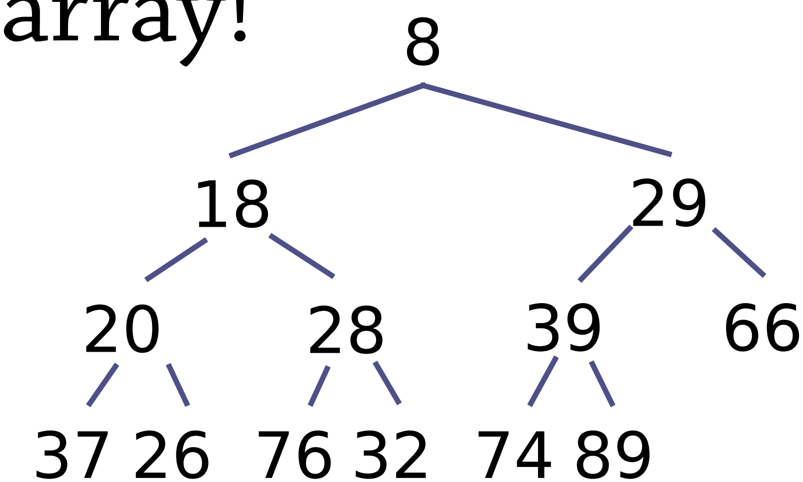
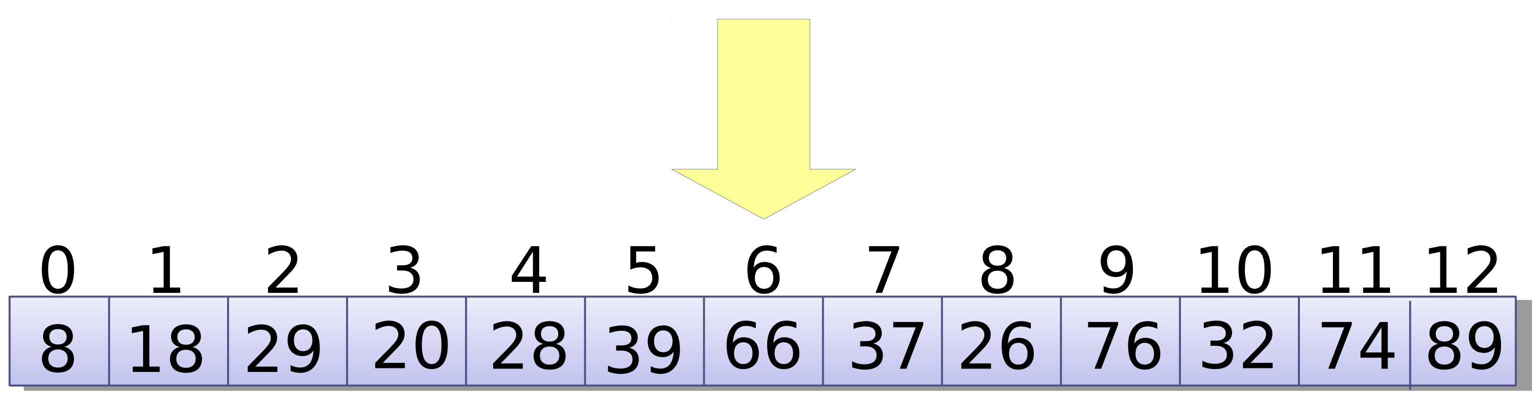
Binary heaps
(chapters 20.3 – 20.5)
Leftist heaps

Binary heaps are arrays!

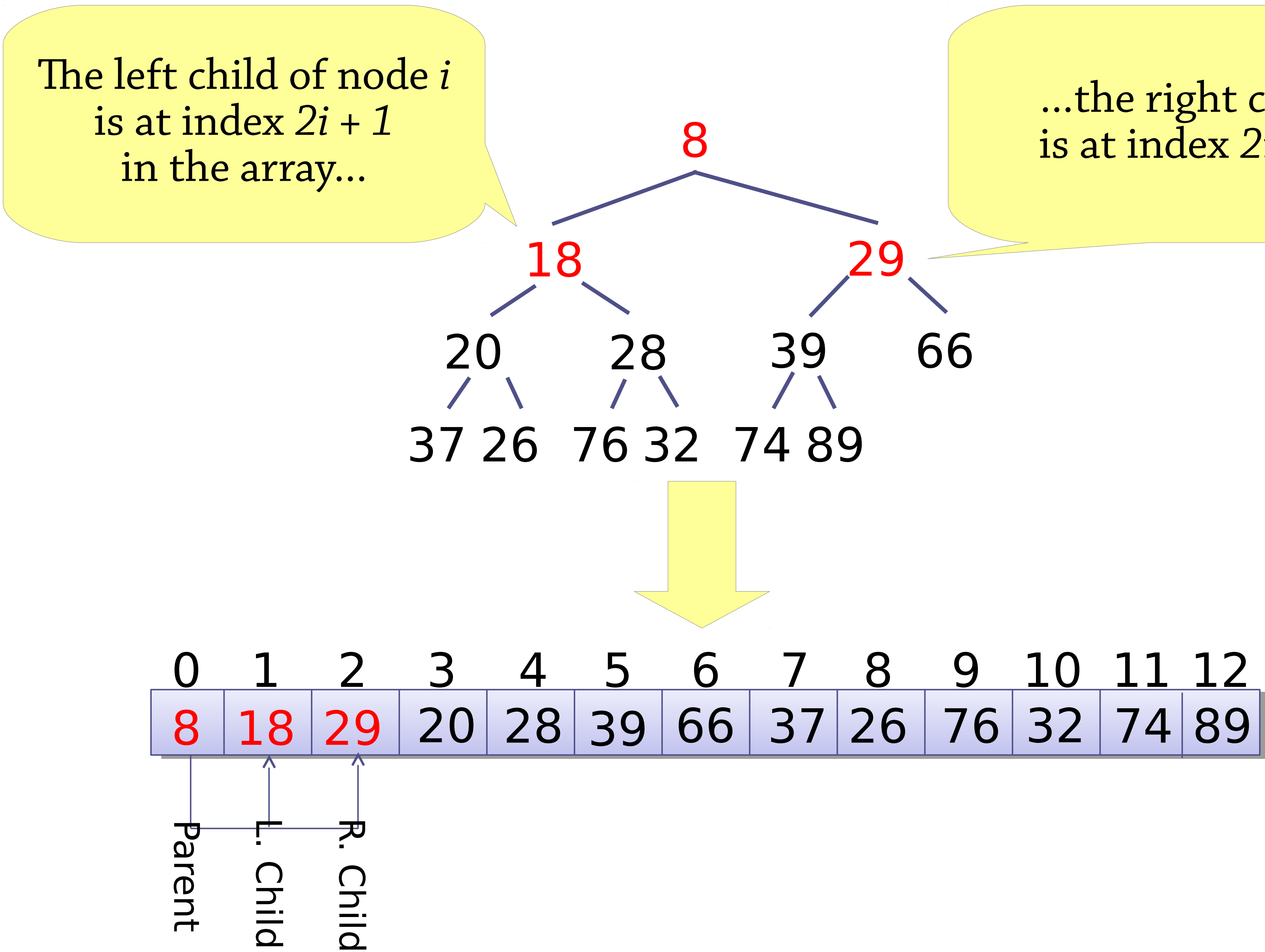
A binary heap is really implemented using an array!



Possible because of completeness property



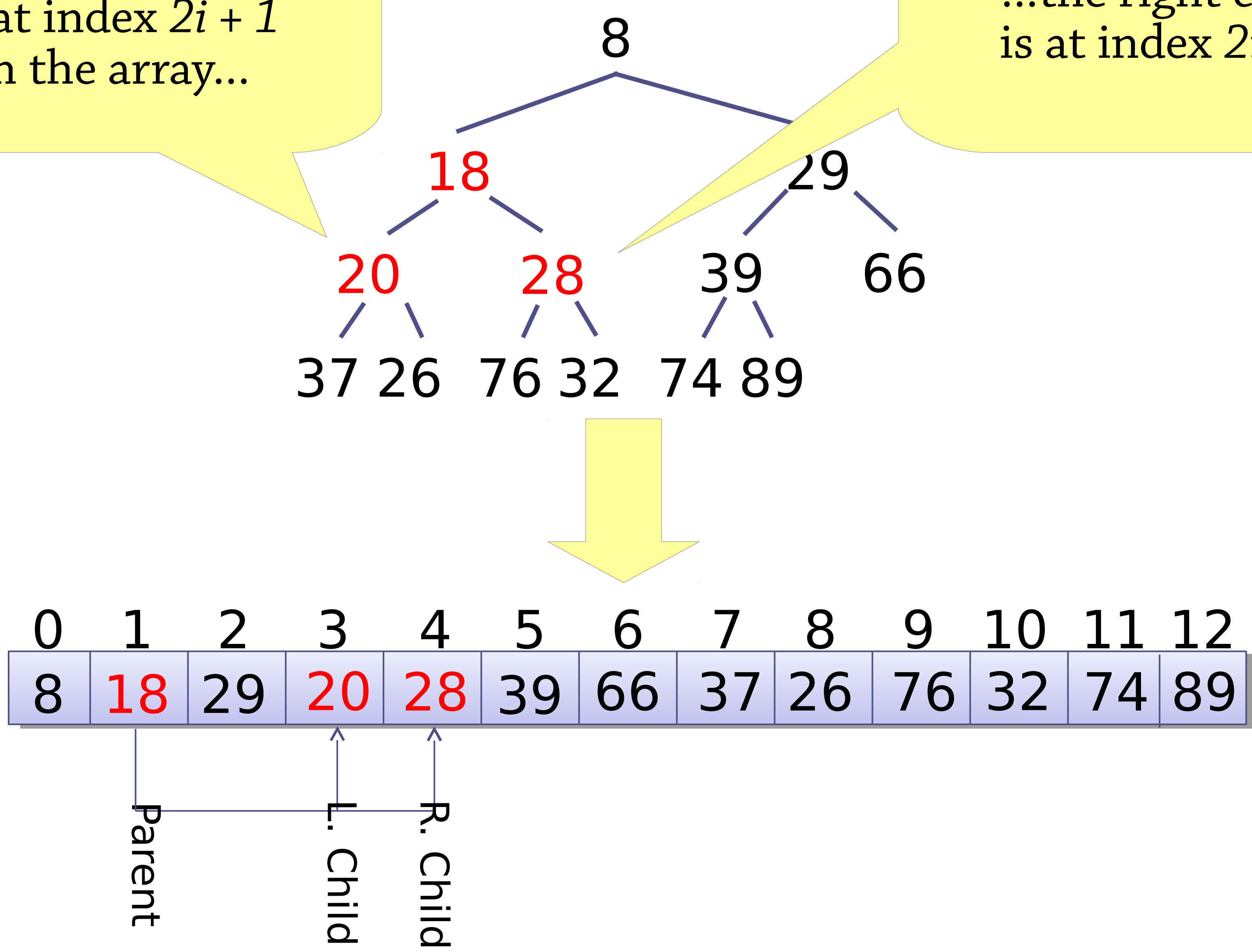
Child positions



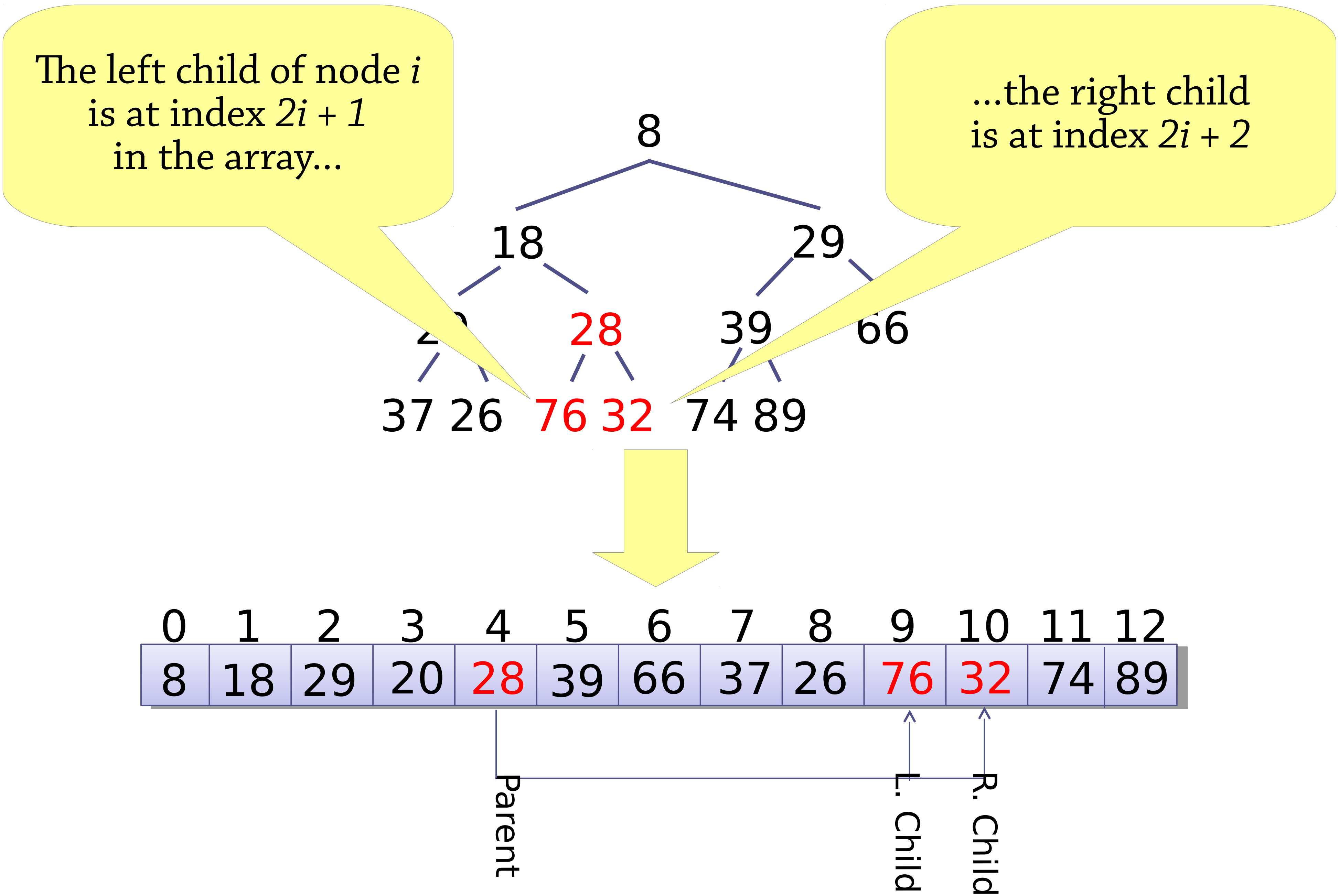
Child positions

The left child of node *i* is at index *2i* + *1* in the array...

