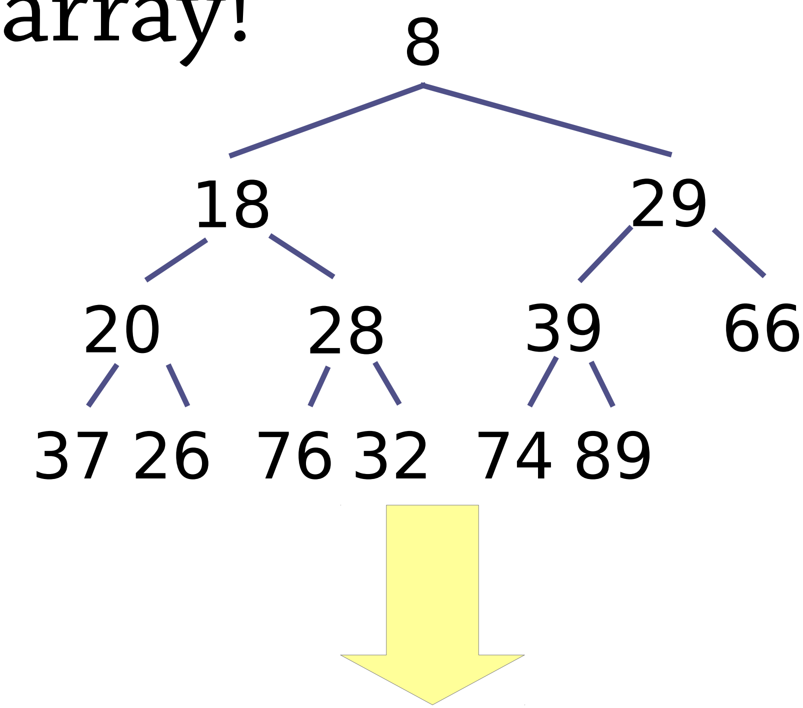
# Binary heaps
*(chapters 20.3 – 20.5)*
# Leftist heaps

# Binary heaps are arrays!

A binary heap is really implemented using an array!



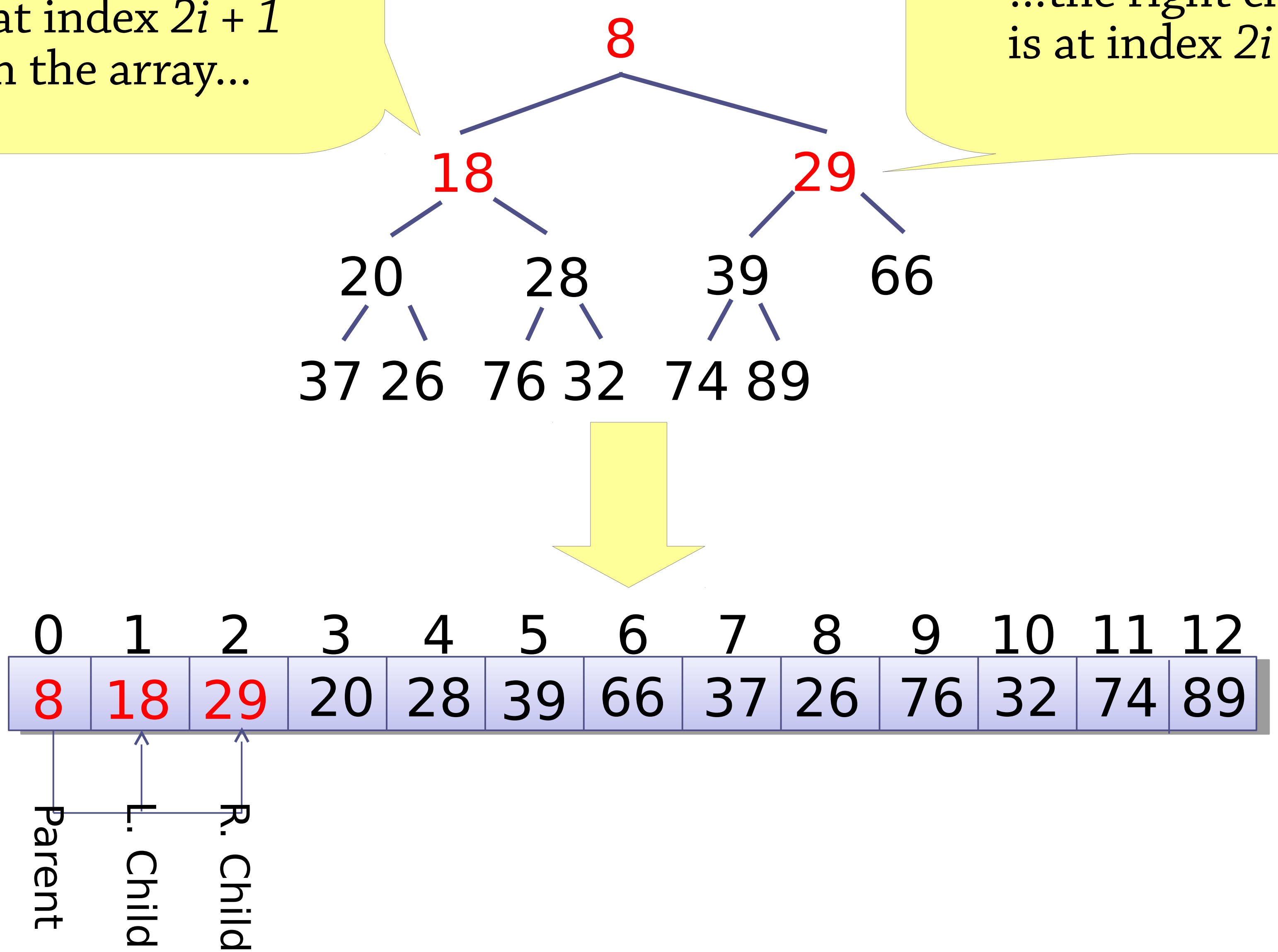

Possible because of completeness property

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 18 | 29 | 20 | 28 | 39 | 66 | 37 | 26 | 76 | 32 | 74 | 89 |

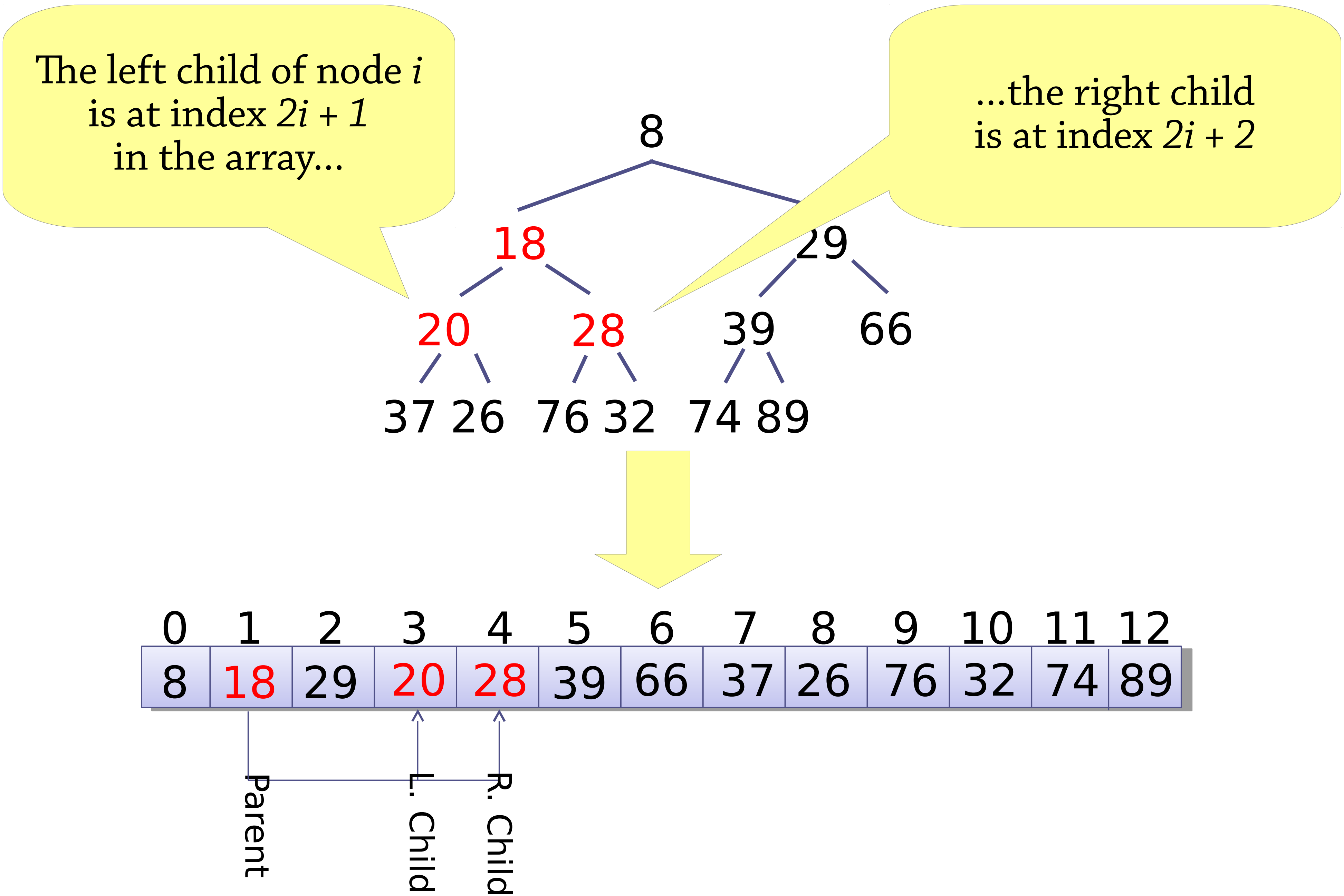
# Child positions

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

…the right child is at index *2i + 2*



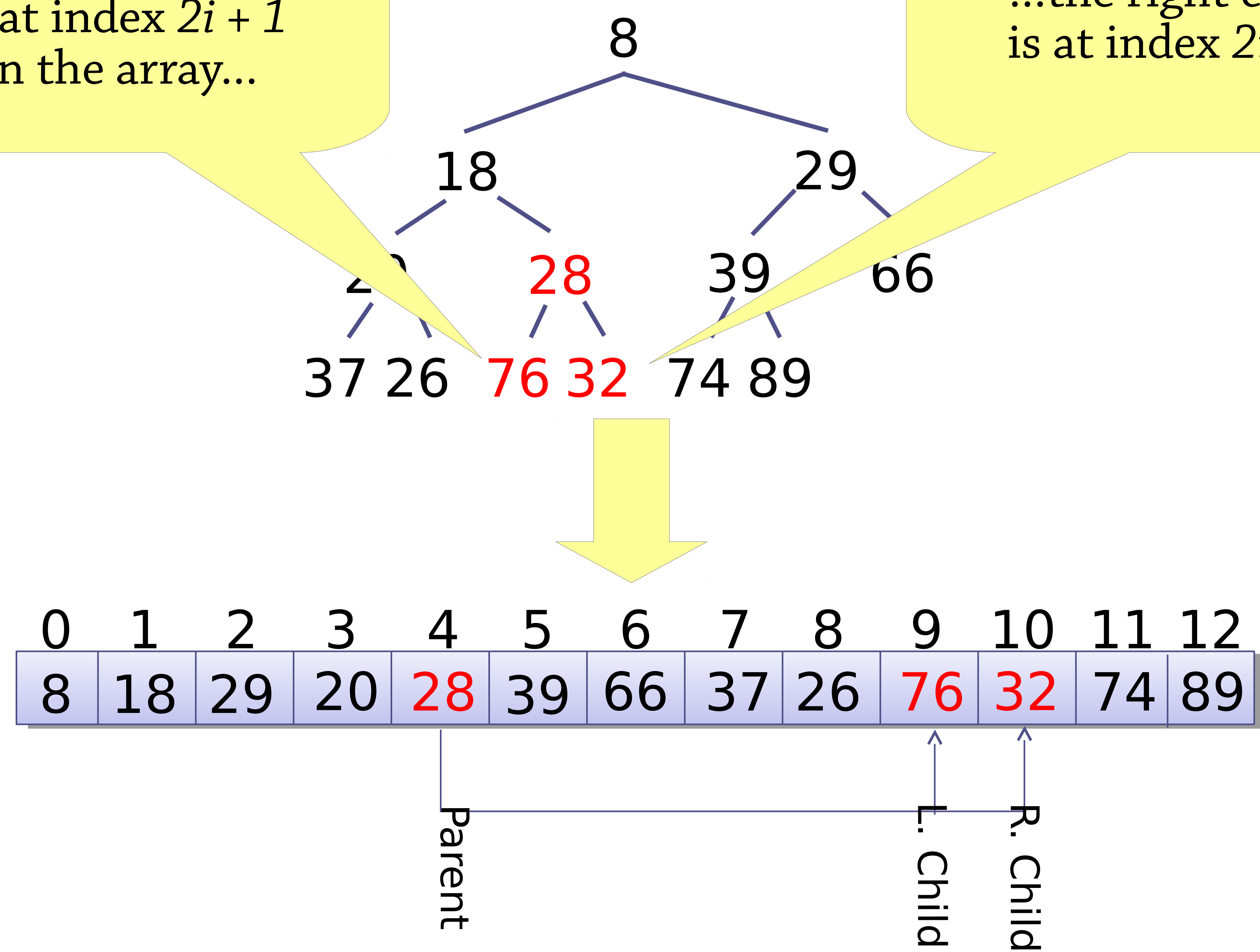
# Child positions



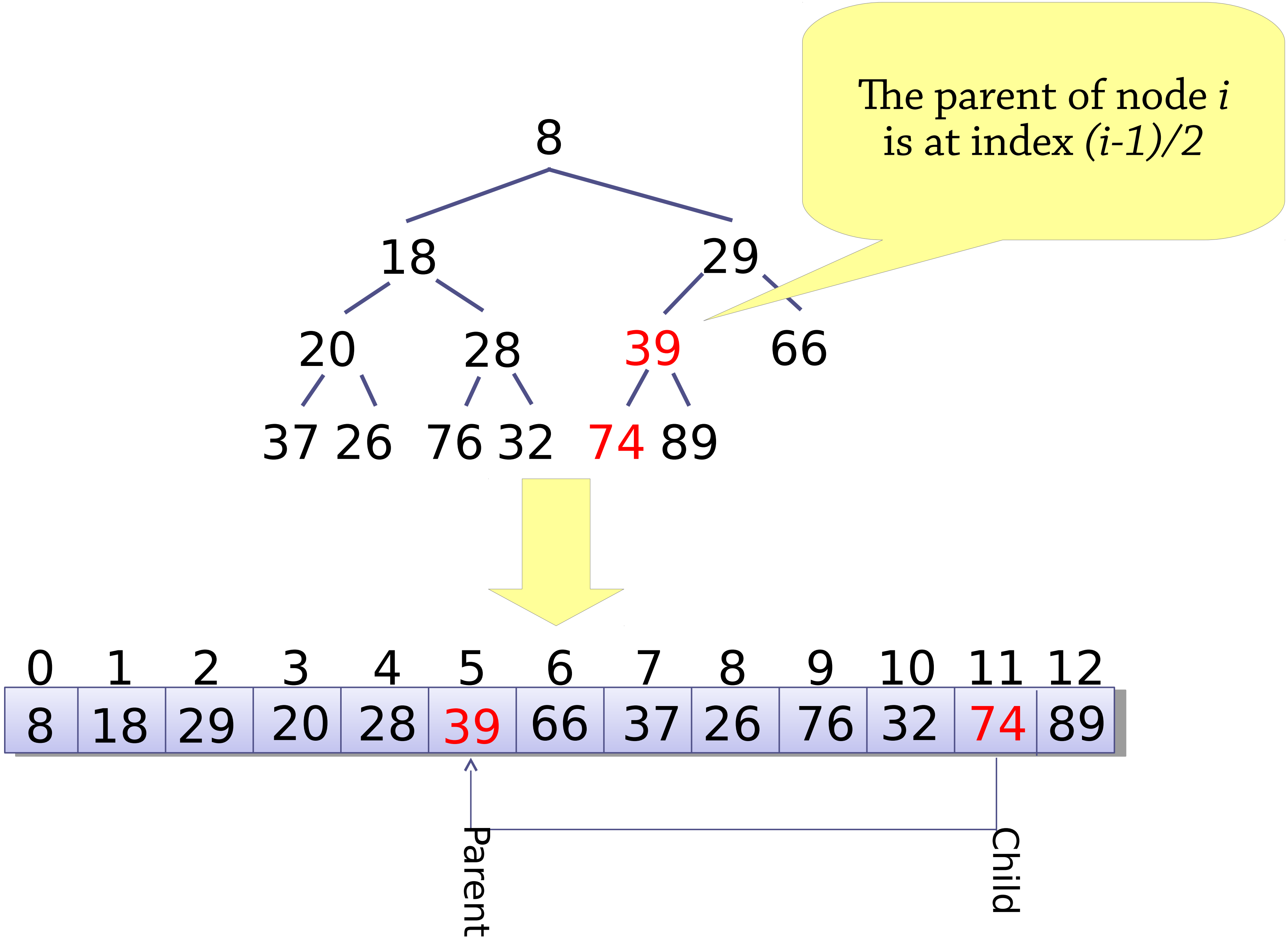
# Child positions

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

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



# Parent position



# Reminder: inserting into a binary heap

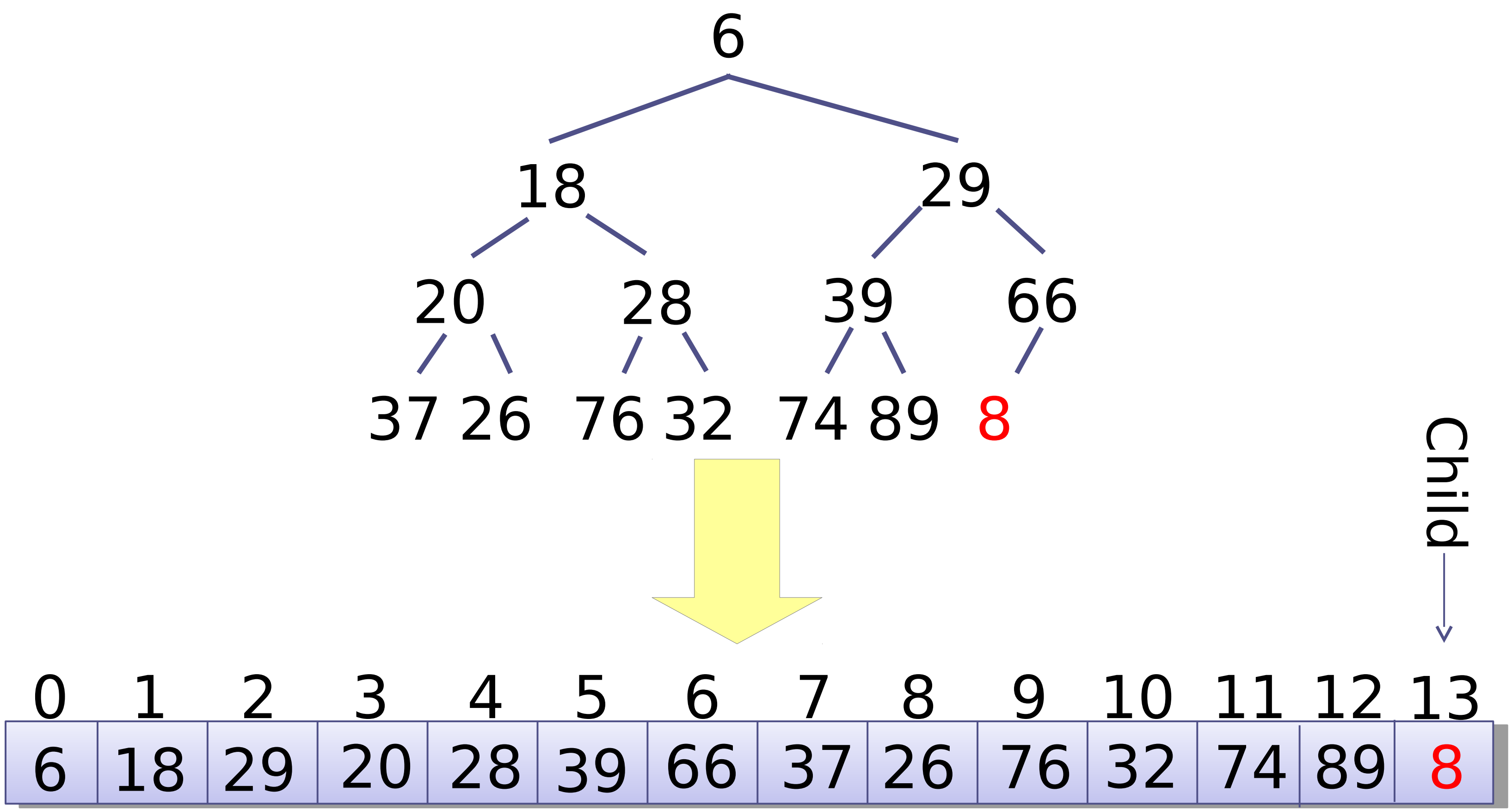
To insert an element into a binary heap:

- 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!

