Fibonacci Heaps

CS2040 AY2024/25 S1

Slides by Enzio Kam Hai Hong

Previously ... on heaps

Binomial heaps

- Fast operations
- Has a “nice” well-defined structure
- Faster *merge* operation compared to binary heap

However, the “nice” structure is also a disadvantage

- Very rigid structure, must always maintain the binomial trees
- Is costly to maintain - almost every operation is $O(\log n)$

What if we can make some operations **cheaper**?

Heaps

Procedure	**Binary Heap** (worst-case)	**Binomial Heap** (worst-case)	**Fibonacci Heap** (amortized)	**Leftist Heap** (worst-case)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Find-Min	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Merge	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$O(\log n)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$

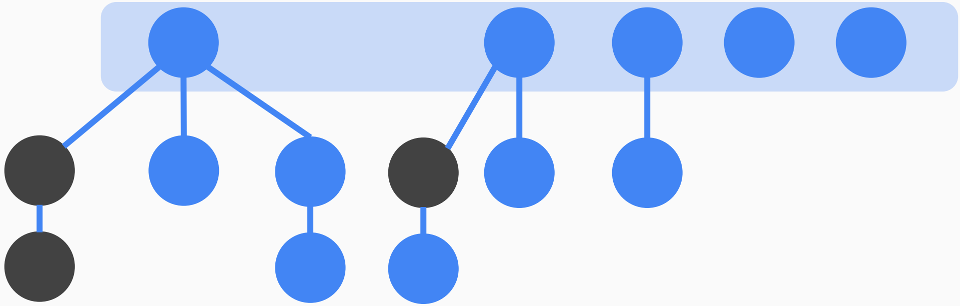
Why Fibonacci Heaps?

- Fast *insert* and *decrease-key* operations
- We can be **lazy** - work only when required
- No need to maintain binomial trees!

Note: The slides will not cover the asymptotic analysis of the operations in detail, which requires understanding of amortised analysis and potential functions. Please see CLRS Section 19 for the full analysis and detailed description of the algorithms.

Structure of a Fibonacci Heap

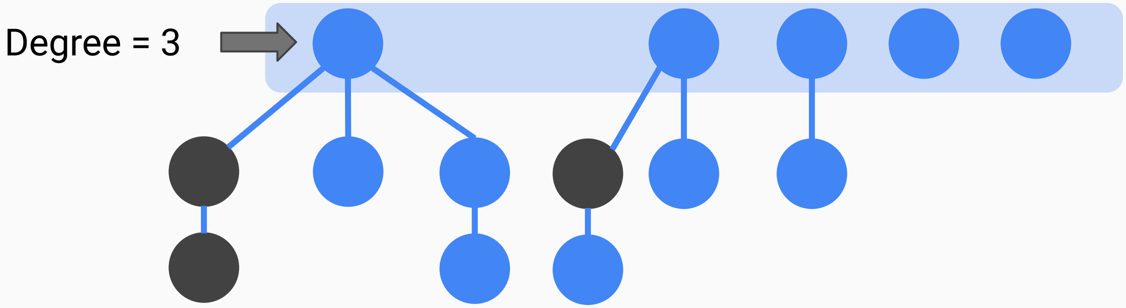
Fibonacci heaps, like binomial heaps, are made up of trees maintaining the **heap property**. But these trees need not be binomial trees!



Structure of a Fibonacci Heap

Each Fibonacci heap node has to store some information

- degree - number of child nodes (sometimes called rank)
- mark - whether a particular node is marked



This will be useful to us later.

Structure of a Fibonacci Heap

Let's also keep track of some useful information about the entire heap

- $t(H)$ - the number of trees in the fibonacci heap H
- $d(n)$ - the maximum degree in a fibonacci heap with n nodes
- $m(H)$ - the number of marked nodes in the fibonacci heap H

In fact, we can show that $d(n) \leq \lfloor \log_\phi n \rfloor = O(\log n)$! (CLRS Section 19.4)

* $\phi = (1 + \sqrt{5}) / 2$, the golden ratio.

It is time to be

LAZY

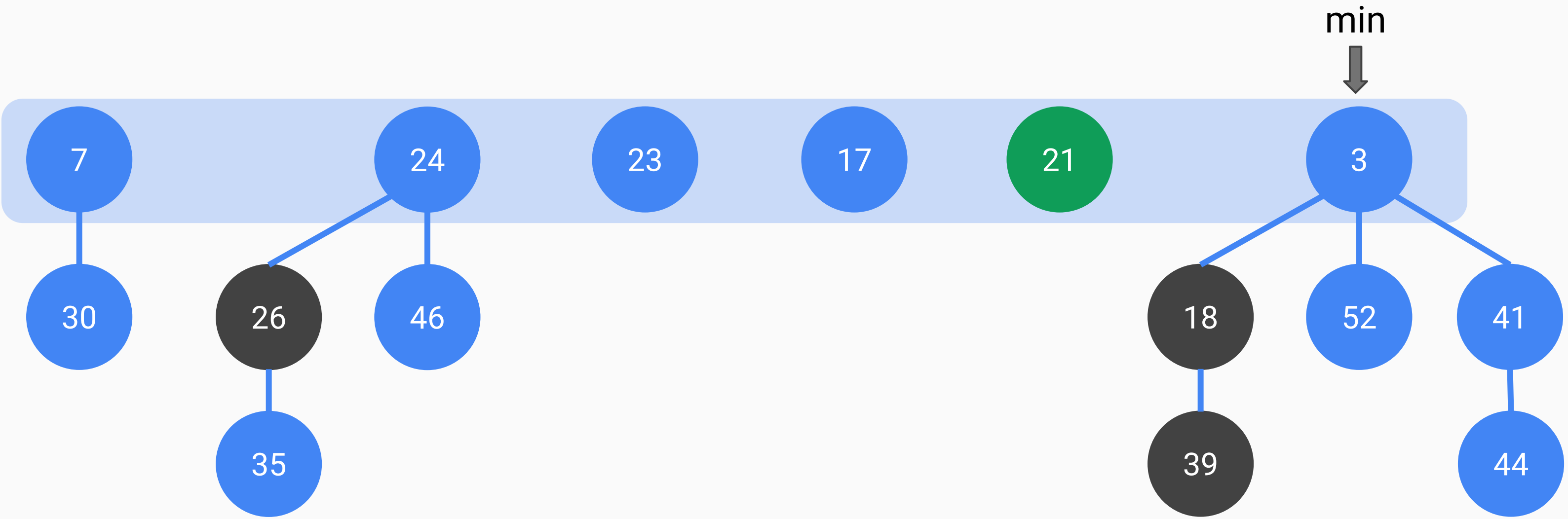
Find-Min

By the heap property, every tree in the heap will have the smallest element at the root. We can maintain a *min* variable to let us quickly access the minimum value in O(1) time.

Insert

For insertion, we add the new node.

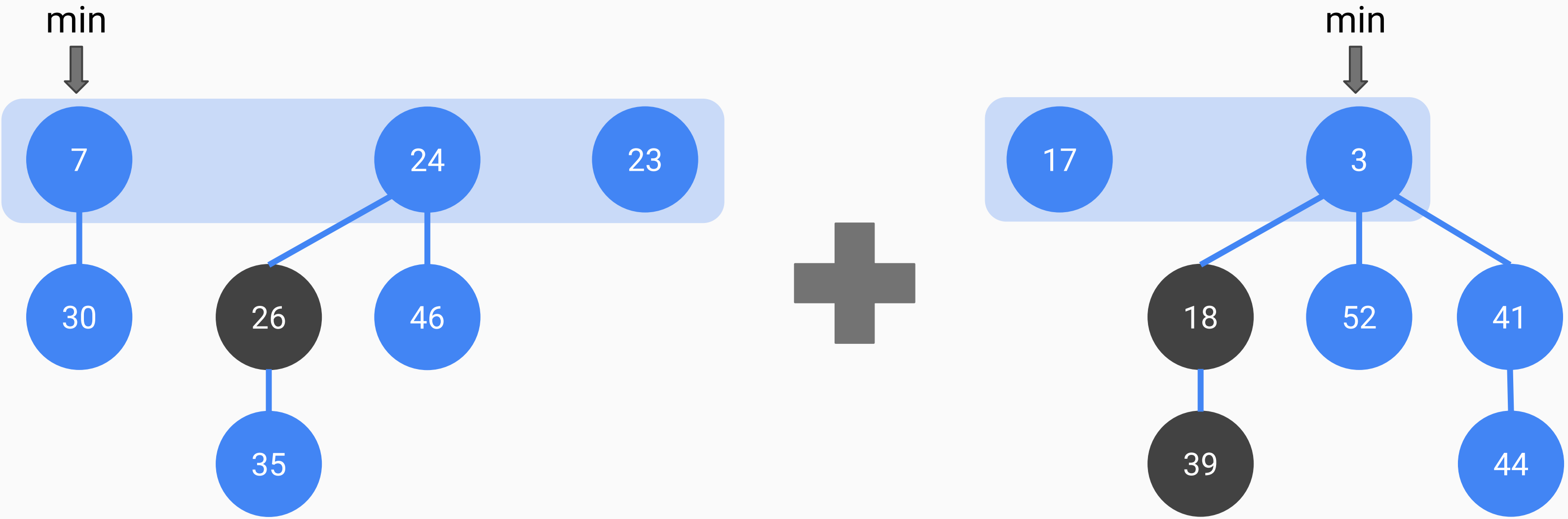
And that's it!



* Alternatively, use a merge like in binomial heaps.

Merge

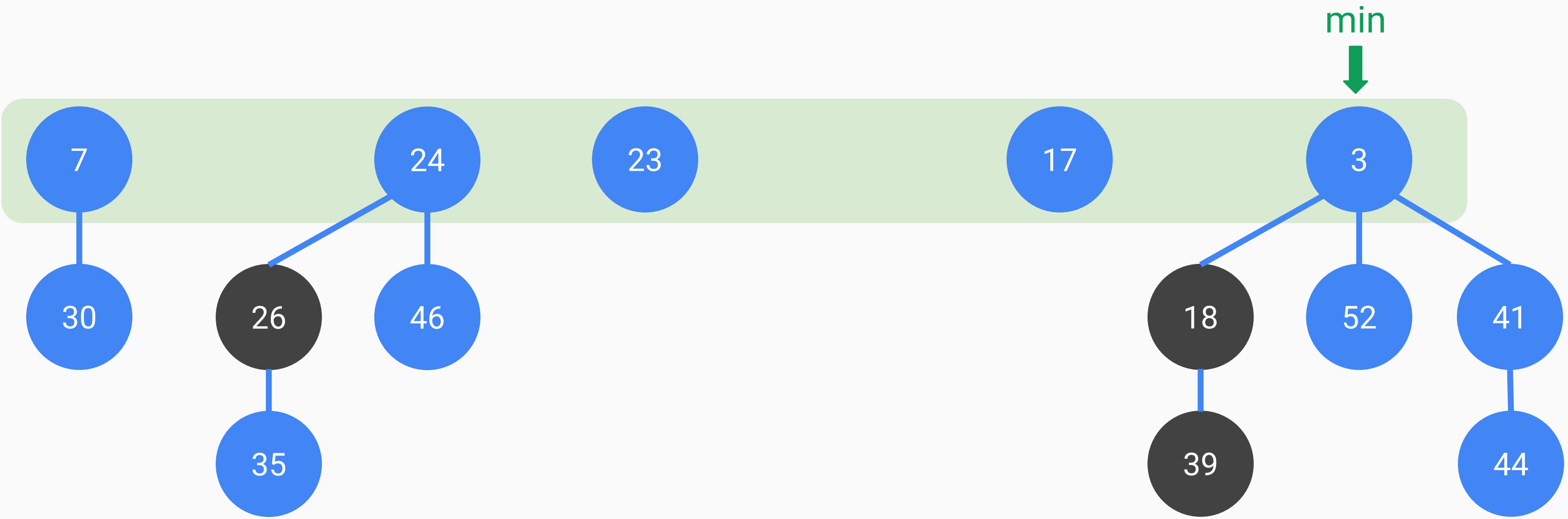
How do I combine two fibonacci heaps?