...the right child is at index *2i* + *2*

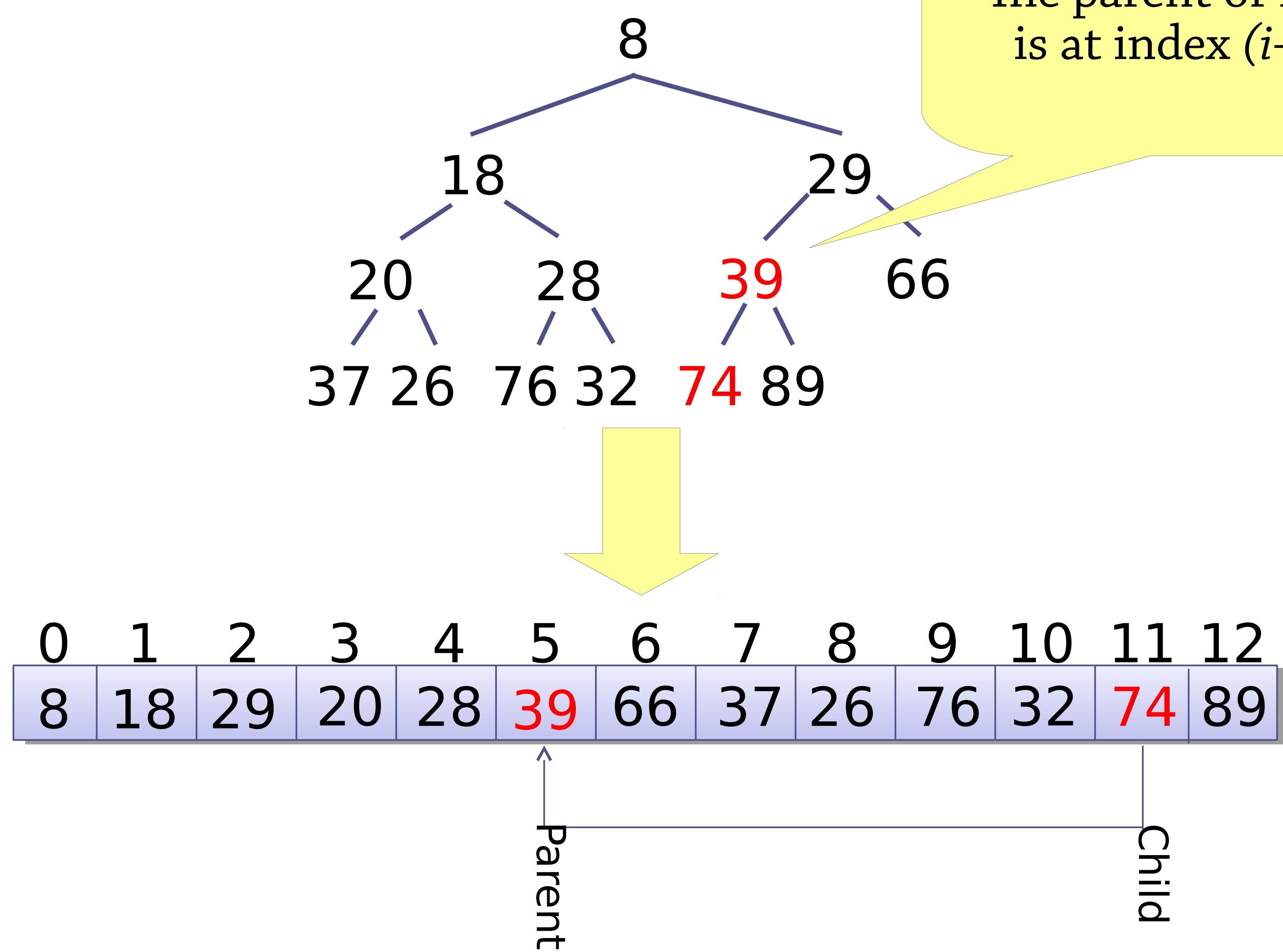


Child positions



Parent position

The parent of node *i* is at index *(i-1)/2*



Reminder: inserting into a binary heap

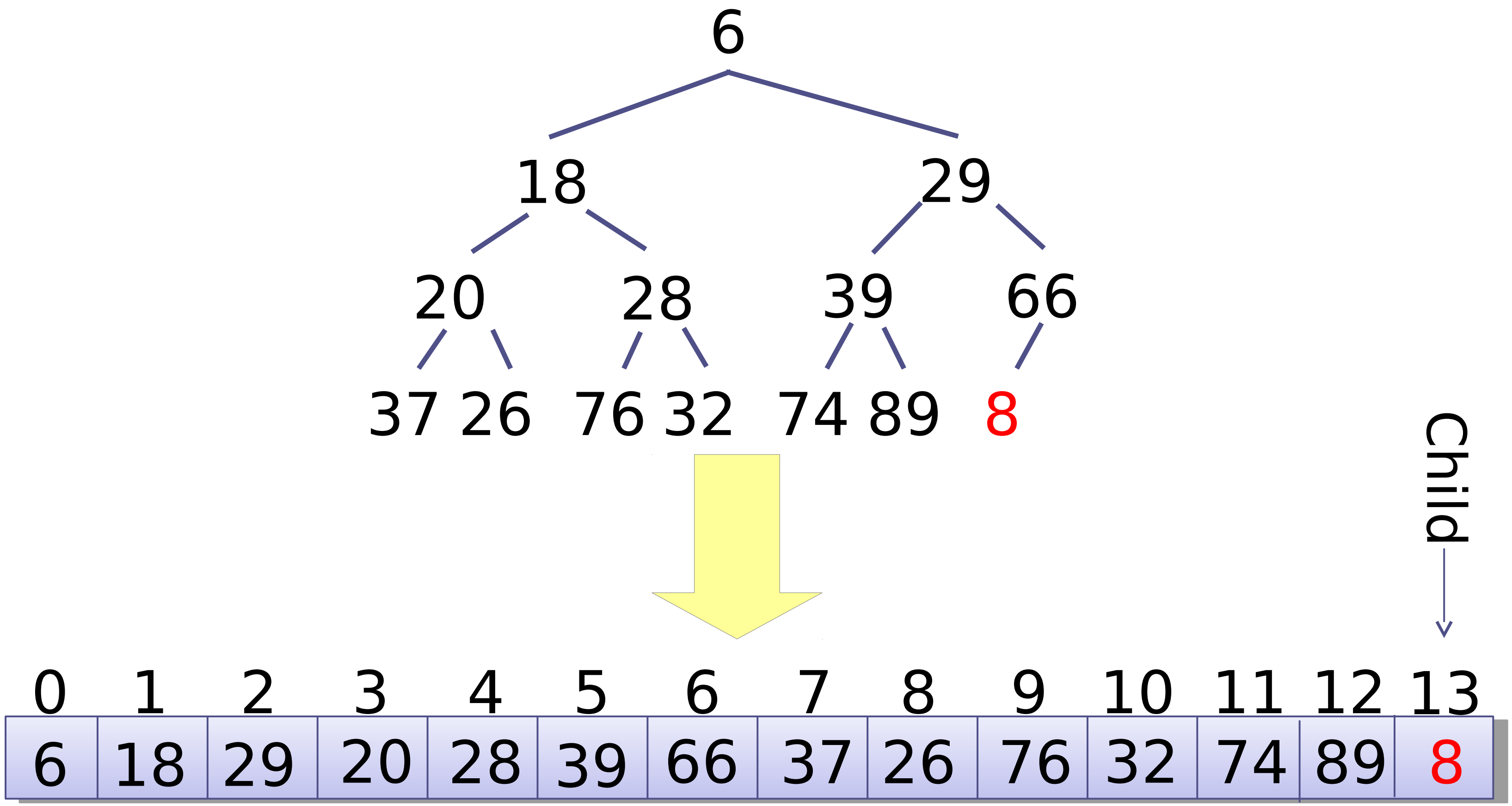
To insert an element into a binary heap:

- Add the new element at the end of the heap
- Sift the element up: while the element is less than its parent, swap it with its parent

We can do exactly the same thing for a binary heap represented as an array!