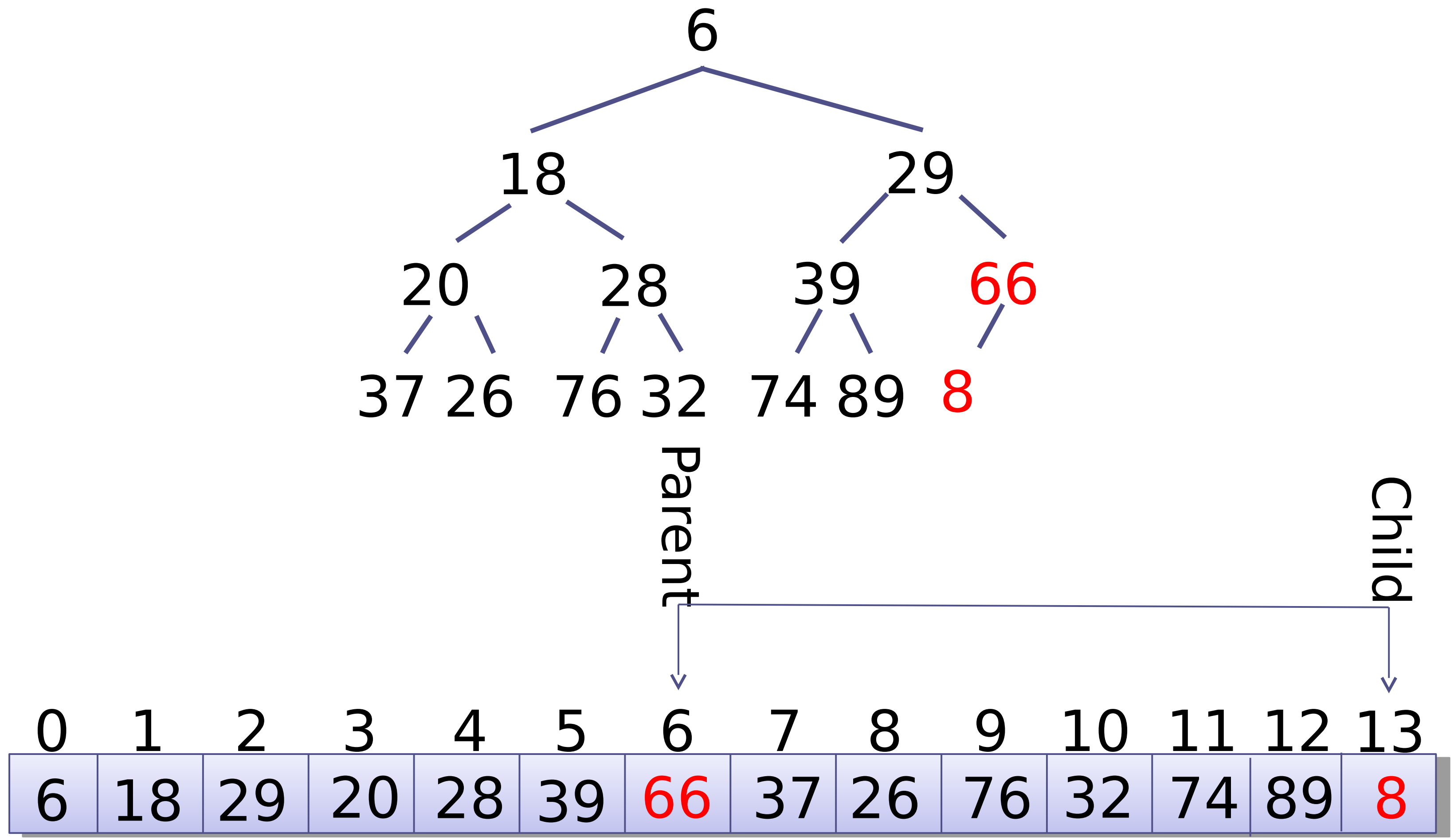
# Inserting into a binary heap

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



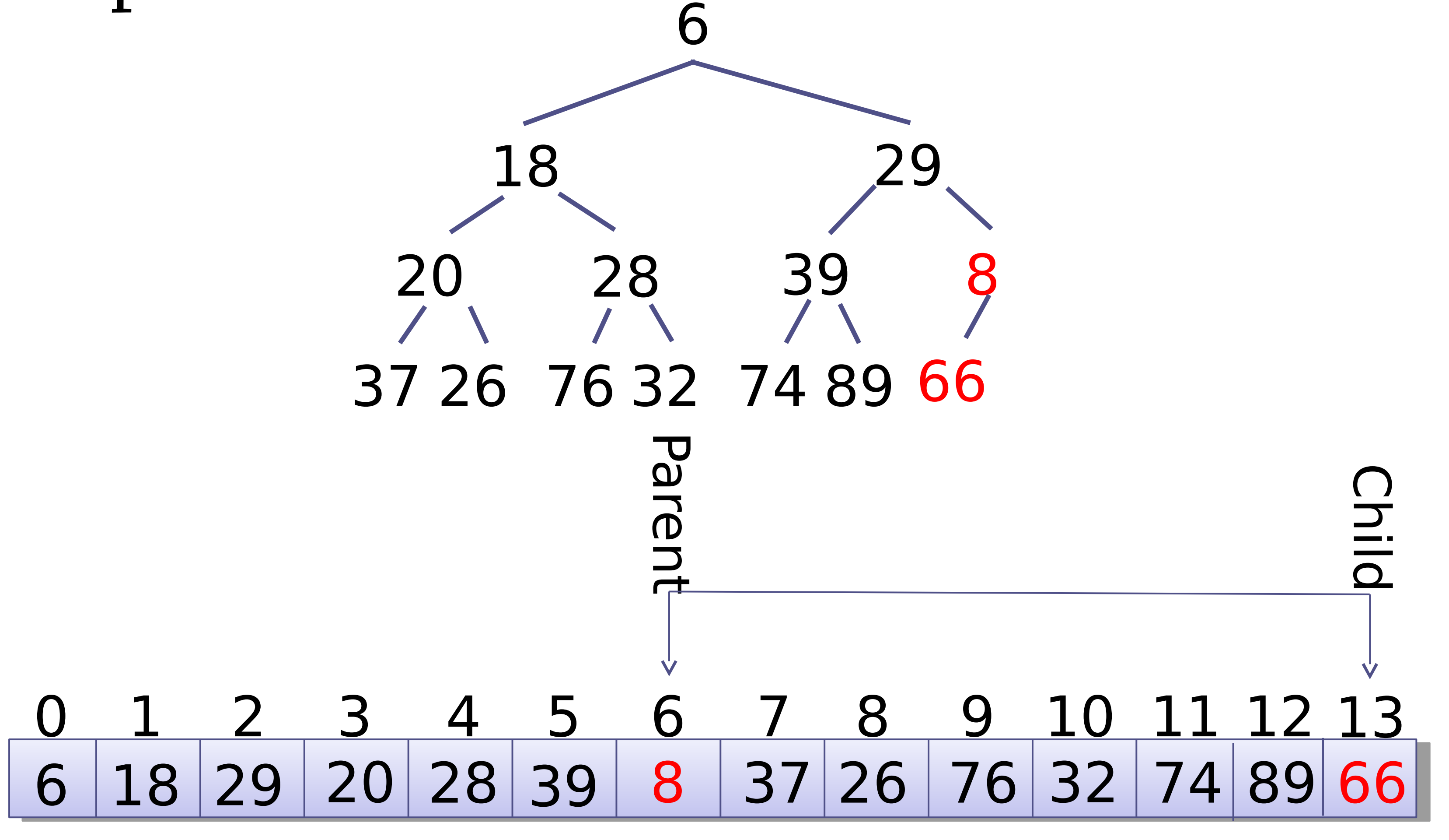
# Inserting into a binary heap

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



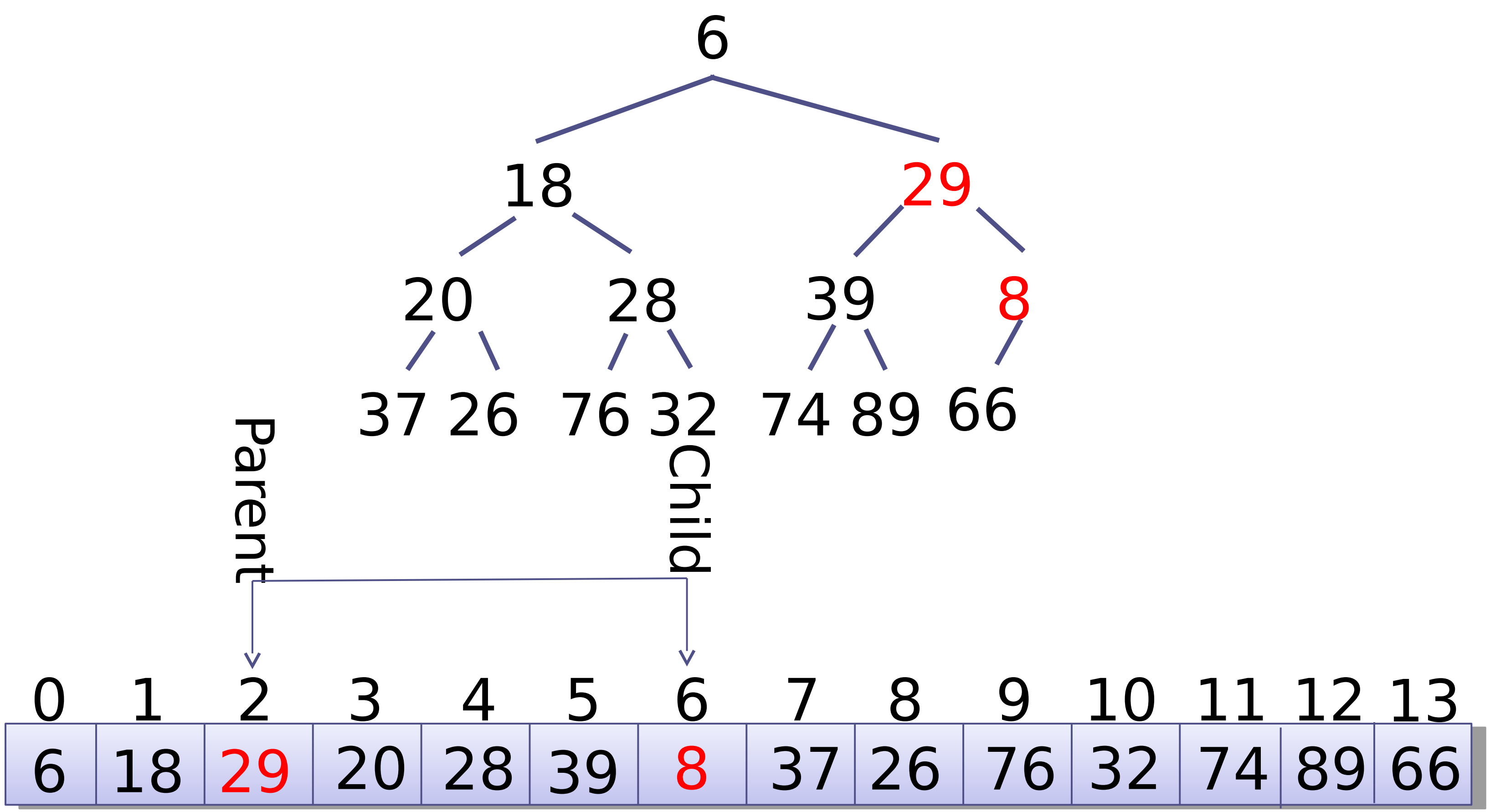
# Inserting into a binary heap

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



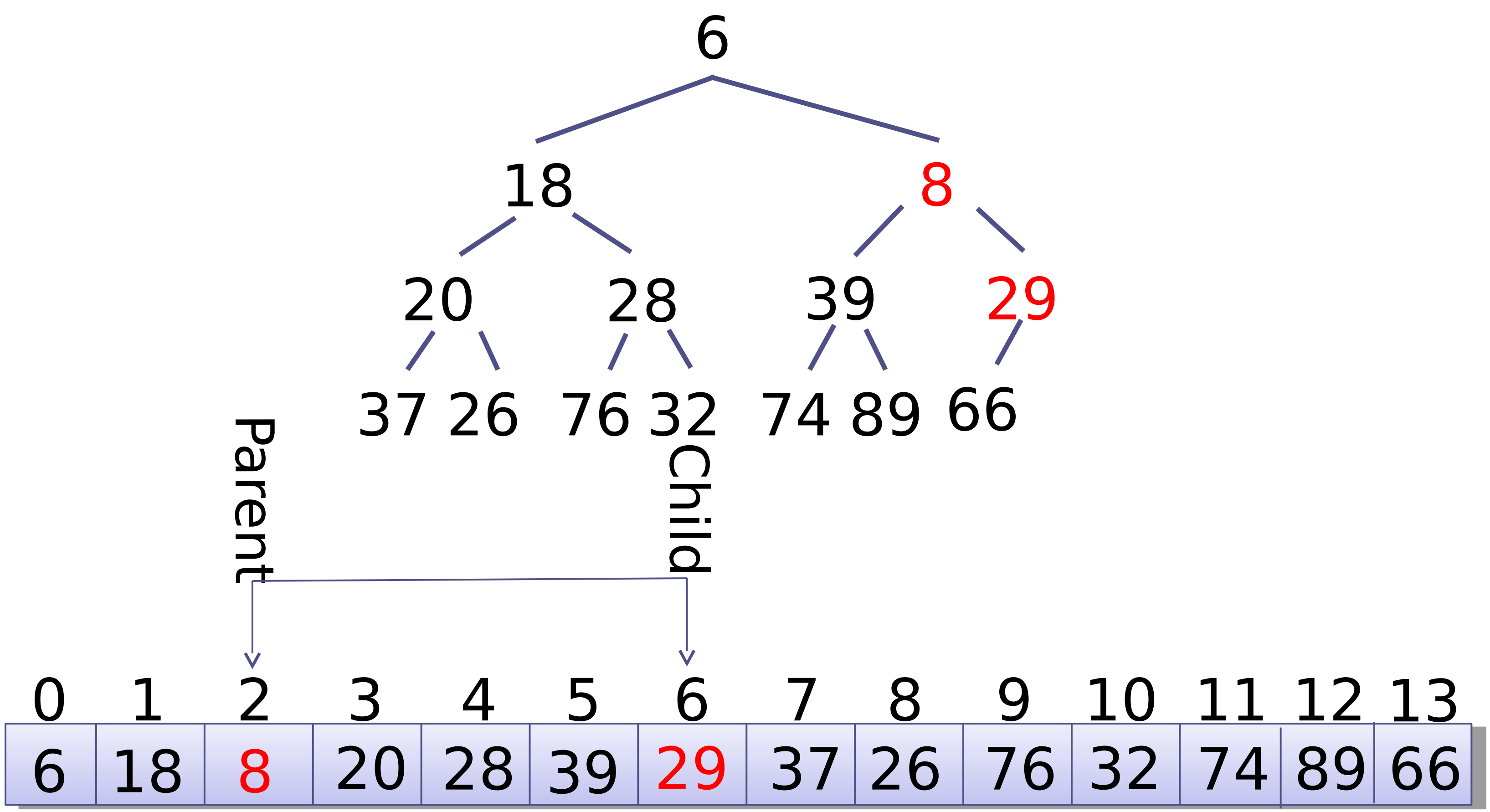
# Inserting into a binary heap

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

In the rest of the lecture, we will only show the heaps as trees

- But you should have the "array view" in your head when looking at these trees

# Building a heap

One more operation, *build heap*

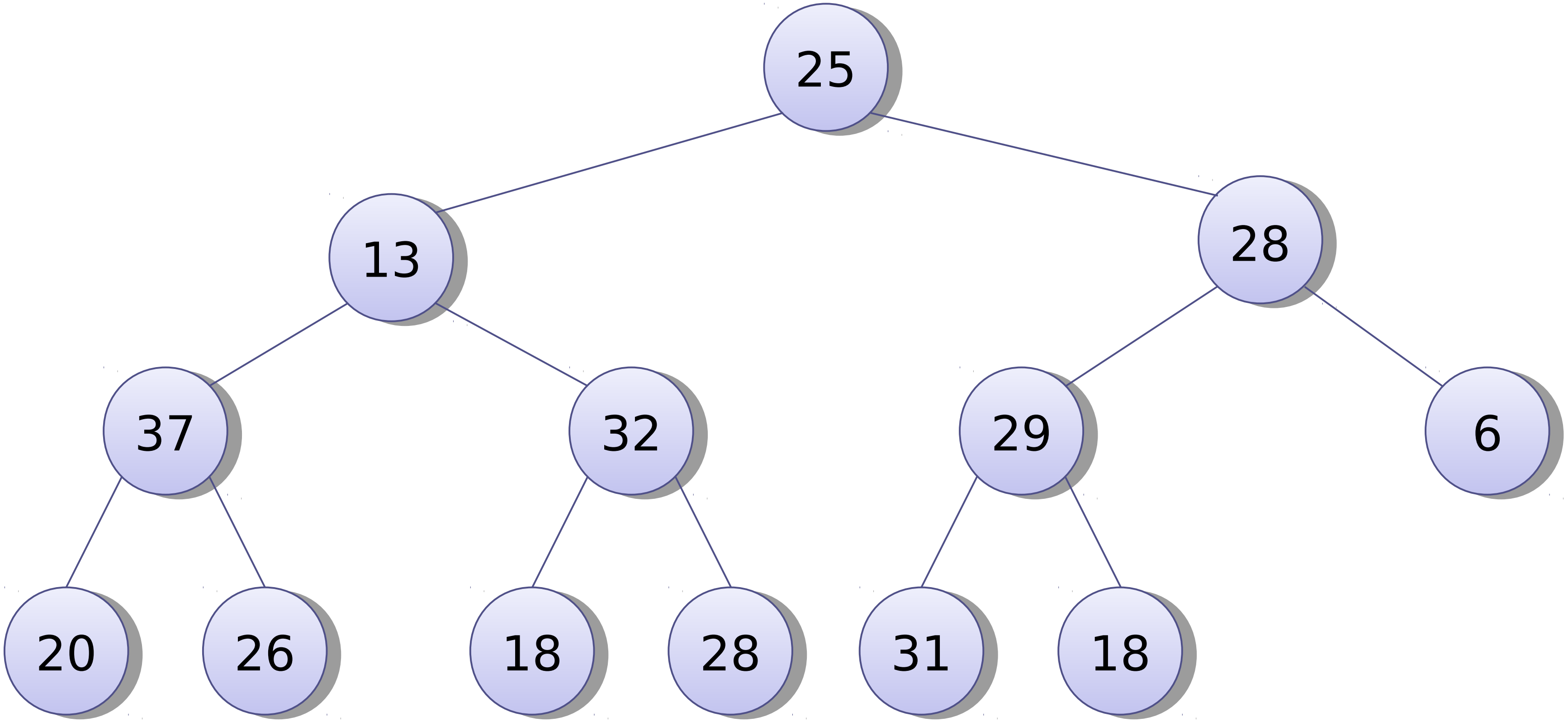
- Takes an arbitrary array and makes it into a heap
- In-place: moves the elements around to make the heap property hold

