Priority queues, binary heaps, and heapsort (6.9, 21.1 – 21.5)

Priority queue

A *priority queue* is an ADT supporting three operations:

- `insert`: add an element
- `findMin`: return the smallest element
- `deleteMin`: remove the smallest element

Allows you to maintain a set of elements while always knowing the smallest one

Java: `PriorityQueue<E>` class defining `add`, `element` and `remove`

Implementing a priority queue

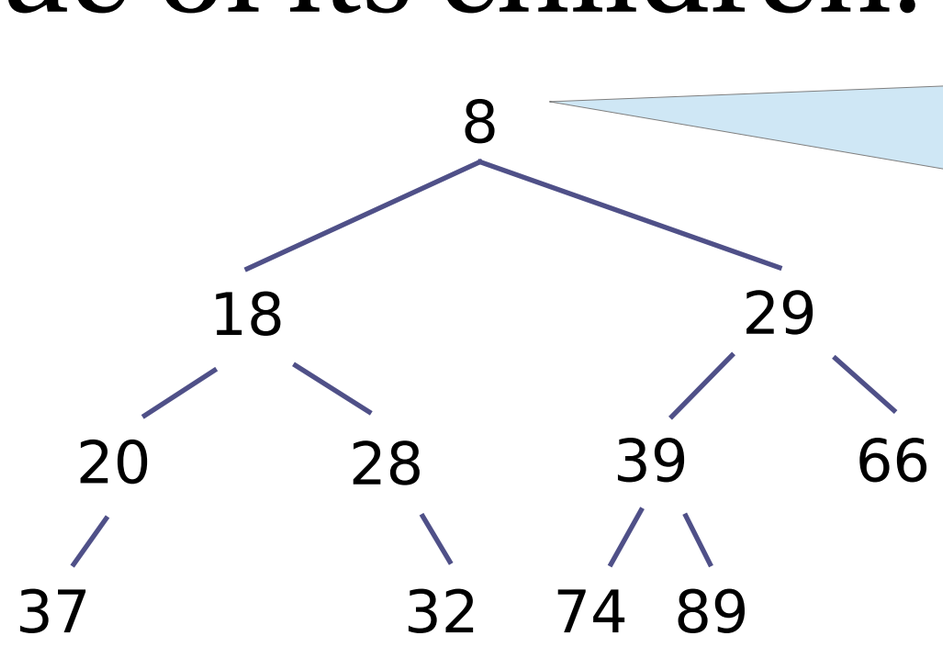
Several possible ways:

- An unsorted array?
- A sorted array?
- A binary search tree?
- A balanced binary search tree?

A nicer way: a *binary heap*

The heap property

A tree satisfies the *heap property* if the value of each node is less than (or equal to) the value of its children:

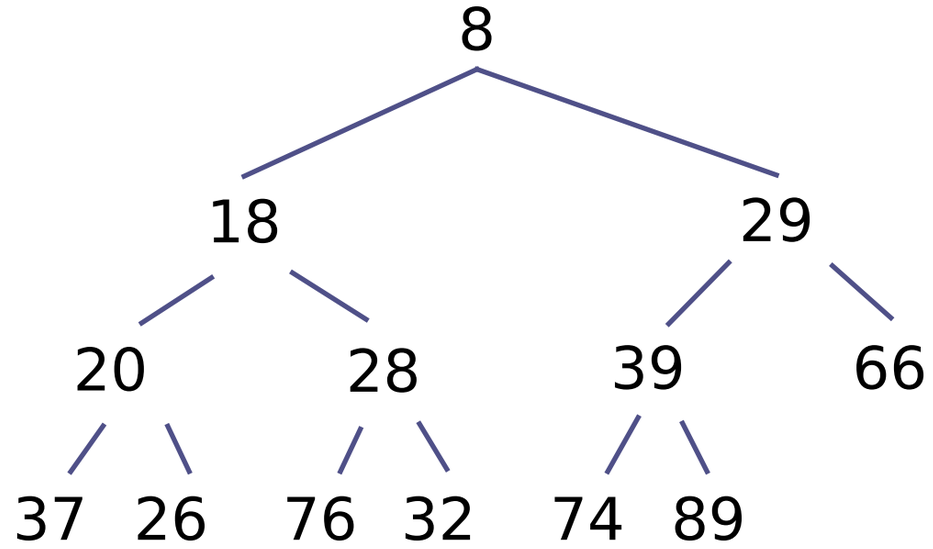


Root node is the smallest – can do findMin in O(1) time

What can we say about the root node?

Binary heap

A *binary heap* is a *complete* binary tree that satisfies the heap property:



Note: it is not a binary search tree!

Complete means that all levels except the bottom one are full, and the bottom level is filled from left to right (see diagram)

Binary heap invariant

The binary heap invariant:

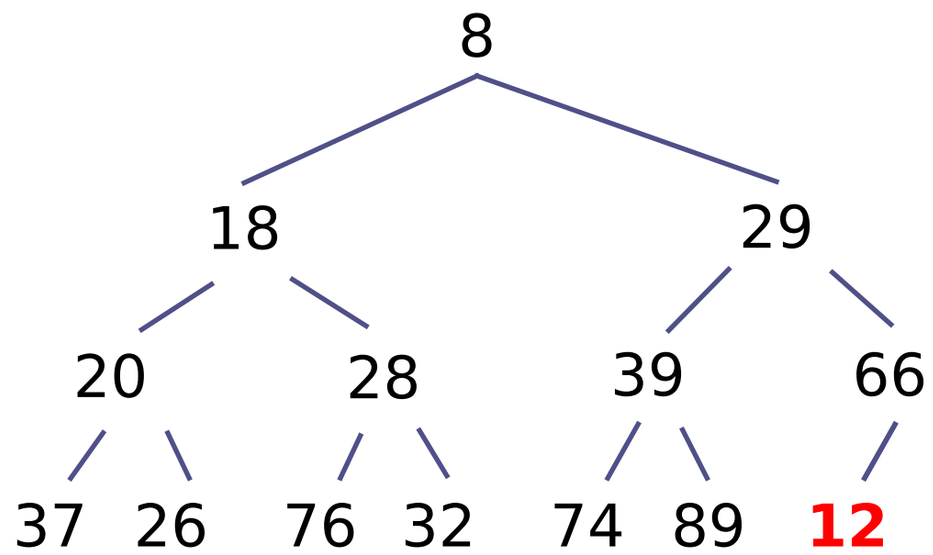
- The tree must be *complete*
- It must have the *heap property* (each node is less than or equal to its children)

Remember, all our operations must preserve this invariant

(Why this invariant? See later!)

Adding an element to a binary heap

Step 1: insert the element at the next empty position in the tree

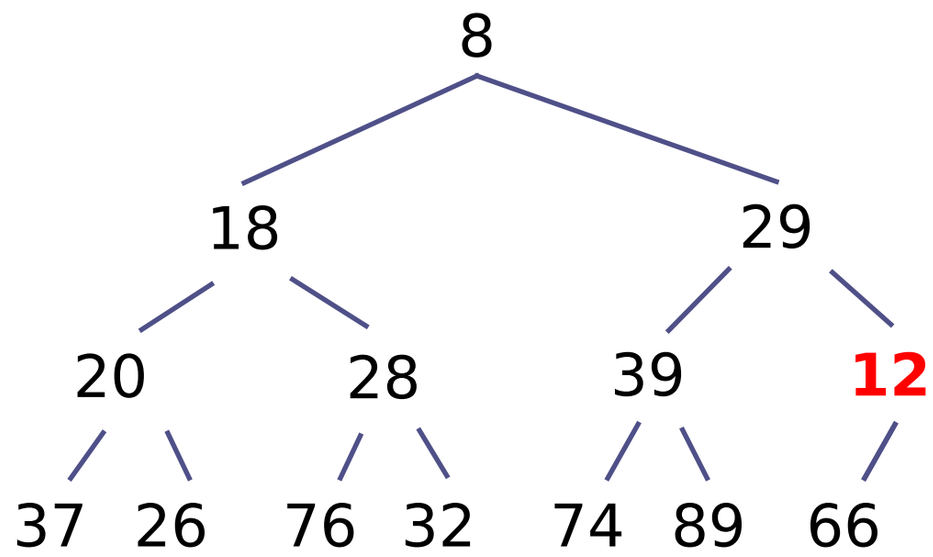


This might break the heap invariant!

In this case, 12 is less than 66, its parent.

Adding an element to a binary heap

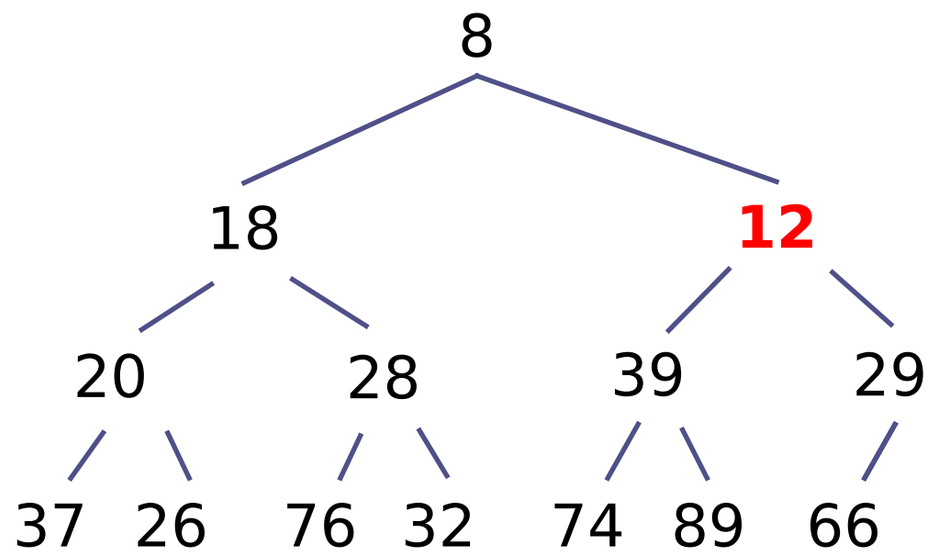
Step 2: if the new element is less than its parent, swap it with its parent



The invariant is still broken, since 12 is less than 29, its new parent

Adding an element to a binary heap

Repeat step 2 until the new element is greater than or equal to its parent.

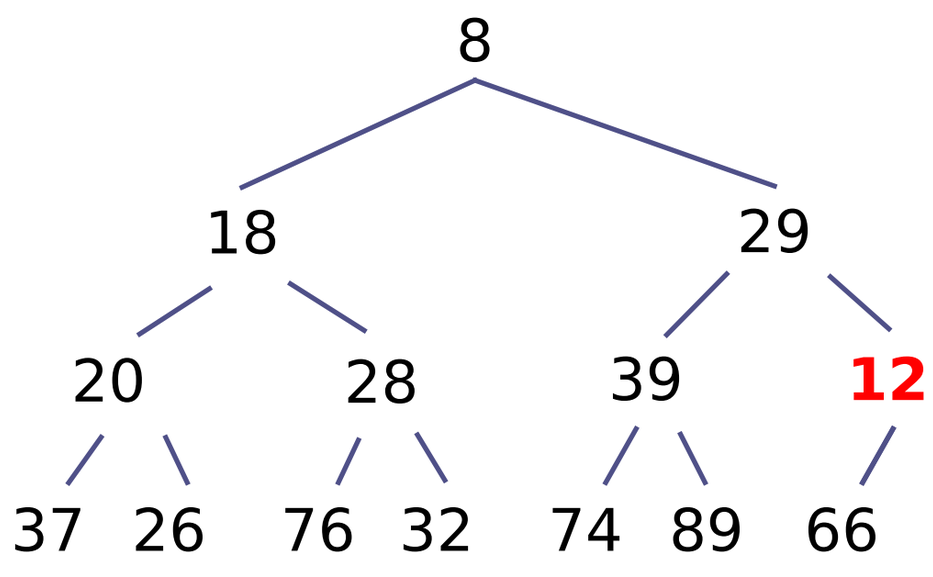


Now 12 is in its right place, and the invariant is restored. (Think about why this algorithm restores the invariant.)