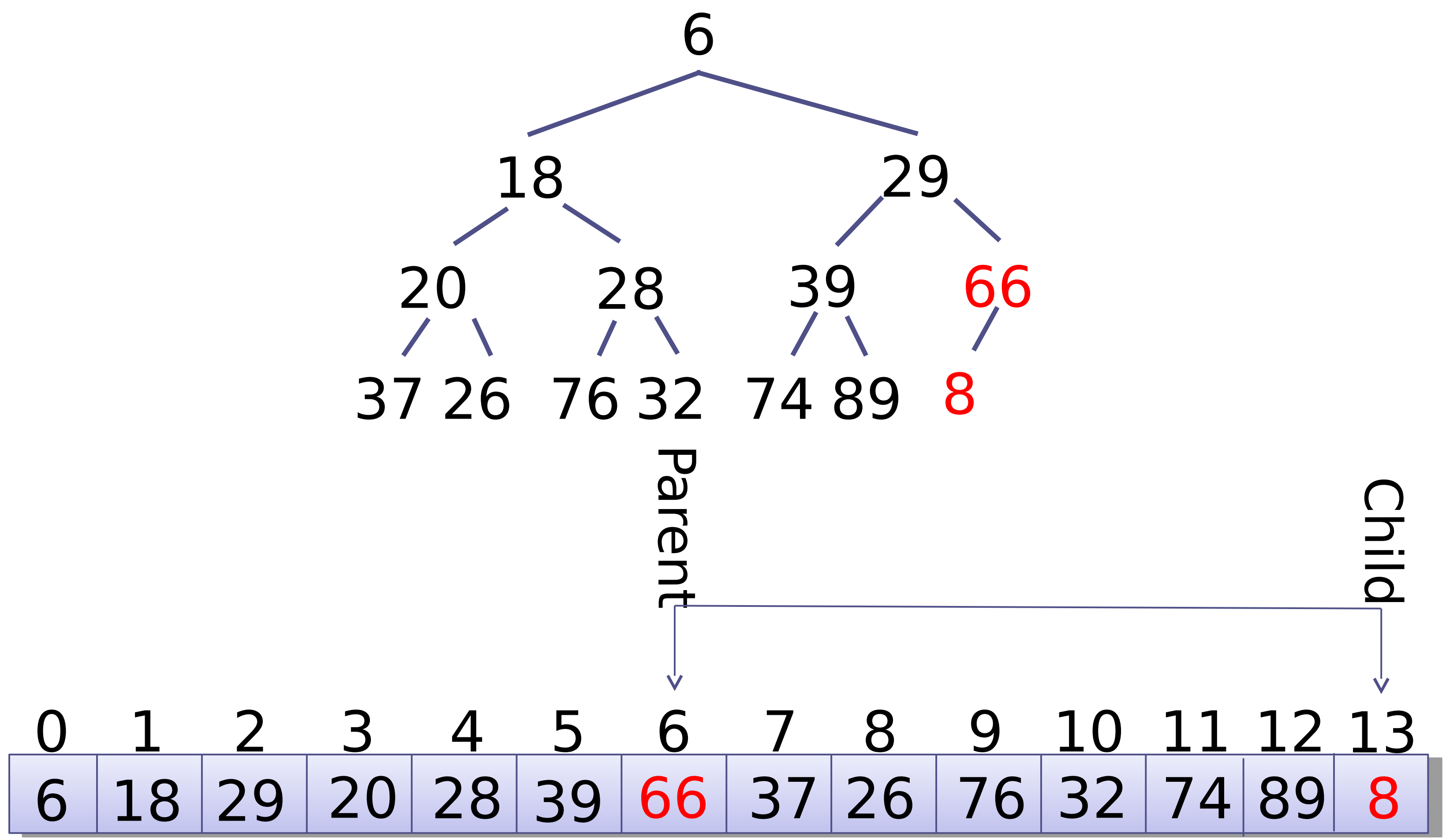
Inserting into a binary heap

Step 1: add the new element to the end of the array, set `child` to its index



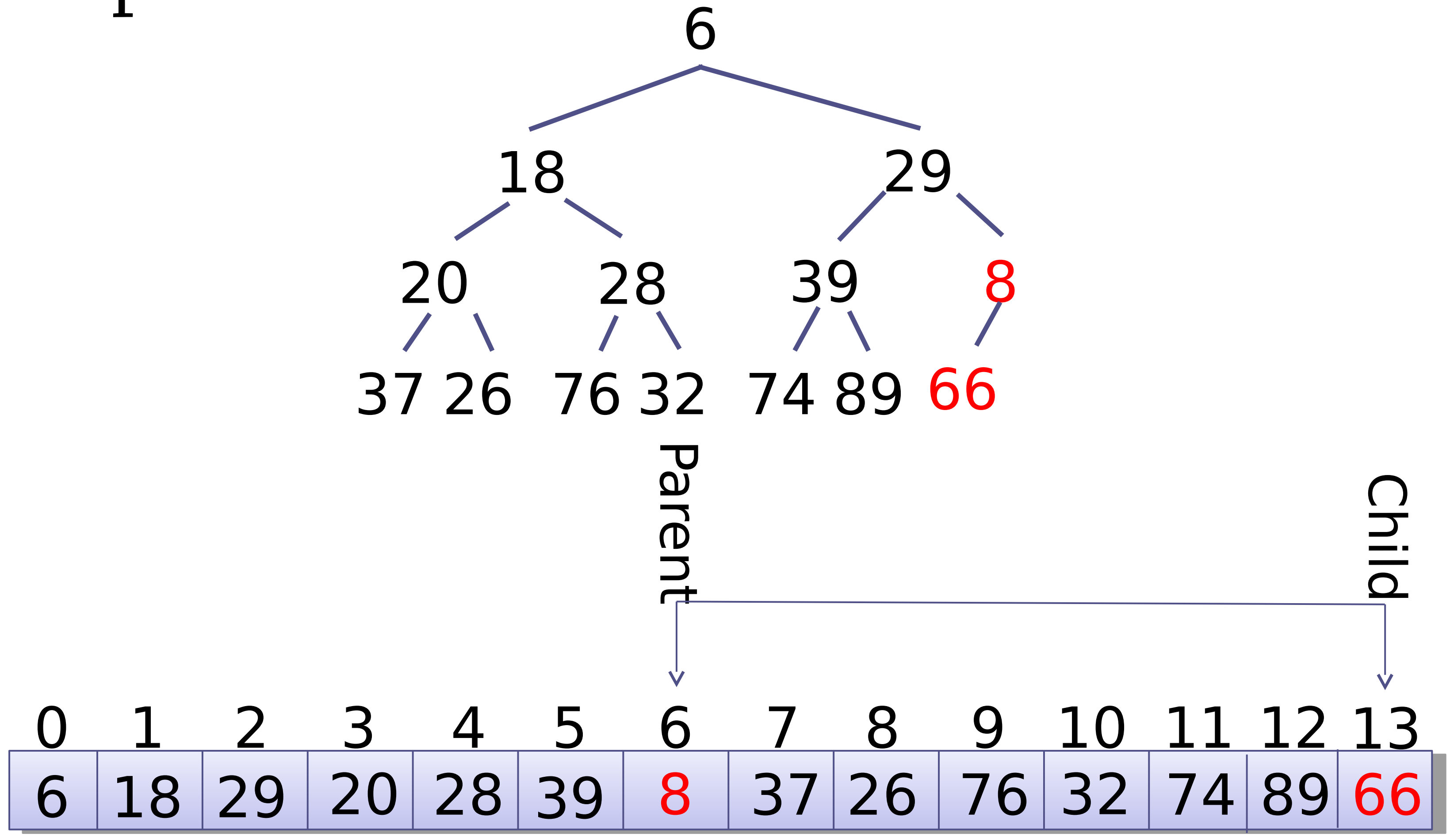
Inserting into a binary heap

Step 2: compute `parent = (child-1)/2`



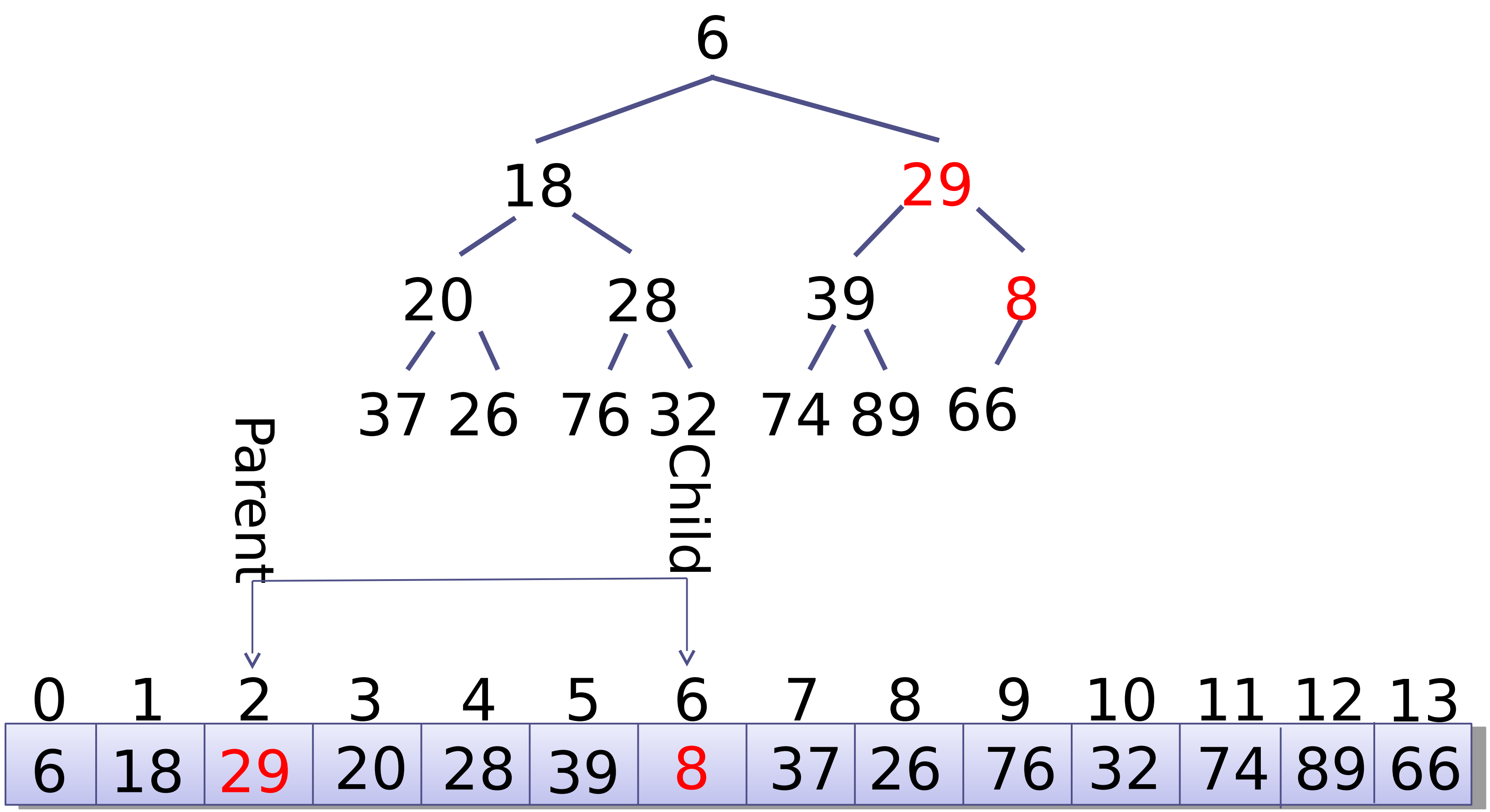
Inserting into a binary heap

Step 3: if `array[parent] > array[child]`, swap them



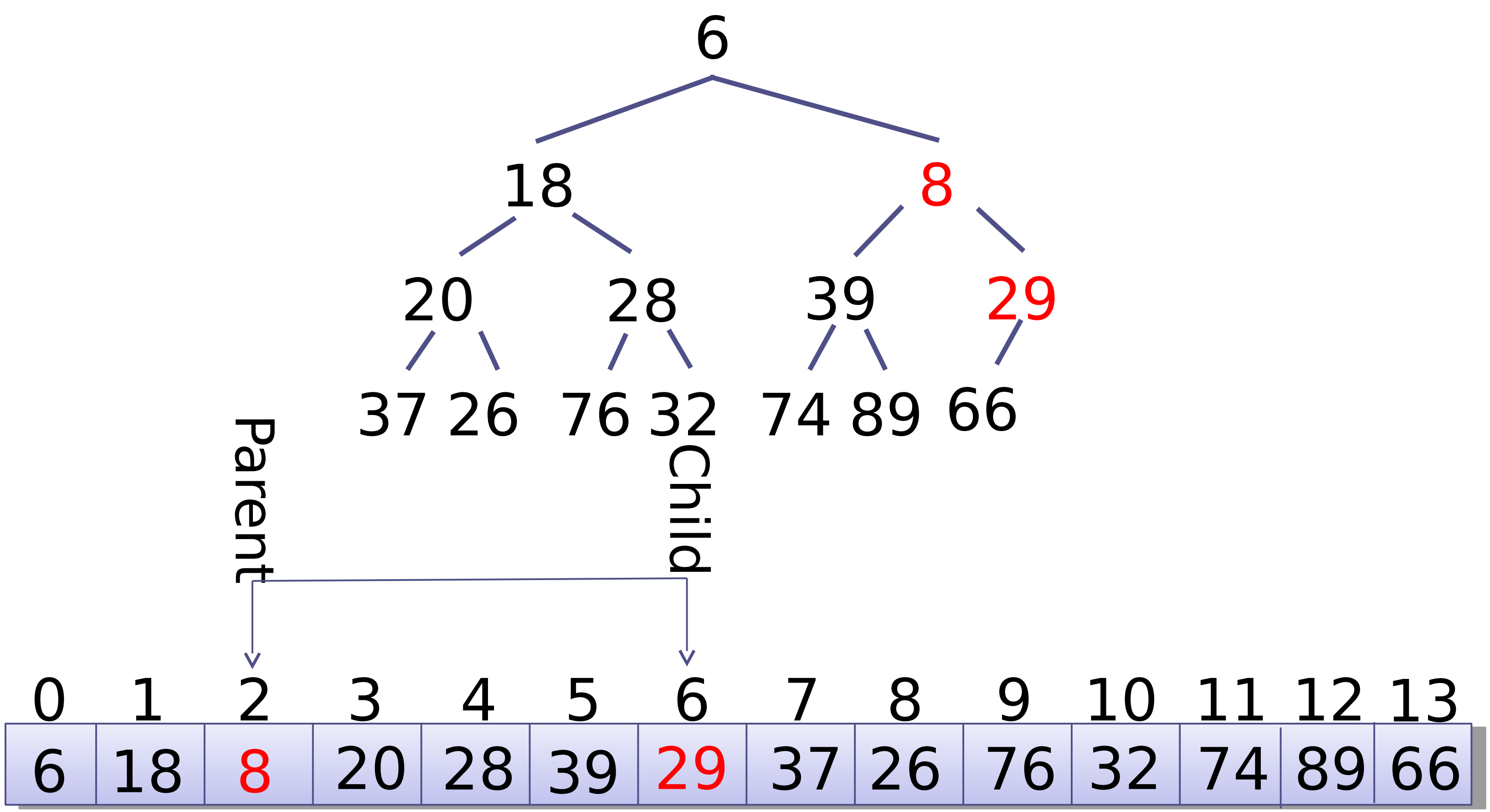
Inserting into a binary heap

Step 4: set `child = parent`, `parent = (child – 1) / 2`, and repeat



Inserting into a binary heap

Step 4: set `child = parent`, `parent = (child – 1) / 2`, and repeat



Binary heaps as arrays

Binary heaps are “morally” trees

- This is how we view them when we design the heap algorithms

But we implement the tree as an array

- The actual implementation translates these tree concepts to use arrays

When you see a binary heap shown as a tree, you should also keep the array view in your head (and vice versa!)

Building a heap

One more operation, *build heap*

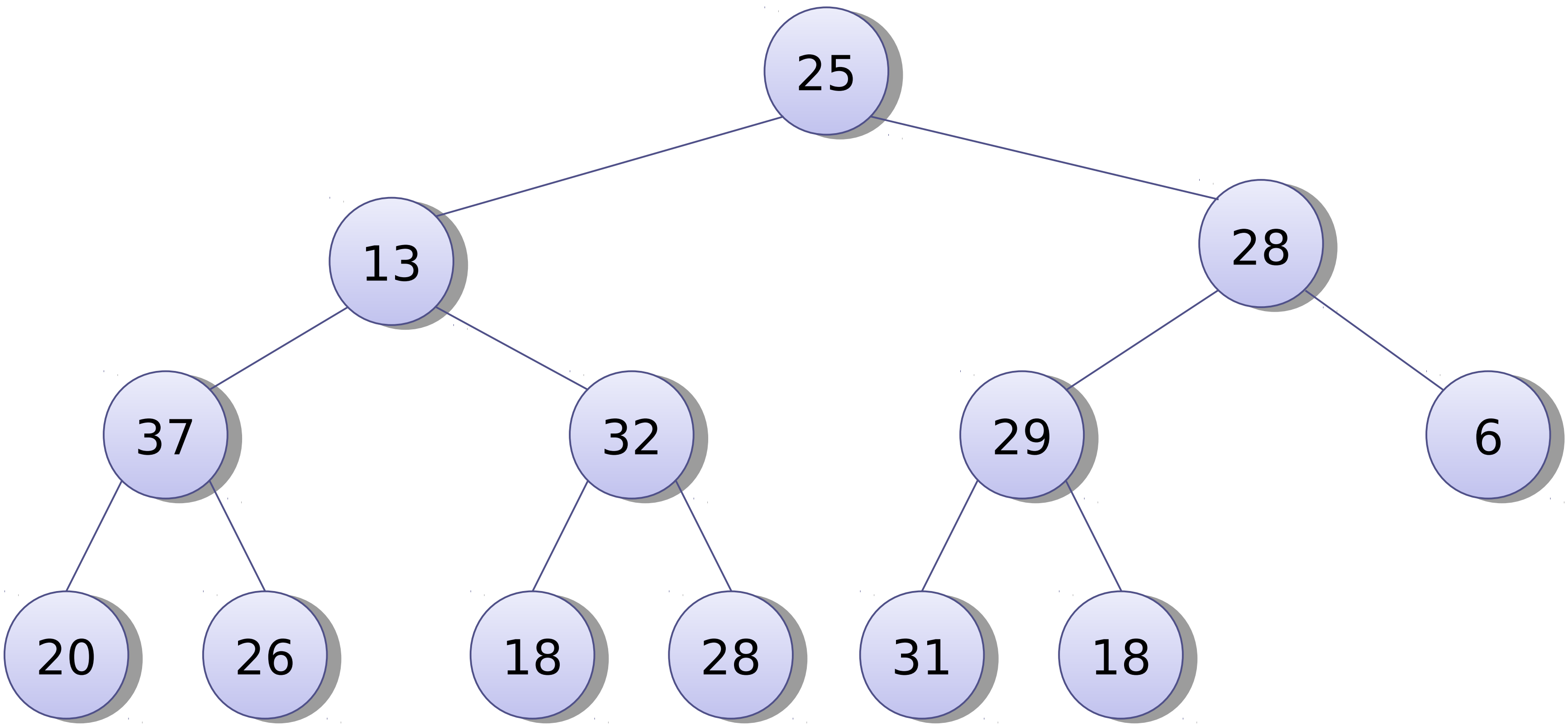
- Takes an arbitrary array and makes it into a heap
- In-place: moves the elements around to make the heap property hold

Idea: use *sifting down* (see next slide)

- Sift down each node, starting at the leaves and working up to the root
- By the time we sift down a node, its children will already be heaps
- Sifting down makes the node itself into a heap

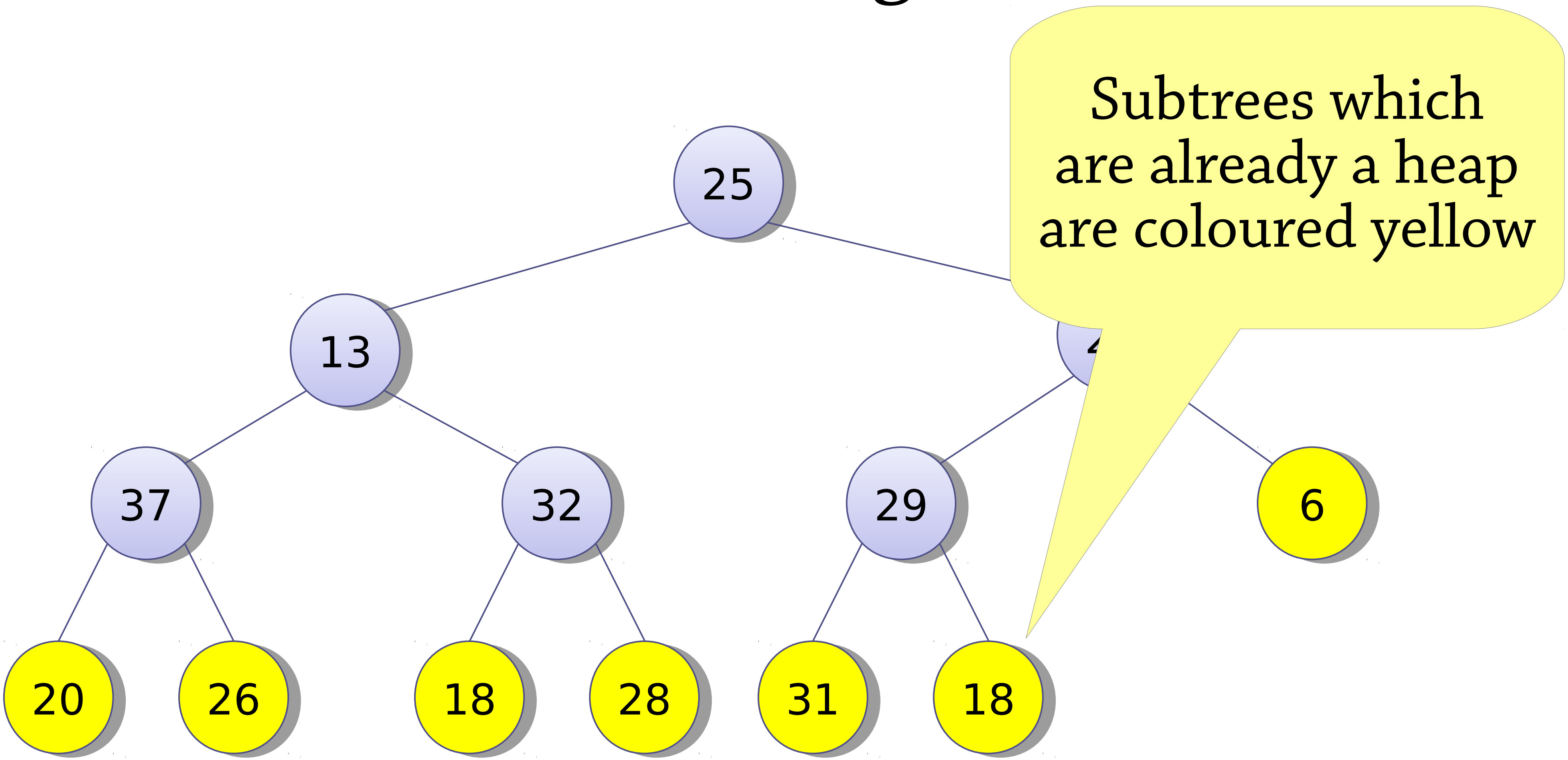
Building a heap

Go through elements in reverse order, sifting each down



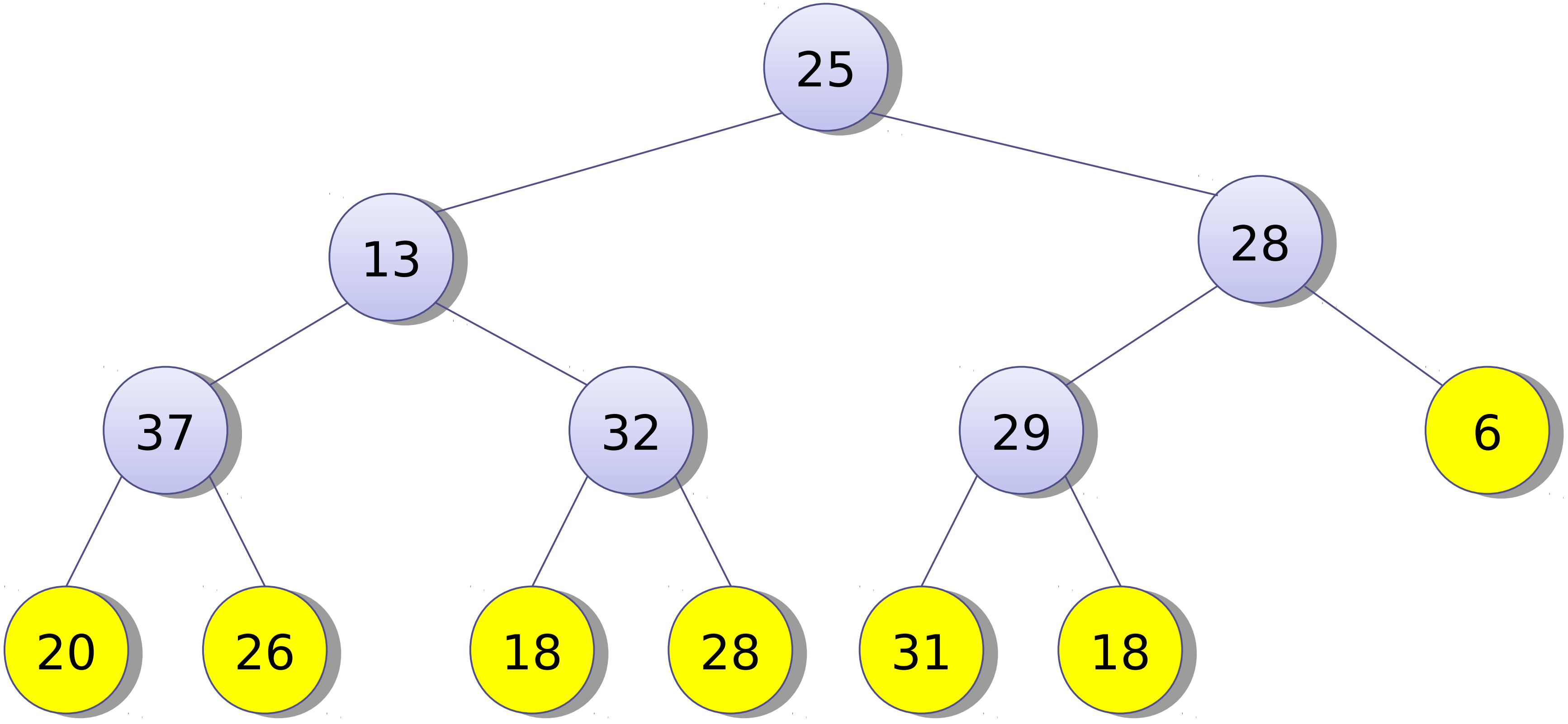
Building a heap

Leaves never need sifting down!

