CIS 575. Introduction to Algorithm Analysis
Material for March 4, 2024

The Heapsort Algorithm

The topic of this note is covered in *Cormen*'s Section 6.4.

1 HeapSort

We shall present an algorithm, known as HEAPSORT (and described, for example, on p.170 in *Cormen*) that uses a binary heap to sort an array $A[1..n]$ (with $n \geq 1$).

The **first** step of the algorithm is to convert, as described in the previous note, the given array into a **binary heap**.

In the **second** step of the algorithm, we from that heap repeatedly extract the largest element and put it at the end. This is done by a loop of the form

for i $\leftarrow n$ **downto** 2
  B

whose body B we shall now develop.

The **invariant** for that loop is (besides $A[1..n]$ being a permutation of its original values):

1. $A[i+1..n]$ consists of the $n-i$ largest elements, in non-decreasing order
2. $A[1..i]$ has the heap property.

With that loop invariant, the desired postcondition will hold at loop exit, since then $i = 1$ and thus the first part of the loop invariant tells us that $A[2..n]$ consist of the $n-1$ largest elements, in non-decreasing order — but then $A[1..n]$ will indeed consist of the n largest elements, in non-decreasing order.

As required, the loop invariant is established before the first loop iteration, when $i = n$, since then the first part holds *vacuously*, and the second part is true since the first step of the algorithm did convert $A[1..n]$ into a heap.

As the loop trivially terminates, our only remaining obligation is to construct a loop body that **maintains** the loop invariant. We can use a loop body B that

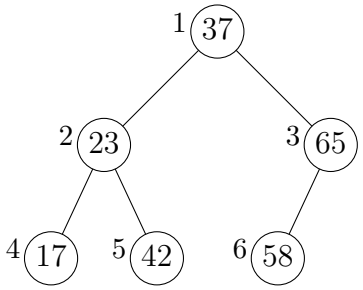
1. swaps $A[1]$ (the largest element of $A[1..i]$) with $A[i]$
2. sifts down $A[1]$ in the heap $A[1..i-1]$

since the first action will re-establish the first part of the loop invariant, and the second action will re-establish the second part of the loop invariant.

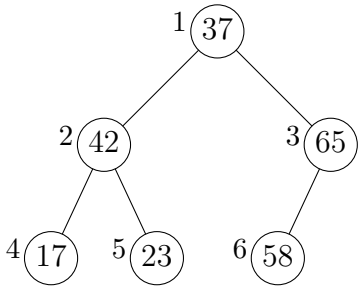
Example Let us see how to use heapsort to sort the array

1	2	3	4	5	6
37	23	65	17	42	58

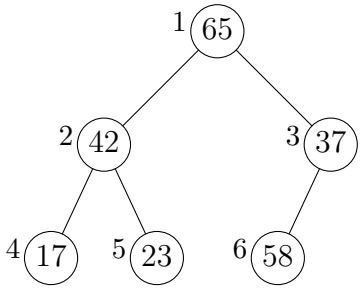
which represents the binary tree

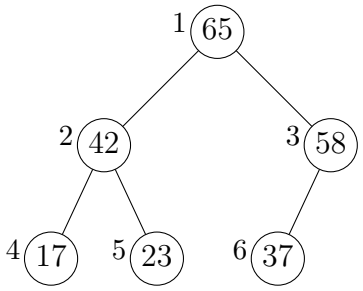


which we shall first convert to a binary heap: first we sift down 23

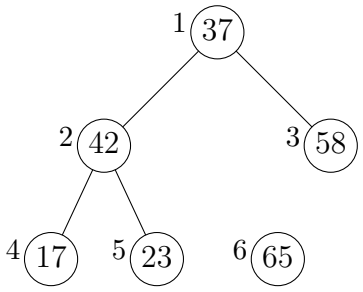


and next in two steps we sift down 37:

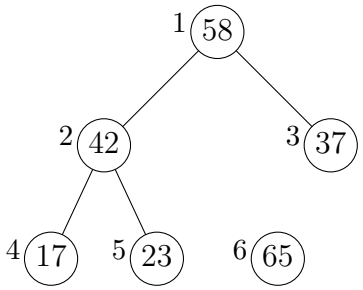




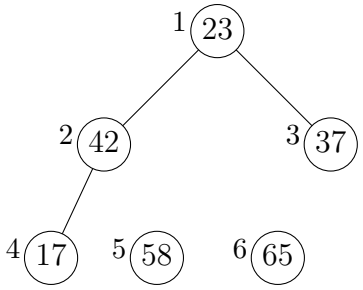
The root now contains the largest element, 65, which we swap with the last array element, $A[6]$:



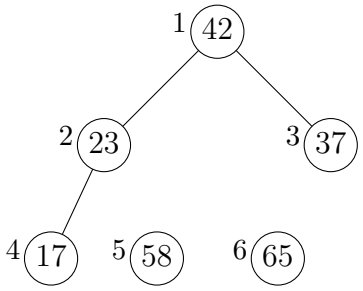
but for $A[1..5]$ to be a heap we have to sift down 37:



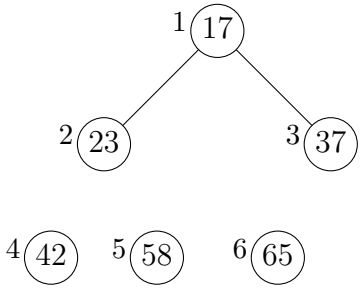
The root now contains the 2nd largest element, 58, which we swap with the 2nd last array element, $A[5]$:



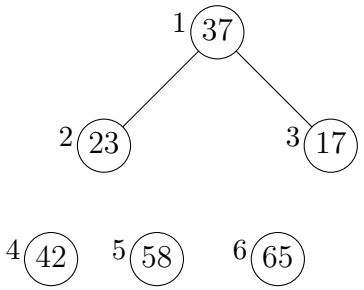
but for $A[1..4]$ to be a heap we have to sift down 23:



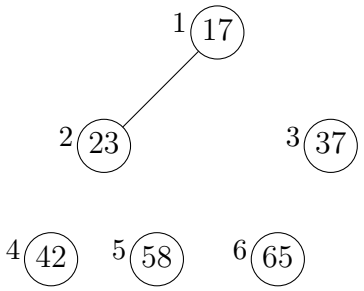
The root now contains the 3rd largest element, 42, which we swap with the 3rd last array element, $A[4]$:



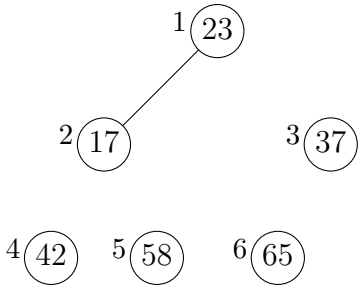
but for $A[1..3]$ to be a heap we have to sift down 17:



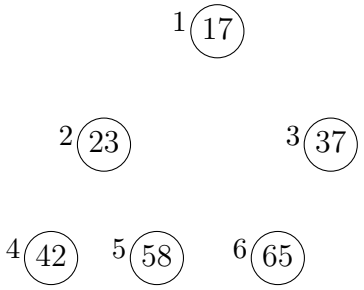
The root now contains the 4th largest element, 37, which we swap with the 4th last array element, $A[3]$:



but for $A[1..2]$ to be a heap we have to sift down 17:



The root now contains the 5th largest element, 23, which we put in the 5th last array location, $A[2]$:



and there is no more work to do (actually, the last four steps are redundant as they cancel out each other!)

We end up with the sorted array

1	2	3	4	5	6
17	23	37	42	58	65

Running Time Analysis A key part of the algorithm is to sift down $A[1]$ in the binary heap $A[1..i-1]$ which in the worst case takes time in $\Theta(\lg(i))$; the total time spent on that

is thus proportional to

$$\sum_{i=1}^{n} \lg(i)$$

which we know is in $\Theta(n \lg(n))$. As this dominates the other costs of the algorithm, such as converting into a heap which we saw could be done in $\Theta(n)$, we see that the **heapsort** algorithm

- runs in time $\mathbf{\Theta(n \lg(n))}$ which is like *merge sort* but which improves *insertion sort*
- and also (as it is iterative and based on swapping element) is **in-place** which is like *insertion sort* but which improves *merge sort*.