论题 2-9 作业

姓名：陈劭源　　　　学号：161240004

1 [TC] Problem 6.1-2

We use mathematical induction to prove this.

For base step, we have $n = 1$, and the height is 0, so the conclusion is obviously correct.

For induction step, assume that for $n = k$, the conclusion is correct. Consider a heap with $n = k+1$ elements. If $k = 2^p - 1$ where p is a positive integer, then a k-element heap is a complete binary tree. Thus, a $(k+1)$-element heap has one more level than a k-element heap, and the height is $\lfloor \lg k \rfloor + 1 = \lfloor \lg(k+1) \rfloor$. If $k \neq 2^p - 1$ for every positive integer p, then a k-element heap is nearly but not exactly a complete binary tree. Thus, a $(k+1)$-element heap has the same level as a k-element heap, and the height is $\lfloor \lg k \rfloor = \lfloor \lg(k+1) \rfloor$. Therefore, the conclusion is correct for $n = k+1$.

By mathematical induction, we conclude that an n-element heap has height $\lfloor \lg n \rfloor$.

2 [TC] Problem 6.1-4

The smallest element must reside in a leaf. If it does not reside in a leaf, then it is smaller than its child(ren), which is contradict to the max-heap property.

3 [TC] Problem 6.1-7

We know that a leaf has no child. For index $i \leq \lfloor n/2 \rfloor$, the index of its left child is $2i$, which is not out of range, i.e. the node indexed by i is not a leaf. However, for index $i > \lfloor n/2 \rfloor$, either the index of its left child $2i$ or the right child $2i+1$, which is out of range, i.e. the node indexed by i is a leaf. Therefore, the leaves are nodes indexed by $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots$, n.

4 [TC] Problem 6.2-2

```
MIN-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] < A[i]
    smallest = l
else smallest = i
if r ≤ A.heap-size and A[r] < A[smallest]
    smallest = r
if smallest ≠ i
    exchange A[i] with A[smallest]
    MIN-HEAPIFY(A, smallest)
```

It has the same running time as MAX-HEAPIFY asymptotically.

5 [TC] Problem 6.2-5

```
MAX-HEAPIFY(A, i)
while TRUE
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        i = largest
    else return
```

6 [TC] Problem 6.2-6

When the value of the node which causes the MAX-HEAPIFY is the smallest in the whole tree, it must be swapped to a leaf. However, for each recursive call, it can only be swapped to its child. Therefore, it must be swapped k times, where $k = \lfloor \lg n \rfloor$ is the height of the tree. Hence, the worst-case running time is $\Omega(\lg n)$.

7 [TC] Problem 6.3-3

We have the following claims:

Claim 1: For any n-element heap $A[1\,..\,n]$, $A[\lfloor n/2 \rfloor + 1\,..\,n]$ exactly contains the elements of height 0, i.e. the leaves of the heap.

Claim 2: For any n-element heap $A[1..n]$, every left subarray of A is still a heap.

By **Claim 1** and **Claim 2**, we obtain the following lemma:

Lemma: The height of element $A[i]$ in heap $A[1..\lfloor n/2 \rfloor]$ is the height of element $A[i]$ in heap $A[1..n]$ minus 1.

Proof: assume that the height of $A[i]$ in heap $A[1..n]$ is h, that means, the length of the longest path from $A[i]$ to a leaf, say $A[l]$, is h. By **Claim 1**, $A[l]$ is in $A[\lfloor n/2 \rfloor + 1..n]$, but not in $A[1..\lfloor n/2 \rfloor]$, and its parent, $A[\lfloor l/2 \rfloor]$, is in $A[1..\lfloor n/2 \rfloor]$. That means, the longest path from $A[i]$ to a leaf of heap $A[1..\lfloor n/2 \rfloor]$ is from $A[i]$ to $A[\lfloor l/2 \rfloor]$, whose length is $h-1$, i.e. the height of $A[i]$ in $A[1..\lfloor n/2 \rfloor]$ is $h-1$. □

Let $f(n,h)$ denote the number of nodes of height h in an n-element heap. By **Lemma** and **Claim 1** we get the following recurrence:

$$f(n,h) = \begin{cases} f(\lfloor n/2 \rfloor, h-1) & h > 0 \\ \lceil n/2 \rceil & h = 0 \end{cases}$$

Solve this recurrence by iteration, we obtain $f(n,h) \le \lceil n/2^{h+1} \rceil$. Therefore, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

8 [TC] Problem 6.4-2

Initialization: Prior to the first iteration, $i = n$, $A[1..i]$ is a max-heap because BUILD-MAX-HEAP has been executed, and of course it contains the i smallest elements of $A[1..n]$. $A[i+1..n]$ is an empty array, so it contains the $n-i$ largest elements of $A[1..n]$ sorted trivially. Therefore, the loop invariant holds before the loop.

