Name: ______________________ **Entry number:** ______________

COL106 Re-major Exam Duration: 2 hours

Read the following instructions before you begin writing.

1. Keep a pen, your identity card, and optionally a water bottle with you. Keep everything else away from you, at the place specified by the invigilators.
2. Write your entry number and name on every page. (Sheets will be separated prior to grading.)
3. Answer only in the designated space. Think before you use this space. No additional space will be provided for writing answers. Use the last blank sheet for rough work, if needed. Do not separate sheets from one another.
4. No clarifications will be given during the exams. If something is unclear or ambiguous, make reasonable assumptions and state them clearly. The instructors reserve the right to decide whether your assumptions were indeed reasonable.

---

1. The standard `Stack` data structure supports the following operations.

   1. `Stack()`: constructor which creates an empty stack.
   2. `push(item)`: inserts `item` into the stack.
   3. `pop()`: removes the last inserted item from the stack and returns the item; throws an exception if the stack is empty.
   4. `isEmpty()`: returns `True` if the stack is empty, and `False` otherwise.

   The goal of this problem is to implement a variant of the queue data structure which, in addition to the usual operations `enqueue`, `dequeue`, and `isEmpty`, also supports the `findMin` method which returns the minimum element of the queue. More formally, we want to write a class `MinQueue` which has the following methods.

   1. `MinQueue()`: constructor which creates an empty queue.
   2. `enqueue(item)`: inserts `item` into the queue.
   3. `dequeue()`: removes the earliest inserted item from the queue and returns the item; throws an exception if the queue is empty.
   4. `isEmpty()`: returns `True` if the queue is empty, and `False` otherwise.
   5. `getMin()`: returns the minimum element in the queue (without changing the queue; this is not the same as the `extractMin` operation of a priority queue); throws an exception if the queue is empty.

   We would like all of these operations to have amortized $O(1)$ time. We come up with an implementation in two steps.

   (a) (6 points) Let us first design a class `MinStack` which, in addition to `push`, `pop`, and `isEmpty`, also supports the `getMin` method which returns the minimum element of the stack, without modifying the stack, and throws an exception if the stack is empty. Let us assume that we have an implementation of the standard `Stack` class which only supports `Stack()`, `push(item)`, `pop()`, and isEmpty(); all in $O(1)$ time. The idea is that a MinStack object internally has a stack whose alternate elements are the elements of the MinStack object, and each of the remaining elements of the stack is the minimum of all elements below itself. For example, if ms is a MinStack object which is supposed to contain the elements $2, 7, 1, 8, 2$ from bottom to top, then ms.stack contains the elements $2, 2, 7, 2, 1, 1, 8, 1, 2, 1$ from bottom to top. Complete the following python implementation of of the MinStack class.

```
class MinStack:
    def __init__(self):
        self.stack = Stack()
    def getMin(self):    # Must run in O(1) time.
        # Your code starts here.
        rv = self.stack.pop()
        self.stack.push (rv)
        return rv
        # Your code ends here.
    def push(self, item):    # Must run in O(1) time.
        if self.stack.isEmpty():
            self.stack.push(item)
            self.stack.push(item)
        else:
            # Your code starts here.
            prev min = self.getMin()
            Self.stack.push (item)
            Self.stack.push (min(item, prevmin))

            # Your code ends here.
    def pop(self):    # Must run in O(1) time.
        # Your code starts here.
        prevmin = self.stack.pop()
        return self.stack.pop ()
        # Your code ends here.
    def isEmpty(self):
        return self.stack.isEmpty()
```

(b) (7 points) Next, observe that a queue can be implemented using two stacks, say sf (the "front" stack) and sr (the "rear" stack) as follows. The first few (possibly zero) elements in the front of the queue appear from top to bottom in the stack sf, while the remaining elements appear from bottom to top in sr. Thus, the top of sf is the front of the queue, and the top of sr is the rear of the queue. For example, if sf contains $2, 7$, and sr contains $3, 1, 4, 1$ from bottom to top, then these two stacks represent the queue containing the elements $7, 2, 3, 1, 4, 1$ from front to rear. Using this idea, complete the following implementation of the MinQueue class.

```
class MinQueue:
    def __init__(self):
        self.sf = MinStack()
        self.sr = MinStack()
    def enqueue(self,item):
        self.sr.push(item)
    def dequeue(self):
        # Need not run in O(1) time, but
        #  must run in O(1) amortized time.
        # Your code starts here.
        if self.sf.isEmpty():
              while not self.sr.isEmpty():
                    self.sf.push(self.sr.pop())
        # Your code ends here.
        return self.sf.pop()
    def isEmpty(self):
        return self.sf.isEmpty() and self.sr.isEmpty()
    def getMin(self):    # Must run in O(1) time.
        # Your code starts here.
        if self.sf.isEmpty():
              return self.sr.getMin()
        if self.sr.isEmpty():
              return self.sf.getMin()
        return (min(self.sf.getMin(), self.sr.getMin()))
        # Your code ends here.
```

(c) (3 points) Argue that the dequeue operation of MinQueue runs in $O(1)$ amortized time. In other words, consider an algorithm which starts with an empty MinQueue object, and performs n enqueue and n dequeue operations in an arbitrary order. Prove that the total time spent in these operations is $O(n)$.

Name: ______________________ **Entry number**: ______________

Each of the n enqueued items results in
- $O(1)$ time for enqueue
- $O(1)$ time for being transferred from sr to sf
- $O(1)$ time for being popped from sf.

