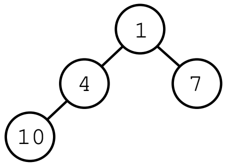
Heap (Priority Queue)

A *heap* (priority queue) supports these operations:

- insert keys
- retrieve and delete the minimum key

We expect each operation to take at most logarithmic time.

Heaps are often implemented by *heap-ordered trees*: binary trees with every parent less than its children.



Leftist Heap

A *leftist heap* is a heap-ordered tree with the *leftist property*:

> The "rank" of a left child is greater than or equal to that of its right sibling.

Generally, there are various definitions of "rank", all based on the size of the subtree. We will use this definition:

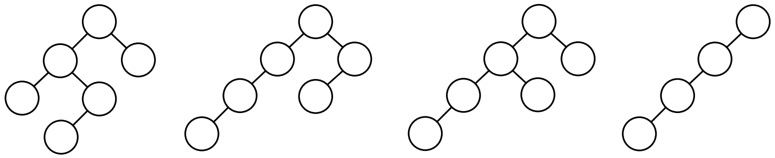
> The length of its right spine, i.e., the number of internal nodes on the rightmost path from it to Nil.
>
> Note: the rank of Nil is 0.

The idea is that there are more nodes on the left; thus the name *leftist*.

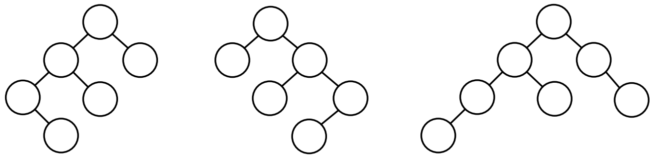
In fact, the right spine is the shortest path, and it contains at most $\lfloor \lg(n+1) \rfloor$ nodes, where n is the number of nodes in the tree.

Leftist Heap Examples

The following are leftist heaps:



The following are not leftist heaps:



Leftist Heap Implementation

Declaration: (the `Int` field stores the rank)

```
data Ord a => LHeap a = Nil | Node Int a (LHeap a) (LHeap a)
```

Retrieving the minimum key is trivial:

```
findMin (Node _ k _ _) = k
```

Next, assume that we can merge two leftist heaps in logarithmic time. Then to delete the minimum, we just merge its two child subtrees:

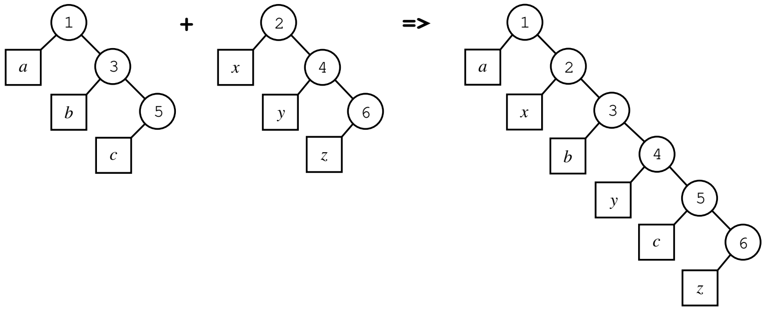
```
deleteMin (Node _ _ t1 t2) = merge t1 t2
```

Inserting a key is, in principle, just merging the heap with the new node:

```
insert d h = merge h (Node 1 d Nil Nil)
```

Leftist Heap Merging

Merge two leftist heaps as follows: since the right spines are sorted and short, we can simply merge the two right spines.



But this may break the leftist property, so we will swap left and right children wherever necessary (and update ranks) as we go back up.

Leftist Heap Merging Code

So `merge` can be written as:

```
merge Nil h = h
merge h Nil = h
merge h1@(Node _ x a1 b1) h2@(Node _ y a2 b2)
  | x<=y      = leftist x a1 (merge b1 h2)
  | otherwise = leftist y a2 (merge h1 b2)
```

where `leftist` constructs a node with rank and leftist property done right:

```
leftist k a@(Node ra _ _ _) b@(Node rb _ _ _)
  | ra>=rb    = Node (rb+1) k a b
  | otherwise = Node (ra+1) k b a
```


Queue

A queue supports first-in first-out operations:

- add keys to the end
- get and delete keys from the front

Queues were a weak point of functional programming: it is expensive to append naïvely. We will describe:

- the classical two-list queue (ephemeral, amortized $O(1)$)
- the lazy two-list queue (persistent, amortized $O(1)$)
- the lazy, scheduled two-list queue (persistent, worst-case $O(1)$)

Ephemeral vs Persistent

A data structure could be *ephemeral* or *persistent*:

ephemeral destructive updates; the old version is gone or unusable

persistent immutable, non-destructive updates; you can keep both the old and the new versions

E.g., compare concatenation in imperative and functional programming:

- Imperative: the "last" pointer of the front list is modified, so the original front list is lost. This is ephemeral.
- Functional: the front list is copied and prepended to the back list, so the original front list is still available. This is persistent.

Classical Two-List Queue

Removing keys from the front is trivial, but how do we append keys quickly?

Idea: have two lists, one ready for removal as usual, the other to keep newly added keys. E.g.,

1. start with this queue:
   `[3,1,4], [9,5,1]`
2. take the 3 from the front:
   `[1,4], [9,5,1]`
3. add 2 to the back:
   `[1,4], [2,9,5,1]`

So we "append" by prepending to the back list.

When the front list is exhausted, we reverse the back list and make it the front list.

Two-List Queue Complexity

Each operation on the two-list queue takes $O(1)$ amortized time:

- Adding a key is $O(1)$.
- Removing a key is $O(1)$ except when the front list becomes empty.
- If we need to reverse the back list, it takes as many steps as its length.

  But: there must have been equally many additions before this happens, so the cost of the reversal can be amortized over those additions.

In other words, n operations take only $O(n)$ steps altogether, even though some take more.

Two-List Queue Implementation

Declaration:

```
data Queue a = Q [a] [a]
```

Adding to the end:

```
snoc d (Q [] e) = Q [d] e  --e supposedly []
snoc d (Q fs bs) = Q fs (d:bs)
```

(We keep the front list non-empty unless the back list is also empty.)

Getting and removing the head:

```
hd (Q (x:_) _) = x
tl (Q (x:fs) bs) = case fs of [] -> Q (reverse bs) []
                              _  -> Q fs bs
```


(blank slide)