CS 410, Spring 1999: Review Solutions

1. Notice that I’ve corrected the problem. It should say “Suppose $T(n) = T([an]) + T([bn]) + [cn]$, where $a+b < 1\ldots$” Choose $d$ such that $d = \max(c/(1-a-b), T(1))$. Note for future reference that since $d \geq c(1-a-b)$, it follows $d \geq d(a+b)+c$.

We show by strong induction that $T(n) \leq dn$ for all $n \geq 1$.

The base case is easy, since $T(1) \leq d$, by choice of $d$. Suppose $T(n') \leq dn$ for all $n' \leq n$; we show that $T(n+1) \leq d(n+1)$. Since $a,b < 1$, we must have $[a(n+1)] \leq n$ and $[b(n+1)] \leq n$. Using the definition of $T$ and the induction hypothesis, we get

$$T(n+1) = T([an+1]) + T([bn+1]) + [c(n+1)]$$
$$\leq d[a(n+1)] + d[b(n+1)] + [c(n+1)]$$
$$\leq da(n+1) + db(n+1) + c(n+1)$$
$$\leq (d(a+b) + c)(n+1)$$
$$\leq d(n+1)$$

The last step uses the fact that $d(a+b) + c \leq d$, which we observed above followed from the fact that $d \geq c(1-a-b)$.

2. In problem 10.3-5, suppose we want to find the $k$th best out of $n$. We first find the median of $n$ elements; call it $m$. (To clarify matters, if $n$ is even, we assume that the median actually returns $[n/2]$.) If $k = [n/2]$, we are done. If $k > [n/2]$, then we construct an array consisting of the $[n/2] - 1$ elements of the original array that are greater than $m$; that means we are now interested in finding the element that is $(k - [n/2])$th best out of the elements in this array. If $k < [n/2]$, we construct an array consisting of the $n - [n/2]$ elements of the original array that are less than $m$; we are now interested in finding the element that is $k$th best of these elements. We continue in this way recursively. This gives us the following algorithm (where MEDIAN(A) is the black-box algorithm that you are presumed to be given for finding the median element of $A$; for simplicity, I'll assume that all elements of $A$ are distinct). Here is the code:

SELECT(A, k, n) [Select the $k$th best element in $A[1..n]$

1 $m \leftarrow$ MEDIAN(A)
if $k = \lceil n/2 \rceil$ then return $m$
if $k < \lceil n/2 \rceil$
  then $j \leftarrow 1$
  for $i \leftarrow 1$ to $n$
    do if $A[i] < m$
      then $B[j] \leftarrow A[i]$
      $j \leftarrow j + 1$
  return SELECT($B, k, n - \lceil n/2 \rceil$)
if $k > \lceil n/2 \rceil$
  then $j \leftarrow 1$
  for $i \leftarrow 1$ to $n$
    do if $A[i] > m$
      then $B[j] \leftarrow A[i]$
      $j \leftarrow j + 1$
  return SELECT($B, k - \lceil n/2 \rceil, \lfloor n/2 \rfloor - 1$)

We still have to show that this algorithm runs in linear time, but that's easy. Let $T(n)$ be the worst-case running time of SELECT($A, k, n$). Then in the worst case, we have to do some linear-time work (to create the array $B$) and apply SELECT recursively to an array of size $\leq n/2$. Thus, we get that $T(n) \leq T(n/2) + cn$. Now applying the Recurrence Theorem, we see that $T(n) = O(n)$.

12. (a) Suppose $G$ is represented by the matrix $A = (a_{ij})$. Start with an array $R[1..n]$ with each entry initialized to 0. For each $u$ such that $a_{ru} = 1$, set $R[u] = 1$. Also set $R[r] = 1$. Thus, so far, $R[v] = 1$ iff $v$ is reachable by a path of length $\leq 1$ from $r$. Now for each of the $d$ vertices $v$ other than $r$ such that $R[v] = 1$, for each $u$ such that $A_{vu} = 1$, set $R[u] = 1$. This works. Altogether, we check $d + 1$ rows in the matrix. Since each row has $n$ entries, this takes time $O(nd)$.

(b) Go through the loop of BFS($G, r$) twice (that is, starting at line 9 on p. 470). The first pass will get you all the vertices distance 1 from $4$; the next pass will get you all vertices distance 2 from $4$. Doing the first 8 lines of the algorithm takes time $O(n)$. Since the degree of of $r$ is $d$, after the first pass, you will have $d$ vertices; after the second pass, you will have at most $d^2$ vertices. Thus, going through the loop can be done in time $O(d^2)$ (with an adjacency list representation). Altogether, this gives us an $O(n + d^2)$ algorithm. Note that because we initially colored all vertices white and only add a vertex when it is first colored gray, we will end up with a list without duplicates. We can get an $O(d^3)$ algorithm by not bothering to color the vertices and going through the loop twice. This will give us a list with $d^2$ elements, but it will in general have duplicates. To weed out duplicates, for each element on the list, we make a pass through the list and throw out all of its duplicates. We’ll need $d^2$ passes through the list; this gives us an $O(d^4)$ algorithm. (Note that it is possible that $O(d^3)$ is better than $O(n + e^2)$.
14. (a) Suppose we start with a graph $G$ with $n$ vertices. Since a minimum weight spanning tree is a tree, it must have exactly $n - 1$ edges (each vertex but the root has exactly one edge going to its parent). That means if we add a constant $c$ to the weight of each edge, then the total weight of a minimum spanning tree changes by $c(n - 1)$. That’s true for every minimum spanning tree of $G$, so the minimum spanning tree before you added $c$ is the same as the minimum spanning tree after you add $c$. (That would be true even if we allowed negative edge weights and we allowed $c$ to be negative.)

