CS 466/666: Assignment 1 Sample Solutions

Fall 2003

Prof. Ian Munro

Problem 6-3 [CLRS]

1. [2 marks] A Young tableau can be thought of as a modified heap, in which each element has *two* parent nodes, and two children nodes. For example, in the tableau below, 4 and 3 are parents of 8, while 9 and 14 are children of 5. The heap property (a parent’s key is smaller than either child’s key) is maintained as in the standard heap. Solutions other than the one given below are possible.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 12 |  |
| 3 | 8 |  |  |
| 5 | 9 |  |  |
| 14 | 16 |  |  |

1. [2 marks] It follows from the heap property that the element that is leftmost and topmost is the minimum value in the tableau. That is, any element to the right of this element, or below it, is by definition greater than it. Hence, if Y[1,1] = , it is a nonexistent element, that is, no elements exist to the right of it or below it. The tableau is thus empty.

If Y[m, n] < , and if one of its parent slots is *not* , then Y[m, n] would have to be exchanged with that parent to maintain the heap invariant, and the comparison with its (new) parents would have to be repeated, until the heap invariant is satisfied. Hence, if the tableau satisfies the heap invariant with Y[m,n] < , it follows that all elements to the left and above Y[m, n] are less than it, that is, there are no empty spaces in the tableau.

1. [6 marks: 4 for the algorithm, 2 for the timebound] EXTRACT-MIN in a tableau is similar to the operation in a standard heap. The minimum value is retrieved from the cell Y[1,1], which is then set to . The value of  is “percolated” down the tableau, so that at each step the *lesser* of the children replaces the value. This is repeated until no elements that are less than  are to the right or below it. The pseudocode is given below:

ExtractMin(Y)

minValue = Y[1, 1]

Y[1, 1] = 

**percolateDown** (Y, 1, 1)

return minValue

percolateDown (Y, x, y)

if Y[x, y+1] = Y[x+1, y] = 

return

if Y[x, y+1] < Y[x+1, y] // compare two children

Y[x, y] = Y[x, y+1]

Y[x, y+1] = 

return **percolateDown** (Y, x, y+1)

else

Y[x, y] = Y[x+1, y]

Y[x+1, y] = 

return **percolateDown** (Y, x+1, y)

1. [4 marks] Insert can be performed in a fashion analogous to the operation in the standard heap, by inserting the new value in an existing -cell, and percolating it “up” until it is greater in value than both its parents. Percolating involves exchanging it with the *greater* of its two parents. The pseudocode is given below:

Insert (Y, value)

if Y[m, n] = 

return “tableau is full”

Y[m, n] = value

**percolateUp** (Y, m, n)

return “insert done”

percolateUp (Y, x, y)

if Y[x, y] greater than both Y[x-1, y] and Y[x, y-1]

return

if Y[x, y-1] > Y[x-1, y] // compare two parents

swap Y[x, y] and Y[x, y-1]

return **percolateUp** (Y, x, y-1)

else

swap Y[x, y] and Y[x-1, y]

return **percolateUp** (Y, x-1, y)

1. [2 marks] An Insert can be done into an n x n tableau in *O(n+n) = O(n)* time. An Extract-Min can also be performed in the same time bound. To sort *n2* numbers, we simply perform *n2* Insert operations, followed by *n2* Extract-Mins. The Inserts populate the tableau, ordered by values in the rows & columns. The Extract-Min operations return the values in ascending order. Since there are *n2* Insert & Extract-Min operations, the total time = , as required.
2. [4 marks] To search the tableau for a given value *k*, we can modify the Insert procedure given above slightly. The idea is to start off as if we are inserting *k*. While percolating it up, however, we check whether either of the parent cells is equal to *k*. If so, we’ve found the entry we’re looking for. If not, we percolate up as before, and repeat. Note that we do not actually modify the tableau as we go along; the insertion and percolateUp are conceptual. If in this process we get to two parent cells which are both lesser in value than *k*, we conclude that *k* does *not* exist in the tableau. It is easily seen that this algorithm runs in the same time-bound as Insert.