Why this works

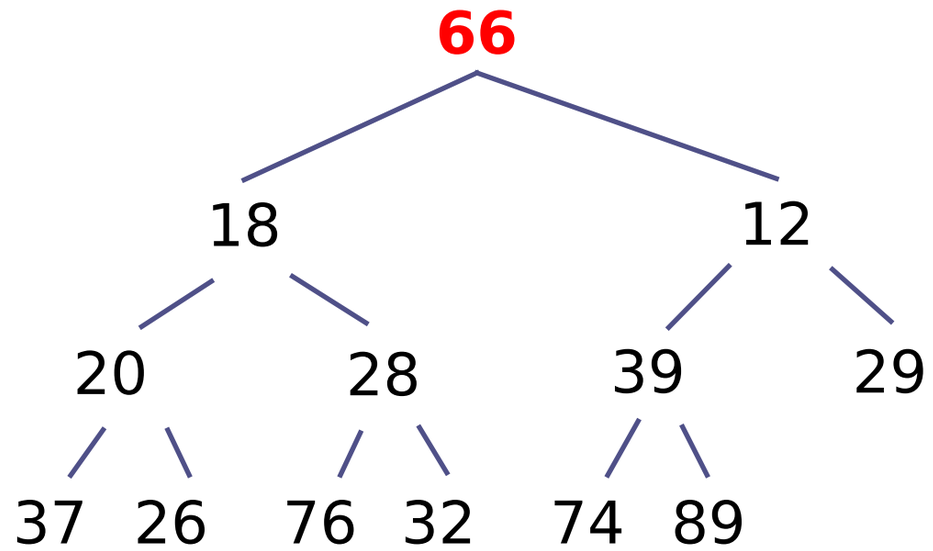
At every step, the heap property almost holds *except* that the new element might be less than its parent

After swapping the element and its parent, still only the new element can be in the wrong place (why?)



Removing the minimum element

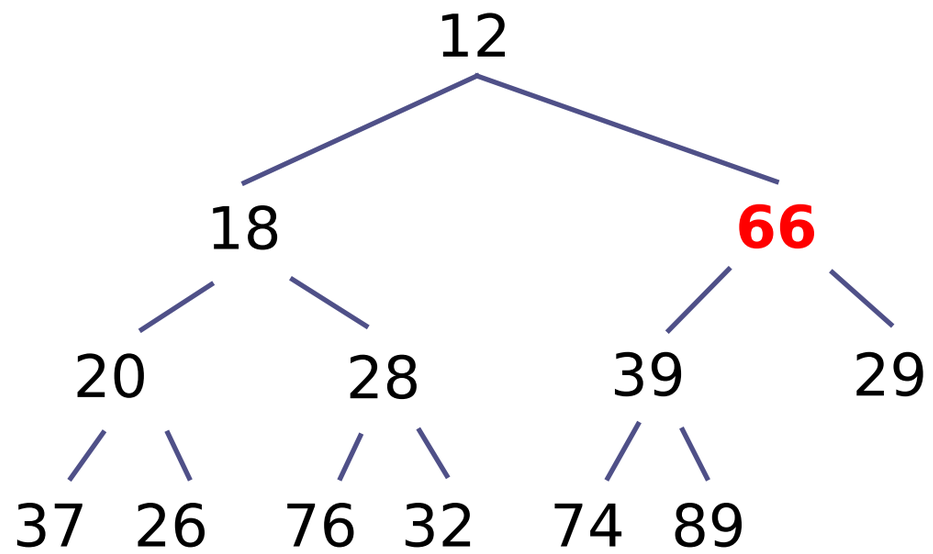
Step 1: replace the root element with the *last element* in the heap



The invariant is broken, because 66 is greater than its children

Removing the minimum element

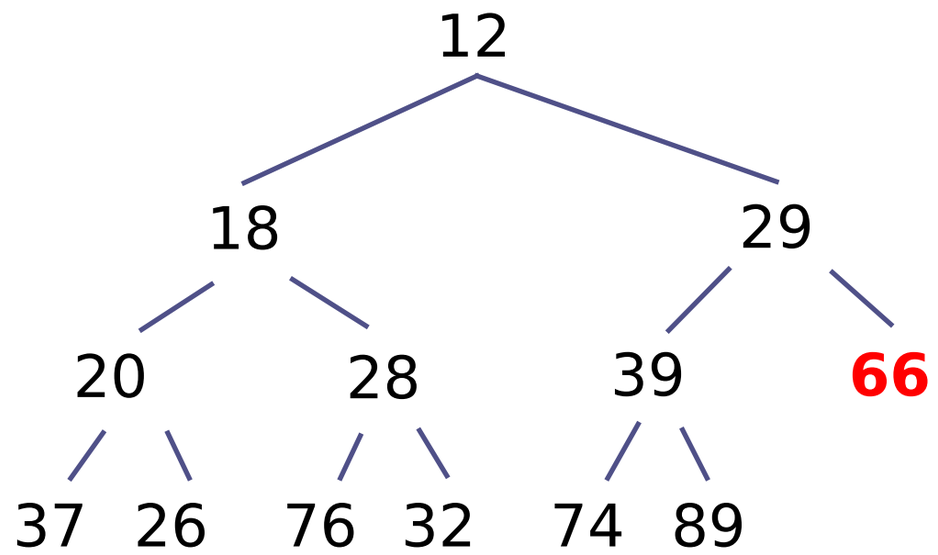
Step 2: if the moved element is greater than its children, swap it with its *least child*



(Why not its greatest child?)

Removing the minimum element

Step 3: repeat until the moved element is less than or equal to its children



Sifting

Two useful operations in implementing a heap

Sift up: if an element might be less than its parent, i.e. "too low" (used in insert)

- Repeatedly swap the element with its parent

Sift down: if an element might be greater than its children, i.e. "too high" (used in deleteMin)

- Repeatedly swap the element with its least child

Summary so far

Binary heap: complete binary tree where every node is less than or equal to its children

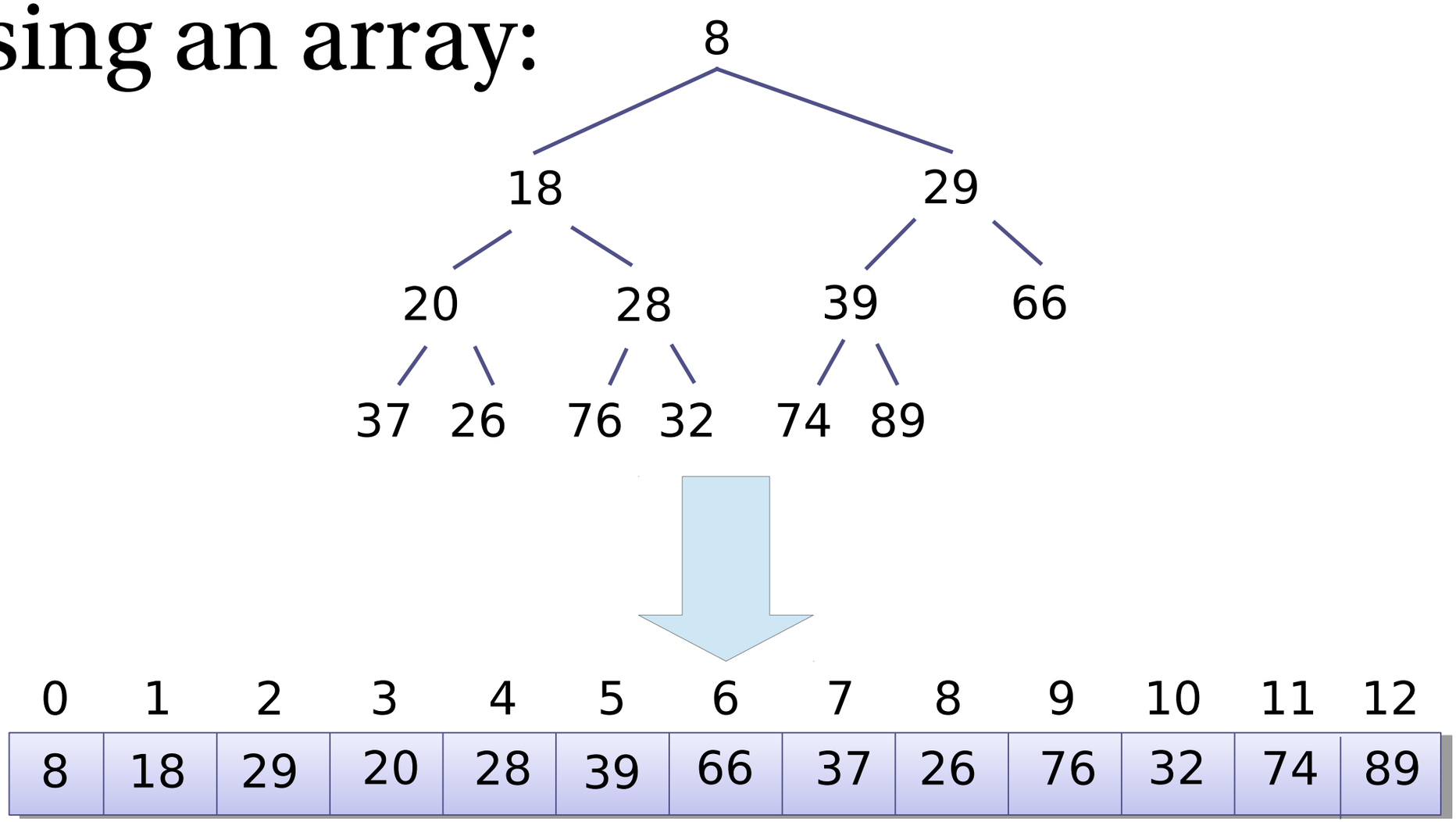
Minimum is root node

Insert: add to end of tree and *sift up* (called *percolate up* in book)

Delete minimum: swap with final element and *sift down* (called *percolate down* in book)

Binary heaps are arrays!

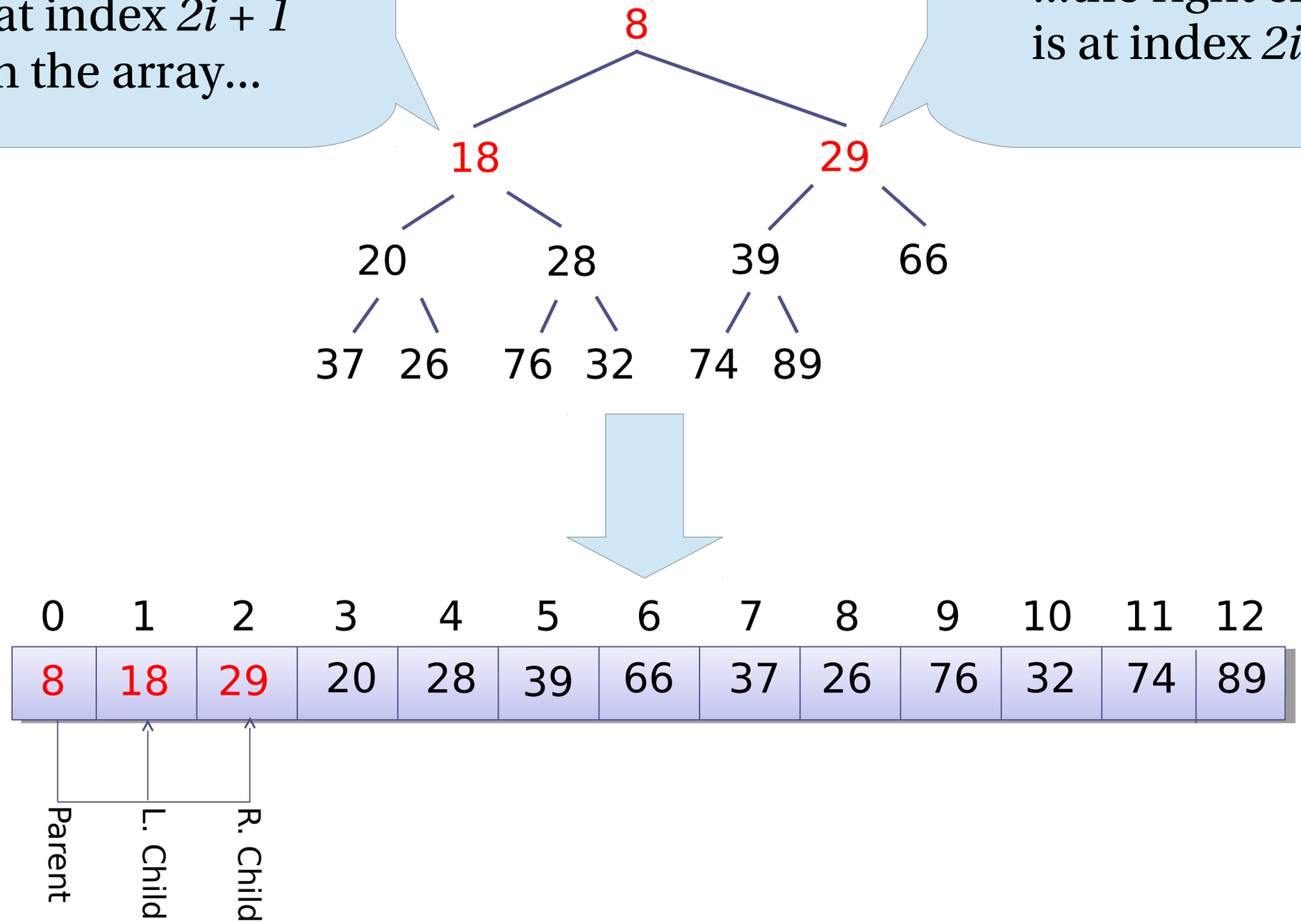
A binary heap is really implemented using an array:



Child positions

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

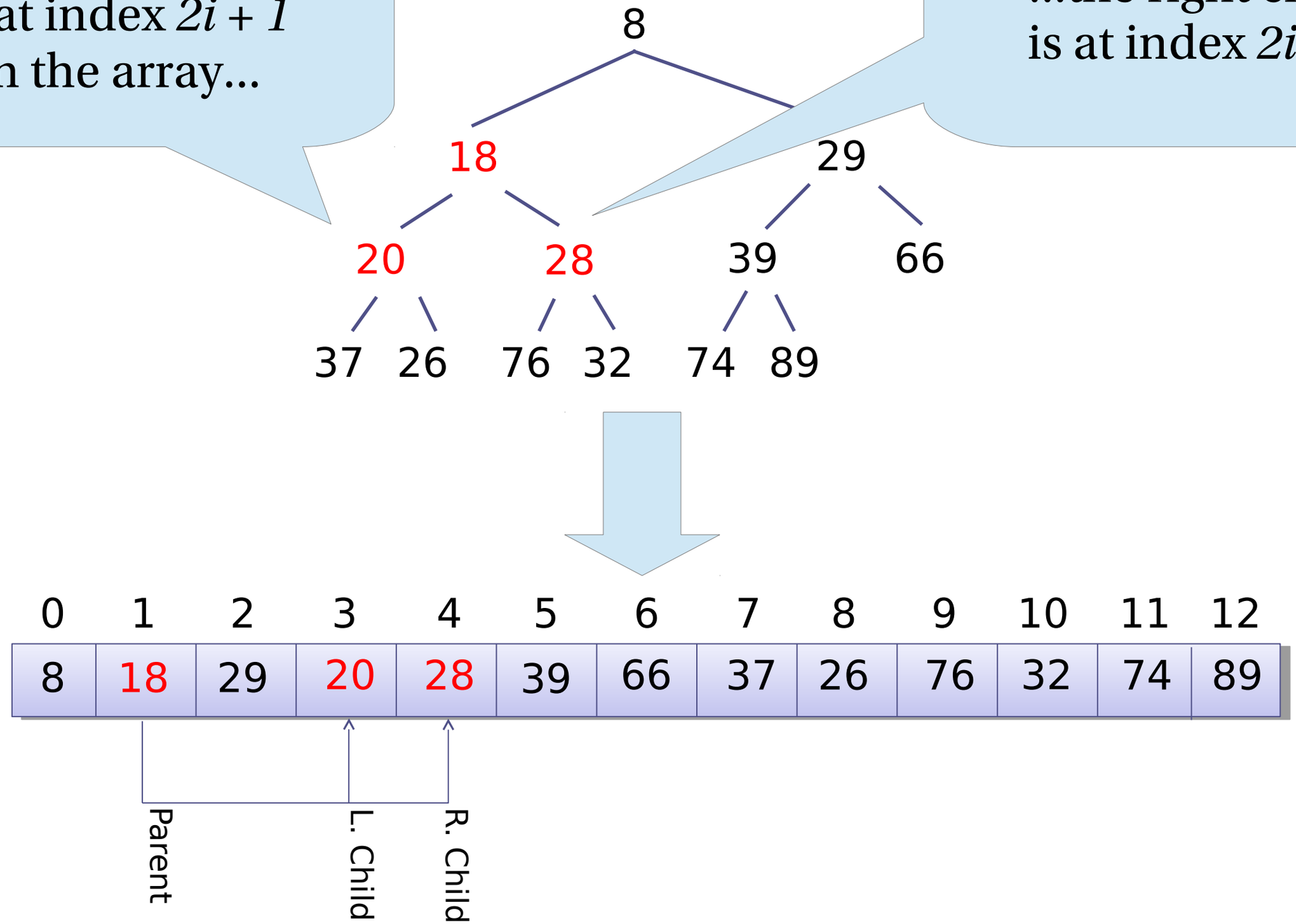
...the right child is at index $2i + 2$



Child positions

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

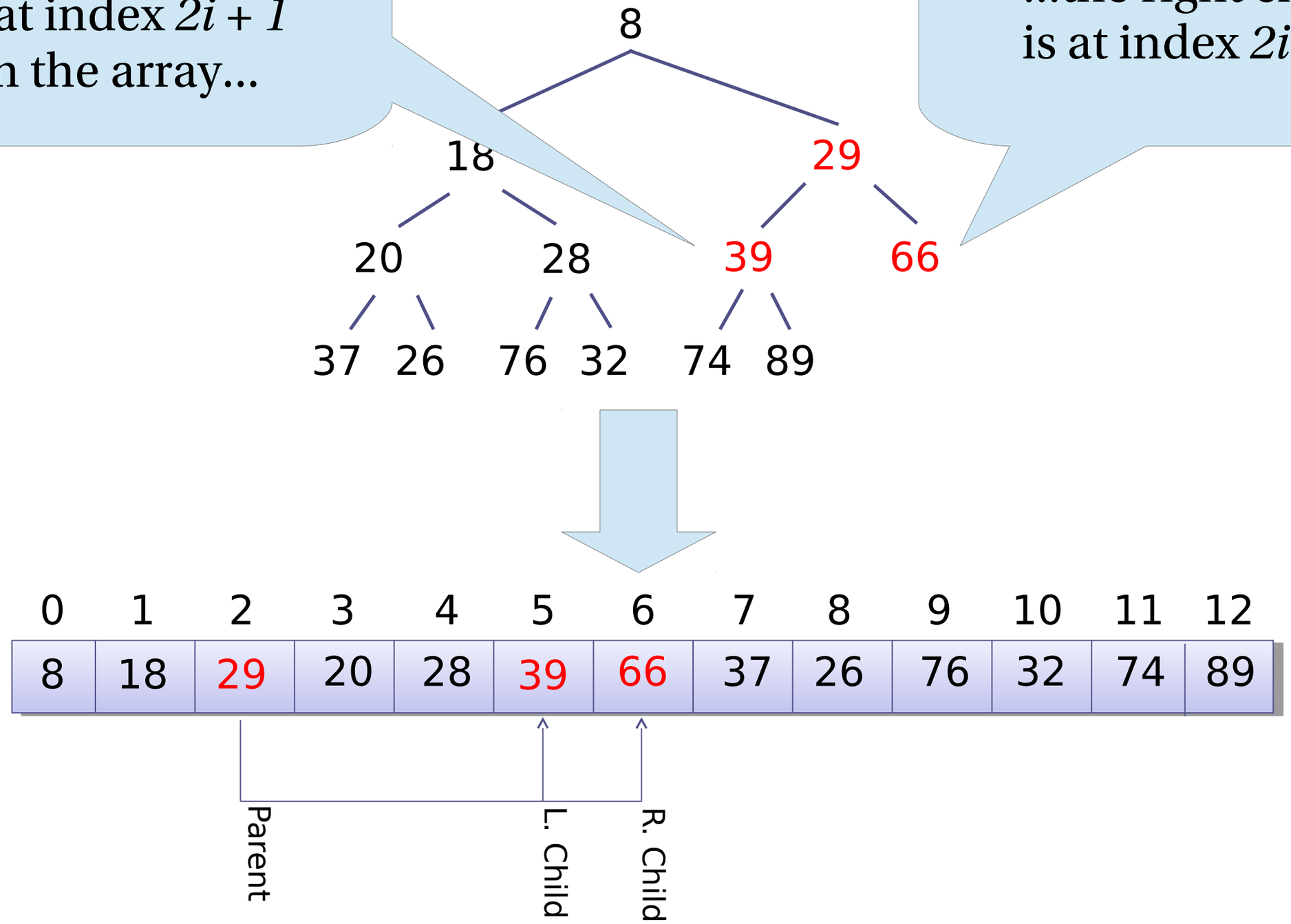
...the right child is at index $2i + 2$



Child positions

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

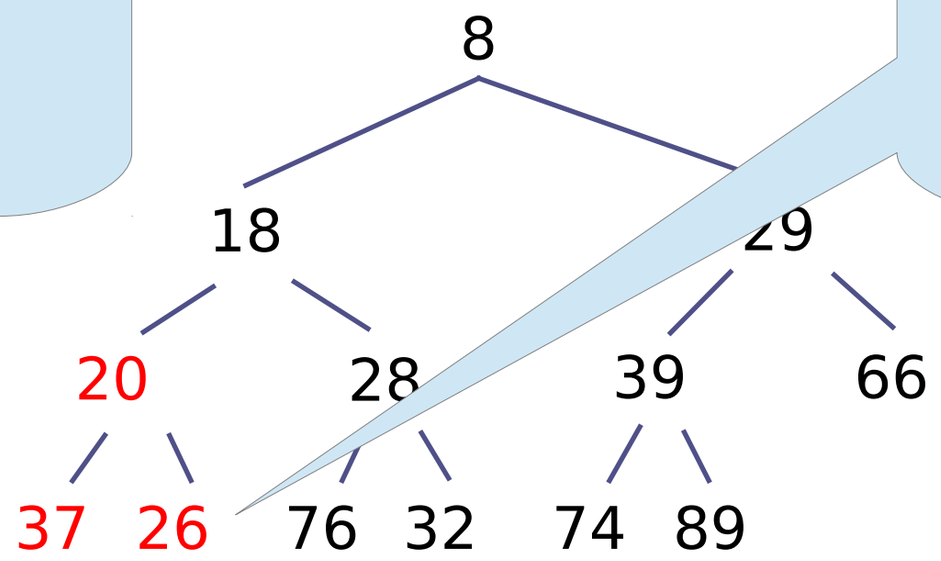
...the right child is at index $2i + 2$

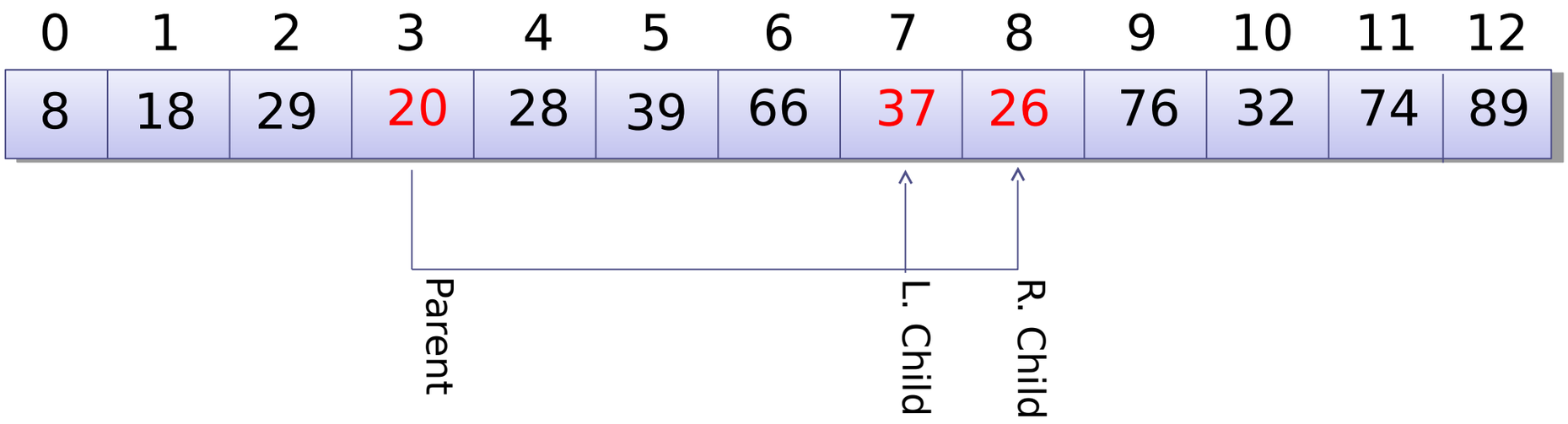


Child positions

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

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

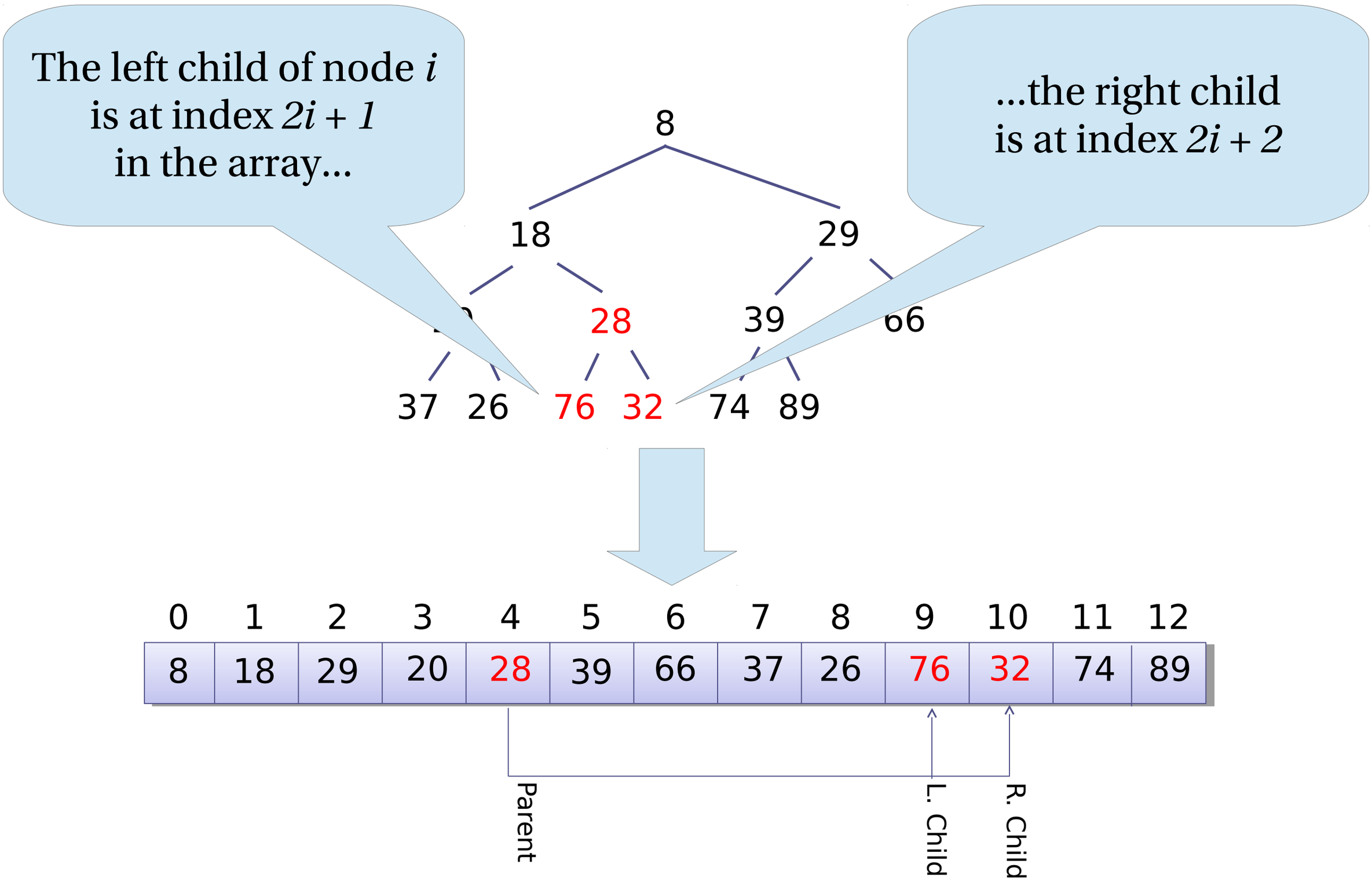




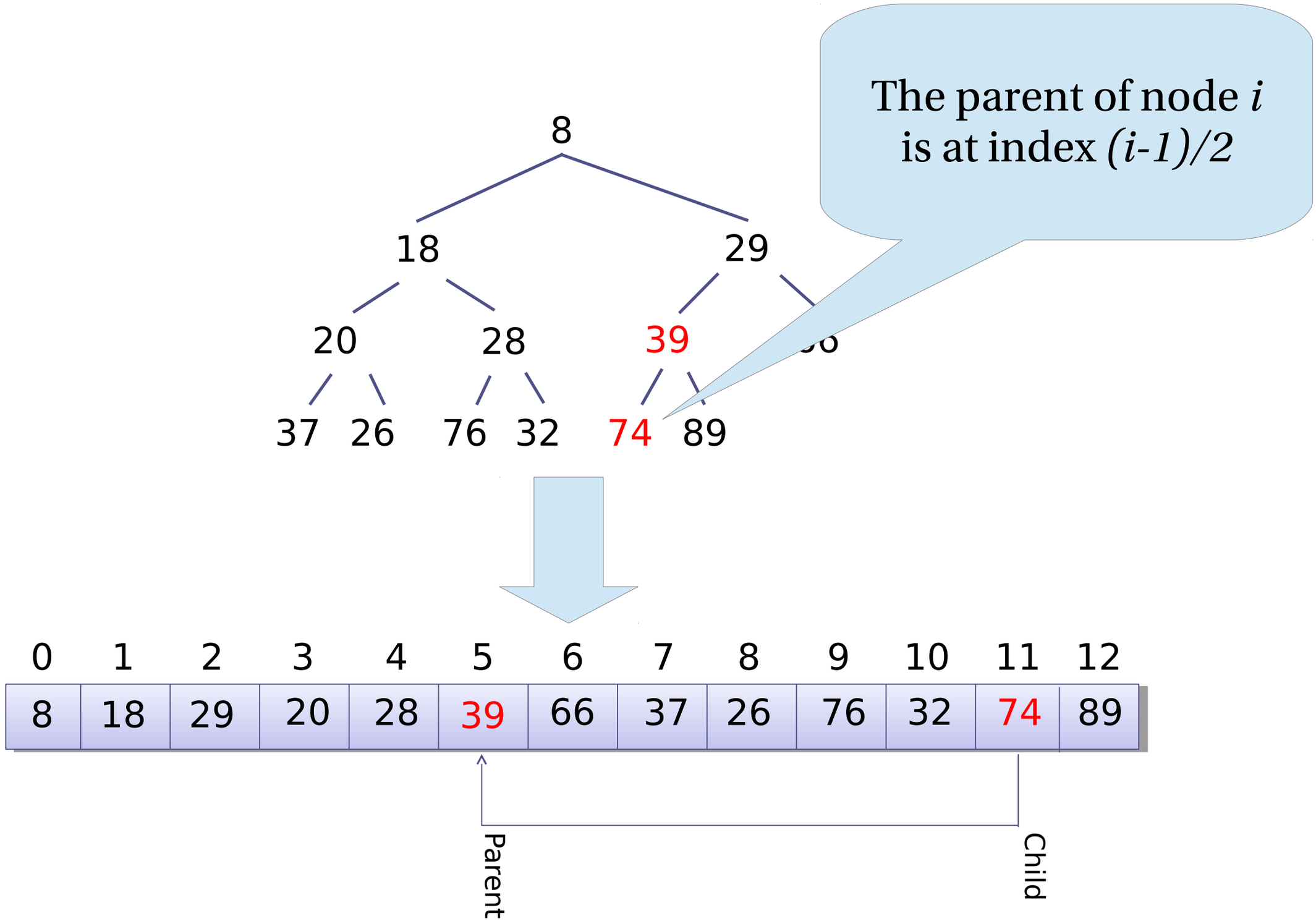
Child positions

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

