王若琛 2230025907

4.3-2

Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.

Solution: We guess $T(n) \le c \lg(n-a)$.

$$
\begin{aligned}
T(n) &\le c \cdot \lg(\lceil n/2 \rceil - a) + 1 \\
&\le c \cdot \lg((n+1)/2 - a) + 1 \\
&= c \cdot \lg((n+1-2a)/2) + 1 \\
&= c \cdot \lg(n+1-2a) - c \cdot \lg 2 + 1 && (c \ge 1) \\
&\le c \cdot \lg(n+1-2a) && (a \ge 1) \\
&\le c \cdot \lg(n-a)
\end{aligned}
$$

4.4-3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.

Solution: The subproblem size for a node at depth i is $n/2^i$. Thus, the tree has $\lg n + 1$ levels and $4^{\lg n} = n^2$ leaves. The total cost over all nodes at depth i is $4^i(n/2^i + 2) = 2^i n + 2 \cdot 4^i$.

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n - 1} (2^i n + 2 \cdot 4^i) + \Theta(n^2) \\
&= \sum_{i=0}^{\lg n - 1} 2^i n + \sum_{i=0}^{\lg n - 1} 2 \cdot 4^i + \Theta(n^2) \\
&= \frac{2^{\lg n} - 1}{2 - 1} n + 2 \cdot \frac{4^{\lg n} - 1}{4 - 1} + \Theta(n^2) \\
&= (2^{\lg n} - 1)n + \frac{2}{3}(4^{\lg n} - 1) + \Theta(n^2) \\
&= (n - 1)n + \frac{2}{3}(n^2 - 1) + \Theta(n^2) \\
&= \Theta(n^2)
\end{aligned}
$$

We guess $T(n) \le c(n^2 - dn)$

$$
\begin{aligned}
T(n) &= 4T(n/2 + 2) + n \\
&\le 4c[(n/2 + 2)^2 - d(n/2 + 2)] + n \\
&= 4c(n^2/4 + 2n + 4 - dn/2 - 2d) + n \\
&= cn^2 + 8cn + 16c - 2cdn - 8cd + n \\
&= cn^2 - cdn + 8cn + 16c - cdn - 8cd + n \\
&= c(n^2 - dn) - (cd - 8c - 1)n - (d - 2) \cdot 8c \\
&\le c(n^2 - dn)
\end{aligned}
$$

Where the last step holds for $cd - 8c - 1 \ge 0$.

6.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

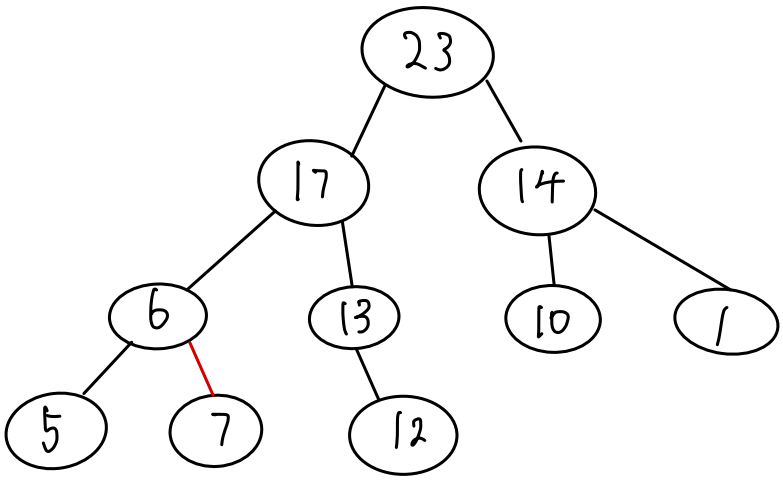
Solution: This is way too obvious, that it is hard to prove it.

The number of internal nodes a complete binary tree has is $2^h - 1$ where h is the height of the tree. A heap of height h has at least one additional node (otherwise it would be a heap of length $h-1$) and at most 2^h additional nodes (otherwise it would be a heap of length $h+1$).

Thus, if $n \in (2^h, 2^{h+1} - 1)$, then the height will be $\lfloor \lg n \rfloor$

6.1-6
Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

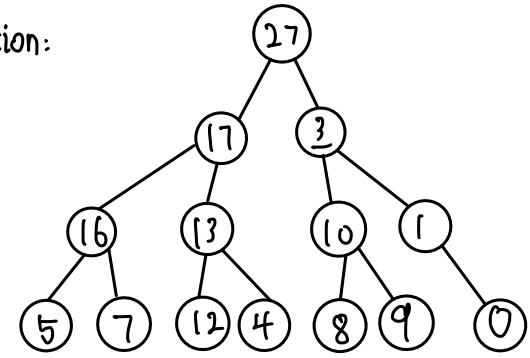
Solution: No. The property is violated by the next-to-last leaf (illustrated below in red)

