Quiz 1 Solutions

Problem 1. Asymptotic workout [15 points]

For each function $f(n)$ along the left side of the table, and for each function $g(n)$ across the top, write O, Ω, or Θ in the appropriate space, depending on whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. If more than one such relation holds between $f(n)$ and $g(n)$, write only the strongest one. The first row is a demo solution for $f(n) = n^2$.

		$g(n)$		
		n	$n \lg n$	n^2
$f(n)$	n^2	Ω	Ω	Θ
	$n^{1.5}$	Ω	Ω	O
	$\sqrt{2^n}$	Ω	Ω	Ω
	$n\sqrt{\lg n}$	Ω	O	O
	$n \log_{30} n$	Ω	Θ	O
	n^3	Ω	Ω	Ω

Problem 2. Table of Speed [20 points]

For each of the representations of a set of elements along the left side of the table, write down the asymptotic running time for each of the operations along the top. For hashing, give the expected running time assuming simple uniform hashing; for all other data structures, give the worst-case running time. Give tight asymptotic bounds using Θ notation. If we have not discussed how to perform a particular operation on a particular structure, answer for the most reasonable implementation you can imagine.

		Operation			
		Insert	Extract-min	Contains	Minimum
Data Structure	Unsorted linked list	$\Theta(1)$ Insert at beginning	$\Theta(n)$ Check every element	$\Theta(n)$ Check every element	$\Theta(n)$ Check every element
	Sorted linked list	$\Theta(n)$ Walk down list	$\Theta(1)$ Remove first element	$\Theta(n)$ Walk down list	$\Theta(1)$ Return first element
	Min heap	$\Theta(\log n)$ Seen in class	$\Theta(\log n)$ Seen in class	$\Theta(n)$ Check every element	$\Theta(1)$ Return first element
	Max heap	$\Theta(\log n)$ Seen in class	$\Theta(n)$ Check every element	$\Theta(n)$ Check every element	$\Theta(n)$ Check every element
	Hashsing with chainig and $\alpha = 1$	$\Theta(1)$ Seen in class	$\Theta(n)$ Check every element	$\Theta(1)$ Seen in class	$\Theta(n)$ Check every element

Definitions of operations:

- Insert(S, x): add element x to the set S.
- Extract-min(S): remove the minimum element from the set S and return it.
- Contains(S, x): return whether element x is in the set S.
- Minimum(S): return the minimum element in set S (without extraction).

Problem 3. Indie heap operations [15 points] (2 parts)

(a) [10 points] Suppose you have a max-heap stored in an array $A[1..n]$, where $A[1]$ stores the maximum element. Give pseudocode for an efficient algorithm to implement the operation find-second-maximum(A), which finds the next-to-largest key stored in the max-heap A. Your algorithm should not modify the heap.

Solution:

```
return max(A[2], A[min(3,n)])
```

(b) [5 points] Suppose you have an array $A[1..n]$ of n elements in arbitrary order. Does the following alternate implementation of build-max-heap work? In other words, does it correctly build a max heap from the given elements $A[1..n]$? Why or why not?

build-max-heap(A):
    for i from 1 to $n/2$:
        max-heapify(A, i)

This algorithm calls heapify starting at the root and working its way down the tree, instead of the other way around.

Solution: No. Starting with $[1, 2, 3, 4]$ we would get $[2, 4, 3, 1]$, which is not a max-heap.

Problem 4. Madly merging many menus [30 points] (2 parts)

Professor Sortun uses the following algorithm for merging k sorted lists, each having n/k elements. She takes the first list and merges it with the second list using a linear-time algorithm for merging two sorted lists, such as the merging algorithm used in merge sort. Then, she merges the resulting list of $2n/k$ elements with the third list, merges the list of $3n/k$ elements that results with the fourth list, and so forth, until she ends up with a single sorted list of all n elements.

(a) [10 points] Analyze the worst-case running time of the professor's algorithm in terms of n and k.

Solution: The i^{th} merge has one list of size $\frac{i*n}{k}$, so it takes $\Theta(\frac{i*n}{k})$ time. Adding up each merge, we get:

$$T(n) = \sum_{i=1}^{k} \frac{i * n}{k}$$
$$T(n) = \Theta(kn)$$

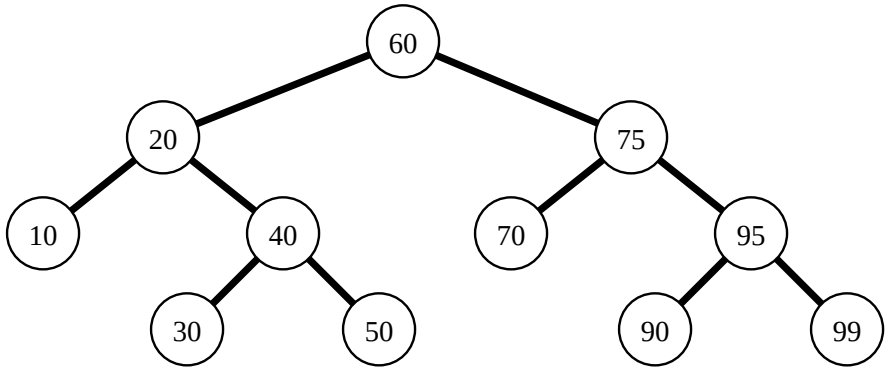
(b) [20 points] Briefly describe an algorithm for merging k sorted lists, each of length n/k, whose worst-case running time is $O(n \lg k)$. Briefly justify the running time of your algorithm. (If you cannot achieve $O(n \lg k)$, do the best you can for partial credit.)

Solution: Put the minimum element of each list into a min-heap. Then, repeatedly extract the minimum element from the heap, and replacing it by inserting the next element from the same list.

The heap will never be bigger than k elements, so each operation (either extract-min or insert) takes $O(\lg k)$ time. There are $O(n)$ operations (one insert and one extract-min for each element), so the running time is $O(n \lg k)$.

Problem 5. Being Adel'son-Vel'skiĭ & Landis [10 points]

Suppose that we insert 42 into the AVL tree below using the AVL insertion algorithm. On the next page, show the resulting tree after rebalancing. We also recommend showing intermediate steps for partial credit.

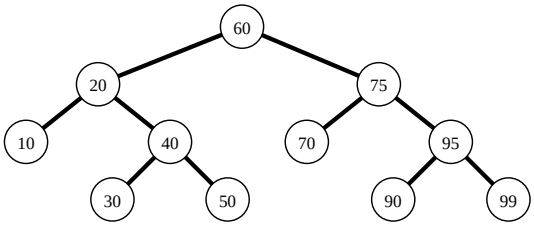


For reference, we include the Python code for AVL insertion, written in terms of rotations:

```
def insert(self, t):
    node = bst.BST.insert(self, t)
    while node is not None:
        if height(node.left) >= 2 + height(node.right):
            if height(node.left.left) >= height(node.left.right):
                self.right_rotate(node)
            else:
                self.left_rotate(node.left)
                self.right_rotate(node)
        elif height(node.right) >= 2 + height(node.left):
            if height(node.right.right) >= height(node.right.left):
                self.left_rotate(node)
            else:
                self.right_rotate(node.right)
                self.left_rotate(node)
        node = node.parent
```

Solution: First, 42 gets inserted, hanging as a left child of 50. Then, 20 is unbalanced, so it is rotated left. 40 takes the place of 20, 20 becomes the left child of 40, and, 30 becomes the right child of 20.

Note: the actual solution requires a picture.



Initial AVL tree

Problem 6. Honey, I shrunk the heap [15 points]

Suppose we have a heap containing $n = 2^k$ elements in an array of size n, and suppose that we repeatedly extract the minimum element, n times, never performing insertions. To make the heap space efficient, we move the heap over to an array of size 2^j whenever an extraction decreases the number of elements to 2^j for any integer j. Suppose that the cost of each such move is $\Theta(2^j)$. What is the total movement cost caused by n extract-mins starting from the heap of n elements? (Ignore the $\Theta(n \lg n)$ cost from the heapify operations themselves.)

Solution: The first move occurs when the number of elements is $2^{(k-1)}$, and the i^{th} move occurs when the number of elements is $2^{(k-i)}$. Total number of movements is k, and total cost is $\sum_{j=k-1}^{0} 2^j = 2^k - 1 = \Theta(2^k) = \Theta(n)$

Problem 7. Trash hees [15 points] (2 parts)

Suppose we store n elements in an m-slot hash table using chaining, but we store each chain (set of elements hashing to the same slot) using an AVL tree instead of a linked list. Also suppose that $m = n$, so the load factor $\alpha = n/m = 1$.

(a) [5 points] What is the expected running time of insert, delete, and search in this hash table? Why? Assume simple uniform hashing.

Solution: Assume uniform hashing and use linked list, all the operations take $O(1+\alpha)$ time, $O(1)$ for applying hash function and access the slot, and $O(\alpha)$ is for inserting, deleting, and searching in this list. Using AVL tree, we can reduce the term $O(\alpha)$ to $O(\lg \alpha)$. In this case,we have load factor α as 1, so all the operations cost $O(1+\lg 1) = O(1)$ time.

(b) [10 points] What is the worst-case running time of insert, delete, and search in this hash table? Why? (Do not assume simple uniform hashing.)

Solution: In the worst case, every element ends up in the same slot. Replacing $O(\alpha)$ in part (a) by $O(n)$, all operations cost $O(1 + \lg n)$.

SCRATCH PAPER

SCRATCH PAPER