Idea:

- Heap property: each element must be less than its children
- If a node breaks this property, we can fix it by *sifting down*
- So simply looping through the array and sifting down each element in turn ought to fix the invariant
- But when we sift an element down, its children must already have the heap property (otherwise the sifting doesn't work)
- To ensure this, loop through the array in *reverse*

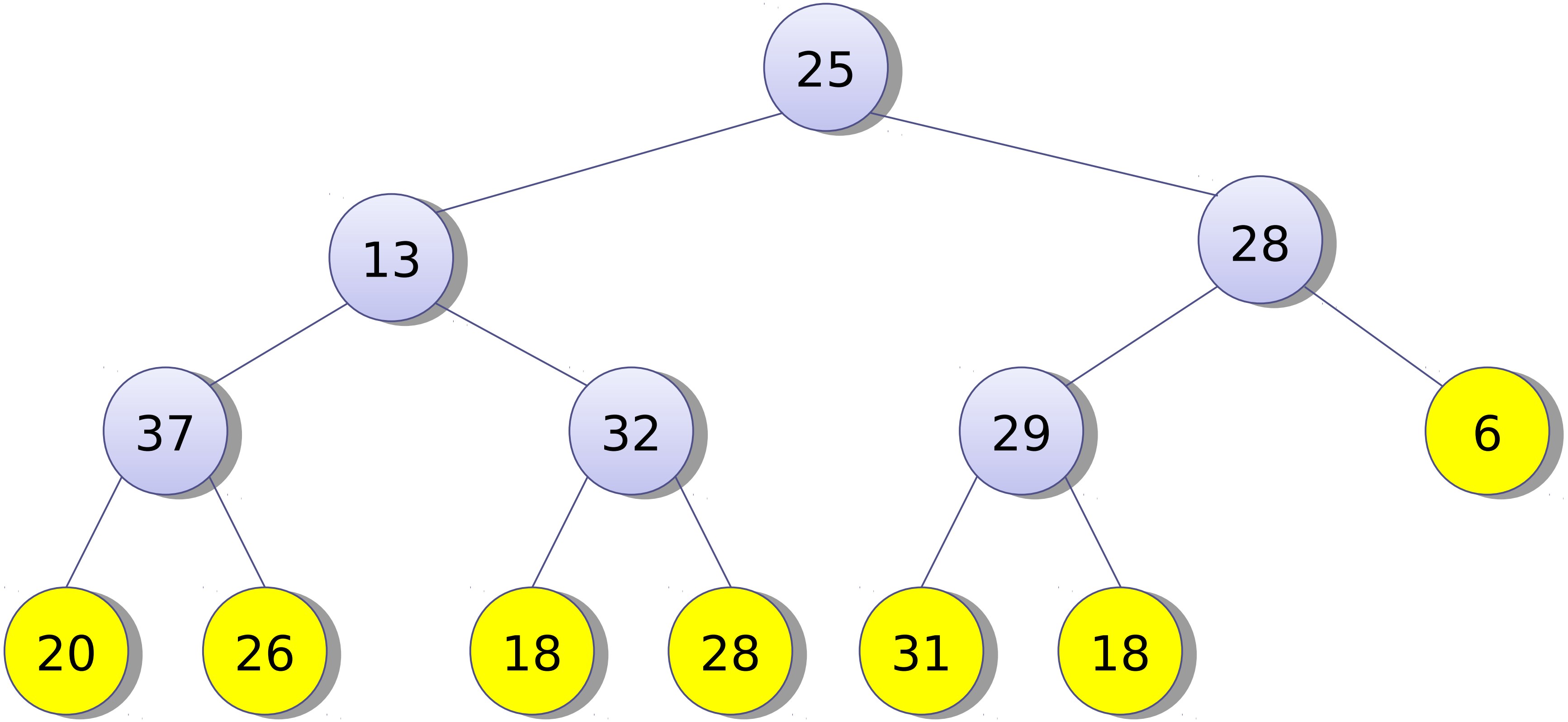
# Building a heap

Go through elements in reverse order, sifting each down



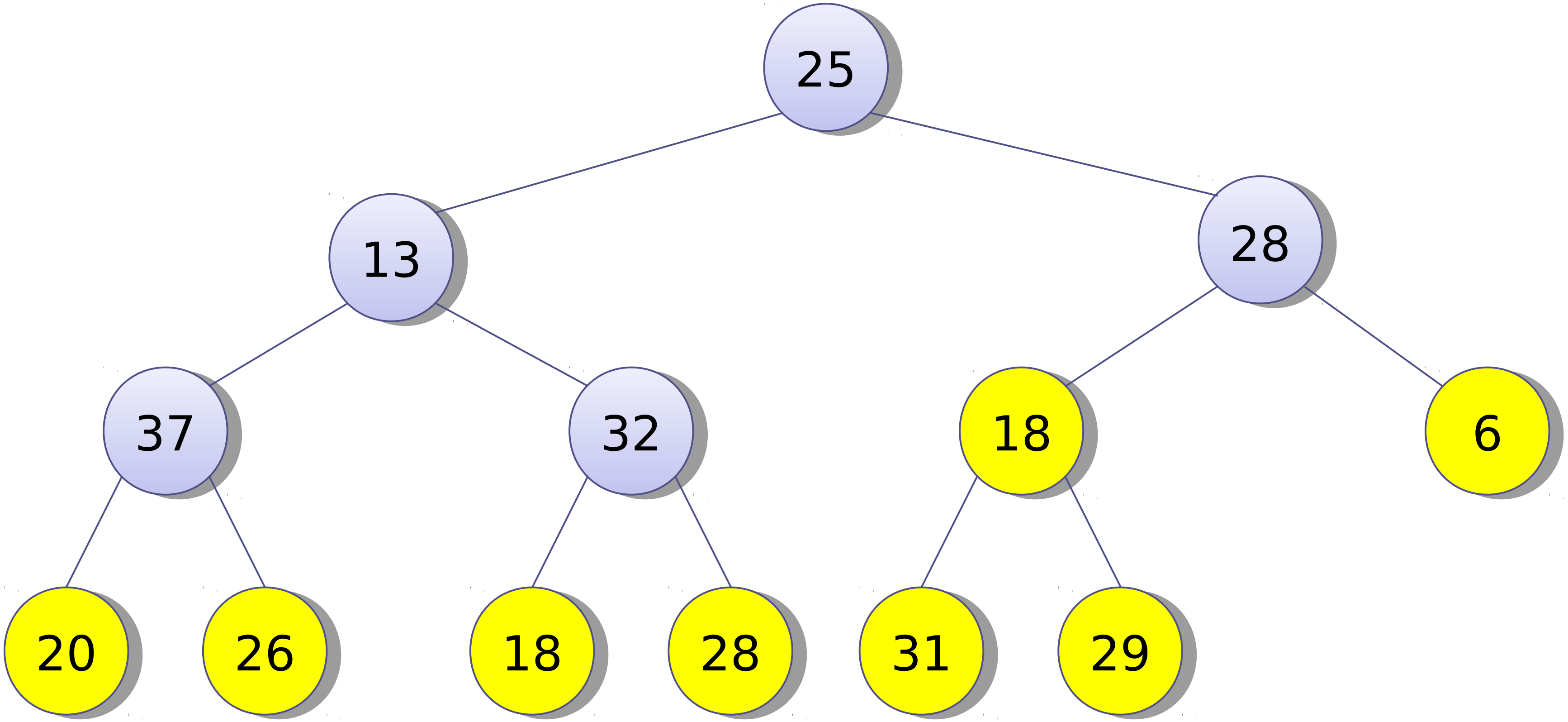
# Building a heap

Leaves never need sifting down!



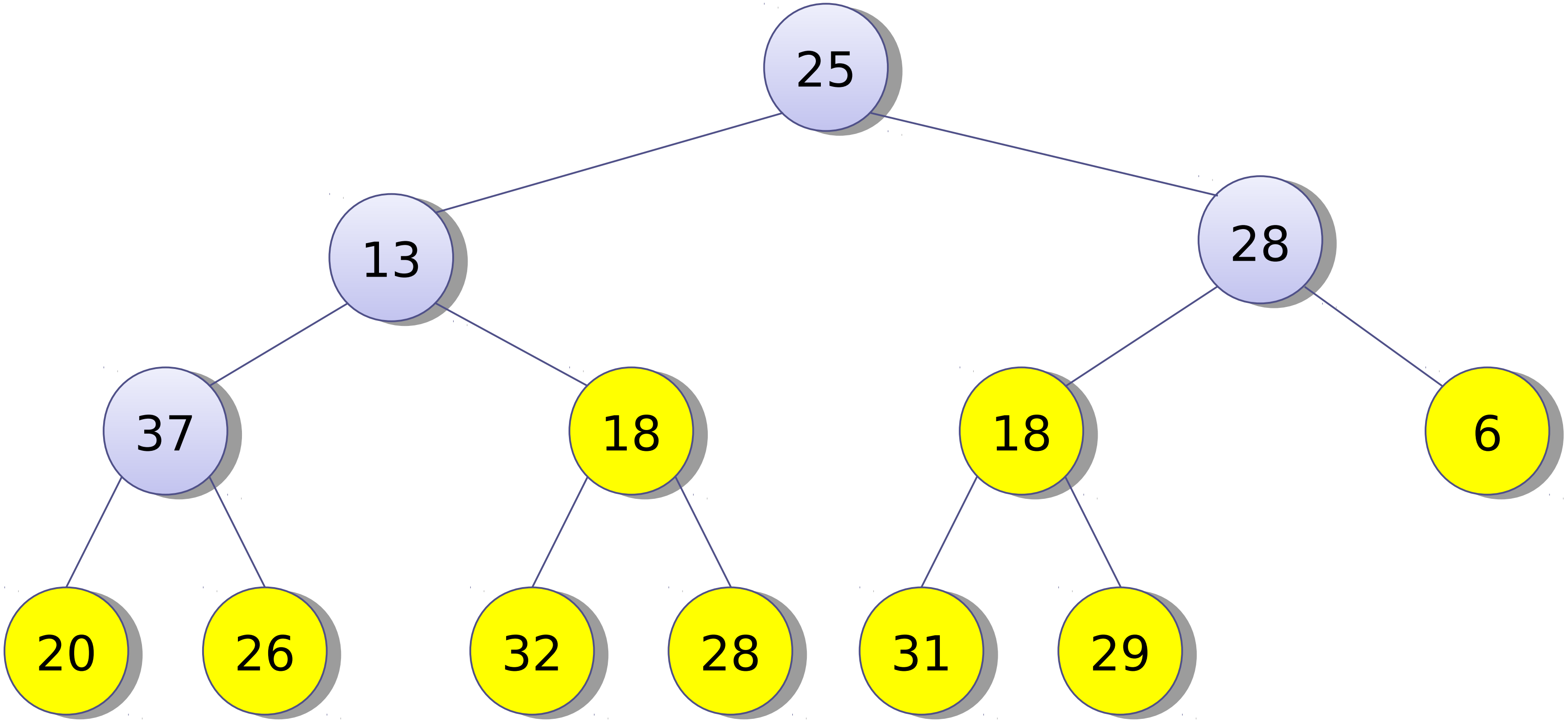
# Building a heap

29 is greater than 18 so needs swapping



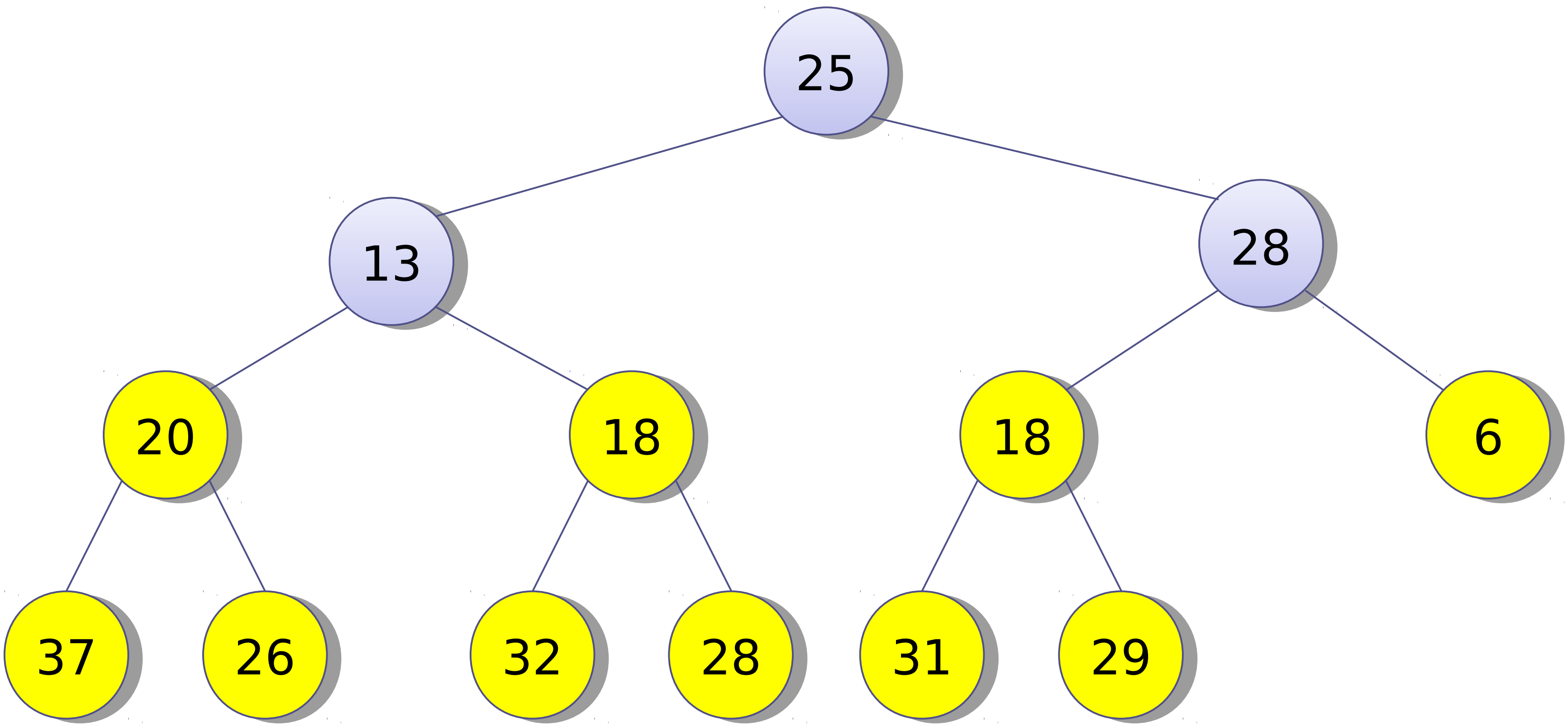
# Building a heap

32 is greater than 18 so needs swapping



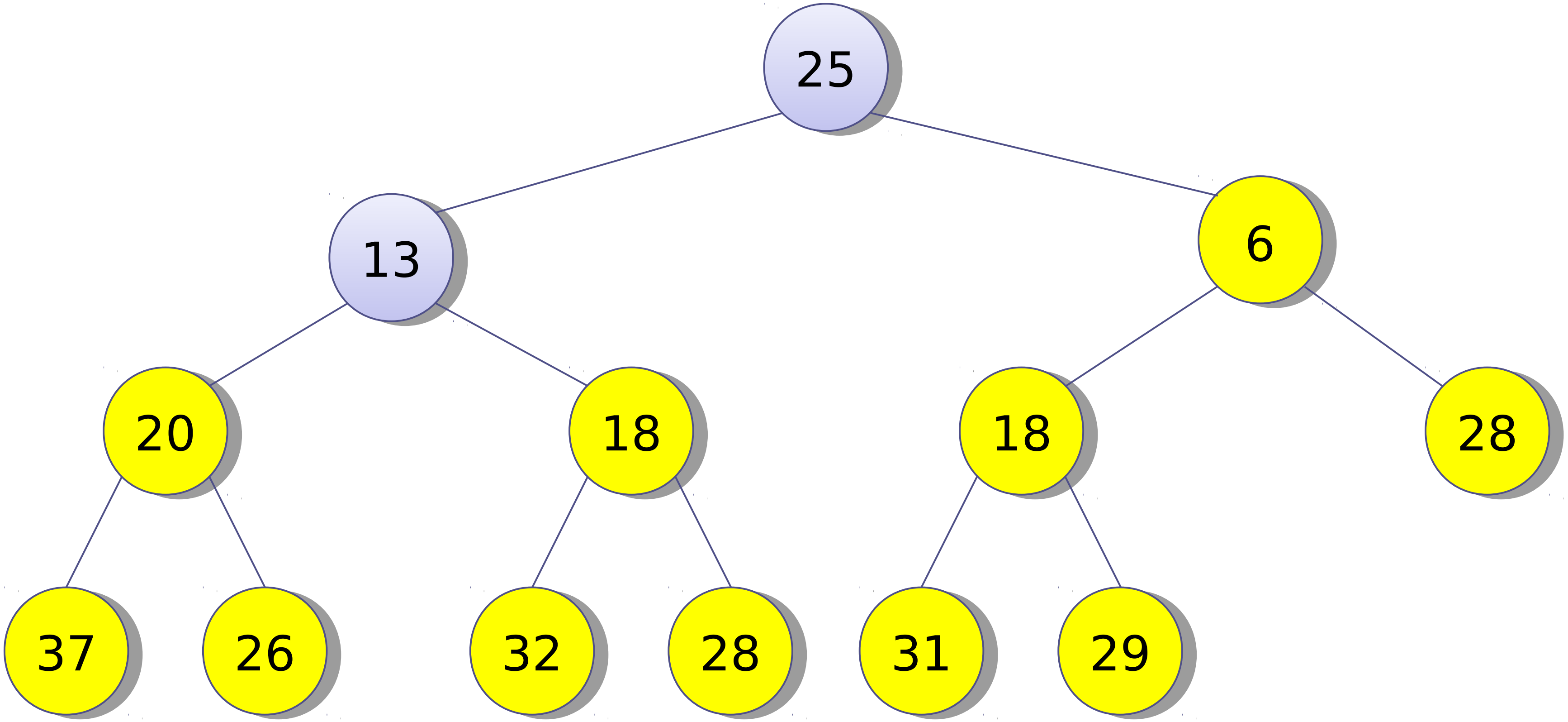
# Building a heap

37 is greater than 20 so needs swapping



# Building a heap

28 is greater than 6 so needs swapping



# Build heap complexity

You would expect O(n log n) complexity:

- n “sift down” operations
- each sift down has O(log n) complexity

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 are going to repeatedly remove the *largest* value from the array and put it in the right place

- using a so-called *max heap*, a heap where you can find and delete the *maximum* element instead of the minimum

We'll divide the array into two parts

- The first part will be a heap
- The rest will contain the values we've removed

(This division is the same idea we used for in-place selection and insertion sort)

