Name:_____ **Entry No:**_____

COL106: Data Structures. I semester, 2015-16.
Major
10 AM to 12 PM, 24 November 2015.

Attendance Sheet Serial Number: _____

Question	1 (4 marks)	2 (8 marks)	3 (7 marks)	4 (2 marks)	5 (3 marks)	6 (8 marks)	Total (32 marks)
Marks							

Please write your name on every page and enter your serial number in the box above. **If you miss out any of these: -1 and no rechecking.**

No pseudocode means: For every loop you use, you must explain what it will do to the input. You cannot write things like "x = x + y", you must explain the significance of every step in words. You cannot write "for i = 1 to ..." or "while <condition>...", you must *explain* what the loop achieves. In summary: if we need to interpret how your description will treat a particular input then it is pseudocode.

Q1. (4 marks) Marks:

You are given two data structures: one is a storage stack (S) and the other one is a display queue (D). A sequence of alphabets **abcdefgh** is used as input. Starting with **a** and ending with **h**, one alphabet at a time from the sequence is either pushed onto S or enqueued in D. An alphabet popped from S is enqueued in D. There is no other memory available. At the end all the items of D are dequeued and the resulting output is **cedbghfa**. Show the sequence of operations that leads to this output.

**** Solution ****

The sequence is

push a; push b; enqueue c; push d; enqueue e; pop d; enqueue d; pop b; enqueue b; push f; enqueue g; enqueue h; pop f; enqueue f; pop a; enqueue a

Marks. In case the "enqueue" is missing after a "pop", do not deduct marks. But the "pop" needs to be there, e.e., the 5th operation is "pop d" followed by "enqueue d". Apart from that, it's a simple question so the grading can be stringent.

Q2. (Total 8 marks) Marks:

Here is another algorithm for finding a minimum weight spanning tree of a graph $G = (V, E)$, it's called Prim's algorithm. It proceeds by iteratively increasing the size of the minimum spanning tree by one vertex, finally ending with a minimum spanning tree that spans all of G.

1. Initially let the spanning tree T be just a single vertex x.
2. Find the minimum weight edge between a vertex in T and a vertex in the remaining graph $G - T$. Add that edge to T.
3. Repeat step 2 till T contains all the vertices of V.

In this question we are not concerned with proving the correctness of this algorithm. We actually want to implement it efficiently. For that, we first need a new heap operation.

Name:_____ **Entry No:**_____

Q2.1. (2 marks) Given a min-heap on n elements, explain **in words (no pseudocode)** how to reorganise the heap in $O(\log n)$ time when a single element's priority value is decreased. Note that the element whose priority value is to be decreased could be anywhere in the heap. Note also that you have to provide justification that the running time is $O(\log n)$.

**** Solution ****

Since decreasing an element's priority value cannot cause a conflict below–it can only cause a conflict above–this leads to a simple bubble-up as in the case of insertion in a min-heap. Hence the time is $O(\log n)$.

Marks. If the student has said that it bubbles up but has not argued that there is no conflict below, then -1 should be given.

Q2.2. (6 marks) Explain in words **(no pseudocode)**, how the operation described above, i.e. reducing the priority of an element in a heap, can be used to make Prim's algorithm run in time $O(m + n \log n)$ where m is the number of edges in E and n is the number of vertices in V.

**** Solution ****

The basic idea of the solution is that we will create a heap on n elements corresponding to the n vertices. The priority value corresponding to the vertex v is the weight of the lowest weight edge incident on v that we have seen so far. Initially all the priorities have value $+\infty$.

We choose a first vertex x arbitrarily and reduce its priority value in the heap to 0 and then we delete it from the heap. This is our starting vertex. We then modify the priority value of all the nodes that have an edge to x, according to the weight of the edge joining them to x. Store the edge name along with the priority value.

We now remove vertices from the heap one at a time as follows.

- Delete-min from the heap. Say it is u.
- This u has an edge stored with it. Add that edge to the MST.
- Look at all the edges incident on u that have not already been visited. If any neighbours of u, say w, has a priority value higher than the weight of the edge (u, w), decrease the priority of w and store the edge (u, w) with it, overwriting whatever was stored earlier. This way at all times the heap entry contains the weight of the smallest edge (amongst all edges seen so far) incident on the vertex, and also the name of the edge.

