CS 410, Spring 1999: Prelim Solutions

1. (a) Note that $(n+1)^2 = n^2 + 2n + 1 \leq 4n^2$ for all $n \geq 1$. Thus $(n+1)^2 = O(n^2)$, taking $c = 4$ and $n_0 = 1$. Alternatively, you could observe that $2n^2 - (n^2 + 2n + 1) = n^2 - 2n - 1$. It is easy to see that if $n \geq 3$, $n^2 - 2n - 1 > 0$, so if $n \geq 3$, $2n^2 \geq n^2 + 2n + 1$. Thus, we can also take $c = 2$ and $n_0 = 3$ in the definition of $O(n^2)$. Other combinations are possible too.

   (b) Applying the master theorem to the recurrence $T(n) = T(n/4) + n^2$, we have $a = 1$, $b = 4$, and $f(n) = n^2$. Since $\log_4(1) = 0$, clearly for any $\epsilon \leq 2$ we have $f(n) = \Omega(n^{\log_4(1)+\epsilon})$. Thus, we are in case 3 of the master theorem. We also have to check that there is a $c < 1$ such that $af(n/b) \leq cf(n)$. Now $af(n/b) = (n/4)^2 = (1/16)n^2$. Thus, we can take any c between 1/16 and 1. Using the master theorem, we get that $T(n) = \Theta(n^2)$. For full credit, you must explicitly show the value of a and b when you apply Master theorem. Also, you must show the existence of an appropriate $c \leq 1$. Furthermore notice that T(n) is $\Theta(n^2)$, not $\Omega(n^2)$, not $O(n^2)$, and definitely n^2.

   [Grading: 1 point for getting a, b, and $f(n)$; 1 point for getting c; 2 points for putting it all together to get the right answer. Note this was worth 4 points, making question 1 worth 7, not 6.]

2. (a) QuickSort runs faster in practice that MergeSort, because it does sorting in place. MergeSort has worst-case running time of $O(n \lg n)$; for QuickSort, the worst case is $O(n^2)$. [It was also acceptable to observe that MergeSort is stable, while QuickSort is not.]

   (b) QuickSort has an average-case time of $O(n \lg n)$; the average-case time for InsertionSort in $O(n^2)$. On the other hand, InsertionSort does not have the overhead of recursive calls and is also in place, so it tends to run faster on small inputs. [It was also acceptable to say that InsertionSort is much faster than QuickSort on almost-sorted inputs.]

   (c) In hashing with chaining we can use smaller tables, since the load factor can be greater than one. [It was also acceptable to say that in hashing with chaining there is no bound on the size of the inputs. (This is true if there is no bound on the size of the linked lists.) Also acceptable was saying that if we also delete keys, then hashing with chaining typically runs better.] In hashing with open addressing, we save the overhead of linked lists and pointers.

(d) Hashing does the dictionary operations (search, delete, insert) in expected $O(1)$; for red-black trees it is $O(\lg n)$. On the other hand, min and max run in time $O(\lg n)$ in red-black trees; they are not really supported by hashing (in fact, they can be implemented to run in time $O(n)$). [It was also acceptable to say that the worst-case running time of the dictionary operations in hashing is $O(n)$, although that's not a compelling reason to use red-black trees.]

(e) Linear probing is easier to compute and implement than quadratic probing. On the other hand, quadratic probing has better performance than linear probing if the load factor is significantly greater than $1/2$, because with linear probing we get primary clustering.

[Grading: You got one point for giving an advantage of the first method over the second, and one point for giving an advantage of the second over the first. For full credit, you needed to show knowledge of both data structures/techniques, and make a concrete comparison on some basis.]

3. (a) If x and y are the min and max of S, then their difference is at least as great as the difference between any two other elements. We can compute min and max in worst-case time $O(n)$.

   [Grading:] A common mistake here was taking w and z to be fixed numbers whereas they are not. The inequality has to hold for all w, z in S. Up to 2 points were deducted for misunderstanding the problem thus. 1 point was given to students who understood the question completely but made an (incorrect) attempt. Also note that you don't need to sort the array to find its min and max. Sorting makes the algorithm $O(n \lg n)$ as the numbers need not be integers (and therefore you cannot use any of the linear sorting algorithms) - up to 2 points were deducted if your algorithm includes sorting.

