Binary Trees

Definitions:

- *binary tree*: a tree in which every node has at most two children.
- *binary search tree*: a *binary tree* with the additional requirement that for each node:
  - the values in the *left subtree are smaller* than the node's value
  - the values in the *right subtree are greater* than the node's value.

Node classification:

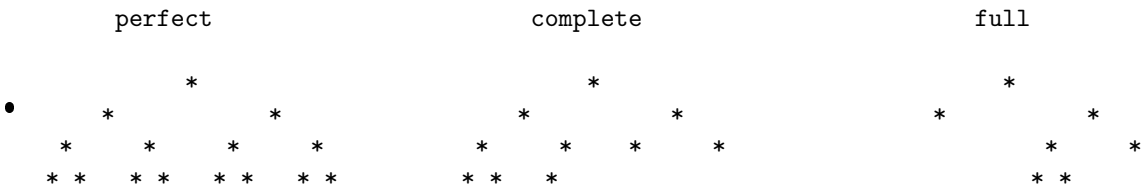
- *leaf*: a node that has no children
- *internal*: a node that is not a leaf node (i.e. has at least one child)

Height definition:

- the *height of a node* is the longest path (i.e. number of hops) from the node to a leaf
- the *height of a binary tree* is the height of the *root node* (i.e. number of levels minus 1)

Binary Tree classification (varies across textbooks):

- *perfect*: binary tree in which all levels, including the last, are *fully packed*, i.e.
  - all *internal nodes have two* children
  - all *leaves* are at the *same level*
- *complete*: binary tree in which all levels are *fully packed*, except for the last level, which may be missing nodes at the end; for example, *Heap*
- *full*: binary tree in which all nodes have wither 2 or 0 children; for example, *Huffman Tree*

- 

Height of a Perfect Binary Tree is $O(\log n)$

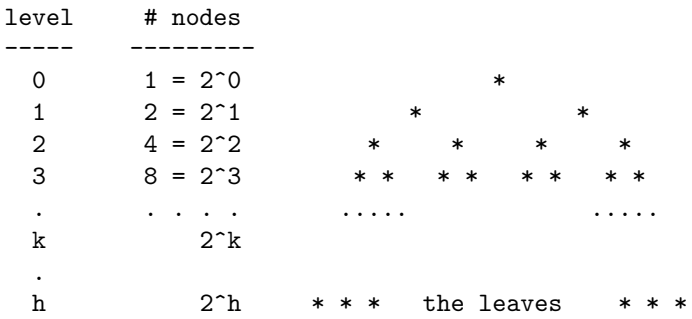
Let n be the number of nodes in a *perfect binary tree*.

Let l_k denote the umber of nodes on level k, where the levels are numbered 0, 1, 2, ..., h.

The last level, h, represents the *height* of the tree, i.e. the number of *hops*. Note, however, that the total number of levels is $h + 1$ since we count from 0.

Note that:

- $l_k = 2l_{k-1}$, i.e. each level has exactly twice as many nodes as the previous level (since each *internal* node has *exactly* two children)
- $l_0 = 1$, i.e. on the "first level" we have only one node (the root node).
- from CS201 the recurrence $l_k = 2l_{k-1}$ solves to $l_k = 2^k$, but we can also observe this as a pattern in the tree:



Our tree has a total of n nodes. Another way to count the total is to add the number of nodes on the individual levels:

$$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = n$$

From CS 201 we know that:

$$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = 2^{h+1} - 1$$

Therefore:

$$\begin{aligned} 1 + 2^1 + 2^2 + 2^3 + ... + 2^h &= n \\ 2^{h+1} - 1 &= n \\ 2^{h+1} &= n + 1 \\ \log_2 2^{h+1} &= \log_2(n+1) \\ (h+1)\log_2 2 &= \log_2(n+1) \\ h + 1 &= \log_2(n+1) \\ h &= \log_2(n+1) - 1 \end{aligned}$$

Finally, we have $h = \log_2(n+1) - 1$, or $h \approx \log_2 n$, so h is $O(\log n)$

Now that we know the *height of the tree* we can compute the number of leaves, l_h, in the tree. We observed earlier that $l_h = 2^h$ so we can substitute the value of h in this expressions:

$$l_h = 2^h = 2^{\log_2(n+1)-1} = 2^{\log_2(n+1)}/2^1 = (n+1)/2$$

(using $a^{b-c} = a^b/a^c$ and $a^{\log_a b} = b$)

In summary, we learned that:

- the *height* is $h = \log_2(n+1) - 1$, i.e. h is $O(\log n)$
- the *number of leaves* is $l_h = (n+1)/2$, i.e. roughly half of the nodes are at the leaves.