Merge

Just … combine!

And update *min*.



Extract-Min

We can split *Extract-Min* into three phases:

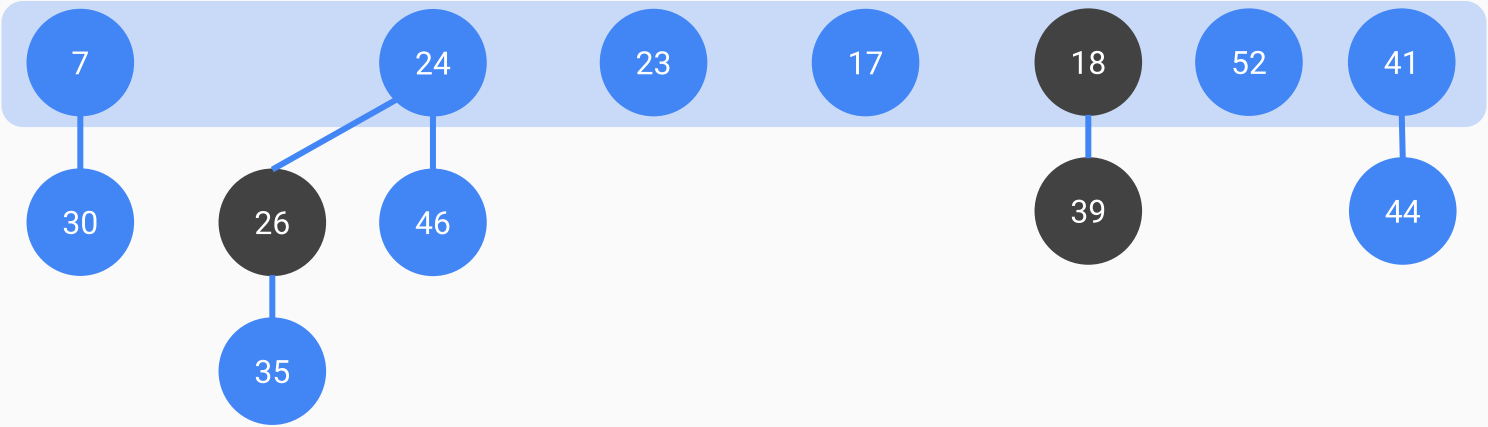
1. Delete the minimum node and add it's children to root list
2. Combine trees - make sure no roots of trees have the same degree
3. Update *min*

Extract-Min

Phase 1: Similar to binomial heaps we remove the minimum node, then add it's children into the root list as individual trees.

Extract-Min

Phase 1: Similar to binomial heaps we remove the minimum node, then add it's children into the root list as individual trees.



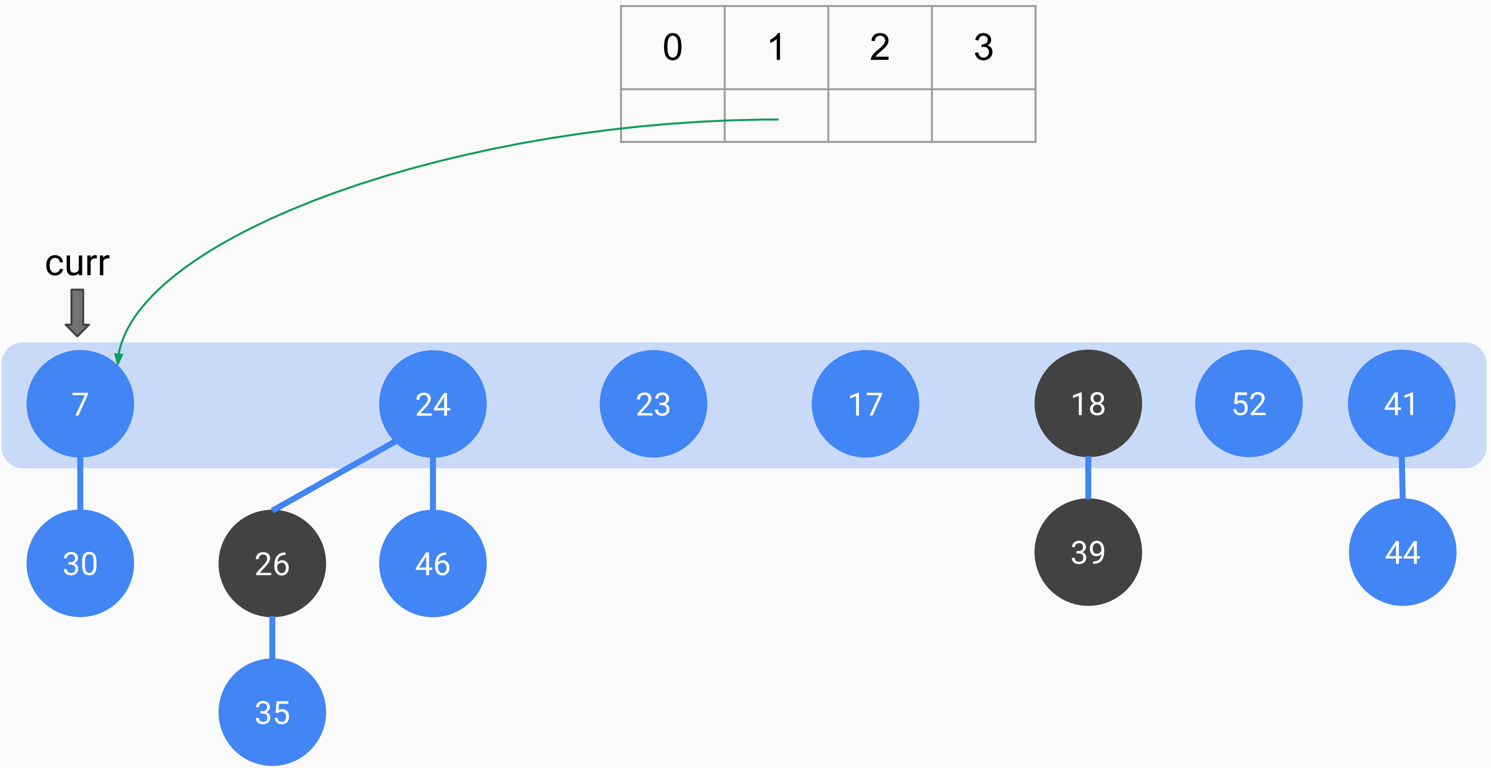
Extract-Min

Phase 2: Combine the trees by merging trees with the same degree.

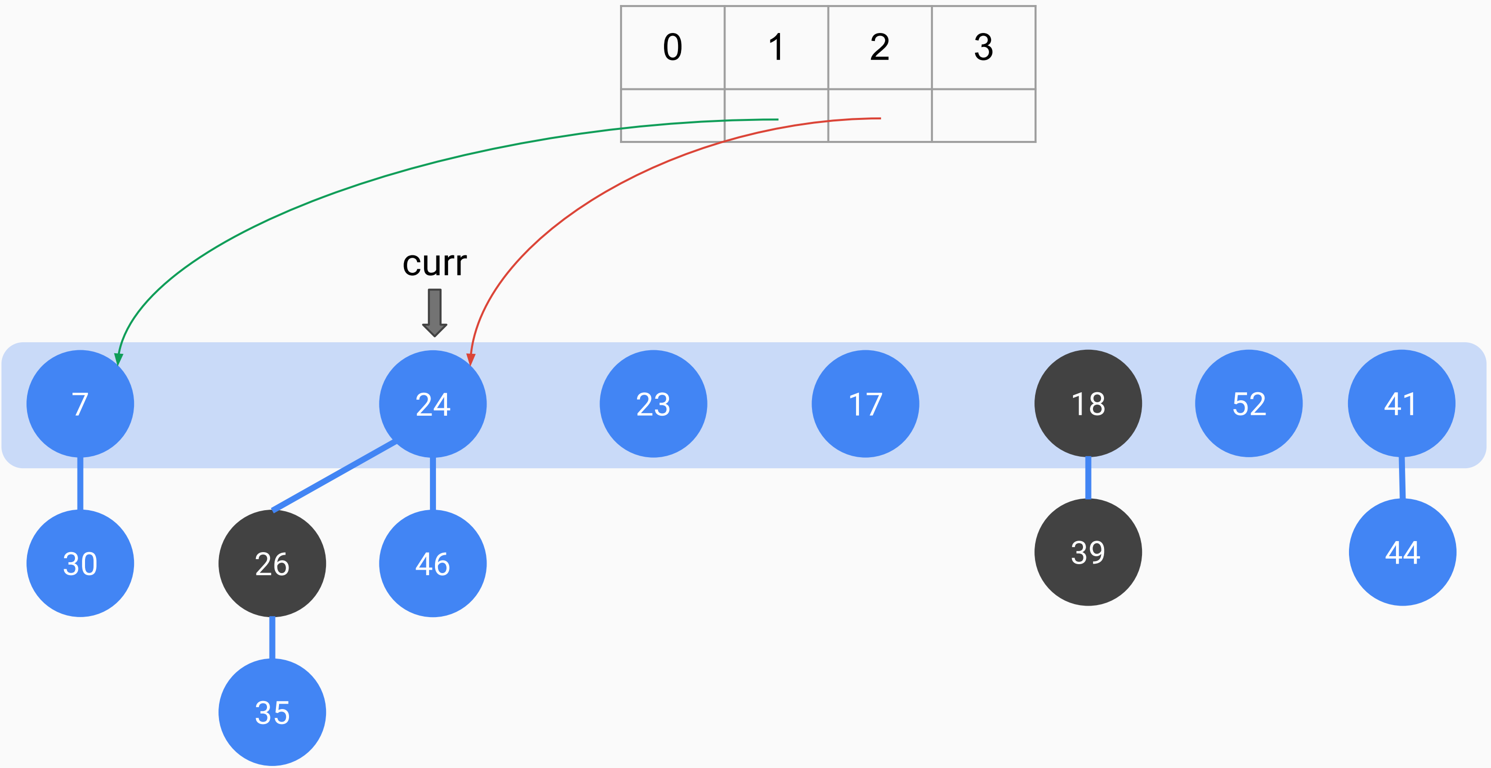
We can do this efficiently by keeping an array of pointers *A*, that can each point to a tree's root of each degree.

We iterate through the roots of the trees in the fibonacci heap. When we find the first root which has degree *d*, we will assign that root to *A[d]*. If we find more than one tree of the same degree, we will combine the two trees and update the array, setting *A[d]* to *null*. (The root list is also updated with the newly combined tree.)

