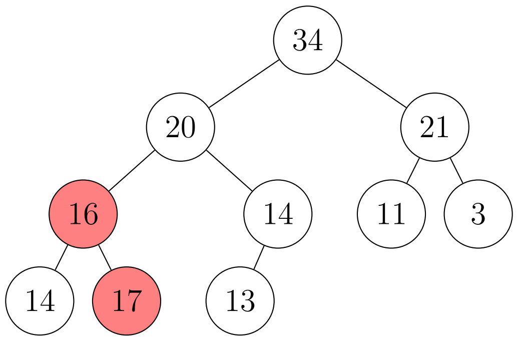
Data Structure and Algorithm Analysis(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

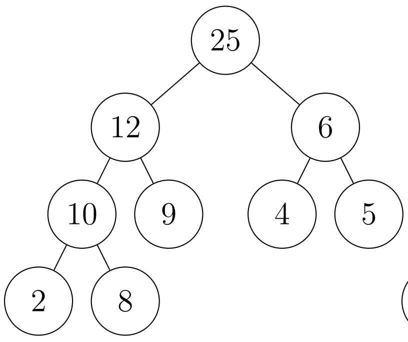
Work Sheet 4

Question 4.1

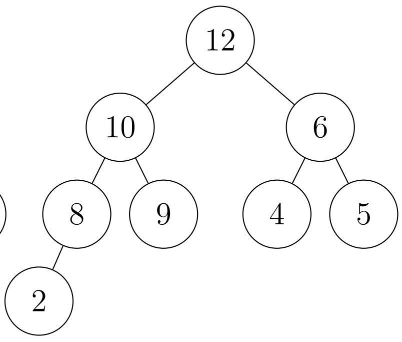


As the graph shows, the array is not a Max-Heap.

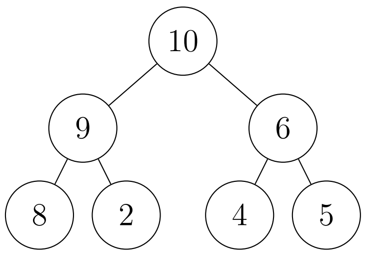
Question 4.2



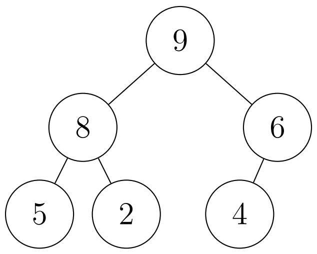
1



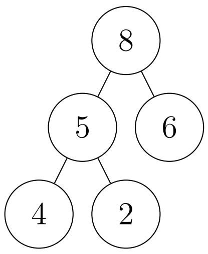
2



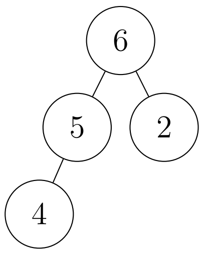
3



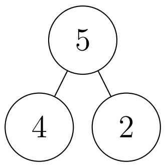
4



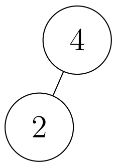
5



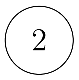
6



7



8



9

Question 4.3

1.

```
MAX-HEAPIFY($A, i$)
1.  while $i \leq A$.heap-size do
2.      $l$ = Left($i$)
3.      $r$ = Right($i$)
4.      if $l \leq A$.heap-size and $A[l] > A[i]$ then
5.          largest = $l$
6.      else
7.          largest = $i$
8.      if $r \leq A$.heap-size and $A[r] > A$[largest] then
9.          largest = $r$
10.     if largest $\neq i$ then
11.         exchange $A[i]$ with $A$[largest]
12.         $i$ = largest
13.     else
14.         break
```

Assume that A.heap-size is updated in HEAPSORT.

2.

Loop invariant: At the start of each iteration of the **while** loop, the array is a Max-Heap except that $A[i]$ might be smaller than its children.

Initialization: Before the first iteration of the loop, the array is a Max-Heap except that one element $A[i]$ might be smaller than its children.

Maintenance: The algorithm finds the largest element A[largest] among $A[i]$, $A[\text{Left}(i)]$ and $A[\text{Right}(i)]$, and exchange $A[i]$ with A[largest]. Then the array is a Max-Heap except that one element A[largest] might be smaller than its children. Then the algorithm sets i = largest and continues the loop.

Termination:

Case 1: The loop terminates when Left(i) is larger than the heap size, which means $A[i]$ is a leaf node. Since the leaf node has no children, $A[i]$ cannot be smaller than its children. Therefore, the array is a Max-Heap.

Case 2: The algorithm breaks the loop if $A[i]$ is larger than its children. Since the array is a Max-Heap everywhere else already, the array is a Max-Heap.

Question 4.4

1.

Assume the right subtree of the root node has the last level empty and the left subtree has the last level full. When the right subtree has height h ($h \geq 0$), the left subtree has height $h+1$. The number of nodes in the right subtree is $2^h - 1$ and the number of nodes in the left subtree is $2^{h+1} - 1$. Therefore, the left subtree contains $\frac{2^{h+1}-1}{2^h-1+2^{h+1}-1+1}$ of the nodes in the tree.