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



Parent position



Why the binary heap invariant

The binary heap invariant: the tree is *complete* and satisfies the *heap property*

- Because of the heap property, we can find the minimum element in O(1) time
- Because the tree is complete, we can represent a heap of n items with an array of size n

Warning

We are using 0-based arrays, the obvious choice in Java

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!

Reminder: inserting into a binary heap

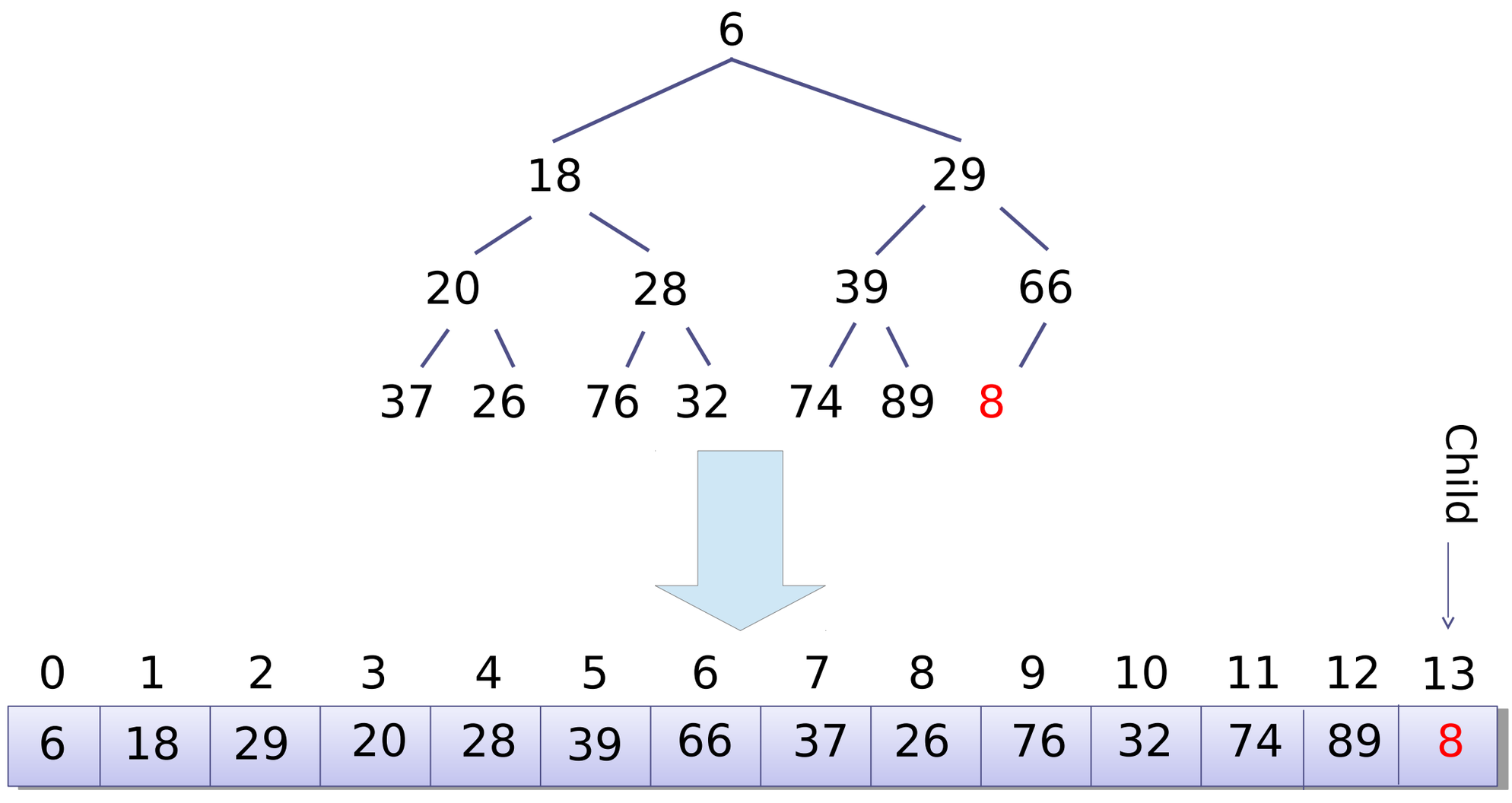
To insert an element into a binary heap:

- 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 just as well implement this using the array representation! Just need to use index calculations instead of following left/right/parent links.

