Meldable Priority Queues

Kim Skak Larsen

Department of Mathematics and Computer Science
University of Southern Denmark

kslarsen@imada.sdu.dk

Advanced Algorithms (DM582)
May 14, 2024

About These Slides

Disclaimer

These slides contain much more text than I usually put on slides.

The reason is that no good text exists for this material at this level. So, the slides should replace a textbook.

Thus, the slides will be less suited for lecturing.

Priority Queues

A priority queue is a data type for a collection of elements, each of which has an associated priority.

The minimal set of operations provided for a priority queue are the following:

`q = PriorityQueue()`: Initializes an empty priority queue.

`q.insert(e, p)`: Inserts the element `e` with priority `p` into `q`.

`q.findMin()`: Returns the element of highest priority (traditionally indicated by smallest value) in `q`.

`q.deleteMin()`: Deletes and returns the element of highest priority from `q`.

A priority queue may have additional operations such as `decreaseKey`, `meld`, and others.

Priority Queues

The most well-known implementation of the priority queue data type is the *binary heap* data structure.

A binary heap provides `findMin` in $O(1)$ time and `insert` and `deleteMin` in $O(\log n)$, where n is the number of elements in the priority queue when the operation is carried out.

We will be interested in the operation `meld`.

`meld(q, p)`: Returns a new priority queue containing all the elements from `q` and `p` (destructive).

The standard binary heap implementation cannot provide an efficient implementation of this operation.

Leftist Heaps [Crane, Stanford, 1972]

A leftist heap is implemented as an *annotated binary tree*.

Each node contains an element with a priority (we just show the priority of the element) and a rank.

The tree is *heap-ordered*, i.e., the priority of a node is at most the priority of its children.

The *rank* is defined as the distance to *nil*[1] in the following sense:

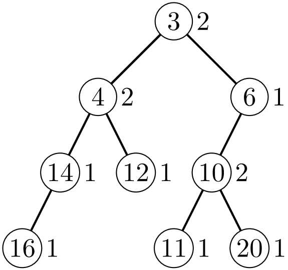
Think of a nil reference a reference to a special node with rank zero. Then a node containing a nil reference has rank one. Other nodes have rank one plus the minimum of the ranks of its children.

The tree *leftist*, which we define to mean that for any node `u`, we have

$$\texttt{u.left().rank()} \geq \texttt{u.right().rank()}$$

[1] Or *None*, *null*, or some other name for an initialized missing reference.

Example Leftist Heap



Melding Two Leftist Heaps

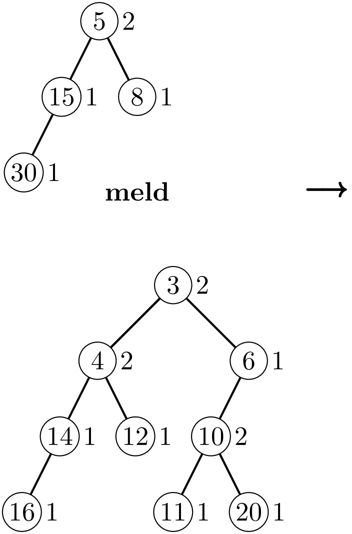
We carry out a meld as follows:

1. Merge the right-most paths of the two argument heaps according to the priorities via their right child references.
2. Adjust the ranks bottom-up on the right-most path in the result.
3. Switch the children of nodes on the right-most path if the leftist requirement is violated.

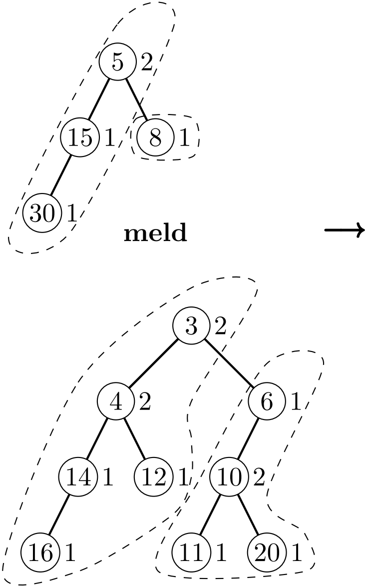
The result is clearly a leftist heap.

