CSCI-UA.0310-004/005 Basic Algorithms March 4, 2015

Solutions to Problem 1 of Homework 4 (12 points)

Name: Jason Yao *Due: Wednesday, February 25*

Recall that we defined a priority queue S together with the following operations (each of which runs in time $\log n$ except the second which runs in time 1).

INSERT(S, x) which inserts x into S.

MAXIMUM(S) which returns the max element in S.

EXTRACT-MAX(S) which returns the max element and removes it from S.

INCREASE-KEY(S, i, x) which increases element i's key to x.

For the purpose of this problem we will call an algorithm "naive" if it only acts on S through these function calls.

Now assume the priority queue is implemented as a max-heap and that you are also given access to the functions (the first four of which run in time 1 and the last in time $\log n$).

PARENT(i) which returns the parent of the i-th element.

LEFT(i) which returns the left child of the i-th element.

RIGHT(i) which returns the right child of the i-th element.

REMOVE(A) which removes the right most leaf of A.

MAX-HEAPIFY(A, i) which lets the i-th element "float" down the heap.

For the purpose of this problem we will call an algorithm "intelligent" if it additionally has access to these 4 functions.

(a) (5 Points) Suppose you would like to find the second max in a heap (i.e. the second largest element of S). One naive approach might be to run the following code:

```
FIND-2NDMAX(S)
    a = EXTRACT-MAX(S)
    b = MAXIMUM(S)
    INSERT(S, a)
    Return b
```

However this runs in time $1 + 2\log n$. Your job is to find an "intelligent" solution which takes time close to 1. Give pseudocode and formally analyze the correctness and runtime of your algorithm.

Solution:

```
Find-2ndMax(S)
  a = LEFT(1)
  b = RIGHT(1)

  RETURN Max(a, b)
END Find-2ndMax
```

Correctness:

Left(1) gets the first value to be compared for the maxSecond value, while Right(1) gets the second value to be compared. As this is a maxHeap, we return the larger value of these two.

Runtime:

$T(n) = \Theta(1)$, as there is only one comparison that is done. □

(b) (5 Points) Now suppose you would like to *extract* the second max. Give a "naive" solution (similar to the example in part [a]) to this algorithm. Argue its correctness and analyze its runtime as precisely as possible.

Solution:

```
Find-2ndMax(S)
  a = EXTRACT-MAX(S)
  b = EXTRACT-MAX(S)
  INSERT(S, a)
  RETURN b
END Find-2ndMax
```

Correctness:

By extracting the first and second values, and then putting the first value back into the heap, we have managed to extract the second value, and only the second value.

Runtime:

$T(n) = 3 \log n$

$T(n) = \Theta(\log n)$, as both extract-Max and Insert methods required $\log n$ time. □

(c) (5 Points) Now give an "intelligent" implementation of EXTRACT-2NDMAX(S) that runs in time close to $\log n$. As usual argue correctness and analyze the runtime. How does this solution compare with the one from part (b)? (**Hint**: Consider using MAX-HEAPIFY.)

Solution:

```
Find−2ndMax(S)
   a, b = max(LEFT(1), RIGHT(1))
   swap(a, S[S.length])
   Remove(S)
   Max–Heapify(S, a)
   RETURN b
END Find−2ndMax
```

Correctness:

By identifying the maximum value of the left and right child of the root, we find the 2nd maximum. In order to extract the node in $\Theta(\log n)$, we swap the maximum value that we found, and swap it with the node at the bottom of the heap. We then remove the heap, and since we already initialized a variable to the maximum value, all we need to do is maxHeapify the priority queue, drifting the bottom-right value down the tree. We then return the variable that was initialized to the maximum value.

Runtime:

$T(n) = \log n + 3$

$T(n) = \Theta(\log n)$

Comparison to solution in (b):

In practice the solution given in (c) may be faster than (b) due to a lower constant, however when dealing with asymptotic runtimes, it should be noted that both algorithms perform in $\Theta(\log n)$ $\square$

CSCI-UA.0310-004/005 Basic Algorithms	March 4, 2015

Solutions to Problem 2 of Homework 4 (16 (+8) points)

Name: Jason Yao *Due: Wednesday, February 25*

Consider the problem of merging k sorted arrays $A_1, \ldots, A_k$ of size n/k each, where $k \geq 2$.

(a) (8 points) Using a min-heap in a clever way, give $O(n \log k)$-time algorithm to solve this problem. Write the pseudocode of your algorithm using procedures Build-Heap, Extract-Min and Insert.