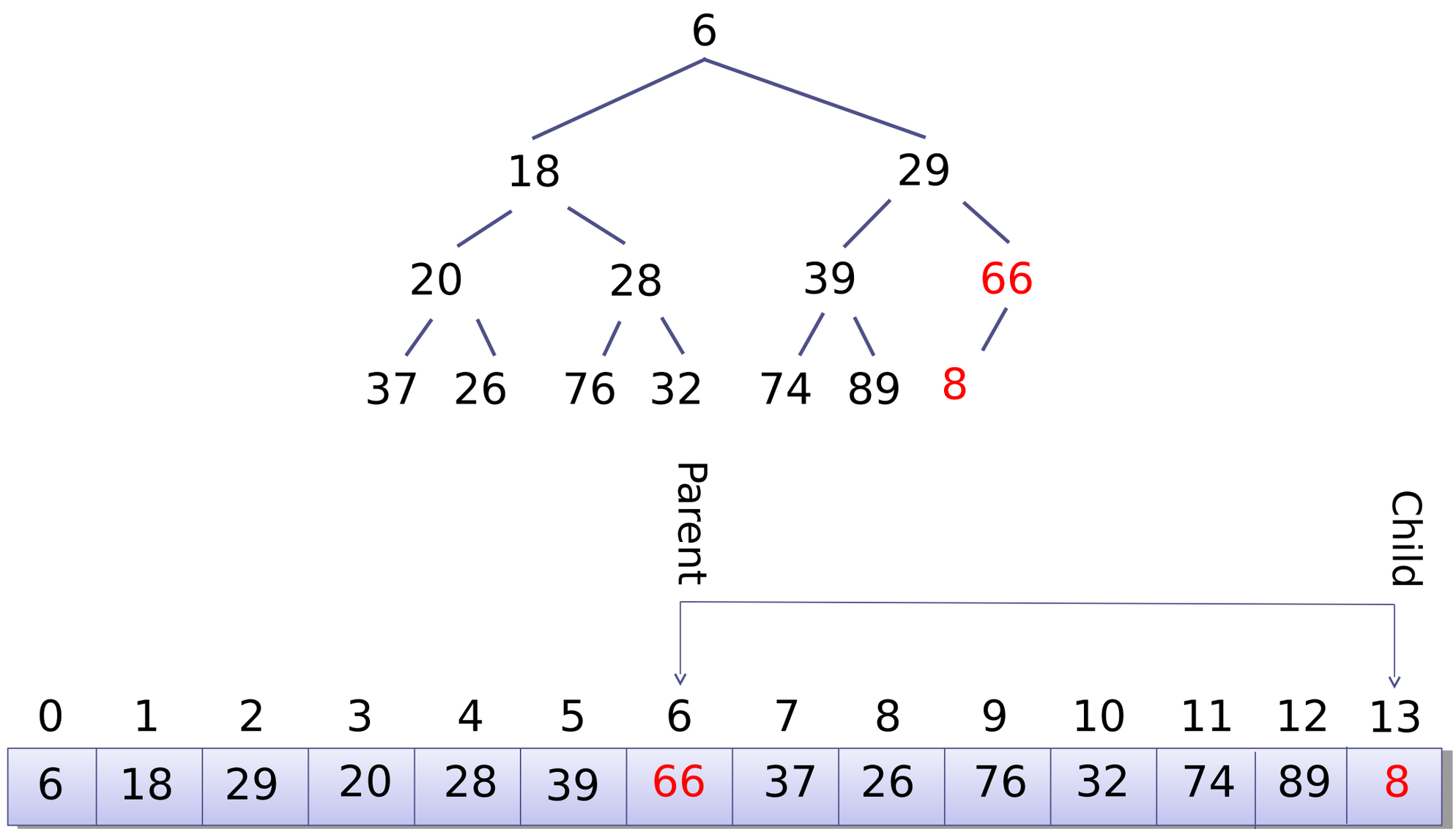
Inserting into a binary heap

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



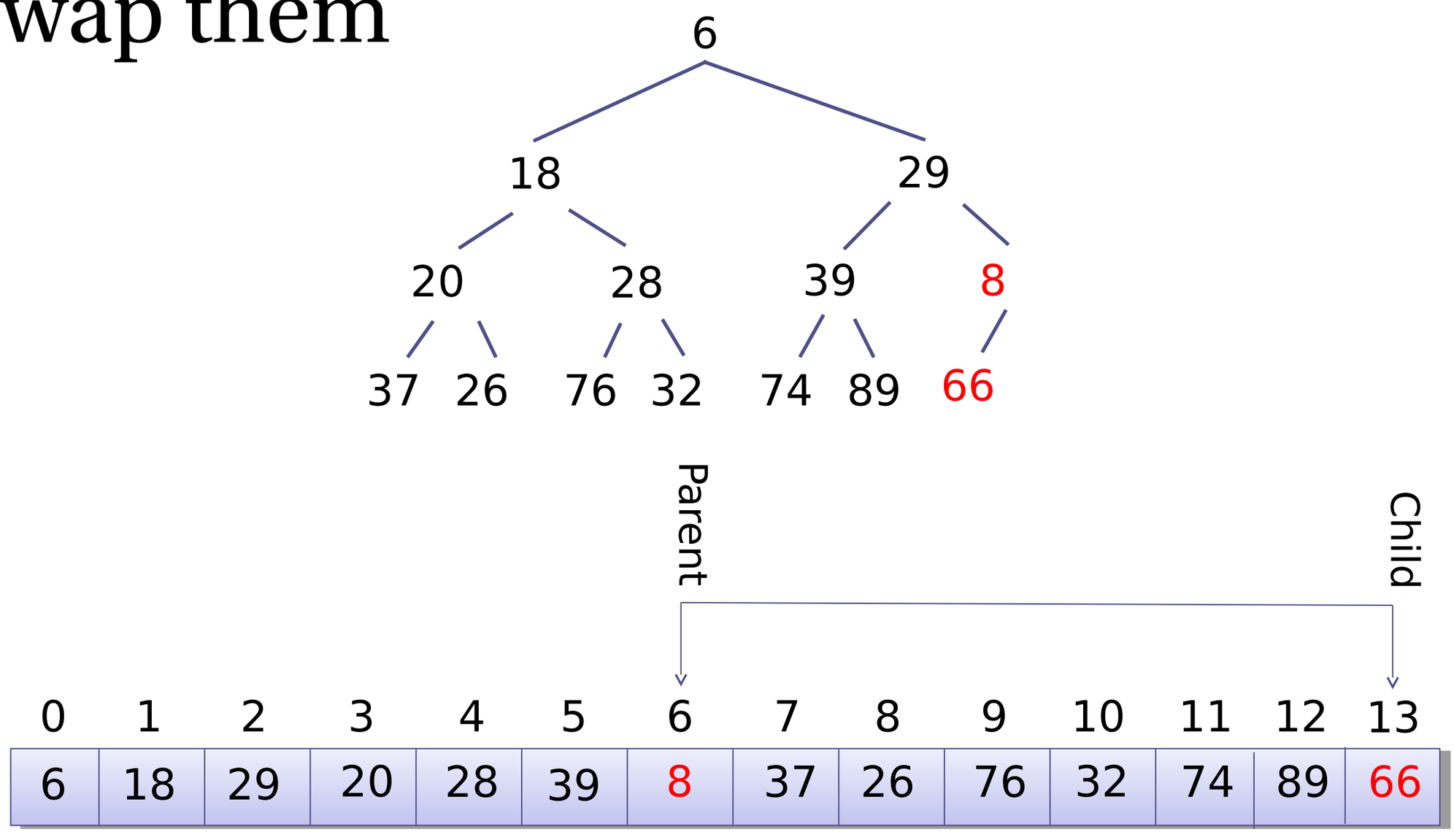
Inserting into a binary heap

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



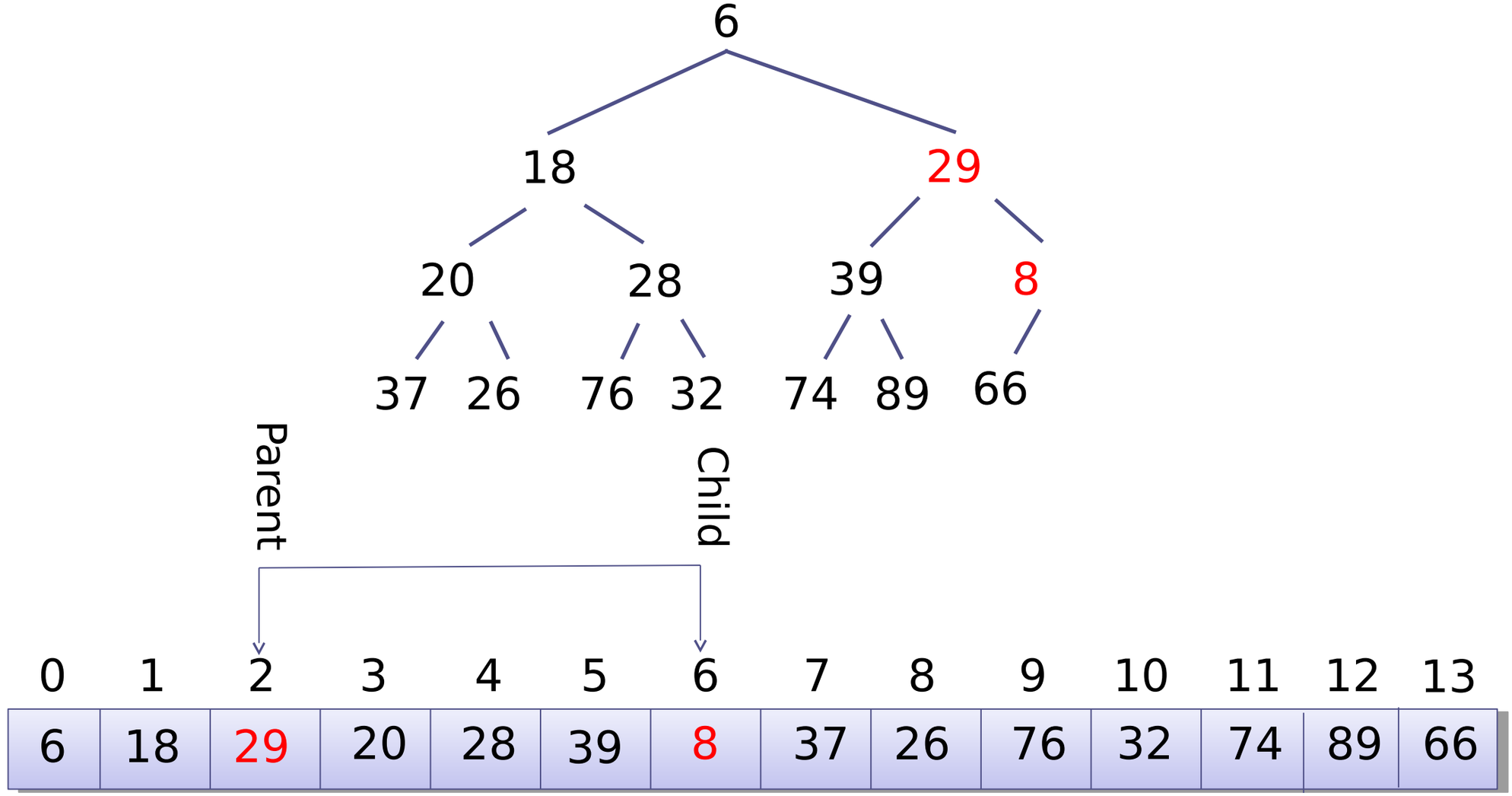
Inserting into a binary heap

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



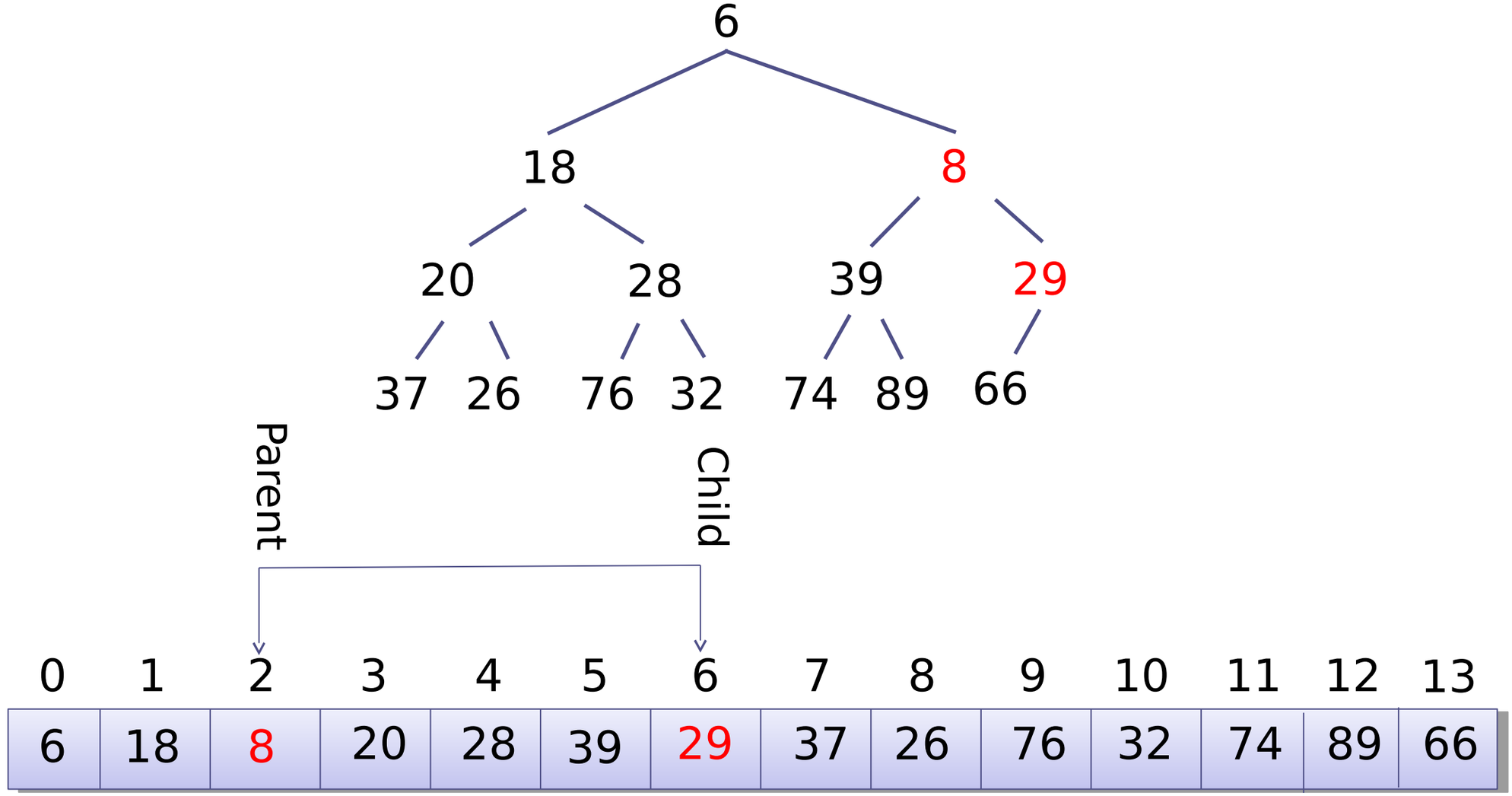
Inserting into a binary heap

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



The insertion algorithm

Insert `x` at the end of the array, let `child` be its index and `parent = (child-1)/2`

While `child > 0` and `array[parent] > array[child]`:

- Swap `array[parent]` and `array[child]`
- Set `child = parent` and `parent = (child-1)/2`

Does many swap operations! Moves the new element many times in the heap.

Just the insertion algorithm from before, but using an array to represent the tree!

An o… …n the book

array[child] was always x before

Insert ~~x~~ **n**… …he array, let child be its i… …x and parent = (child-1)/2

While child > 0 and ~~array[parent] > array[child]~~ **array[parent] > x**:

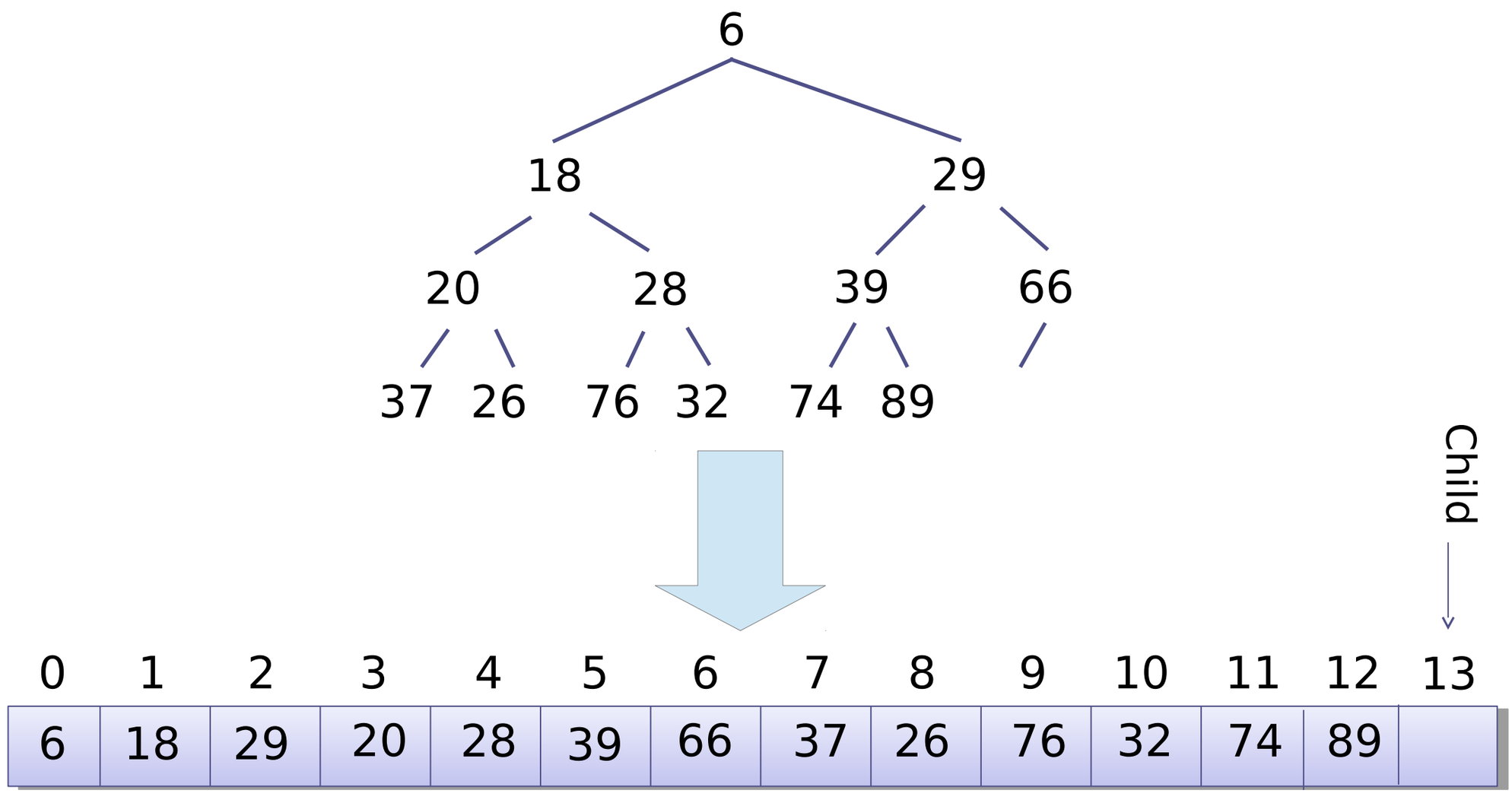
- ~~Swap array[parent] and array[child]~~
  Set array[child] = array[parent]
- Set child = parent and parent = (child-1)/2

