

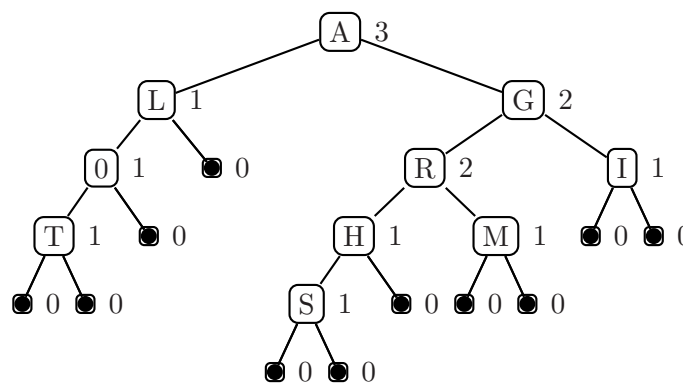
## Leftist Heaps

*Note:* This material on leftist trees will eventually be integrated into a revised Handout #32 on priority queues.

### Leftist Trees

Let the **rank**<sup>1</sup> of a binary tree be the length of its right spine – i.e., the length of the rightmost path from the root to a leaf.

*Example:* Nodes in the following tree are annotated with their ranks:



Call a binary tree **leftist** iff it satisfies the following **leftist condition** for every subtree  $t$  of the tree:  $\text{rank}(\text{left}(t)) \geq \text{rank}(\text{right}(t))$ .

In the above example, the subtrees rooted at L and G (and, necessarily, all of their subtrees) are leftist, but the subtree rooted at A is not leftist.

It is not difficult to show the following facts. Let  $n$  be the number of nodes in a binary tree and  $r$  be its rank.

1.  $n \geq 2^r - 1$ . Show this by induction:

2.  $r \leq \lg(n + 1)$ .

<sup>1</sup>The term “rank” is often used as a name for some relevant property of a data structure. Exactly which property it refers to depends on the particular data structure and what is trying to be proven.

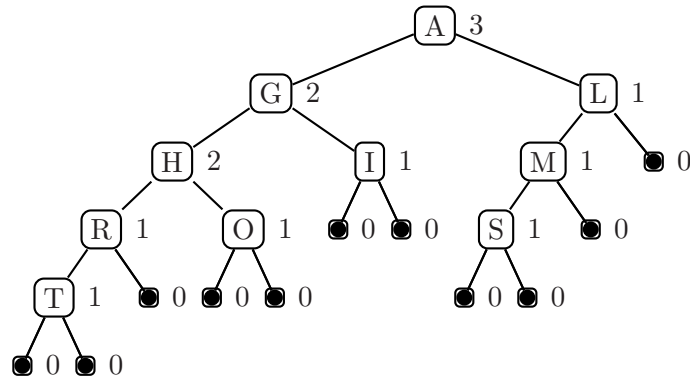
---

## Leftist Heaps

Recall that a binary tree is a **heap** if it satisfies the following **heap condition** at every subtree  $t$ :  $\text{value}(t)$  has a higher priority than all values in  $\text{left}(t)$  and all values in  $\text{right}(t)$ .

A **leftist heap** is simply a binary tree that is both leftist and a heap.

*Example:* The following is one of the many leftist heaps for the letters in the word **ALGORITHMS** (assuming letters earlier in the alphabet have a higher priority than later ones).



---

## Leftist Trees in Haskell

Here is a datatype for manipulating leftist trees in Haskell:

```
data LH a =
  Leaf
  | Node Int    -- rank
          (LH a) -- left subtree
          a      -- node value
          (LH a) -- right subtree

-- Return the rank of a tree
rank Leaf = 0
rank (Node k _ _ _) = k
```

---

## Merging Leftist Trees

The core of most leftist tree operations is the following `merge` function. It is similar to the `merge` function used to merge sorted lists in mergesort:

```
-- Merge leftist trees t1 and t2 into one leftist tree.
merge t Leaf = t
merge Leaf t = t
merge (t1 @ (Node k1 l1 v1 r1))
      (t2 @ (Node k2 l2 v2 r2))
  | v1 >= v2 = make l1 v1 (merge r1 t2)
  | otherwise = make l2 v2 (merge t1 r2)
where make t1 v t2
      | r1 >= r2 = Node (r2 + 1) t1 v t2
      | otherwise = Node (r1 + 1) t2 v t1
      where r1 = rank(t1)
            r2 = rank(t2)
```

*Example:*

Given leftist heaps of size  $m$  and  $n$ , what is the running time of `merge`?

---

## Other Operations

```
-- Make a leftist heap with one node.
singleton x = Node 1 Leaf x Leaf

-- Insert an element into a leftist heap.
insert x t = merge (singleton x) t

-- Return maximum element of heap.
maxElt Leaf = error "maxElt of Leaf"
maxElt (Node _ _ v _) = v

-- Return pair of:
-- (1) maximum element of heap
-- (2) heap without maximum element.
deleteMax Leaf = error "deleteMax of Leaf"
deleteMax (Node _ l v r) = (v, merge l r)

-- Create a leftist heap from a list of values.
fromList [] = Leaf
fromList (x:xs) = insert x (fromList xs)

-- Return a list (sorted high priority to low) of elements in heap.
toList Leaf = []
toList t = v:(toList t')
  where (v,t') = deleteMax t
```

What are the running times of the above operations?

---

## A Cleverer fromList

The above version of `fromList` is not as efficient as it could be. As with building a complete heap, a leftist heap can be constructed in linear time if build from “the bottom up”:

```
-- Linear time construction of a leftist heap from a list of elements
fromList [] = Leaf
fromList xs = mergeLoop (map singleton xs)
  where
    mergeLoop [] = error "shouldnt: mergeLoop []"
    mergeLoop [t] = t
    mergeLoop ts = mergeLoop(mergePairs ts)

    mergePairs [] = []
    mergePairs [t] = [t]
    mergePairs (t1:t2:ts) = (merge t1 t2):(mergePairs ts)
```