Data Structures and Programming

Spring 2016, Final Exam.

June 21, 2016

1. (15 pts) True or False? (Mark ◯ for True; × for False. Score = max{0, Right - $\frac{1}{2}$ Wrong}. No explanations are needed.

   (1) Let A_1, A_2, and A_3 be three sorted arrays of n real numbers (all distinct). In the comparison model, constructing a balanced binary search tree of the set $A_1 \cup A_2 \cup A_3$ requires $\Omega(n \log n)$ time.
   × **False** (Merge A_1, A_2, A_3 in linear time; then pick recursively the middle as root (in linear time).)

   (2) Let T be a complete binary tree with n nodes. Finding a path from the root of T to a given vertex $v \in T$ using breadth-first search takes $O(\log n)$ time.
   × **False** (Note that the tree is NOT a search tree.)

   (3) Given an unsorted array $A[1...n]$ of n integers, building a max-heap out of the elements of A can be performed asymptotically faster than building a red-black tree out of the elements of A.
   ◯ **True** ($O(n)$ for building a max-heap; $\Omega(n \log n)$ for building a red-black tree.)

   (4) In the worst case, a red-black tree insertion requires $O(1)$ rotations.
   ◯ **True** (See class notes)

   (5) In the worst case, a red-black tree deletion requires $O(1)$ node recolorings.
   × **False** (See class notes)

   (6) Building a binomial heap from an unsorted array $A[1...n]$ takes $O(n)$ time.
   ◯ **True** (See class notes)

   (7) Insertion into an AVL tree asymptotically beats insertion into an AA-tree.
   × **False** (See class notes)

   (8) The subtree of the root of a red-black tree is always itself a red-black tree.
   × **False** (Root must be black)

   (9) Decreasing the key of a node in a binomial heap can be done in $O(1)$ time.
   × **False** ($O(\log n)$ time)

   (10) Decreasing the key of a node in a leftist heap can be done in $O(\log n)$ time.
   ◯ **True** (Cut the subtree; then merge with the remaining.)

   (11) The union-find ADT plays a key role in implementing the Dijkstra's shortest path algorithm.
   × **False** (The priority queue ADT is the key.)

   (12) In the disjoint set ADT, the so-called *rank* of a node x in the union-by-rank heuristic refers to the number of nodes in the subtree rooted at x.
   × **False** (See class notes)

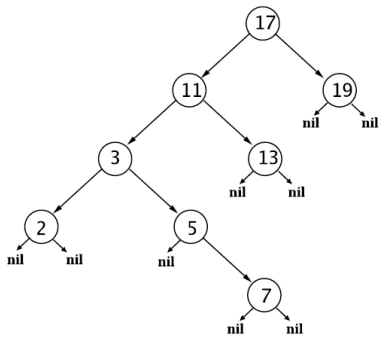
   (13) If a data structure supports a FOO operation such that a sequence of n FOO operations takes $O(n \log n)$ time in the worst case, then the amortized time of a FOO operation is $\Theta(\log n)$.
   ◯ **True** (Definition of amortized time.)

   (14) Building an AA-tree from an unsorted array of n elements can be done in $O(n)$ time.
   × **False** (Otherwise, sorting can be done in $O(n)$ time.)

   (15) Prim's algorithm for minimum spanning trees is based on the technique of dynamic programming.
   × **False** (Based on a greedy approach.)

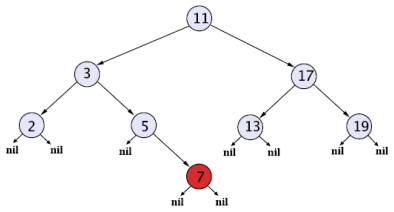
2. (15 pts) Answer the following questions:

   (a) Assign the keys 2, 3, 5, 7, 11, 13, 17, 19 to the nodes of the binary search tree in Figure 1 so that they satisfy the binary-search-tree property.

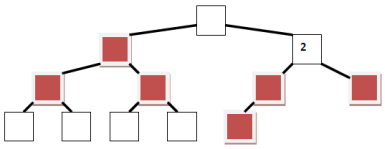


(b) Explain why this binary search tree cannot be colored to form a legal red-black tree.
Sol. We prove this by contradiction. Suppose that a valid coloring exists. In a red-black tree, all paths from a node to descendant leaves contain the same number of black nodes. The path (17, 19, NIL) can contain at most three black nodes. Therefore the path (17, 11, 3, 5, 7, NIL) must also contain at most three black nodes and at least three red nodes. By the red-black tree properties, the root 17 must be black and the NIL node must also be black. This means that there must be three red nodes in the path (11, 3, 5, 7), but this would mean there are two consecutive red nodes, which violates the red-black tree properties. This is a contradiction, therefore the tree cannot be colored to form a legal red-black tree.

