CS21003 Algorithms I, Autumn 2013–14

End-Semester Test

Maximum marks: 80 | Time: 19-Nov-2013 (2:00–5:00 pm) | Duration: 3 hours

---

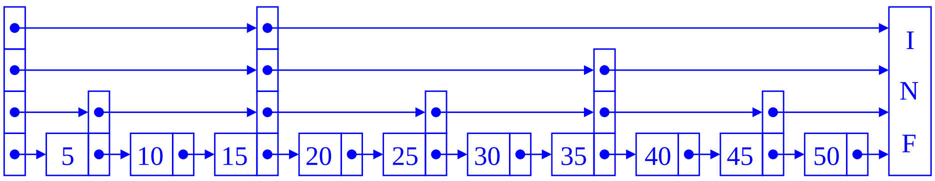
Roll no: ________________ **Name:** ____

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. Draw the skip list storing the following ten key values at the indicated levels. **(10)**

Key value	10	20	30	40	50	5	25	45	35	15
Level	0	0	0	0	0	1	1	1	2	3

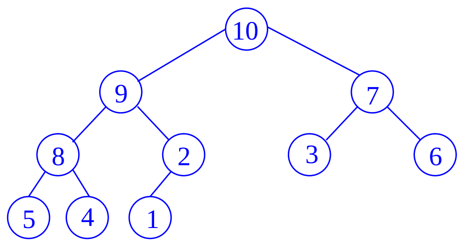
Solution The skip list is shown in the following figure.



2. Prove or disprove the following two assertions.

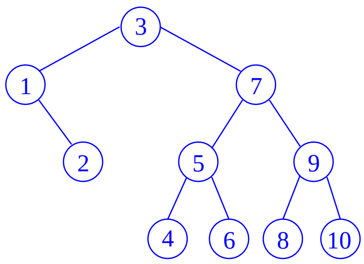
(a) The second minimum in any max-heap with $n \geqslant 10$ pairwise distinct keys must be found in a leaf node. **(5)**

Solution *False.* Here is a counterexample.



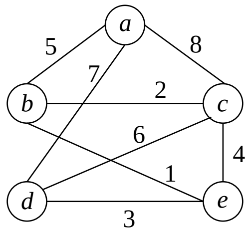
(b) The minimum in any AVL tree with $n \geqslant 10$ keys must be found in one of the last two levels. **(5)**

Solution *False.* Here is a counterexample.

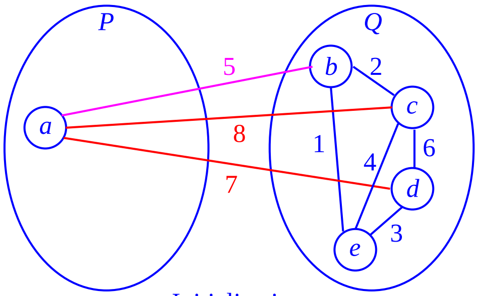


Roll no: __________________ **Name:** ____

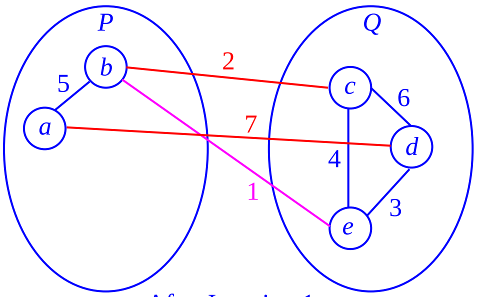
3. Demonstrate how Prim's algorithm computes the minimum spanning tree of the following graph. Let a be the root of the MST. Show the initialization and iterations in Prim's algorithm applied on this graph. **(10)**



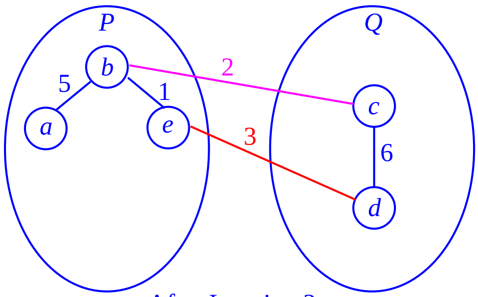
Solution The initialization and the four iterations are described in the following figure.



