Heaps, Heapsort, Priority Queues

A *complete binary tree* is a binary tree where the nodes are numbered from 1 to n (top to bottom and left to right within each level) and node $\lfloor k/2 \rfloor$ is the parent of node k.

Let $K_1, K_2, \cdots, K_n$ be chosen from a set of *keys* on which a *total order* has been defined. In a total order the following two laws hold:

1. for any two keys K_i and K_j, exactly one of the relations $K_i < K_j$, $K_i = K_j$, or $K_i > K_j$ holds.
2. if $K_i < K_j$ and $K_j < K_l$, then $K_i < K_l$.

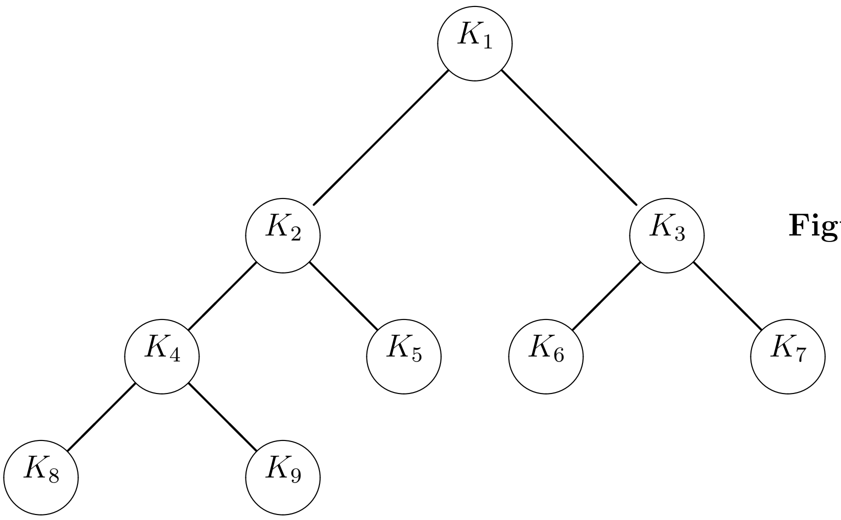

Figure 1.

Further, let the keys K_i be assigned to the nodes of a complete binary tree in level order. Figure 1 illustrates this assignment for $n = 9$. We define such a binary tree to be a *heap* provided the key K_i at each node of the tree is greater than or equal to the keys K_{2i} and K_{2i+1} of its children nodes. We say that the keys $K_1, K_2, \cdots, K_n$ satisfy the *heap property* provided $K_{\lfloor j/2 \rfloor} \geq K_j$

for every j satisfying the relation $1 \leq \lfloor j/2 \rfloor < j \leq n$.

Note that every subtree of a heap is a heap, and by transitivity the root of a heap is the largest key in the heap.

Suppose now that we have a complete binary tree whose nodes contain keys not satisfying the heap property. How can we exchange keys between parent-child pairs in the tree so that a heap can be established? Consider a system of promotions in a hierarchy, in which, if the value of a key K_j at a node is greater than the value of the key $K_{\lfloor j/2 \rfloor}$ at its parent node, we exchange K_j and $K_{\lfloor j/2 \rfloor}$. (This process resembles the mad rush characterized by the Peter Principle, in which each person in a hierarchical organization tends to be promoted until he reaches his level of incompetence - at which point he is said to have achieved *final placement*.) Repeatedly performing such promotions until no more can be performed will convert the hierarchy into a heap. For example, starting with the tree of Figure 2, we could perform the following promotions (let $a \leftrightarrow b$ stand for exchanging a and b):

$$3 \leftrightarrow 2, 3 \leftrightarrow 1, 2 \leftrightarrow 1, 4 \leftrightarrow 2, 4 \leftrightarrow 3, 6 \leftrightarrow 5, 6 \leftrightarrow 4, 4 \leftrightarrow 5, 7 \leftrightarrow 5, 6 \leftrightarrow 7.$$

