# CIS 575. Introduction to Algorithm Analysis
# Material for March 1, 2024

## The Heap Property and How to Preserve it



The topic of this note is covered in *Cormen*'s Sections 6.1–2.

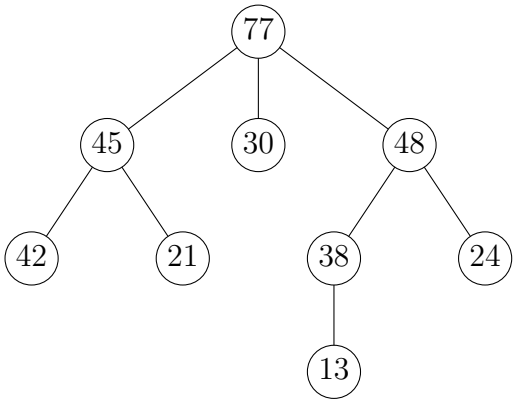
## 1 The Heap Property

A **heap** (not to be confused with garbage collected memory as found in say Java) is a rooted tree that satisfies the **heap property**:

> if $q$ is a child of $p$ then $key(p) \geq key(q)$

Thus parents have keys at least as great as their children. (Sometimes we shall want the dual property, that parents have keys not greater than their children; everything developed in the following can still be applied, *mutatis mutandis*.)

As an example heap, consider (while the nodes may be complex records we shall depict only their keys)




Observe that we do not **yet** impose any restrictions on the shape of the rooted tree forming a heap. Later, we shall restrict our attention to heaps that are binary, and "almost" balanced.

# 2 Maintaining the Heap Property

## 2.1 Sift Down

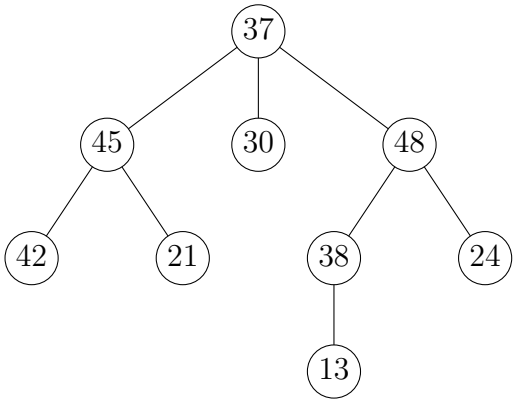
If a given rooted tree is a heap *except* that a node has a key that is **too small** then we have to "sift down" that node, as informally expressed by the algorithm

SiftDown($w$)

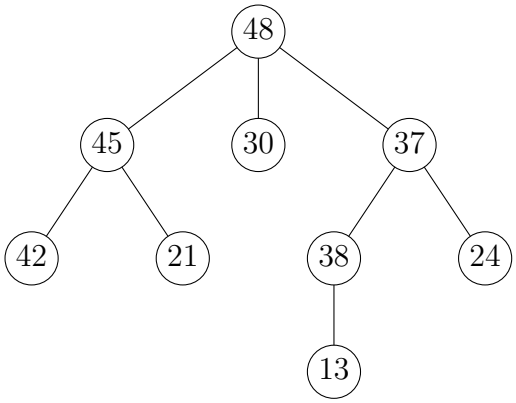
**while** $w$ has a child with greater key
*swap* $w$ and the child with the greatest key

which in *Cormen* is presented (in much more detail) in Section 6.2 as the Max-Heapify algorithm.

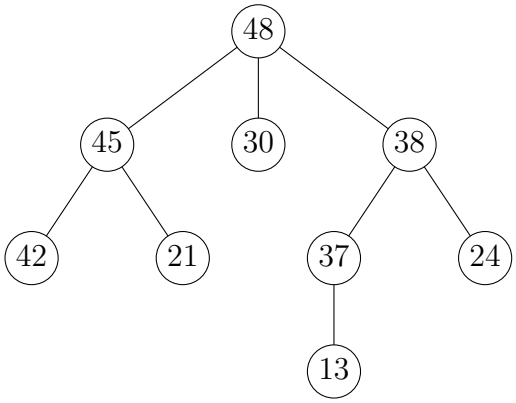
For example, assume that in the above heap, the key of the root is changed from 77 to 37:




Then we need to swap 37 with it largest child 48, giving




but we are not done yet; as 37 is less than its left child 38 we have to swap them:



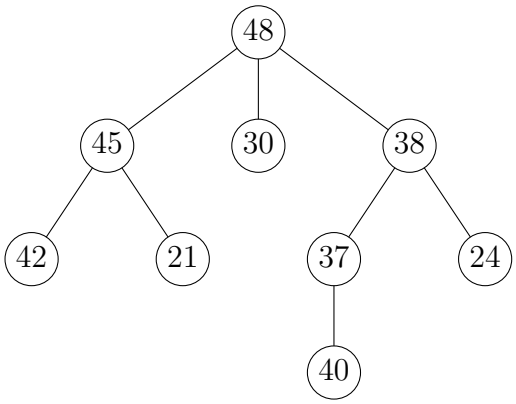

and now (as 37 is greater than its only child 13) we have restored the heap property.
The worst case running time of SiftDown is in $\mathbf{\Theta(h)}$ where $h$ is the **height** of the tree.

## 2.2 Percolate Up

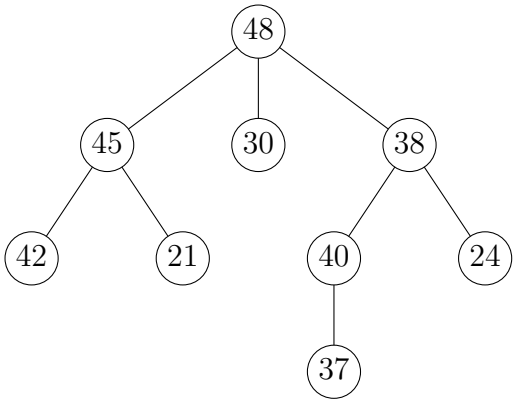
If a given rooted tree is a heap *except* that a node has a key that is **too great** then we have to "percolate up" that node, as informally expressed by the algorithm

Percolate($w$)
    **while** $w$ has parent with smaller key
        *swap* $w$ and its parent

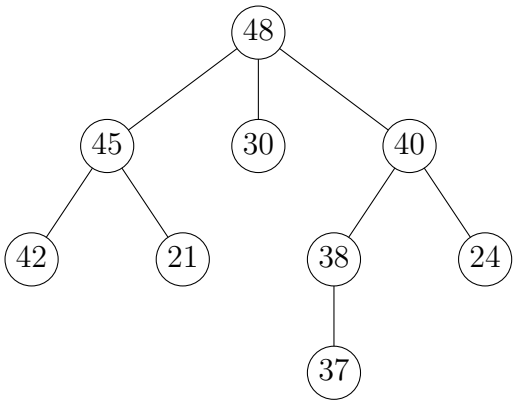
For example, assume that in the most recent heap depicted above, the leaf with key 13 instead gets key 40:




Then we need to swap 40 with its parent 37, giving



but we are not done yet; as 40 is greater than its parent 38 we have to swap them:




and now (as 40 is smaller than its parent 48) we have restored the heap property.
The worst case running time of PercolateUp is in $\mathbf{\Theta(h)}$ where $h$ is the **height** of the tree.

# 3 Priority Queues by Heaps

We shall now describe how to use a heap to implement a priority queue.

- for FindMax, we just return the record at the root.
- for Insert, we may insert the new node as a leaf **anywhere** in the heap (though we shall often prefer a spot that keeps the heap "reasonably balanced") but must then **percolate up** that leaf.
- for DeleteMax, we replace the root with an **arbitrary** leaf (though we shall often prefer a leaf whose removal keeps the shape reasonably balanced) but must then **sift down** that leaf.

Alternatively, for DeleteMax we could move up the largest child of the root into the root, then move up the largest child of that node, etc. But we prefer not to use that approach as it does not allow us to control the shape of the resulting heap.
We see that all operations are in $\Theta(h)$ with $h$ the height of the heap, except for FindMax which is in $\Theta(1)$. This may appear to fulfill our goal from the last notes, except that we have yet no guarantee that $h$ is much smaller than the number of nodes.