# Heapsort, in-place

First turn the array into a heap

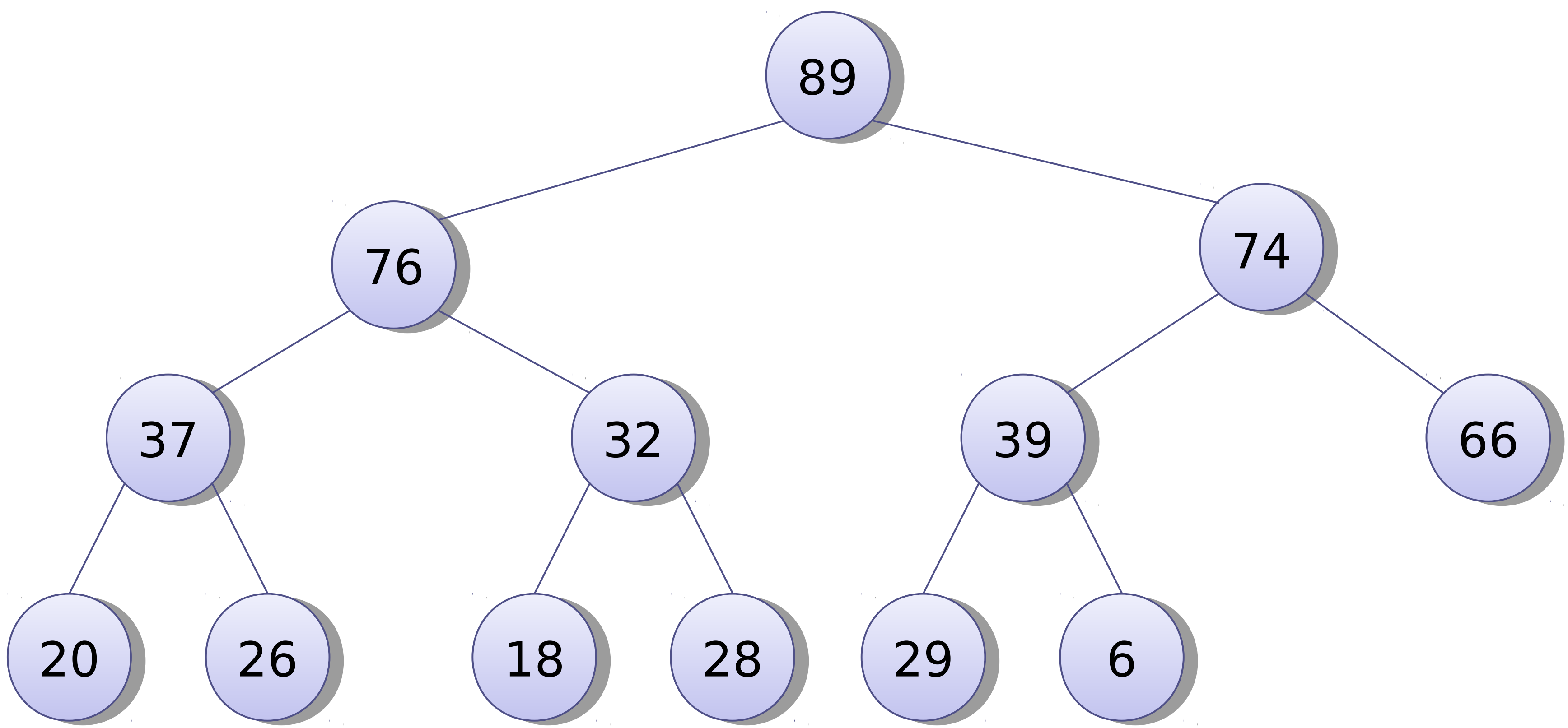
Then just repeatedly delete the minimum element! Remember the deletion algorithm:

- Swap the maximum (first) element with the last element of the heap
- Reduce the size of the heap by 1
  (deletes the first element, breaks the invariant)
- Sift the first element down
  (repairs the invariant)

The *swap* actually puts the maximum element in the right place in the array

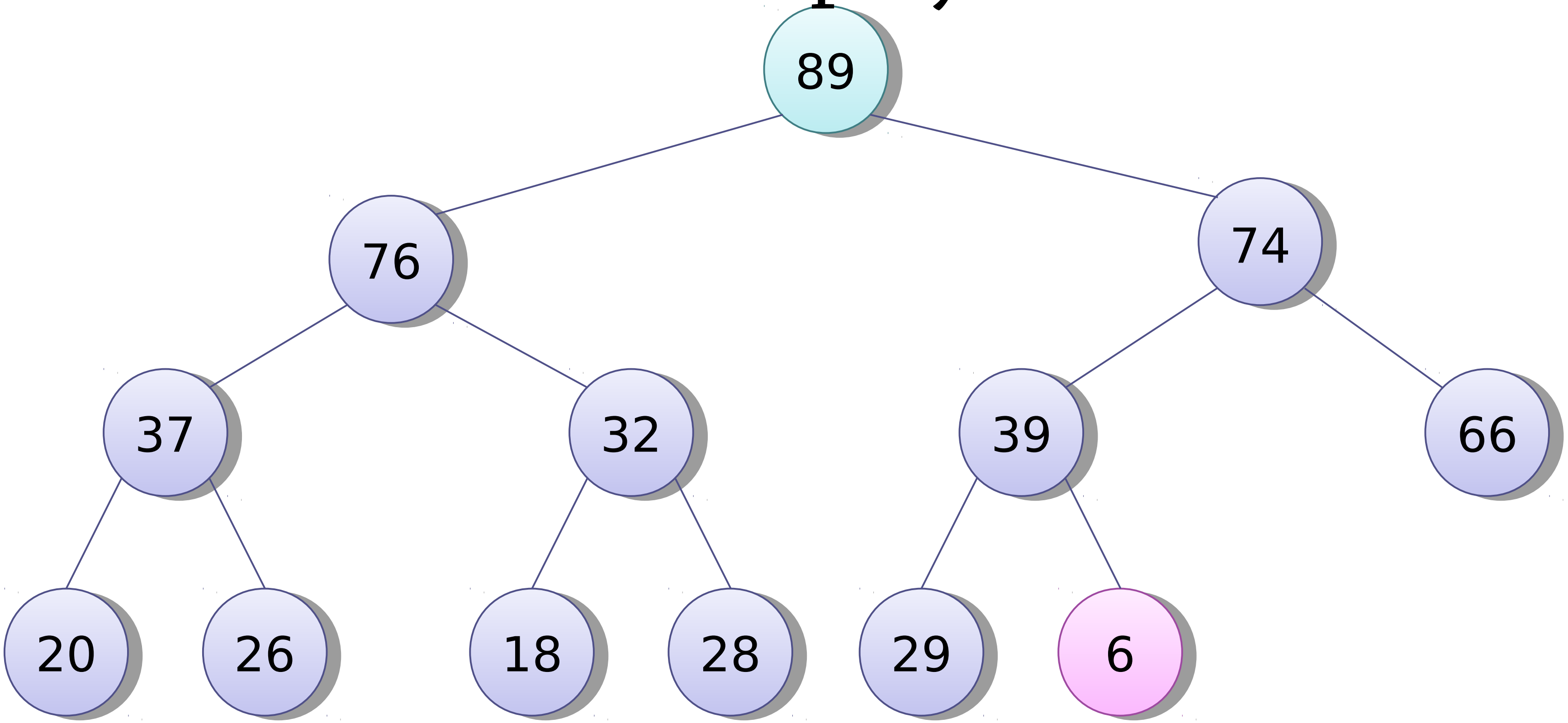
# Trace of heapsort

First build a heap (not shown)



