

- This data structure implements the priority queue ADT. While max binary heaps support melding of two max binary heaps in $O(n)$ time, as we see below, the leftist heap melds a pair of leftist heaps in $o(n)$ time. The time complexities of all other functions of leftist heaps are same as for the binary heaps.
- For any rooted binary tree T' with a key stored at each of its nodes, for the convenience of analysis, we attach a set S' of nodes, called external nodes (dummy nodes), to T' so that the resulting rooted binary tree T satisfies the following two properties: each internal node of T has exactly two children, and the set comprising leafs of T is equal to S' . We assume every tree that we consider here is a rooted binary tree with external nodes.
- For any two nodes v', v'' of a tree T , the distance between v' and v'' is the number of edges on the unique path between v' and v'' in T . For any node v of T , among all the leafs in T_v , let $\ell \in T_v$ be the leaf that is at a minimum distance from v . Then the distance between v and ℓ is denoted by $s(v)$.

For every external node v of T , $s(v) = 0$.

For every internal node v of T , $s(v) = 1 + \min(s(v.left), s(v.right))$.

- A rooted binary tree is a leftist tree whenever $\forall_{v \in T} s(v.left) \geq s(v.right)$. This is known as the leftist tree property. Due to this property, these leftist trees are height-biased. Specifically, for every node v of T , a leaf at minimum distance from v occurs on the right spine of T_v .
- Let v be any internal node of a leftist tree. Then, the number of nodes n' in T_v is $\geq 2^{s(v)} - 1$.
(proof: $n' \geq 2^0 + 2^1 + \dots + (s(x)$ terms))

Since $n' \geq 2^{s(v)} - 1$, significantly, $s(v) \leq \lg_2(n' + 1)$.

- A leftist max-heap is a max heap-ordered leftist tree, that is, it is a leftist tree and the tree is max heap-ordered with respect to keys stored at its internal nodes.

Every node v of a leftist (max-)heap stores $s(v)$ with it.

- Given two leftist heaps H_1 and H_2 , the following algorithm merges their right spines for building a max heap-ordered binary tree and then transforms it into a leftist tree:

- (1) Without loss of generality, let $H_1.root.key$ is the maximum among $H_1.root.key$ and $H_2.root.key$. Then, $H_1.root$ as the root r , $H_1.root.left$ as the left subtree of r , and $meld(H_1.root.right, H_2.root)$ as the right subtree of r is a heap ordered binary tree T .
- (2) While traversing T upwards to root from the last node at which $meld$ is last invoked, for every node v that occurs along this path,
 - if $s(v.left) < s(v.right)$, exchange $v.left$ and $v.right$ so that $v.left$ is the right child of v and $v.right$ as the left child of v
 - update $s(v)$.

Essentially, the $meld$ operation performs a top-to-bottom pass as the recursion unfolds and then a bottom-to-top pass in which subtrees are possibly swapped and s -values are updated.

Let H be the output produced by the above algorithm. We prove the correctness of this algorithm by inducting on the number of nodes in H . The left subtree of root of H is a leftist heap since it is a subtree of leftist heap H_1 . From the induction hypothesis, $\text{meld}(H_1.\text{root.right}, H_2.\text{root})$ is a leftist heap. Due to comparison, $H.\text{root.key}$ is the largest among all the keys in H . Hence, H is heap-ordered. To transform this heap-ordered tree into a leftist tree, if $s(H.\text{root.left}) > s(H.\text{root.right})$, Step (2) exchanges $H.\text{root.left}$ and $H.\text{root.right}$, and $s(H.\text{root})$ is updated.

The length of the right spine formed while traversing root-to-leaf path is $\lg(n_1) + \lg(n_2)$, where n_1 and n_2 are the number of nodes in H_1 and H_2 respectively. Hence, it is $O(\lg n)$. While traversing downwards (Step (1)), at every node, excluding the time for recursive processing, $O(1)$ time is spent in comparing two keys. The Step (2) involves exchanging left and right subtrees at select nodes along the merged right spine, each such exchange takes $O(1)$ time. Therefore, melding a pair of leftist heaps takes $O(\lg n)$ time, where $n = n_1 + n_2$.

- $\text{insert}(H, x)$: initialize a heap H' with node x ; meld H and H'
takes $O(\lg n)$ time
- $\text{maximum}(H)$: takes $O(1)$ time since max key is stored at the root
- $\text{extractMax}(H)$: return $\text{meld}(\text{root.left}, \text{root.right})$
takes $O(\lg n)$ time
- Given a set S of n keys, build a leftist heap comprising these keys.

First, initialize n leftist heaps, each with a unique key from S . The obvious approach of melding the first two of these heaps, melding the resultant with the third heap, etc., is correct but leads to a time complexity, $\lg 2 + \dots + \lg n$, which is $\Theta(n \lg n)$. In an alternative approach, place these n leftist heaps on a queue Q . And, then,

```
while(there exist more than one heap in Q)
    dequeue a pair of heaps from Q, meld them, and enqueue the resultant heap into Q
```

The correctness of this algorithm is immediate from the correctness of algorithm for melding a pair of leftist heaps.

For the convenience of analysis, assume n is a power of two. After melding $\frac{n}{2}$ pairs of leftist heaps, there are $\frac{n}{2}$ leftist heaps, each with two nodes; after melding $\frac{n}{4}$ pairs of leftist heaps, there are $\frac{n}{4}$ leftist heaps, each with four nodes; after melding $\frac{n}{8}$ pairs of leftist heaps, there are $\frac{n}{8}$ leftist heaps, each with eight nodes; etc. Hence, the time complexity is $O(\frac{n}{2} \lg(2) + \frac{n}{4} \lg(4) + \frac{n}{8} \lg(8) + \dots + (\lg n + 1) \text{ terms})$, which is $O(\frac{n}{2} + \frac{n}{4}(2) + \frac{n}{8}(3) + \dots) = O(n \sum_{i=1}^{\infty} \frac{1}{2^i}) = O(n)$.

- $\text{delete}(H, x)$: replace the subtree rooted at x with the leftist heap returned by $\text{meld}(x.\text{left}, x.\text{right})$; update s values on the path from $x.\text{parent}$ to the root and swap subtrees on this path as necessary to maintain the leftist tree property

Since the s value updation pass halts as soon as the algorithm encounters a node whose s value does not change, since the maximum s value is $O(\lg n)$, and since the modified s values form an ascending sequence (incremented by one along that subpath), there are $O(\lg n)$ nodes encountered in updating s values.

At every node v along this subpath, algorithm takes $O(1)$ time to swap subtrees (if required) and to update the s value of v . Hence, the deletion algorithm takes $O(\lg n)$ time in the worst-case.

- `increaseKey(H, x, k): delete(H, x); insert(H, k)`

takes $O(\lg n)$ time

- This data structure has a couple of disadvantages: It is not self-adjusting, as each node v needs to save $s(v)$. The meld operation performs two traversals of a root to leaf path (instead of one).