BLG 381E ADVANCED DATA STRUCTURES

MIDTERM SOLUTIONS - NOVEMBER 12, 2010

10:30-12:30 PM

1 (25pt)	2 (25 pt)	3 (25 pt)	4 (25 pt)	Total (100 pt)

Student Signature:_____

Duration: 120 minutes.
Write your name on each sheet.
Write your answers neatly in the space provided for them.
You must show all your work for credit.
Books and notes are closed.
No questions are allowed during the exam.
Good Luck!

Problem 1 [25 points] Sorting

Suppose that an array, A[1…*n*], contains *n* numbers, each of which is -1, 0, or 1.

a) [8 points] Explain in detail how this array could be sorted using counting sort.

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.

We first add 1 to each of the elements in the input array so that the precondition of counting sort is satisfied. After running counting sort, we subtract 1 from each of the elements in the sorted output array.

b) [4 points] What is the worst-case running time for sorting the array using counting sort?

When the upper bound on the range k = O(n), the sort runs in Θ(n) time. The worst-case running time for sorting the array using counting sort is, therefore, Θ(n).

c) [9 points] Explain in detail how this array could be sorted based on a modification of the partitioning (which we covered in quicksort). Define the loop invariant and how your algorithm works in detail.

A solution based on partitioning is as follows. Let A[1..n] be the input array. We define the invariant

- A[1 .. *i*] contains only -1
- A[*i* + 1 .. *j*] contains only 0, and
- A[*h* .. *n*] contains only +1.

Initially, i = 0, j = 0, and h = n + 1. If h = j + 1, then we are done; the array is sorted. In the loop, we examine A[j+1]. If A[j+1] = -1, then we exchange A[j+1] and A[i+1], and we increment both i and j by 1 (as in partition in quicksort). If A[j + 1] = 0, then we increment j by 1. Finally, if A[j + 1] = +1, then we exchange A[j + 1] and A[h - 1], and we decrement h by 1.

c) [4 points] What is the worst-case running time for sorting the array using partitioning?

The worst-case running time for sorting the array using partitioning is $\Theta(n)$.

Problem 2 [25 points] Stacks and Queues

Assume that a queue (Q) is implemented using two stacks, S1 and S2, which are both initially empty. We add to S1 and remove from S2.

a) [15 points] Write the pseudocode for the queue operations ENQUEUE(Q, x) (adding element x to queue Q) and DEQUEUE(Q) (removing from the queue) in terms of the stack operations PUSH(S,x), POP(S), and STACK-EMPTY(S), which are all operations that take O(1) time. (Hint: If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process, they are now back in order.)

```
ENQUEUE(Q,x)
1    if STACK-EMPTY(S2) = FALSE
2        then while STACK-EMPTY(S2) = FALSE      ▷ Move all of S2 to S1
3            do PUSH(S1, POP(S2))                ▷ Remove top from S2 to and put at top of S1
4    PUSH(S1, x)

DEQUEUE(Q)
1    if STACK-EMPTY(S1) = FALSE
2        then while STACK-EMPTY(S1) = FALSE      ▷ Move all of S1 to S2
3            do PUSH(S2, POP(S1))                ▷ Remove top from S1 to and put at top of S2
4    if STACK-EMPTY(S2) = FALSE
5        then POP(S2)                            ▷ Remove top of S2
6    else error "underflow"
```

b) [5 points] Analyze the running time of the operations in Part (a) in the best case.

In the best case, the correct stack is empty, so the operation reduces to a PUSH or a POP.
The best-case cost of an ENQUEUE or a DEQUEUE is therefore $\Theta(1)$.

c) [5 points] Analyze the running time of the operations in Part (a) in the worst case.

The worst case requires moving the contents of one stack into the other before the PUSH or POP. Therefore, the worst-case cost of an ENQUEUE or a DEQUEUE is $\Theta(n)$, for a queue of n elements.