It takes time proportional to the sum of the lengths of the arguments right-most paths.

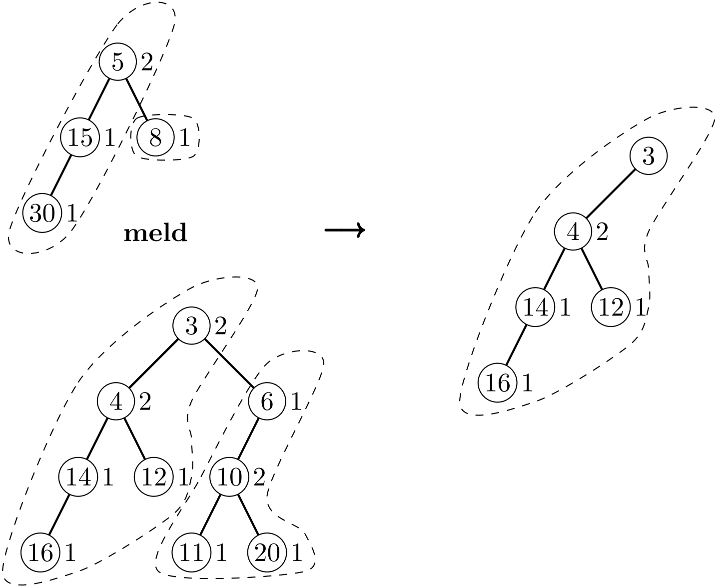
Leftist Heaps



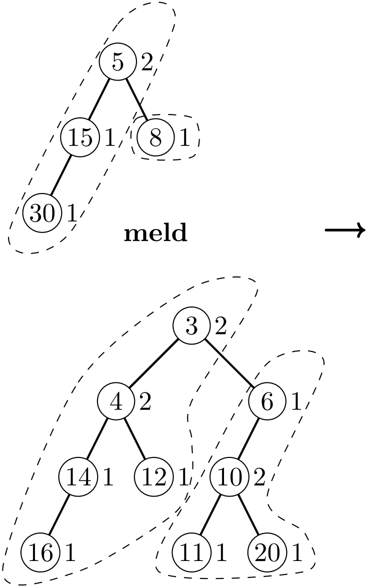
Leftist Heaps

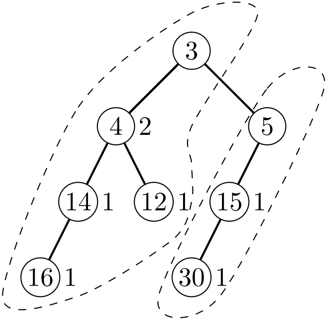


