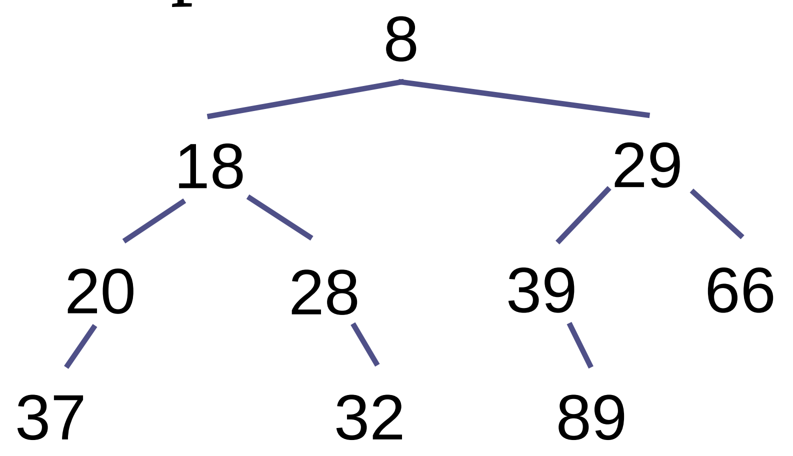
Skew heaps

Heaps with merging

Apart from adding and removing minimum element, another useful operation is *merging two heaps into one.*

To do this, let's go back to *binary trees with the heap property* (no completeness):



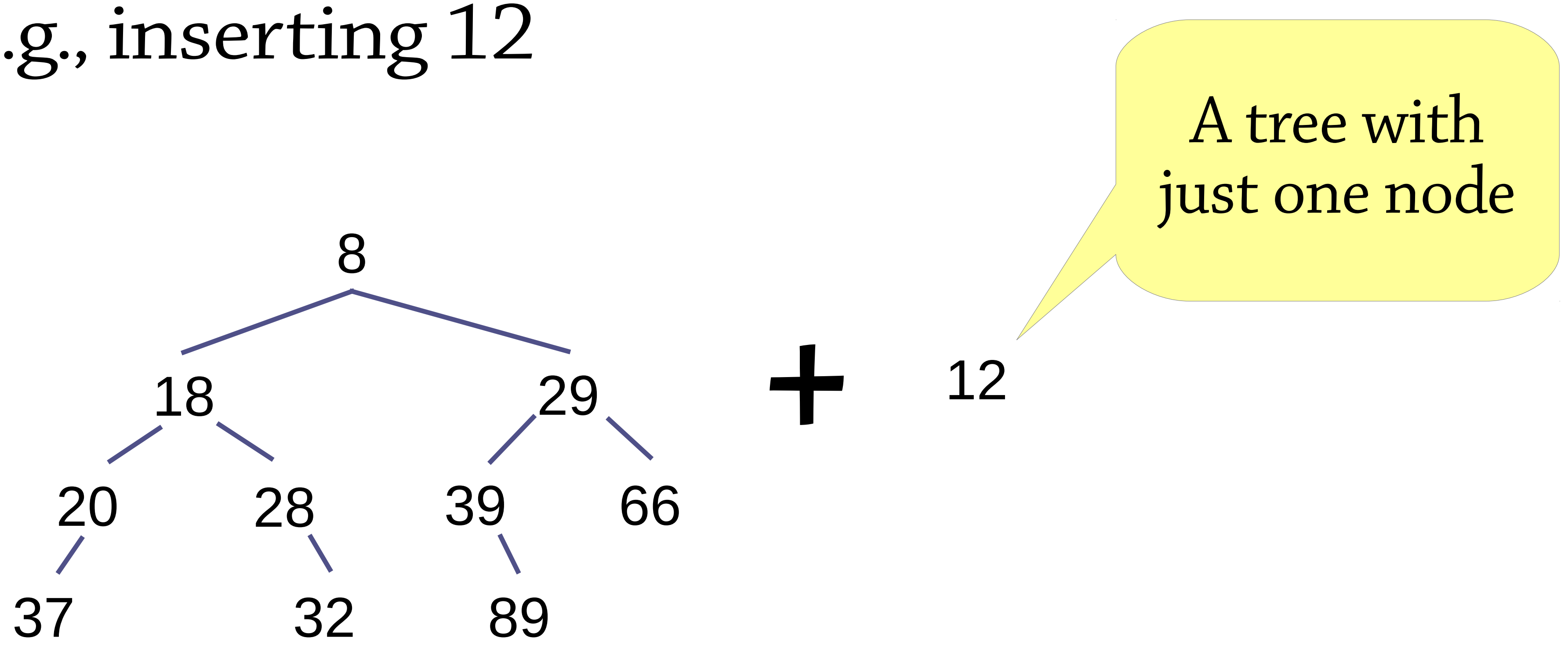
We can implement the other priority queue operations in terms of merging!