Make-Heap Running Time

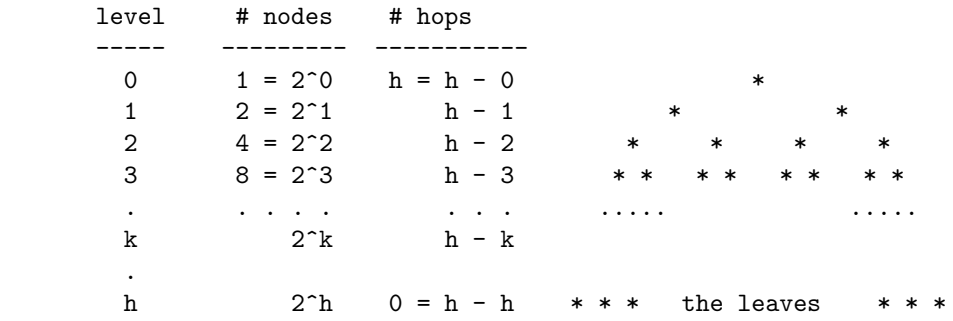
Here is the pseudocode for `Make-Heap`:

```
procedure Make-Heap():
    # in the actual implementation should start at farthest leaf
    for each node starting at the very end:
        PushDown(node)
```

Since there are n nodes in the heap and each node has to travel down at most a full height $h \approx \log n$, the running time appears to be $(n \log n)$.

The actual running time is $O(n)$. To show this, we need to carry out a more careful analysis of the actual work per node.

In the tree below we have indicated how much each node has to travel at worst. Note that the leaves have to make 0 hops (have nowhere to go) and the top node has to travel the full height h.



The work for a single level, k, is $2^k(h-k)$, since the level has 2^k nodes and each node has to make $(h-k)$ hops.

The total work is the sum for all levels:

$$
\begin{aligned}
\sum_{k=0}^{h} 2^k(h-k) &= \sum_{k=0}^{h} 2^k h \quad - \quad \sum_{k=0}^{h} 2^k k \\
&= h\left(\sum_{k=0}^{h} 2^k\right) \quad - \quad \sum_{k=0}^{h} 2^k k \\
&\qquad \downarrow \qquad\qquad\qquad\quad \downarrow \\
&\qquad \text{CS201 formula} \qquad \text{CS201-like formula} \\
&= h(2^{h+1}-1) \quad - \quad (2+(h-1)2^{h+1}) \\
&\qquad\qquad \text{using } h \approx h-1 \approx h+1 \\
&\approx h(2^h-1) \quad - \quad (2+h2^h) \\
&\approx h2^h - h - 2 - h2^h + 2^h \\
&= -h - 2 + 2^h \\
&\qquad\qquad \text{using } h \approx \log_2 n,\ a^{\log_a b} = b \\
&= -\log_2 n - 2 + n \\
&\approx n - \log_2 n - 2 \\
&\approx n
\end{aligned}
$$

This shows that `Make-Heap` is $O(n)$. Essentially, many nodes (bottom levels) had to do very little work, and only a few nodes (at the top) had to do a lot of work, so the amount of work balanced out.