(c) The binary search tree can be transformed into a red-black tree by performing a single rotation. Draw the red-black tree that results, labeling each node with "red" or "black." Include the keys from part (a).
Sol. Rotate right around the root. There are several valid ways to color the resulting tree. Here is one possible answer (all nodes are colored black except for 7).



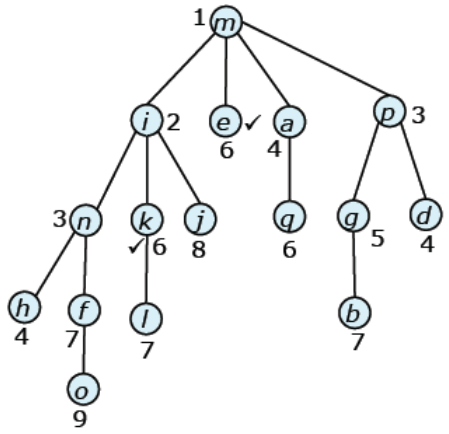
3. (10 pts) Given the structure of a min-heap as sketched in Figure 2, where the second-smallest value in the set is marked with a 2. Mark a 4 for each node that can possibly contain the fourth-smallest value in the set. Assume that there are no duplicate node values.
Sol. The shaded nodes are the answer. Any of the nodes shown below the 2 could be the 4th smallest element. If the left child of the root is the 3rd smallest element, then either of its children, but not its grandchildren, could be the 4th smallest. If the 3rd smallest is under the 2, then the left child of the root could also be the 4th smallest.



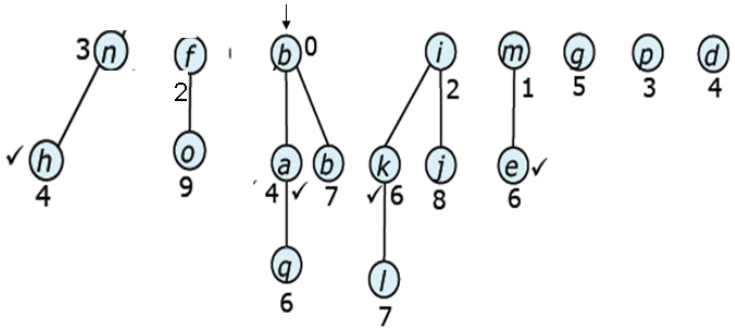
4. (10 pts) In the Fibonacci heap shown in Figure 3, the numbers are the key values and marked nodes are annotated by a check mark.

(a) (2 pts) What is the potential of this Fibonacci heap?
$6 + 2 \times 5 = 16$

(b) (4 pts) Show the heap that results from performing a delete-min on the heap.



(c) (4 pts) Show the heap that results from decreasing the key value of node f from 7 to 2 on the original heap (i.e., the heap in Figure 3).



5. (12 pts) Insert the keys $\{33, 54, 69, 74, 18, 19\}$ (in the order given) into the table of size 10 ($A[0..9]$) using each of the following three open addressing mechanisms. Show the contents of the table.

(a) hash function $h(x) = x \ mod \ 10$ with *linear probing*

Sol.

0	1	2	3	4	5	6	7	8	9
19			33	54	74			18	69

(b) hash function $h(x) = x \ mod \ 10$ with *quadratic probing*

Sol.

0	1	2	3	4	5	6	7	8	9
19			33	54	74			18	69

(c) hash function $h(x) = x \ mod \ 10$ with *double hashing* using $h'(x) = 1 + (x \ mod \ 9)$ as the secondary hash function

Sol.

0	1	2	3	4	5	6	7	8	9
	19		33	54			74	18	69

6. (15 pts) Consider the graph in Figure 4 and its adjacency list representation:

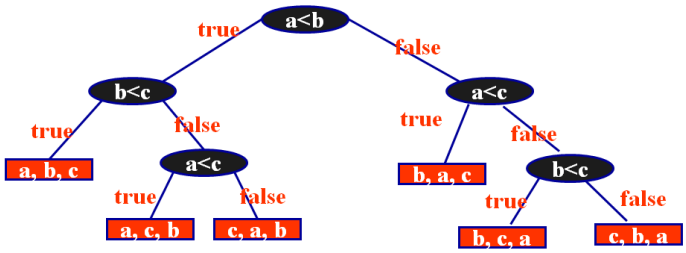
(a) (5 pts) Write out the order in which the vertices in the graph would be visited if a *Breadth-First Search* were started from vertex 3 given the above adjacency list representation of the graph.
3 2 5 1 6 8 4 7

(b) (5 pts) Write out the order in which the vertices in the graph would be visited if a *recursive Depth-First Search* were started from vertex 3 given the above adjacency list representation of the graph.
3 2 1 4 6 8 7 5

(c) (5 pts) Write out the order in which the vertices in the graph would be visited if a *non-recursive Depth-First Search* were started from vertex 3 given the above adjacency list representation of the graph.
3 1 4 6 8 7 5 2

(NOTE: There is only one correct answer since you must use the given adjacency list and follow the rules of accessing such a data structure.)