Insertion

To insert a single element:

- build a heap containing just that one element
- merge it into the existing heap!

E.g., inserting 12

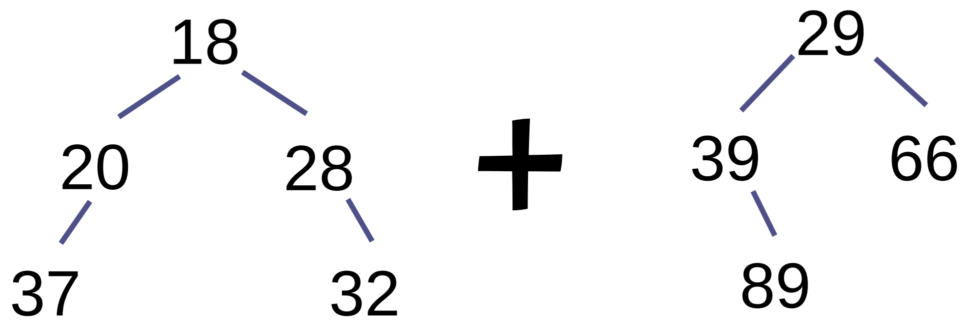


Delete minimum

To delete the minimum element:

- take the left and right branches of the tree
- these contain every element except the smallest
- merge them!

E.g., deleting 8 from the previous heap



Heaps with merging

Using merge, we can efficiently implement:

- insertion
- delete minimum

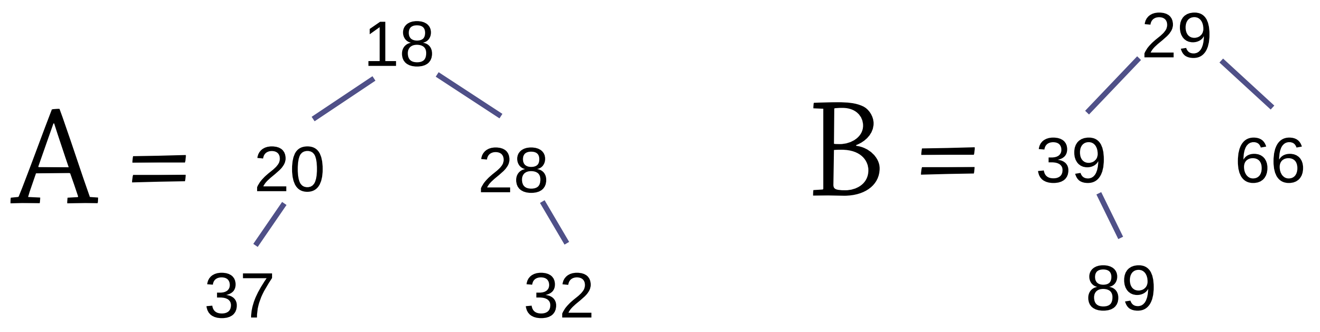
Only question is, how to implement merge?

- Should take O(log n) time

We'll start with a bad merge algorithm, and then fix it

Naive merging

How to merge these two heaps?



Idea: root of resulting heap must be 18

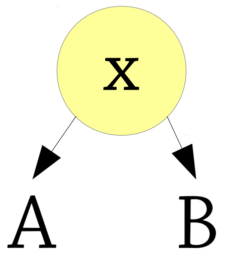
Take heap A, it has the smallest root.
Pick one of its children. Recursively merge B into that child.

Which child should we pick? Let's pick the right child for no particular reason

Naive merging

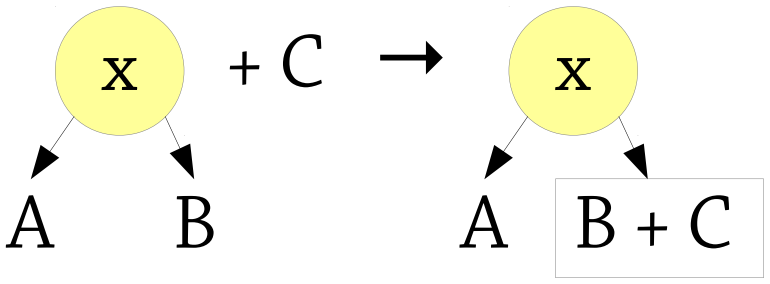
To merge two non-empty heaps:

Pick the heap with the smallest root:



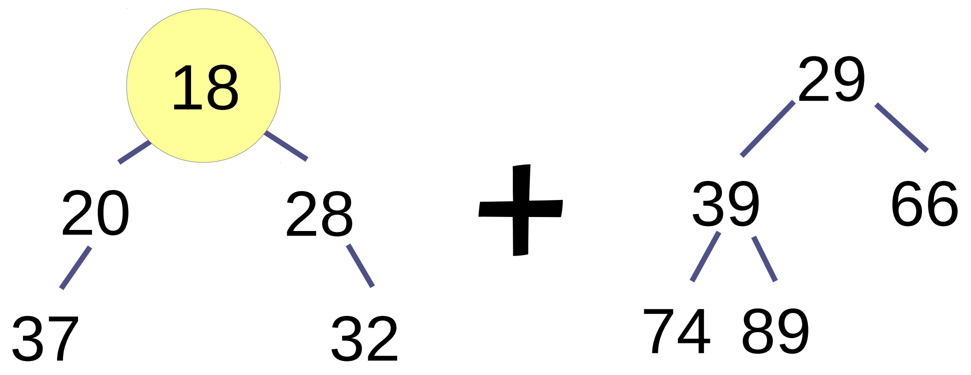
Let C be the other heap

Recursively merge B and C!