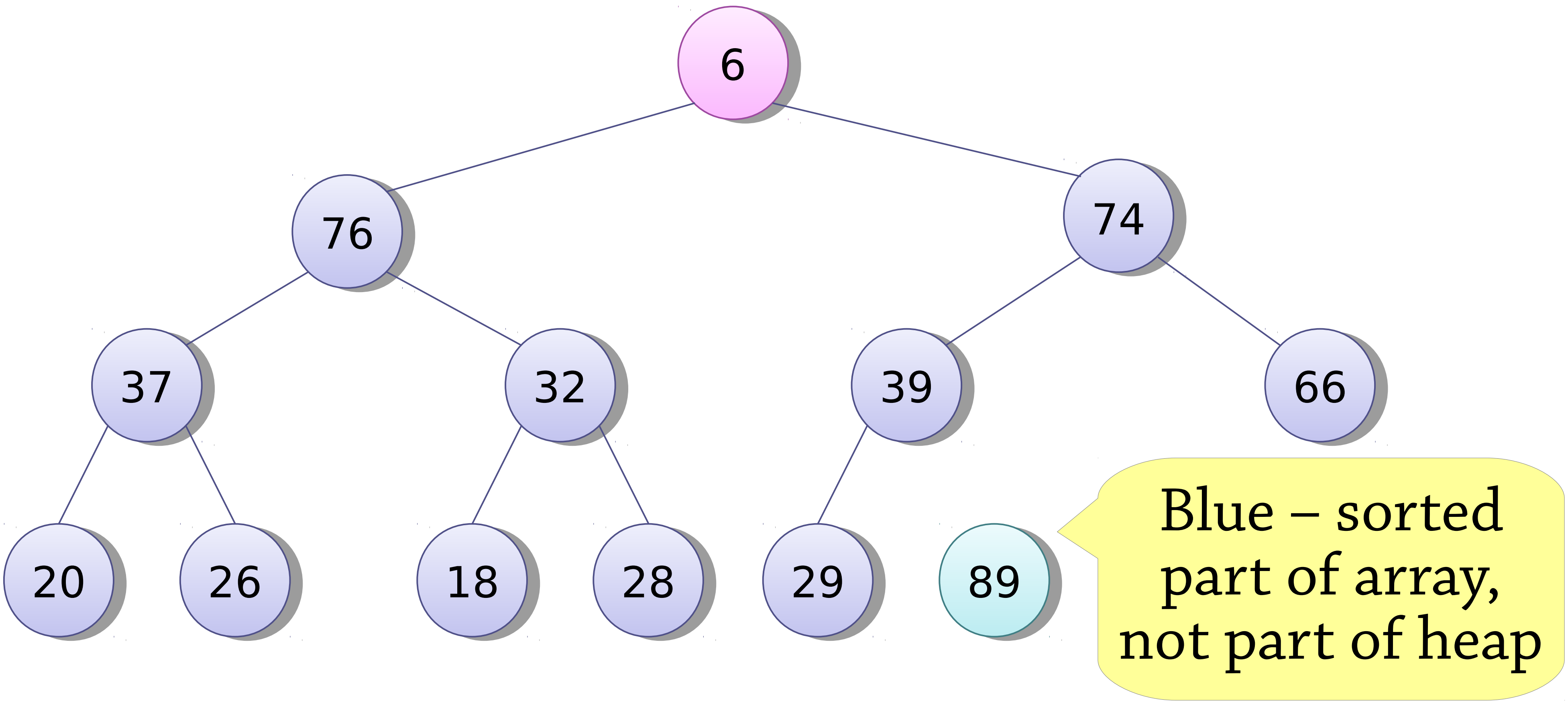
# Trace of heapsort

Step 1: swap maximum and last element; decrease size of heap by 1



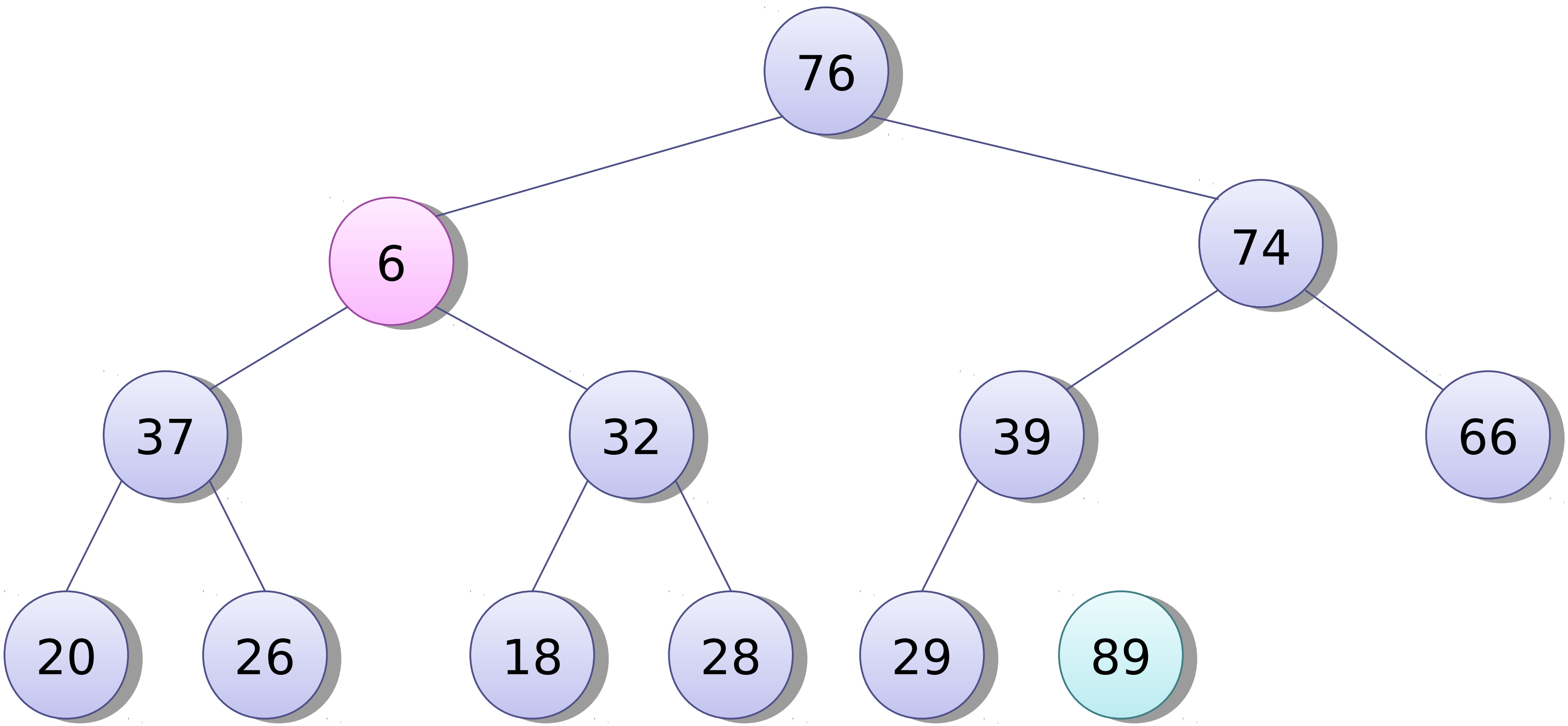
# Trace of heapsort

## Step 2: sift first element down



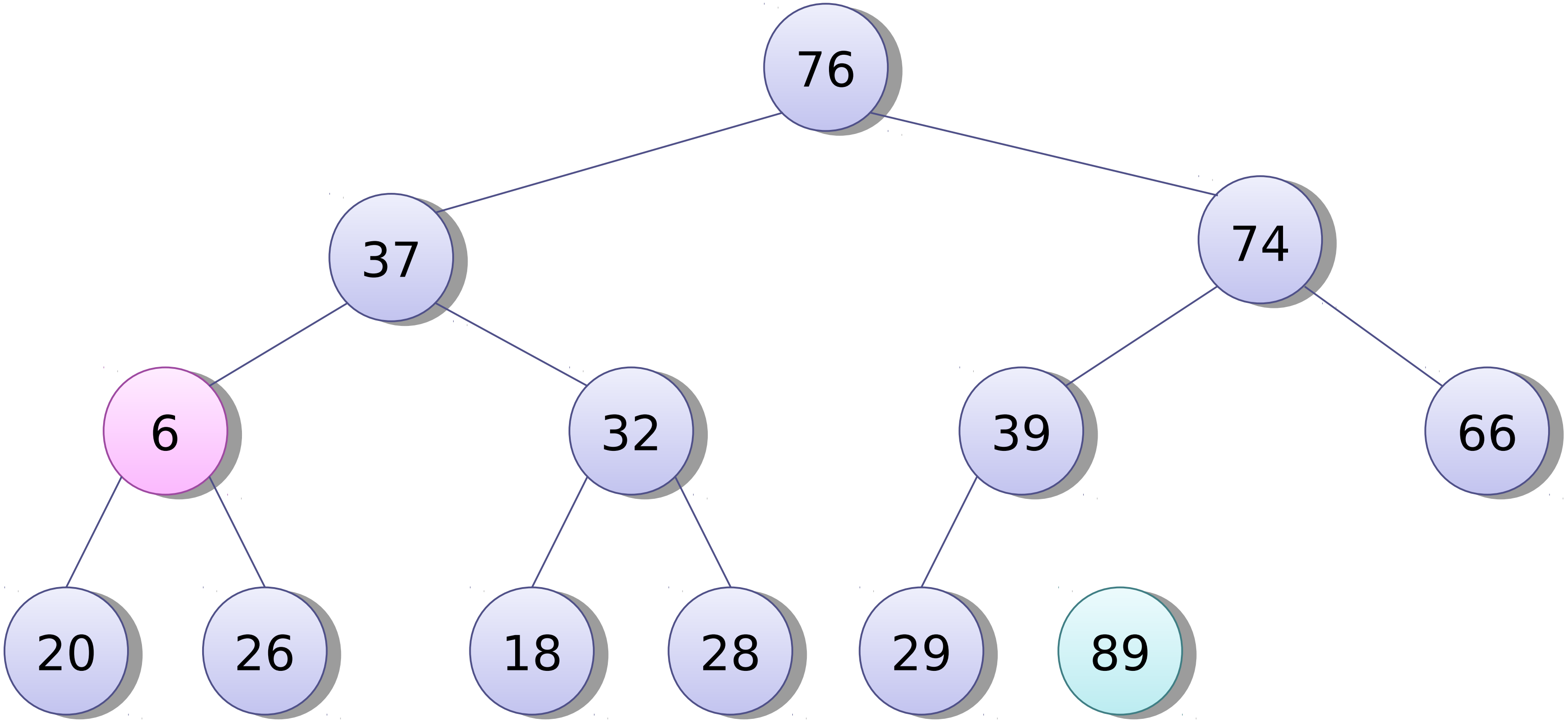
# Trace of heapsort

Step 2: sift first element down



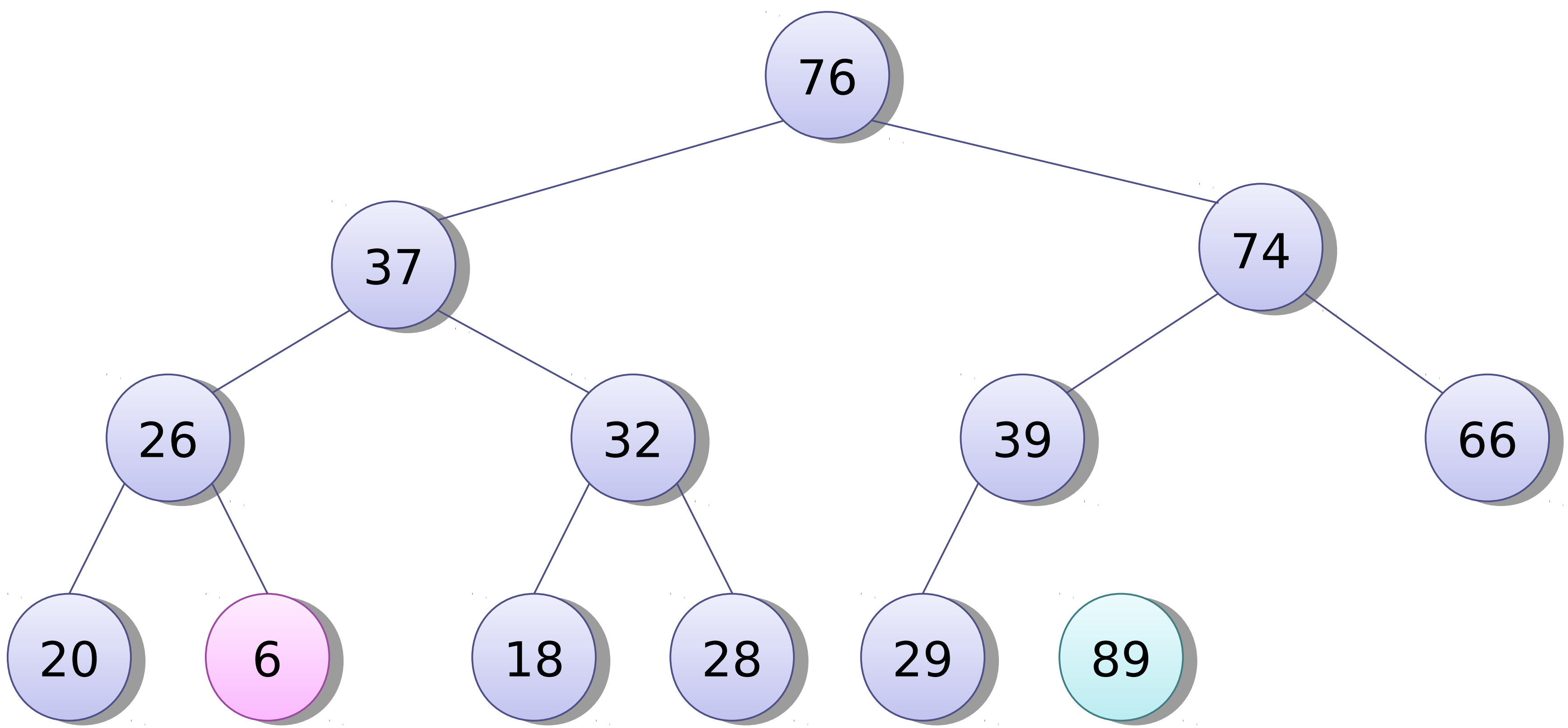
# Trace of heapsort

## Step 2: sift first element down



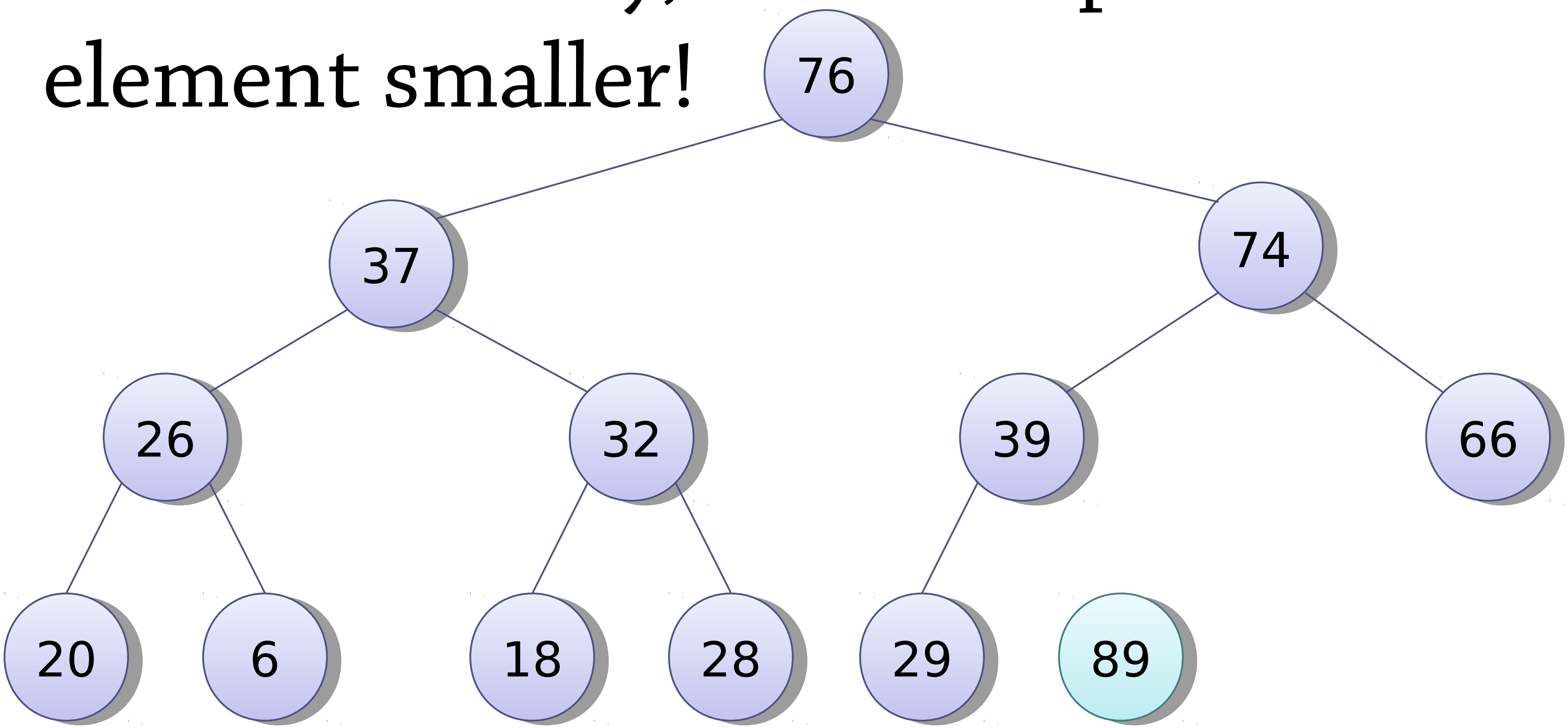
# Trace of heapsort

## Step 2: sift first element down



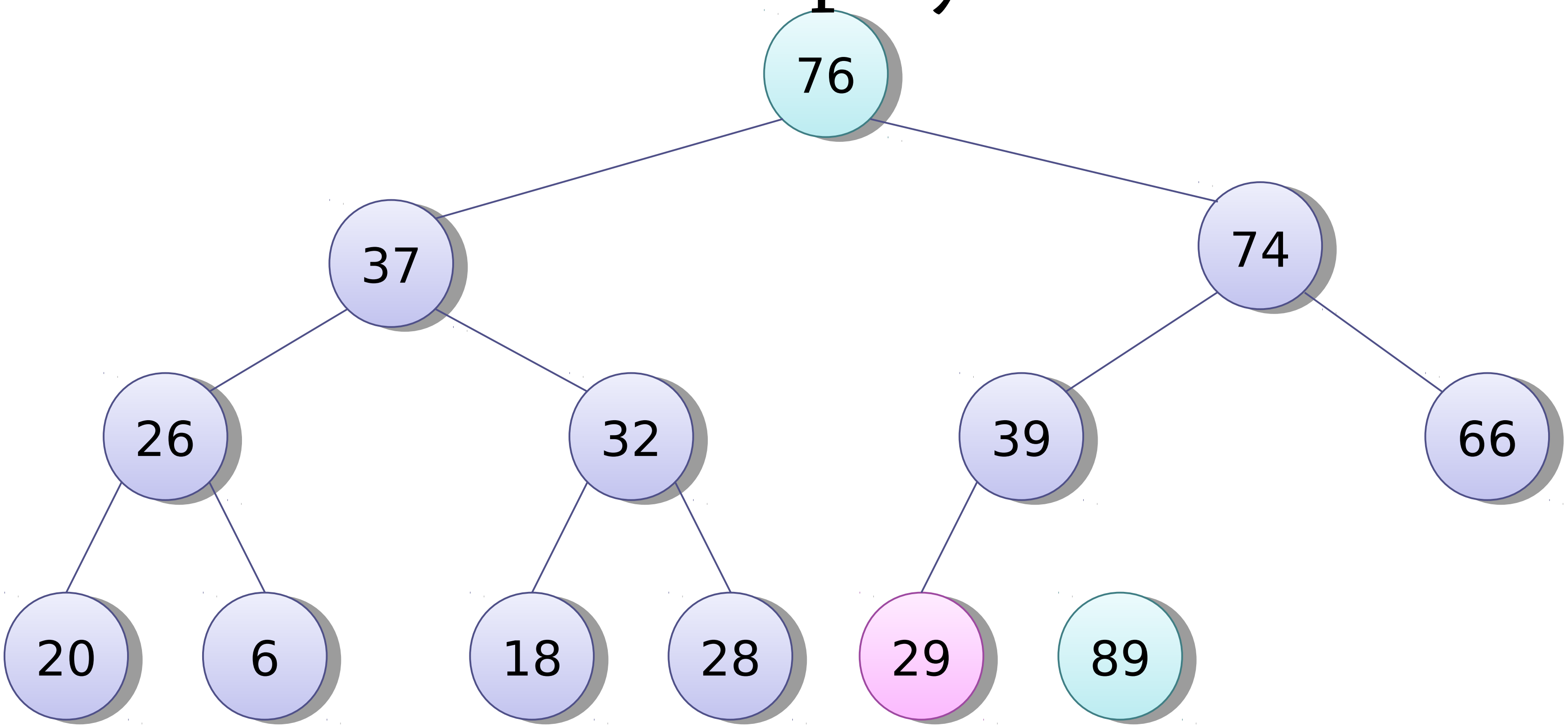
# Trace of heapsort

We now have the biggest element at the end of the array, and a heap that's one element smaller!