Finally, set array[parent] = x

Makes a space for x and then puts it there

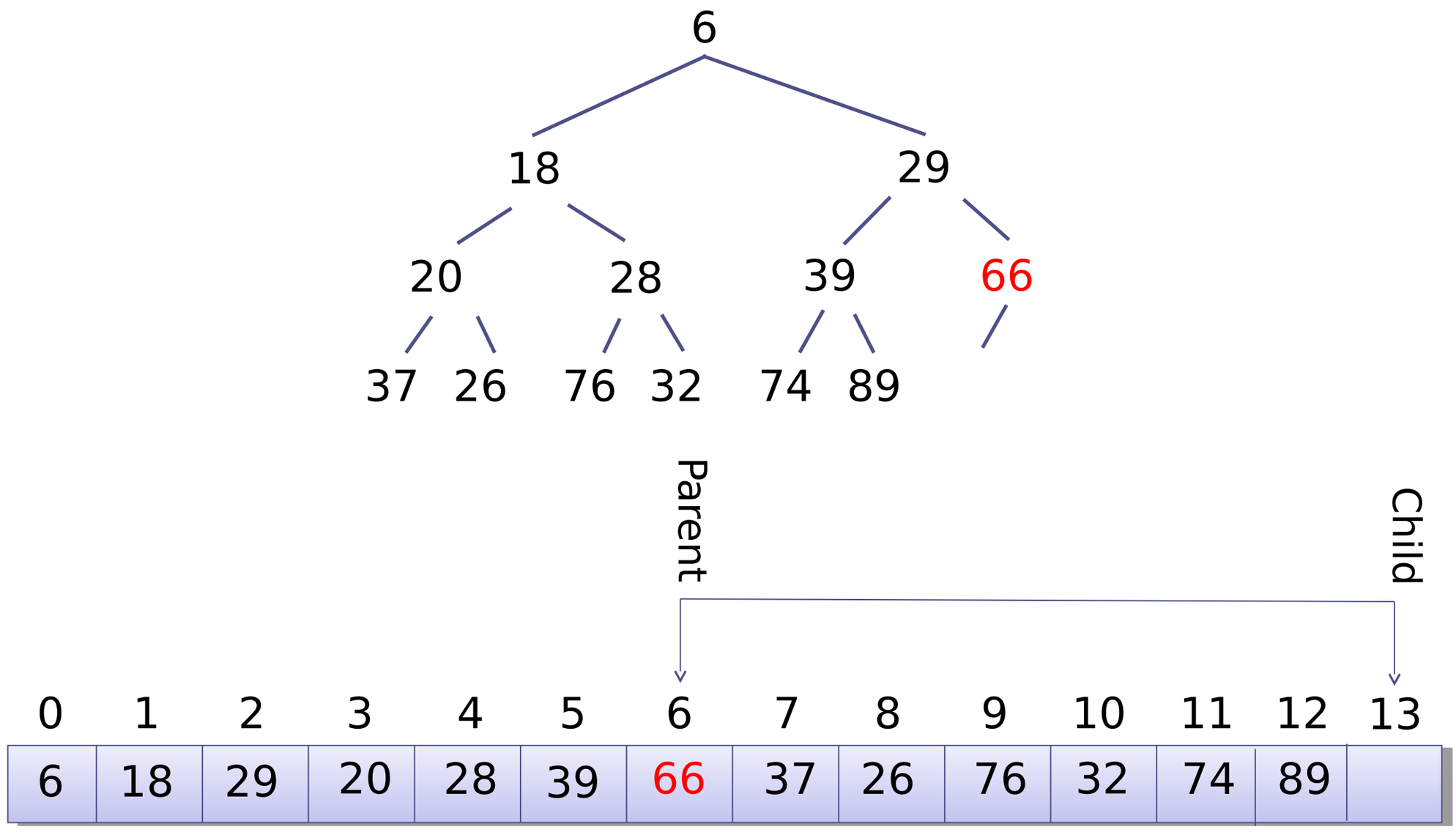
Optimised insertion

Step 1: increase the size of the array, set `child` to the final index



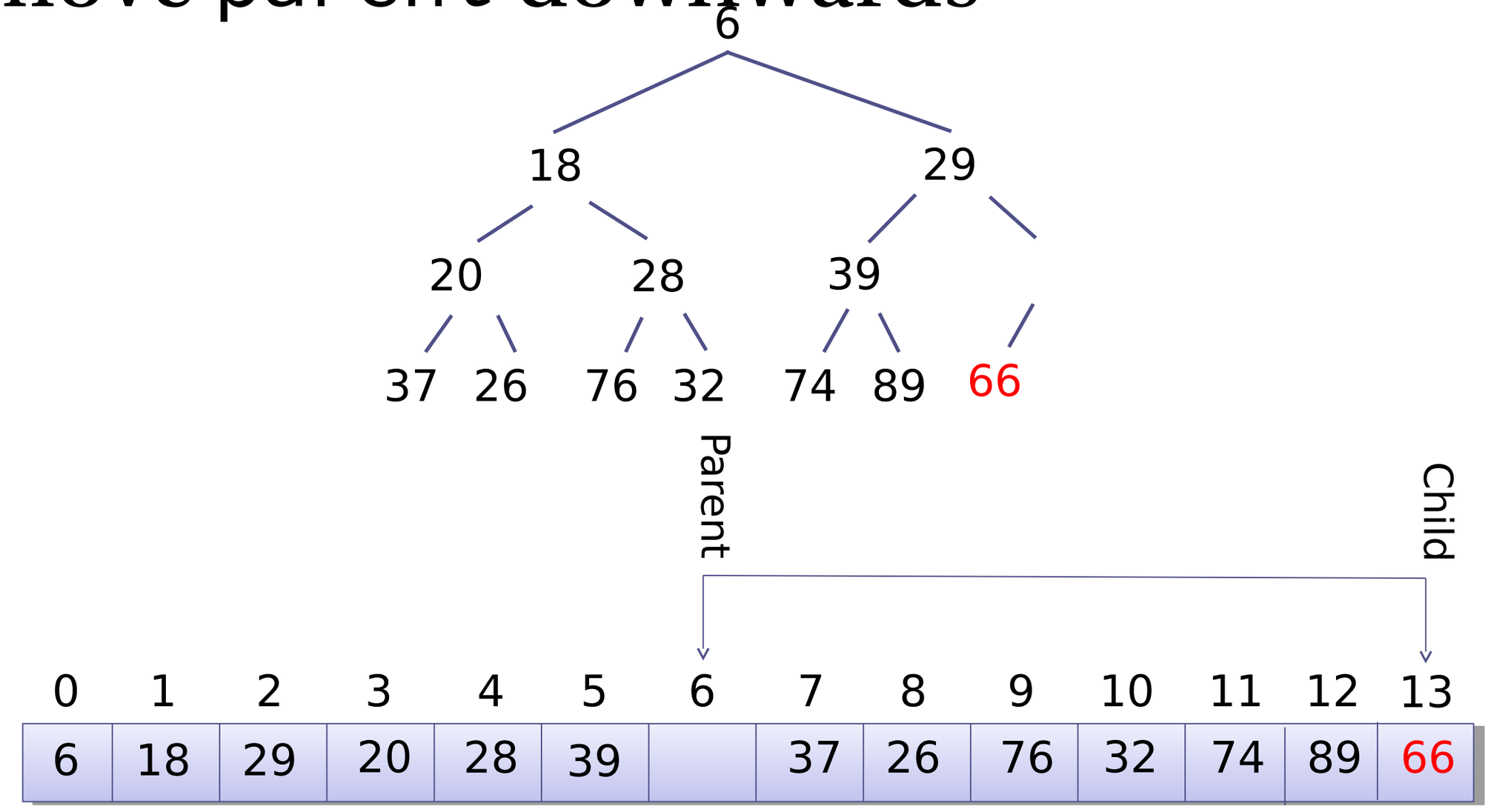
Optimised insertion

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



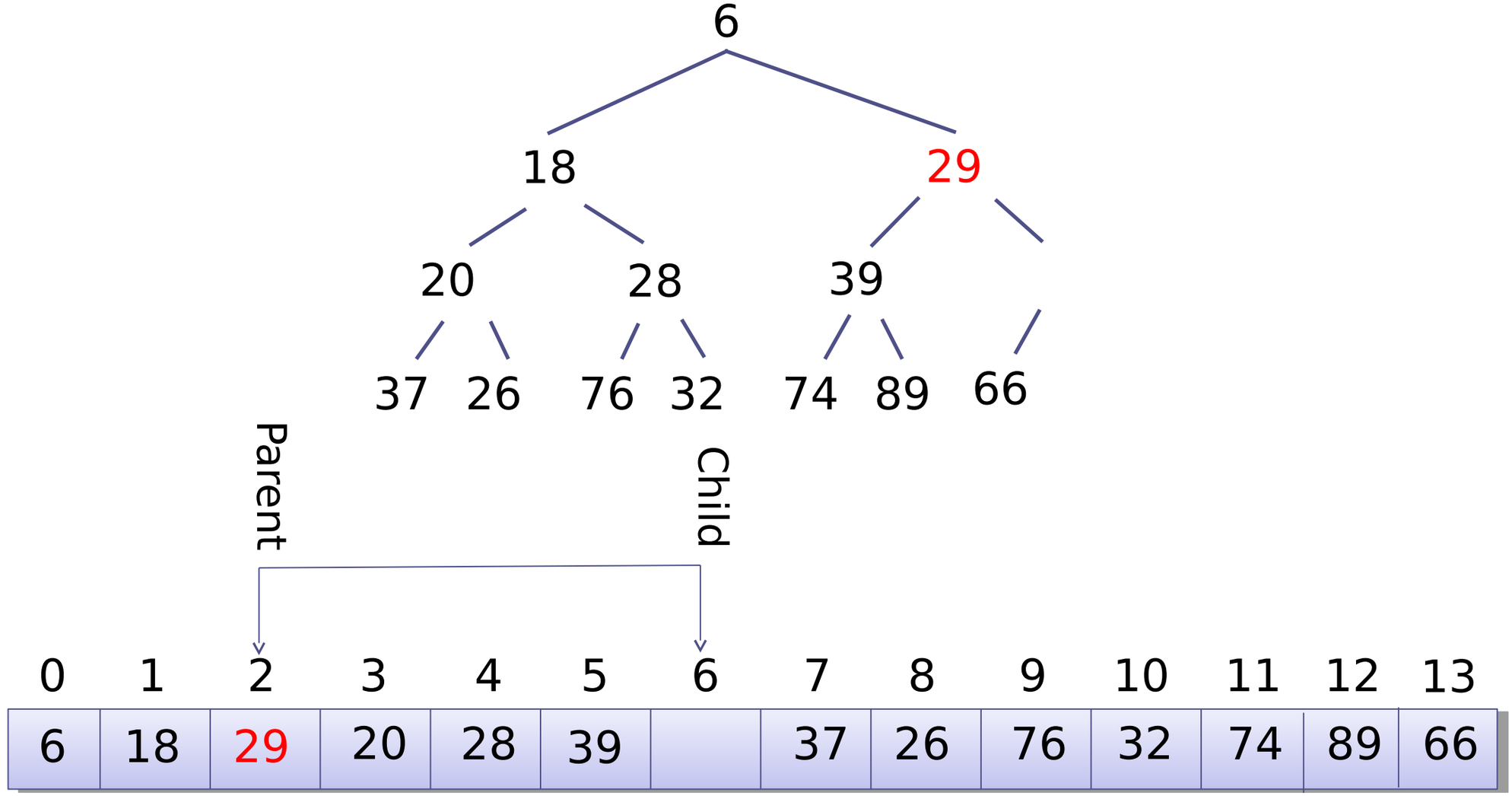
Optimised insertion

Step 3: if `array[parent]` > `x` (8 here), move `parent` downwards



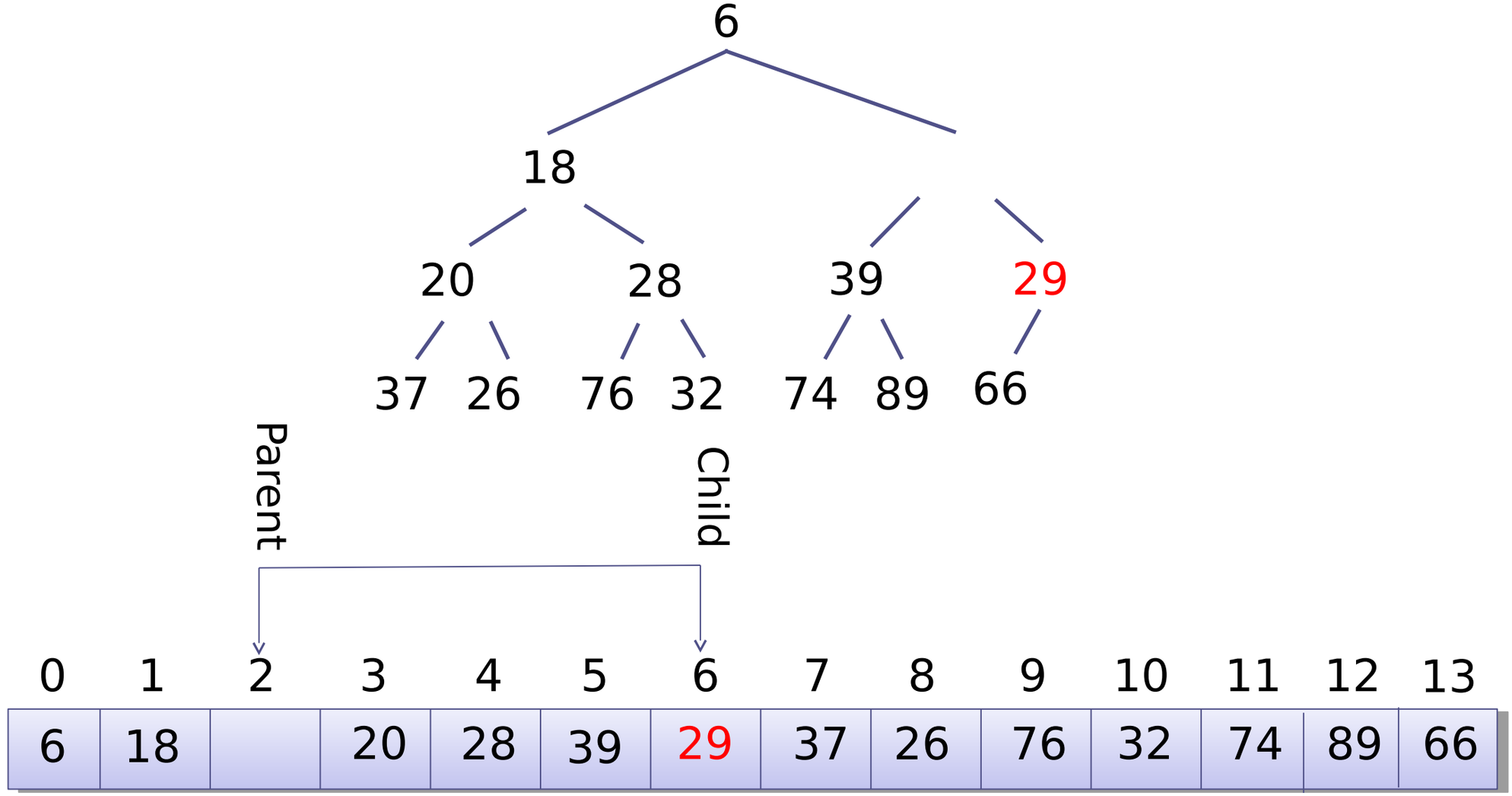
Optimised insertion

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



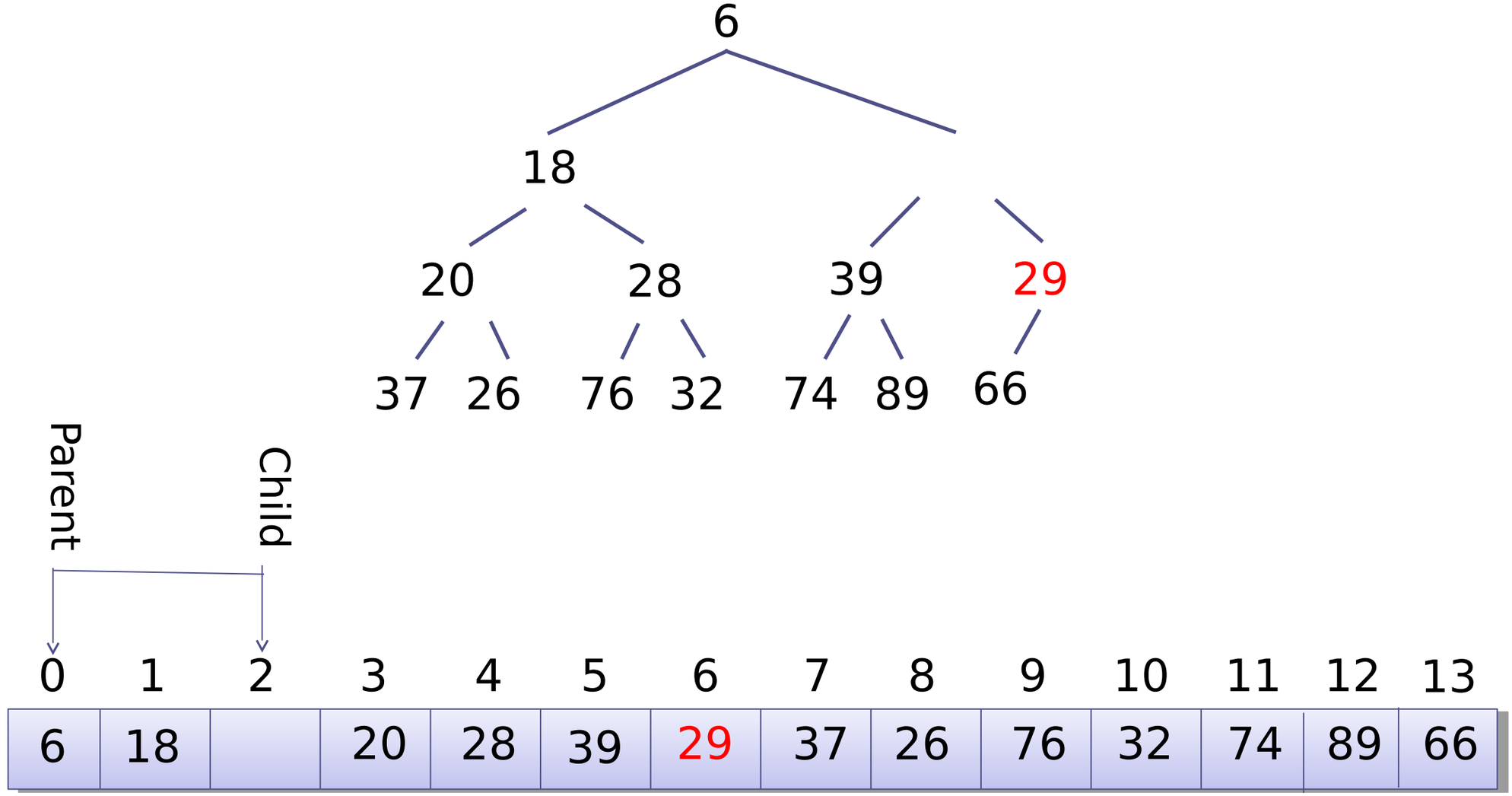
Optimised insertion

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



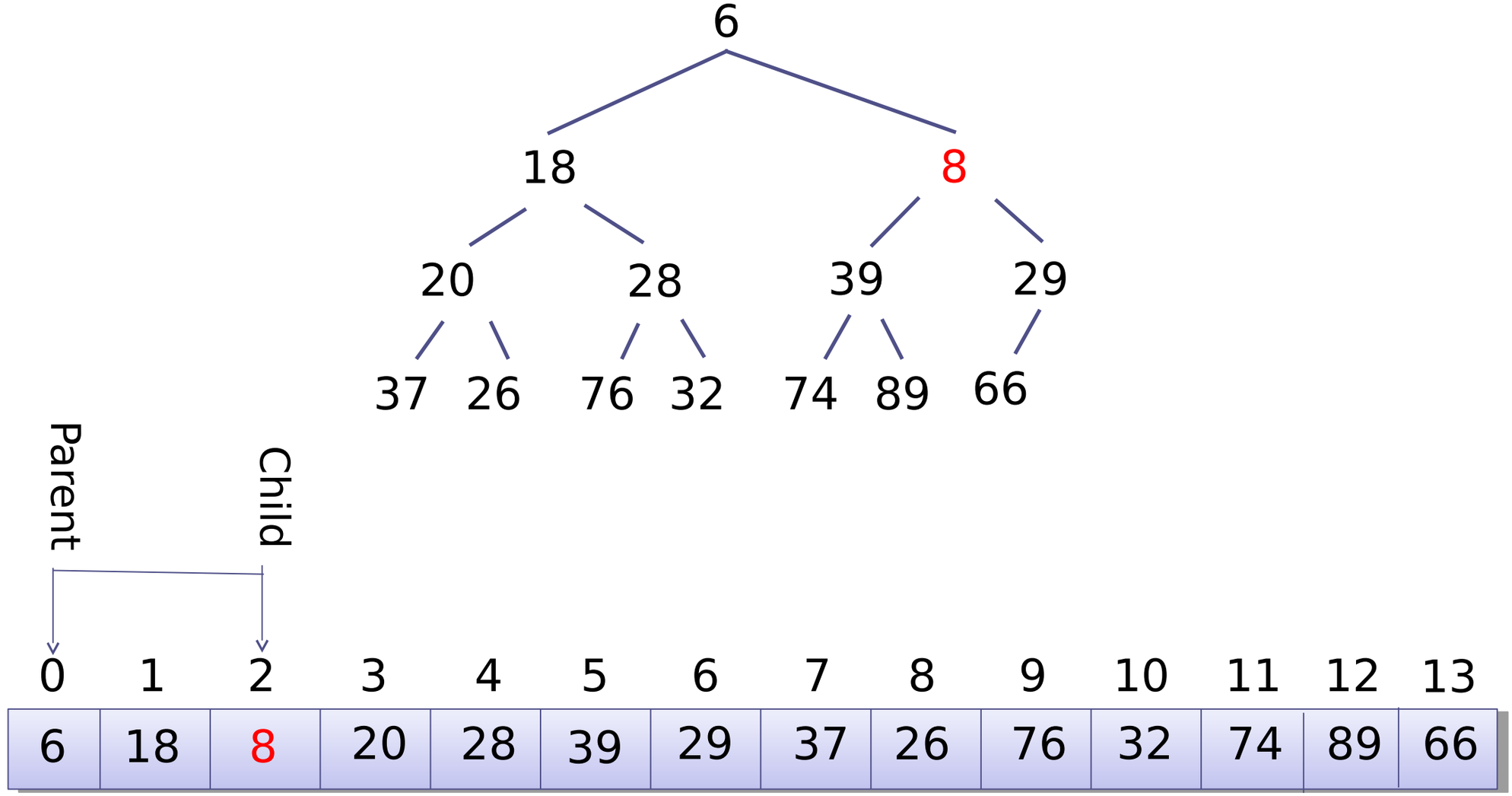
Optimised insertion

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



Optimised insertion

Step 5: write `x` into position `child`



Complexity

The depth of the tree is O(log n)

So O(log n) swaps for insert and deleteMin

So O(1) findMin, O(log n) insert, O(log n) deleteMin

An extra operation: decreaseKey

What if you want to *decrease* the value of an element? (This pops up in some algorithms, and lab 2)