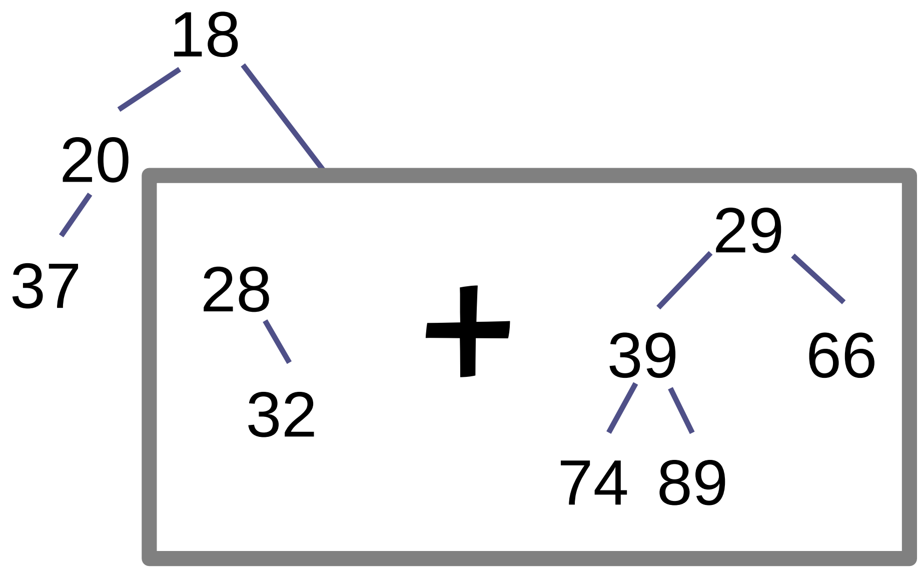
Example

18 < 29 so pick 18 as the root of the merged tree



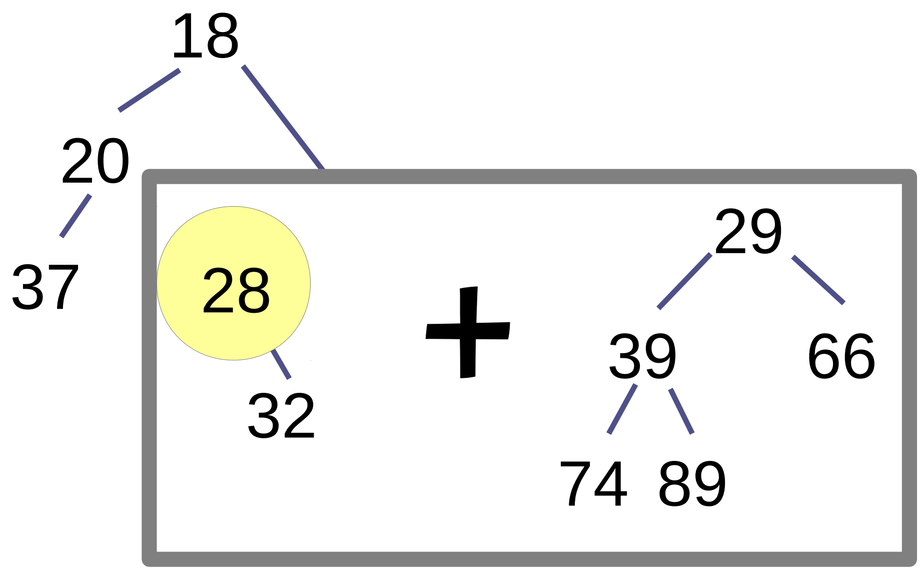
Naive merging

Recursively merge the right branch of 18 and the 29 tree



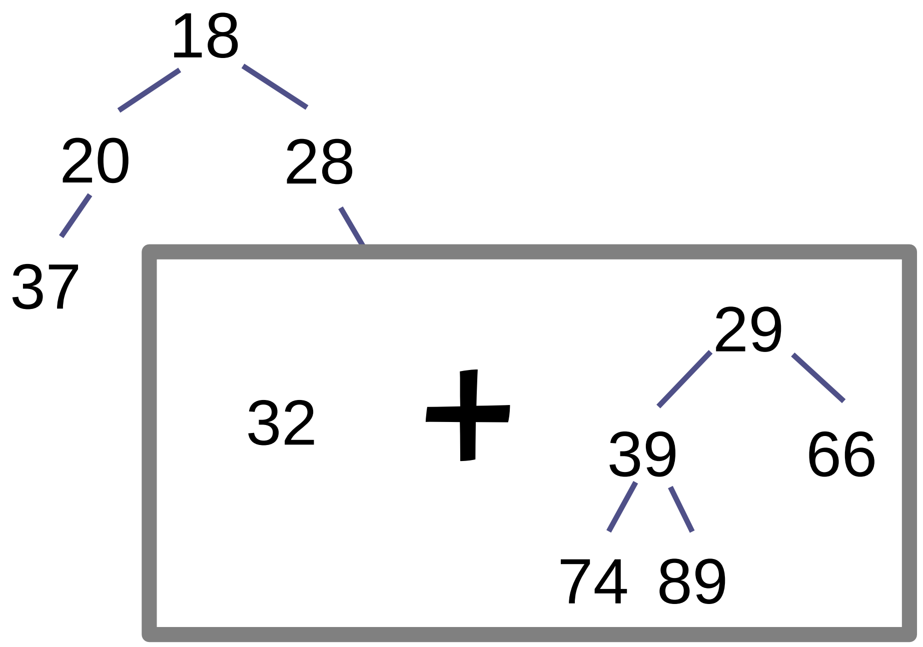
Naive merging

28 < 29 so pick 28 as the root of the merged tree



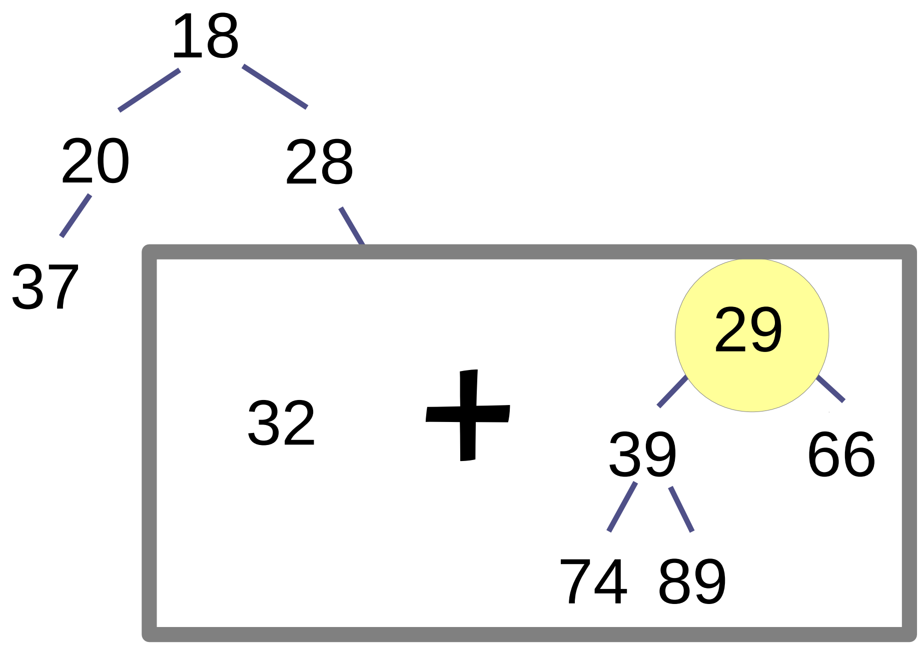
Naive merging

Recursively merge the right branch of 28 and the 29 tree