Problem 8-3 [CLRS]

1. [4 marks] This problem can be solved by combining bucket-sort with radix-sort. The idea is to go through the array of integers and sort them into buckets based on the number of digits. That is, all 3-digit numbers go into bucket #3, 4-digits numbers into bucket #4 and so on. This step takes time proportional to the total size of the array (assuming the #digits in a number can be found in constant time for each number, ignoring leading zeroes).

Once the buckets have been populated, the numbers within each non-empty bucket are sorted using radix-sort. The sorted order of all numbers is got by reading out the sorted list of bucket 1, followed by that of bucket 2, until the bucket with the largest numbers. Recall from CLRS that radix-sort takes *O(d(n+k))* time, where d = #digits, n = #numbers, k=base of the numbers (which is 10, a constant, in this case). Summing over all buckets, we get runtime =. Note that the #digits in all the numbers in the 3-digit bucket = 3 \* #numbers. That is, the expression inside the summation is the number of digits in bucket i. Summing over all buckets, this gives the total number of digits in the entire array, which is given to be *n*. Thus the runtime is *O(n)*.

Note: Some students have suggested padding all numbers with leading zeroes, to make them the same length as the longest one, and then doing radix sort on them. This works, but is not *O(n)* because of the padding step.

1. [2 marks] Note that a longer string (i.e., one with larger length) might occur alphabetically *before* a shorter one. For this obvious reason, the above algorithm fails on strings. The algorithm to be used in this case is a slight variation of the above idea. Firstly, the order of radix sort on strings is from the left (1st character) to the right. Secondly, instead of performing radix sort on buckets based on length, counting sort is done on “buckets” based on the previously examined character position. Counting sort is performed on each “column” of characters, while maintaining the order of shorter strings previously considered with respect to others. A non-existent character is treated as a “null” that is ordered before the first alphabetic character in the alphabet. A recursive algorithm is presented below in pseudocode. Note the following points about the algorithm:

* Letters in a string are counted from left to right. Hence, in “cat”, ‘c’ is the 1st letter, ‘a’ is the 2nd etc.
* Characters of the alphabet are numbered from 0 to 26: 0 is null, ‘a’ is 1, ‘b’ is 2 etc.

//

// stringSort() sorts the input strings **A**[**s**] to **A**[**e**], by counting sort on

// position **p** in the considered strings

// A: input array of strings

// s: (int) start position of 1st string to sort from

// e: (int) end position of last string to sort to

// p: the digit (or letter) position to consider in countingSort

//

**stringSort (A, s, e, p)**

if (p > max length of strings) return

// the array C[] from CLRS has been ignored for simplicity. This call

// sorts A[s] to A[e] based on the letter in position

countingSort (A, s, e, p);

for let = 0 to 26 // alphabet size

si = position of 1st string in A with letter 'let' in column 'p'

ei = position of last string in A with letter 'let' in column 'p'

**stringSort** (A, si, ei, p+1)

Note that upto 26 recursive calls at a level could be made, but they work on disjoint subsets of the input array. That is, there is no duplication of work. Counting sort examines each letter of each string exactly once. It follows that the runtime is *O(n)*, where *n* = total # characters in all strings.

Problem 9-1.1 [CLRS]

[4 marks] In order to find the second smallest element, we first find the smallest element. We do this by partitioning the elements into pairs, and “propagating up” the smaller of the pair in each case. These values in turn are paired up and the process repeated. In this way, the smallest value of the elements is the one that is the smaller of the last comparison between two remaining elements. This is shown in the figure below, with the smallest element shown in black, as it is propagated up in successive steps. Note that the second smallest element is guaranteed to be compared to the smallest one at some point (in which the former “loses” and is not propagated up). This is always true, as the second smallest element can only “lose” a comparison to the smallest element, and if it won all the comparisons it was part of, it would be at the root of the tree. Hence, once we have found the smallest element, we can retrace the path from the root to its original position, observing the elements with which the smallest was compared. The smallest among these observed elements (shown in gray in the figure) is the second smallest one.