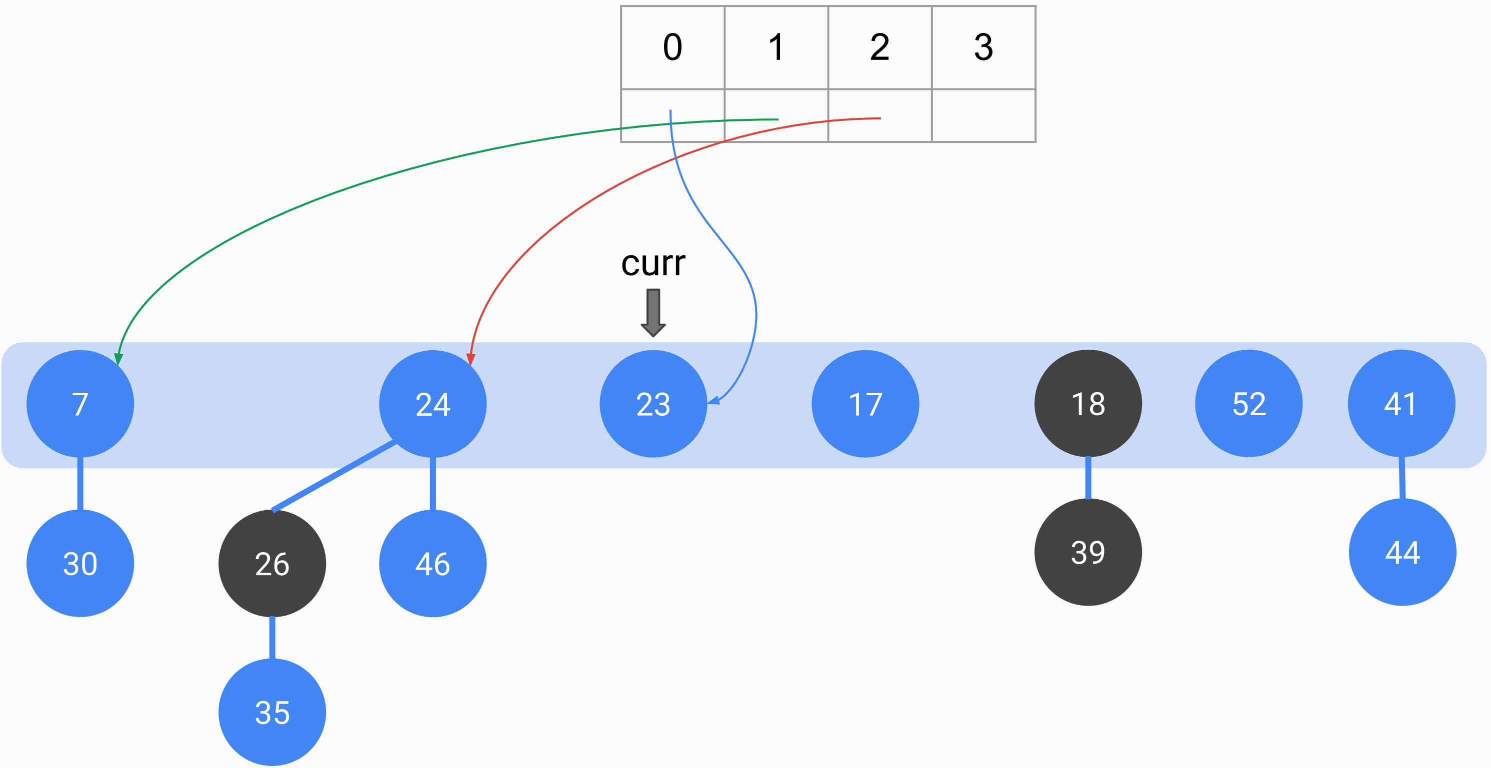
Extract-Min



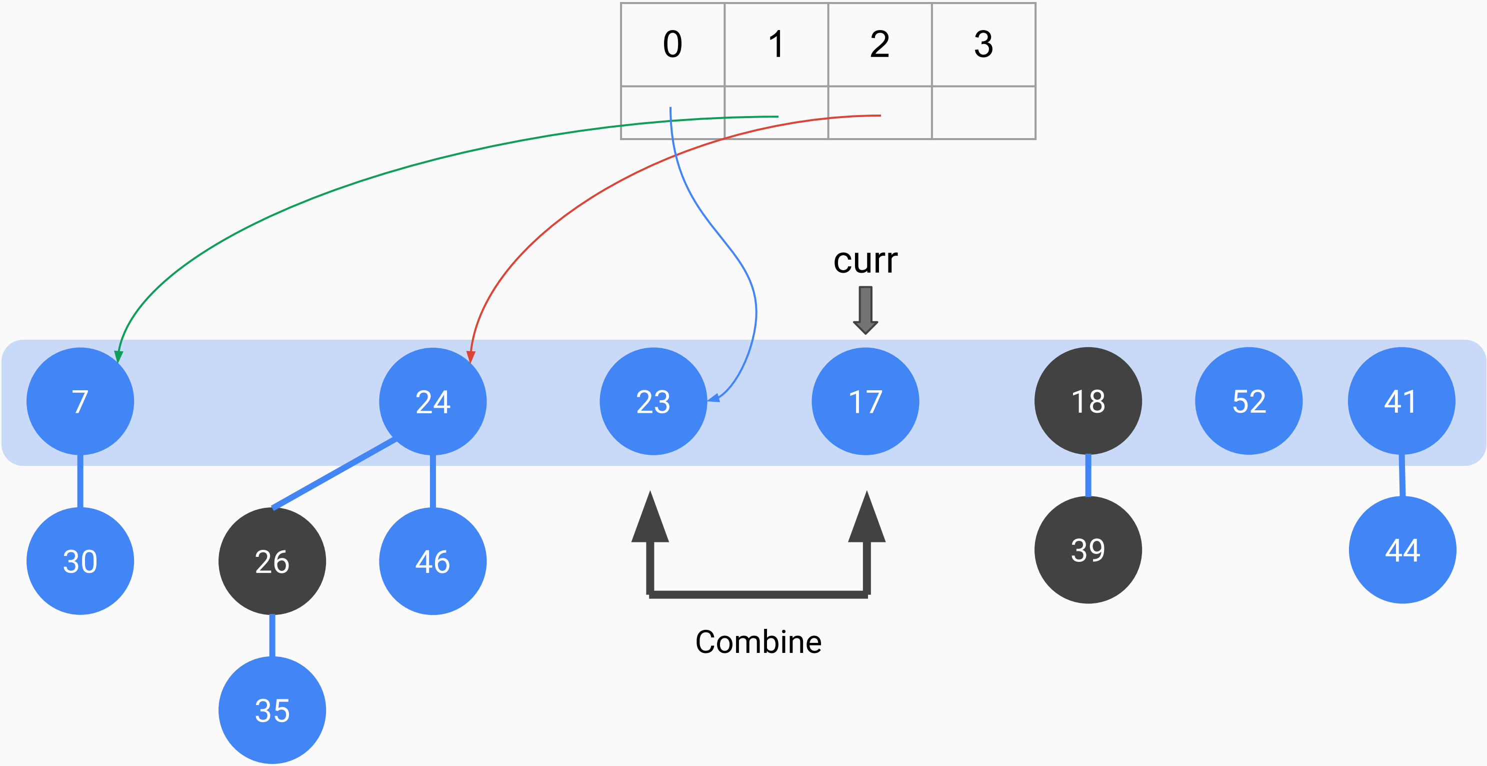
Extract-Min



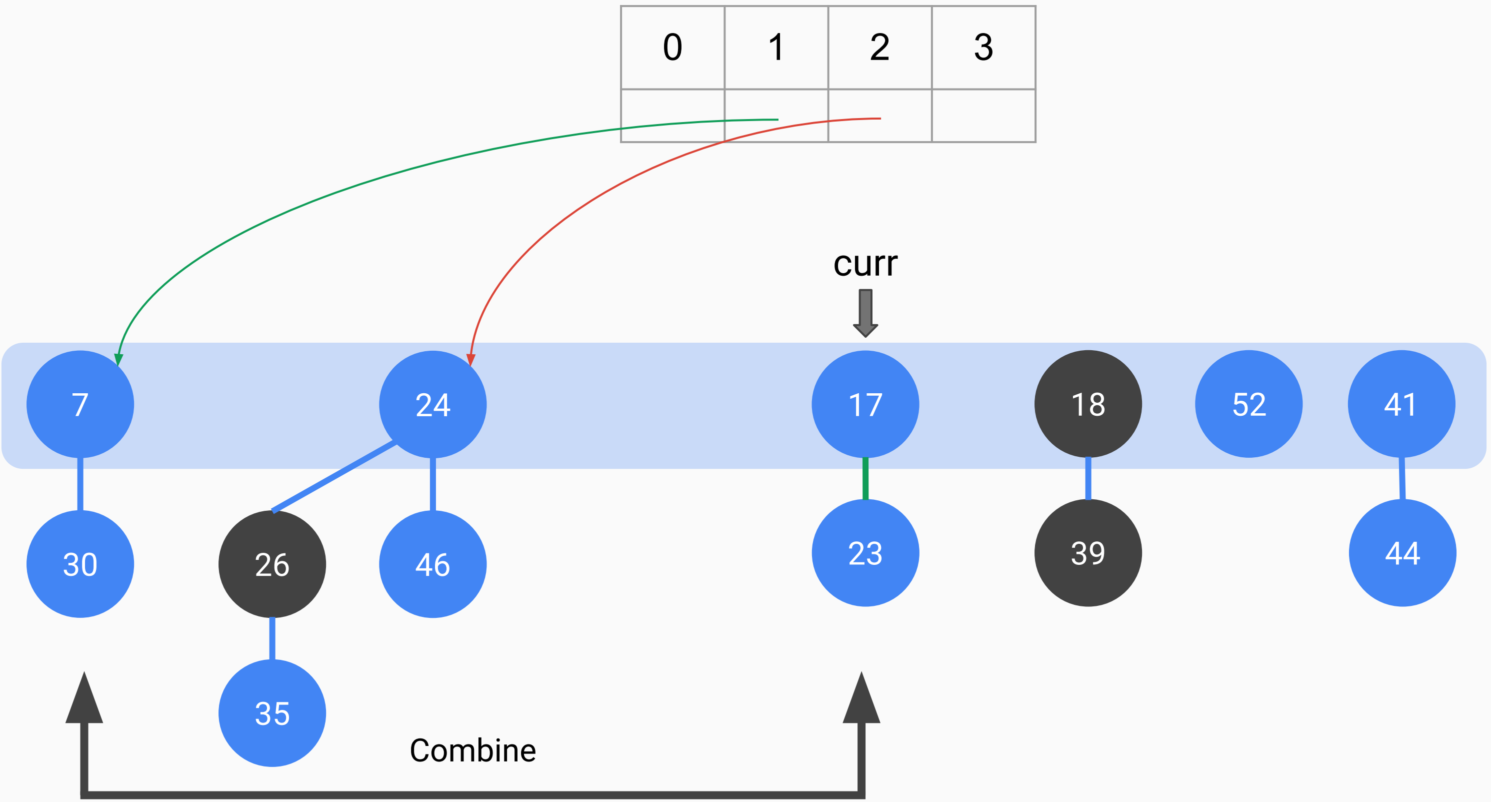
Extract-Min



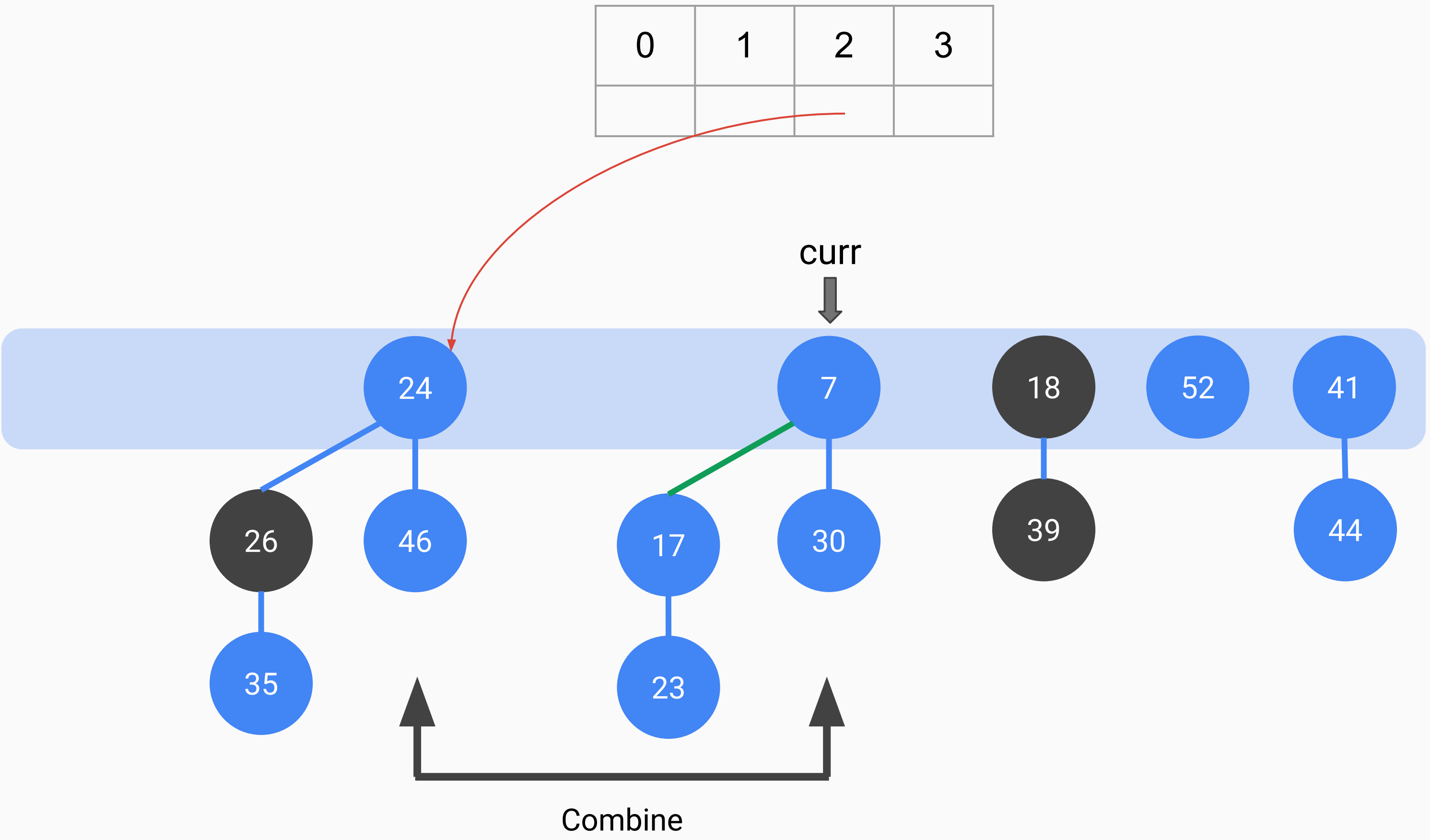
Extract-Min



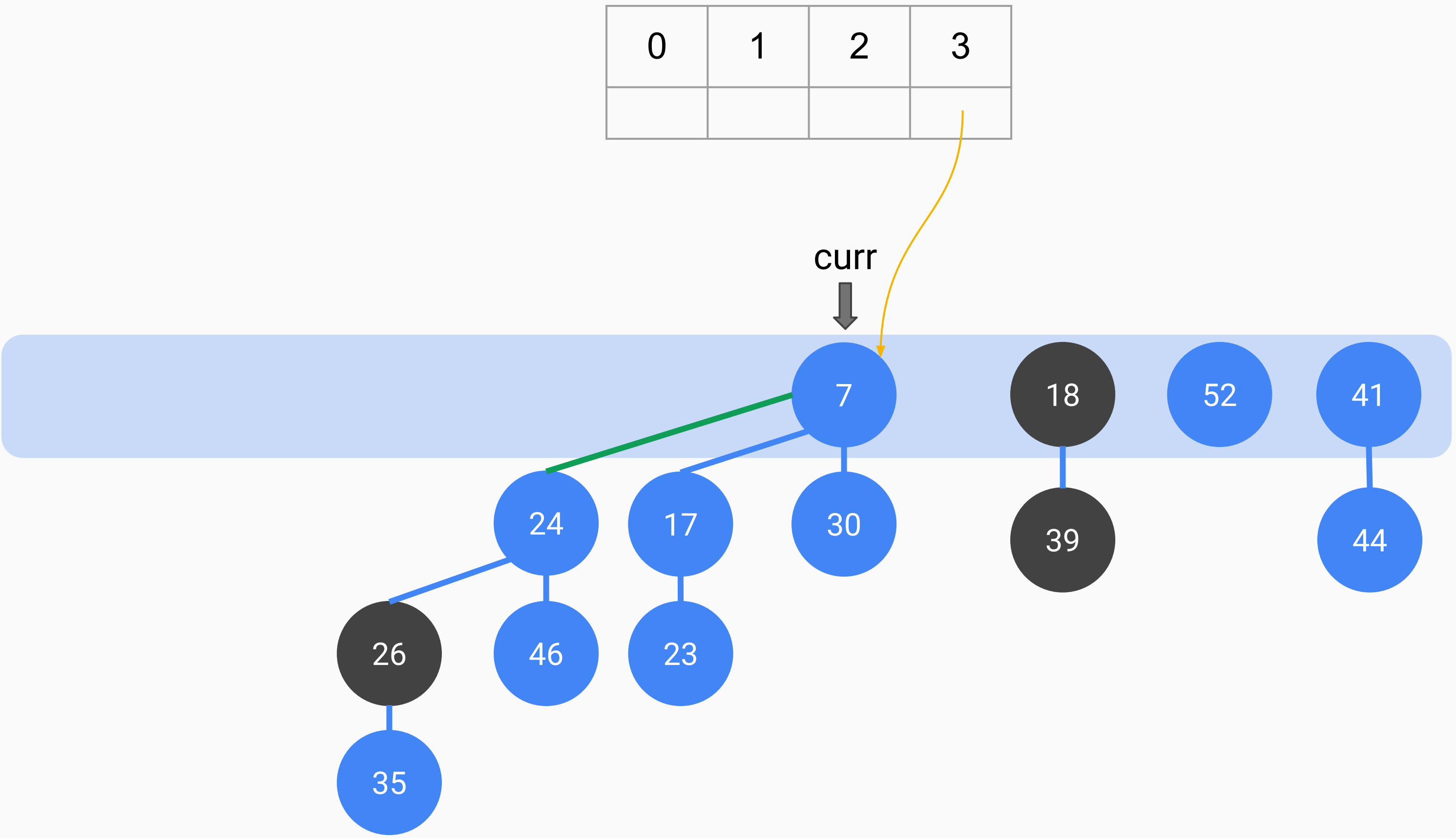
Extract-Min



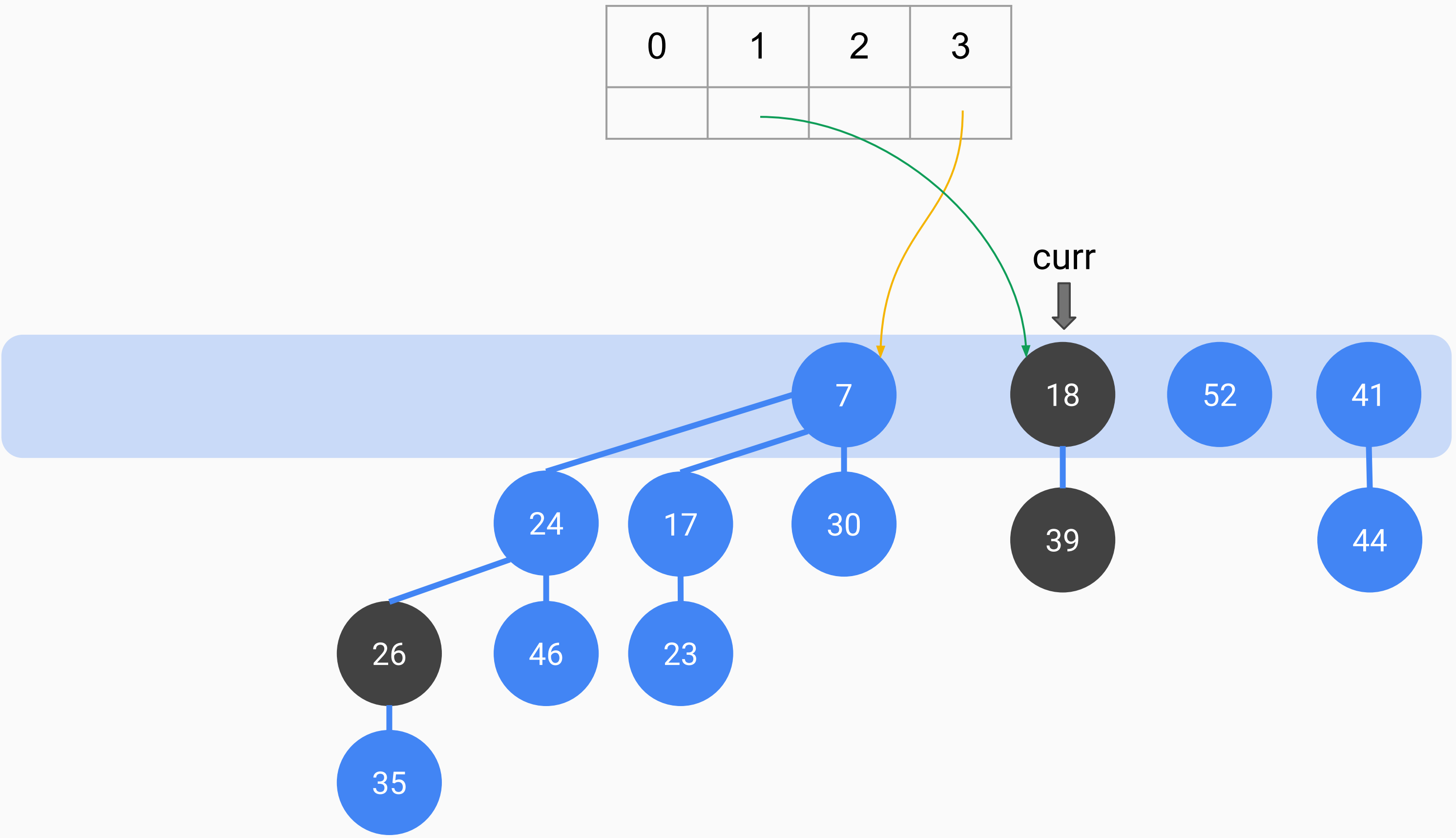
Extract-Min



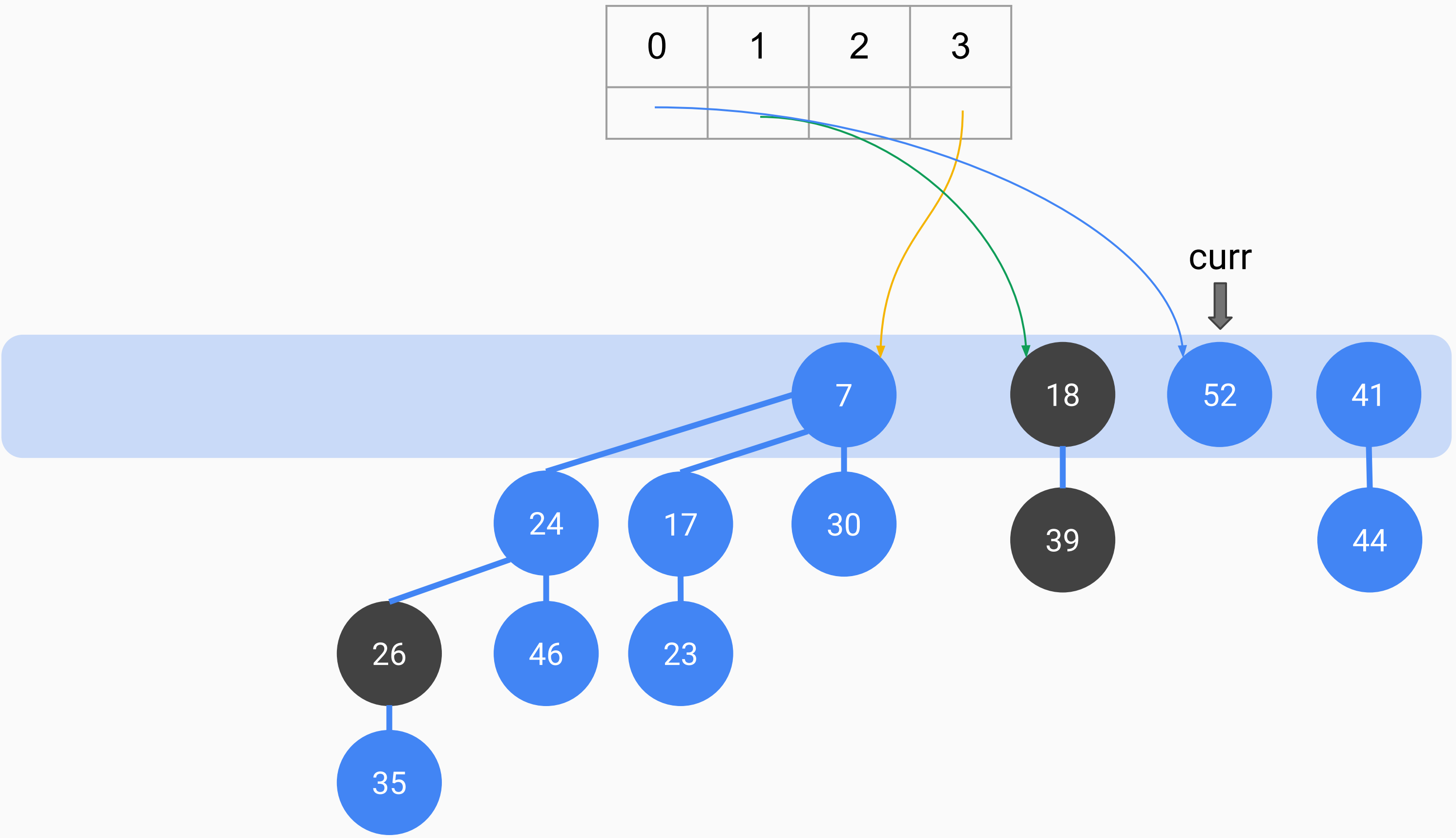
Extract-Min



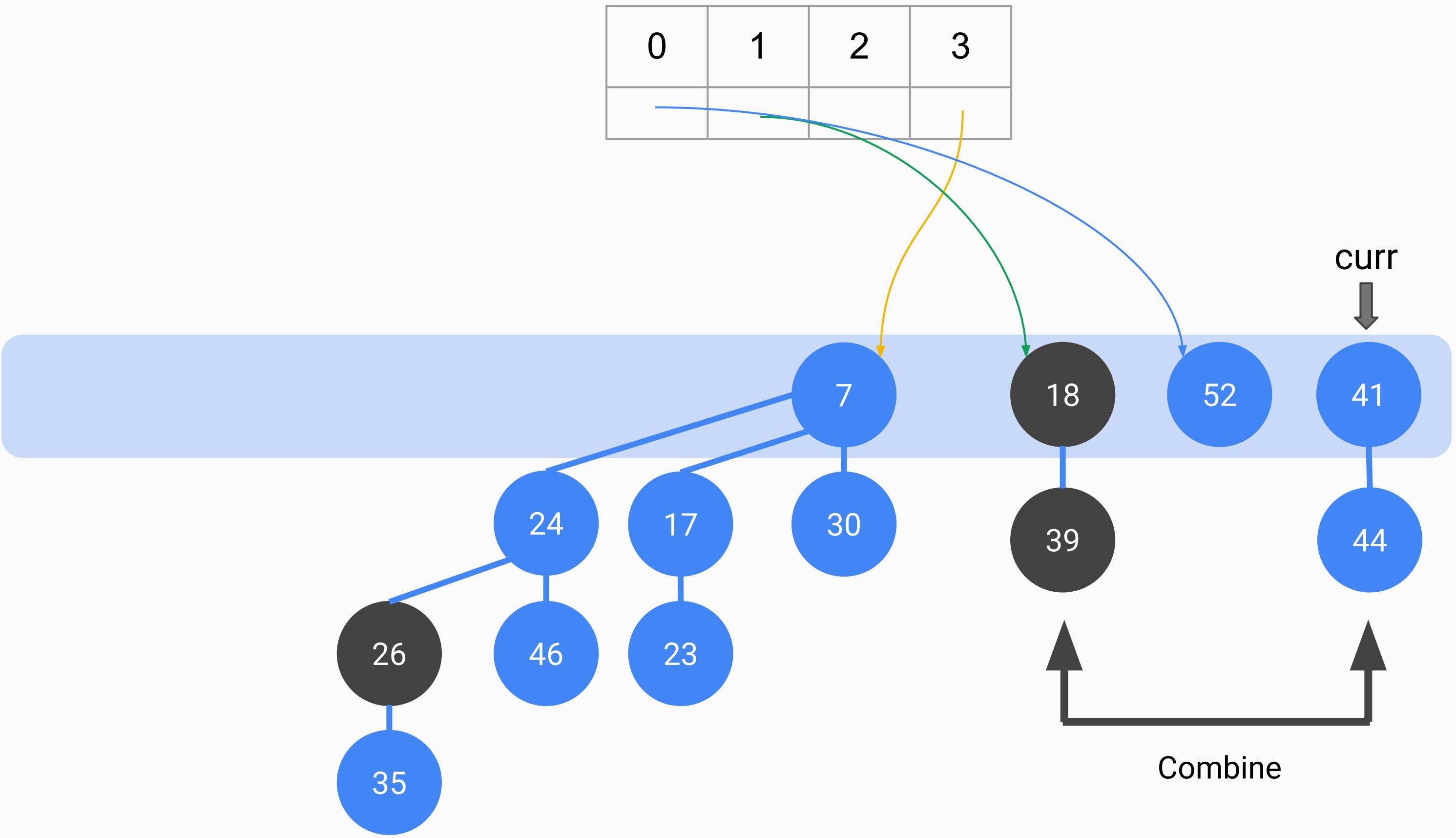
Extract-Min