The maximum percentage can be achieved when h approaches infinity.

$$\begin{aligned}&\lim_{h\to\infty}\frac{2^{h+1}-1}{2^h-1+2^{h+1}-1+1}\\=&\lim_{h\to\infty}\frac{2\cdot 2^h-1}{3\cdot 2^h-1}\\=&\lim_{h\to\infty}\frac{2-\frac{1}{2^h}}{3-\frac{1}{2^h}}\\=&\frac{2}{3}\end{aligned}$$

Therefore, the left subtree contains at most $\frac{2}{3}$ of the nodes in the tree.

2.

Using Master Theorem, the MAX-HEAPIFY algorithm has recurrence equation $T(n) \leq T(\frac{2n}{3}) + \Theta(1)$. Since $a = 1$, $b = \frac{3}{2}$, $f(n) = \Theta(1)$, we have $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0$. This is case 2 of Master Theorem, since there exists $k = 0$ that $f(n) = \Theta(1) = \Theta(n^0 \log^0 n)$. Therefore, $T(n) = \Theta(n^0 \log^{0+1} n) = \Theta(\log n)$.

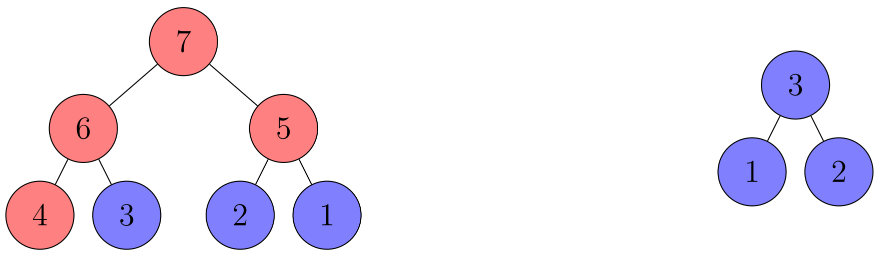
However, this is not the tight bound since we use $\leq$ in the recurrence equation. What we have proved is that $T(n) \leq \Theta(\log n)$. Hence, $T(n) = O(\log n)$.

Question 4.5

Since the runtime of BUILD-MAX-HEAP is in $O(n)$, we don't need to consider it. We will focus on the runtime of MAX-HEAPIFY. To be more specific, we will estimate the runtime of MAX-HEAPIFY on an already sorted array by calculating how many exchange operations and comparison operations it will perform.

Since incomplete binary trees can be turned into complete binary trees by discarding the last level, their runtime will always be bigger than the runtime of corresponding complete binary trees of smaller height. Therefore, we can omit them when calculating the lower bound. Let's assume the heap is a complete binary tree with n nodes. Therefore, the height of the tree is $h = \log(n + 1)$ when the algorithm starts.

In the process of discarding elements until the heap becomes a complete binary tree of height $h-1$, only the last $2^{h-1}-1$ elements remain, since the array is sorted. We color them in red and blue separately.



An example of 7 elements.

Since the number of comparisons is exactly twice the number of exchanges during MAX-HEAPIFY, we only need to count the number of exchanges. To achieve this transformation, the blue nodes will first exchange with the root node and then move downwards. The number of exchanges will be:

$$\begin{aligned}
&2^{h-1} - 1 + \sum_{n=1}^{h-1} (n-1)\, 2^{n-1} \\
=&2^{h-1} - 1 + (h-3)\, 2^{h-1} + 2 \\
=&(h-2)\, 2^{h-1} + 1 \\
=&\frac{1}{2} h 2^h - 2^h + 1
\end{aligned}$$

Since $h = \log(n+1)$, the number of exchanges will be:

$$\begin{aligned}
&\frac{1}{2} h 2^h - 2^h + 1 \\
=&\frac{1}{2}(n+1)\log(n+1) - (n+1) + 1 \\
=&\frac{1}{2}(n+1)\log(n+1) - n
\end{aligned}$$

And $\frac{1}{2}(n+1)\log(n+1) - n$ is in $\Theta(n \log n)$ since:

$$\begin{aligned}
\frac{1}{2}(n+1)\log(n+1) - n &\le \frac{1}{2}(n+1)\log(n+1) \\
&\le (n+1)\log(n+1) \\
&\le 2n\log(2n) \\
&= 2n(\log 2 + \log n) \\
&= 2n\log 2 + 2n\log n \\
&\le 4n\log n \\
&= O(n\log n)
\end{aligned}$$

$$\begin{aligned}
\frac{1}{2}(n+1)\log(n+1) - n &\ge \frac{1}{2} n\log n - n \\
&= \Omega(n\log n)
\end{aligned}$$

Therefore, the runtime of discarding the last level is in $\Theta(n \log n)$. In the best case, the remaining complete binary tree is also a sorted array. We can obtain the recurrence equation:

$$T(n) \ge T(n/2) + \Theta(n \log n)$$

Using Master Theorem, we have $a = 1$, $b = 2$, $f(n) = \Theta(n \log n)$. This is case 3 of Master Theorem, since there exists $\epsilon = 1$ that $f(n) = \Theta(n \log n) = \Omega(n^{\log_b a + \epsilon} n) = \Omega(n)$, and there exists $c = \frac{1}{2}$ that $af(\frac{n}{b}) = f(\frac{n}{2}) = \Theta(\frac{n}{2} \log \frac{n}{2}) \le \frac{1}{2}\Theta(n \log n)$. Therefore, $T(n) = \Theta(n \log n)$.

Since we use $\ge$ in the recurrence equation, what we have proved is that $T(n) \ge \Theta(n \log n)$. Therefore, $T(n) = \Omega(n \log n)$. Hence, the runtime of HeapSort on an already sorted array is in $\Omega(n \log n)$.