Minor Exam COL 106. Solutions and Grading Guidelines

Maximum marks: 50

1. **Question 1 (8 marks)**

```
ReverseDoublyLinkedList(front, rear) {

        /* return L is L is of length 0 or 1 */
        If (front==NULL or front== rear)   (1 mark for base case; 0.5 for partially correct base case)
                return without any changes
        p=front.next
        ReverseDoublyLinkedList(p, rear)  (2 marks for correct recursive call)
        q=front
        front=p (0.5 marks)
        rear.next=q (1 mark)
        q.previous=rear  (1 mark)
        q.next=NULL (1 mark)
        front.previous=NULL  (1 mark)
        rear=q (0.5 marks)
}
```

The above solution moves the front node to the tail of the list. Solutions which move the rear node to the head of the list or move both the front and rear nodes to the tail and head shrinking the list from both ends are also graded using a similar rubric.

2. **Question 2 (8 marks)**

We maintain two lists,

A. one list maintains the queue in the front to rear order as in the usual implementation of a queue.
B. The other list M is as follows: suppose the order of the elements from rear to front in the Q is a_1, a_2, a_3,..., Then the first element of M is a_1, the second element is the first element in the sequence a_1, a_2, a_3,..., which is less than a_1 and so on.
   For example, if the rear to front order in Q is 4, 5, 2, 7, 1, then M will be 4,2,1 (2 marks). (Note that the last element of M will be the smallest element.)

C. Now let us check each operation:

a. **Dequeue**: we remove the first element from Q. If it happens to be the last element of M, we remove this from M also. No change is needed in M. (1 mark)
b. **Enqueue**(x): We add x to the rear of Q.
The first element of M should be x, but in order to maintain invariant of M, we have the following procedure: (2 marks)

Let p be the first node in M

While the key at p is more than x
q=next element in M after p
Remove p from M
p = q
Insert x at the beginning of M.

c. **FindMin**: Return the last key in M (1 mark)

D. To see the running time, we only need to worry about the Enqueue operation. But note that once a key is deleted from M, it doesn't get added to M again. So the number of insertions in M is at most n. (2 marks)

Alternate Approach - Two Stacks

We maintain two stacks, the enqueue stack and the dequeue stack. Both stacks contain (value, prefix_min), i.e., the value along with the min value in the stack below it. So a stack (top) 1, 5, 3, 10 will be stored as: [(1, 1), (5, 3), (3, 3), (10, 10)] (2 marks)

Operations:

a. **Enqueue(x)** - Find the current min **m** of enqueue stack by peeking the top element. Push **(x, min(x, m))** onto the enqueue stack (1 marks)
b. **Dequeue()** - If the dequeue is stack is empty, push all elements in the enqueue stack to the dequeue stack in reverse order (i.e., the top of enqueue stack will become the bottom of dequeue stack), and recalculate the prefix min on each step. Then, simply pop an element from the top of dequeue stack. (2 marks)
c. **FindMin()** - Peek the min values on both stacks and return the minimum of the two (1 marks)

3. **<u>Question 3 (7 marks)</u>**

Intended approach:

We maintain a heap H of size k and k indices, one for each array (1 mark).
Let the arrays be denoted $A_1, A_2, \ldots, A_k$. Let the k index variables be $i_1, i_2, \ldots, i_k$, each of which is initialized to 0 and we add the first element of each of the arrays to the heap H. Let i index the output array B and it is initialized to 0.

While (H is not empty)
(x,j)= deleteMin(H) *(1+1: deleteMin+ storing element along with index of array)*
Since x belongs to array A_j
Increase i_j *and insert* $(A_j[i_j], j)$ *to the heap* H *(1 mark)*
(if x was the last element of A_j*, then we do not insert anything to* H*)*
$B[i] = x$ *and increase* i *(1 mark)*

Justification: Output array contains the smallest i elements when the output array index is at i. This can be shown by a simple induction on i. (1 mark)

Time & Space complexity: Each deletemin or insert operation takes $O(\log k)$ time, and the while loop will run for most nk iterations. For space complexity, the heap will contain at most k elements and the k indices for each array, hence O(k). (0.5+0.5 mark)

