Binary Trees

A *binary tree* is a tree in which every node has at most two children.

A *binary search tree* is 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 sub-tree are greater* than the node's value.

A *leaf node* is a node that has no children.

A *internal node* is a node that is not a leaf node (i.e. has at least one child).

The *height* of a node is the longest path (i.e. number of levels) from the node to a leaf.

The *height* of a binary tree is the height of the *root node* (i.e. total number of levels).

A *complete binary tree* is binary tree in which:

- all *internal nodes* have *exactly* two children
- all *leaves* are at the *same level*
- (the above that all *leaves* are on the last level and the last level is completely full)

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

We showed this for the special type of binary tree called *complete binary tree* that is defined above.

Let n be the number of nodes in a *complete binary tree* and let l_k denote the umber of nodes on level k, where the levels are numbered 1, 2, 3, ..., h. The last level, h represents the *height* of the tree.

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)
- $l_1 = 1$, i.e. on the "first level" we have only one node (the root node).
- from CS201 the previous recurrence solves to $l_k = 2^{k-1}$

Note also that the leaves are at the last level, l_h, where h is the height of the tree, so from the previous bullets we know that the last level has $l_h = 2^{h-1}$ nodes.

The total number of nodes, n, in the tree is equal to the sum of the nodes on all the levels:

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

From CS 201 we know that:

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

Therefore:

$$2^h - 1 = n$$

$$2^h = n + 1$$

$$\log_2 2^h = \log_2(n+1)$$

$$h \log_2 2 = \log_2(n+1)$$

$$h = \log_2(n+1)$$

Therefore 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 $l_h = 2^{h-1}$ so we can substitute the value of h in this expressions:

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

In the above expression we used the fact that $a^{b-c} = a^b/a^c$ and $a^{\log_a b} = b$.

In summary, in a *complete binary tree* with n nodes:

- the *height* is $h = \log_2(n+1)$, 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.

Height of a Red-Black Tree is $O(\log n)$

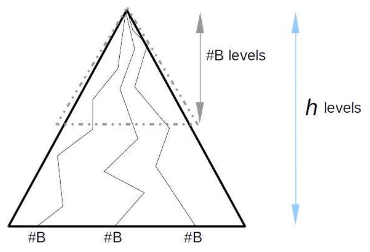
We showed that the height, h, of a *Red-Black Tree* is $O(\log n)$. The proof was based on the following claims:

- Claim 1: A *complete binary tree* of X levels has $2^X - 1$ nodes. (Proof is in above discussion about height of *complete binary tree*.)
- Claim 2: $h \geq \#B \geq h/2$

  The first part, $h \geq \#B$, is due to the fact that the RB-Tree needs to have at least $\#B$ levels if every possible path has at least $\#B$ black nodes. But the total height can be more, since some paths might have Red nodes.

  The second part, $\#B \geq h/2$, is due to the fact that the longest possible path in RB-Tree has alternating Black and Red nodes starting with Black (the root). Thus, $\#B = \#R$ on longest possible path, i.e. longest possible path has $\#B + \#R = 2\#B$ nodes. Since the tree cannot be taller than longest possible path, we have $h \leq 2\#B$ or $h/2 \leq \#B$.

The key point was to find a *complete binary tree* inside a given RB-Tree.



Given RB-Tree of n nodes, h levels, and $\#B$ Black nodes on each possible path, consider the tree formed by the top $\#B$ levels (the dashed triangle in the figure).

We claim that the dashed triangle is a *complete binary tree* of $\#B$ levels. In other words, there are no gaps inside the dashed triangle and any path of $\#B$ zig-zag steps from the root will take us to the dashed/bottom line of that triangle. If we assume that there is a *short path*, i.e. if we fell off the dashed tree before making $\#B$ zig-zag steps, this would mean that we could not have seen $\#B$ Black nodes on that *short path*, which is not possible if the RB-Tree is built correctly.

(Note that we do not claim that the dashed triangle has only Black nodes. We just claim that the dashed triangle is completely packed with nodes.)

Since the dashed triangle of $\#B$ levels is a *complete binary tree*, we can use Claim 1 to establish that it has $2^{\#B} - 1$ nodes.

Now we use the second part of Claim 2, i.e. $\#B \geq h/2$ where h is the height of the given RB-Tree that has $\#B$ nodes on each path:

$$\#B \geq h/2$$

$$2^{\#B} \geq 2^{h/2}$$

$$2^{\#B} - 1 \geq 2^{h/2} - 1$$

The left side is the total number of nodes in the dashed triangle. Since the dashed triangle is part of the given RB-Tree, we have:

$$n \geq 2^{\#B} - 1 \geq 2^{h/2} - 1$$

Focusing on the two ends we get:

$$n \geq 2^{h/2} - 1$$

$$n + 1 \geq 2^{h/2}$$

$$\log_2(n+1) \geq \log_2 2^{h/2}$$

$$\log_2(n+1) \geq \frac{h}{2} \log_2 2$$

$$\log_2(n+1) \geq \frac{h}{2}$$

$$2\log_2(n+1) \geq h$$

or

$$h \leq 2\log_2(n+1)$$

Therefore, h is $O(\log n)$.

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)
```