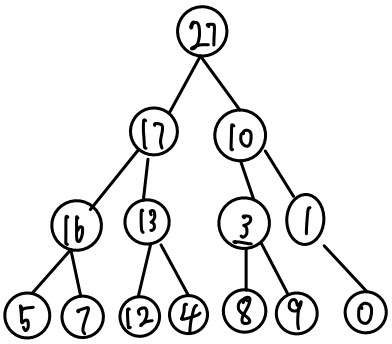


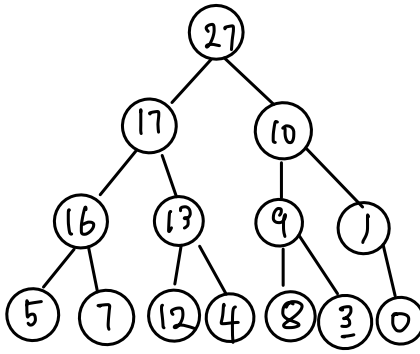
6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

Solution:







6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

Solution:

```
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[i]
        smallest = r
    if smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

The running time is the same. Actually, the algorithm is the same with the exceptions of two comparisons and some names.

6.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Solution: First, let's observe that the number of leaves in a heap is $\lceil n/2 \rceil$. Let's prove it by induction on h.

Base: $h=0$, The number of leaves is $\lceil n/2 \rceil = \lceil 2/2^{0+1} \rceil$

Step: Let's assume it holds for nodes of height $h-1$. Let's take a tree and remove all it's leaves. We get a new tree with $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ elements.

Note that the nodes with height h in the old tree height $h-1$ in the new one.

We will calculate the number of such nodes in the new tree. By the inductive assumption we have that T, the number of nodes with height $h-1$ in the new tree, is:

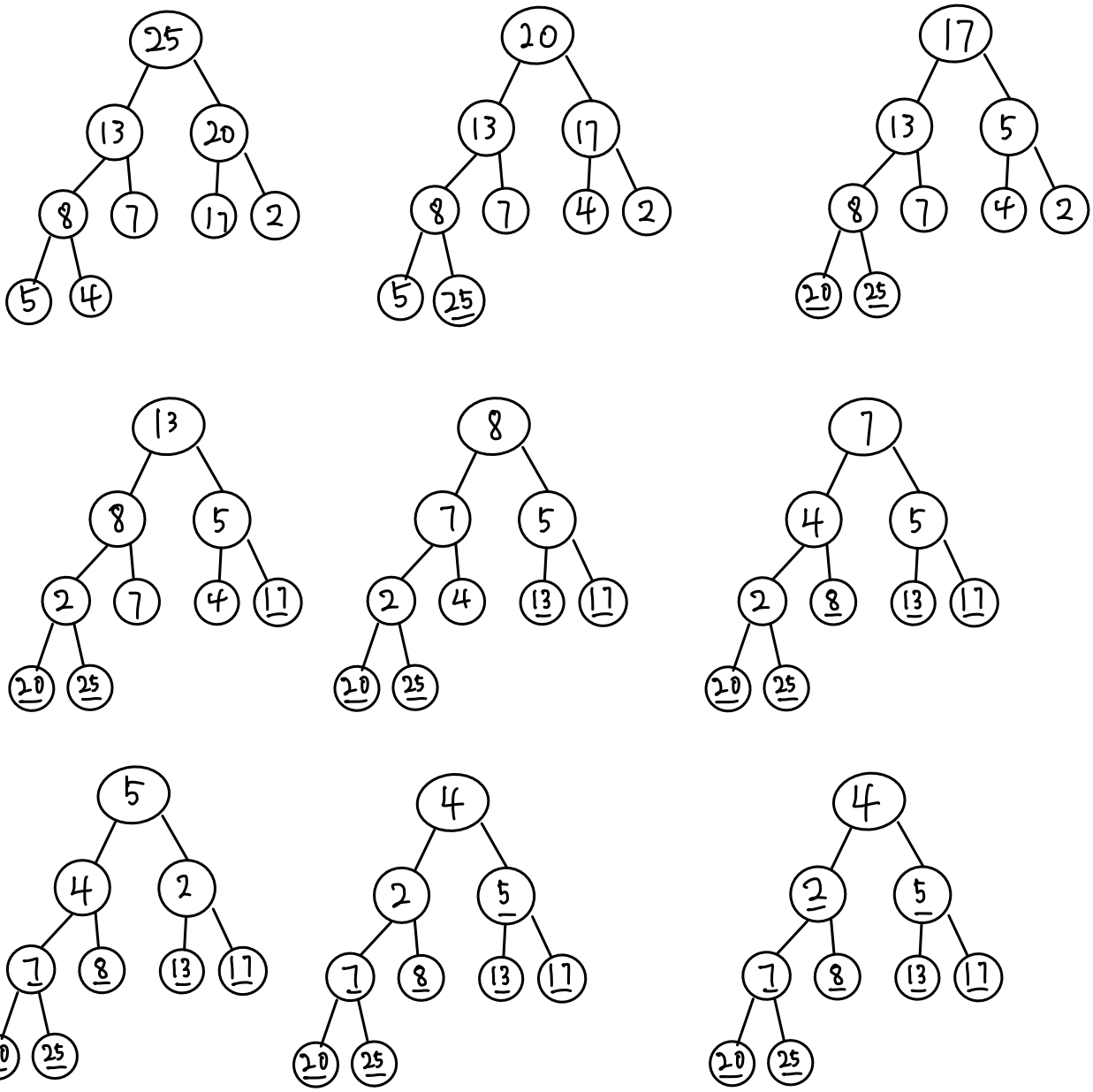
$$T = \lceil \lfloor n/2 \rfloor / 2^{h-1+1} \rceil < \lceil (n/2)/2^h \rceil = \lceil \frac{n}{2^{h+1}} \rceil$$

As mentioned, this is also the number of nodes with height h in the old tree.

6.4-1

Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

Solution :



6.5-3

Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