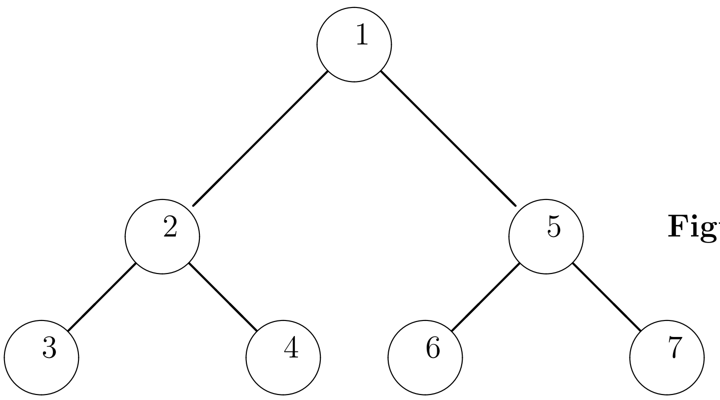

Figure 2.

The tree resulting from these ten promotions is given in Figure 3.

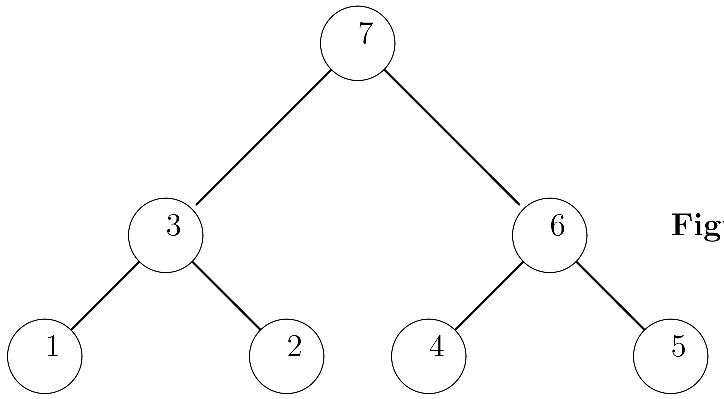

Figure 3.

It turns out that using ten promotions to establish this heap is rather wasteful. For instance, only four promotions,

$$7 \leftrightarrow 5, 4 \leftrightarrow 2, 7 \leftrightarrow 1, 6 \leftrightarrow 7.$$

are needed to establish the heap of Figure 4 starting with the tree of Figure 2.

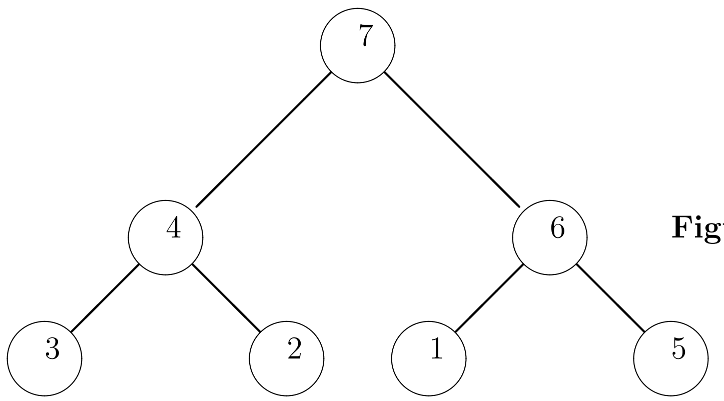

Figure 4.

Given n keys, it is interesting to inquire whether we can arrange them into a heap in at most $O(n)$ exchanges. A bottom-up process can be used to solve this problem. Let us suppose that we are given a binary tree of the form

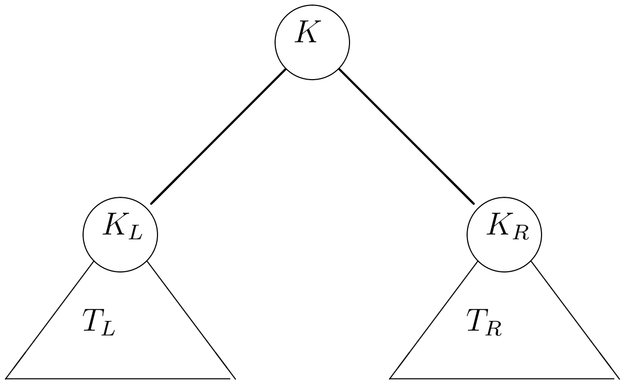

Figure 5.