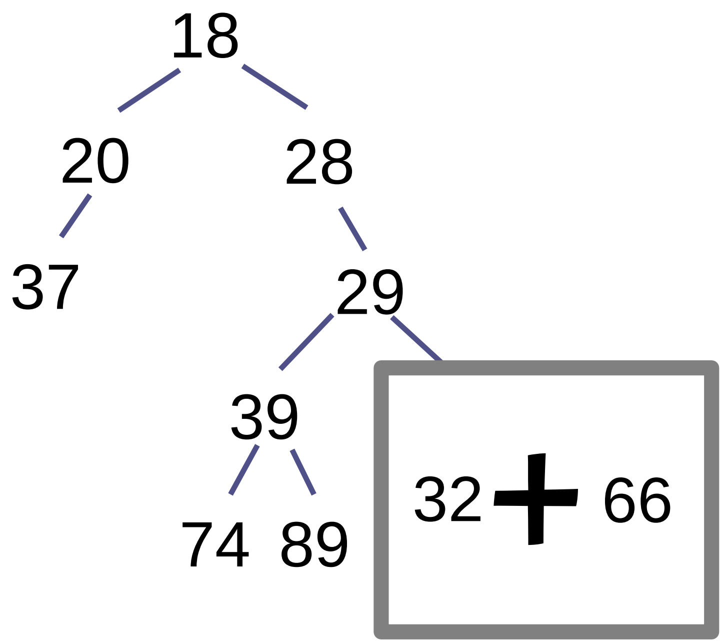
Naive merging

29 < 32 so pick 29 as the root of the merged tree



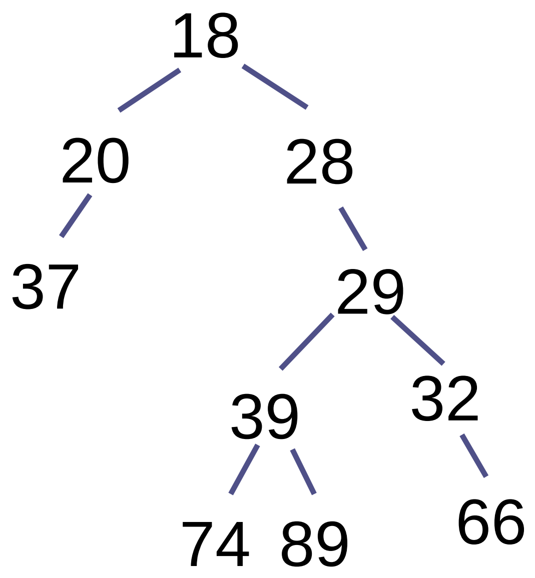
Naive merging

Recursively merge the right branch of 29 with 32



Naive merging

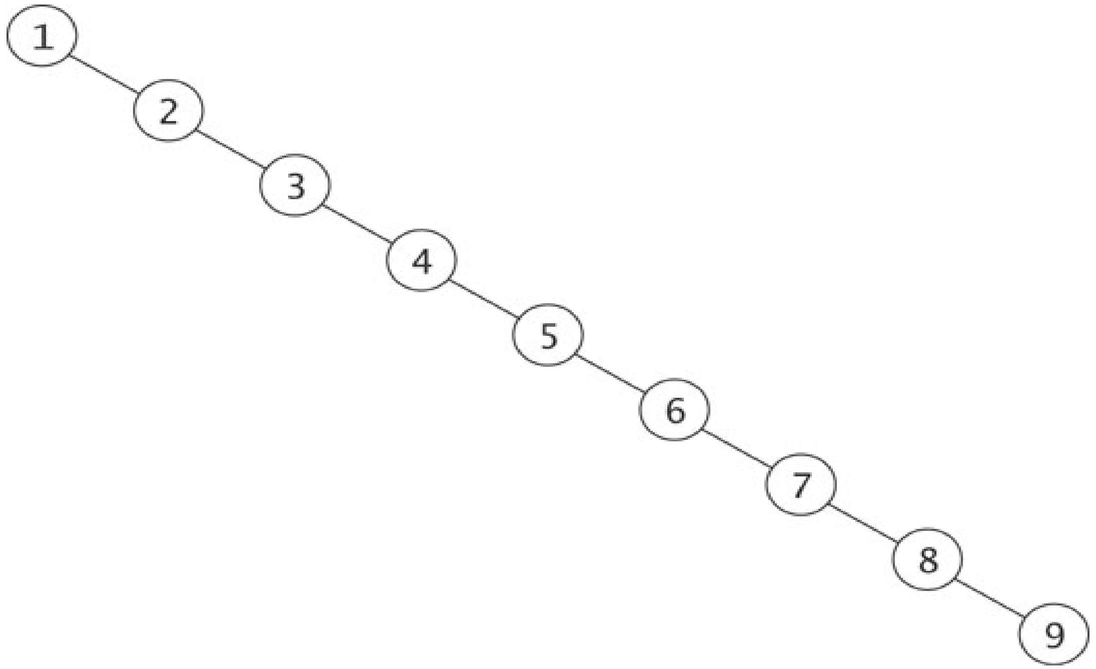
Base case: merge 66 with the empty tree



Notice that the tree looks pretty "right-heavy"

Worst case for naive merging

A right-heavy tree:

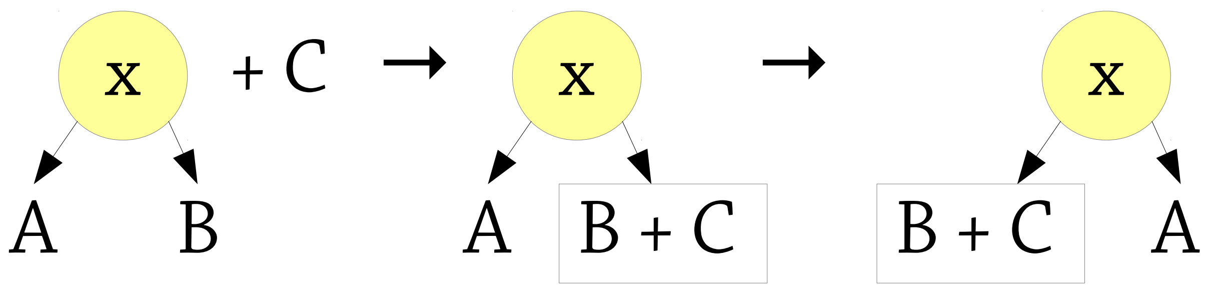


Unfortunately, you get this just by doing insertions! So insert takes O(n) time...

How can we stop the tree from becoming right-heavy?

Skew merging

In a skew heap, after making a recursive call to merge, we *swap the two children*:



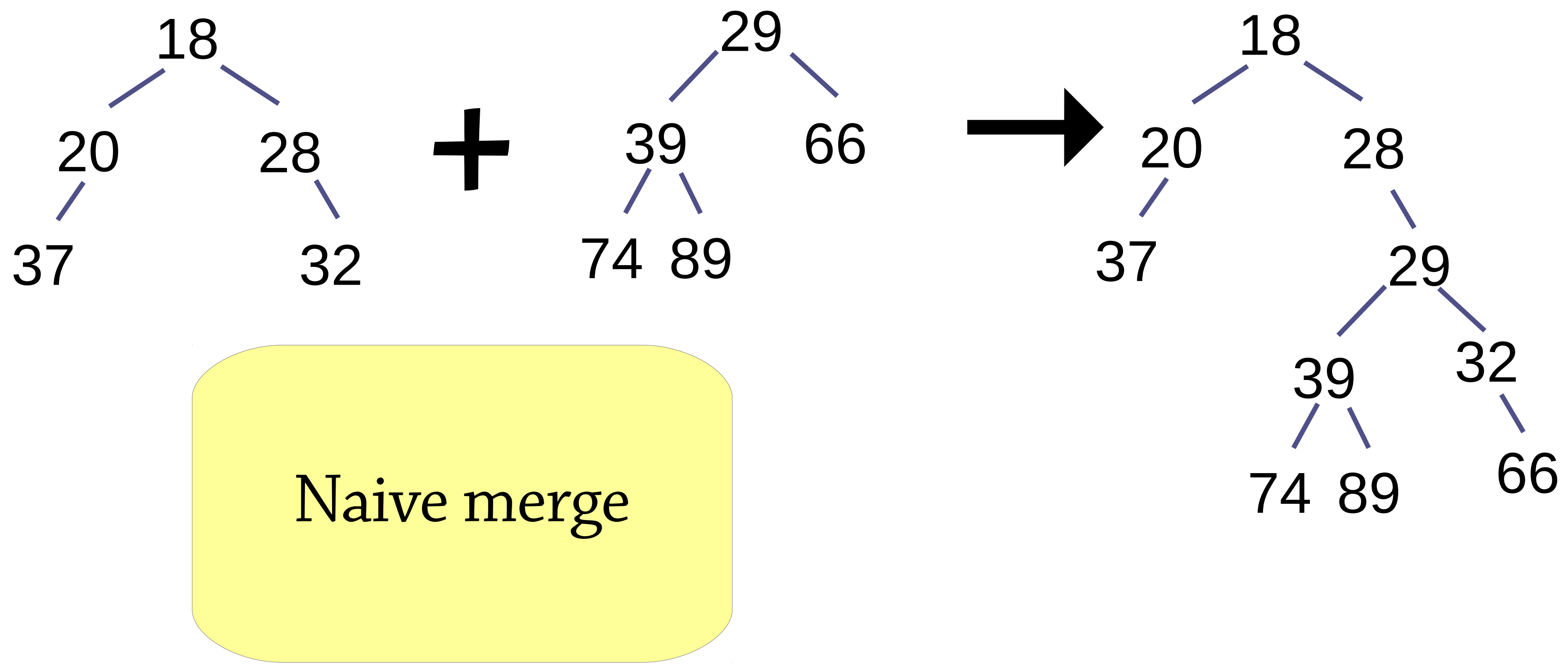
Amazingly, this small change completely fixes the performance of merge!

We almost never end up with right-heavy trees.

We get O(log n) amortised complexity.