$\therefore$ Overall time is $O(n)$.

2. We are given the post-order traversal of a binary search tree on keys $\{0, 1, 2, \ldots, n-1\}$ in the array `A[0..n-1]`. The goal is to construct the tree.

   (a) (3 points) Fill in the blanks and complete the following pseudo-code of the procedure `construct-tree(i,j)` that constructs a binary search tree whose post-order traversal is `A[i..j]` and returns the root of this tree.

```
procedure construct-tree(i,j):
    if (j < i):
        return an empty tree.
    k <-- A[j]
    Create a new tree node p.
    p.key <-- k
    r <-- i

    while (A[r] ___<___ k):
        r <-- r + 1

    p.left <-- construct-tree(___i___, ___r-1___)

    p.right <-- construct-tree(___r___, ___j-1___)

    return ___p___
```

   (b) (2 points) Let $T(n)$ denote the worst-case running time of the above pseudo-code when `A` is an array of size n. Write down a recurrence relation for $T(n)$, and give a tight bound on $T(n)$ using the $O(\cdot)$ notation.

$T(n) = \max\limits_{r \in \{0 \ldots n-1\}} T(r) + T(n-1-r) + O(n)$

$T(n) = O(n^2)$

Name: ______________________ **Entry number:** ______________

COL106 Re-major Exam Duration: 2 hours

3. Given a list L of n integers and a positive integer k, our goal is to output the k largest integers in L (in an arbitrary order). Assume $k \leq n$.

   (a) (4 points) Design an $O(n \log k)$ time algorithm for doing this. You are advised to write your algorithm in plain English, clearly stating the data structures used. If you use data structures discussed in class, then it is not necessary to provide their implementation details.

```
- Build a min-heap H out of the first
  k elements of L.            — O(k) time
- For each of the remaining elements
  x of L:                     — O(n) iterations
      If x < min element of x :   }
          - Extractmin (H)        } O(log k) time
          - Enqueue x into H      }
- Return the list of all elements
  remaining in H              — O(k) time
```

   (b) (2 points) Give a short proof of correctness for your algorithm. For example, you may state an easily verifiable loop invariant for your algorithm that implies correctness.

Loop invariant: After each iteration of the for loop, H contains the k largest elements out of the elements processed.

Name: ______________________ **Entry number:** ______________

COL106 Re-major Exam Duration: 2 hours

4. Recall that a directed graph has a topological sort if and only if it is acyclic. We discussed in class an algorithm to find a topological sort of a directed acyclic graph. Here is another idea to do the same: while the vertex set of the graph is non-empty, remove from the graph an arbitrary vertex v of in-degree zero (that is, a vertex without incoming edges) and all the outgoing edges of v. Answer the following questions.

   (a) (3 points) Prove that a directed acyclic graph with a non-empty vertex set has a vertex of in-degree zero. (This is necessary to ensure that our algorithm doesn't get stuck because it couldn't find such a vertex.)

   Let $v_0, v_1, \ldots, v_m$ be a longest path in the graph. We claim v_0 has indegree 0.
   If w is an in-neighbor of v_0:
   If $w = v_j$ for some j, $v_0, \ldots, v_j$ is a directed cycle $\rightarrow$ contradiction
   else, $w, v_0, v_1, \ldots v_m$ is a longer path $\longrightarrow$ contradiction

   (b) (2 points) How will you construct a topological sort of the graph from the sequence of vertices removed? Explain briefly why your answer is correct.

   The order in which vertices are removed is a topological sort.
   If $v_0, \ldots, v_{n-1}$ is this order, then $\forall i$, v_i has in-degree 0 after removing $v_0, \ldots v_{i-1}$ $\therefore$ No in-neighbor in $v_{i+1} \ldots v_{n-1}$.

   (c) (6 points) Let n and m denote the number of vertices and edges respectively, in a directed acyclic graph. Assume for simplicity that the vertex set of the graph is the set $\{0, \ldots, n-1\}$ and it is given in the out-adjacency lists format. More concretely, you are given an array L of n items, where the i'th item is a reference to a linked list containing the out-neighbors of vertex i. Turn the above idea into a concrete $O(n+m)$-time algorithm and write it, giving all details of the data structures used. After an initial pre-processing, in every iteration, your algorithm must find a vertex v of in-degree zero in $O(1)$ time, and then process its "removal" in time $O(d_v)$, where d_v is the out-degree of v.

Name: ______________________ **Entry number**: ________________

```
Preprocessing:
① Find the in-degree of every vertex:
    indeg[v] ← 0  ∀ v.
    For every v and every out-nbr u of v:          — dv iterations
        indeg[u] ← indeg[u] + 1
② Create a list L of vertices with
   in-degree zero.                                 — O(n) time
Iterations: While L is not empty:
① Remove a vertex, say v from L and G.
② For each out-nbr u of v:                         — dv iterations
      indeg[u] ← indeg[u] - 1      }  O(1)
      If indeg[u] = 0, add u to L. }  time
```

(d) (2 points) Prove that the above algorithm runs in time $O(n + m)$.

For each vertex v: we spend O(dv) time in preprocessing its outneighbors, O(dv) time to decrement indegrees of its outnbrs when we remove v, and O(1) time to initialise indeg[v].

$\therefore$ Total time spent $= O(n + \Sigma_v d_v)$

$= O(n+m)$