (16 points)

Solve these recurrences (give the simplest Big-Θ bound). If the Master Theorem is applicable, state which case you used (write the inequality). If you use unrolling or the tree method, show your work.

(a)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7 \cdot T(n/3) + n^2 & \text{otherwise} \end{cases}$$

$$\log_b a < c$$
$$\Theta(n^2)$$

(b)

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot T(n-1) + 2^n & \text{otherwise} \end{cases}$$

$$\left(\sum_{i=0}^{n-1} 2^i \cdot 2^{n-i}\right) + 2^n = \left(\sum_{i=0}^{n-1} 2^n\right) + 2^n = n2^n + 2^n \in \Theta(n2^n)$$

(c)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 \cdot T(n/3) + n & \text{otherwise} \end{cases}$$

$$\log_b a = c$$
$$\Theta(n \log n)$$

(d)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 16 \cdot T(n/4) + n & \text{otherwise} \end{cases}$$

$$\log_b a > c$$
$$\Theta(n^2)$$

3. Big-O Notation (10 points)

Let $f(n) = 10^6 \cdot n^{1.5} + 10^{-6} \cdot n^2$. Show that $f(n) = O(n^2)$ by finding a constant c and an integer n_0 and applying definition of Big-O.

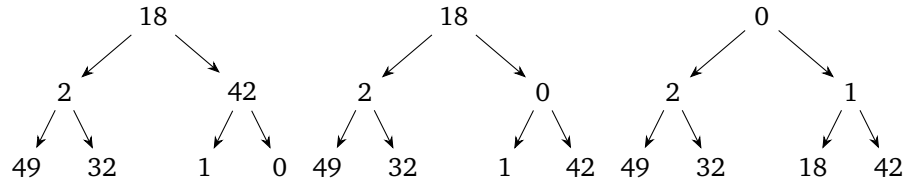
$$f(n) = 10^6 \cdot n^{1.5} + 10^{-6} \cdot n^2 \overbrace{\leq 10^6 \cdot n^2 + 10^{-6} n^2 = (10^6 + 10^{-6}) n^2}^{\forall n \geq 1}$$

So, if $c = 10^6 + 10^{-6}$ and $n_0 = 2$, then $f(n) \leq cn^2 \ \forall n \geq n_0$. Thus $f(n) \in O(n^2)$.

4. Heaps (15 points)

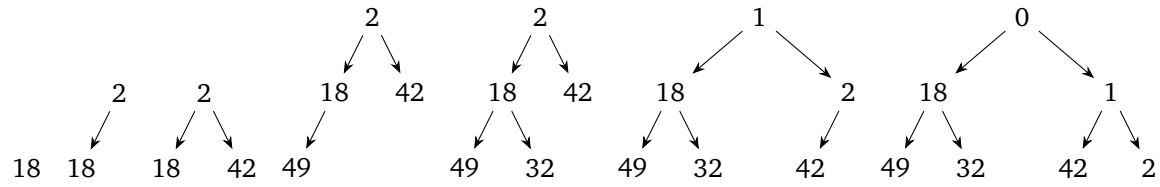
Consider the following sequence of numbers: $18, 2, 42, 49, 32, 1, 0$. This question asks you to build a binary **min**-heap with these numbers in two ways.

(a) Use Floyd's build-heap to build the heap. Draw the heap before and after each percolation (if two consecutive steps result in an identical heap, you don't need to re-draw the heap). Finally, write out the final **array representation** of the heap.



0	2	1	49	32	18	42

(b) Build the heap using repeated insertions (in the order given: $18, 2, 42, 49, 32, 1, 0$) - draw the heap after each insertion. At the end, draw the **array representation** of the final heap.



0	18	1	49	32	42	2

(c) Is one of these techniques (for a general array) asymptotically faster than the other? What are the big-O worst case runtimes for Floyd's build-heap and repeated insertion? Explain why there is or is not a difference in runtime between these two approaches (a formal proof of runtime is not necessary here, just a qualitative explanation of the similarities and/or differences between the algorithms).