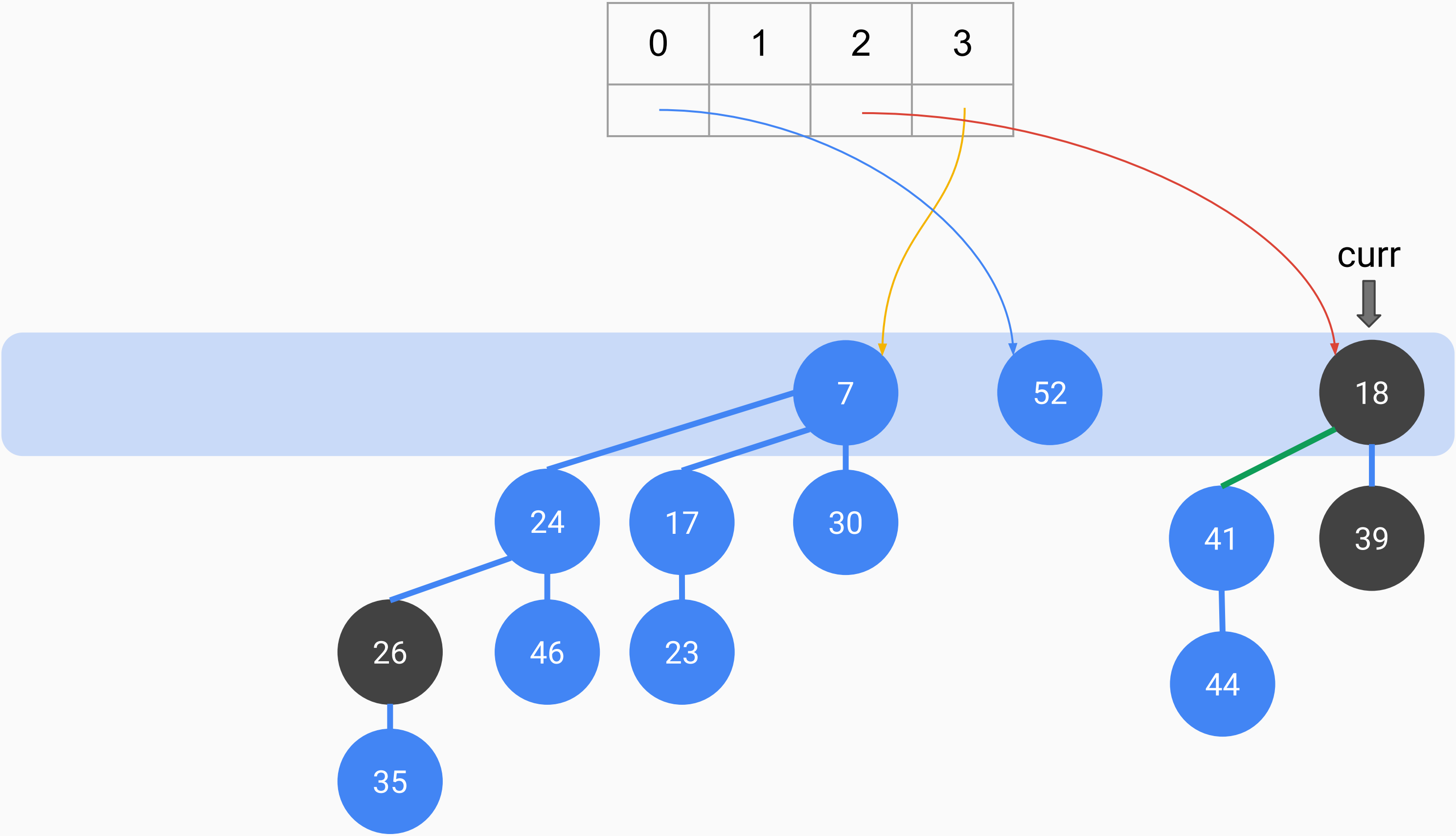
Extract-Min



Extract-Min

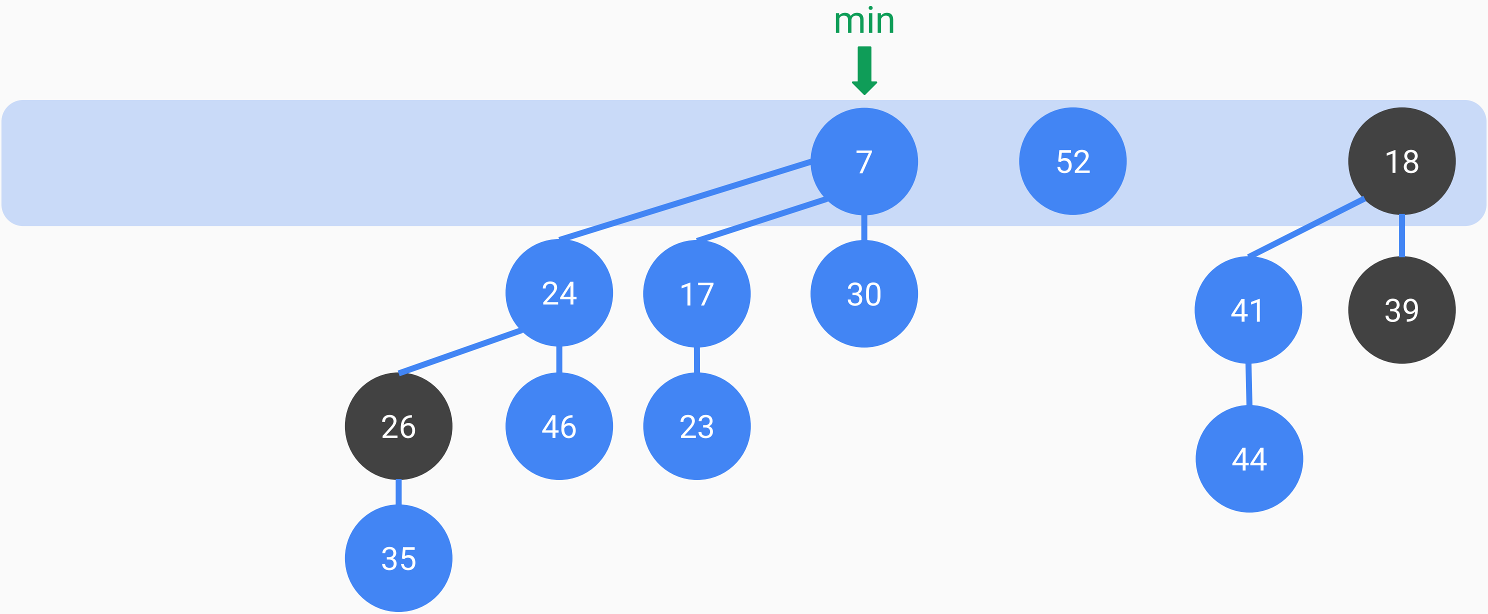


Extract-Min



Extract-Min

Phase 3: Update *min* by iterating through all the roots



Analysis of Extract-Min

1. Delete the minimum node and add it's children to root list
   - O(log *n*) since we add up to d(*n*) nodes to root list
2. Combine trees - make sure no roots of trees have the same degree
   - O(log *n* + t(*H*)), since there are at most d(*n*) + t(*H*) roots at start of Phase 2, and at most O(d(*n*)) combines
3. Update *min*
   - O(log *n*), because after Phase 2 there will only be O(log *n*) trees

The time complexity in the worst-case would be O(log *n* + t(*H*)). However, the amortised cost of Extract-Min is actually O(log *n*). The idea here is that we are not always going to have exactly d(n) + t(*H*) trees to merge in Phase 2, at may only need less than O(d(*n*)) combines.