Leftist Heaps

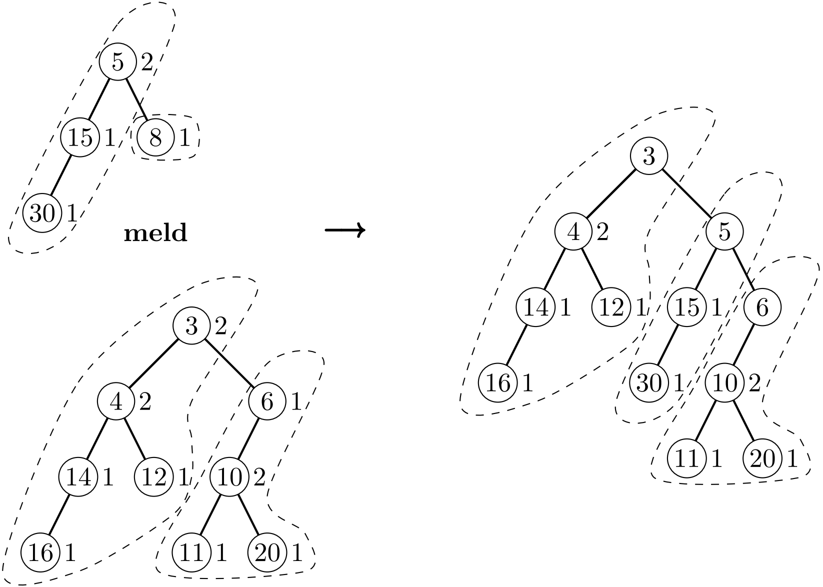


Leftist Heaps

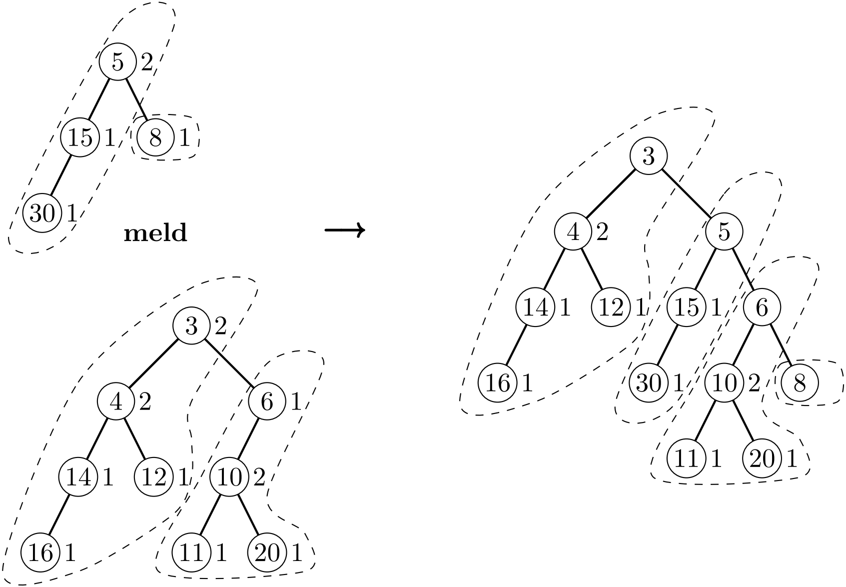




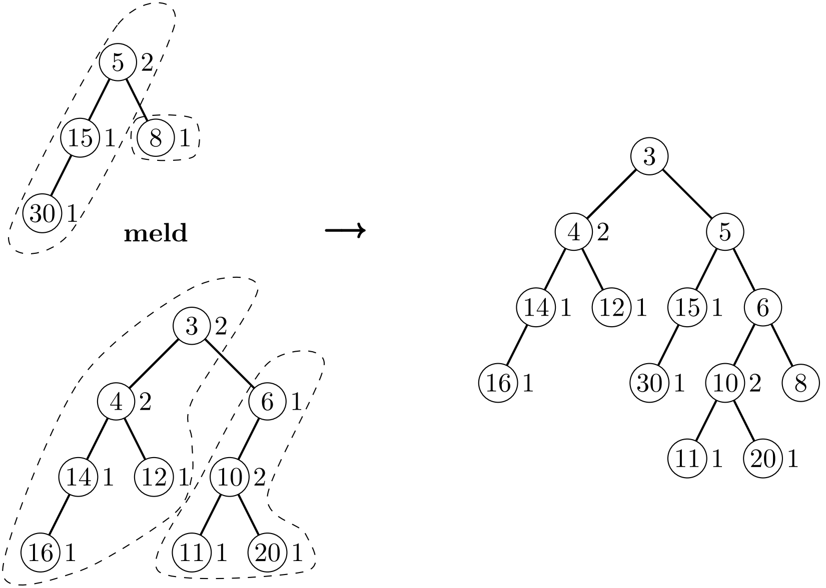
Leftist Heaps



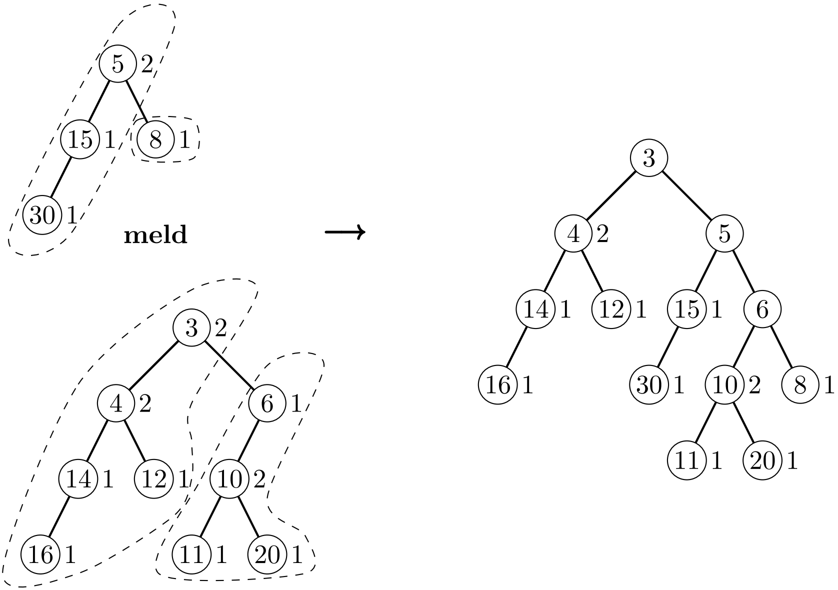
Leftist Heaps



