Selected Answers to the Final Exam: CMSC 420, Fall 2002

Problem A. In order to implement an up-tree, recall that in addition to the tree itself, we also need a data structure that gives us, for each key K, a pointer to where K is in the tree. Below are three possibilities for what this data structure might be. For each possibility, describe the conditions under which it would work better than the other two possibilities.

1. **[5 points]** An array.

   Answer. Let k be the size of the key space, i.e., the range of values from the smallest to largest possible values of $(int)(K)$. If we have enough room to allocate an array of size k without running into memory problems, then we would prefer to use an array rather than a hash table or an AVL tree, because an array is the easiest to implement and will give the fastest running time.

2. **[5 points]** A hash table.

   Answer. Suppose the key space is too large for an array to be feasible, but suppose we know that the total number of keys in the tree will never exceed m, and we have enough room to allocate a hash table that is big enough to accomodate m keys efficiently. Then we would prefer a hash table because it will be easier to implement than an AVL tree and will probably run faster.

3. **[5 points]** An AVL tree.

   Answer. We would prefer an AVL if neither of the other alternatives is feasible. This would happen, for example, if we cannot afford to allocate a large array or hash table at the time that we create the up-tree.

Comments: Here are some of the most common mistakes people made: (1) I wasn't asking what's good and bad about each data structure—I was asking when we would want to use it rather than the others. (2) The data structure isn't a replacement for the up-tree itself—instead, it's what tells us where to start in the up-tree. (3) In the book's definition of up-trees, there are never any deletions, and no two nodes ever have the same key.

Problem B. True or false:

1. **[2 points]** If there is a constant k such that $\lim_{n\to\infty}(f(n) - g(n)) = k$, then $f(n) = \theta(g(n))$.

   Answer. False, e.g., consider the case where $g(n) = 1/n$.

2. **[2 points]** If $f(n) = \theta(g(n))$, then there is a constant k such that $\lim_{n\to\infty}(f(n) - g(n)) = k$.

   Answer. False, e.g., consider the case where $f(n) = n$ and $g(n) = 2n$.

3. **[2 points]** If there is a constant k such that $\lim_{n\to\infty}(f(n)/g(n)) = k$, then $f(n) = \theta(g(n))$.

   Answer. False, e.g., consider the case where $f(n) = n$ and $g(n) = n^2$. It would have been true if I had required $k > 0$.

4. **[2 points]** If $f(n) = \theta(g(n))$, then there is a constant k such that $\lim_{n\to\infty}(f(n)/g(n)) = k$.

   Answer. False, e.g., consider the case where $f(n) = 1$ when n is even, $f(n) = 2$ when n is odd, and $g(n) = 3 - f(n)$.

Problem C. John wants to write a program that uses an array $A[1..n]$, and is considering two alternative ways to implement it:

1. An ordinary array. Suppose that in this case, initializing $A[1..n]$ takes time $3cn$ (where c is a constant), retrieving a value $A[i]$ takes time c, and assigning a value to $A[i]$ also takes time c.

2. An array that uses the constant-time array-initialization technique described in the book. Suppose that for such an array, initializing $A[1..n]$ takes time $3c$, retrieving $A[i]$ takes time $3c$, and assigning a value to $B[i]$ takes time $11c$.

[10 points] Suppose that John's program will do q number of array initializations, r number of retrievals, and r number of assignments. How big does q need to be (as a function of n and r) for Alternative 2 to be more efficient than Alternative 1? (Give an exact answer, not a big-O or Θ or Ω value. Your answer should be in simplest form.)

Answer. The amount of time taken by Alternative 1 is $t_1 = q(3cn) + rc + rc = c(3qn + 2r)$. The amount of time taken by Alternative 2 is $t_2 = q(3c) + r(3c) + r(11c) = c(3q + 14r)$. For Alternative 2 to be better, we need $t_1 > t_2$, which happens if$q > 4r/(n-1)$.

Problem D. Suppose we run the Lempel-Ziv algorithm on a string consisting n occurrences of the character "a". Suppose that n is a large number, but that the dictionary is big enough that the algorithm never runs out of room.

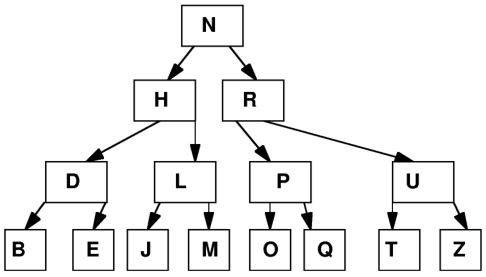
1. **[5 points]** If m is the size of the output, then how big is m as a function of n? (Give a Θ value.)

   Answer. Let k = the size of the longest entry in the dictionary. Then n is approximately $k(k+1)/2$ and m is approximately $2k$, so $m = \Theta(\sqrt{n}.)$

2. **[10 points]** By the time the algorithm ends, its dictionary will include a code number c_1 that represents "a", a code number c_2 that represents "aa", a code number c_3 that represents "aaa", and so forth. How many times will each code number appear in the output?

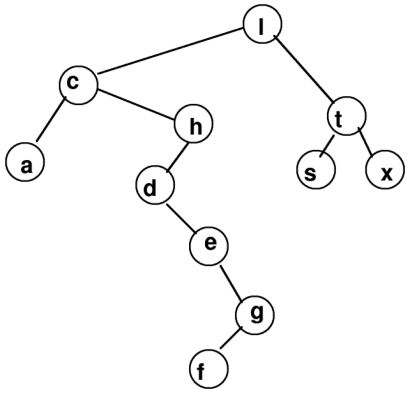
   Answer. Two times for each c_i, with a few exceptions: the last code in the dictionary may appear fewer than two times, and at the very end, one of the codes may appear a third time.