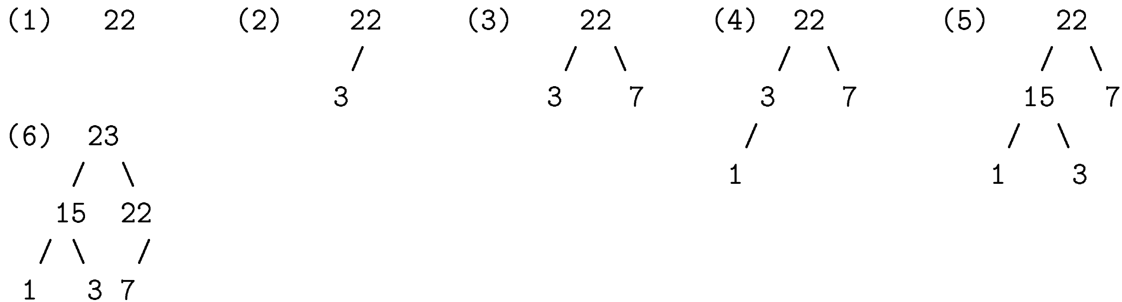
   (b) Sort the elements of S in time $O(n \lg n)$. Then compute the difference of each consecutive pair of elements (that is, the difference between the elements of rank i and rank $i + 1$, for $i = 1, \ldots, n - 1$) and pick a pair of elements whose difference is smallest. Once the list is sorted, this can clearly be done in time $O(n)$. This pair of elements clearly has a difference at least as small as any other pair.

   [Grading:] 3 points were given for using some form of sorting, 2 points for finding the minimum of the differences between consecutive elements. Again the common mistake of taking w and z as given constants was made here. The inequality has to hold for all w, z in S. Up to 2 points were deducted for misunderstanding the problem thus.

   (c) First hash each of the elements of S into a hash table. Since inserting takes expected time $O(1)$, inserting all the elements of S takes expected time $O(n)$. Then for each element $x \in S$, search to see if $3 - x$ in the hash table. If it is, return $(x, 3 - x)$; otherwise, return "no such x and y". Each search takes

expected time $O(1)$, so all n searches take expected time $O(n)$. The whole procedure thus takes expected time $O(n)$.

4. Here is the sequence of heaps:



[Grading: 2 point for getting the first 4 insertions right; 1 more for adding 15 right to whatever heap you had at that point; 2 more for 23 right to the previous heap.]

5. We can represent each integer m in the range 1 to n^2 in a unique way as an ordered pair (i, j), $0 \leq i, j \leq n-1$, where $m - 1 = in + j$. This is basically writing m in base n. (Note we can't write $m = in + j$, since n^2 can't be written this way.) Computing this representation takes constant time for each integer, so doing this for all n integers takes time $O(n)$. We then apply radix sort to the pairs—that's another $O(n)$. Finally, we convert a pair (i, j) back to an integer m in the range 1 to n^2. The whole procedure takes time $O(n)$.

   [Grading: Two points for realizing you need to convert m to a pair (i, j). One point off for not subtracting 1 from m. Three points for using radix sort. (Counting sort won't work — it would take time $O(n^2)$.) One more point for converting back to an integer between 1 and n^2.]

6. Most people had serious problems with this one. There was some confusion about the difference between an item x in the table and its key. To delete an element x from the hash table, first you have to hash $key[x]$ to find where x would have gone. If $key[x] = k$, then you have to probe the whole list of elements that are in a slot of the form $h(k, i)$ until you find x or hit NIL. If you find x, delete it. However, you're not done yet!! (You only got 2/6 if you stopped at this point.) The problem is that there is now a hole in the probe sequence. If there's another element x' that had a collision with x (and thus got put later in the table), you won't find it next time you look for it if you just replace x by NIL. Thus, you have to move elements back to fill in the hole. Thus, your code should have the form: HASH-DELETE(T, x)

   1 $i \leftarrow 0$
   2 $k \leftarrow key[x]$
   3 **while** $T[h(k,i)] \neq$ NIL and $T(h(k,i) \neq x$

```
4      do i ← i + 1
5  if T(h(k, i) = NIL
6      then return "x is not in table"
7      else FIX-HOLE(k, i)
```