Decrease-Key

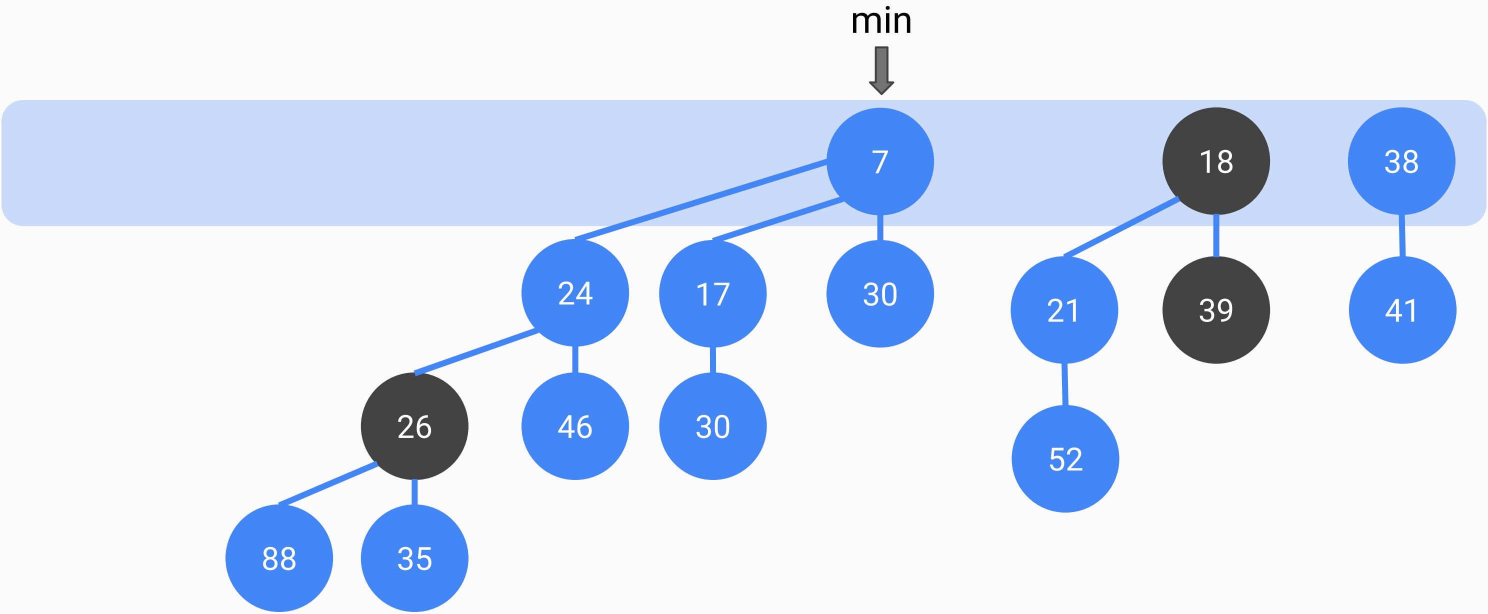
Again, we can split *Decrease-Key* into two phases:

1. Decrease the value of the node to the new value
2. Cuts and marks
   a. If node violates heap property, cut it off from the parent and put it into the root list (as a new tree).
   b. Assign the parent node to the variable *y*. If *y* was marked, cut off *y*, put it into the root list, and unmark it. Set the new value of *y* to be the original *y*'s parent. Repeat this step until *y* is an unmarked node.
   c. Mark *y* if it is not a root.

* Phase 2b-2c is also known as cascading cuts

Decrease-Key

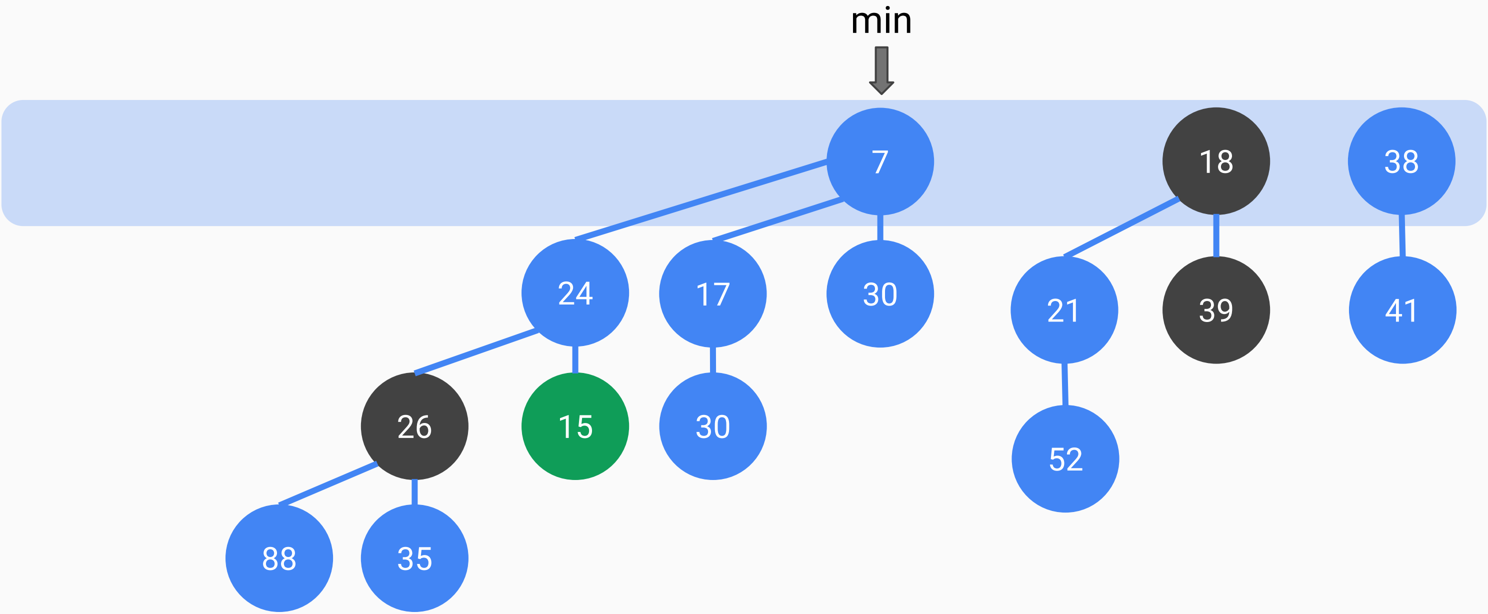
Decrease-Key(46, 15)



Decrease-Key

Decrease-Key(46, 15)

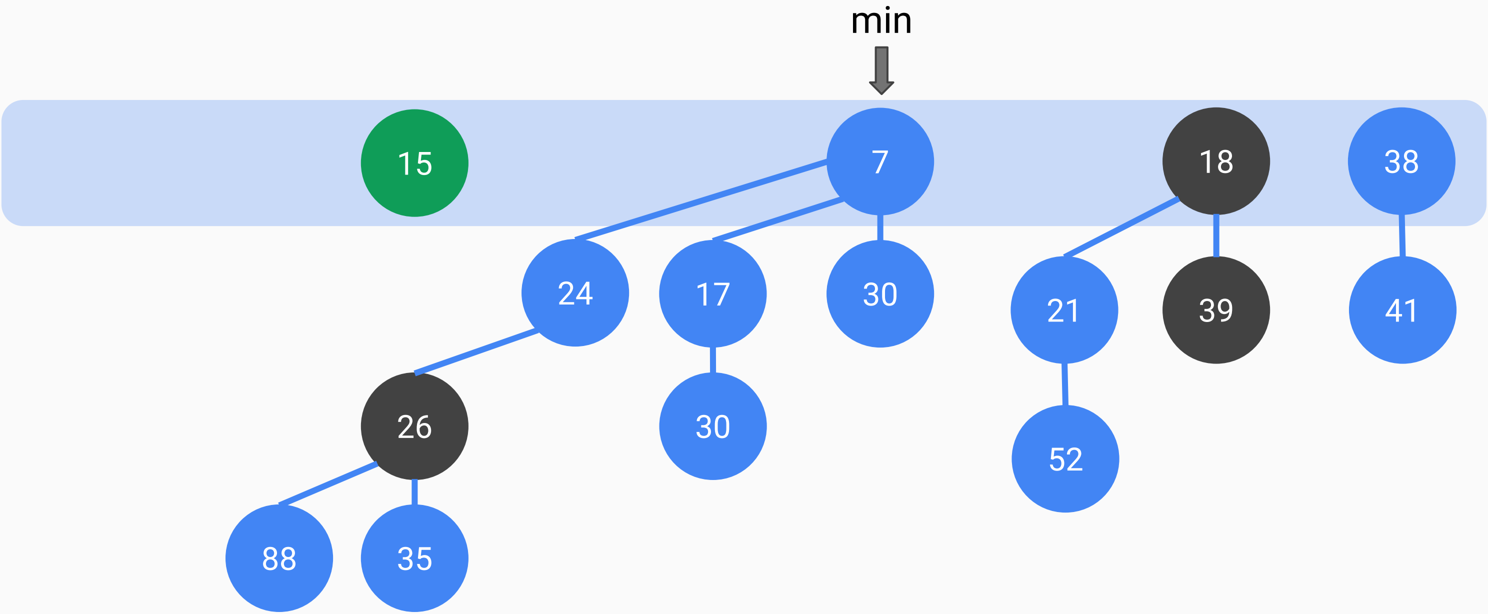
Phase 1: Decrease the value of the node to the new value



Decrease-Key

Decrease-Key(46, 15)

