Introduction To Algorithm

Third Edition

Answer

Xia Ding

December 17, 2015

6.1

6.1-1

Maximum and minimum numbers of elements of elements in a heap of height h is $2^{h+1} - 1$ and 2^h.

6.1-2

Suppose height of a n-element heap has height h.

$$\begin{aligned} 2^h \leq \quad n \ &\leq 2^{h+1} - 1 \\ h \leq \lg n &< h + 1 \\ h &= \lfloor \lg n \rfloor \end{aligned}$$

6.1-3

According to the max-heap-property, A[PARENT(i)] $\geq$ A[i]. A **max-heap** is built recursively from many **max-heaps**, so the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4

In the lowest level of the heap-tree.

6.1-5

If it's sorted non-increased, it does. Otherwise, it doesn't form a min-heap.

6.1-6

No, 6 is less than 7.

6.1-7

Suppose a leaf's index $i < \lfloor n/2 \rfloor$, then it has a left child's indexed by $2i$, less than n. So the array representation n-element heap's leaves are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2

6.2-1

1. swap 3 and 10
2. swap 3 and 9

6.2-2

Algorithm 1 MIN-HEAPIFY(A, i)

```
$i = LEFT(i)$
$r = RIGHT(i)$
if $l \leq A.heap\text{-}size$ and $A[i] < A[l]$ then
    $smallest = i$
else
    $smallest = l$
end if
if $r \leq A.heap\text{-}size$ and $A[r] < A[smallest]$ then
    $smallest = r$
end if
if $smallest \neq i$ then
    exchange $A[i]$ with $A[smallest]$
    MIN-HEAPIFY$(A, smallest)$
end if
```

Their running time is equal.

6.2-3

It will check condition and then do nothing.

6.2-4

It will do nothing

6.2-5

Algorithm 2 MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
largest = i
while l ≤ A.heap-size do
    if A[l] > A[largest] then
        largest = l
    end if
    if r ≤ A.heap-size and A[r] > A[largest] then
        largest = r
    end if
    if largest == i then
        break
    else
        exchange A[i] with A[largest]
        i = largest
        l = LEFT(i)
        r = RIGHT(i)
    end if
end while
```

6.2-6

If an array of size n is non-decreasing, it's height is $\lfloor \lg n \rfloor$, runnning time of MAX-HEAPIFY from top to buttom is $\Omega(\lg n)$

6.3

6.3-1

Iteration

1. $[5, 3, 17, 22, 84, 19, 6, 10, 9]$
2. $[5, 3, 19, 22, 84, 17, 6, 10, 9]$
3. $[5, 84, 19, 22, 3, 17, 6, 10, 9]$
4. $[84, 22, 19, 10, 3, 17, 6, 5, 9]$

6.3-2

If $A[1] > A[2]$ *and* $A[1] > A[3]$, but A[1] isn't the global maximum, then increase from 1 is a mistake.

6.3-3

Using induction can show this.

6.4

6.4-1

It's easy to do by yourself according to the model.

6.4-2

Initialization: Prior to the first iteration of the loop, $i = A.heapsize$. The loop invariant is true.

Maintenance: Suppose the invariant hold before some some iteration, $A[1 \dots i+1]$ has the smallest elements of the array. In this iteration, A[1] is swapped with A[i], and then MAX-HEAPIFY. After that iteration, A[i] is larger then all elements of $A[1 \dots i-1]$, $A[1 \dots i-1]$ has the smallest elements of the array. Hold the loop invariant.

Termination: Now $i = 1$, according to loop invariant, A[1] is the smallest element of A[1. . .n]. By the loop invariant, A[1,2] has the smallest elements of array, so $A[1] < A[2]$. Like this, we can conclude $A[i] < A[i+1]$ for $i = 1 \dots n-1$, so the heapsort is correct.

6.4-3

Increasing order: Total using $O(n \lg n)$. BUILD-MAX-HEAP takes time $O(n)$, then each of the $n-1$ calls to MAX-HEAPIFY takes time $0(\lg n)$.

Decreasing order: Total using $O(n \lg n)$. BUILD-MAX-HEAP takes time $O(n)$, then each of the $n-1$ calls to MAX-HEAPIFY takes time $0(\lg n)$.

6.4-4

I will prove the running time of heapsort is $\Theta(n \lg n)$, which will be used to 6.4-5 too.

Build a max-heap will take time $\Theta(n)$. We will focus on the reminded part of the procedure.

First, suppose there is a complete heap with $n = 2^h - 1$ elements. Think about the $\lfloor n/2 \rfloor = 2^{h-1}$ largest nodes, seen as set S. Because each element of S should be moved up one level by one level to top then be taken away from the heap. So we can see the running time as:

$$\Omega(\text{total steps for all elements of S moving to the top})$$

Now, let's see the distribution of S. We see level x as level of height x. In level 0, there are at most $|S|/2$ nodes of the set S. Because by contradiction, every node's parent is larger than it's child, if there are at least$2^{h-2} + 1$ nodes of set S in level 0. Then in level 1, there are at least $2^{h-3} + 1$ nodes of set S, like this, from $h = 0$ *to* $\lg n$, sum of them is larger than 2^{h-1}, contradicting $|S| = 2^{h-1}$. So, in level 0, there are no more than $|S|/2$ nodes of set S. So there are no more than 2^{h-2} nodes in level 0 and at least 2^{h-2} nodes of S in first $h-2$ levels. Because at most 2^{h-3} nodes in first $h-3$ levels, so at least 2^{h-3} nodes of S in level 1. Move all them up to top take $2^{h-3} * (h-2)$ steps, because $h = \lg n$, to the running time is $\Theta(n \lg n)$.

Now let's see the case the heap isn't completed. The running time of it is larger than running time of first h-1 levels(complete heap), which is $\Theta(n \lg n)$, so the running time is still $\Theta(n \lg n)$.

6.4-5

See 6.4-4

6.5

6.5-1

It's easy to do by yourself.

6.5-2

It's easy to do by yourself.

6.5-3

Algorithm 3 HEAP-MINIMUM(A)