where K is the key in the root of the tree, and K_L and K_R are the keys in the respective roots of the left and right subtrees T_L and T_R. Suppose, further that the left and right subtrees have already been arranged into heaps. Then the only way in which the whole tree T could fail to be a heap is if K is less than K_L or less than K_R. If this is the case, we can exchange K with the *larger* of K_L and K_R. Without loss of generosity, suppose $K_R \geq K_L$. Then we exchange K and K_R, giving a new tree

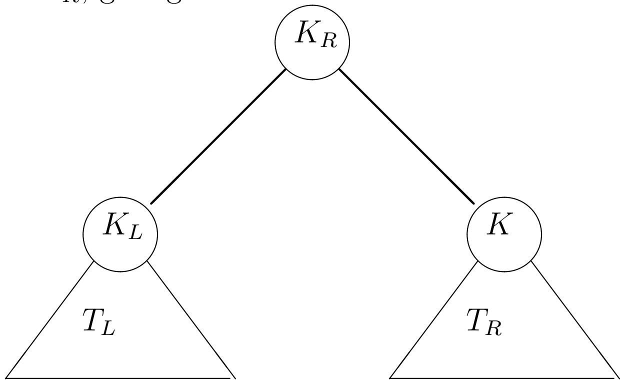

Figure 6.

Since T_L is already a heap, and since $K_R \geq K_L$, the only way in which this new tree could fail to be a heap is that key K could be less than the keys of one or both of its new children in tree T_R. These new children are already roots of respective heaps since they are roots of subtrees of the original tree T_R. Hence, the altered subtree T_R with new root K is a tree which, like the original tree, departs from the heap property only at its root if at all. The key at the root of T_R can be exchanged repeatedly with the larger of the keys of its children, until, after some amount of downward travel, it comes to rest at a node where it is not less than the keys of its children. At this point, the entire original tree is a heap.

This process, originally called the *sift-up* procedure by Floyd is given more precisely by the following algorithm.

Algorithm 1 *Sift up.*

Let T point to the root of a nonempty binary tree with key K at its root, and let T_L and T_R be subtrees that are heaps. The key K is repeatedly exchanged with the larger of the keys of the children of the node where it currently resides until T is made into a heap. For convenience, if a subtree is the empty tree Λ, we assume that the key of its root is a quantity $-\infty$ less than every key K_i.

1. [Initialize.] Set $N \leftarrow T$. (N is the current node containing key K).

2. [Extract keys and subtrees.] $K \leftarrow$ key of N. $T_L \leftarrow$ left subtree of N. $T_R \leftarrow$ right subtree of N. $K_L \leftarrow$ key of T_L. $K_R \leftarrow$ key of T_R.

3. [Terminate?] If $K \geq K_L$ and $K \geq K_R$, the algorithm terminates.

4. [Exchange with bigger child.] If $K_L > K_R$, then exchange the keys of N and T_L, set $N \leftarrow T_L$, and goto step 2. Otherwise, exchange the keys of N and T_R, set $N \leftarrow T_R$, and goto step 2.

Here is the same algorithm using array representation for the heap.

MAX-HEAPIFY(A, i)
$l \leftarrow$ Left(i)
$r \leftarrow$ Right(i)
$largest \leftarrow i$ *This step is mising in your book*
if $l \leq$ *heap-size*$[A]$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq$ *heap-size*$[A]$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY(A, $largest$)

What is the running time of MAX-HEAPIFY?? Could you write a recurrence relation for it?

Given a complete binary tree T, not initially a heap, we can convert T into a heap by repeatedly applying the MAX-HEAPIFY procedure, first to its smallest subtrees, and then later to subtrees whose left and right subtrees have already been made into heaps. Any order of application of sift-up to the nodes of T which processes the subtrees of each node before it processes the node itself will suffice to create a heap out of the whole tree. Specifically, we can assume that we apply MAX-HEAPIFY to the nodes in the reverse level order–i.e., from bottom to top and right to left within each level.