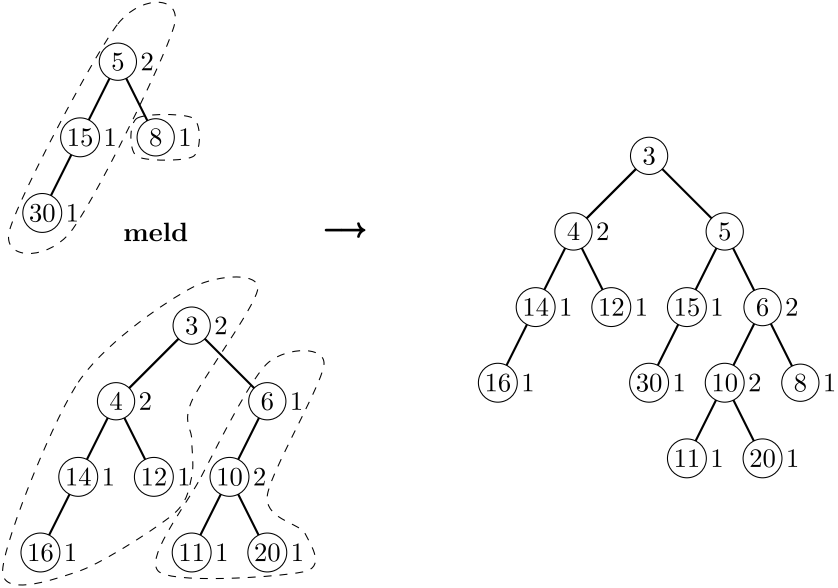
Leftist Heaps



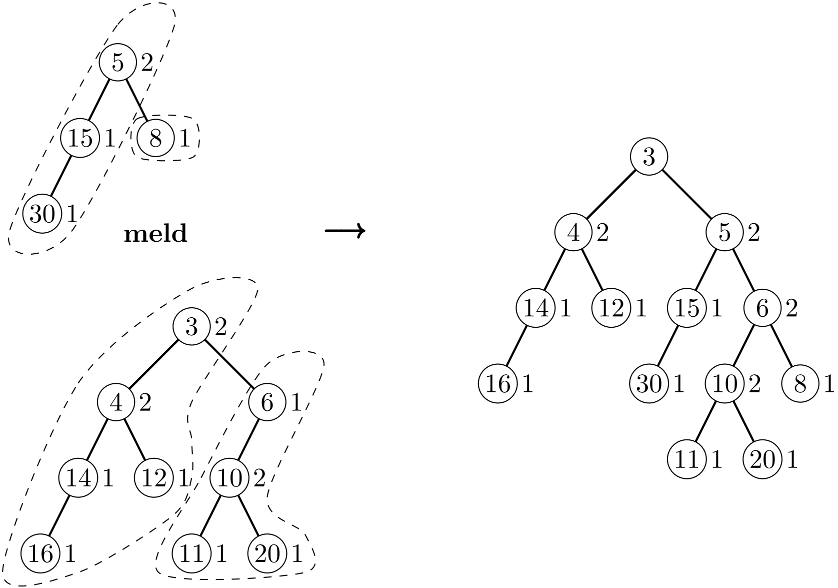
Leftist Heaps



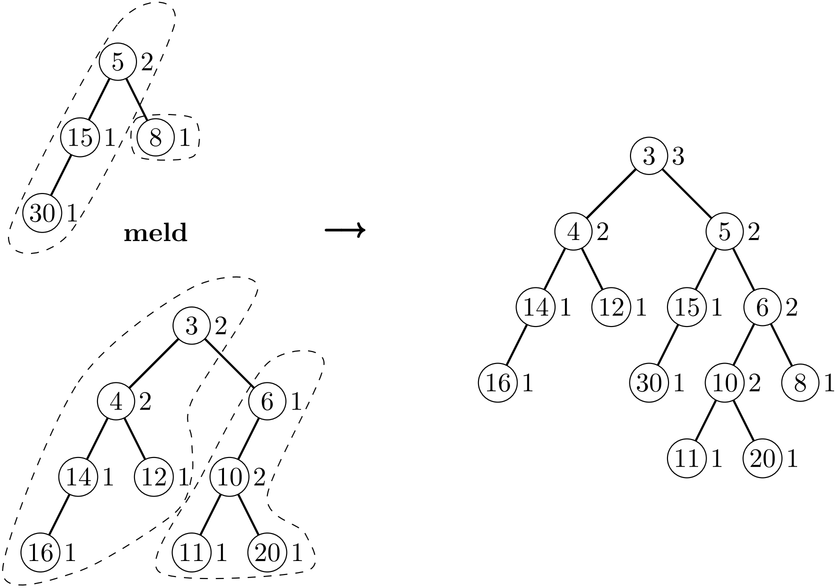
Leftist Heaps



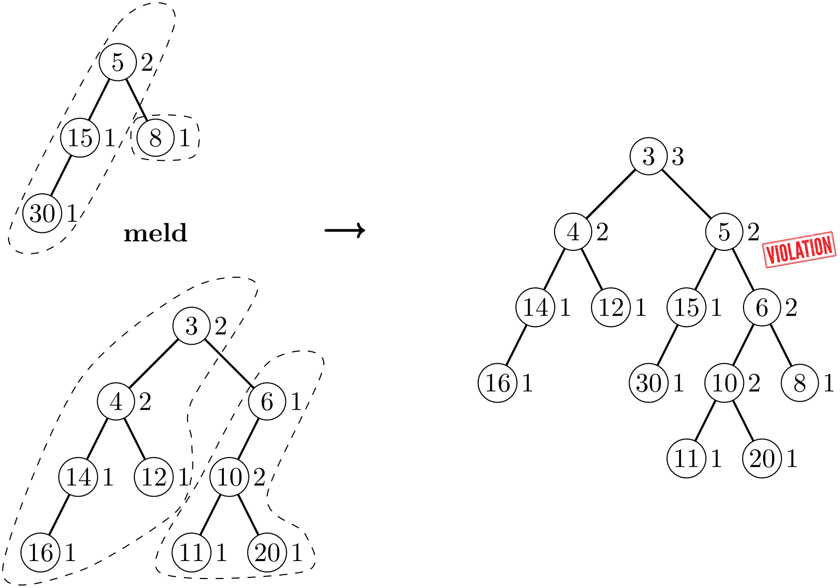
Leftist Heaps