```
1: return A[1]
```

Algorithm 4 HEAP-EXTRACT-MIN(A)

```
1: if A.heap-size < 1 then
2:     error "heap under flow"
3: end if
4: $min = A[1]$
5: A[1] = A[A.heap-size]
6: A.heap-size = A.heap-size − 1
7: MIN-HEAPIFY(A, 1)
8: return min
```

Algorithm 5 HEAP-DECREASE-KEY(A, i, key)

```
1: if $key > A[i]$ then
2:     error "new key is larger than current key"
3: end if
4: $A[i] = key$
5: while $i > 1$ and $A[PARENT(i)] > A[i]$ do
6:     exchange A[i] with A[PARENT(i)]
7:     $i = PARENT(i)$
8: end while
```

Algorithm 6 MAX-HEAP-INSERT(A, key)

```
1: A.heap-size = A.heap-size + 1
2: A[A.heap-size] = $+\infty$
3: HEAP-DECREASE-KEY(A, A.heap-size, $key$)
```

6.5-4

If we don't set it and the desired value of key is no bigger than random initialized value, then we can't insert the node correctly.

6.5-5

Initialization: Before the first iteration, because the heap is max heap, after one key is increased, there is at most one violation to the max-heap property. The loop invariant is true.

Maintenance: Suppose the loop invariant is held before some iteration. In that iteration, the only violation is $A[i] > A[PARENT(i)]$. After that iteration, there are two cases. One is: A[i] is still larger than A[PARENT(i)], only one violation to the max-heap property. The other case is: reach the t termination case. So the loop invariant is true.

Termination: There are two cases for termination. ① $\mathbf{i = 1}$. Then the A[i] is the largest. After increase it's key, it's still the largest, holding the max-heap property. ② $\mathbf{A[PARENT(i)] \geq A[i]}$, holding the max-heap property. So the loop invariant is true.

6.5-6

Algorithm 7 HEAP-INCREASE-KEY(A, i, key)