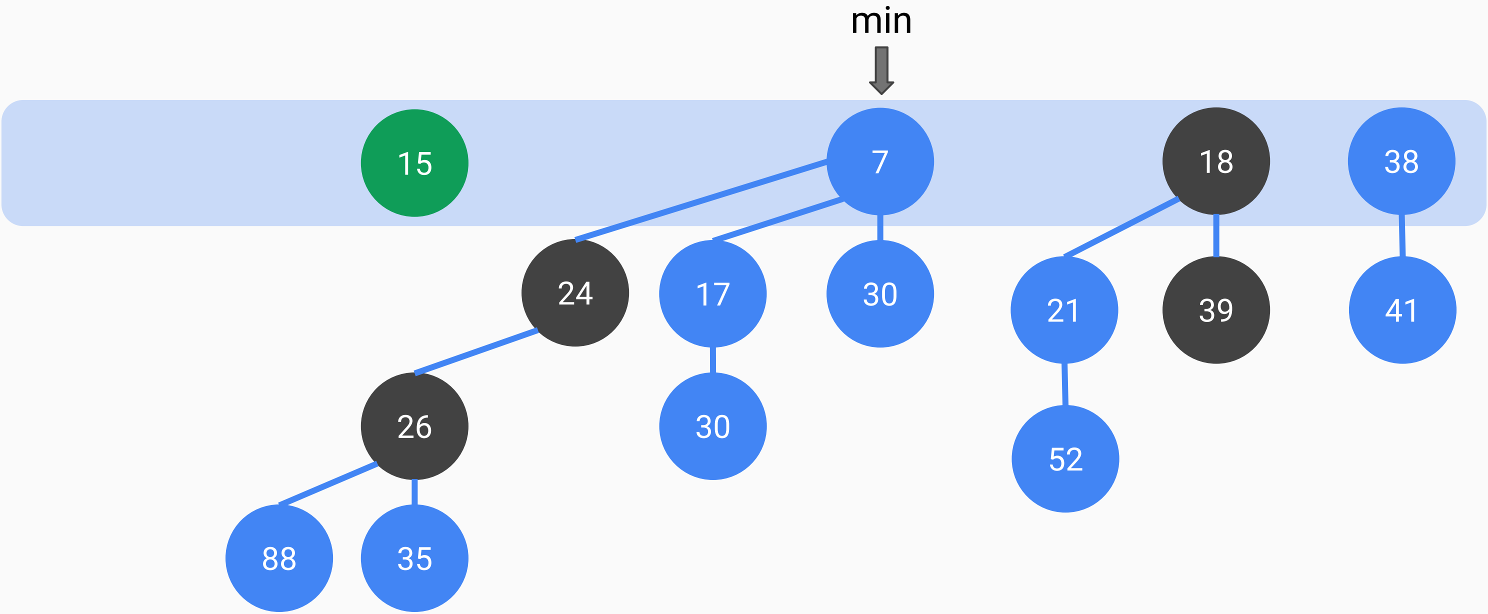
Phase 2a: If node violates heap property, cut it off from the parent and put it into the root list.



Decrease-Key

Decrease-Key(46, 15)

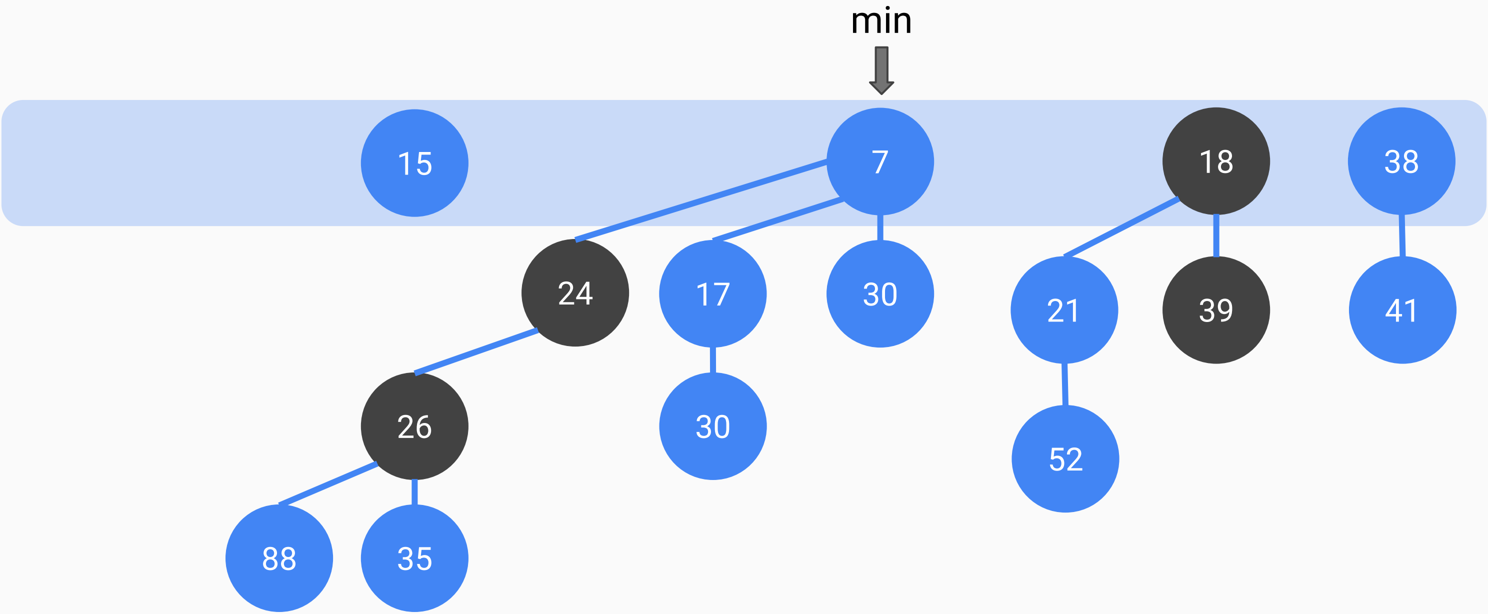
Phase 2c: Mark y if it is not a root.



Decrease-Key

Decrease-Key(46, 15)

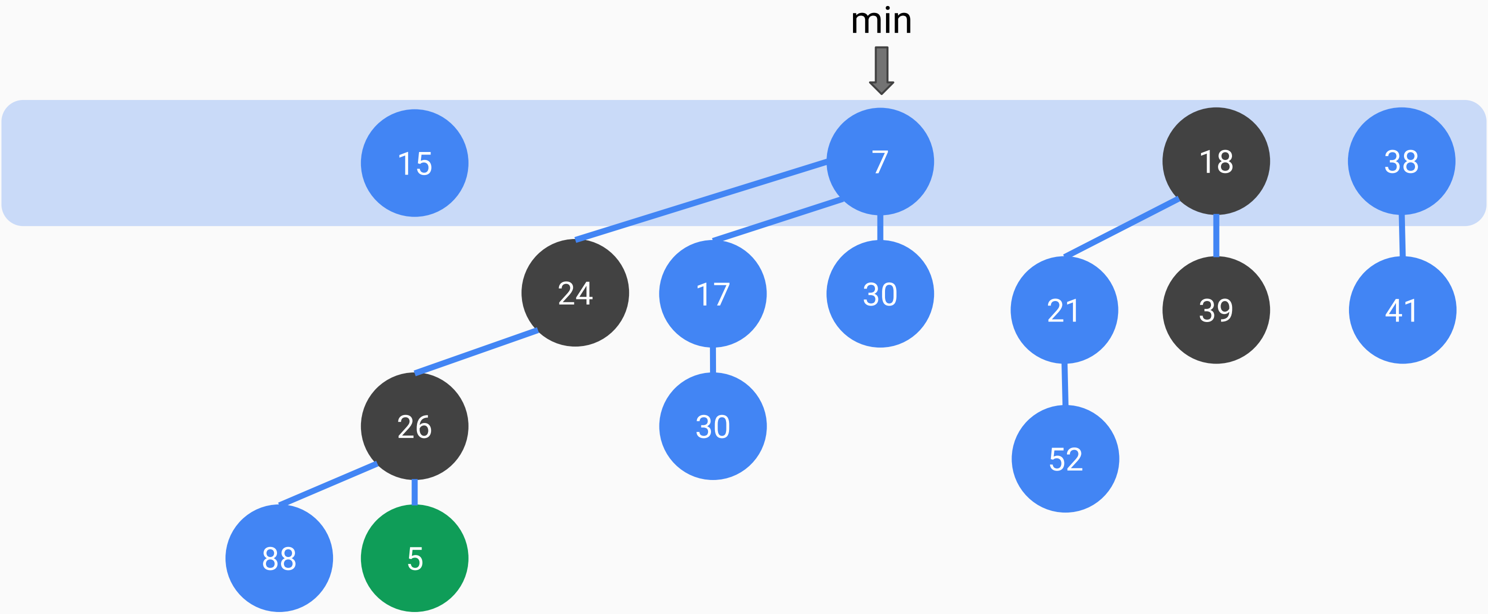
Done!



Decrease-Key

Decrease-Key(35, 5)

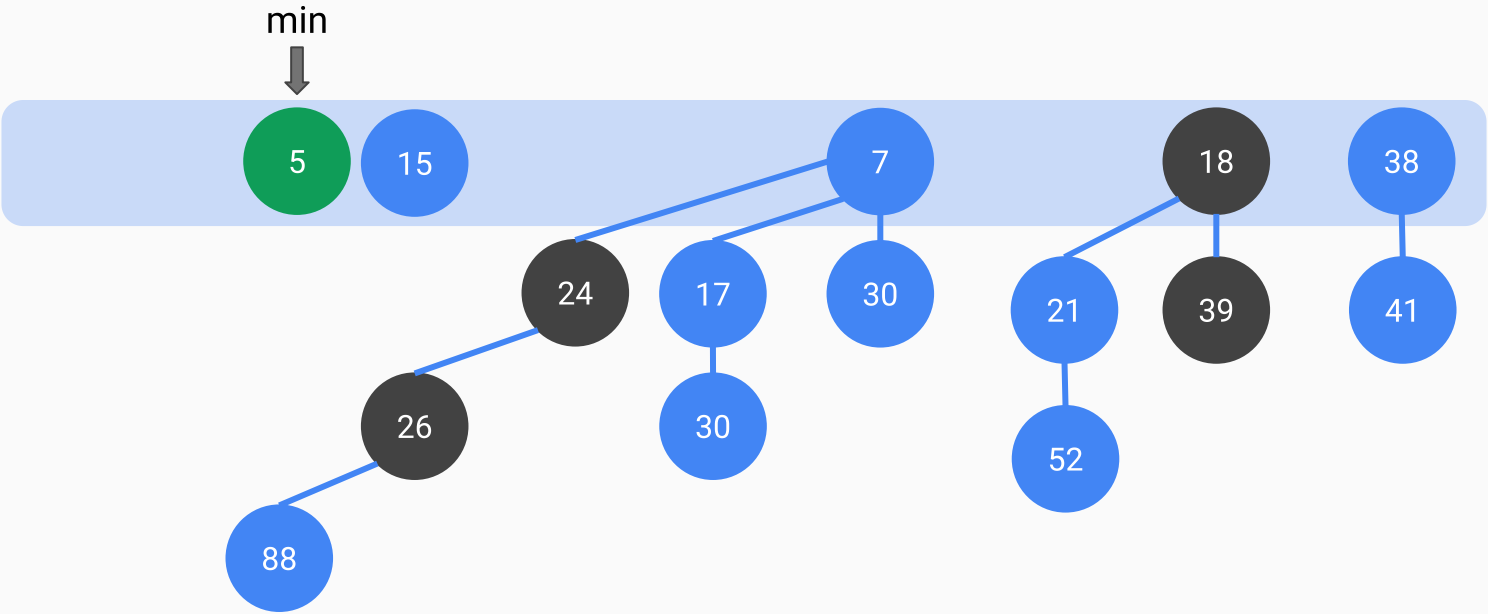
Phase 1: Decrease the value of the node to the new value



Decrease-Key

Decrease-Key(35, 5)