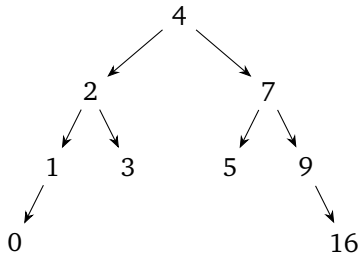
Yes. Floyd's build heap is faster at $O(n)$ that repeated insertions at $O(n \log n)$. This is because Floyd's build heap does the shortest percolations from the largest layers (down - a distance of 0, from the bottom layer), while repeated insertions does the longest percolations (up - a max distance of $\log n$) from the largest layer.

5. AVL Trees (15 points)

Execute the following operations on an AVL tree:

$\mathsf{Insert}(2), \mathsf{Insert}(1), \mathsf{Insert}(4), \mathsf{Insert}(5), \mathsf{Insert}(9), \mathsf{Insert}(3), \mathsf{Insert}(6), \mathsf{Insert}(7), \mathsf{Insert}(16), \mathsf{Insert}(0), \mathsf{remove}(6).$

You are only required to show the final tree, although drawing intermediate trees may result in partial credit. If you draw intermediate trees, please circle your final tree for ANY credit. Assume remove always draws replacement nodes from the right subtree.



6. Hashing (15 points)

(a) Let the capacity of the hash table be 10. Insert elements

$$12, 11, 17, 22, 14, 25, 15$$

to a hash table

(i) that uses quadratic probing, and the hash function is the most significant digit (the first digit in the number: $h(42) = 4$.

(ii) that uses double hashing, and the first hash function is the most significant digit and the second hash function is the least significant digit (the last digit: $g(42) = 2$). Use the second hash function as-is, *do **not*** modify it to make it relatively prime to the table size.

You do not need to consider array re-sizing. Write down the total number of collisions and the hash table after all insertions in both cases. Assume array indices start at 0.

Quadratic Probing:

14	12	11	22		17	25	15		

Number of Collisions: 13

Double Hashing:

	12	11		22	14	15	25	17	

Number of Collisions: 6

(b) Make a quantitative argument for why double hashing leads to fewer collisions than quadratic probing.

For an array of capacity T, double hashing has $T(T-1) \in O(T^2)$ distinct probing sequences (one for each combination of $h(x)$ and $g(x)$, while quadratic probing only has T distinct probing sequences (one for each $h(x)$).

7. Algorithm Design (14 points)

The **Set** ADT defines an order-less collection of unique objects (no object exists more than once in the same set, i.e. a set cannot contain two 3's). The set ADT support the following methods

- **add(item)** - Adds an item to the set. If the item is already contained in the set, the set remains unchanged.
- **remove(item)** - Removes an item from the set. If the item is not contained in the set, the set remains unchanged.
- **contains(item)** - Returns *true* if item is contained within the set, *false* otherwise.

(a) In your project, you implemented the Dictionary ADT. A dictionary can be used to implement a Set. Describe how you could use a Dictionary to implement Set. You can instantiate Dictionaries, and use any of the methods of the ADT, but you cannot rely on its implementation details. List any fields that your Set implementation has and their initial values, and write psuedocode for the **add**, **remove**, and **contains** methods.

Algorithm 1 Dictionary Set

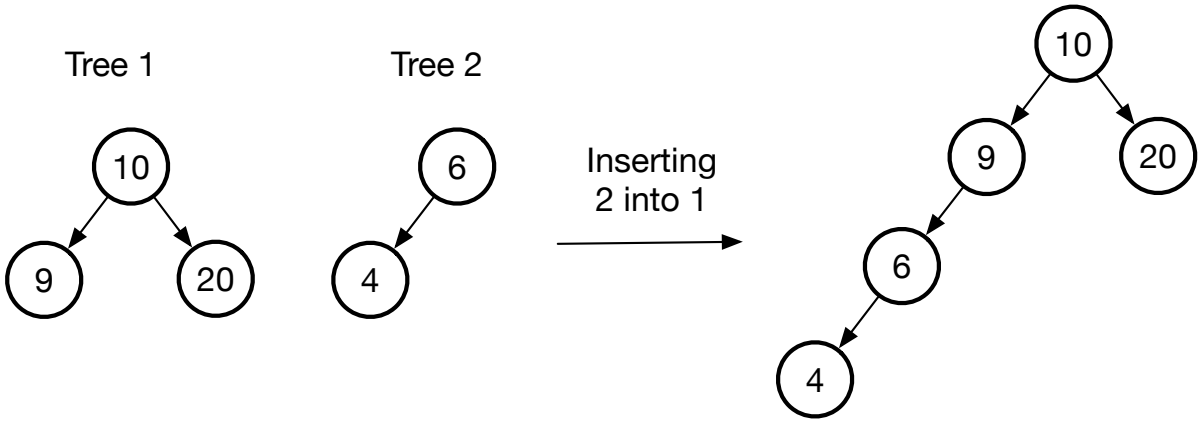
```
dict is a dictionary with boolean values
function add(item)
    dict.put(item, true)
function remove(item)
    dict.put(item, false)
function contains(item)
    return dict.getOrDefault(item, false)
```

