Question 1. [5 MARKS]

Part (a) [1 MARK]

Alice claims the array below holds a heap.

55	22	4	5	20	3	2	1		

What type of heap is this? (Write an "×" in the box next to the correct answer.)

☐ the array is a MIN heap

☒ the array is a MAX heap

☐ the array is NOT a valid heap

Part (b) [2 MARKS]

If you answered in Part (a) that the array was NOT a valid heap, explain your reasoning here.
If you decided that the array was a valid heap, fill in values in the array below to show the resulting heap after inserting the value 25.

55	25	4	22	20	3	2	1	5	

Part (c) [2 MARKS]

Bob claims that he can super-efficiently implement a max-priority queue using an **unsorted doubly-linked list**, together with an extra variable ***max*** that stores a pointer to the maximum element in the linked list. According to Bob, the worst-case running times of the operations would be as follows:

- INSERT: just insert at the head of the linked list in $\mathcal{O}(1)$ time.
- EXTRACTMAX: just delete the node pointed to by *max* in $\mathcal{O}(1)$ time (because the list is doubly-linked).
- MAX: just return the priority of the node pointed to by *max* in $\mathcal{O}(1)$ time.

Point out Bob's mistake. Be concise.

SAMPLE SOLUTION: Updating *max* after performing EXTRACTMAX would take $\Omega(n)$ time in the worst case.

Question 2. [6 MARKS]

Part (a) [2 MARKS]

The load factor α of an open addressing hash table must satisfy $\alpha \leqslant 1$. Explain why in one short sentence.

SAMPLE SOLUTION: Each bucket holds at most 1 element.

Part (b) [3 MARKS]

Consider a hash table with $m = 10$ slots using the hash function $h(k) = k \bmod m$ with open addressing (storing every item directly into the table, **without** linked lists). The collision strategy is non-linear probing using the probe sequence $h(k,i) = (k + i^2) \bmod m$ (for $i = 0, 1, \dots, m-1$).

Starting from an empty table, insert the following keys in order: 52, 1005, 96, 92, 2. Write the result directly in the table provided below.

0	
1	2
2	52
3	92
4	
5	1005
6	96
7	
8	
9	

Part (c) [1 MARK]

Give one reason why 10 is a poor choice of m for the hash table in Part (b).

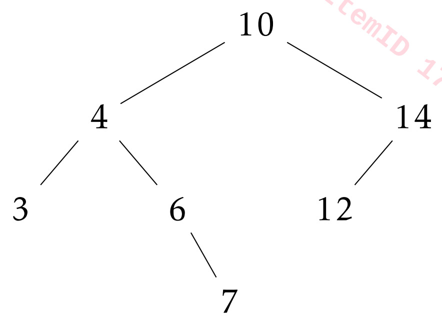
SAMPLE SOLUTION: Any of the following, **or other similar answer**:

- 10 is not prime.
- Using k mod 10 only looks at the units digit of key k, not all the digits.
- In a real hashing application, 10 is a ridiculously small number of keys since it would mean that the maximum number of records that the table could hold would be 10.

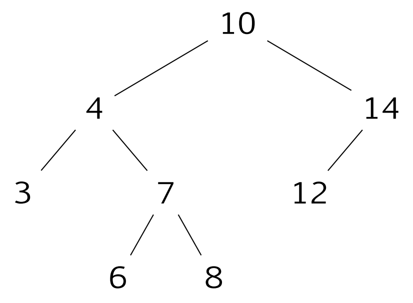
Question 3. [4 MARKS]

Part (a) [2 MARKS]

Insert a node with key 8 into the AVL tree pictured below. Show the resulting tree by either drawing directly on the original tree or by drawing a new tree in the space on the right.



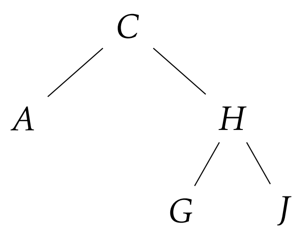
SAMPLE SOLUTION:



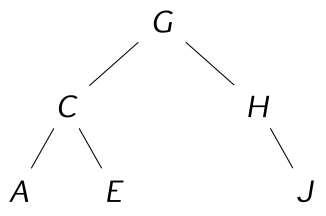
CONT'D...

Part (b) [2 marks]

Insert a node with key E into the AVL tree pictured below. Show the resulting tree by either drawing directly on the original tree or by drawing a new tree in the extra space.