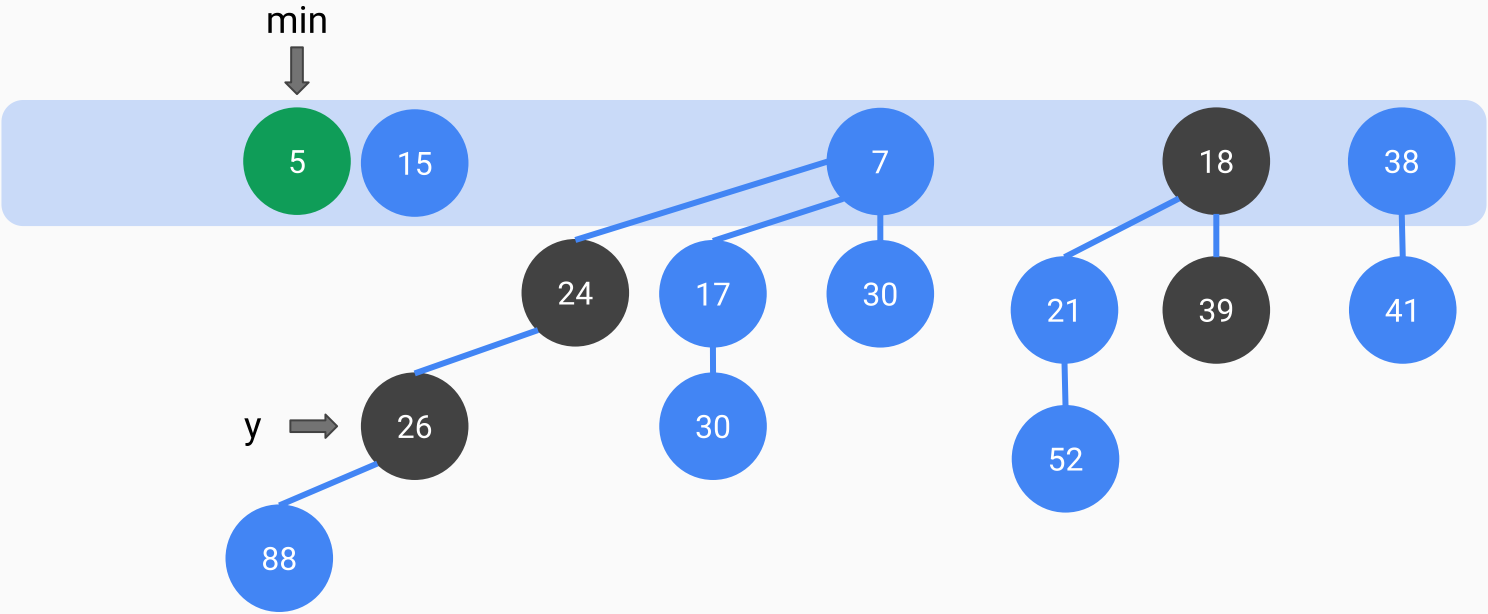
Phase 2a: If node violates heap property, cut it off from the parent and put it into the root list.



Decrease-Key

Decrease-Key(35, 5)

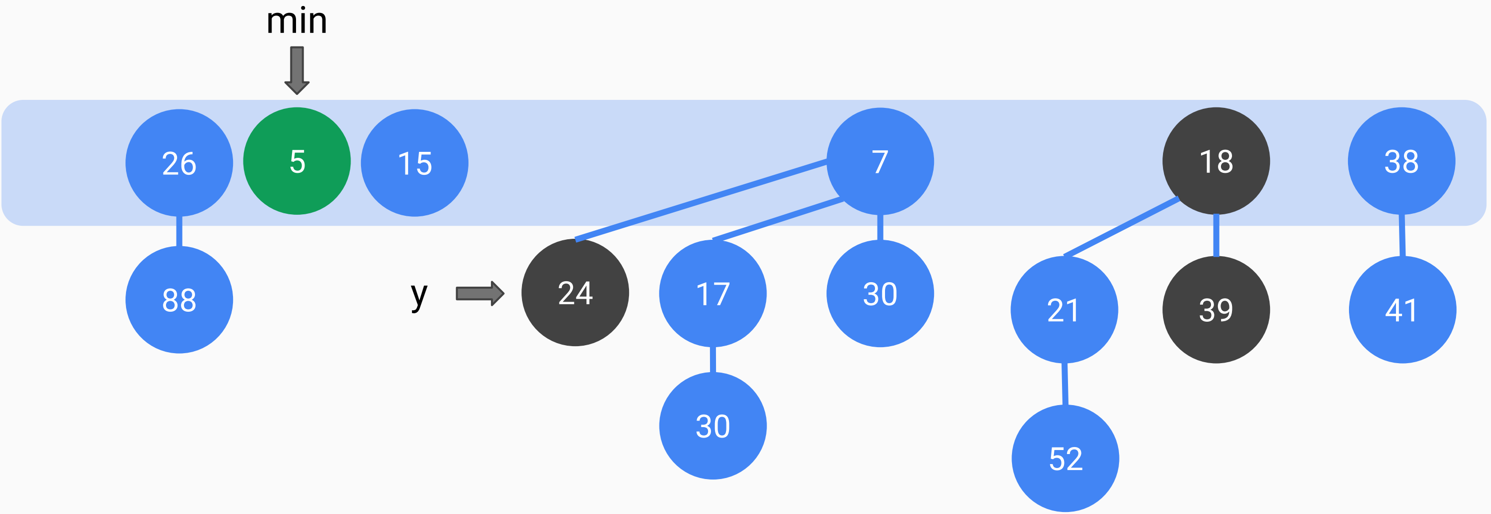
Phase 2b: Assign the parent node to the variable y. If y was marked, cut off y, put it into the root list, and unmark it. Set the new value of y to be the original y's parent. Repeat this step until y is an unmarked node.



Decrease-Key

Decrease-Key(35, 5)

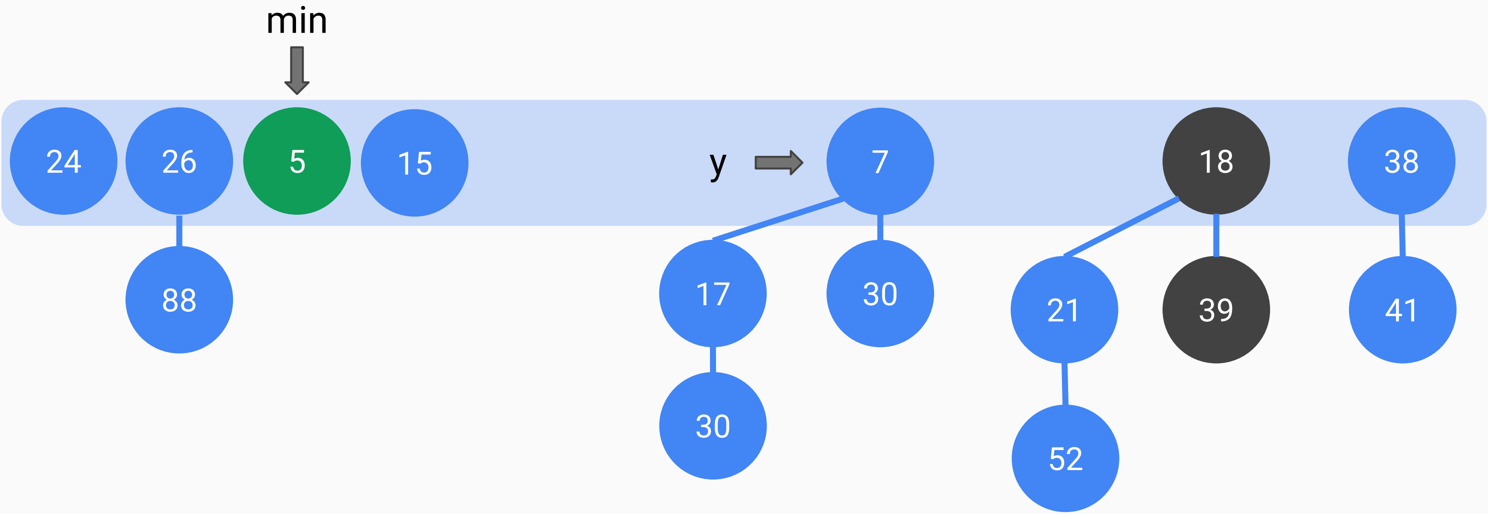
Phase 2b: Assign the parent node to the variable y. If y was marked, cut off y, put it into the root list, and unmark it. Set the new value of y to be the original y's parent. Repeat this step until y is an unmarked node.



Decrease-Key

Decrease-Key(35, 5)

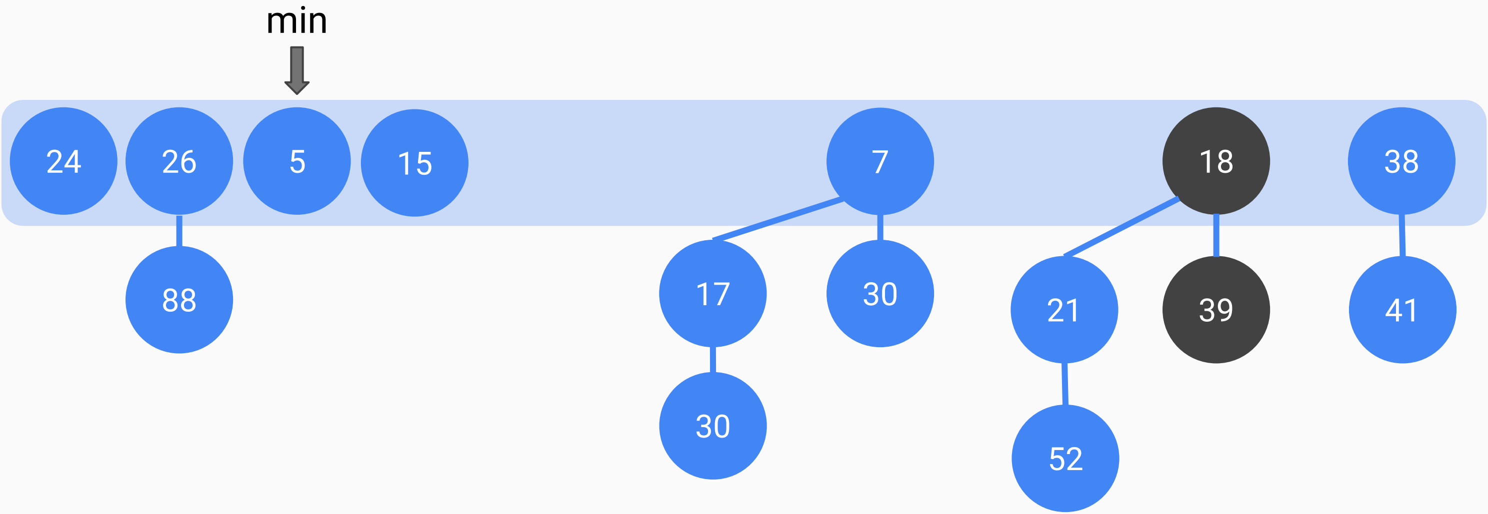
Phase 2b: Assign the parent node to the variable y. If y was marked, cut off y, put it into the root list, and unmark it. Set the new value of y to be the original y's parent. Repeat this step until y is an unmarked node.



Decrease-Key

Decrease-Key(35, 5)

Done!



Analysis of Decrease-Key

In total, we need to make a total of $k \geq 1$ cuts, where k is the number of new trees created (including the node with the changed value). The time complexity in the worst case would be $O(k)$.

However, the amortised time complexity is $O(1)$ time. The intuition behind this is that most of the time, we will not be making a lot of cuts at the start when there are not many marked nodes, and there will only be some "expensive" operations where there will be several cuts needed in the upwards path towards the root.

Delete

Exactly the same as binomial heap: Decrease the key to $-\infty$ and run Extract-Min.

Before we go

- Fibonacci heaps give good theoretical guarantees on the operations, but are not necessarily good in practice
  - Not easy to implement
  - Not fast in practice due to high constant factor
  - High memory usage
- The idea of being “lazy”, or doing things later can be applied in many more situations (eg. lazy propagation in BSTs)
- So which heap should I use?
  - Best heap to choose might be input / operation dependent
  - When you do not need *decrease-key*, array based implementations are good.
  - If you need *decrease-key*, consider heaps like pairing heaps
  - See https://arxiv.org/pdf/1403.0252.pdf