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

**Leftist Heap**

Insert a node in a leftist heap:

1. Insert the new node as a leaf node.
2. Merge the new node with its parent.
3. Repeat step 2 until the new node is in the heap.

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

1. Determine (node - 1) / 2 = merge 1.
2. Determine (node - 1) / 2 = merge 1:
3. Determine the minimum key is trivial.
4. There are <= leaves = node | Node: at least two leaf keys in logarithmic time. Then:
5. Reversing the minimum key is trivial.
6. Define: (the Izzy field stores the rank)

**Leftist Heap Implementation**

Node: The rank of N = 0.

The leftmost path from N to Nil.

The length of its right spine, i.e., the number of internal nodes on the spine. We will use this definition:

- The length of a key spine, e.g., the number of internal nodes on the spine.

Generally, there are various definitions of "rank," all based on the size of siblings.

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

The leftist heap is a heap-ordered tree with the leftist property.

The following are not leftist heaps:

The following are leftist heaps:

Leftist Heap Examples

Heap (Priority Queue)
original front list is still available. This is persistent.

Functional: the front list is copied and prepended to the back list, so the
original front list is lost. This is ephemeral.

Imperative: the "next" pointer of the front list is modifed, so the original
permanent

ephemeral
destructive updates: the old version is gone or unstable
persistent
destructive updates: you can keep both the old
ephemeral vs persistent

A data structure could be ephemeral or persistent:

Ephemeral vs Persistent

Queue

| a | a -> Node (i, b) | a -> Node (i, b) |
| a | a -> Node (i, b) |

Test test $a$ node with rank and leftist property done right:

$\text{left}$ $a$ node $b$ node $c$ node $d$ node $e$ node $f$ node $g$

$\text{merge}$ $h$ node $i$ node $j$ node $k$

So $\text{merge}$ can be written as:

Least-Heap Merging Code

Least-Heap Merging

The lazy scheduled two-list queue (persistent, worst-case $O(1)$)

The lazy two-list queue (persistent, amortized $O(1)$)

The classical two-list queue (ephemeral, amortized $O(1)$)

Append naïvely. We will describe

Queues were a weak point of functional programming: it is expensive to
get and delete keys from the front

and keys to the end

A queue supports first-in-first-out operations:

Queue

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

Short, we can simply merge the two right piles:

Merge two leftist heaps as follows: since the right piles are sorted and
some take more.
In other words, in operations take only $O(u)$ steps altogether, even though
so the cost of the reversal can be amortized over those additions. But: there must have been equally many additions before this happens. If we need to reverse the back list, it takes as many steps as its length.

Removing a key is $O(1)$ except when the front list becomes empty.
Adding a key is $O(1)$.

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

Two-list Queue Complexity

Front List
When the front list is exhausted, we reverse the back list and make it the
so we „append“ by prepending to the back list.

[1.4], [2.5], [3.6]
3. add 2 to the back.
[1.4], [2.5], [3.6]
2. Take the 3 from the front.
[3.4], [2.5], [3.6]
1. start with this queue.

Added keys: E, G.
Removed keys: E.

Two-list Queue