

HOMEWORK 1 – SAMPLE SOLUTION

1

1.1. **6.5-5.** Show how to implement a first-in first-out queue with a priority queue. Show how to implement a stack with a priority queue.

- (1) Typically, a queue should have following two procedures: enqueue, and dequeue. We could implement it with a minimum priority queue Q_1 . Basically, in the enqueue procedure, for each inserted element, we assign a key to the inserted element by inserting order, and then insert it to Q_1 . In dequeue procedure, we call the extract minimum method of the minimum priority queue, Q_1 to obtain the element with the minimum key. Then we return it. Clearly, it behaviors like a queue. The formal algorithm is described in Algorithm 1, and Algorithm 2. We have a global variable k to count the inserted elements, and its initial value is 0.
- (2) Typically, a stack should have following two procedures: push, and pop. We could implement it with a maximum priority queue Q_2 . Basically, in the push procedure, for each inserted element, we assign a key to the inserted element by inserting order, and then insert it to Q_2 . In pop procedure, we call the extract maximum method of maximum priority queue, Q_2 to obtain the element with the maximum key. Then we return it. Clearly, it behaviors like a stack. Please see the formal algorithm in Algorithm 3, and Algorithm 4. We have a global variable k to count the inserted elements, and its initial value is 0.

Algorithm 1 Enqueue

Input: a minimum priority queue Q_1 , a key k , and an input element e .

Output:

$k \leftarrow k + 1$.
create a new element e'
 $e'.key \leftarrow k$
 $e'.value \leftarrow e$
 $Q_1.enqueue(e')$

Algorithm 2 Dequeue

Input: a minimum priority queue Q_1 , a key k

Output: an element e .

```
if  $k = 0$  then
   $e \leftarrow null$ 
else
   $e' \leftarrow Q_1.dequeue()$ 
   $e \leftarrow e'.value$ 
   $k \leftarrow k - 1$ 
end if
return  $e$ 
```

Algorithm 3 Push

Input: a maximum priority queue Q_2 , a key k , and an input element e .

Output:

```
 $k \leftarrow k + 1.$ 
create a new element  $e'$ 
 $e'.key \leftarrow k$ 
 $e'.value \leftarrow e$ 
 $Q_2.enqueue(e')$ 
```

Algorithm 4 Pop

Input: a minimum priority queue Q_2 , a key k

Output: an element e .

```
if  $k = 0$  then
   $e \leftarrow null$ 
else
   $e' \leftarrow Q_2.dequeue()$ 
   $e \leftarrow e'.value$ 
   $k \leftarrow k - 1$ 
end if
return  $e$ 
```

1.2. **7-3 Stooge Sort. a.** Argue that if $n = \text{length}[A]$, then $\text{STOOGESORT}(A, 1, \text{length}[A])$ correctly sorts the input array $A[1 \dots n]$.

Proof. We could prove the correctness by induction on n .

Base case:

- when $n = 1$, only one element in the sorted range. It returns without doing anything in line 4.
- when $n = 2$, the two elements will be sorted in the first two lines, and then return in line 4.

It is straightforward that the the STOOGESORT correctly sorts the input array in the base cases.

Assume STOOGESORT correctly sorts an array of size less than n . We will prove that it also sorts the array of size equal to n . Without lose of generality, we could assume $n = 3c$ where c is an integer. The second one third elements are no less than the first one third elements after it sorts the first two-third of elements in line 6. The last one third elements will be the biggest ones among the all after it calls line 7, meanwhile the first two-third elements are no bigger than any element in the last one-third. It correctly sorts the array after it calls line 8 to sort the first two-third elements again. \square

b. Given a recurrence for the worst-case running time of STOOGESORT and a tight asymptotic bound on the worst-case running time.

The worst case recurrence for STOOGESORT is $T(n) = 3T(\frac{2n}{3}) + \theta(1)$. By master theorem, we could conclude that $T(n) = \theta(n^{2.7})$

c. Compare the worst case running time of STOOGESORT with that of insertion sort, merge sort, heapsort and quicksort. Do the professors deserve tenure?

The worst case running time for insertion sort and quicksort is $\theta(n^2)$, and the worst case running time for merge sort and heapsort is $\theta(n \ln n)$. It is very obviously the performance of STOOGESORT is not worse than any one of them. So the professors does not deserve tenure.

Algorithm 5 STOOGESORT

```

STOOGESORT(A, i, j)
if  $A[i] > A[j]$  then
    exchange  $A[i] \leftrightarrow A[j]$ 
end if
if  $i \geq j$  then
    return
end if
 $k \leftarrow \lfloor \frac{j-i+1}{3} \rfloor$ 
STOOGESORT(A, i, j-k)
STOOGESORT(A, j+k, j)
STOOGESORT(A, i, j-k)

```

1.3. **Problem 9-1.** Largest numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison based algorithm. Find the algorithm that implements each of the

following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i

a. sort the numbers, and list i largest.

It could be done by using merge sort. The worst case running time is $\theta(n \ln n)$. It takes $\theta(i)$ time to list the i largest ones. The overall time complexity is $\theta(n \ln n + i)$.

b. build a max-priority queue from the n numbers and call EXTRACT-MAX i times.

It takes $\theta(n)$ time to build a the max-priority queue. It takes $\theta(\ln n)$ to extract the maximum one from the queue. So the overall time complexity of this case is $\theta(n + i \ln n)$.

c. use an order-statistic algorithm to find the i -th largest number, partition around that number, and sort the i largest numbers.

It take $\theta(n)$ time to find the i -th largest number by using SELECTION algorithm. Partitioning around this element takes $\theta(n)$ time, too. Sorting the first i -th largest numbers takes $\theta(i \ln i)$ time by using merge sort. So the overall time complexity of this case is $\theta(n + i \ln i)$