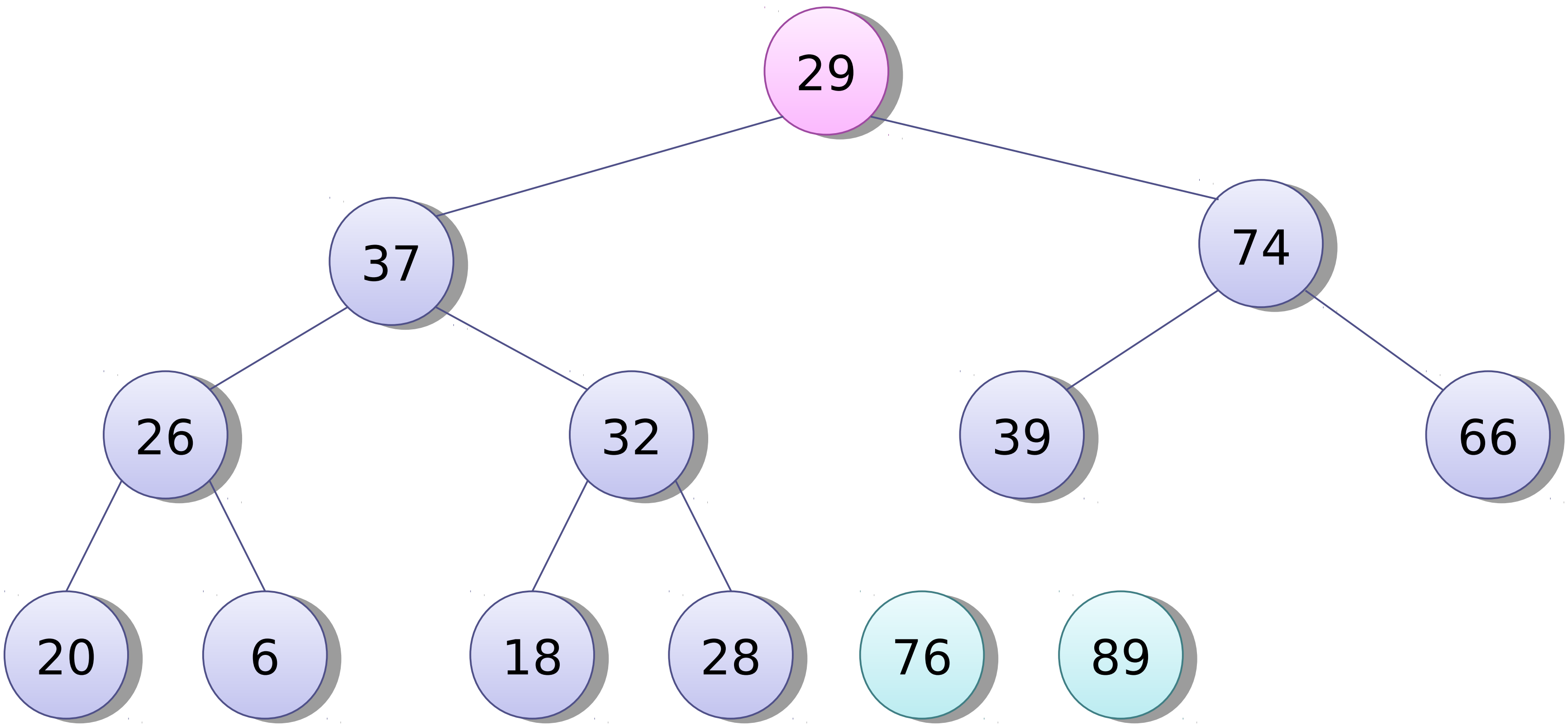
# Trace of heapsort

Step 1: swap maximum and last element; decrease size of heap by 1



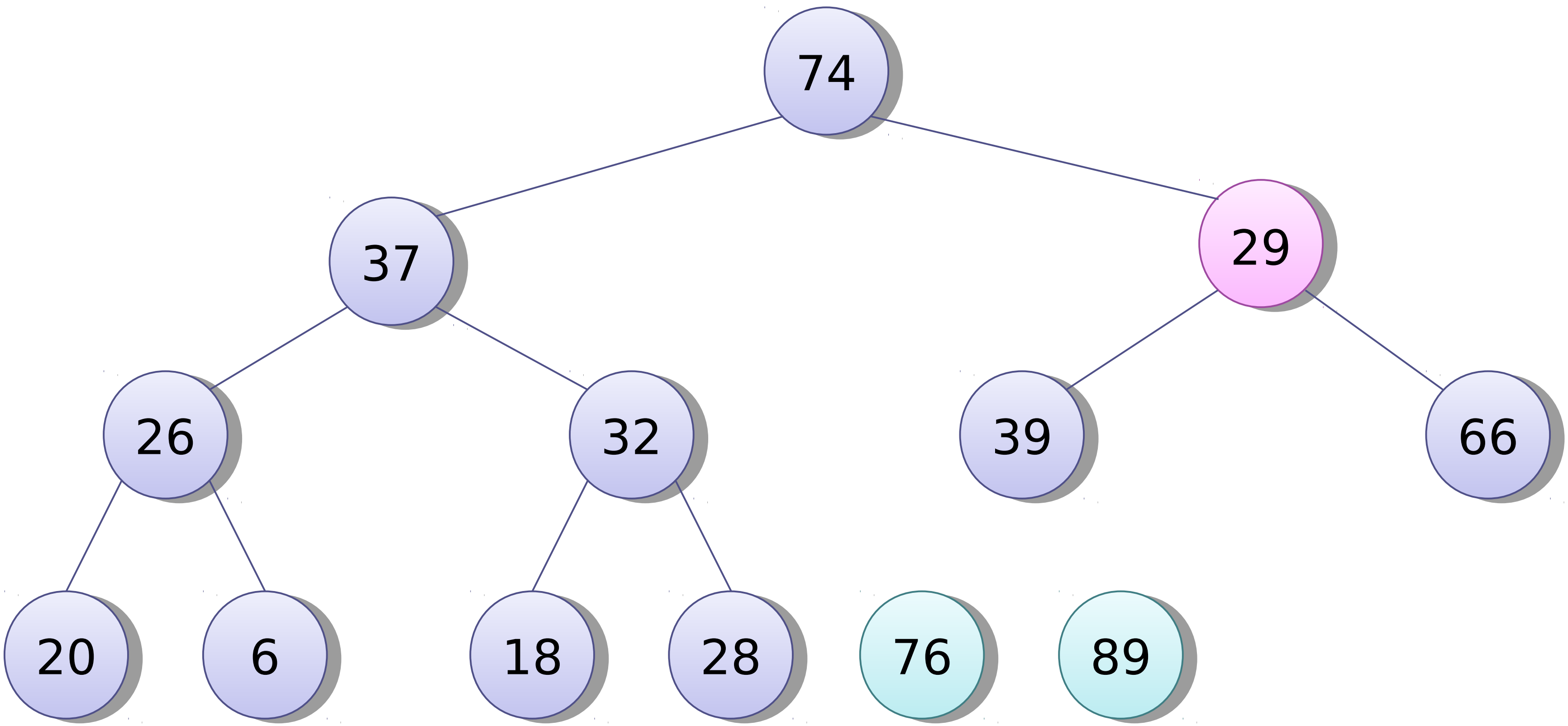
# Trace of heapsort

## Step 2: sift first element down



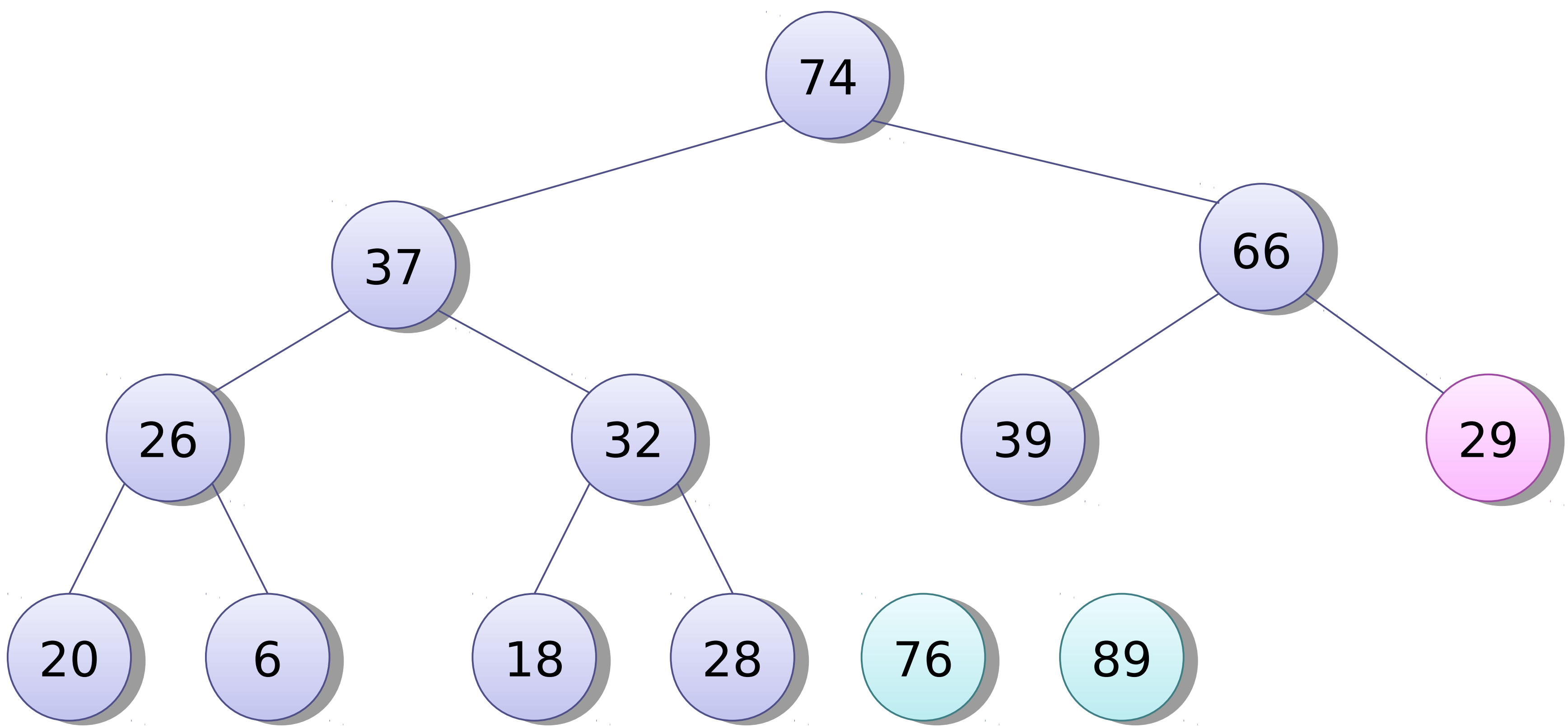
# Trace of heapsort

## Step 2: sift first element down

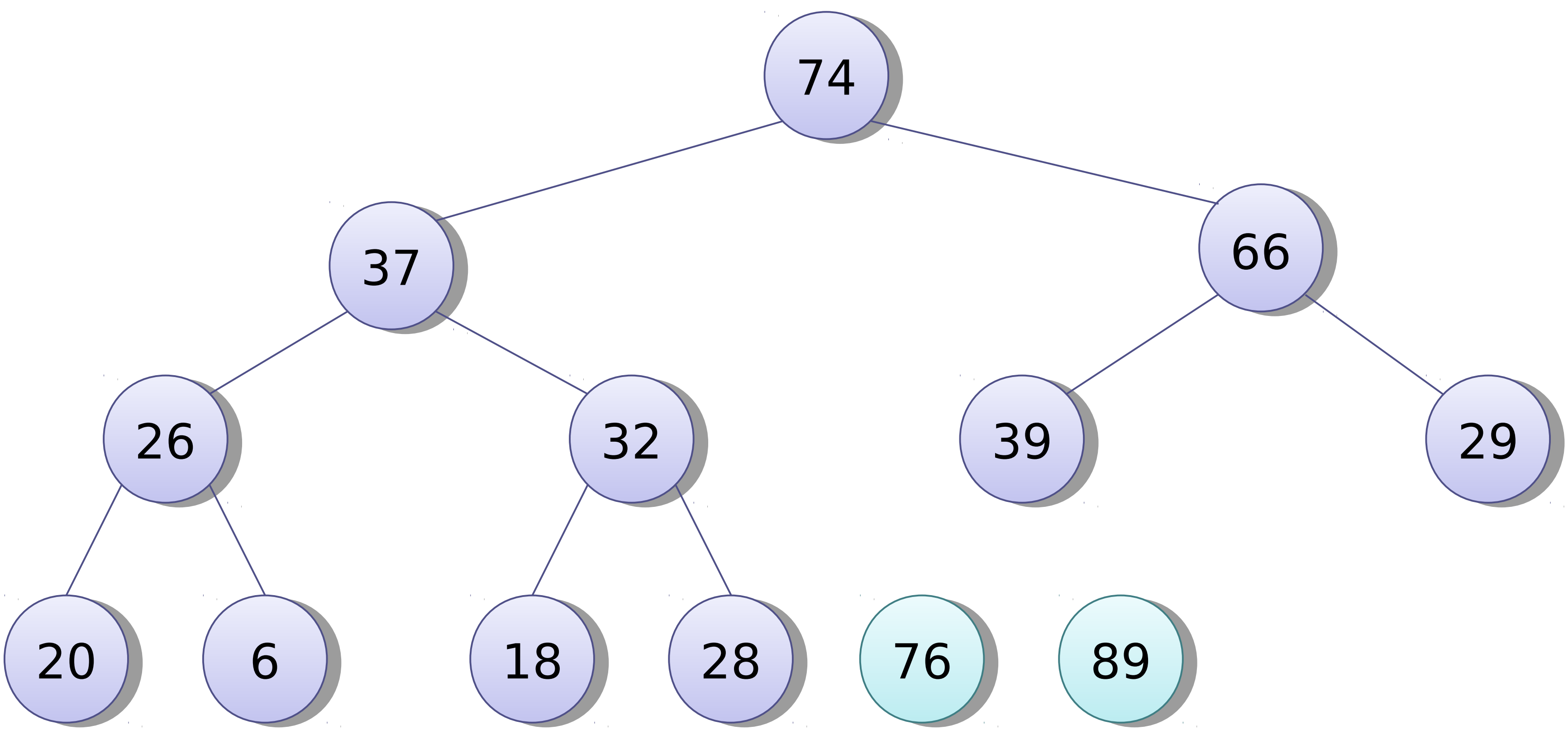


# Trace of heapsort

## Step 2: sift first element down

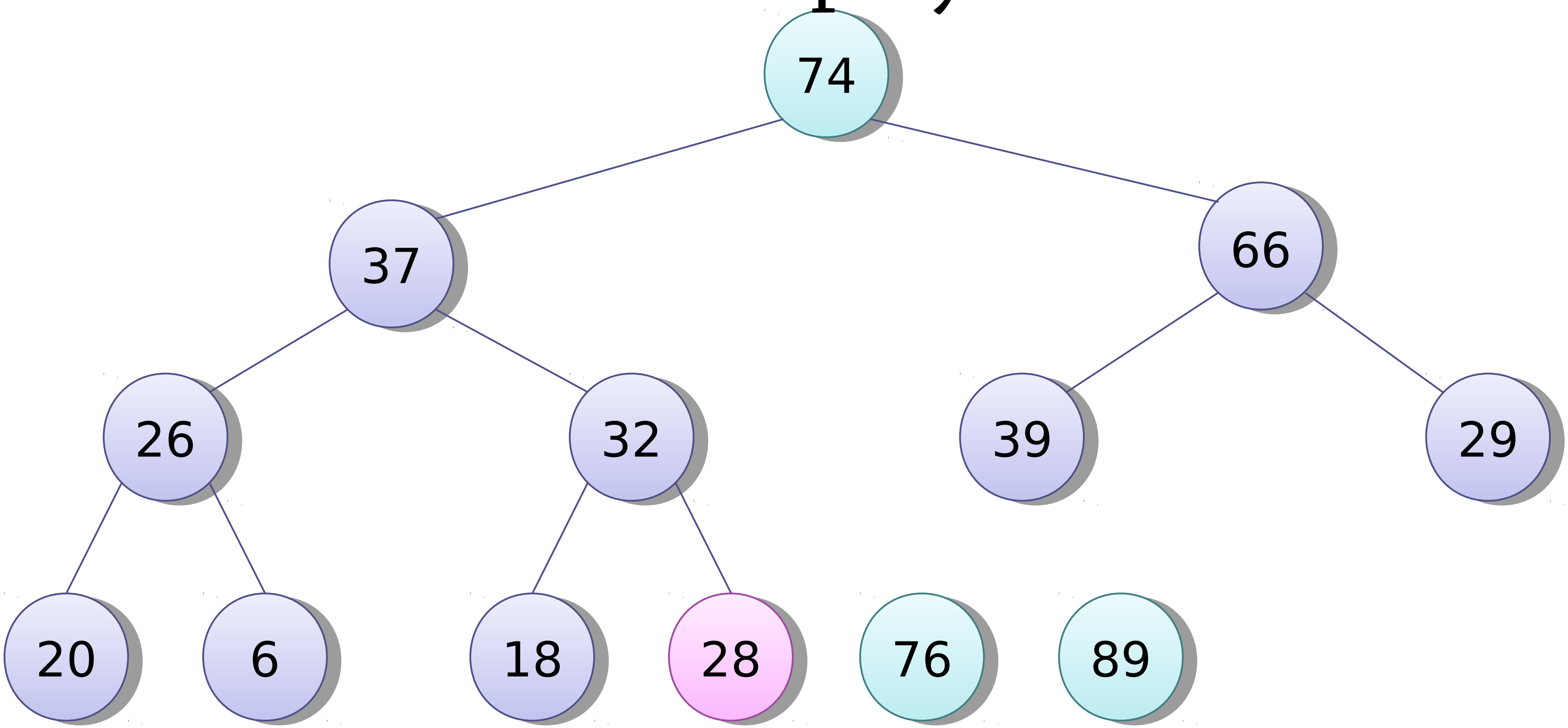


# Trace of heapsort



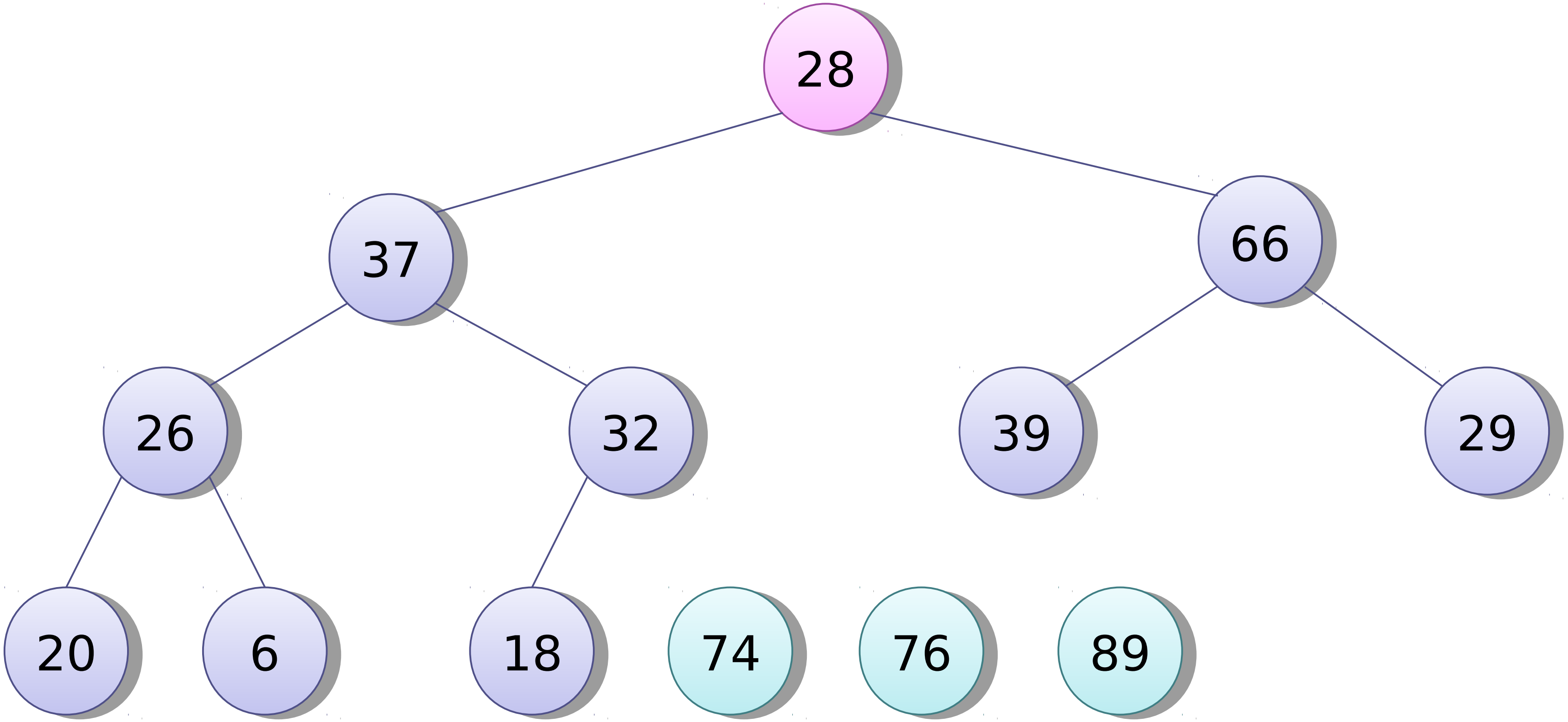
# Trace of heapsort

Step 1: swap maximum and last element; decrease size of heap by 1



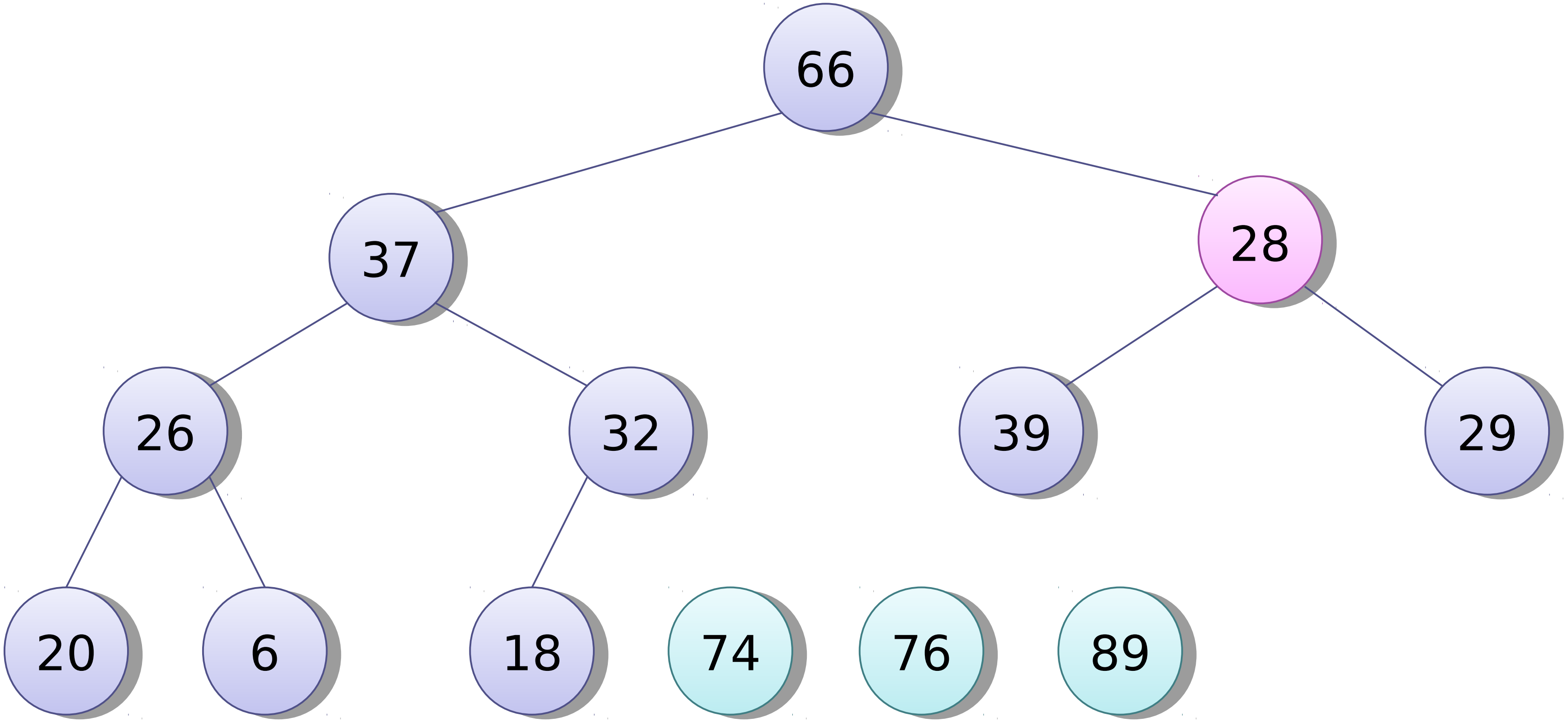
# Trace of heapsort

## Step 2: sift first element down



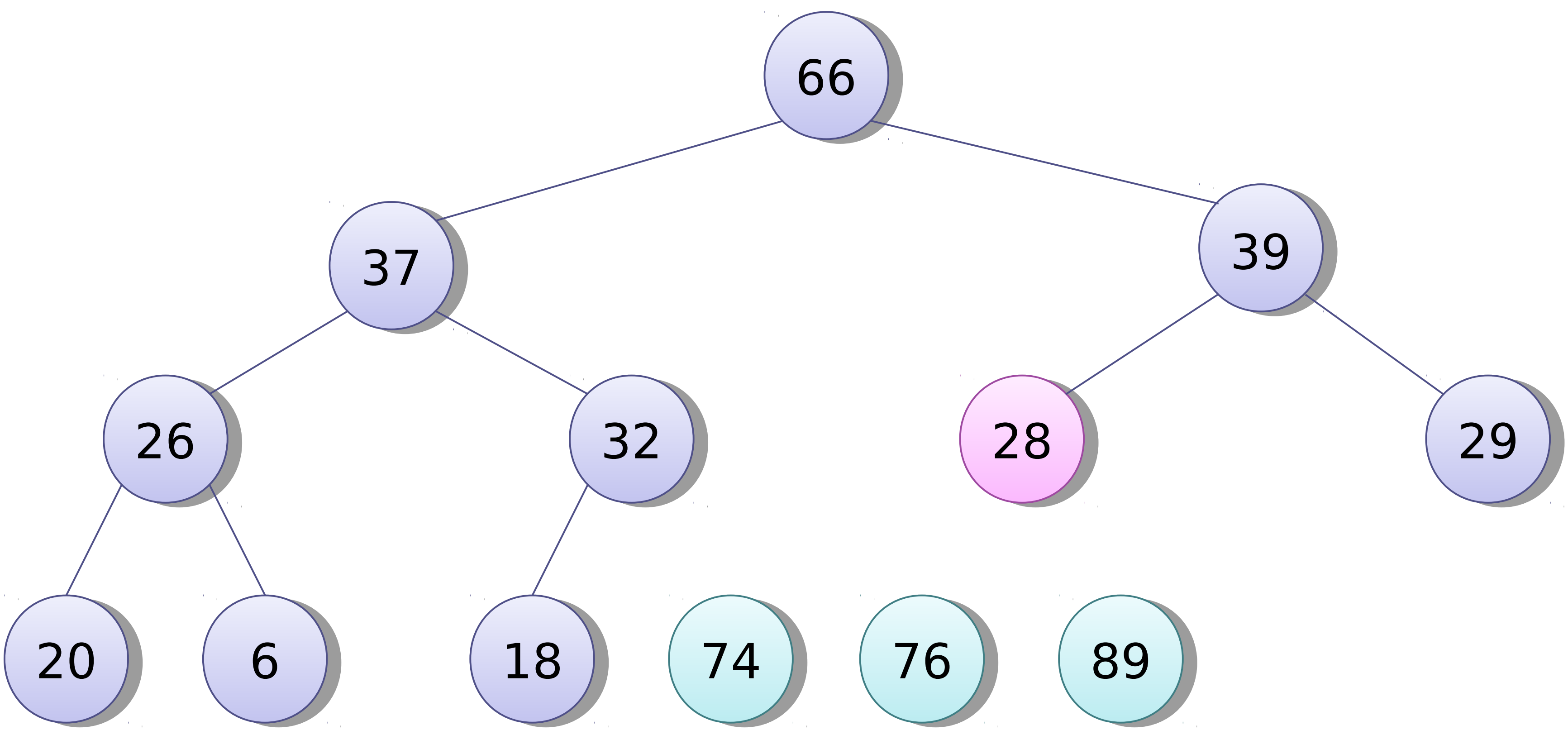
# Trace of heapsort

## Step 2: sift first element down

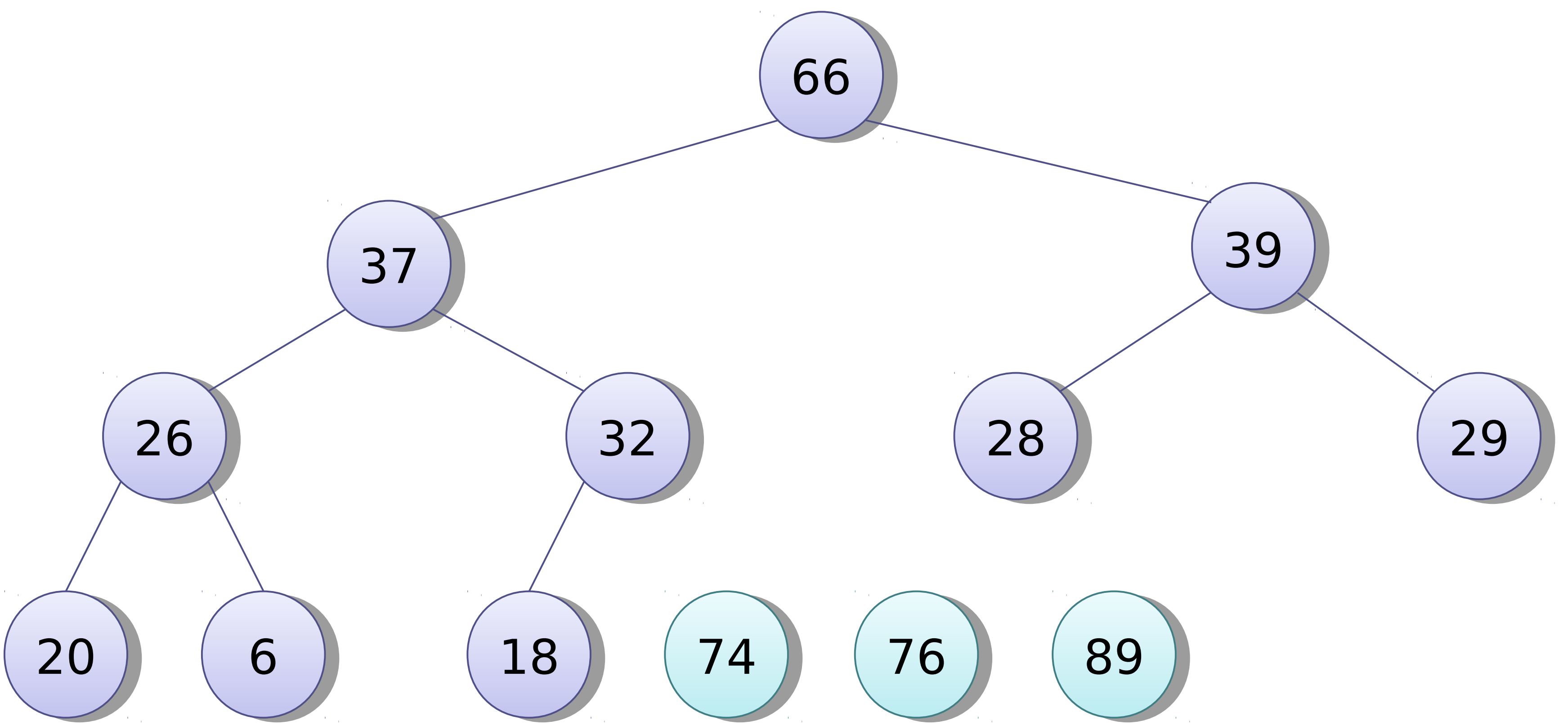


# Trace of heapsort

## Step 2: sift first element down



# Trace of heapsort



# 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: build a max heap, repeatedly remove last element and place at end of array

- Can be done in-place, O(n log n)

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)

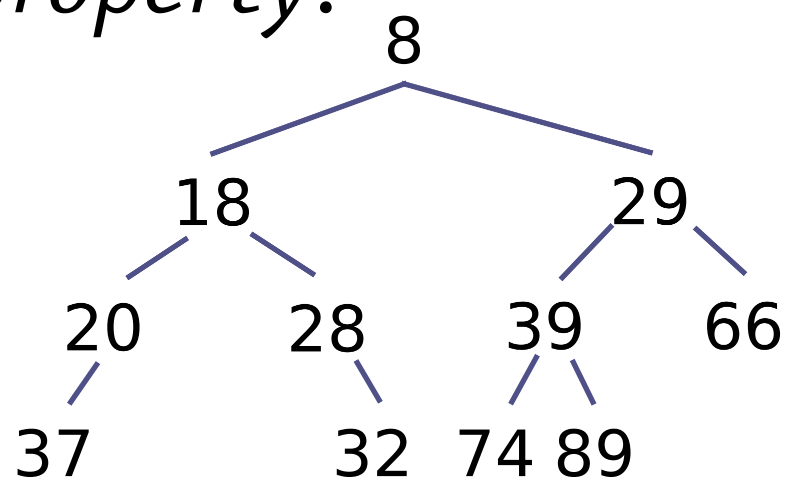
For binary heaps:

- 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!

