Analysis of Algorithms: Solutions 4

```
                     X
                     X  X  X
                     X  X  X
               X     X  X  X  X  X
number of      X     X  X  X  X  X
homeworks      X     X  X  X  X  X
               X  X  X  X  X  X  X
            X  X  X  X  X  X  X  X
            ----------------------
            5  6  7  8  9  10 11 12
                    grades
```

Problem 1
Give an efficient implementation of a Heap-Increase-Key(A, i, k) algorithm, which sets $A[i] \leftarrow \max(A[i], k)$ and updates the heap structure appropriately. Determine its time complexity and briefly explain your answer.

Heap-Increase-Key(A, i, k)
if $k > A[i]$
  then **while** $i > 1$ and $A[\text{Parent}(i)] < k$
    do $A[i] \leftarrow A[\text{Parent}(i)]$
      $i \leftarrow \text{Parent}(i)$
  $A[i] \leftarrow k$

The worst-case running time is proportional to the height of the heap; hence, it is $O(\lg n)$.

Problem 2
Using Figure 8.1 (page 155) in the textbook as a model, illustrate the operation of the Partition algorithm (which is a subroutine of Quick-Sort) on the following array:

4 5 1 4 2 1 8 3

The values in the array are as follows:

initially:	4	5	1	4	2	1	8	3
after the first exchange:	3	5	1	4	2	1	8	4
after the second exchange:	3	1	1	4	2	5	8	4
after the third exchange:	3	1	1	2	4	5	8	4

Problem 3
Briefly describe how to adapt (a) MERGE-SORT and (b) QUICK-SORT to sort elements stored in a linked list, without copying them into an array. Give the time complexity of your algorithms; is it the same as the complexity of sorting an array?

We assume that every element x of a linked list has two fields, $next[x]$ and $key[x]$. The *next* field points to the next element of the linked list, whereas *key* contains a numeric value. If x is the last element in the list, then $next[x]$ is NIL.

We describe the modified versions of MERGE-SORT and QUICK-SORT procedures. The time complexity of both procedures is the same as that for sorting arrays.

(a) The MERGE-SORT procedure gets two arguments, the first element of a linked list and the number of elements in the list. The procedure finds the middle of the list and cuts it in two sublists, y and z. Then, it makes recursive calls to sort these sublists. The MERGE procedure is similar to that for arrays; however, it may be implemented to sort in-place.

```
MERGE-SORT(x, n)        ▷ x is the first element; n is the number of elements
if n > 1
   then q ← ⌊n/2⌋
        y ← x
        for i ← 1 to q − 1        ▷ find the middle of the list
           do x ← next[x]
        z ← next[x]        ▷ beginning of the second sublist
        next[x] ← NIL        ▷ end of the first sublist
        y ← MERGE-SORT(y, q)        ▷ sort the first sublist
        z ← MERGE-SORT(z, n − q)        ▷ sort the second sublist
        return MERGE(y, z)        ▷ return the sorted list
```

(b) The QUICK-SORT procedure gets the first element of a linked list, and calls PARTITION to divide the input list into two lists. The PARTITION procedure uses the *key* value of the first element as the "pivot" for partitioning, and constructs two new linked lists: one with the values smaller than the pivot, and the other with the values larger than the pivot; it returns both lists. After calling PARTITION, the QUICK-SORT procedure makes recursive calls to sort the two lists, and then appends the second sorted list to the end of the first one.

```
QUICK-SORT(x)        ▷ x is the first element of the list
if next[x] ≠ NIL        ▷ more than one element?
   then y, z ← PARTITION(x)        ▷ PARTITION returns two lists
        y ← QUICK-SORT(y)
        z ← QUICK-SORT(z)
        x ← y
        while next[x] ≠ NIL        ▷ find the end of the first list
           do x ← next[x]
        next[x] ← z        ▷ append the second list to the end of the first one
        return y
```

```
PARTITION(x)
k ← key[x]         ▷ k is the pivot for partitioning
y ← NIL        ▷ list of elements smaller than k
z ← NIL        ▷ list of elements greater than k
while x ≠ NIL
    do x-next ← next[x]
       if key[x] ≤ k
          then      ▷ add x to the smaller-element list
                next[x] ← y
                y ← x
          else      ▷ add x to the larger-element list
                next[x] ← z
                z ← x
       x ← x-next       ▷ move to the next element
return y, z
```

Problem 4

A *d-ary heap* is like a binary heap, but instead of 2 children, nodes have d children.

(a) How would you represent a d-ary heap in an array? What are the expressions for determining the parent of a given element, $\textsc{Parent}(i)$, and a j-th child of a given element, $\textsc{Child}(i, j)$, where $1 \leq j \leq d$?

The following expressions determine the parent and j-th child of element i (where $1 \leq j \leq d$):

$$\begin{aligned} \textsc{Parent}(i) &= \left\lfloor \frac{i + d - 2}{d} \right\rfloor, \\ \textsc{Child}(i, j) &= (i - 1)d + j + 1. \end{aligned}$$

(b) What is the height of a d-ary heap of n elements in terms of n and d? You need to give an *exact* expression for the height, without using the Θ-notation.

The height h of a heap is *approximately* equal to $\log_d n$. The exact height is

$$h = \lceil \log_d(nd - n + 1) - 1 \rceil.$$