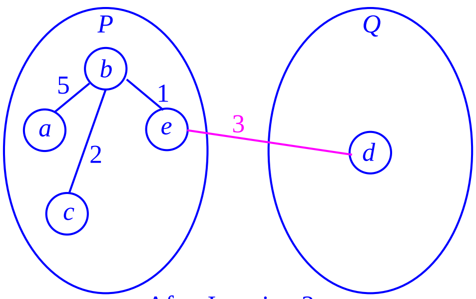
Initialization



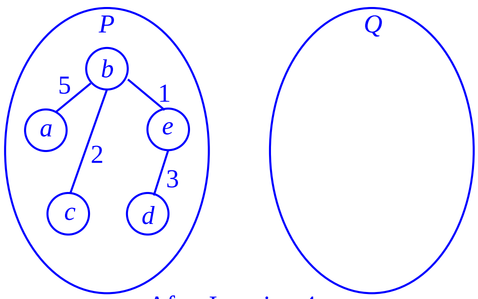
After Iteration 1



After Iteration 2



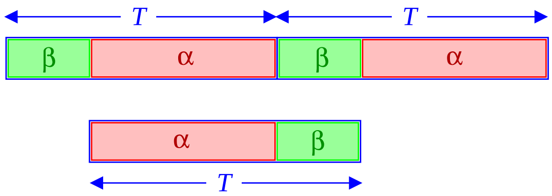
After Iteration 3



After Iteration 4

4. Let T be a string of length m. Propose an $O(m)$-time algorithm to determine whether T can be represented as $T = \alpha\beta = \beta\alpha$ for two non-empty strings α and β. **(10)**

Solution Search for T in TT using the KMP string-matching algorithm. The first and the last positions are trivial matching positions. If there is any non-trivial matching position, we have a representation of T as in the question. The following figure demonstrates this.



5. Let S and T be strings of lengths n and m respectively, with $m \leqslant n$. T is called a *cover* of S if every position in S belongs to some match of T in S. For example, $T = aba$ is a cover of $S = ababaaba$. Indeed, the three matches of T in S cover all the positions in S as demonstrated here: $\overline{ab\underline{a}}\underline{ba}\overline{aba}$. On the other hand, $T = ab$ is not a cover of $S = ababaaba$ as demonstrated here: $\overline{ab}\underline{ab}\boldsymbol{a}\overline{ab}\boldsymbol{a}$ (the uncovered positions are shown in bold face). Propose an $O(n+m)$-time algorithm to determine whether T is a cover of S. **(10)**

Solution We first run the KMP string-matching algorithm for finding all matches of T in S. We assume that there are t matches, and the KMP algorithm prepares an array M of size t storing the match positions in sorted order. We then check whether there is a gap between any two consecutive matches.

> Run the KMP algorithm: $t = \mathrm{KMP}(S, T, n, m, M)$.
> Initialize nextMatchReqd = 0.
> for $i = 0, 1, 2, \ldots, t-1$ (in that order) {
>   if ($M[i] >$ nextMatchReqd), then return *False*.
>   Update nextMatchReqd $= M[i] + m$.
> }
> If (nextMatchReqd $\geqslant n$), return *True*.
> Return *False*.

The KMP algorithm takes $O(n+m)$ running time. The remaining part runs in $O(t)$ time. Since $t \leqslant n - m + 1$, the overall running time is $O(n+m)$.

6. Let $G = (V, E)$ be a directed graph. A vertex s in G is called a *source* if its in-degree is zero. Likewise, a vertex t in G is called a *target* (or *sink*) if its out-degree is zero.

(a) Propose an $O(|V| + |E|)$-time algorithm to locate all the sources and all the targets in G. **(5)**

Solution We assume the adjacency-list representation of the graph. We use two arrays S and T indexed by V to mark whether a vertex can be a source or a target (respectively). Initially, we mark each vertex as a potential source and a potential target. For each (directed) edge $(u, v) \in E$, we unmark u in the target array T, and unmark v in the source array S. After all the edges are considered, those vertices that are still marked in S are the sources, and those vertices that are still marked in T are the targets.

(b) Prove that a directed acyclic graph must contain at least one source and at least one target. **(5)**

Solution Assume that a DAG does not contain a source. This means that for every vertex v, there is (at least) an edge $(u, v) \in E$. Let $n = |V|$. We start with any arbitrary vertex v_0, and obtain a sequence of vertices $v_1, v_2, v_3, \ldots, v_n$ such that $(v_i, v_{i-1}) \in E$ for all $i = 1, 2, 3, \ldots, n$. Since G contains only n vertices, there must be a repetition in $v_0, v_1, v_2, \ldots, v_n$. Let $v_i = v_j$ with $0 \leqslant i < j \leqslant n$. By construction, $v_j, v_{j-1}, v_{j-2}, \ldots, v_{i+1}, v_i$ is a cycle in G, a contradiction.

Analogously, the existence of a target in G can be proved.