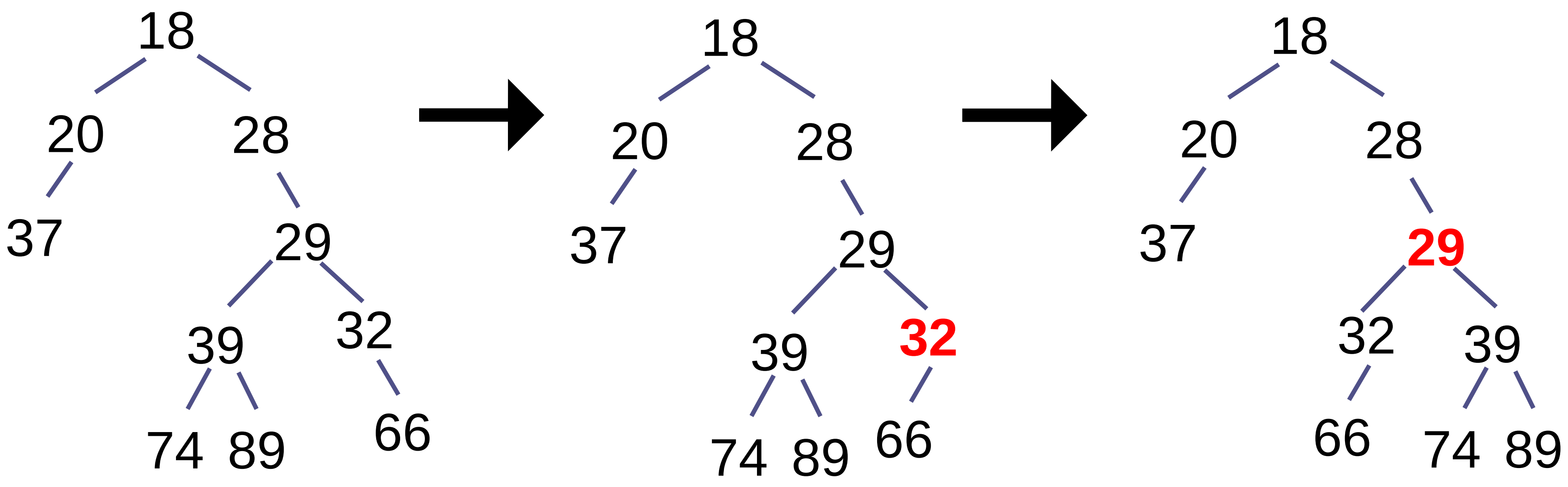
Example

One way to do skew merge is to first do naive merge, then go up the tree swapping left and right children...



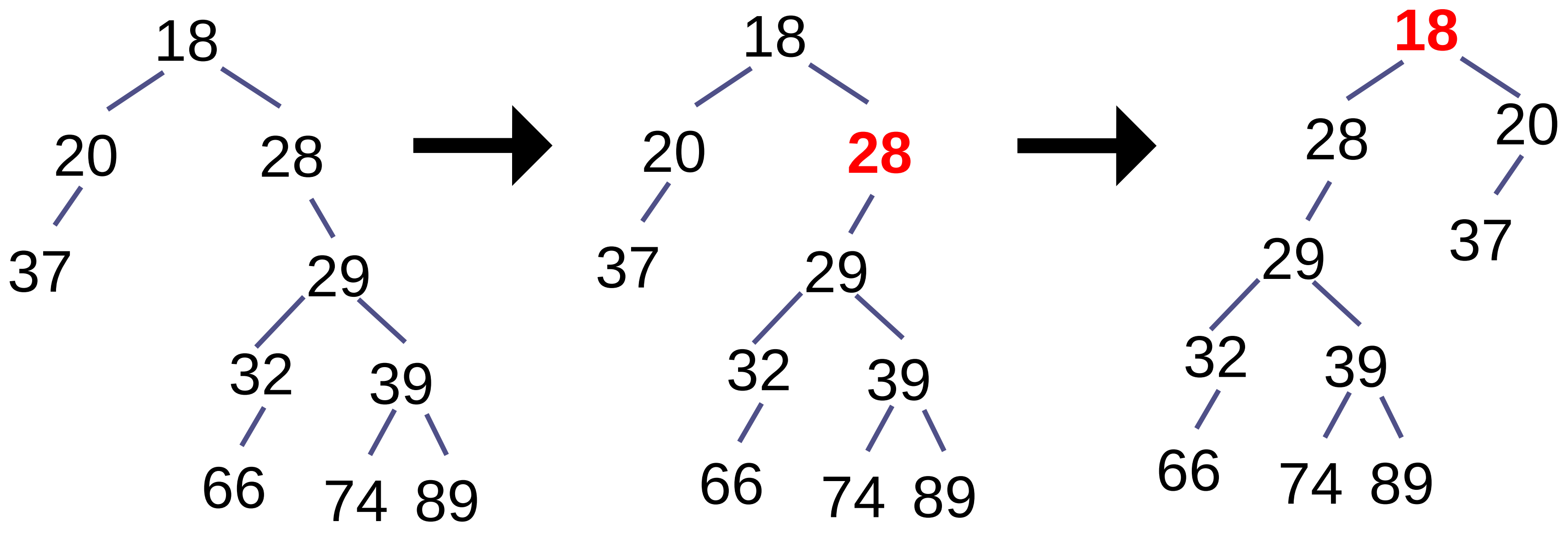
Example

...like this:



Example

...like this:



Skew heaps

Implementation of priority queues:

- binary trees with heap property
- skew merging avoids right-heavy trees, gives O(log n) amortised complexity
- other operations are based on merge

A good fit for functional languages:

- based on trees rather than arrays, tiny implementation!

Other data structures based on naive merging + avoiding right heavy trees:

- leftist heaps (swap children when needed)
- meldable heaps (swap children at random)

See webpage for link to visualisation site!

Skew heap merge operation has amortized logarithmic complexity (Not on exam)

- Amortized complexity: If any sequence of *k* operations has complexity O(*f*) (where *f* depends on *k* and parameters representing the size of data), then the *amortized* complexity of the operation is O(*f*/*k*)
- Example: for dynamic arrays, inserting at the end has best case complexity O(1) and worst case complexity O(*n*). However, *k* insertions has complexity O(*k*), so the amortized complexity is O(1)

Skew heap merge operation has amortized logarithmic complexity

- The *potential method* is one way to calculate amortized complexity.
- You define a potential function, Φ, that maps states of the data to a number.
- The potential can be seen as a bank account where you save money in good times (when execution is quick) and withdraw money in bad times (when execution is slow).
- Provided that the potential always is non-negative and the running time plus change in potential is $O(f(n))$, the amortized complexity is $O(f(n))$
- Example: Dynamic arrays. Let Φ be 0 for the empty list and increase by 2 for each addition of an element. When array grows, deduce 1 for each copy to new array. Let the running time, T, be the number of writes to the array. Then T is 1 each time the array does not grow. When the array grows T is n and Φ will decrease by $n - 3$. Φ will never be negative. (Check this.) $T + \Delta\Phi = 1 + 2 = 3$ when array doesn't grow and $n - (n - 3) = 3$ when array does grow. So, amortized complexity is $O(1)$

Skew heap merge operation has amortized logarithmic complexity

- Let's use the potential method to show that the skew heap merge operation has logarithmic amortized complexity.
- Let a node be called right heavy if the size of the right subtree is greater than the size of the left subtree.
- Define the potential, Φ, to be the number of right heavy nodes in the heap.
- By definition Φ is non-negative. So what we need to show is that $T + \Delta\Phi$ is $O(\log n)$ where n is the size of the heap.
- Note that the right spine (the rightmost path) of any binary tree contains at most $\log_2 n$ nodes which are *not* right heavy.

Skew heap merge operation has amortized logarithmic complexity

- The merge operation interleaves the right spines of the two heaps and makes the left subtrees right subtrees instead.
- Let's assume that the heaps have size n_1 and n_2 and that their right spines contain h_1 and h_2 right heavy nodes, respectively.
- Let's assume that the right spines of the heaps contain k_1 and k_2 nodes in total.
- We know that $k_1 \leq h_1 + \log_2 n_1$ and $k_2 \leq h_2 + \log_2 n_2$.
- Let T be $k_1 + k_2$ (for each node in the two spines the execution takes constant time)
- $T \leq h_1 + h_2 + \log_2 n_1 + \log_2 n_2$

Skew heap merge operation has amortized logarithmic complexity

- So what's the change in potential?
- Right spine nodes in the original heaps which were right heavy no longer are in the resulting heap. This is because the children are swapped and the left child will contain at least as many nodes as before (in the right child), while the right child will contain the same amount (as the left child did before).
- This transformation of right heavy nodes to non-right heavy nodes means a decrease of the potential by $h_1 + h_2$
- The non-right heavy nodes of the left spines might have become right heavy in the new heap.
- This transformation of some of the non-right heavy nodes to right heavy nodes means an increase of the potential by at most the total number of non-right heavy nodes in the two right spines, which is at most $\log_2 n_1 + \log_2 n_2$.

Skew heap merge operation has amortized logarithmic complexity

- To sum up $\Delta\Phi$ <= $-h_1 - h_2 + \log_2 n_1 + \log_2 n_2$ and we already concluded that $T \leq h_1 + h_2 + \log_2 n_1 + \log_2 n_2$
- So $T + \Delta\Phi \leq 2 \log_2 n_1 + 2 \log_2 n_2 \in O(\log n)$
- For a similar, more detailed explanation, see http://www.cse.yorku.ca/~andy/courses/4101/lecture-notes/LN5.pdf