BUILD-MAX-HEAP(A)
heap-size$[A] \leftarrow$ *length*$[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i)

What is running time of BUILD-MAX-HEAP?

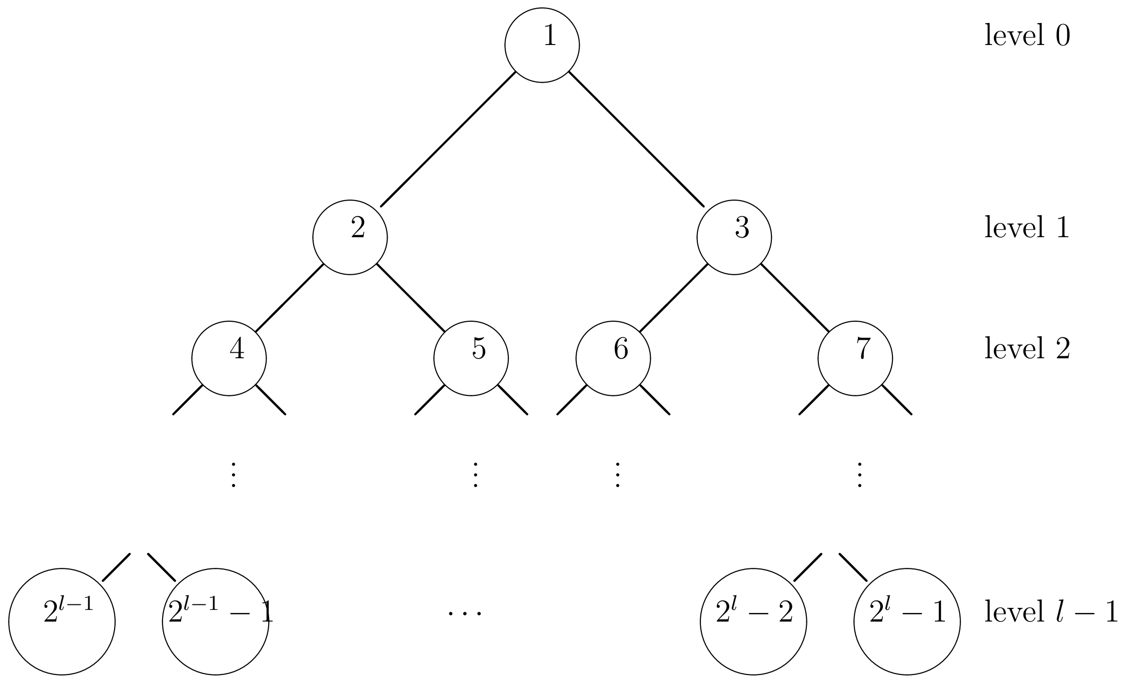

Figure 7.

The running time for sift-up is proportional to the number of times it exchanges keys in step 4. Let us now show that the number of key exchanges required in applying sift-up to all n nodes of a complete binary tree is at most $O(n)$.

Suppose we have a tree of l levels as in Figure 7. A key K in a node at level i could be exchanged with children along any downward path at most $(l-1-i)$ times before coming to rest–since, in the worst case, it would come to rest in a leaf at the bottommost level $l-1$. Since there are 2^i nodes on level i of the tree, each of whose keys could be exchanged at most $(l-1-i)$ times, the total number of exchanges required in applying sift-up to all nodes in the tree could not exceed E, where

$$E = \sum_{0 \le i \le l-1} 2^i \,(l-1-i)$$

By exchanging $(l-1-i)$ and i, the sum for E becomes

$$E = \sum_{0 \le l-1-i \le l-1} 2^{l-1-i}(i) = \sum_{0 \le i \le l-1} i2^{(l-1)-i}$$