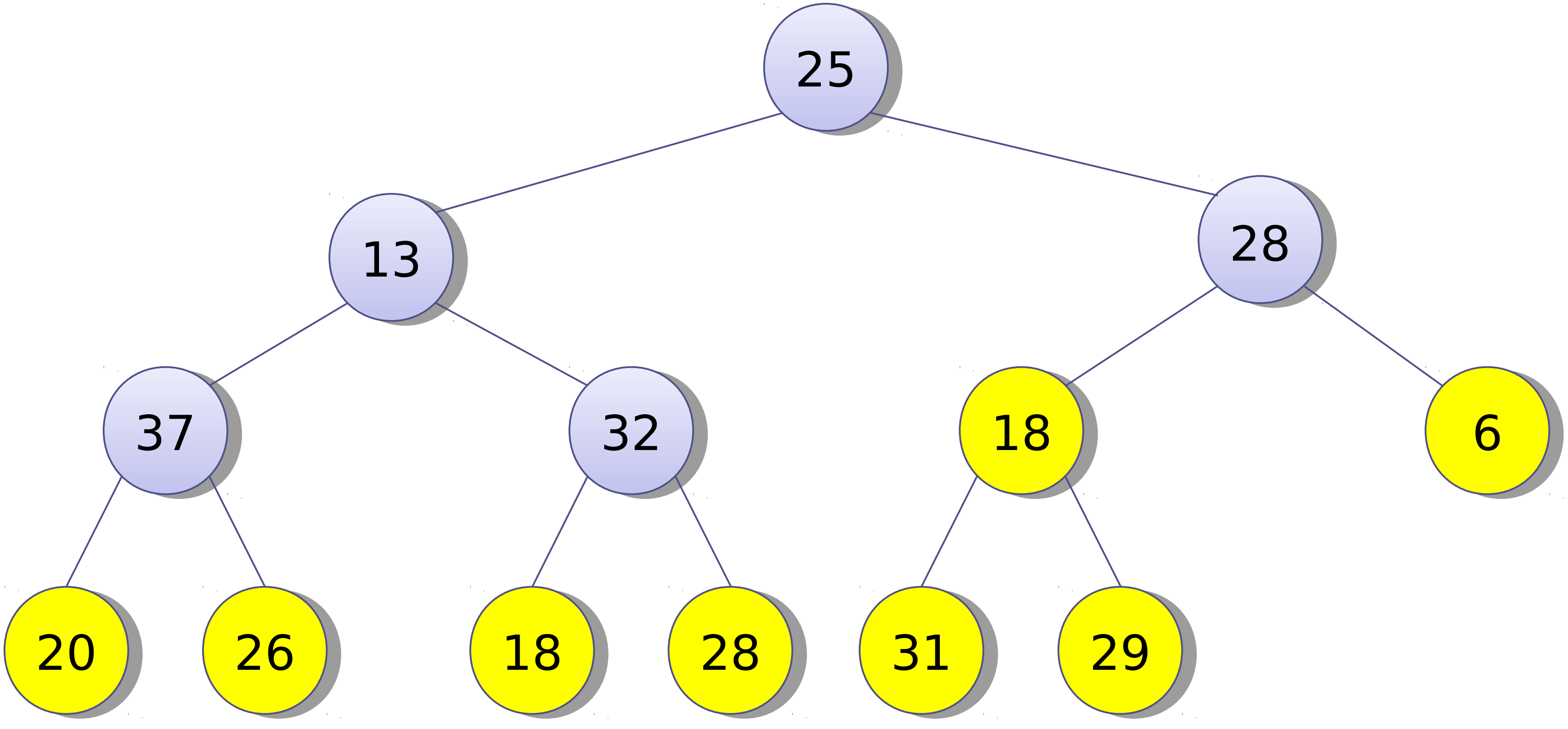


Building a heap

Sift down 29: swap it with 18

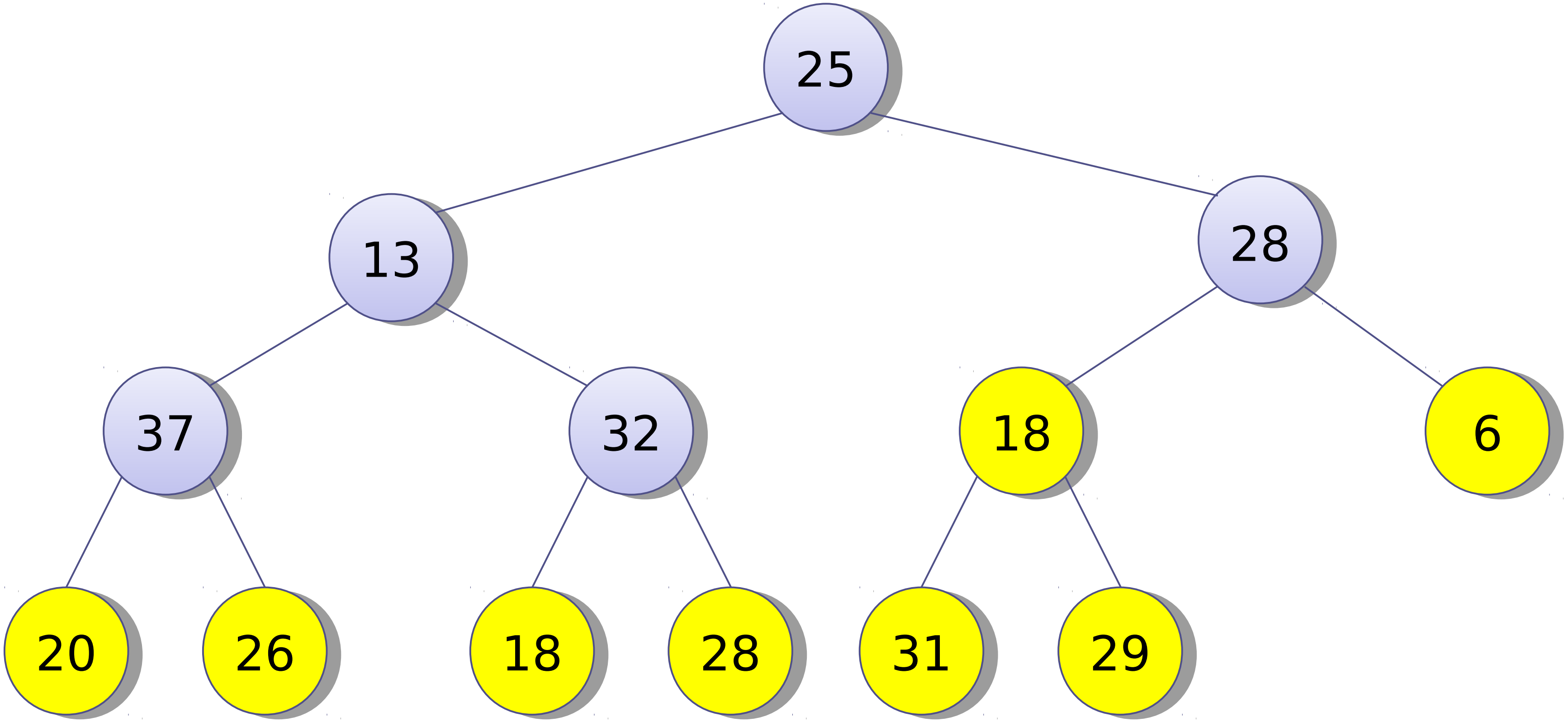


Building a heap

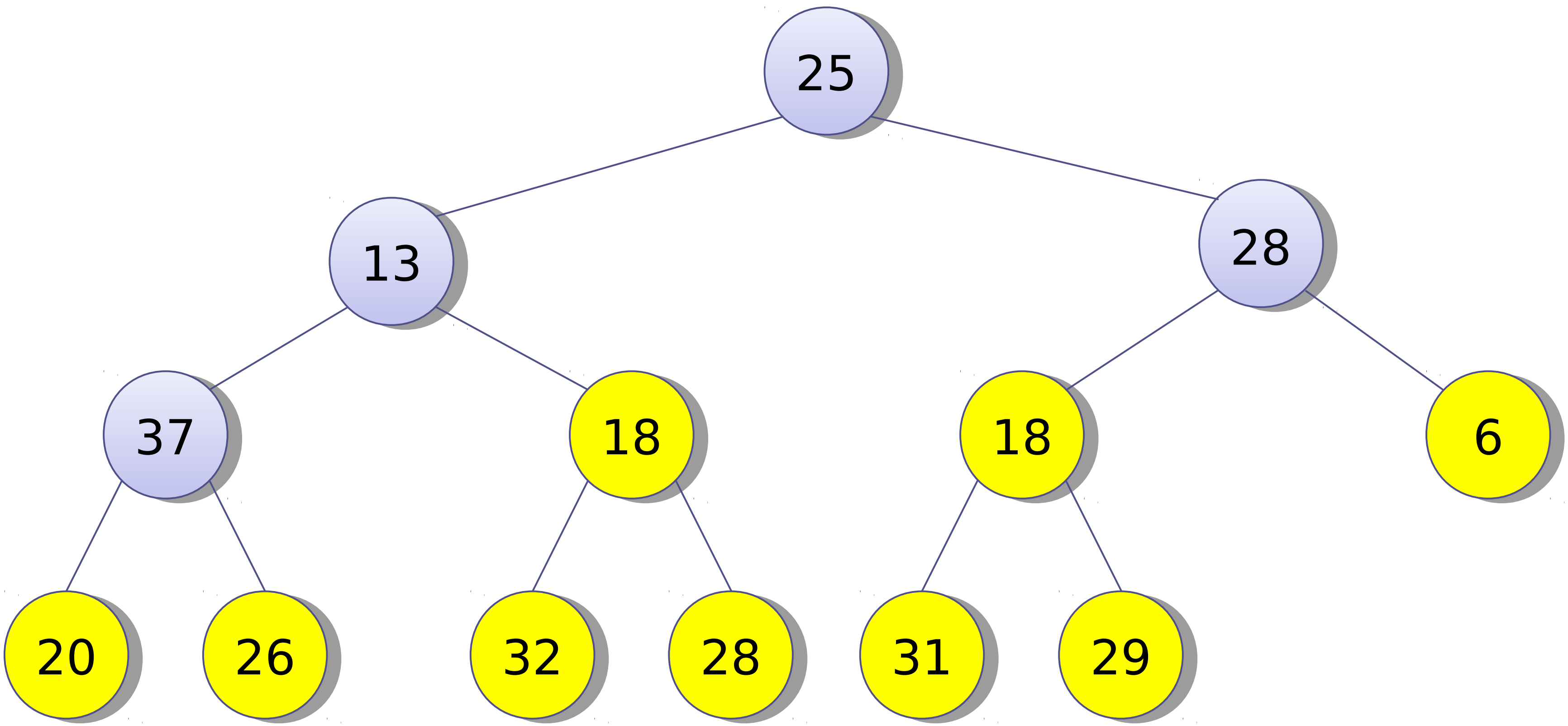


Building a heap

Sift down 32: swap it with 18

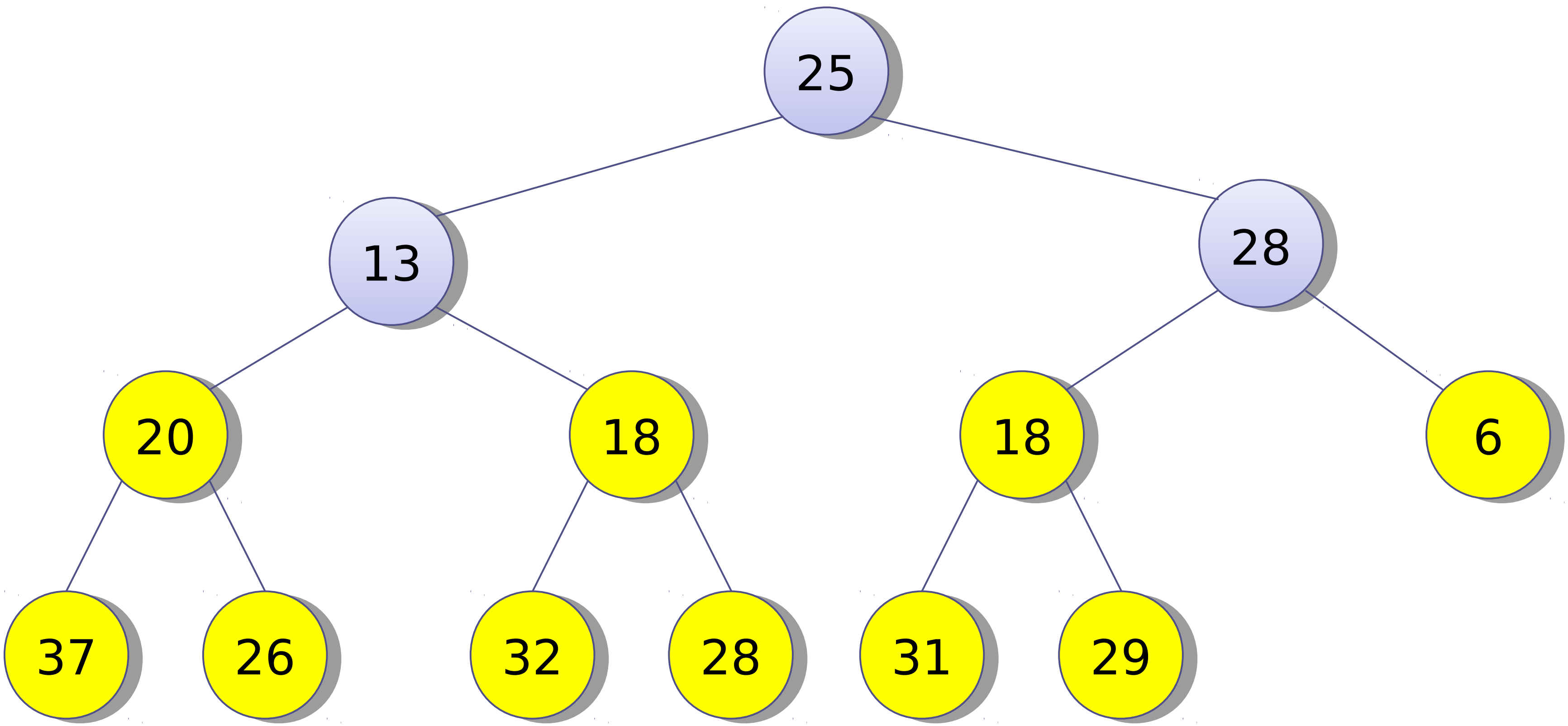


Building a heap



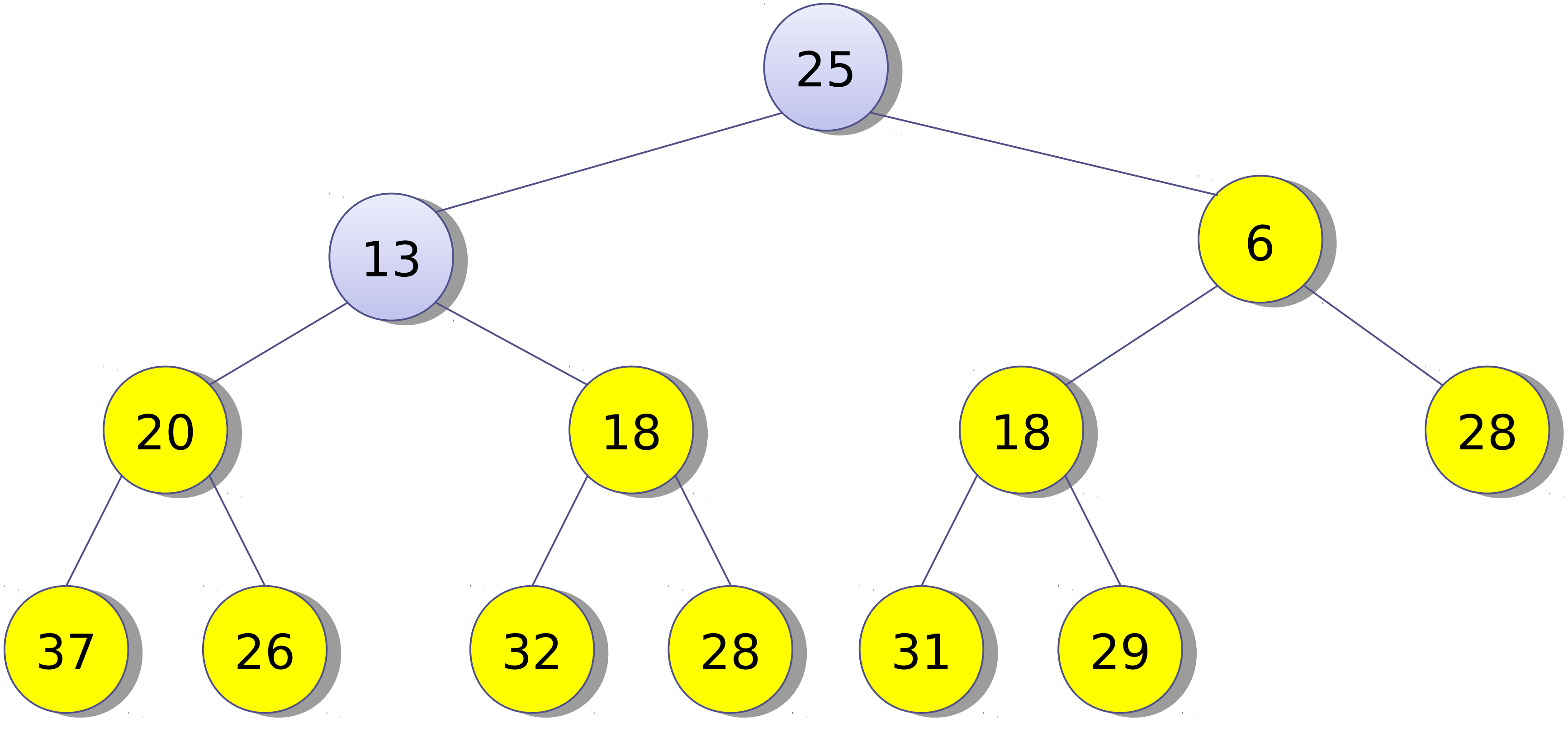
Building a heap

Swap 37 and 20



Building a heap

Swap 28 and 6



Build heap complexity

You would expect O(n log n) complexity:

- n "sift down" operations
- each sift down has O(log n) complexity because the height of the tree is at most log n

Actually, it's O(n)! See book 20.3.