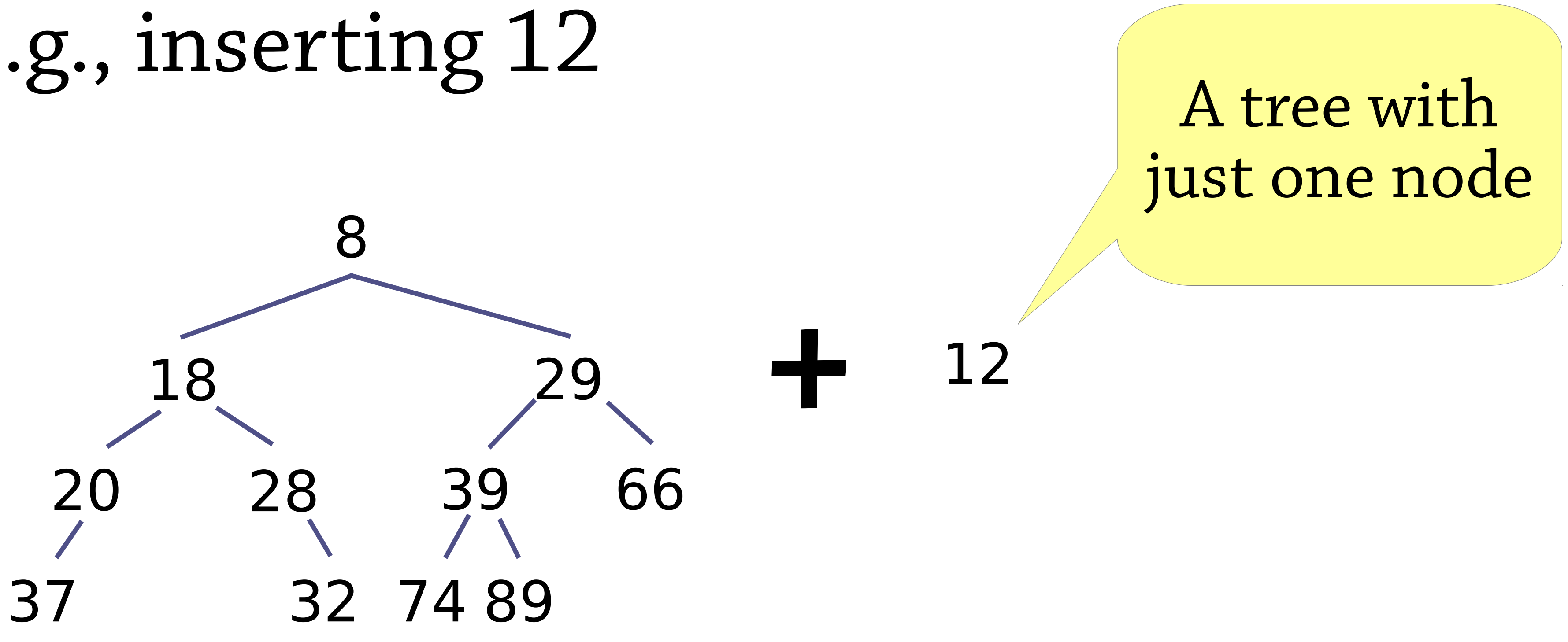
(We'll see how to implement merge later)

# Insertion

To insert a single element:

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

E.g., inserting 12

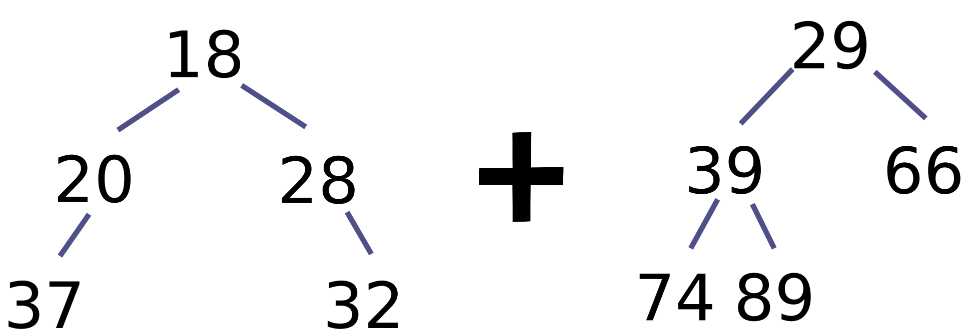


# Delete minimum

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

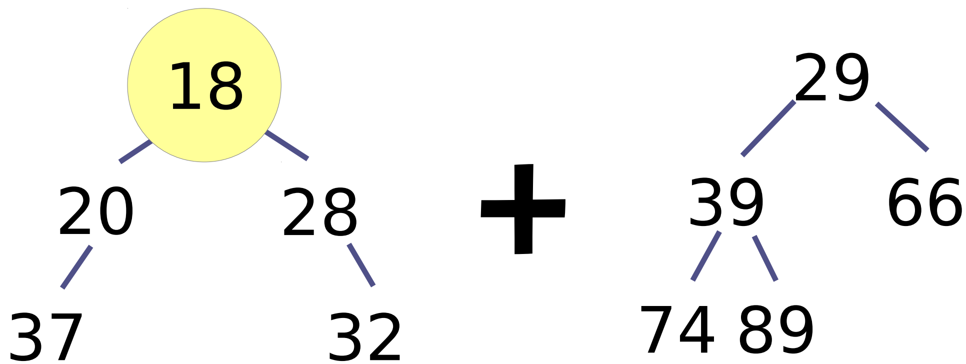
There are lots of heaps based on this idea:

- skew heaps, Fibonacci heaps, binomial heaps

We will study one: *leftist heaps*

# Naive merging

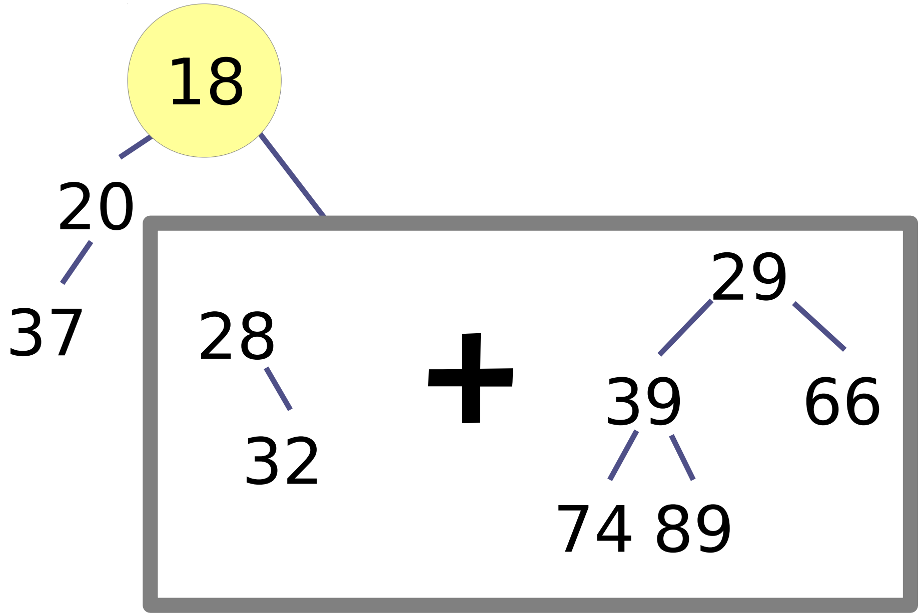
1. Look at the roots of the two trees




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

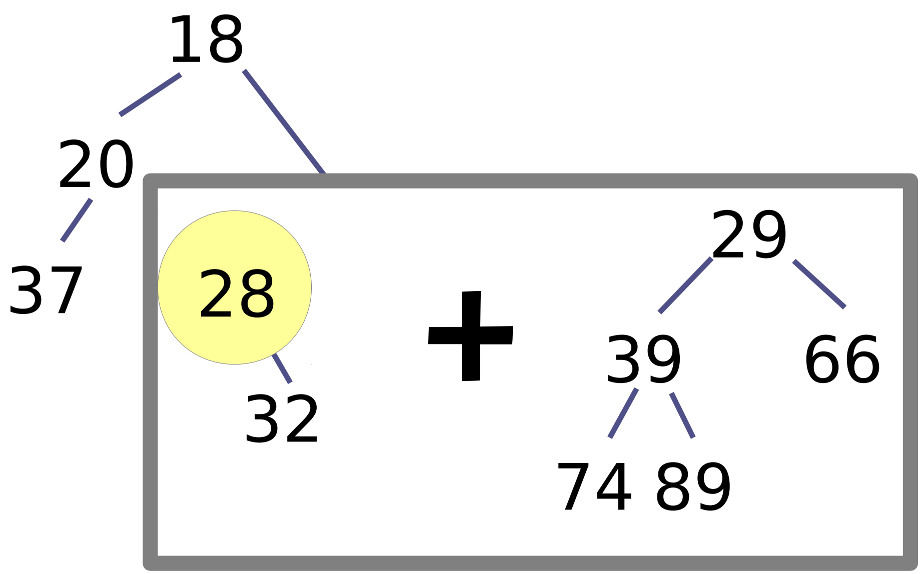
# Naive merging

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



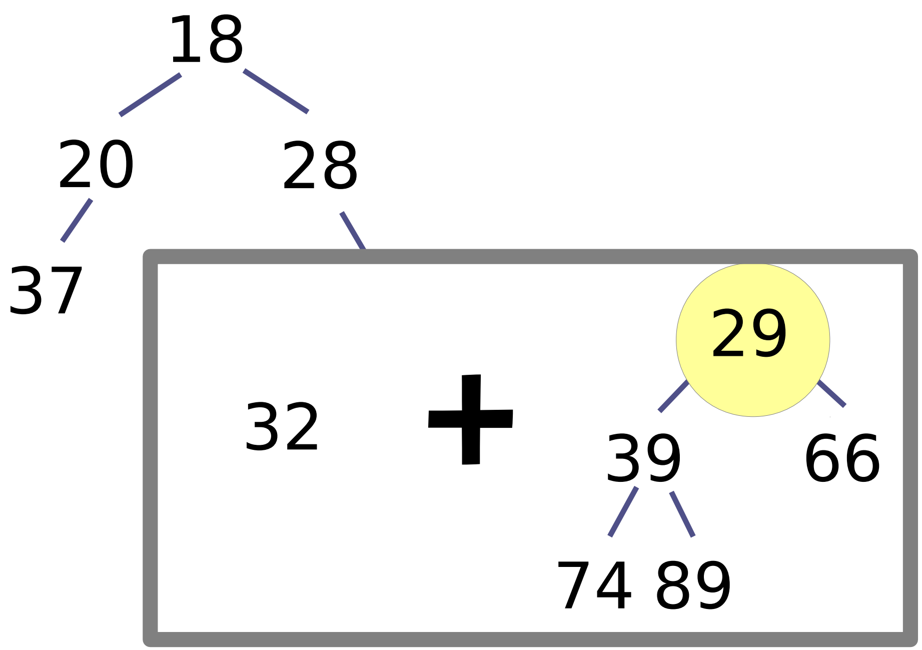
# Naive merging

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



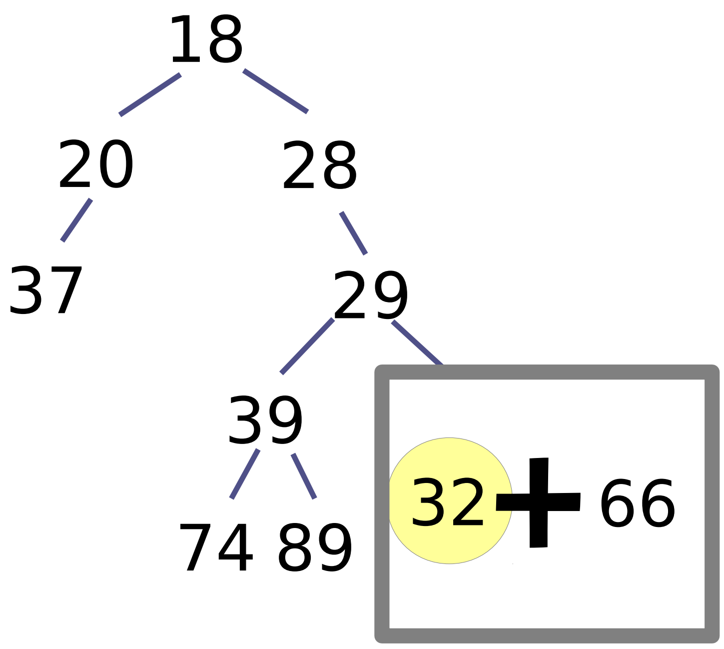
# Naive merging

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



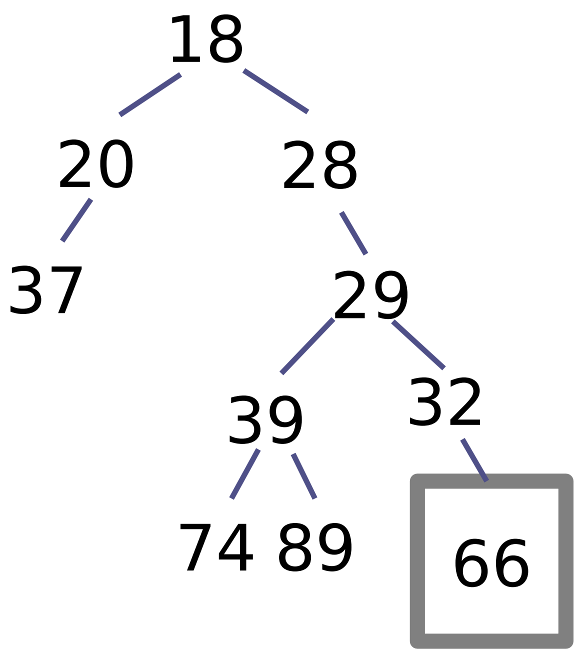
# Naive merging

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



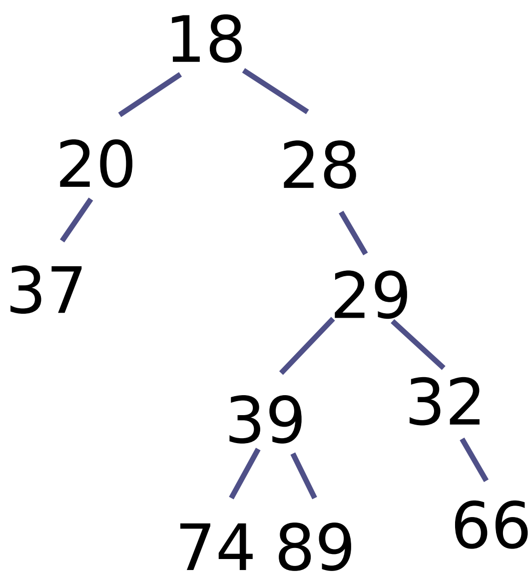
# Naive merging

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*

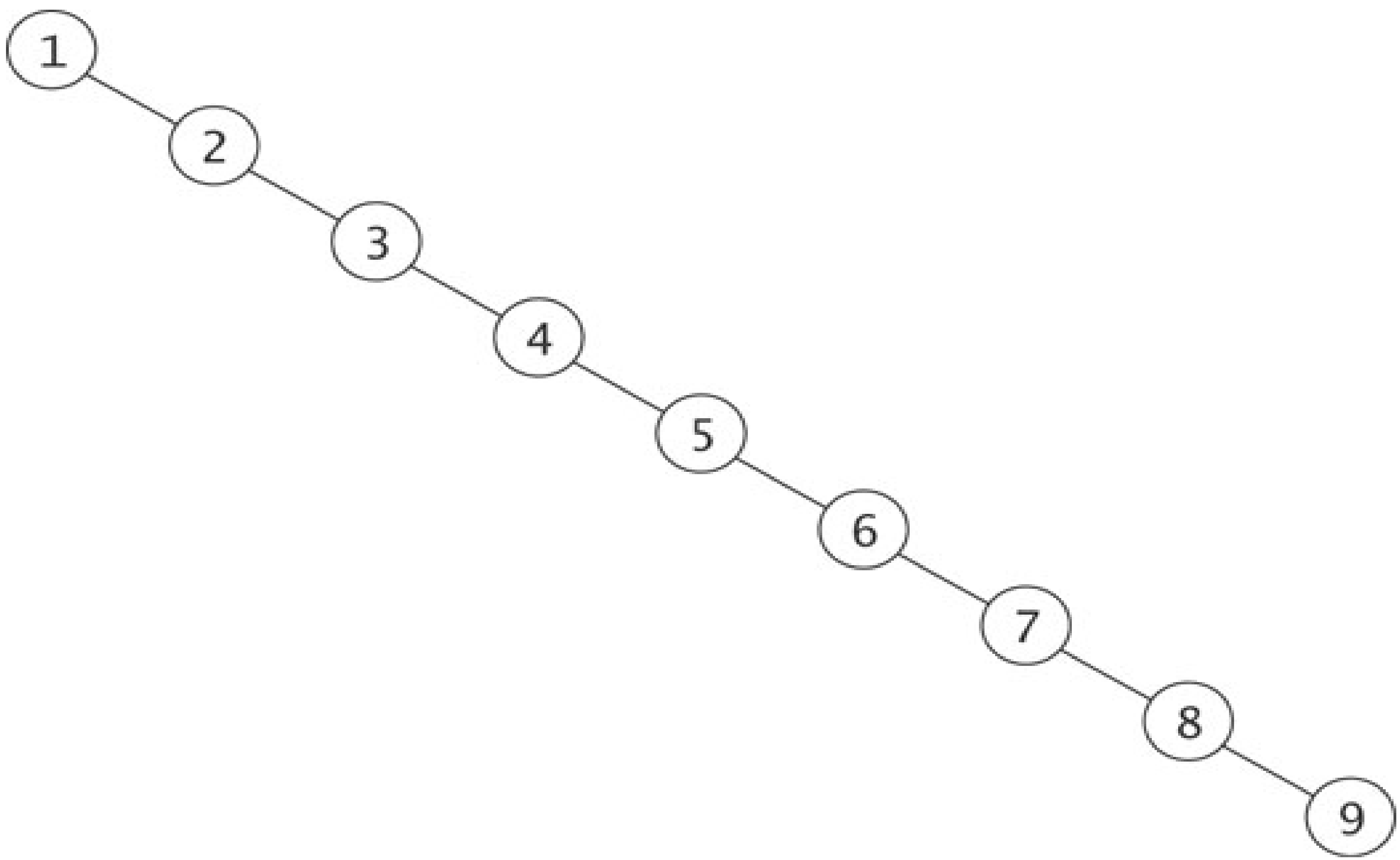
Complexity: O(m+n)