FIX-HOLE(k, i) is suppose to fill the hole at slot $h(k, i)$, by putting NIL in the slot if there's nothing to pull back, or by pulling later elements in the probe sequence back. When I originally wrote the problem, I thought that all you need to do is to move the later elements in the sequence back one–that is, move a non-NIL element in slot $h(k, i+1)$ (if there is one) back to slot $h(k, i)$, a non-NIL element in slot $h(k, i+2)$ to slot $h(k, i+1)$, etc. That's not quite right for two reasons: (1) some of these later elements might be in their preferred slot (as opposed to having gotten where they are due to a collision with x), in which case they shouldn't be moved, and (2) you may have to look in slots other than ones of the form $h(k, j)$. To see what's going on, suppose we are doing linear probing (so $h(k, i) = h'(k) + i$), the hash table has size 10, with slots numbered 0, ..., 9, and $h'(k) = k \bmod 10$. Suppose we put 6, 7, 16, and 27 in the table, in that order; 6 goes in slot 6, 7 goes in slot 7, 16 goes in slot 8 (since slots 6 and 7 are already filled up), and 27 goes in slot 9. Now if we want to delete 6, we don't want to move back 7 (it's in its preferred slot, slot 7), but we do want to move 16 back to slot 6 and 27 back to slot 8. If we don't move back 16 and 27, we won't find them when we search. Note that $h'(27) \neq h'(6)$, but we still have to move it back. In the case of linear probing, it's easy to figure out what to move back. You simply move back everything of the form $h(k, i) + j$ that isn't where it's supposed to be, up to the point when you hit a NIL. Here's the code for FIX-HOLE with linear probing:

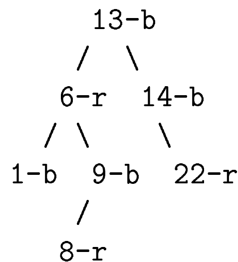
FIX-HOLE(k, i)

```
1  j ← 0
2  last ← h(k, i)
2  while T[h(k, i) + j + 1] ≠ NIL
3      do if h(k, i) + j + 1 ≠ h(key[T[(h(k, i) + j + 1], 0)
          [if the element in slot h(k, i) + j + 1 is not in its preferred slot]
4            then T[last] ← T[h(k, i) + j + 1]
5                 last ← h(k, i) + j + 1
6         j ← j + 1
7  T[last] ← NIL
```

Of course, you couldn't assume that we were doing linear probing. The code for general $h(k, i)$ is much more complicated; the only way that I can think of involves either (a) adding extra data structures (e.g., using a separate table to keep track of what items had collisions), (b) searching the whole table, or (c) using a special DELETED marker (which you weren't allowed to use in this question). Because the

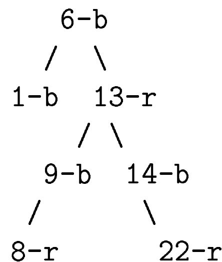
answer isn't as simple as I thought, we accepted almost anything here, as long as there was some attempt to fill in the hole (even if it wasn't quite done correctly). [I regraded a number of prelims on Saturday to make sure we were consistent, so you might find your grade went up since you looked on Friday.]

7. For part (a), the tree has to be labeled so as to ensure that the nodes in the left subtree of a node x are left than x and the nodes in the right subtree are greater than x. It's best to order the entries first — 1,6,8,9,13,14,22 — to see how to do this. There's only one acceptable way of doing it, described below.



For part (b), when it comes to labeling the nodes with r and b, don't forget you have to imagine there are virtual leaves labeled NIL, and we need to ensure that paths to the NILs also have the same number of black nodes. It turns out that there's a unique way of doing it. The root must be black. We can't make 14 red, for otherwise the path from 13 to the virtual NIL that is the left child of 14 would have only one black node. That would mean that all nodes below 13 would have to be red, which would violate the property that there can't be two consecutive red node on a path. Thus, node 14 has to be black. This means the path to the NIL on the left of 14 has two blacks. So all path from the root have to have exactly two blacks. This forces 22 to be red. If 6 is black, then one of 8 or 9 would have to be black, and we would have three blacks on the path from 13 to 8; that won't work. So we have to make 6 red. That forces 1 and 9 to be black and 8 to be red, to get the count right. The black height of the root (and hence of the tree) is 2 (notice we don't count the root itself, but we do count the virtual leaves).

For part (c), we get the tree below:



Again, the labeling of red and black is unique; no other labeling can ensure the same number of black nodes on each path.

[Grading: 4 points for each part. In part (a), you can get 2 if your labeling is off in one place (e.g., the 8 and 9 are switched, but otherwise correct). In part (b), you can get 2 if your red-black labeling is correct ignoring the NILs. (There are then several ways of doing it.) You also get 1 point for stating the red-black height correctly. In part (c), you get 2 points for the rotation and 2 more for the labeling.]