Alternate approach (Partial marks):

- If the algorithm & its time complexity are correct but space complexity is wrong (2 mark)
- If the time complexity or algorithm is incorrect (0 mark)

4. **Question 4 (7 marks)**

By induction over the number of levels in the tree (or the number of nodes in the tree).
We will prove that n_l = N_{l-1} +1, where n_l denotes the number of nodes at level l, and N_{l-1} denotes the number of keys in the level up to level l-1. (Note that root is level 0). (2 marks for correct statement of the induction hypothesis)

Base case: The root is at level 0. By construction, the number of its children is one more than the number of keys at the root. (1 mark)

Induction hypothesis:

Now, assume that n_l = N_{l-1} + 1
Consider level l+1. Then the number of nodes at level l+1 = number of keys in level l + number of nodes in level l (using induction hypothesis). Note that this equation is true for any level by design of an a-b tree . This is because for any node at level l-1 with r keys, it has r+1 children.
Therefore, n_{l+1} = number of keys in level l + N_{l-1} +1 (from the induction hypothesis). Since, N_{l-1} is the number of keys up to level l-1, the equation becomes: n_{l+1} = N_{l} +1. (2 marks for how to use the induction hypothesis + 2 marks for the argument after this)

5. **Question 5 (6 marks)**

Let x be the largest key in T_1 and delete it from T_1. (1 mark) (or a similar strategy by removing smallest key of T2)
This will change the height of T_1 by at most 1 – let T_1' be the new tree and h_1' be its new height.
If the two trees had the same height to begin with, we can just make a new root with key x and the two trees as its children (2 marks).

Another approach
Make T1 as the left child of the leftmost node of T2 and then balance. (or similar strategy make T2 as the right child of the rightmost leaf node of T1 and then balance)
This will satisfy the AVL tree condition property.
(1 mark for BST property , 1 mark for height balance property and 1 mark for running time.)

6. **Question 6 (8 marks)**

Claim: After iteration k, the first k elements are in the heap order. We can prove by induction on the number of elements (1 mark)

(1) the base case is true since one element is in heap order by definition.
(2) Assume holds true for k. (1 mark including base case)
(3) Then, at k+1 iteration, we can think of the algorithm as inserting a new element at the index k+1, and calling upheap. Since, first k elements are in heap order by inductive hypothesis, the only out of order element is (potentially) at k+1, which is placed in the order by upheap operation. (2 marks)

Time Complexity: (assume n = 2^k for some integer k):

log(1) + log(2) + .. + log(n) <= T(n) <= log (1) + log(2) + .. log(n) + n

(1 mark for upper bound and 1 mark for lower bound)
(note that there is an input, namely, keys in decreasing order for which every new key travels all the way to the root, and hence the lower bound on the running time is achieved). ……*(1)*

(1 mark: proof of lower bound)

Now: let us analyze f(n) = log(1) + log(2) + … log(n)

Clearly, f(n) <= n log n (since log k < log n for k < n).

(1 mark: proof of upper bound)
Now, f(n) >= n/2 log (n/2) (take the last n/2 terms of the expression, and replace log k by log n/2) ……..*(2)*

From (1) and (2) f(n) = $\theta(n \log n)$

Since, f(n) <= T(n) <= f(n) + n, T(n) = $\theta(n \log n)$

Other grading notes: Showing a run of the algorithm on an example is not a valid proof. There are no marks for that or for simply detailing the steps of the algorithm without arguing for its correctness.

7. **Question 7 (6 marks)**

FindkeysRange(Node v, int a) {

If v is a leaf return 0. (1 mark)

Case 1: If (v.key <= a) (1 mark)

Return 1+v.leftchild.numdesc+FindKeysRange(v.rightchild,a) (1 mark)

Case 2: If (v.key > a) (1 mark)

Return FindKeysRange(v.leftchild, a) (1 mark)

}

To see the running time, note that after each recursive call, we go down one level in the tree. (1 mark)