1 (25pt)	2 (25 pt)	3 (25 pt)	4 (25 pt)	Total (100 pt)

Problem 3 [25 points]

3a) [4 points] The worst case time complexity of a comparison sorting algorithm to sort n items is Ω(n lg n) [which means it is at least n lg n]

3b)[6 points] Assuming that items are distributed according to a .....uniform... distribution, average case time complexity of quicksort when sorting n items is
O(n lg n)

3c) [7points] Give a bound as simple and tight as possible.

$10^{20}n^2 + 5n^3 + 10^{50}sin(n)$ **is** $\Theta(n^3)$

3d) [8points]
Explain the structure property of a heap.
A heap is a nearly complete binary tree.
Which means:
Every node has at most two children.
The leaf level nodes are filled from left to right.

Explain the order property of a heap.
Min heap: for every node q, parent(q)<= q
Max heap: for every node q, parent(q) >=q

Why do we need the structure property of a heap?
We need the structure property so that:
The heap can be stored in an array and it is possible to find the children and parent of a node easily in the array.
The nearly complete property also guarantees that the height of the heap is at most log2(n). Since most operations' complexity is dependent on height, it is important to have a bound on the height.

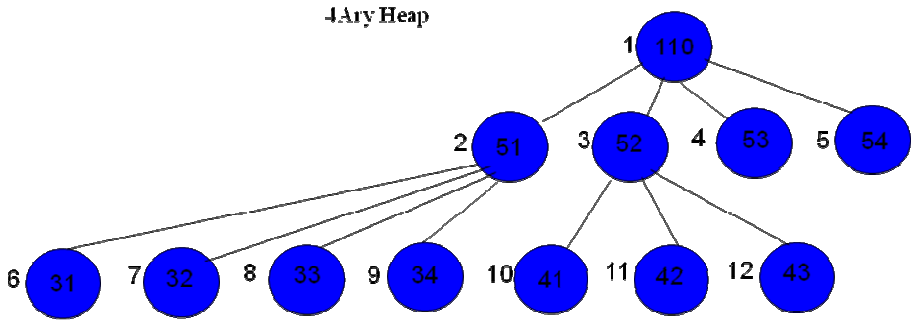
Why do we need the order property of a heap?
[For the max-heap] The order property guarantees that:
The root is the maximum (so that the extract-max can be performed easily)
All the nodes below a node are smaller than the node (so that max-heapify runs at most in heap height = log2n time)

Problem 4 [25 points]

A 4-ary heap is like a binary heap, but with the exception of the root, non-leaf nodes have 4 children instead of 2 children.

4a) [10points] How would you represent a 4-ary heap in an array?

	1	2	3	4	5	6	7	8	9	10	11	12
Array:	110	51	52	53	54	31	32	33	34	41	42	43



Parent[i] = Ceiling((i-1)/4), i>1

Child(i,k) = 4*i+k-3, k=1,2,3,4

4b) [15 points] Give an efficient implementation of EXTRACT-MAX in a 4-ary max-heap. Analyze its running time in terms of n (no of items in the heap).

```
HEAP4-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] - 1
6  MAX-HEAPIFY4(A,1)
7  return max

8 MAX-HEAPIFY4(A, i)
9 largest = i;
10 for k=1 to 4
11       q = CHILD4(i,k)
12       if q <= heap-size[A] and A[q]>A[largest]
13              then  largest ← q
```

```
14 if largest ≠ i
15   then exchange A[i] ←→A[largest]
16        MAX-HEAPIFY4(A, largest)

CHILD4(i,k)
18      return 4*i+k-3
```

HEAP4-EXTRACT-MAX takes constant time plus the time for MAX-HEAPIFY4
MAX-HEAPIFY4 is called at most the height of the heap, because at each step it is either called recursively on an item which is a child of the current node, or it stops.
Since the tree height is log4(n), time complexity of MAX-HEAPIFY4 and HEAP4-EXTRACT-MAX are both O(log4(n))

Extra sheet

Extra sheet