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

Converting a Tree Into a Heap

The topic of this note is covered in *Cormen*'s Section 6.3.

1 Converting a Tree Into a Heap

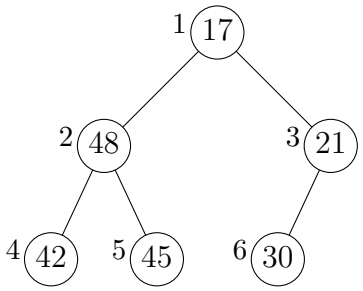
We shall now show how to convert a binary tree, represented (cf. the previous note) as an array $A[1..n]$ (and thus balanced except for some rightmost leaves possibly missing), into a binary heap.

One may consider the following approach: with the insertion operator as defined for priority queues (cf. a recent note), we first insert $A[2]$ into the binary heap $A[1]$ so as to form a binary heap $A[1..2]$, next insert $A[3]$ into the binary heap $A[1..2]$ so as to form a binary heap $A[1..3]$, etc. But recall that insertion involves percolating up a node which in the worst case will run in time proportional to the height of the current heap; hence the total time to insert n nodes will be proportional to

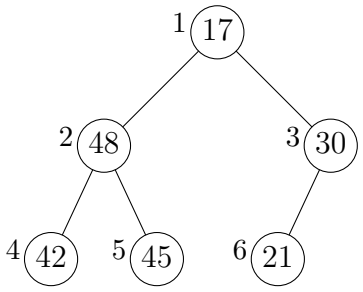
$$\sum_{i=1}^{n} \lg(i)$$

and hence (by *Howell*'s Theorem 3.28) in $\Theta(n \lg(n))$.

It turns out to be better to employ a bottom-up approach which we shall now illustrate on the below tree (the numbers depicted outside the nodes are array indices)

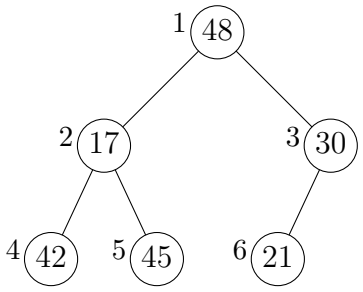


We first observe that each leaf is already trivially a heap. But the right child of the root is not (the root of) a heap, as it is smaller than its only child; hence we need to sift it down into

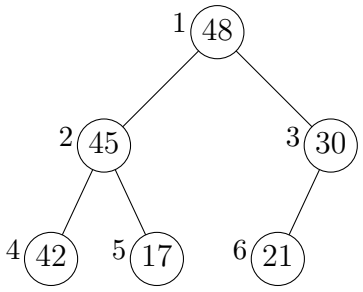


On the other hand, the left child of the root is (the root of) a heap, as it is greater than its two children.

We have thus converted all subtrees into heaps, except the tree itself. For that purpose, we need to sift down the root: first we get



and finally we arrive at what is indeed a binary heap:



We can describe this algorithm, in *Cormen*'s Section 6.3 referred to as Build-Max-Heap, as a loop where we process the nodes *downwards* (safely ignoring the last half as they are leaves):

```
Convert(A[1..n])
    for i ← ⌊n/2⌋ downto 1
        SiftDown(i)
```

Running Time Analysis Since each (non-leaf) node may be sifted down, an operation that may run in logarithmic time, we may think that the total time for heap conversion is in $\Theta(n \lg(n))$ (as for our first approach). But keep in mind that for most nodes, the sift down will involve at most one or two swaps; only for the nodes near the root there may be a significant number of swaps.

To formalize this reasoning, observe that we can give a *top-down* description of the conversion algorithm:

> for each node, recursively convert its child(ren) into heaps;
> then sift down the node.

This description suggests a recurrence

$$T(n) = 2T(\frac{n}{2}) + \lg(n) \tag{1}$$

which we can solve using the last case of (our version of) the Master Theorem: with $a = b = 2$ and thus $r = \log_b(a) = 1$, and with $\lg(n) \in O(n^q)$ for $q = 0.5$ (or any $q > 0$) and thus $q < r$, we have $T(n) \in \Theta(n^r) = \mathbf{\Theta(n)}$.

Still, you should be a little suspicious whether (1) is indeed an accurate recurrence; it may easily happen (when the tree is not fully balanced) that the left child is much larger than the right child and the proper recurrence is rather say $T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + \lg(n)$.

But take comfort in the observation that the Master Theorem as presented in *Howell* only requires (1) to hold when n is a power of 2. And that is indeed the case, since for such n, the tree is almost fully balanced.

Summary We have presented an algorithm that in **linear** time converts a binary tree, represented as an array, into a binary heap.