(b) We didn’t really talk about shortest-path trees in class (it’s on pp. 523–526 of the text, and you’re not responsible for them for the final) but it is just what you would expect: if the root is $s$, then for every vertex $v$, the path from $s$ to $v$ in the shortest-path tree is a shortest path from $s$ to $v$. Since shortest paths are closed under prefixes (that is, if $u$ is on a shortest path from $s$ to $v$, then the prefix of the path is a shortest path from $s$ to $u$), we can construct shortest-path trees using, for example, Dijkstra’s algorithm. The shortest path tree would definitely change if we added constants to every edge. For a simple example, consider the graph below:

```
1
a --- b --- c
|___________|
3
```

Clearly the shortest path tree starting at $a$, is just

```
1
a --- b --- c
```

In particular, the shortest path from $a$ to $c$ goes through $b$. (This is also a minimum spanning tree.)

Now suppose we add 2 to the weight of every vertex, to get the graph

```
3
a --- b --- c
|___________|
5
```

Now the shortest-path tree is

```
3
a --- b
5 |
|
c
```

3
20. **In case it wasn’t obvious, the question is asking you to output the MST that you get from Kruskal’s algorithm (or Prim’s algorithm).** With Kruskal’s algorithm, you add the edges in the following order: 
\((h, i), (h, g), (g, f), (i, c), (a, b), (e, f), (c, d), (a, h)\). I have added the edges in lexicographic order here. Without that assumption, it would be OK to switch \((i, c)\) and \((g, f)\), to switch \((e, f)\) and \((a, b)\), and to add \((b, c)\) instead of \((a, h)\). With Prim’s algorithm, you add the edges in the following order: \((e, f), (f, g), (f, c), (g, h), (h, i), (i, c), (c, d), (a, h), (a, b)\). If you don’t care about lexicographic order, then you can add \((b, c)\) instead of \((a, h)\); the next edge is still \((a, b)\).

22. **One way to implement this ADT is to conceptually think of each element as being simultaneously in a queue and in a priority queue.** We represent the queue as a doubly-linked list, with the head being the first element inserted and the tail being the last element inserted. We represent the priority queue as a heap, with the maximum element at the root. Thus, each element has a pointer to its successor and predecessor in the queue (unless it’s the head or tail, in which case the pointers point to **NIL**) and pointers to its left and right children and parent in the heap. To Insert an element \(x\), we both insert it at the tail of the queue and in the appropriate position in the priority queue. Thus, Insert takes time \(O(\lg n)\). To do GetNext, we return the head of the queue, and delete this element from both the queue and the heap. Since deletion from the heap takes time \(O(\lg n)\) (it’s similar to **HEAP-INSERT** — see Exercise 7.5-5), GetNext takes time \(O(\lg n)\). Finally, for GetMax, we extract the top element from the heap (which takes time \(O(\lg n)\)), and adjust the pointers of its successor and predecessor in the queue to delete it from the queue (which takes constant time). Thus, each operation takes time \(O(\lg n)\).

Another way of doing it is to think of each element being conceptually in two red-black trees, one ordered by key value (as usual) and the other ordered by time of insertion in the queue. (Think of insertion time as just another key value.) Thus, for each item, we have the fields color\(_1\), key\(_1\), left\(_1\), right\(_1\), and \(p_1\) and color\(_2\), key\(_2\), left\(_2\), right\(_2\), and \(p_2\). key\(_1\)[\(x\)] is \(x\)’s key, while key\(_2\)[\(x\)] is the order in which \(x\) was inserted. We also have two counters Inserted and Deleted (initialized to 0) that keep track of how many elements have been inserted and deleted so far. To do Insert[\(x\)], we do standard RB-tree insert with respect to all the fields with subscript 1, then we set key\(_2\)[\(x\)] to Inserted, set Inserted to Inserted+1, and do standard RB-tree Insert with respect to all the fields with subscript 2. To do GetNext, we search the second tree (the one with subscript 2) for the element with key Deleted+1, delete that element from both trees, and set Deleted to Deleted+1. Finally, to do GetMax, we search the first tree with the element with the maximum key, delete it from both trees, and set Deleted to Deleted+1. Again, all operations take time \(O(\lg n)\).