Roll no: ____________________ **Name:** ____

(c) Let $G = (V, E)$ be a directed acyclic graph. Propose an $O(|V| + |E|)$-time algorithm to count the total number of paths from all the sources in G to all the targets in G. **(10)**

Solution Let $s_1, s_2, \ldots, s_k$ be all the sources and $t_1, t_2, \ldots, t_l$ be all the targets in G (these can be identified in $O(|V| + |E|)$ time by Part (a)). We convert G to a new DAG G' whose vertex set contains two additional vertices s and t. We add the edges (s, s_i) for all $i = 1, 2, \ldots, k$ and also the edges (t_j, t) for all $j = 1, 2, \ldots, l$. G' is a DAG with a unique source s and a unique target t. Moreover, the count of all (s_i, t_j) paths (for all i, j) in G is the same as the count of all (s, t) paths in G'. The size of G' continues to remain $O(|V| + |E|)$.

We make a topological sorting of the vertices in G'. This can be done in $O(|V| + |E|)$ time. Let the listing be $s = v_0, v_1, v_2, \ldots, v_n, t = v_{n+1}$. We use an array C indexed by the vertices in G' to store the count of paths from s to the vertices.

> Initialize $C[v_0] = 1$ and $C[v_i] = 0$ for all $i = 1, 2, 3, \ldots, n+1$.
> For $i = 0, 1, 2, \ldots, n$ {
>   For all edges (v_i, v_j) in G', set $C[v_j] = C[v_j] + C[v_i]$.
> }
> Return $C[v_{n+1}]$.

Since there are no back edges (that is, edges (v_i, v_j) with $i > j$), the for loop does not miss a path from s to t. With the adjacency list representation of G', this phase can again be finished in $O(|V| + |E|)$ time.

The introduction of the new vertices s, t could have been avoided. In that case, we start by setting $C[s_i] = 1$ for all the sources s_i in G. At the end, we return $C[t_1] + C[t_2] + \cdots + C[t_l]$. However, a topological sorting of G is necessary for the correctness of this algorithm.

7. You are given n real intervals (a_i, b_i) standing for the running times of n processes. That is, (a_i, b_i) stands for a process that starts at time a_i and finishes at time b_i. Assume that $a_i < b_i$ for all i. Your objective is to schedule *all* the processes, using as few processors as possible. You are not allowed to schedule two or more conflicting processes (that is, processes having overlapping running times) on the same processor. A process running in a processor is allowed to continue until it finishes. Propose an efficient greedy algorithm to solve this problem. Supply an optimality proof for your greedy algorithm, and deduce its running time. **(10)**

Solution We first sort the intervals with respect to their left endpoints. We then try to schedule the intervals one by one in this sorted order. A processor with earliest finish time is chosen for each scheduling. If no existing processor can accommodate a new job, a new processor is introduced. We use a min-priority queue Q to store the right endpoints of the intervals currently scheduled—only one entry per processor. Processors are numbered $1, 2, 3, \ldots$. Each entry (p, f) in Q stores a processor number p and the finish time f of the last process assigned to p. The heap-ordering is with respect to the second component f.

> Sort the given intervals with respect to their left endpoints.
> Let the sorted list be $(a_0, b_0), (a_1, b_1), (a_2, b_2), \ldots, (a_{n-1}, b_{n-1})$.
> Initialize nproc $= 0$, the min-priority queue Q to empty, and $f = a_0 - 1$.
> For $i = 0, 1, 2, \ldots, n-1$ {
>   If ($i > 0$), set $(p, f) = \min(Q)$. /* Q is empty only for $i = 0$ */
>   If ($f > a_i$) {
>   Use a new processor: nproc++.
>   Set $p =$ nproc.
>   } else {
>   Make a deleteMin in Q.
>   }
>   Schedule the i-th process (a_i, b_i) to processor number p.
>   Insert (p, b_i) in Q.
> }

For the correctness, let x be a real number. The number of intervals (a_i, b_i) to which x belongs is denoted by $N(x)$. Let $N = \max_{x \in \mathbb{R}} N(x)$. We cannot schedule all the processes with less than N processors. The above algorithm clearly uses exactly N processors and is therefore optimal.

The initial sorting of the intervals can be done in $O(n \log n)$ time. Subsequently, there are n deleteMin and insert operations in Q. The maximum size of Q is N, since Q stores only one entry for each processor. So the total time for preparing the schedule (after the sorting phase) is $O(n \log N)$. Finally, $N \leqslant n$, so the overall running time of this greedy algorithm is $O(n \log n)$.

For rough work and leftover answers