- (Rough reason: sifting down is most expensive for elements near the root of the tree, but the vast majority of elements are near the leaves)

Heapsort

To sort a list using a heap:

- start with an empty heap
- add all the list elements in turn
- repeatedly find and remove the smallest element from the heap, and add it to the result list

(this is a kind of *selection sort*)

However, this algorithm is not in-place. Heapsort uses the same idea, but without allocating any extra memory.

Heapsort, in-place

We're going to use a *max heap*

- This is a heap where you can find and delete the *maximum* element instead of the minimum
- Implementation is exactly the same as a normal (min) heap, just the order of all comparisons is reversed

Step 1: turn the array into a max heap, in-place

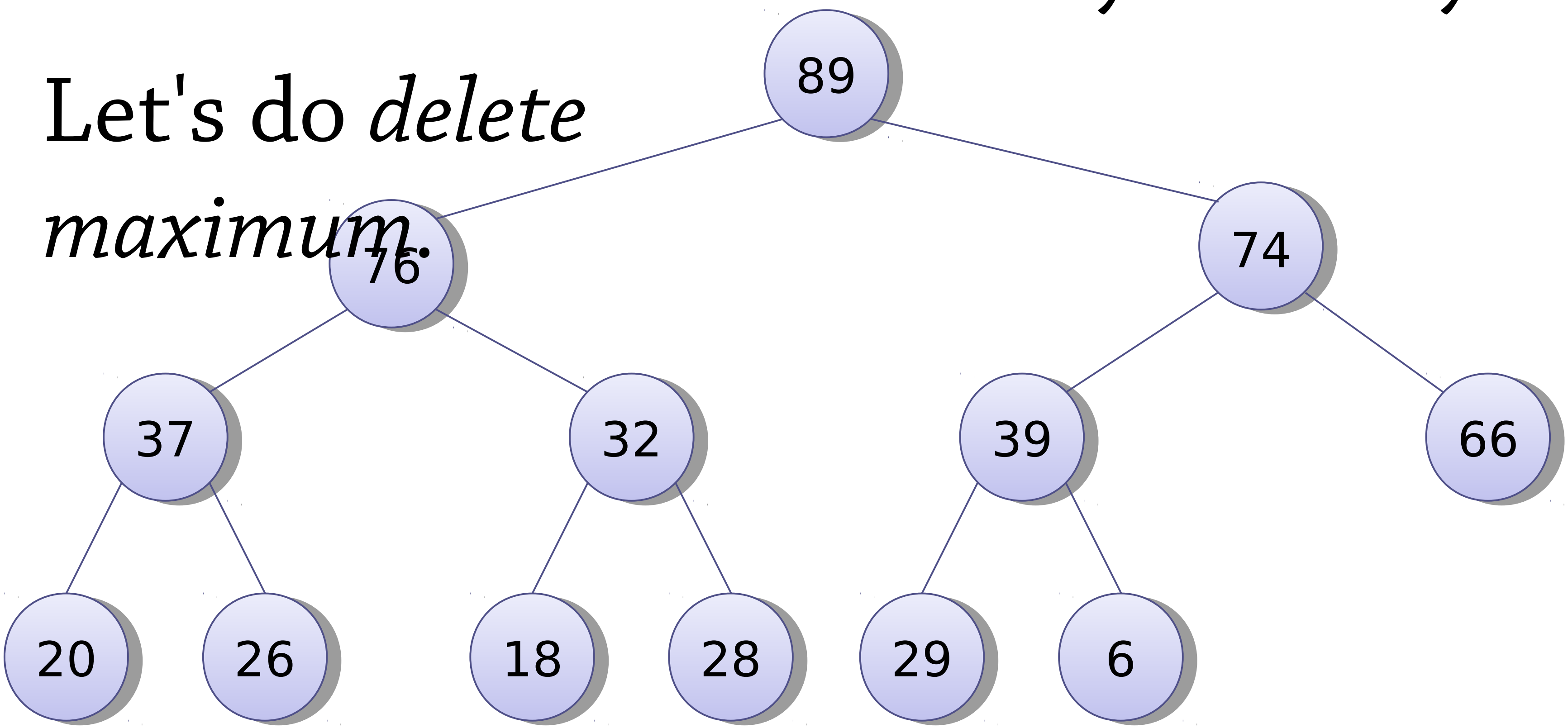
- using *build heap* algorithm

Step 2: let's see!

Idea of heapsort

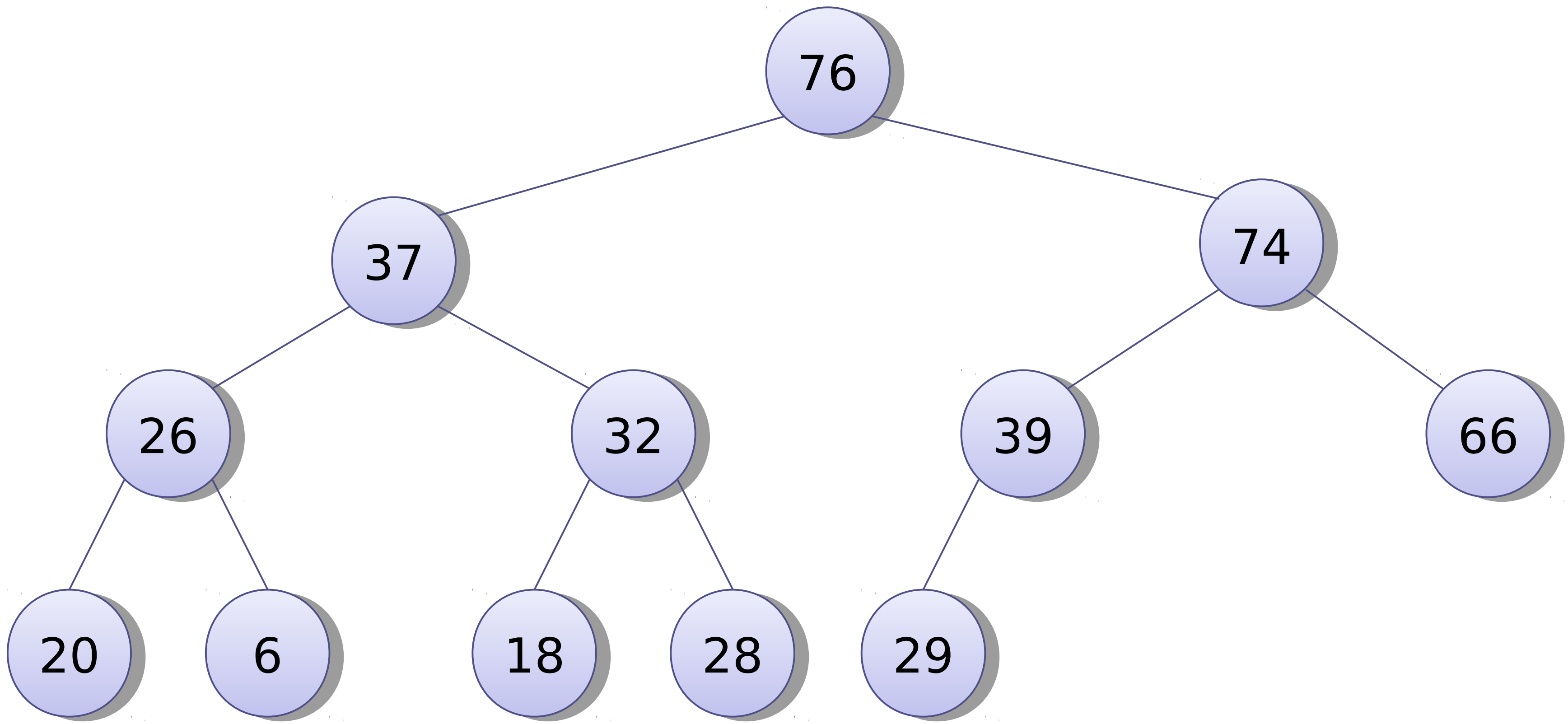
Here is our max heap.
Bear in mind that it is really an array!

Let's do *delete maximum*.



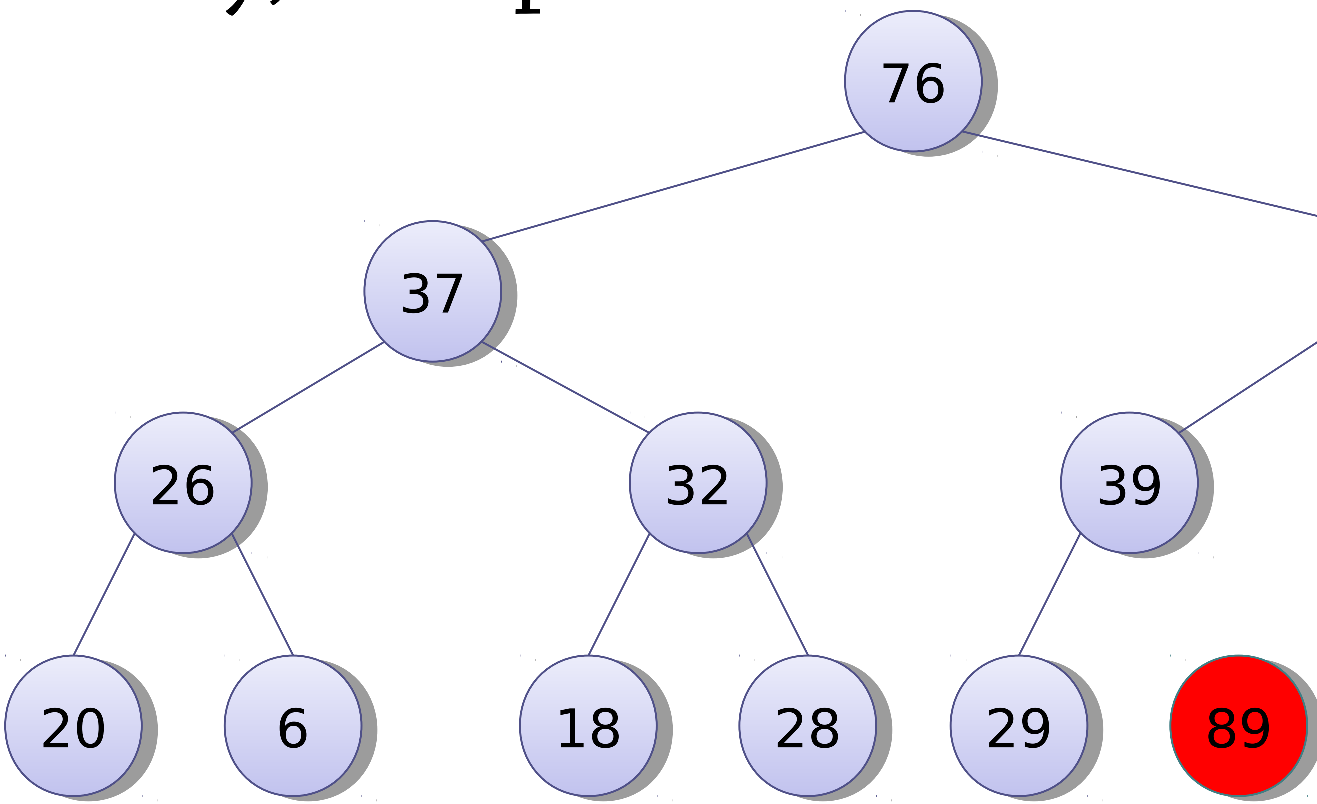
Idea of heapsort

We've deleted 89. The array is now one element shorter.



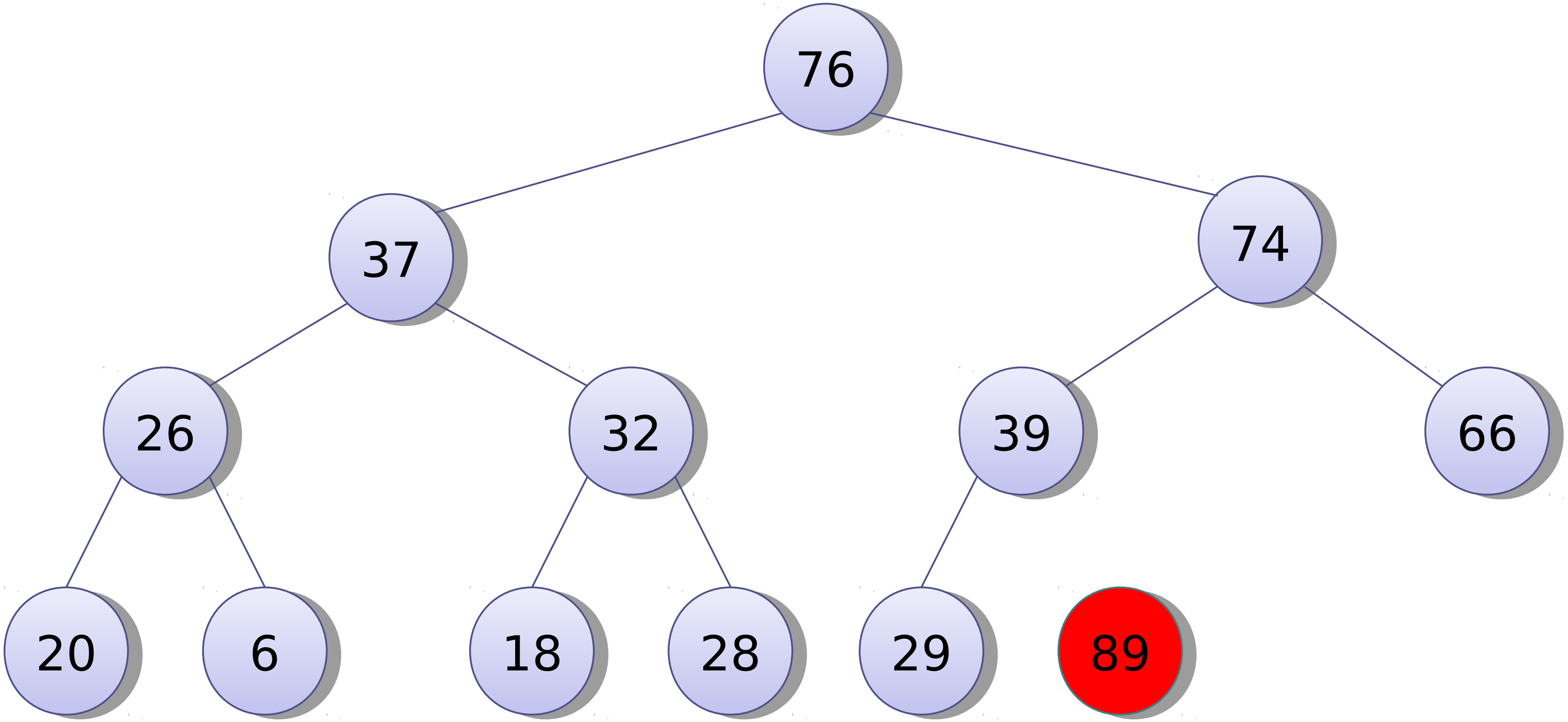
Idea of heapsort

There's an unused space at the end of the array, let's put 89 there!