Maintenance: By the property of the max-heap, $A[i]$ is the largest element in $A[1..i]$, but it is smaller than every element in $A[i+1,n]$ by the loop invariant. So after swapping, $A[i..n]$ contains the $n-i+1$ largest elements $A[1..n]$, sorted, and after MAX-HEAPIFY executed, $A[1..i]$ is a max-heap containing the $i-1$ smallest elements. So the loop invariant holds after each iteration.

Termination: By the loop invariant, we know that after the loop $A[1..n]$ contains all the elements sorted. Hence the procedure is partially correct.

The **for** loop will be exactly executed for $A.length - 1$ times, thus the procedure can terminate. Therefore, HEAPSORT could sort a given array totally correctly.

9 [TC] Problem 6.4-4

Consider a case, that the input array $A[1..n]$ stores a monotonously decreasing sequence, then it is already a max-heap, and BUILD-MAX-HEAP does not change the array. For each iteration of **for** loop, the smallest element is swapped to the root, and it takes at least $\lfloor \lg(i-1) \rfloor - 1$ swaps to max-heapify the array. Therefore, the total running time is

$$\Omega\left(\sum_{i=2}^{n} \lfloor \lg(i-1) \rfloor - 1\right) = \Omega\left(-n+1+\sum_{i=(n-1)/2}^{n-1} \lfloor \lg i \rfloor\right)$$

$$
\begin{aligned}
&= \Omega\left(-n+1+\lfloor\frac{n-1}{2}\rfloor\lfloor\lg\lceil\frac{n-1}{2}\rceil\rfloor\right) \\
&= \Omega(n\lg n)
\end{aligned}
$$

Therefore, the worst-case running time of HEAPSORT is $\Omega(n\lg n)$.

10 [TC] Problem 6.5-5

Initialization: Before the **while** loop. only $A[i]$ is increased, so it still satisfies the max-heap property, except that "$A[i] \le A[\text{PARENT}(i)]$" may be violated. Therefore the loop invariant holds before the loop.

Maintenance: Before each iteration to execute, we have $A[\text{PARENT}(i)] < A[i]$, and the tree rooted at $A[\text{PARENT}(i)]$ is the largest element except $A[i]$, according to the loop invariant. After exchanging $A[i]$ with $A[\text{PARENT}(i)]$, the tree rooted at $A[\text{PARENT}(i)]$ is a heap. Then, i is assigned $A[\text{PARENT}(i)]$, and after the assignment "$A[\text{PARENT}(i)] < A[i]$" may be violated. Therefore, the loop invariant holds after each iteration.

Termination: After the loop, $i == 1$ or $A[\text{PARENT}(i)] > A[i]$. In the former case, PARENT(i) does not exist, thus the whole array is a heap. In the latter case, "$A[\text{PARENT}(i)] < A[i]$" is in fact not violated, therefore the whole array is also a heap. Hence, the procedure is partially correct.

For each iteration, the depth of $A[i]$ is decremented by 1, so the procedure can terminate. Therefore, the procedure is totally correct.

11 [TC] Problem 6.5-7

Let H be a max-priority queue. $H.insert(k,x)$ inserts the element x associated with key k into the priority queue, $H.extract()$ returns the element with the maximum key in H, and deletes it from H. Let I be a global integer variable with initial value 0.

Implementation of queue:

ENQUEUE(H,x)
$H.insert(I,x)$
$I = I - 1$

DEQUEUE(H)
return $H.extract()$

Implementation of stack:

PUSH(H,x)
$H.insert(I,x)$
$I = I + 1$

POP(H)
return $H.extract()$

12 [TC] Problem 6.5-9

```
MULTI-WAY-MERGE(lists)
Let C[1..lists.count] be a new array
Let H be a min-priority queue of length lists.count
n = 0
for i = 1 to lists.count
    C[i] = lists[i].length
    n = n + C[i]
    if C[i] > 0
        H.insert(lists[i][C[i]], i)
        C[i] = C[i] - 1
Let A[1..n] be a new array
j = 0
while j < n
    j = j + 1
    A[j] = H.min-key
    i = H.extract()
    if C[i] > 0
        H.insert(lists[i][C[i]], i)
        C[i] = C[i] - 1
return A
```

During the whole procedure, the size of the priority queue is at most k, and each element in all the input lists is inserted into and removed from the priority queue once, and the running time of each operation is $O(\lg k)$. Therefore, the algorithm merges k sorted lists into one sorted list in a running time of $O(n \lg k)$.