Sample Solution:



Question 4. [4 marks]

Part (a) [2 marks]

Randomized quicksort compares individual pairs of elements but it does not necessarily compare every element to every other element. When the input is the array $[1,5,3,2]$, what is the probability that randomized quicksort compares 1 and 5 directly to each other? Explain your reasoning.

Sample Solution: The probability is .5. If the first pivot is 1 or 5, then 1 and 5 will be compared. If the first pivot is 3 or 2, then 1 and 5 will end up in different sub-arrays and will never be compared to each other.

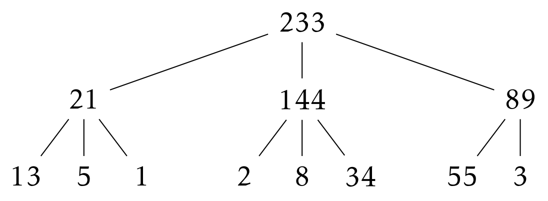
Part (b) [2 marks]

When the input is the array $[1,5,3,2]$, what is the probability that randomized quicksort compares 1 and 2 directly to each other? Explain your reasoning.

Sample Solution: The probability is 1. No matter how the pivots are chosen the 1 must be compared to the 2 because they can never be placed on different sides of the partition (there is no value that falls in between 1 and 2).

Question 5. [5 marks]

In a *ternary* max-heap, every node has exactly three children (except for the leaves and, possibly, one node with fewer than three children). The values stored in each node are ordered according to the same principle as for binary heaps: the value at each node is greater than or equal to the values in the node's children. For example, the following depicts a ternary max-heap.



Just like for binary heaps, we would like to store the heap elements in an array (without creating a linked structure based on Node objects).

Part (a) [2 marks]

Draw the array representation for the ternary max-heap above.

SAMPLE SOLUTION:

233	21	144	89	13	5	1	2	8	34	55	3	

Part (b) [3 MARKS]

Given the index i of an element in an array that represents some ternary max-heap, give an *exact expression* for each of the following:

- The index of the root of the heap: ROOT = 0—or 1
- The index of the left child of i: LEFT(i) = $3i+1$—or $3i-1$
- The index of the middle child of i: MID(i) = $3i+2$—or $3i$
- The index of the right child of i: RIGHT(i) = $3i+3$—or $3i+1$
- The index of the parent of i: PARENT(i) = $\lfloor (i-1)/3 \rfloor$—or $\lfloor (i+1)/3 \rfloor$

Question 6. [14 MARKS]

Given an unsorted list of n items where the value of each item is chosen independently from a universe of m values, you want to find the k largest values. Assume that $m \gg n$ (m is much larger than n) and $n > k$. Also, note that the values are **not** necessarily chosen at random.

Analyze each of the approaches below by stating whether it WILL NOT WORK, WORKS BUT MAY BE IMPRACTICAL or is A FINE IDEA. In all cases, explain your thinking. For approaches that will not work, explain why not. For all other approaches, provide the worst-case time complexity and show your work.

Part (a) [2 MARKS]

Insert all n items into an AVL tree using their values as the keys. Starting at the right-most element of the tree, return k of the elements.

[✓ WILL NOT WORK] [✓ WORKS BUT MAY BE IMPRACTICAL] [✓ A FINE IDEA]

Explanation/Worst-case time complexity:

SAMPLE SOLUTION: Complexity is $\mathcal{O}(n \log n)$ to create the tree and then $\mathcal{O}(k)$ to return the k items. Total is $\mathcal{O}(k + n \log n)$. It was also acceptable to say impossible if duplicate values are allowed (not without modifying the basic AVL tree).

Part (b) [2 MARKS]

Create an array of size m where there is a position for each possible value in the universe. Insert all n items into this data structure. Starting with the highest m value and examining each list in decreasing order of m, return items until you have the k items you need.

[✓ WILL NOT WORK] [✓ WORKS BUT MAY BE IMPRACTICAL] ☐ A FINE IDEA

Explanation/Worst-case time complexity:

SAMPLE SOLUTION: One might say it is impossible for space reasons. They might also say impractical for space reasons. Complexity would be $\mathcal{O}(m)$ to create the array of null pointers and then $\mathcal{O}(n)$ to insert and $\mathcal{O}(k)$ to return items. Total complexity is $\mathcal{O}(m + n + k)$.