7. (9 pts) Draw the decision tree with respect to the *insertion-sort* algorithm on an array $A[0..2]$ containing three keys a, b, c.



8. (14 pts) Say we are given a weighted directed graph G with strictly positive edge weights $w(u, v)$ and a source node s in G. We would like to modify Dijkstra's shortest-path algorithm to produce a count of the number of different minimal length paths from s to v for every node v. Recall that Dijkstra's algorithm builds a set X of nodes incrementally, starting with $X = \{s\}$. Normally, the algorithm maintains a value $D(v)$ for each node v giving the length of the shortest path from s to v through only intermediate nodes in X. We will modify the algorithm to maintain an extra value $N(v)$ giving the number of paths of length $D(v)$ from s to v through only intermediate nodes in X. Describe how to initialize $D(v)$ and $N(v)$ and how to update the values of $D(v)$ and $N(v)$ when a new node u is added to X. To this end, fill in the following 8 blanks.

(a) Initialize $D(v)$ as in Dijkstra's algorithm.
Initialize $N(s) =$ .....**1**....., $N(v) =$ .....**1**..... for all edges (s, v), and $N(v) =$ .....**0**..... for all other nodes.

(b) To update N when a node u is added to X, for each edge (u, v),

- if .....$D(u) + w(u, v) < D(v)$....., set $N(v) = N(u)$;
- if .....$D(u) + w(u, v) = D(v)$....., set $N(v) =$ .....$N(u) + N(v)$.....;
- if .....$D(u) + w(u, v) > D(v)$....., do nothing.

Update $D(v)$ for all edges (u, v) as in Dijkstra's algorithm.

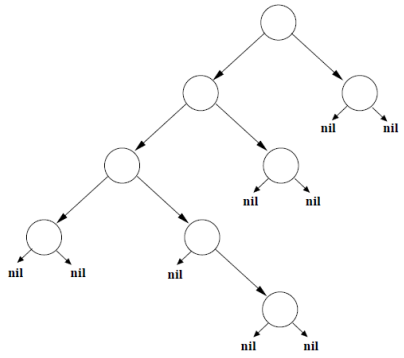


Figure 1: A red-black tree

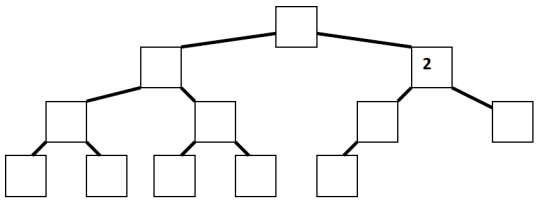


Figure 2: A min-heap

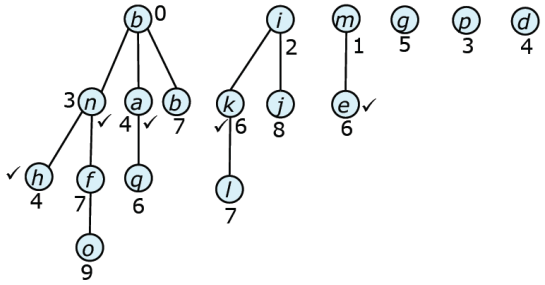


Figure 3: A Fibonacci heap

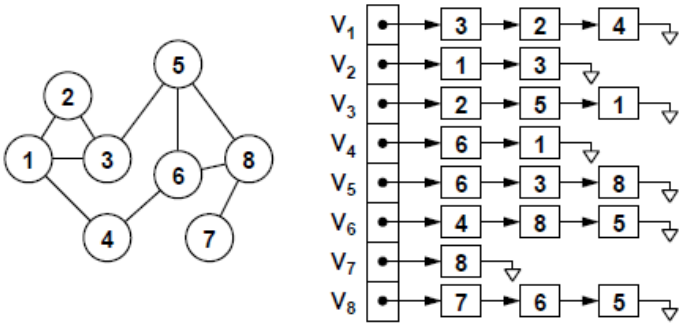


Figure 4: Adjacency list of a graph