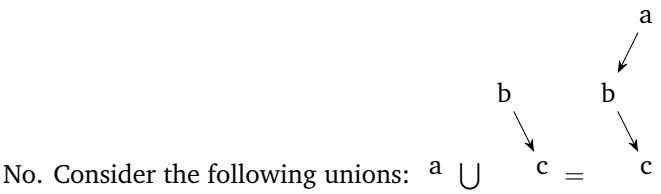
For the remainder of this question, you can assume that the underlying implementation of the Set is a BST. We call this type of set a TreeSet. The following questions are about two methods that create Sets out of other sets: **union** and **intersect**:

- **union(setA, setB)** - Returns a set containing all items that are in either or both of setA and setB.
- **intersect(setA, setB)** - Returns a set containing only items that are contained in both setA and setB.

Some of these questions will ask you to write psuedo-code. You may assume that you have access to, and can modify the underlying data structure. You can also use any of the standard tree methods we have discussed in class or in homework (add, remove, find, contains, isEmpty, balance, findMin, findMax, in-order traversal, pre-order traversal, post-order traversal), and can assume that these methods give their results in a format most convenient for you (i.e. get can return either the value of the node or the node object itself, traversals can return a list or an iterator to nodes or values, etc.)

(b) You want to implement Union for tree set, but think that adding each element one-by-one into the output set sounds too slow at $O(n \log n)$ so you decide to instead find where the root of the second tree would be added into the first, and append the entire second tree at that location (add the second tree's root into the first with all of its children in-tow - see diagram below), giving you $O(\log n)$ for union (in place)! Does this strategy work? If so, make an argument why, if not, provide a counterexample and explain why it is a counterexample.



No. Consider the following unions: 

The resulting structure is not a valid BST.

(c) Now you want to implement **intersect(setA, setB)**. Suppose you are using an AVL tree to represent your Sets. Devise an $O(n \log n)$ algorithm to compute the intersection of two sets. Write psuedo-code for **intersect(setA, setB)**, and analyze the psuedo-code to argue that your algorithm is $O(n \log n)$. The return value of your function should be a new Set.

Algorithm 2 Intersect

```
function intersect(setA, setB)
    result = empty tree set
    for all x in setA.inOrderTraversal do
        if setB.find(x) then
            result.add(x)
    return result
```

The for loop runs once for each element of $setA$, so at most n times. Since $setB$ is implemented as an AVLtree, $find$ and add are both $O(\log n$ operations, giving an overall runtime of $O(n \log n)$.

(d) Now suppose you just want to print out the intersection of two sets rather than construct a new Set data structure. Devise an $O(n)$ algorithm to do this. Write the psuedo-code for **printIntersection(setA, setB)**, and analyze you psuedocode to show that it is $O(n)$. The return value of your function should be void.

Algorithm 3 Print Intersectiont

```
function printIntersection(setA, setB)
    A = inorder traversal of setA (as a queue)
    B = inorder traversal of setA (as a queue)
    while A not empty and B not empty do
        if A.peek < B.peek then
            A.dequeue
        else if B.peek < A.peek then
            B.dequeue
        else
            print(A.dequeue)
            B.dequeue
```

Each traversal is $O(n)$, and we iterate through each traversal at most once, so the overall runtime is $O(n)$.

This page is intentionally left empty

Feel free to use this page for scratch paper.