```
1: if $key < A[i]$ then
2:     error "new key is smaller than current key"
3: end if
4: while $i > 1$ and $A[PARENT(i)] < key$ do
5:     $A[i] = A[PARENT(i)]$
6:     $i = PARENT(i)$
7: end while
8: $A[i] = key$
```

6.5-7

We use Min-heap to implement queue, use Max-heap to implement stack.

Now, we will use a **class**:**node** to encapsulate the value we insert in the priority queue. node.value is what we want to insert, node.key is an integer used to compare node. If $A.key < B.key$, then $A < B$.

We first set a global variable **count = 0**, each time we insert a value, we create a **node** and set **node.value = value, node.key = count, count = count + 1**. So that first-in node has a lower key than others, so each time we use **EXTRACT-MIN**, we get the first-in value. this is the queue.

Stack is similar to queue. Each time we use **EXTRACT-MAX**, we get the last-in value because it's node has a higher key.

6.5-8

Algorithm 8 HEAP-DELETE(A, i)

```
1: $A[i] = -\infty$
2: MAX-HEAPIFY(A, $i$)
3: $A.heap\text{-}size = A.heap\text{-}size - 1$
```

6.5-9

Suppose lists are decreasing. First extract all lists' first elements to build a k-element min heap, take time $\Theta(n)$. Then loop until all all lists become empty. In each iteration, extract a min element from heap to the final list, then extract an element from some list and insert to heap, both procedure take time $O(\lg n)$, because there are n elements and in each iteration extract one element. Because k lists are sorted, so final list is sorted. Total time is $nO(\lg n) = O(n \lg n)$.

Problems

6.1 *Building a heap using insertion*

a Not always. If we input $[1, 2, 3]$. **BUILD-MAX-HEAP** will produce $[3, 2, 1]$, **BUILD-MAX-HEAP'** will produce $[3, 2, 1]$.

b An upper bound of $O(n \lg n)$ time follows from the $n - 1$ calls to **MAX-HEAP-INSERT**, each taking $O(\lg n)$ time. For a lower bound, consider the case in which the input array is in strictly increasing order. Each call to **MAX-HEAP-INSERT** causes **HEAP-INCREASE-KEY** to go all the way up to the root. Since the depth of node i is $\lfloor \lg i \rfloor$, the total time is

$$\begin{aligned}\sum_{i=1}^{n} \Theta(\lg i) &\geq \sum_{i=\lceil n/2 \rceil n/2}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\ &= \sum_{i=\lceil n/2 \rceil n/2}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\ &\geq n/2 * \Theta(\lg n) \\ &= \Omega(n \lg n)\end{aligned}$$

So **BUILD-MAX-HEAP'** requires $\Theta(n \lg n)$ time.

6.2 *Analysis of d-ary heaps*

a. Similar to 2-ary heap.

$PARENT(i) = \lfloor i/d \rfloor$
$CHILD(i, x) = di + x$ {x is i's x-th child, $0 \leq x < d$}

b.

$$\begin{aligned}\frac{d^h - 1}{d - 1} < n &\leq \frac{d^{h+1} - 1}{d - 1} \\ \log_d(nd - n + 1) - 1 \leq h &< \log_d(nd - n + 1) \\ h &= \lceil \log_h(nd - n + 1) \rceil - 1\end{aligned}$$

Because usually d $\ll$ n, so we can say $h = \Theta(\log_d(n))$.

c. Algorithms 9-12.Running time is $O(h) = O(\log_d(n))$.

Algorithm 9 PARENT(d, i)

```
1: return ⌊(i − 2)/d + 1⌋
```

d. Algorithm 13. Have used INCREASE-KEY(A, *d*, *i*, *key*) in **e**. Running time is $O(h) = O(\log_d(n))$.

e. Algorithm 14. Running time is $O(h) = O(\log_d(n))$.

Algorithm 10 CHILD(d, i, x)

```
return $d(i-1)+x$
```

Algorithm 11 EXTRACT-MAX(A, d)

```
$max = A[1]$
$A[1] = A[heap\text{-}size]$
$A.heap\text{-}size = A.heap\text{-}size - 1$
MAX-HEAPIFY(A, 1)
return $max$
```

Algorithm 12 MAX-HEAPIFY(A, d, i)