The messier calculations:

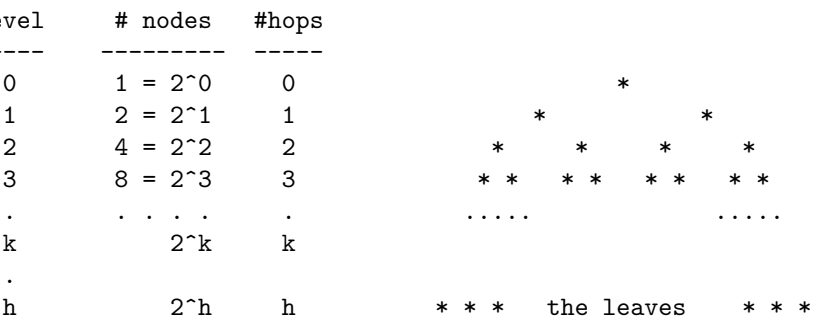
$$
\begin{aligned}
\sum_{k=0}^{h} 2^k(h-k) &= \sum_{k=0}^{h} 2^k h \quad - \quad \sum_{k=0}^{h} 2^k k \\
&= h\left(\sum_{k=0}^{h} 2^k\right) \quad - \quad \sum_{k=0}^{h} 2^k k \\
&\qquad \downarrow \qquad\qquad\qquad\quad \downarrow \\
&\overset{\text{CS201 formula}}{=} h(2^{h+1}-1) \quad - \quad \overset{\text{CS201-like formula}}{(2+(h-1)2^{h+1})} \\
&= h2^{h+1} - h - 2 - h2^{h+1} + 2^{h+1} \\
&= -h - 2 + 2^{h+1} \\
&\overset{\text{using } h = \log_2(n+1)-1,\ a^{\log_a b} = b}{=} -(\log_2(n+1)-1) - 2 + (n+1) \\
&= n - \log_2(n+1) \\
&\approx n
\end{aligned}
$$

Heap-Sort Running Time

Here is the pseudocode for `Heap-Sort`:

```
procedure Heap-Sort():
    # in the actual implementation repeat n-1 times
    repeat n times:
        item <- PopMax()
        put item at the end
```

Each time we pop the max, a value from the bottom level (from the leaves) is moved to the top and pushed down. This means that this time the leaves may have to travel quite a bit, so we have the following situation:



The work for a single level, k, is $2^k k$. This assumes that k has become the last level, so as we pop a max, the top is replaced by an item from the last/k-th level, and this item may end up zig-zagging back down to the k-th level. (For the last level $2^h h$ is not the exact value and the table above is a bit misleading?!)

The total work is the sum for all levels:

$$\begin{aligned}
\sum_{k=0}^{h} 2^k k &= 2 + (h-1)2^{h+1} && \text{(CS201-like formula)} \\
&\approx 2 + h2^h && \text{using } h \approx h-1 \approx h+1 \\
&\approx 2 + \log_2 n \cdot n && \text{using } h \approx \log_2 n,\ a^{\log_a b} = b \\
&\approx n \log_2 n
\end{aligned}$$

This shows that `Heap-Sort` is $O(n \log n)$. This time many nodes (bottom levels) may end up doing a lot of work, and few nodes (at the top) do a little work, so the amount of work doesn't balance out.

The messier calculations:

$$\begin{aligned}
\sum_{k=0}^{h} 2^k k &= 2 + (h-1)2^{h+1} && \text{(CS201-like formula)} \\
&= 2 + (\log_2(n+1) - 2) \cdot (n+1) && \text{using } h = \log_2(n+1) - 1,\ a^{\log_a b} = b \\
&= (n+1)\log_2(n+1) - 2n && \approx n \log_2 n
\end{aligned}$$

For the final accurate result need to subtract h (one extra was included):

$$(n+1)\log_2(n+1) - 2n - (\log_2(n+1) - 1) = n\log_2(n+1) - 2n + 1 \approx n \log_2 n$$

Examples of Recursive Methods

Adding the values of the nodes in a binary tree:

```
procedure ADD(root):
    if root is nil:
        return 0
    else:
        s1 = ADD(left[root])
        s2 = ADD(right[root])

        return data[root] + s1 + s2
```

Calculating the height of the tree (for empty tree defined height to be 0):

```
procedure HEIGHT(root):
    if root is nil:
        return 0
    else:
        h1 = HEIGHT(left[root])
        h2 = HEIGHT(right[root])

        return 1 + MAX(h1, h2)
```