Each run of this loop involves one delete-min operation which takes $O(\log n)$ time. There are n iterations, so the time for the delete-mins is $O(n \log n)$. Each edge is viewed at most twice and leads to some priority value being decreased. If we look at it this way, we should say that every time we examine an edge in the third step of the algo above, we should get a time $O(\log n)$ which will give us a total of $O(m \log n)$. But one more observation gives us the correct answer: Every time we decrease the priority of a node it bubbles up a little bit. The sum of all the bubbling up can't be more than the height of the heap which is $O(n \log n)$. Hence the total bubbling up takes $O(n \log n)$ steps at most. Since each edge has to be examined once, we also get an $O(m)$, which completes the proof.

Marks. This is a tricky one, the really difficult question of the exam. You will have to read it carefully. The break up is 4 marks for the alg and 2 marks for the analysis but only if the

algo is correct or close to correct. If the algo is wrong then there are no marks for analysis. The basic idea involves maintaining a heap for the vertices. If this basic idea is there then you can give 1 mark, even if the rest is completely wrong.

Q3. (Total 7 marks) Marks:

We are given a special kind of array, $A[]$, of size n that has the following property. It is possible to access $A[0]$ to $A[n/2-1]$ in 1 unit of time, $A[n/2]$ to $A[3n/4-1]$ in 2 units of time and so forth. In general it is possible to access an array location with index between $\sum_{j=0}^{i-1} n/2^{j+1}$ and $\sum_{j=0}^{i} n/2^{j+1}$ in time 2^{i-1}. For this array answer the following questions.

Q3.1. (2 marks) Assuming the array is used to store integers, what is the time taken to sum the elements of the array? Give a brief mathematical justification, not just the answer.

**** Solution ****

The time taken is $O(n \log n)$. They have to write a line or two to justify it. If only $O(n \log n)$ is written then give them at most 0.5 marks.

Q3.2. (1 mark) Is the time taken to sum the integers using the array asymptotically better or worse than if the integers had been stored in a simple linked list with the usual access times? Or is it asymptotically the same. Justify.

**** Solution ****

It is asymptotically worse since a linked list will take $O(n)$ time. We need them to say explicitly that linked list takes $O(n)$ time or uses "linear" time. Without that, give them only 0.5 marks.

Q3.3. (3 marks) What is the worst-case time taken to run a binary search algorithm if we have integers stored in sorted order in this special kind of array? Justify your answer mathematically.

**** Solution ****

In the worst case the binary search algorithm will always move to the right, going further and further and incurring higher and higher cost. The cost will be

$$\sum_{j=1}^{\log n} 2^{j-1} = O(n)$$

Even if the expression is not exactly as written above but basically correct, we can give full marks. But if the justification (that binary search always moves right in the worst case) is not given then give at most 1.5 marks.

Q3.4. (1 marks) Is the worst-case time taken to run binary search using the special array asymptotically better or worse than if the integers had been stored sorted in a simple linked list with the usual access times? Or is it asymptotically the same. Justify.

**** Solution ****

It is asymptotically the same since both are $O(n)$. Do not give more than 0.5 marks if the student has not explicitly mentioned that searching in a sorted linked list takes $O(n)$ time.

Q4. (2 marks) Marks:

Consider the 2-4 tree given in Figure 1.

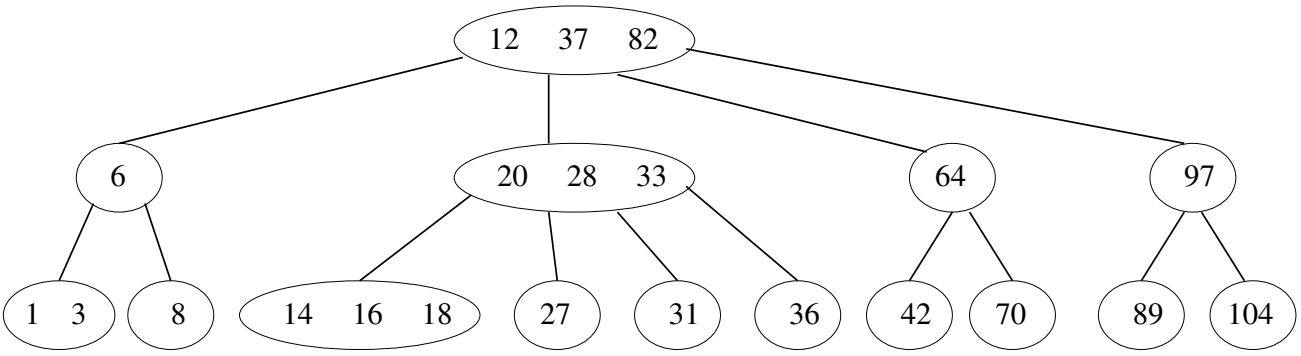

Figure 1: A 2-4 tree

Draw the state of the tree after inserting 17.