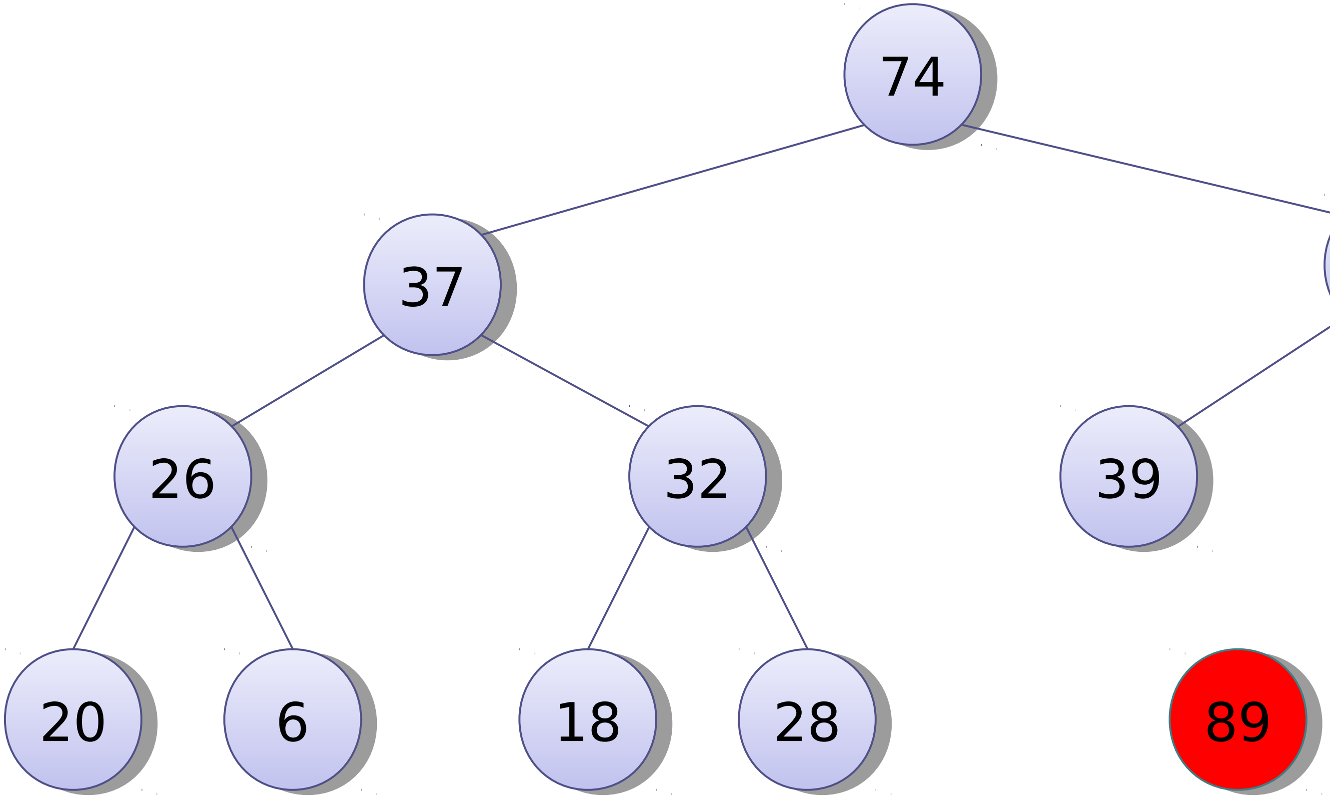
Idea of heapsort

Next step: delete maximum again.

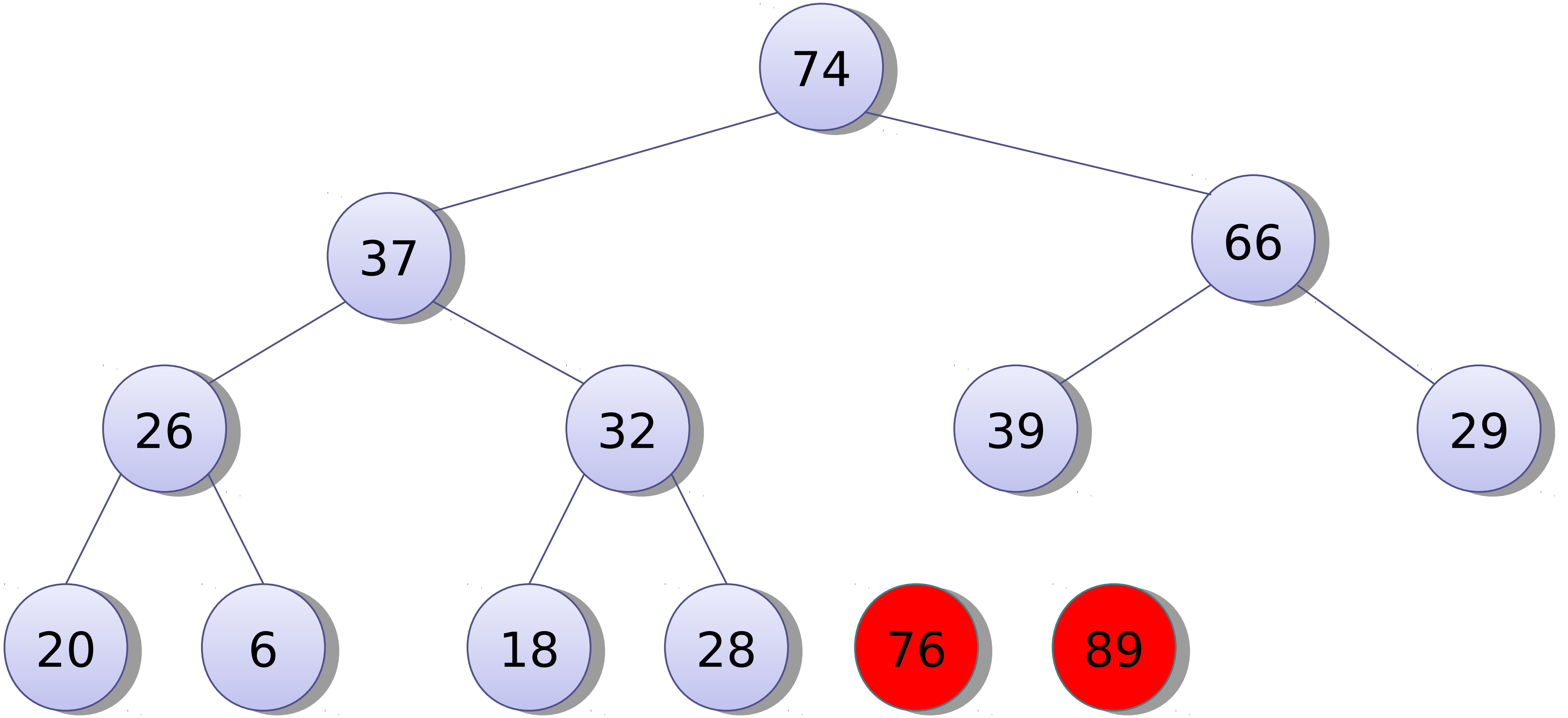


Idea of heapsort

Again there's an empty space, let's put 76 there!



Idea of heapsort



Heapsort – summary

Take an array

Step 1: turn it into a max heap, in-place

Step 2:

- Delete the maximum element
- Put that element in the newly-available space at the end of the array

Repeat until heap is empty!

Complexity of heapsort

Building the heap takes O(n) time

We delete the maximum element n times, each deletion taking O(log n) time

Hence the total complexity is O(n log n)

Warning

Our formulas for finding children and parents in the array assume 0-based arrays

The book, for some reason, uses 1-based arrays (and later switches to 0-based arrays)!

In a heap implemented using a 1-based array:

- the left child of index *i* is index *2i*
- the right child is index *2i+1*
- the parent is index *i/2*

Be careful when doing the lab!

Summary of binary heaps

Binary heaps: O(log n) insert, O(1) find minimum, O(log n) delete minimum

- A complete binary tree with the heap property, represented as an array

Heapsort: in place sorting algorithm based on heaps, takes O(n log n) time

In fact, heaps were originally invented *for* heapsort!

Leftist heaps

Merging two heaps

Another operation we might want to do is *merge* two heaps

- Build a new heap with the contents of *both* heaps
- e.g., merging a heap containing 1, 2, 8, 9, 10 and a heap containing 3, 4, 5, 6, 7 gives a heap containing 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

For our earlier naïve priority queues:

- An unsorted array: concatenate the arrays
- A sorted array: merge the arrays (as in mergesort)

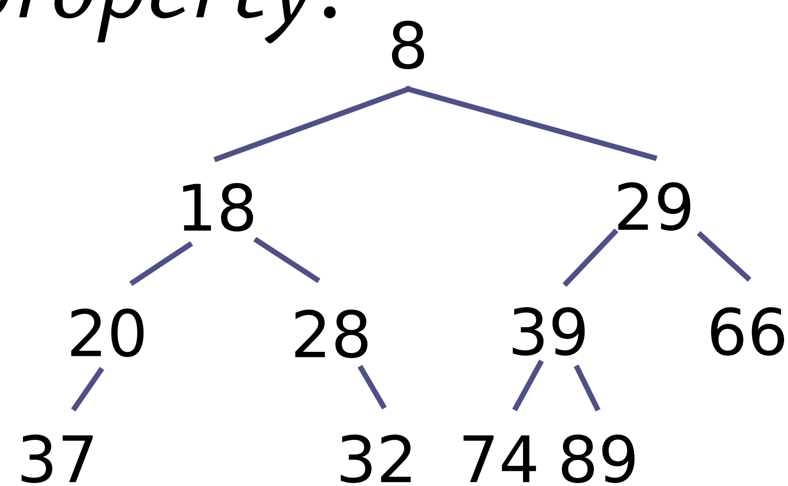
For binary heaps:

- Takes O(n) time because you need to at least copy the contents of one heap to the other

Can't combine two arrays in less than O(n) time!

Merging tree-based heaps

Go back to our idea of *a binary tree with the heap property*:



If we can merge two of these trees, we can implement insertion and delete minimum!

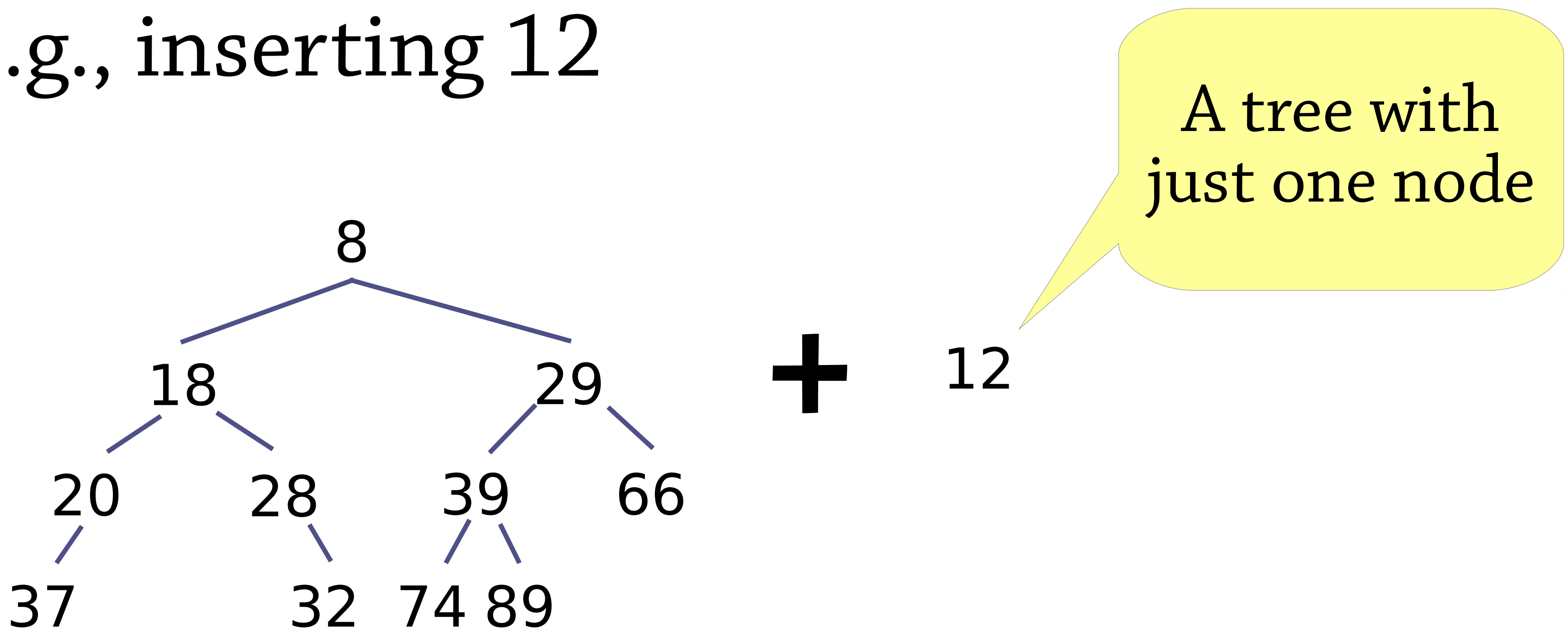
(We'll see how to implement merge later)

Insertion

To insert a single element:

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

E.g., inserting 12

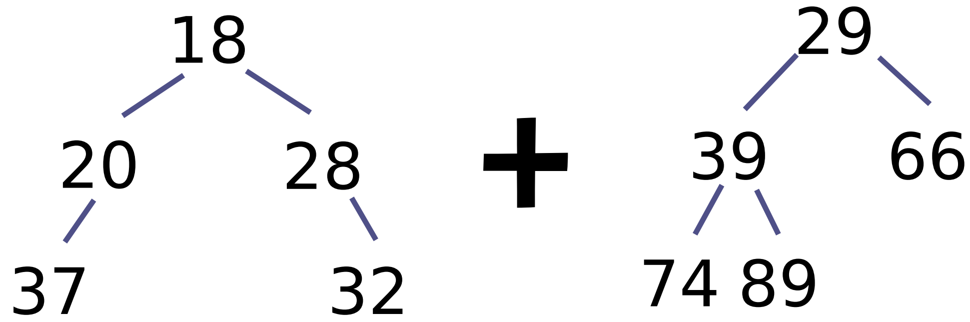


Delete minimum

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 based on merging

If we can take *trees with the heap property*, and implement merging with O(log n) complexity, we get a priority queue with:

- O(1) find minimum
- O(log n) insertion (by merging)
- O(log n) delete minimum (by merging)
- plus this useful merge operation itself

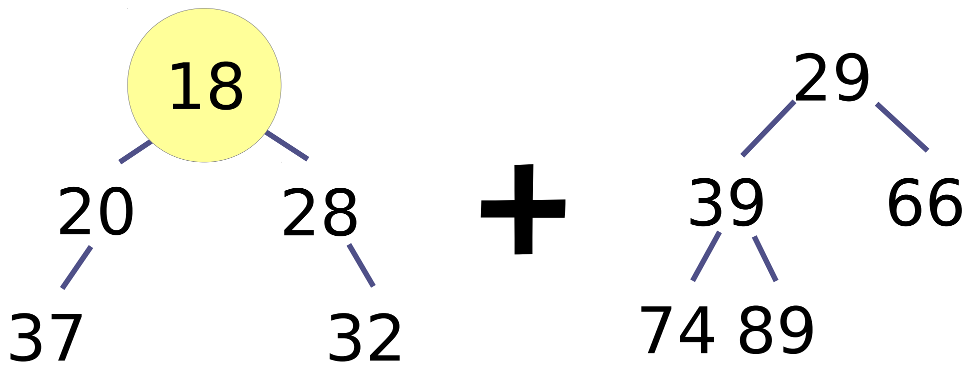
There are lots of heaps based on this idea:

- skew heaps, Fibonacci heaps, binomial heaps

We will study one: *leftist heaps*

Naive merging

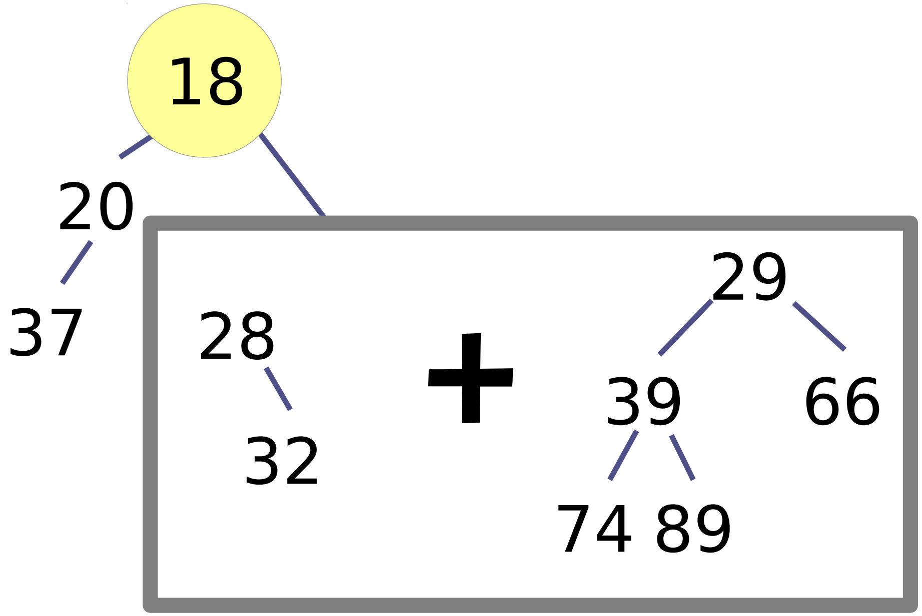
1. Look at the roots of the two trees



We are going to pick the smaller one as the root of the new tree

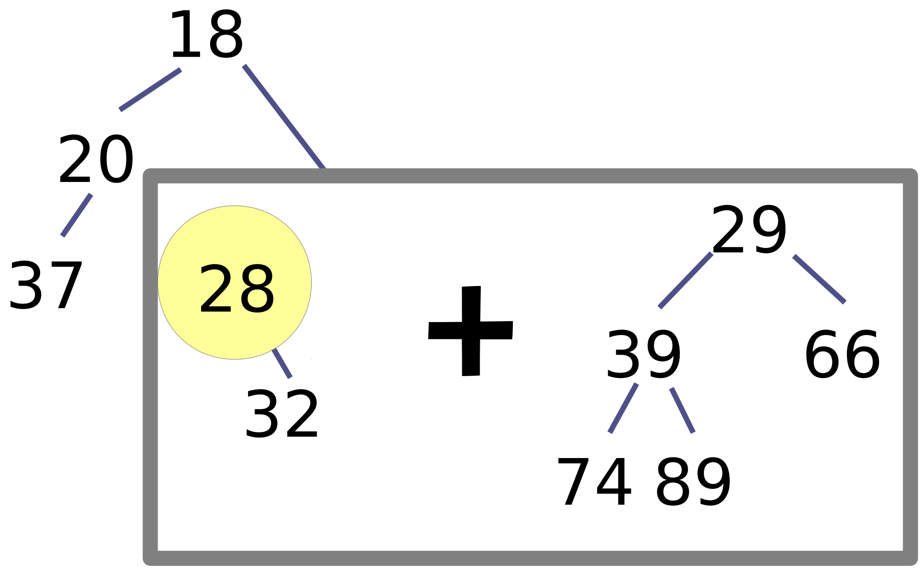
Naive merging

2. *Recursively merge* the right branch and the second tree



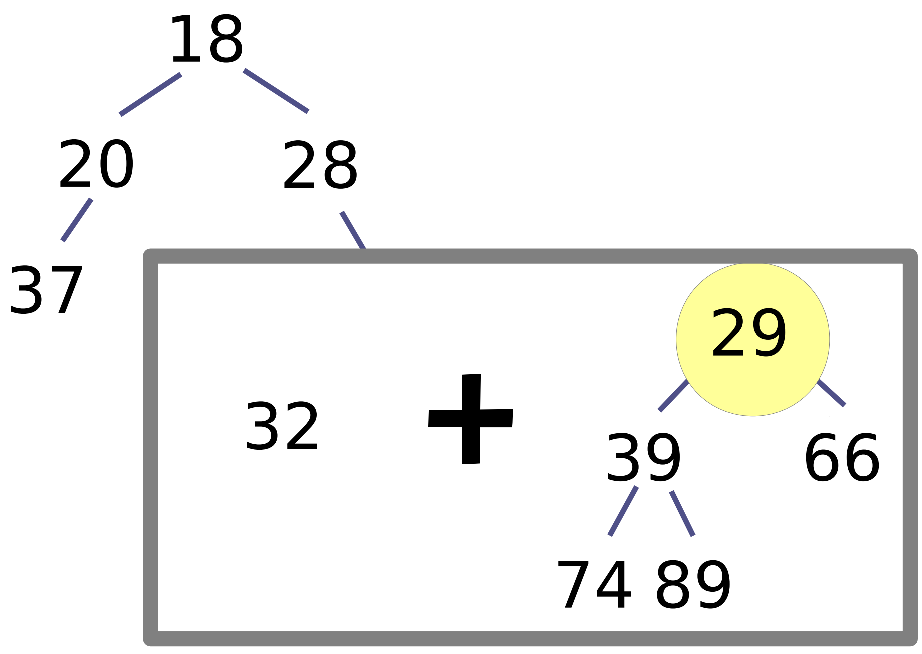
Naive merging

2. *Recursively merge* the right branch and the second tree



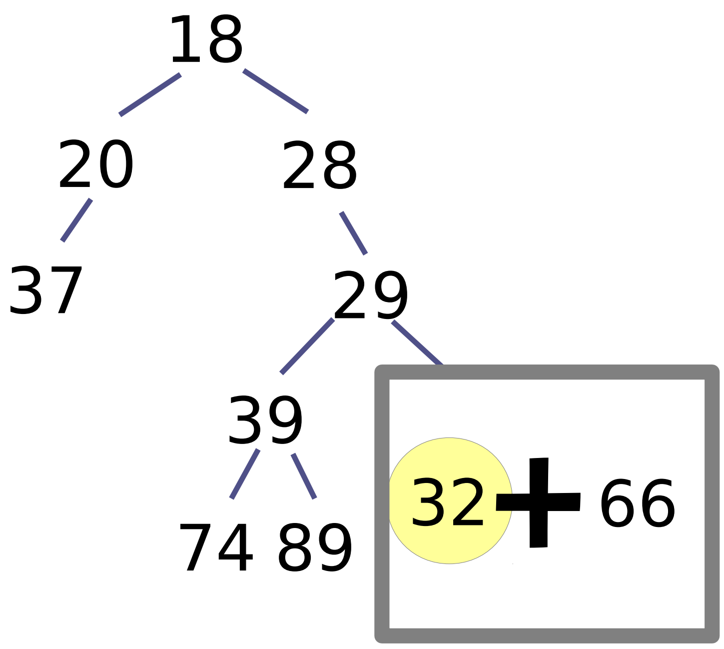
Naive merging

2. *Recursively merge* the right branch and the second tree



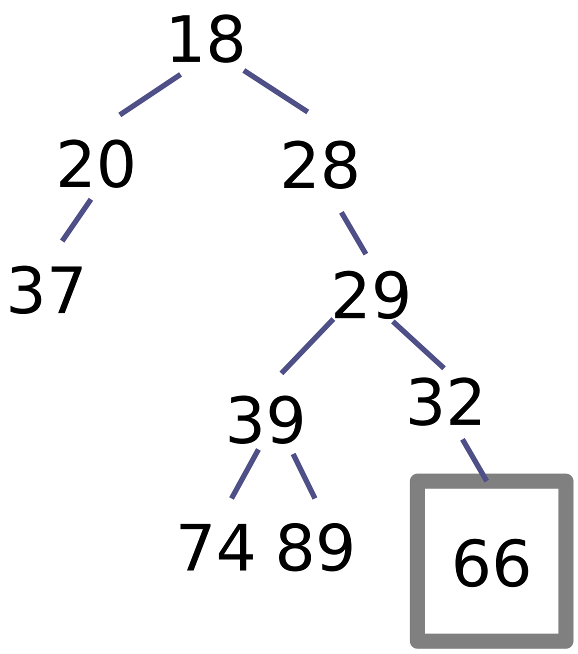
Naive merging

2. *Recursively merge* the right branch and the second tree



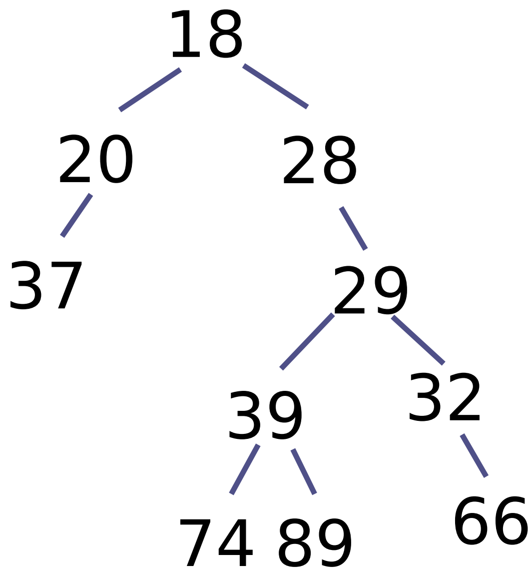
Naive merging

2. *Recursively merge* the right branch and the second tree



Naive merging

2. *Recursively merge* the right branch and the second tree



Performance of naïve merging

The merge algorithm descends down the *right branch* of both trees

So the runtime depends on *how many times you can follow the right branch before you get to the end of the tree*

- Let's call this the *right null path length*

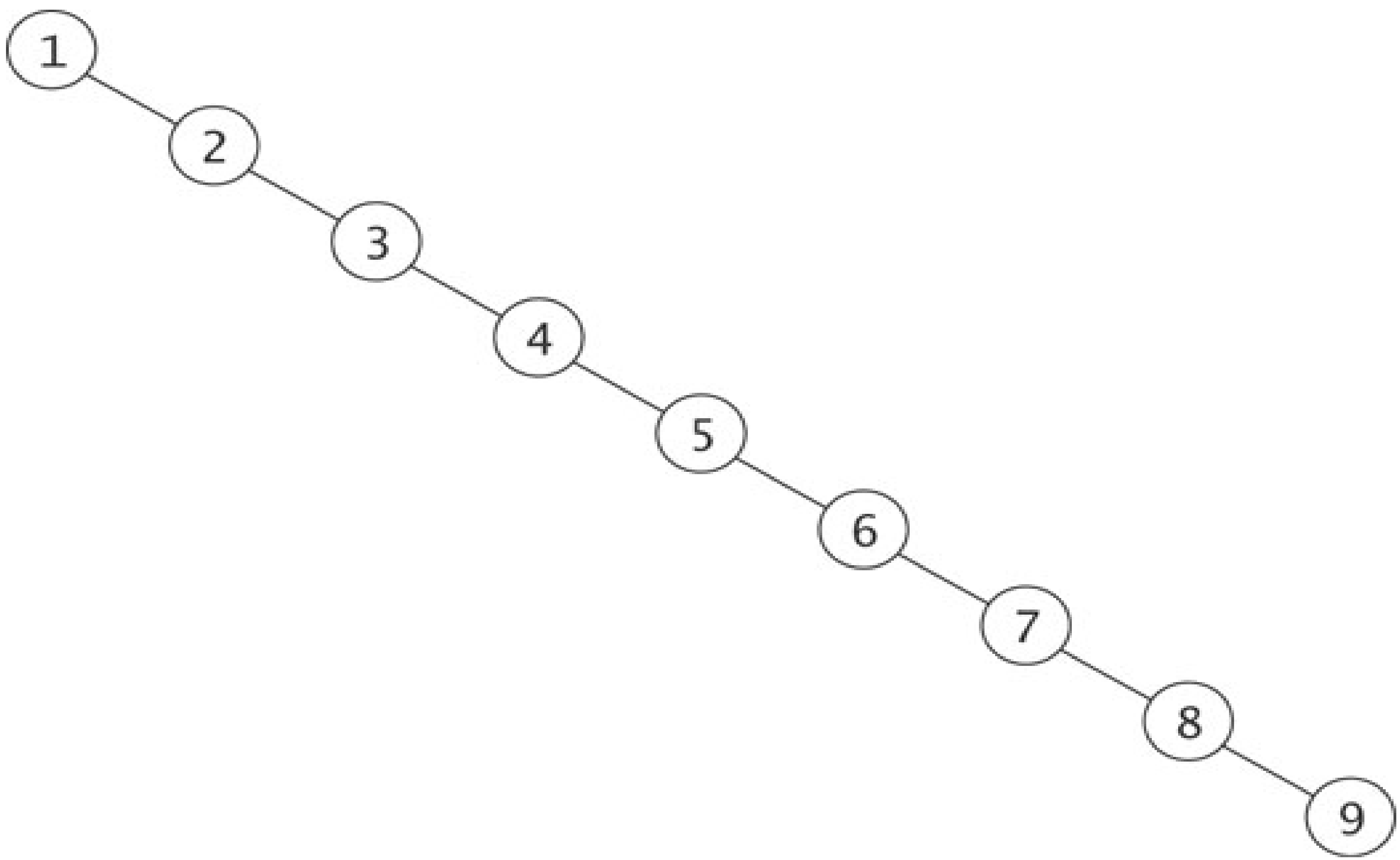
Complexity: O(m+n)

- where m and n are the right null path lengths of the two trees

Logarithmic complexity for balanced trees, but linear if the trees are heavily "right-biased"

Worst case for naïve merging

A heavily right-biased tree:



Leftist heaps – observation

Naive merging is:

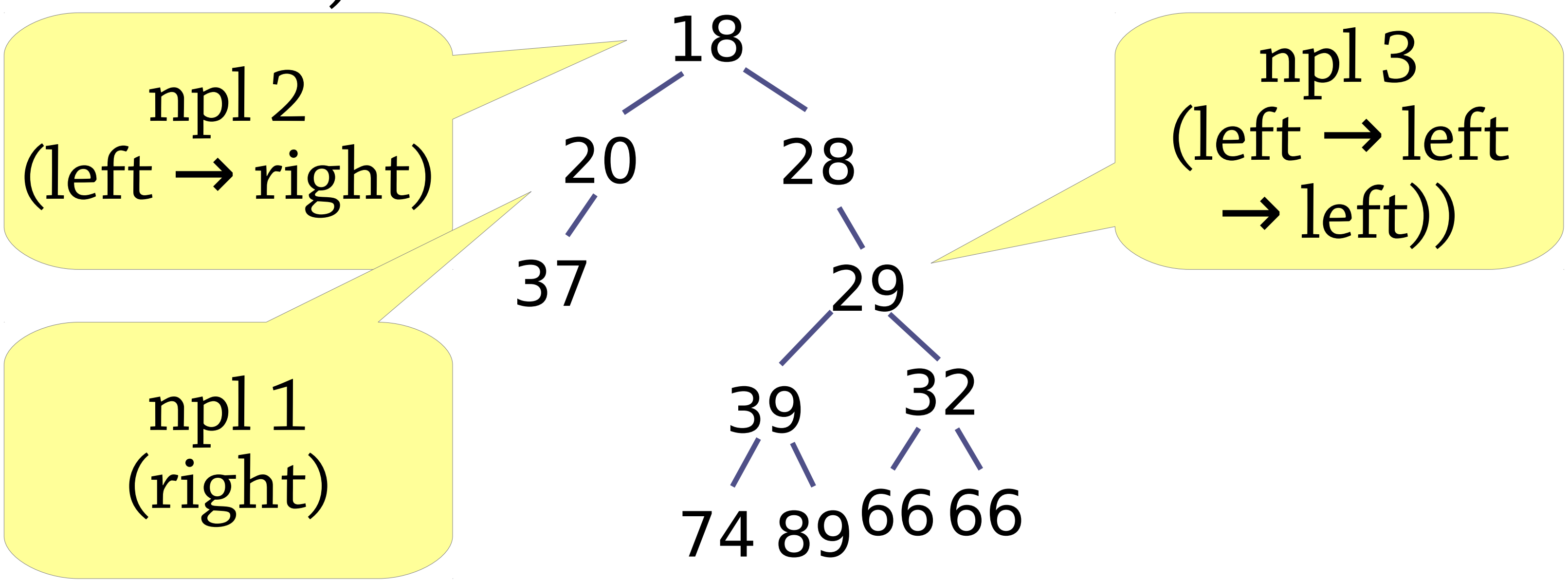
- bad (linear complexity) for right-biased trees
- good (logarithmic or better) for other trees

Idea of leftist heaps:

- Add an invariant that stops the tree becoming right-biased
- In other words, by repeatedly following the right branch, you quickly reach the end of the tree

Null path length

We define the *null path length (npl)* of a node to be the shortest path that leads to the end of the tree (a *null* in Java)

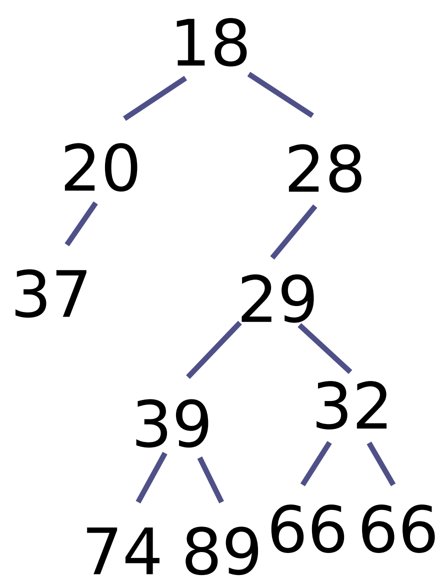


The null path length of *null* itself is 0

Similar concept to *height*, but with height we measure the *longest* path in the tree

Leftist heaps

Leftist invariant: the npl of the left child ≥ the npl of the right child



This means: the quickest way to reach a *null* is to follow the right branch

Leftist merging

We start with the naïve merging algorithm from earlier:

- The leftist invariant means that naïve merging stops after O(log n) steps

But the merge might break the leftist invariant!