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

Logarithmic complexity for balanced trees, but can become linear is the trees are heavily “right-biased”

# Worst case for naïve merging

A heavily right-biased tree:



# Leftist heaps – observation

Naive merging is:

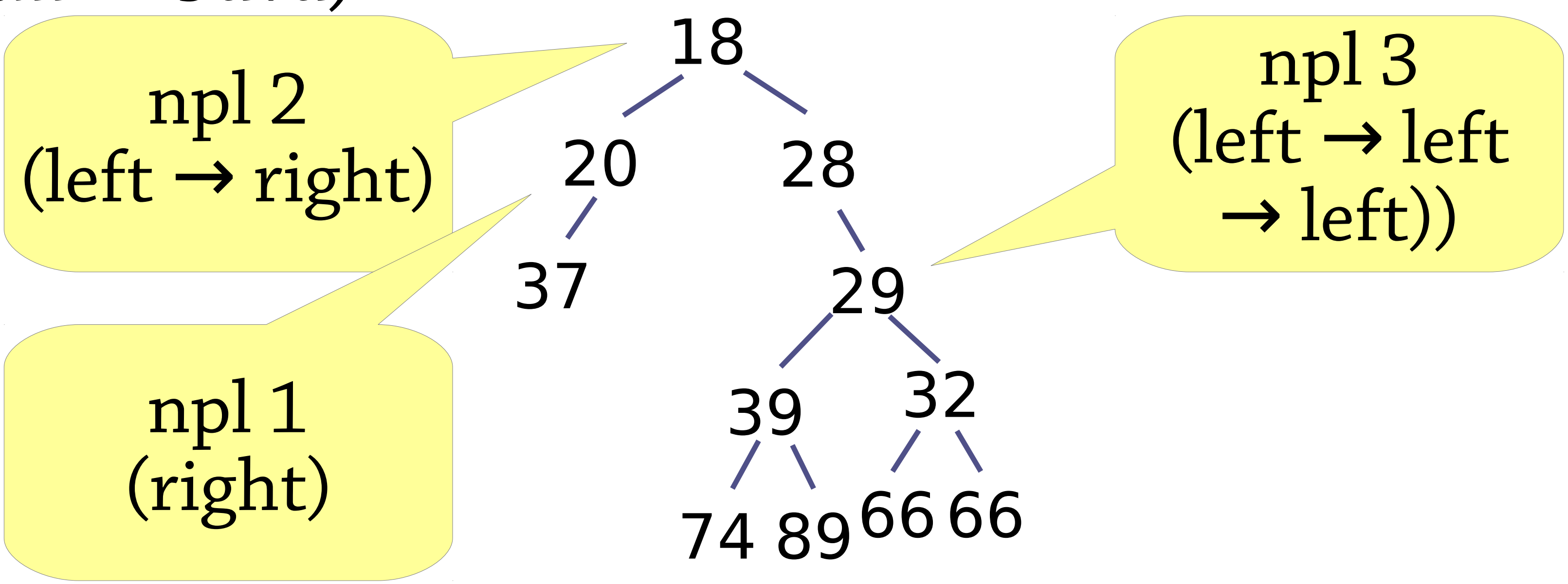
- bad (linear complexity) for right-biased trees
- good (logarithmic or better) for left-biased trees

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)

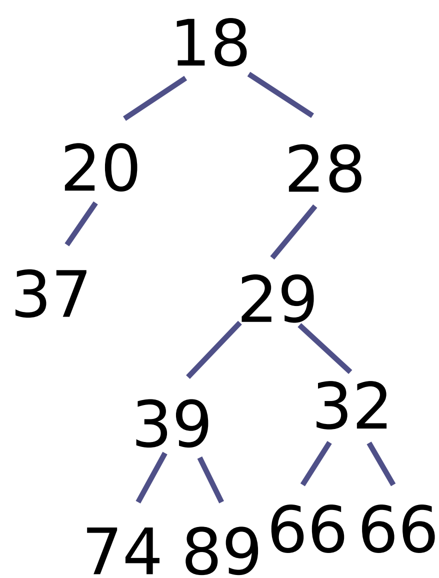



The null path length of *null* itself is 0

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!

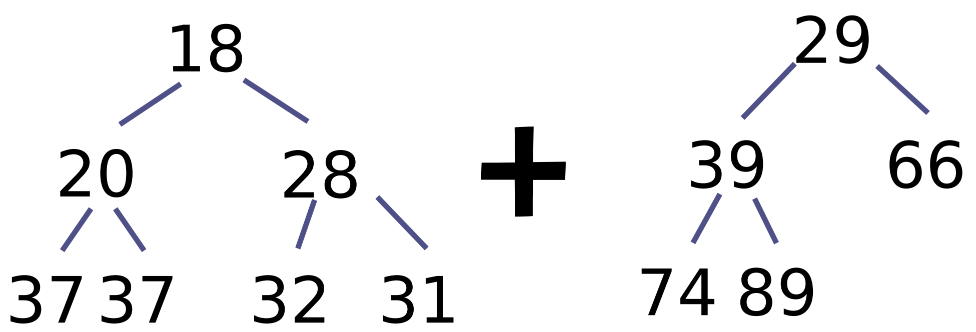
- When we descend into the right child, its npl might increase, and become greater than the left child

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!

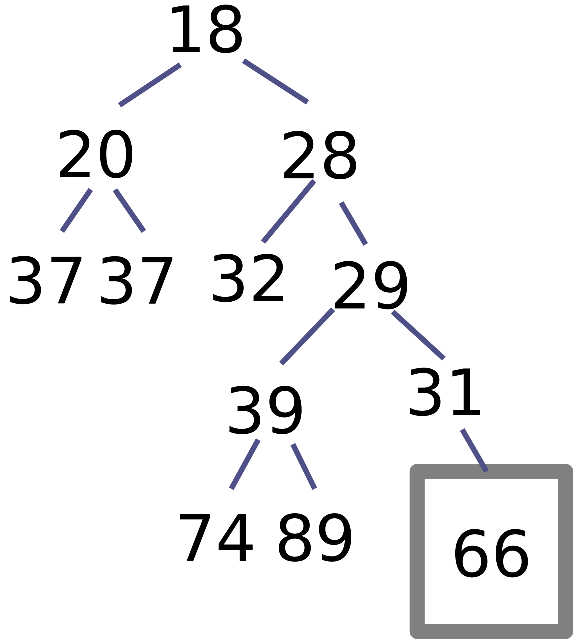
# Leftist merging

1. Start with naïve merging from earlier



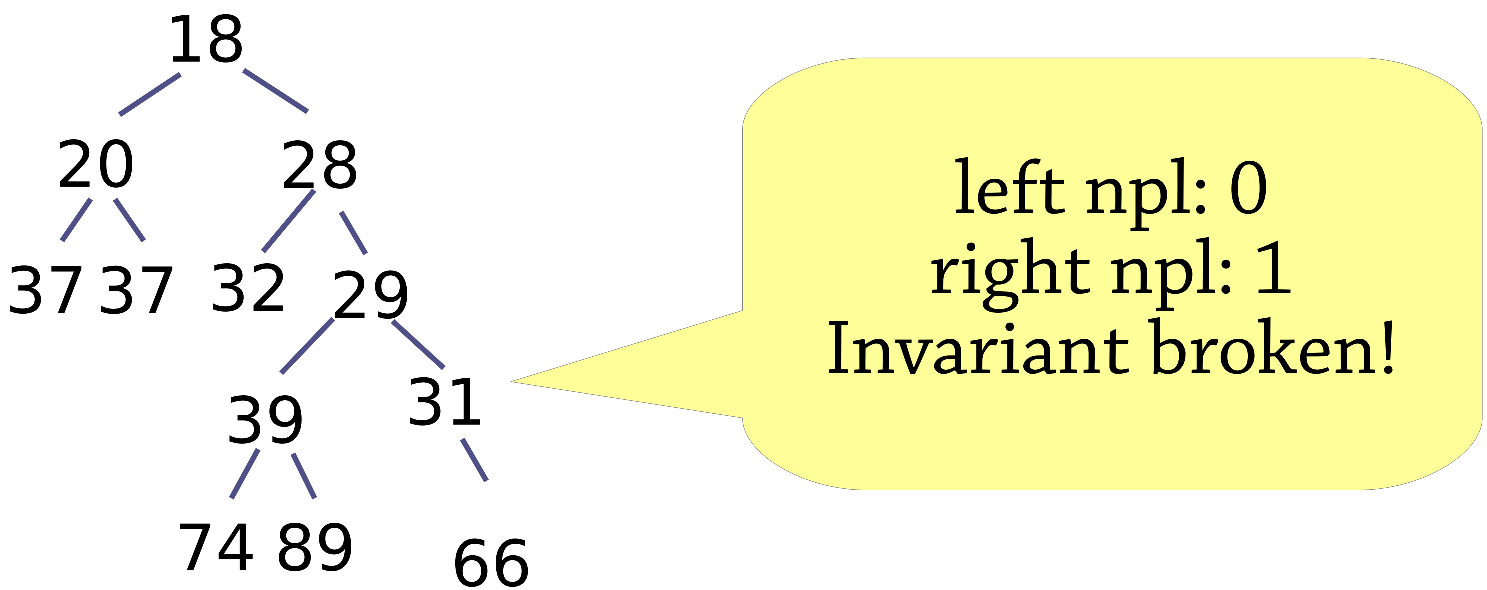
# Leftist merging

2. The recursion "bottomed out" at 66 here



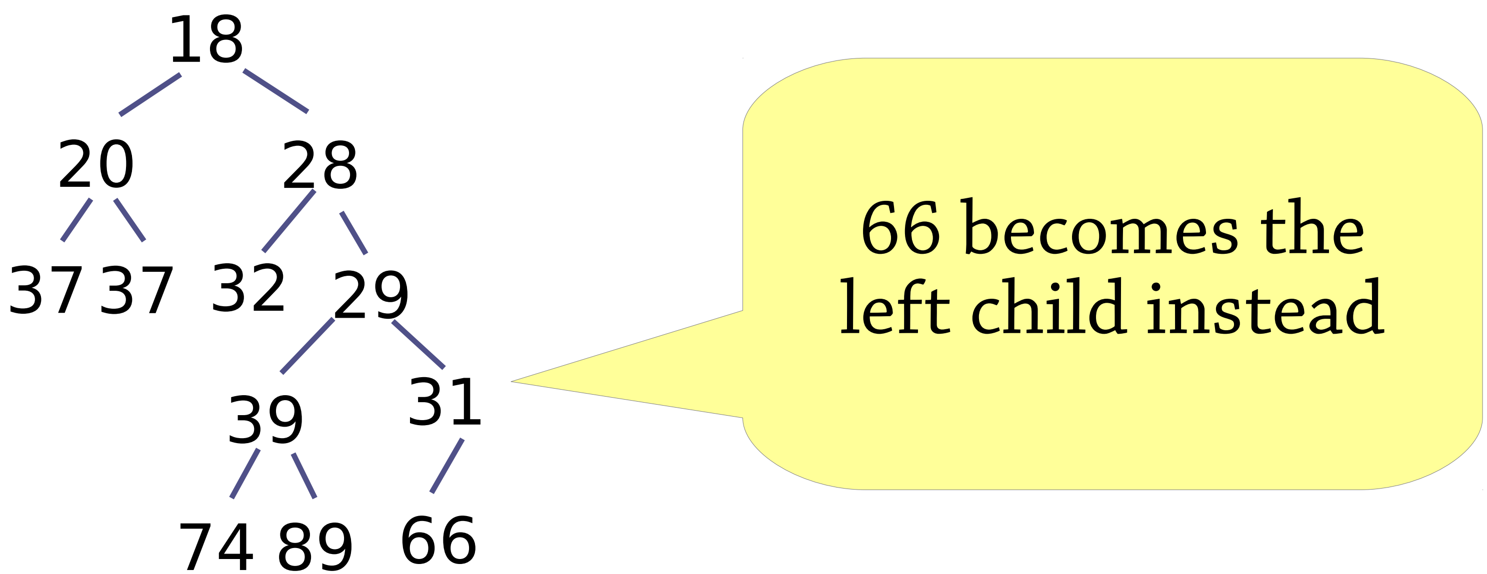
# Leftist merging

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



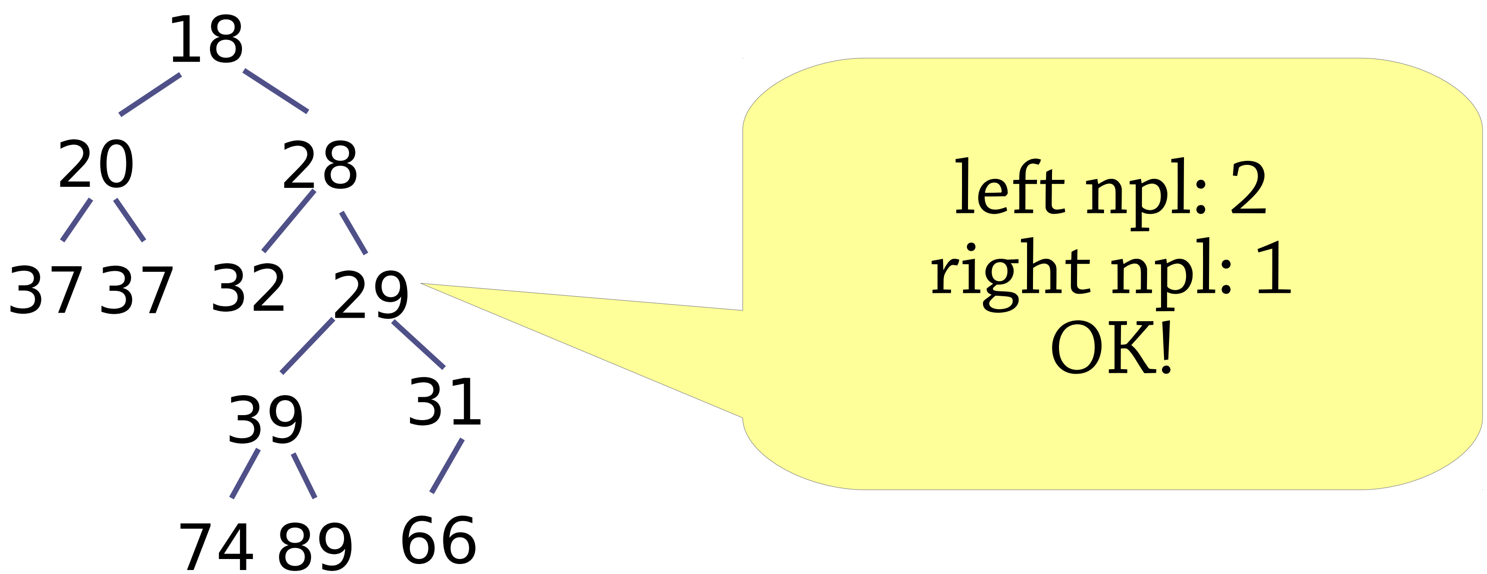
# Leftist merging

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



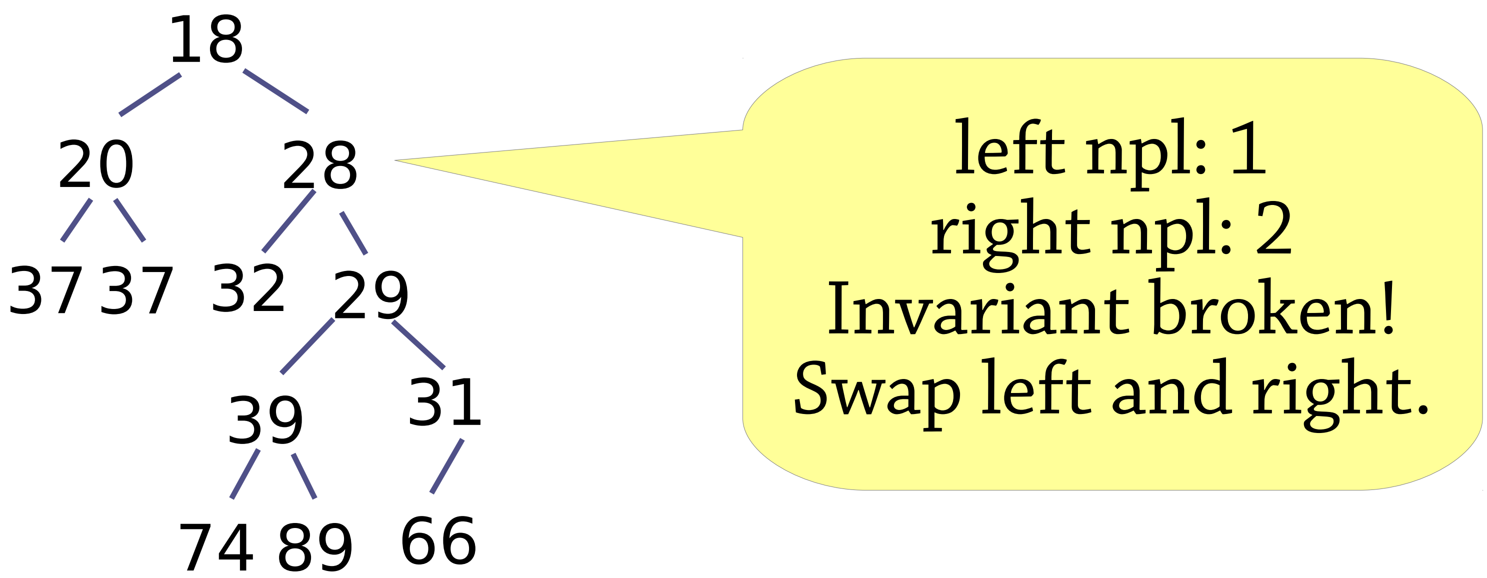
# Leftist merging

5. Go up again and repeat!



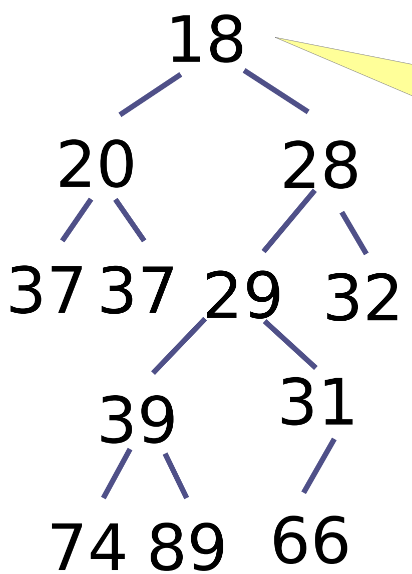
# Leftist merging

5. Go up again and repeat!



# Leftist merging

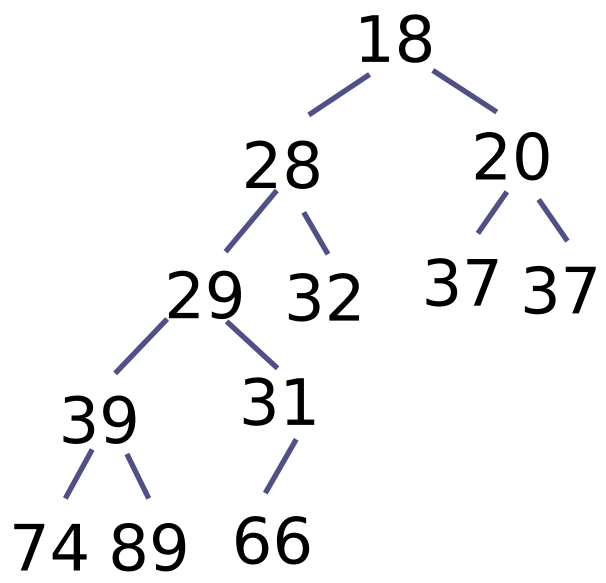
5. Go up again and repeat!




left npl: 1
right npl: 2
Invariant broken!
Swap left and right.

# 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