Here 2^{l-1} can be removed as a factor, and the term for $i = 0$ can be dropped, giving

$$E = 2^{l-1} \sum_{1 \le i \le l-1} \frac{i}{2^i}$$

But since it can be shown that $\sum_{1 \le i \le l-1} (i/2^i) < 2$ we get

$$E = 2^{l-1} \sum_{1 \le i \le l-1} \frac{i}{2^i} < 2^{l-1} \cdot 2 = 2^l$$

So E is bounded above 2^l, where l was the number of levels in the original tree. If n is any number of nodes sufficient for at least one node to reside on level l, then n lies in the range $2^{l-1} \le n \le 2^l - 1$. Hence, $2^l \le 2n$. Putting these results together, we see that the total number of exchanges required in applying sift-up to all nodes of the tree is at most E, which is, in turn, bounded above by 2^l, which is at most $2n$. Therefore, the number of exchanges required to make an n-node complete tree into a heap is $O(n)$.

A heap is used as the basic data structure of the Heapsort algorithm. The basic ideas behind this sorting algorithm are as follows:

1. Take the n keys to be sorted $K_1, K_2, \cdots, K_n$ and consider them to be arranged in a complete binary tree.

2. Convert this tree into a heap by applying sift-up to the nodes in reverse level order. (This takes time $O(n)$.)

3. Repeatedly do the following steps (a), (b), (c) until the heap is empty:

   (a) Remove the key at the root of the heap (which is the largest in the heap) and place it on an output queue.

   (b) Detach from the heap the rightmost leaf node at the bottommost level, extract its key K, and replace the key at the root of the heap with K.

   (c) Finally, apply sift-up to the root to convert the tree to a heap once again.

Here is the algorithm:

HEAPSORT(A)
BUILD-MAX-HEAP(A)
for $i \leftarrow length[A]$ **downto** 2
  do exchange $A[1] \leftrightarrow A[i]$
    $heap\text{-}size[A] \leftarrow heap\text{-}size[A]$ - 1
    MAX-HEAPIFY(A, 1)

Application of Heaps–Priority queues: In a number of applications, we have a set of items on which we perform only two operations:

- add an item to the current set
- extract the item from the set having maximum (minimum) value.

For example, in discrete-event simulation systems, we wish to simulate events in the temporal order in which they occur. The simulator may schedule future events by adding events to the current set, and it must be able to extract next (i.e., minimum time) event, in order to know which future event to simulate next. Often, the operations *add* and *extract* come in pairs or, over the course of an algorithm occur in equal numbers. Similar requirements for a *largest-in, first-out* set representation are found in algorithms for such tasks as: (a) operating-system task scheduling, (b) iteration in numerical schemes based on the idea of repeated selection of an item with smallest test criterion.

Let us call a set with a largest-in, first-out behavior a *priority queue* (since the value of each item establishes its priority for leaving the queue). It is obvious that there are a number of different representations for priority queues. We could keep all n items in the queue in a sorted list– in which case, extraction takes constant time but addition takes time $O(n)$; or we could leave the items in random order in a sequential list– in which case addition takes constant time, but extraction takes time $O(n)$. An advantage of the heap representation is that addition and extraction each take $O(\log n)$ time. This becomes very advantageous for large n. For example, it has been shown that for a heap containing 1000 items, the average number of comparisons needed to do an insertion followed by an extraction is just 12!.

A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called **key**. A **max-priority queue** supports the following operations.

- INSERT(S, x) inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$
- MAXIMUM(S) returns the element of S with the largest key
- EXTRACT-MAX(S) removes and returns the element of S with the largest key
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value

Here are the pseudo code for the max-priority queue operations:

HEAP-MAXIMUM(A)
return$A[1]$

HEAP-EXTRACT-MAX(A)
if $\textit{heap-size}[A] < 1$
   then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[\textit{heap-size}[A]]$
$\textit{heap-size}[A] \leftarrow \textit{heap-size}[A]$ - 1
MAX-HEAPIFY (A, 1)
return max

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
     do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

```
MAX-HEAP-INSERT (A, key)
heap-size ← heap-size[A] + 1
A[heap-size[A]] ← −∞
HEAP-INCREASE-KEY(A, heap-size[A], key)
```