It takes (n-1) comparisons to find the smallest element. Then, to retrace the path from root to the smallest leaf node takes  -1 comparisons, since the tree is well balanced, and there are  nodes on the path from root to leaf. Adding these for the total, we get n+-2 comparisons in the worst case, as required.



Problem 9-1.2 [CLRS]

[4 marks] We use the “tournament” argument from class. We start with *n* candidates for max, *n* for min. We must end up with 1 candidate for each. Hence, 2n-2 “candidatures” must be eliminated.

Call an element that has never been compared a “V”. An element that won but not lost is a “W”. An element that has lost but not won is a “L”. An element that has both won and lost is a “B”.

Our adversarial approach is to say:

* a W always wins versus a V, L or B
* a L always loses versus a V, W or B
* for other comparisons, we don’t care about the outcome.

If we have a V:V comparison, 2 “candidatures” are removed. Otherwise, any comparison results in the loss of at most one candidature.

Therefore any algorithm requires at least *(2n – 2) – #V:V comparisons*. (Let #V:V comparisons be *v*. Then, those comparisons eliminate *2v* candidatures. The remaining *(2n-2-2v)* candidatures can only be eliminated one at a time, since comparisons other than V:V eliminate one candidature each. Hence, we need *(2n-2-2v)* comparisons in addition to the *v* already performed. The total # comparisons is *2n – 2 – 2v + v* = *2n – 2 – v*, as given above.)

Observe that on doing a V:V comparison, neither element remains a V. So no element can be involved in more than 1 V:V compare. So there are at most  V:V compares. Hence, any algorithm requires at least

 compares.

Question 3

1. [4 marks] The idea is to use the algorithm to find the median in deterministic linear time (CLRS, Sec. 9.3) to find the median of the given set of numbers. In linear time, using the median as the “pivot” the set can be partitioned into three groups (=, < and >) of numbers that are equal to, less than and greater than the median respectively. If the size of the “=” group is more than n/2, the median is the mode, and we’re done. If not, we have to operate of the “<” and “>”sets. A recursive call will cover all portions in a depth first manner. We want a breadth first approach. Consider the set of regions which may contain more than one element value. Repeatedly partition the largest such region – giving its median and ties and also (1) those larger and (2) those smaller. The set of smaller values go back into the set of regions as does the set of larger values. Continue this process until the largest group of equal values you have found is at least as large as the largest of these sets that may have several different values. A slick implementation is actually to think of the process as recursive calls but having a queue rather than a stack handling what to call next. Essentially, we proceed until some block of equal elements is larger than any block of elements not known to be equal.
2. [4 marks] The runtime of the algorithm will be *O(n.lg(n/m))* where *m* denotes the frequency of the mode. If we have a *O(n)* median algorithm, then  comparisons suffice to find the median of several groups of values with the total number of elements in all groups together being at most *n*. Hence, from discussions in class, *7n* comparisons suffice. The “3n” median algorithm alluded to in class actually takes *3n+o(n)* comparisons. This o(n) term causes some difficulty which can be overcome. More interesting is the idea of reconstituting the columns of length’s after one median selection to prepare for the next level. With work this can be used to reduce the number of comparisons to *n.lg(n/m) + o(n.lg(n/m)) + O(n).*
3. [2 marks] The method is essentially optimal, since any mode finding algorithm must partition the elements into blocks of at most *m* elements, such that all elements in one block are known to be < (or all <) than those in another. Given this we would complete a sort of all elements by sorting each block. This would take at most *(n/m)(m.log m)* comparisons as the case in which we would know the least about the elements would give us (n/m) blocks of m elements each. But sorting takes atleast  comparisons by any comparison based method (in average case). Therefore, any mode finding algorithm must take atleast  comparisons.