```
$largest = i$
for all $j$ such that $0 \leq j < d$ do
  $child = CHILD(d, i, j)$
  if $child \leq A.heap\text{-}size$ and $A[child] > A[largest]$ then
    $largest = child$
  end if
end for
if $largest \neq i$ then
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY(A, $largest$)
end if
```

Algorithm 13 INSERT(A, d, key)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = $-\infty$
INCREASE-KEY($A, d,$ A.heap-size, $key$)
```

Algorithm 14 INCREASE-KEY(A, d, i, key)

```
if $key < A[i]$ then
  error "new key is smaller than current key"
end if
$A[i] = key$
while $i > 1$ and $A[PARENT(i) < A[i]]$ do
  exchange $A[i]$ with $A[PARENT(I)]$
  $i = PARENT(i)$
end while
```

Young tableaus

a. It's easy to do by yourself.

b. We suppose each row and column are increasing order from left to right.

If Y[1, 1] = ∞, then for all i such $1 \le i \le n$, we have Y[1, i] = ∞. So the all elements of first row are ∞. Because each column is sorted, so each element beneath the first row are larger than first element in it's column, then all elements beneath the first row is ∞.So such Y is empty.

If Y[m, n] $< \infty$, we prove by contradiction. If there is a element Y[i, j] = ∞ then the subtableau begin from Y[i, j] is empty using before conclusion, so Y[m, n] = ∞, contradict the given condition. So Y is full.

c. Algorithms 15-16. We use (a, b, ...) to represent a tuple, used to simplify assignment and multi-dimension array's index. $(a, b) > (c, d)$ if and only if $a > c$ and $b > d$.

Algorithm 15 MIN-HEAPIFY($Y, (m,\ n), (i,\ j)$)

```
1: $right = (m,\ n+1)$
2: $down = (m+1,\ n)$
3: $smallest = (i,\ j)$
4: **if** $right \le (m,\ n)$ **and** $Y[right] < Y[smallest]$ **then**
5:     $smallest = right$
6: **end if**
7: **if** $down \le (m,\ n)$ **and** $Y[down] < Y[smallest]$ **then**
8:     $smallest = down$
9: **end if**
10: **if** $smallest \ne (i,\ j)$ **then**
11:     exchange Y[smallest] and Y[i, j]
12:     MIN-HEAPIFY($Y, (m,\ n), smallest$)
13: **end if**
```

Algorithm 16 EXTRACT-MIN(Y, (m, n))

```
1: $min = Y[1,\ 1]$
2: $Y[1,\ 1] = Y[m,\ n]$
3: $Y[m,\ n] = \infty$
4: MIN-HEAPIFY($Y, (m,\ n), (1,\ 1)$)
5: **return** $min$
```

d. Algorithm 17.Because each time the new element move left or up, it will take at most m+n steps to reach the top, So the running time is $O(m+n)$.

e. Construct a n*n Young tableau. So insert takes time $O(n)$. Using n^2 times insert, take time $O(n^3)$. After that, use n^2 times extract-min, take time $O(n^3)$. So the total time is $O(n^3)$.

Algorithm 17 INSERT$(Y, (m,\ n), key)$

```
if $Y[m,\ n] \neq \infty$ then
    error "Y is full."
end if
$Y[m,\ n] = key$
$position = (m,\ n)$
$largest = (m,\ n)$
while $position \neq (1,\ 1)$ do
    $left = position - (1,\ 0)$
    $right = position - (0,\ 1)$
    if left >= (1, 1) and Y[left] > Y[largest] then
        $largest = left$
    end if
    if right >= (1, 1) and Y[right] > Y[largest] then
        $largest = right$
    end if
    if $largest == position$ then
        break
    else
        $Y[position] = Y[largest]$
        INSERT$(Y, largest, key)$
    end if
end while
```

f. Taking turn using binary search to Y's row and column, each time make the length of row or column half, so that the size of tableau become half. So running time is

$$O(\lg mn) = O(\lg m + \lg n) = O(m + n)$$