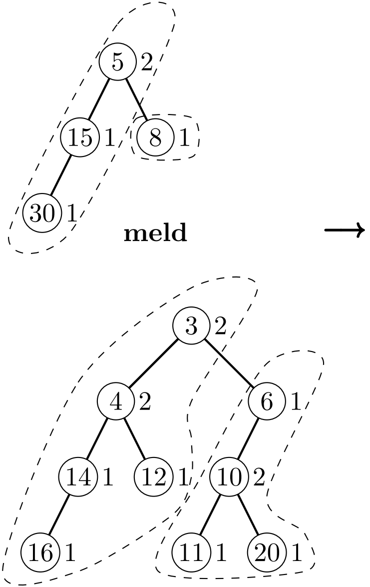
Leftist Heaps

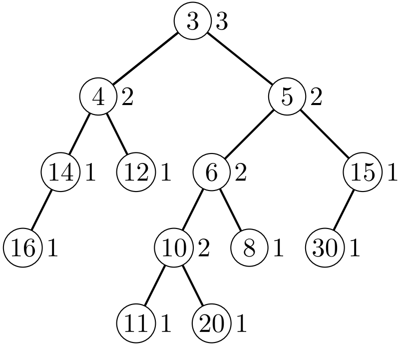


Leftist Heaps



Leftist Heaps





Leftist Heap Complexity

SDU

Lemma

In a leftist tree, the subtree of a node with rank r contains at least $2^r - 1$ nodes.