Part (c) [2 marks]

Create an array A of size k. It will represent the k largest values we have seen at each point in time. $A[0]$ will be the largest value. Examine each of the n items in turn comparing it to the elements in A. If it is larger than any one of the elements then insert it into the array and shift the others down dropping the element that was previously at $A[k-1]$.

☐ Will Not Work [✓ Works But May Be Impractical] [✓ A Fine Idea]

Explanation/Worst-case time complexity:

Sample Solution: Practicality depends on the size of k. If k is very small this is fine. Complexity is $\mathcal{O}(kn)$.

Part (d) [2 marks]

Insert all the items into a max-heap. Call ExtractMax k times on the heap.

☐ Will Not Work ☐ Works But May Be Impractical [✓ A Fine Idea]

Explanation/Worst-case time complexity:

Sample Solution: Complexity is $\mathcal{O}(n)$ to create the max-heap and then $\mathcal{O}(k \log n)$ to pop the k items. Total is $\mathcal{O}(n + k \log n)$.

Part (e) [2 marks]

Insert all the items into a min-heap. Return the **last** k items from the heap.

[✓ Will Not Work] ☐ Works But May Be Impractical ☐ A Fine Idea

Explanation/Worst-case time complexity:

Sample Solution: The last k items in a min-heap are not necessarily the largest k items, for example, $[1, 2, 5, 3, 4]$ with $k = 2$.

Part (f) [2 marks]

Sort the items using quick-sort or merge-sort and then return the first k values.

☐ Will Not Work ☐ Works But May Be Impractical [✓ A Fine Idea]

Explanation/Worst-case time complexity:

Sample Solution: Sort is $\mathcal{O}(n \log n)$. Return is $\mathcal{O}(k)$. Total is $\mathcal{O}(n \log n + k)$.

Part (g) [2 marks]

Create a hash-table of size p where p is as large as you can manage given space constraints. Hash all n items to the table. Then remove the largest k items from the table and return them.

[✓ Will Not Work] ☐ Works But May Be Impractical ☐ A Fine Idea

Explanation/Worst-case time complexity:

Sample Solution: You can't get the largest k items out of the table without knowing the keys.

Question 7. [10 marks]

In this question, you will augment AVL trees to implement the following new operation:

- NumGreater(k): return the number of elements with a key strictly greater than k.

Part (a) [2 marks]

What additional information will you store at each node of the AVL tree? Be specific.

Sample Solution: *x.size*: The size of the subtree rooted at the node.

Part (b) [3 marks]

Explain how to maintain your new information during Insert operations, without affecting the running time by more than a constant factor.

SAMPLE SOLUTION: The insertion can only change the *size* attribute of the ancestors of the newly inserted node. So the information can be maintained at the same time as the insertion is carried out (including any necessary rotations), using only a constant amount of additional time on each level. Since in an AVL tree the number of ancestors of a node is in $\mathcal{O}(\log n)$, the maintenance takes $\mathcal{O}(\log n)$ time, without affecting the running time of INSERT.

Part (c) [5 MARKS]

Give a detailed implementation for operation NUMGREATER. Explain what you are doing (use comments or a brief English description of the main idea of your implementation). Then, analyse the worst case complexity of your implementation.

SAMPLE SOLUTION:
Iterative solution: First do a TREESEARCH to determine whether key k exists in the tree. If not, perform a TREEINSERT to insert a node with key k.

```
NUMGREATER($k$):
    $x \leftarrow$ TREESEARCH($k$)    # first locate the node with key $k$
    $r \leftarrow x.right.size$
    while $x \neq root$:
        if $x = x.p.left$:    # $x$ is a left child
            $r \leftarrow r + x.p.right.size + 1$
        $x \leftarrow x.p$
    return $r$
```

Recursive solution (in the style of François′ lecture notes): No need to check whether or not k is already in the tree with this solution.

```
NUMGREATER($k$):
    return TREESUM($root, k$)

TREESUM($root, k$):
    if $root$ = NIL:
        return 0
    else if $k > root.key$:
        # Values in the left subtree can be ignored: they all have key less than $k$.
        return TREESUM($root.right, k$)
    else if $k < root.key$:
        # Values in the right subtree must be counted: they all have key larger than $k$.
        return TREESUM($root.left, k$) + $root.right.size + 1$
    else:
        return $root.right.size$
```

Question 8. [12 MARKS]

Consider the following algorithm that inserts a value x into a sorted array A. Remember that in this course, the word "array" means a C-style array with a *fixed size* (**not** a Python-style list that grows and shrinks automatically). In this context, assume that A is not entirely full when x is inserted: A contains $A.size$ elements ($A[0], A[1], \ldots, A[A.size-1]$) out of $A.length$ possible locations (the rest of the locations $A[A.size], A[A.size+1], \ldots, A[A.length-1]$ store the special value NIL).