**** Solution ****

I am not drawing the solution here since this is very simple. Note that when 17 is inserted in the node containing 14, 16 and 18, then either 17 or 16 can be promoted. The same thing will happen at all three levels. Either way it is correct.

This is a simple question so **no partial credit**.

Q5. (3 marks) Marks:

Suppose we start with an AVL tree T . We do an insertion of a new key in the tree just using the binary search tree algorithm without yet doing any rotations. The result is a tree T'. It turns out that the deepest node in T' which is not balanced is the root, so we do an appropriate rotation at the root to produce the final AVL tree T''. If the height of T'' is 10, what were the heights of T and T' respectively ? Give an argument for your answer.

**** Solution ****

Firstly, the height of T is the same as the height of T'', i.e., 10, since a rotation at the time of insertion *always restores* the height of the subtree being rotated.

The height of T' is 11. This is because the height of T was 10 and an insertion led to an imbalance forming. This means that the heights of the subtrees of T were 8 and 9 (since if they were equal then a single insertion could not have caused an imbalance) and the subtree that was of height 9 received the new item and its height changed to 10 causing an imbalance. If the subtree's new height became 10, the trees new height became 11.

Marks. This is a little tricky so you will have to read the answer carefully. Give 1.5 marks for each part and try to see if the key idea is being communicated. If you can't understand what they are saying, that is good enough reason to give them no marks, since this question is

testing their understanding *and* their ability to communicate the understanding.

Q6 (Total 8 marks) Marks:

We are given a graph $G = (V, E)$, a weight function $w : E \to R_+$ (i.e. weights are all positive real numbers) and some vertex $s \in V$. We run Dijkstra's algorithm starting from s and assign labels $\ell[v]$ to each vertex $v \in V$ such that $\ell[v]$ is the length of the shortest path from s to v. Now, we add a new vertex u to V to form a new vertex set $V' = V \cup \{u\}$. Along with u we add positive weighted edges joining u to some of the vertices of V to form a new edge set E'.

Q6.1 (5 marks) Describe an efficient algorithm that takes the already computed labels $\ell[v]$ on the vertices of V along with the modified graph $G' = (V', E')$ to compute new labels $\ell'[v]$ for all the vertices of V' such that $\ell'[v]$ is the length of the shortest path from s to v in G'.

**** Solution ****

The idea here is as follows. First we label the new vertex based on the weights of its edges and the labels of its neighbours by choosing the least of the possible labels. This label is its final label. Then, once this label has been computed, we check all the neighbours of the new vertex. We "unfix" all the labels that decrease due to the new neighbour, and then "fix" the least of these. Now we examine the neighbours of the new fixed vertex and do the same with their labels, till we reach a stage where no labels can be "unfixed". At this point the algorithm ends.

Marks. Note that the statement of the question does not ask for a justification of the correctness of the algorithm, but we will keep 1 mark for this, since a question like this demands justification. For the algorithm itself you will have to read the answer carefully and see what is going on. The main idea is that we have to unfix labels and then use Dijkstra's logic to fix them again. If this main idea is there then you can give at least 2.5 marks. Otherwise give less than that.

Q6.2 (3 marks) If the vertices for which $\ell'[v] < \ell[v]$ are $v_1, v_2, \ldots, v_k$ and the degree of vertex v_i is denoted $d(v_i)$, write the running time of your algorithm as a function of k and $d(v_1), d(v_2), \ldots, d(v_k)$. You have to give an argument justifying your answer.

**** Solution ****

The running time is

$$O(k \log k + \sum_{i=1}^{k} d(v_i)).$$

This requires analysis of the algorithm given above. Essentially every edge of every node whose label changes is examined and, of course, those nodes are examined. The other nodes are not touched. Every time we examine an edge we update the label of a node. In a heap implementation this leads to a bubble up, just like in Q2. As in Q2, the total bubbling per node is at most the height of the heap, which is $(\log k)$ in the worst case. This justifies the answer above. Note that if the algorithm is not correct in the previous part then you should not give more than 1 mark even if the answer of this part is correct.

Also note, that since I had not properly explained the running time of Dijkstra in class we will give full marks (if the algorithm is correct) also for the following answers.

$$O(k + \sum_{i=1}^{k} d(v_i)),$$

because in class I had suggested that the running time of Dijkstra was $O(n + m)$, and,

$$O(\log k(k + \sum_{i=1}^{k} d(v_i))),$$

because in Goodrich and Tamassia, the running time of Dijkstra is given as $O((n + m) \log n)$.