Proof By structural induction. For the base case, a node with no children has rank 1 and its subtree contains $2^1 - 1 = 1$ nodes. For the induction step, a node cannot have rank r unless both of its children have rank at least $r - 1$. By induction, its subtree has at least $2(2^{r-1} - 1) + 1 = 2^r - 1$ nodes. □

Corollary

The maximal rank of the root of a leftist heap with n elements is $\log(n + 1)$.

Proof Let r be the rank of the root. By the above lemma, $n \geq 2^r - 1$, so $r \leq \log(n + 1)$. □

Leftist Heap Complexity

SDU

Theorem

A `meld` of two leftist heaps with n_1 and n_2 elements takes time $O(\log n)$, where $n = n_1 + n_2$.

Proof For any node of rank r with left and right children ranks of r_l and r_r, since $r_l \geq r_r$ (the leftist property), $r = r_r + 1$. Thus, there are exactly r nodes on the right-most path of a root with rank r.

The time to meld the two heaps is proportional to the sum of the lengths of the two right-most paths, which amounts to at most

$$\log(n_1 + 1) + \log(n_2 + 1) \leq 2 \log(\max\{n_1, n_2\} + 1) \leq 2 \log n.$$

□

Leftist Heap Operations

Operations other than `meld` are either trivial or can be reduced to `meld`, so we get the following results are corollaries.

`q = PriorityQueue()`: Clearly $O(1)$.

`q.insert(e, p)`: Make singleton heap and meld with `q` in $O(\log n)$.

`q.findMin()`: Clearly $O(1)$.

`q.deleteMin()`: Remove the root, meld its two children in $O(\log n)$.

`q = buildHeap(elements)`: Notice that the shape of a classic heap makes it a leftist heap that we can annotate with ranks in linear time and get this operation in $O(n)$.

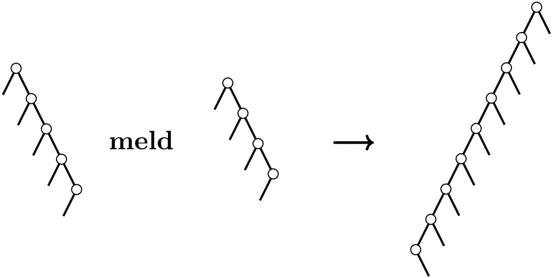
Skew Heaps [Sleator & Tarjan]

We try to do as well or better with less information!

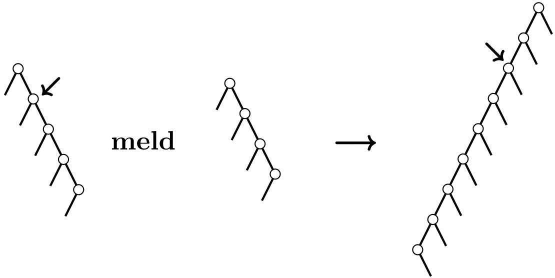
A skew heap is mostly the same as a leftist heap, but we do not keep any rank information. Instead, after merging the right-most paths according to priorities, we switch the subtrees of every node on that path!

So, the two right-most paths become one left-most path.

Skew Heaps Example



Skew Heaps Example

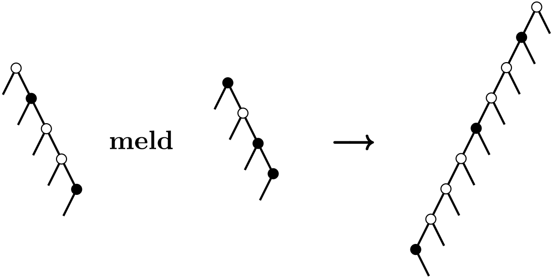


Skew Heaps Analysis

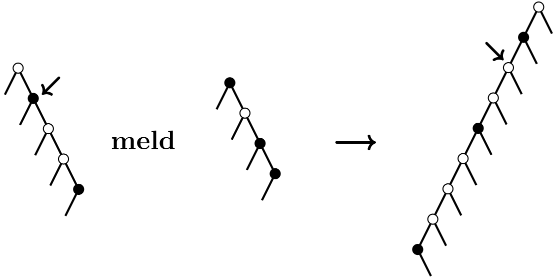
A node is *heavy* (•) if its right subtree contains more nodes than its left subtree. Otherwise, it is called *light* (◦).