Problem E. **[10 points]** Show all of the steps involved in deleting the node N from the following 2-3 tree:



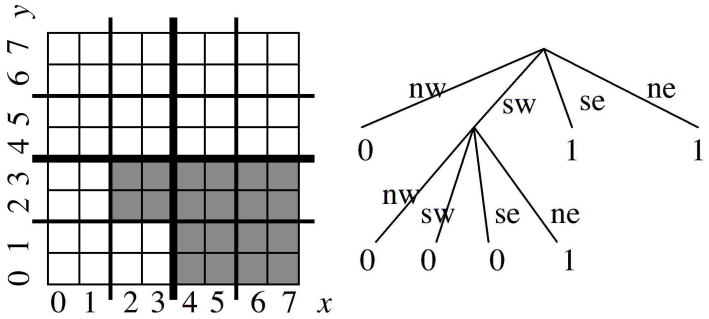
Answer. (Omitted.)

Problem F. [**15 points**] Write down the sequence of rotations that will be done by the procedure lookup(d, T) if T is the splay tree shown below. Draw the final splay tree produced by the procedure.



Answer. (Omitted.)

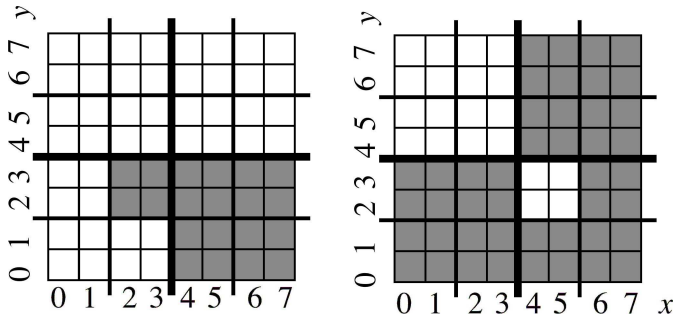
Problem H. If I is a black-and-white video image, then one way to represent it is to use a point quadtree in which each point is one of the black pixels. Another way is to use a region quadtree in which each leaf node is labeled with either 0 (to represent a completely white region) or 1 (to represent a completely black region). As an example of the latter, here is a black-and-white video image I_1 that measures 8×8 pixels (the numbers along the bottom and the left of the image I_1 give the x and y coordinates of each pixel), and a region quadtree that represents I_1.



1. [**15 points**] If an image measures $2^n \times 2^n$ pixels, then what is the maximum possible number of nodes in the region quadtree for I (give an exact value). For the case where $n = 2$, draw an image that will give this maximum number (you don't need to draw the tree).

   Answer. (Omitted.)

2. **[5 points]** If I and J are two images that both are $n \times n$ pixels in size, then their *intersection* $I \cap J$ is an $n \times n$-pixel image that is black at every pixel where both I and J are black, and white everywhere else. Draw the region quadtree that represents the intersection of the two images shown below.



Answer. (Omitted.)

3. **[10 points]** Give an algorithm that takes as input two region quadtrees that represent two images I and J, and returns a region quadtree that represents $I \cap J$.

Answer. (Omitted.)

Problem J. **[10 points]** Show the sequence of heaps produced by the following sequence of insertions into an initially empty heap: 11, 16, 23, 5, 12, 24, 20, 15.

Answer. .

Problem K. **[5 points]** What can we conclude from the 2/3 rule in the case of collecting by copying?

Answer. For this question, I accepted any of the following answers: (1) in steady state, no more than 1/3 of the total memory (i.e., 2/3 of the 1/2 that is currently active) will be in use; (2) for the same reason, requests for more than 1/6 of the total memory are likely to be unsatisfied; (3) we can't conclude anything because collecting by copying doesn't satisfy the assumptions required for the 2/3 rule.

Problem L. **[5 points]** In the boundary tag method for memory management, we store the size of a block at the end of each block, duplicating the information stored at the beginning of the block. Is there any practical advantage to storing the size of the block here, rather than the block's starting address?

Answer. I guess I should have worded this more clearly. My intent was to ask "Is there any practical advantage to storing the size of the block here, rather than storing the block's starting address here?" However, it appears that most of you thought I was asking, "Is there any practical advantage to storing the size of the block here, rather than just storing it at the block's starting address?" Thus, I gave credit for any reasonable answer to either question.

Problem M. **[5 points]** Professor Prune says, "Sorting doesn't need to take time $O(n \log n)$. It can always be done in time $O(n)$. Here is the algorithm. First, scan the input to find the smallest number i and the largest number j. Next, allocate an array of size $j - i + 1$, and use this array to do a bucket sort." What is wrong with Professor Prune's argument?

Answer. There are a bunch of problems. The argument requires that we can either scan the input twice or that n is small enough for the entire input to fit in memory; that the input will always consist of integers (or keys that can easily be mapped into integers); that the array $A[i..j]$ will always be small enough to fit in memory; and that $j - i = O(n)$.

Problem N. **[5 points]** The topological-sorting procedure in the book has worst-case running time time $O(n^2)$, where n is the number of nodes. However, when I discussed topological sorting in class, I said I thought it was possible to have a topological sorting procedure whose worst-case running time is $O(n)$ instead. Was I correct or incorrect? Explain why.

Answer. I was wrong. For a topological sorting algorithm to work correctly, it must examine every node and every edge, for a total time of $\Omega(n + m)$ where m is the number of edges. In the best case, $m = O(1)$, but in the worst case $m = \Theta(n^2)$.