Solution:

```
#define each node in the heap to contain:
  value
  parent
  left
  right
  indexNumber
  arrayNumber

mergekArrays(S)

  B[] = [k]
  for i = 1 to k
    b[i] = A_i[1]
  endfor

  C[] = [n]
  H = BUILD-HEAP(B)
  for j = 1 to n
    minValue = EXTRACT-MIN(H)
    C[i] = minValue
    if A_{minValue.arrayNumber}[minValue.indexNumber + 1] < k
      INSERT(H, A_{minValue.arrayNumber}[minValue.indexNumber + 1])
  endfor

  RETURN C
END mergekArrays
```

□

(b) (8 points) Let the number of arrays $k = 2$. Assume all n numbers are distinct. Using the decision tree method and the fact (which you can assume without proof) that $\binom{n}{n/2} \approx \frac{2^n}{\Theta(\sqrt{n})}$, show that the number of comparisons for any comparison-based 2-way merging is at least

$n - O(\log n)$.
(**Hint**: Start with proving that that the number of possible leaves of the tree is equal to the number of ways to partition an n element array into 2 sorted lists of size $n/2$, and then compute the latter number.)

Solution:

show number of leaves = number of ways to partition n element array into 2 sorted lists of size $n/2$:

$\binom{n}{n/2} \leq 2^h$

$h \geq \log_2 \frac{2^n}{\Theta\sqrt{n}}$

$h \geq n - 1/2 \log_2 n$

$h \geq n - O(\log n)$ □

(c*) **Extra Credit:** Show that any correct comparison-based 2-way merging algorithm *must* compare any two consecutive elements a_1 and a_2 in merged array B, where $a_1 \in A_1$ and $a_2 \in A_2$. Use this fact to contruct an instance of 2-way merging which *requires* at least $n - 1$ comparisons, improving your bound of part (b).

Solution:

Since the two elements (a_1 and a_2) are from different sorted arrays, we must compare them to each other, in order to place them in the correct order. If they were instead in the same array (i.e. A_1), then there would be no need for the comparison, due to it being already sorted.

Instance of 2-way merging which requires at least n - 1 comparisons:

$< a_1, b_1, a_2, b_2, ..., a_n, b_n >$, in which $a_{1...n} \in A_1$, and $b_{1...n} \in A_2$ □

(d**) **Extra Credit:** Show that for general k, any comparison-based k-way merging much take $\Omega(n \log k)$ comparisons, showing that you solution to part (a) is asymptotically optimal.
(**Hint**: You can either try to extend part (b) (easier) or part (c) from $k = 2$ to general k. Beware that calculations might get messy...)

Solution: For general k, any comparison-based k-way merging must take $\Omega(n \log k)$

Instance of worst case k-way merging:

$< a_1, b_1, c_1, ..., j_n, k_n >$

As we have now a number of comparisons between each element from each of the arrays, there are now instead $\log k$ comparisons required for each set of arrays, and for n values, that would result in a runtime of:

$T(n) = n \log k$

□

CSCI-UA.0310-004/005 Basic Algorithms	March 4, 2015

Solutions to Problem 3 of Homework 4 (10 points)

Name: Jason Yao — *Due: Wednesday, February 25*

You receive a sales call from a new start-up called *MYPD* (which stands for "Manage Your Priorities... Differently"). The MYPD agent tells you that they just developed a ground-breaking *comparison-based* priority queue. This queue implements *Insert* in time $\log_2(\sqrt{n})$ and *Extract_max* in time $\sqrt{\log_2 n}$. Explain to the agent that the company can soon be sued by its competitors because either (1) the queue is not comparison-based; or (2) the queue implementation is not correct; or (3) the running time they claim cannot be so good. To put differently, no such comparison-based priority queue can exist.
(**Hint**: You can use the following Sterling's approximation: $n! \approx \left(\frac{n}{e}\right)^n$ (where e is euler's constant))

Solution:

For any comparison-based priority queue, if priorityQueueSort is less than $T(n) = \Theta(n \log n)$, then it is not physically possible, or it is not a comparison based sort, as no comparison-based sort can be faster than $T(n) = \Theta(n \log n)$

```
PriorityQueueSort(A[])
H = BUILDHEAP(A)

n = A.length
for i = 1 to n
  INSERT(H, A[i])
endfor

B[] = [n]
for j = 1 to n
  B[j] = EXTRACTMAX(H)
endfor

RETURN B
```

Runtime analysis:

If $T(n) \leq \Theta(n \log n)$, then we have proved that their comparison-based priority queue is incorrect, or is not a queue.

$T(n) = n + n * (INSERT\_TIME) + n * (EXTRACT\_MAX\_TIME)$

$T(n) = n * (\log_2 \sqrt{n}) + n * (\sqrt{\log_2 n})$

$T(n) = \frac{n}{2} \log_2 n + n * (\sqrt{\log_2 n})$

$T(n) = n * (1 + \frac{1}{2} \log_2 n + \sqrt{\log_2 n})$

As $1 + \frac{1}{2} \log_2 n + \sqrt{\log_2 n} \geq \log n$, this means that their runtime is thus physically impossible, or is not a comparison-based priority queue. $\square$