During the merge and the switches, nodes on the right-most paths before the `meld` can change status from heavy to light or light to heavy.

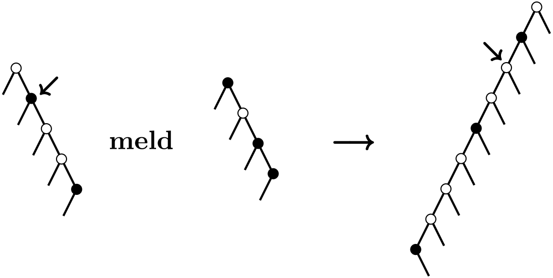
Skew Heaps



Skew Heaps



Skew Heaps

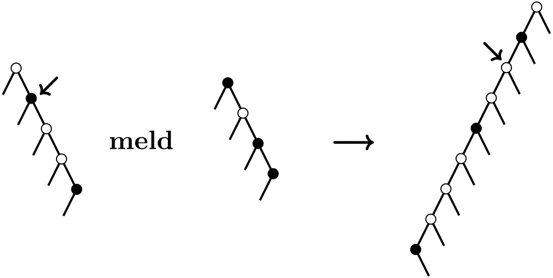


During a merge, a heavy node may be come even heavier!

So, when we switch the subtrees, it will definitely become light.

We do not know if a light node changes status or not.

Skew Heaps



During a merge, a heavy node may be come even heavier!

So, when we switch the subtrees, it will definitely become light.

We do not know if a light node changes status or not.

heavy	→	light
light	→	?

Skew Heaps

Lemma

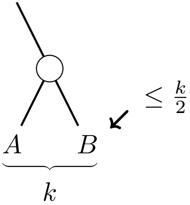
There are at most $\log n$ light nodes on the right-most path of a skew heap.

Proof A heavy node would have $|B| > |A|$.

But it is light, so $|B| \leq |A|$.

Thus, traversing the right-most path from root to leaf, considering the number of nodes in A and B, we always move towards the subtree with at most half of the nodes.

This can only happen $\log n$ times.



□

Skew Heaps

SDU

Theorem

For skew heaps, `meld` is $O_A(\log n)$ (amortized $O(\log n)$).

Proof Let l_i and h_i denote the number of light and heavy nodes, respectively, on the right-most path of argument i, $i \in \{1, 2\}$.

As for leftist heaps, the cost of `meld` is $(l_1 + h_1) + (l_2 + h_2)$.

Define the potential function $\Phi(T)$ to be the number of heavy nodes in T. This is initially zero and always non-negative, so results are valid.

In the worst case, all the light nodes become heavy so we need to pay into the potential for them.

Operation	Cost	$\Delta\Phi$	Amortized Cost
`meld`	$(l_1 + h_1) + (l_2 + h_2)$	$-h_1 - h_2 + l_1 + l_2$	$2(l_1 + l_2)$

The result follow by the lemma. □

Skew Heaps

As for leftist heaps, all the other operations follow.

`q = PriorityQueue()`: Clearly $O(1)$.

`q.insert(e, p)`: Make singleton heap and meld with `q` in $O_A(\log n)$.

`q.findMin()`: Clearly $O(1)$.

`q.deleteMin()`: Remove the root, meld its two children in $O_A(\log n)$.

`q = buildHeap(elements)`: Notice that the shape of a classic heap makes all nodes light, so we can perform the operation in $O(n)$ and the potential is zero, so the amortized results for the above operations hold.

References I

C. A. Crane.
Linear Lists and Priority Queues as Balanced Binary Trees.
Tech. report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

Daniel Dominic Sleator, Robert Endre Tarjan.
Self-Adjusting Binary Trees.
In *Proc. 15th Annual ACM Symp. on the Theory of Computing*, pages 235–245, 1983.