For example, starting with $A = [5, 8, 21, 34, \text{NIL}, \text{NIL}]$, $x = 13$ and calling SORTEDINSERT(A, x) changes the contents of A to $[5, 8, 13, 21, 34, \text{NIL}]$.

```
SortedInsert($A, x$):
    # Precondition: $A[0] \leqslant \ldots \leqslant A[A.size-1]$ and $A[A.size] = \cdots = A[A.length-1] =$ NIL.
    # First, store $x$ in the first available location.
    $k \leftarrow A.size$
    $A[k] \leftarrow x$
    # Next, repeatedly swap $x$ with the previous element, as long as they are out of order.
    # Loop Invariant: $x = A[k] < A[k+1] \leqslant \ldots \leqslant A[A.size]$.
    while $k > 0$ and $A[k-1] > A[k]$:
        swap $A[k-1]$ with $A[k]$        # Count only this operation!
        $k \leftarrow k-1$
    $A.size \leftarrow A.size + 1$
```

In this question, you will analyse the time complexity of algorithm SortedInsert by counting only the number of *swaps* executed (ignoring all other operations).

Part (a) [2 marks]

What is the **best** case number of swaps executed by SortedInsert? Justify your answer.

Sample Solution: When $x > A[A.size - 1]$, SortedInsert executes 0 swap, the minimum possible.

Part (b) [2 marks]

What is the **worst** case number of swaps executed by SortedInsert? Justify your answer.

Sample Solution: When $x < A[0]$, SortedInsert executes $A.size$ swaps, the maximum possible.

Part (c) [3 marks]

Define a sample space and a probability distribution that *could* be used to analyse the average case number of swaps executed by SortedInsert.

Sample Solution:

Sample space: $S_n = \Big\{(A,x) : A = [1,3,5,\ldots,2n-1]$ and $x = 0,2,4,\ldots,2n\Big\}$ (one input for each possible insertion position for x in A).

Probability distribution: Uniform: $\Pr[(A,x)] = 1/(n+1)$ for each $(A,x) \in S_n$.

Part (d) [5 marks]

Let $A = [1,3,5,7]$ and choose x as follows: pick two values a, b independently and uniformly at random from the set $\{0,2,4\}$ and let $x = a + b$.

Compute the **average** case number of swaps executed by SortedInsert(A,x), where (A,x) is the random input described above—**do NOT use your probability distribution from Part (c)**; instead, use the probability distribution defined implicitly above. Explain your reasoning, show your work, and simplify your final answer.

Sample Solution: The probability of any one pair $(a,b) \in \{0,2,4\}^2$ is $1/9$, so we get the following probabilities for each possible value of x.

value of x	probability	relevant pairs (a,b)
0	$1/9$	$(0,0)$
2	$2/9$	$(0,2), (2,0)$
4	$3/9$	$(0,4), (2,2), (4,0)$
6	$2/9$	$(2,4), (4,2)$
8	$1/9$	$(4,4)$

Let X represent the number of swaps executed. Then X takes on values between 0 and 4. So the average case complexity is equal to:

$$\begin{aligned}
E[X] &= \sum_{s=0}^{4} s \cdot \Pr[X = s] \\
&= 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] + 2 \cdot \Pr[X = 2] + 3 \cdot \Pr[X = 3] + 4 \cdot \Pr[X = 4] \\
&= 0 \cdot \Pr[x = 8] + 1 \cdot \Pr[x = 6] + 2 \cdot \Pr[x = 4] + 3 \cdot \Pr[x = 2] + 4 \cdot \Pr[x = 0] \\
&= 0 \cdot 1/9 + 1 \cdot 2/9 + 2 \cdot 3/9 + 3 \cdot 2/9 + 4 \cdot 1/9 \\
&= \frac{0 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 1}{9} = \frac{18}{9} = 2
\end{aligned}$$