```
import sys
from math import inf

def parent(i):
    return (i - 1) // 2

def left(i):
    return 2 * i + 1

def right(i):
    return 2 * i + 2

class MinHeap:
    def __init__(self, capacity):
        self.elements = [0] * capacity
        self.length = capacity
        self.heap_size = 0

    def heap_minimum(self):
        return self.elements[0]

    def min_heapify(self, i):
        l = left(i)
        r = right(i)
        smallest = i

        if l < self.heap_size and self.elements[l] < self.elements[i]:
            smallest = l
        if r < self.heap_size and self.elements[r] < self.elements[smallest]:
            smallest = r

        if smallest != i:
            self.elements[i], self.elements[smallest] = self.elements[smallest], self.elements[i]
            self.min_heapify(smallest)

    def heap_extract_min(self):
        if self.heap_size == 0:
            raise Exception("heap underflow")

        min_element = self.elements[0]
        self.elements[0] = self.elements[self.heap_size - 1]
        self.heap_size -= 1
        self.min_heapify(0)

        return min_element

    def heap_decrease_key(self, i, key):
        if key > self.elements[i]:
            raise Exception("new key is larger than current key")

        self.elements[i] = key
        while i > 0 and self.elements[parent(i)] > self.elements[i]:
            self.elements[i], self.elements[parent(i)] = self.elements[parent(i)], self.elements[i]
            i = parent(i)

    def min_heap_insert(self, key):
        if self.length == self.heap_size:
            raise Exception("heap overflow")

        self.heap_size += 1
        self.elements[self.heap_size - 1] = inf
        self.heap_decrease_key(self.heap_size - 1, key)
```

7.2-2
What is the running time of QUICKSORT when all elements of array A have the same value?

Solution: It's $\Theta(n^2)$, since one of the partitio is always empty.

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

Solution: If the array is already sorted in decreasing order, then, the pivot element is less than all the other elements. The partition steps takes $\Theta(n^2)$ time, and then leaves you with a subproblem of size $n-1$ and a subproblem of size $n-1$ and a subproblem of size 0. This gives us the recurrence considered in that:

$$T(n) = T(n-1) + c_2 n$$
$$\leq c_1(n-1)^2 + c_2 n$$
$$= c_1 n^2 - 2c_1 n + c_1 + c_2 n \quad (2c_1 > c_2, n \geq c_1/(2c_1 - c_2))$$
$$\leq c_1 n^2$$

Which we showed has a solution that is $\Theta(n^2)$.

22-3 Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

b. Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.)

Solution: a. First, we'll show that it is necessary to have in degree equal out degree for each vertex. Suppose that there was some vertex v for which the two were not equal, suppose that in-degree(v) - out-degree(v). Note that we may assume that in degree is greater because otherwise we would just look at the transpose graph in which we traverse the cycle backwards. If v is the start of the cycle as it is listed, just shift the starting and ending vertex to any other one on the cycle. Then, in whatever cycle we take going though v, we must pass thrugh v some number of times, in particular, after we pass through it a times, the number of unused edges coming out of v is zero, however, there are still unused edges goin in that we need to use. This means that there is no hope of using those while still being a tour, because we could never be able to escape v and get back to the vertex where the tour started. Now, we show it is sufficient to have the in degree and out degree equal for every vertex. To do this, we will generalize the problem slightly so that it is more amenable to an inductive approach. That is, we will show that for every graph G that has two vertex v and u so that all the vertics have the same in and out degree except that the indegree is one greater for v, then there is an Euler path from v to u. This clearly lines up with the

Original statement if we pick $u = v$ to be any vertex in the graph. We now perform induction on the number of edges. If there is only a single edge. If there is only a single edge, then, suppose that we start at v and take any edge coming out of it. Consider the graph that is obtained from removing that edge, it inductively contains an Euler tour that we can just post-pend to the edge that we took to get out of v.

b. To actually get the Euler circuit, we can just arbitrarily walk any way that we want so long as we don't repeat an edge, we will necessarily end up with a valid Euler tour. This is implemented in the following algorithm, EULER-TOUR(G) which takes time $O(|E|)$. It has this runtime because the for loop will get run for every edge, and takes a constant amount of time. Also, the process of initializing each edge's color will take time proportional to the number of edges.