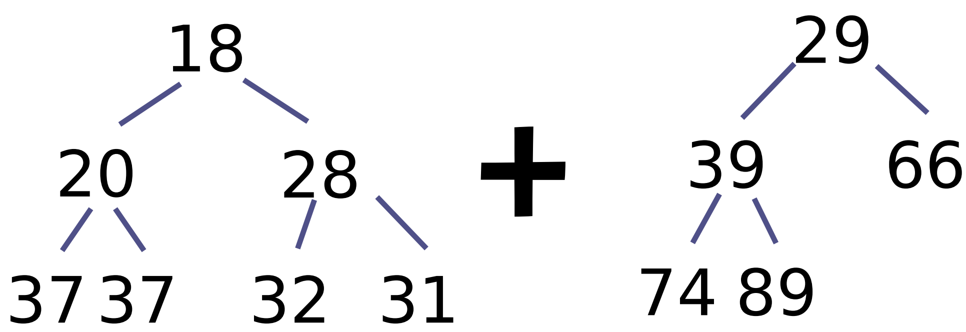
- When we descend into the right child, its npl might increase, and become greater than the left child

Fix it by:

- Going *upwards* in the tree from where the merge finished, and wherever we encounter a node where left child's npl < right child's npl, swap the two children!

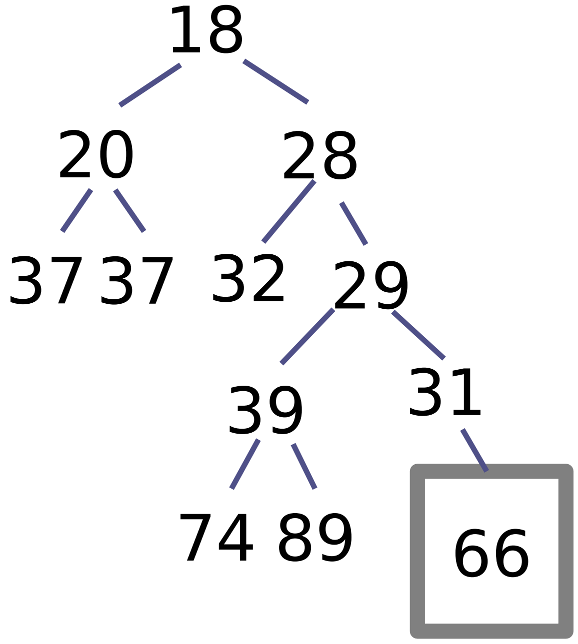
Leftist merging

1. Start with naïve merging from earlier



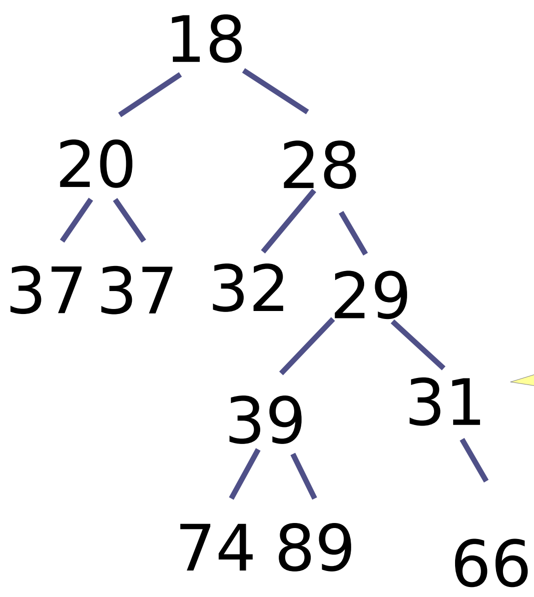
Leftist merging

2. The recursion “bottomed out” at 66 here



Leftist merging

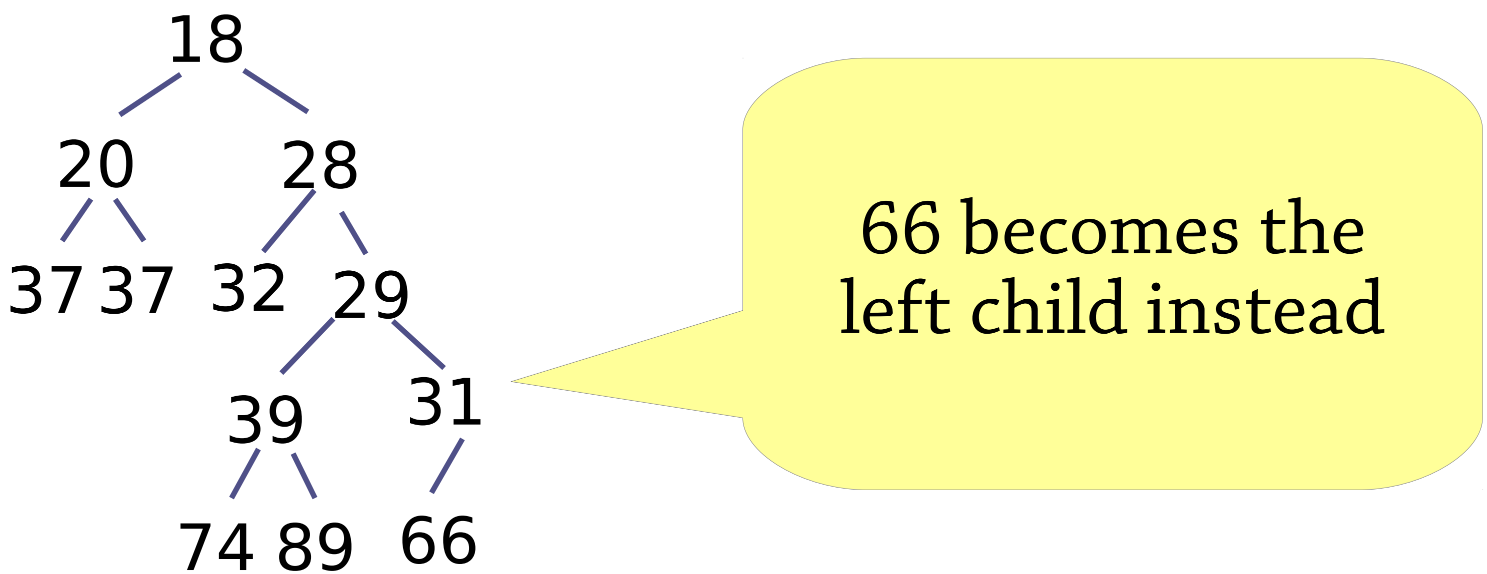
3. Go up to the parent, compare left and right child's npl



left npl: 0
right npl: 1
Invariant broken!

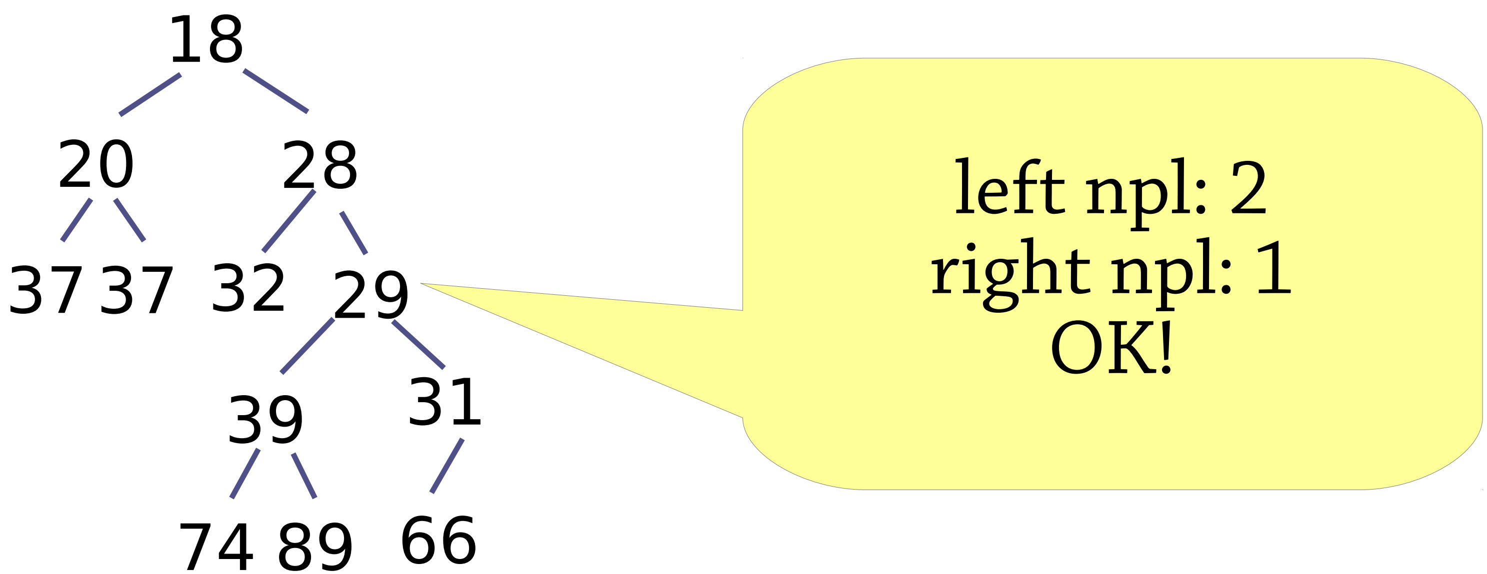
Leftist merging

4. If the leftist invariant is broken, swap the left and right children



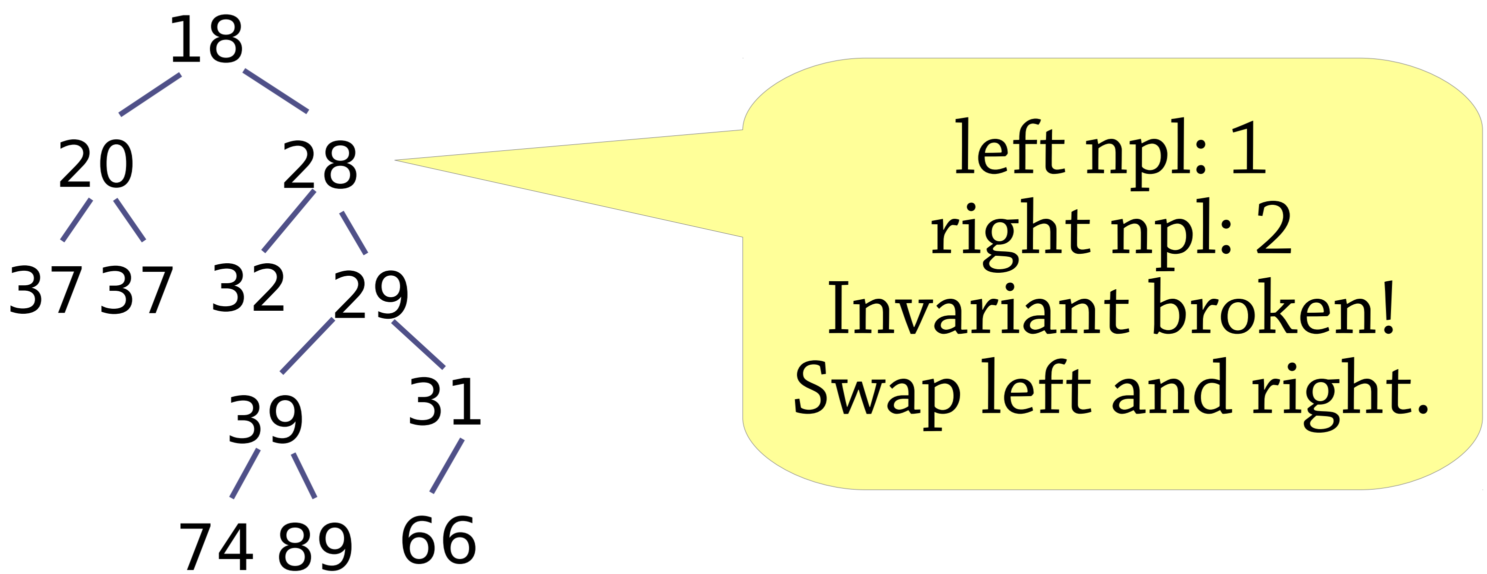
Leftist merging

5. Go up again and repeat!



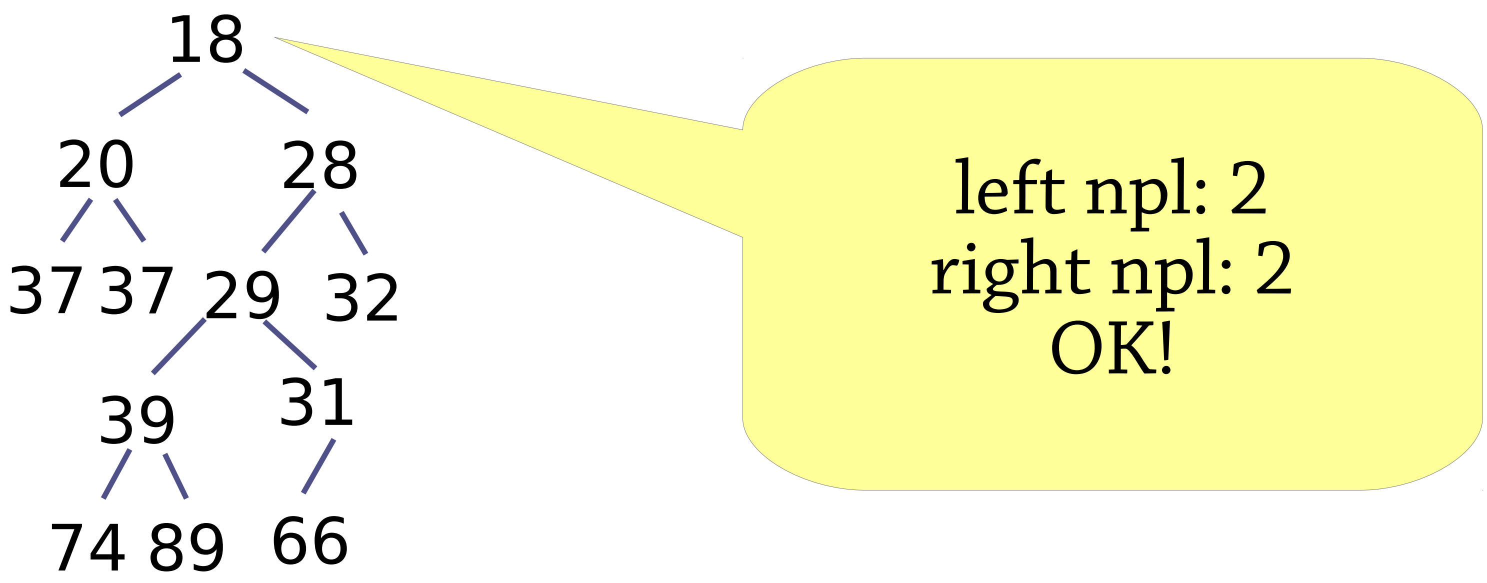
Leftist merging

5. Go up again and repeat!



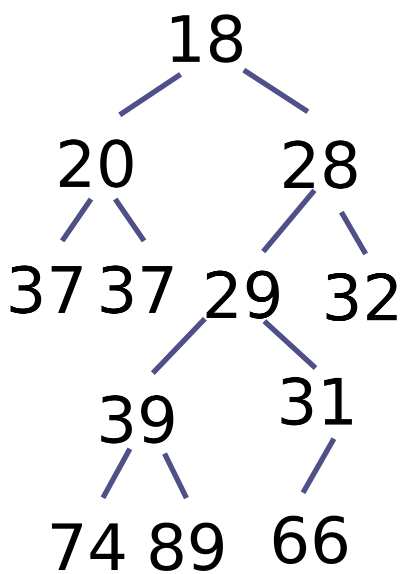
Leftist merging

5. Go up again and repeat!



Leftist merging

6. When we've reached the root, we've finished!



Notice how the final heap "leans to the left".

Implementation

Implementation:

- Need to be able to compute npl efficiently
- Add a field for the npl to each node, and update it whenever we modify the node
- Update by computing: npl = 1 + right child's npl

Complexity of leftist merging

I claim: the npl of a tree of size n is O(log n)

- Check it for yourself :)
- For balanced trees, the npl is O(log n), much like height
- By unbalancing a tree, we make some paths longer, and some shorter. This increases the height, but *decreases* the npl!

Hence, in a leftist heap, by following the right branch O(log n) times, you reach a *null*

So merge takes O(log n) time!

- log n steps down the tree to do the naïve tree
- then log n steps upwards while repairing the leftist invariant

Leftist heaps

Implementation of priority queues:

- binary trees with heap property
- leftist invariant for O(log n) merging
- other operations are based on merge

A good fit for functional languages:

- based on trees rather than arrays