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*
-
```
      perfect                       complete                        full

           *                            *                             *
     *           *                 *          *                  *          *
  *     *     *     *           *     *    *     *                       *     *
 * *   * *   * *   * *         * *   *                                  * *
```

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:

```
level     # nodes
-----    ---------
  0        1 = 2^0                 *
  1        2 = 2^1           *           *
  2        4 = 2^2        *     *     *     *
  3        8 = 2^3       * *   * *   * *   * *
  .        . . . .      .....             .....
  k            2^k
  .
  h            2^h    * * *   the leaves    * * *
```

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:

$$l_h = 2^h = 2^{\log_2(n+1)-1} = 2^{\log_2(n+1)}/2^1 = (n+1)/2$$
$$(\text{using } a^{b-c} = a^b/a^c \text{ and } a^{\log_a b} = b)$$

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.

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)
```