Step 1: alter the element. Might break the heap invariant, because the element might be smaller than its parent. So...

Step 2: sift the element up

However, you need to know the index of the element to sift it up! Solution: maintain a *multimap* from elements to their indices, and update it whenever you modify the heap

Heapsort

It's quite easy to sort a list using a heap:

- add all the list elements to an empty heap
- 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

It's quite easy to sort a list using a heap:

- add all the list elements to an empty heap
- 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 will build a *max heap*, a heap where you can find and delete the *maximum* element instead of the minimum

- Simple change to heap operations

First turn array into a max heap, in-place

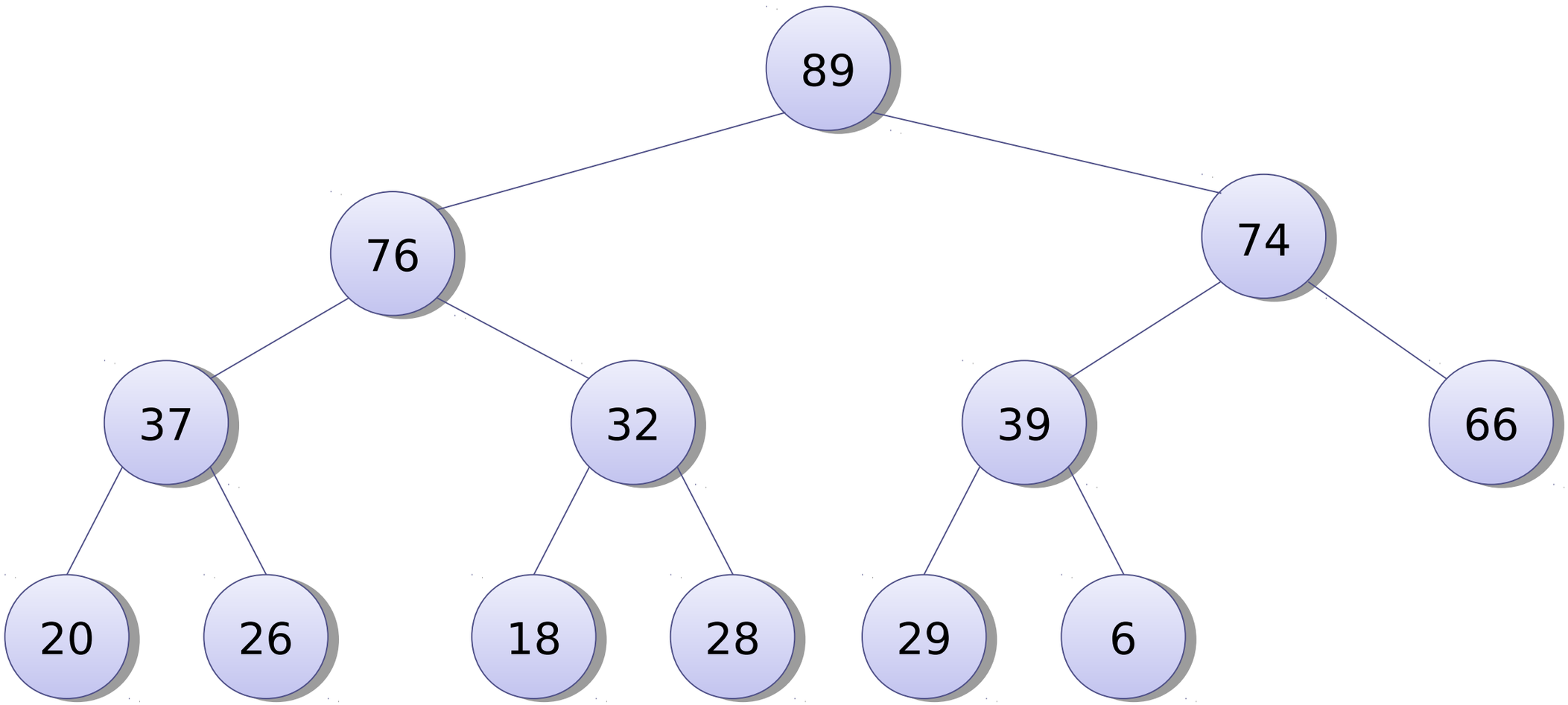
Now, we have a heap; how to turn it into a sorted array?

- Swap the maximum (first) element with the last element
- Reduce the heap's size by 1, so the heap no longer includes the maximum element
- Sift the first element down

This has the effect of deleting the biggest element and putting it at the end of the array – at every stage, the *beginning* of the array is a heap and the *end* contains sorted data

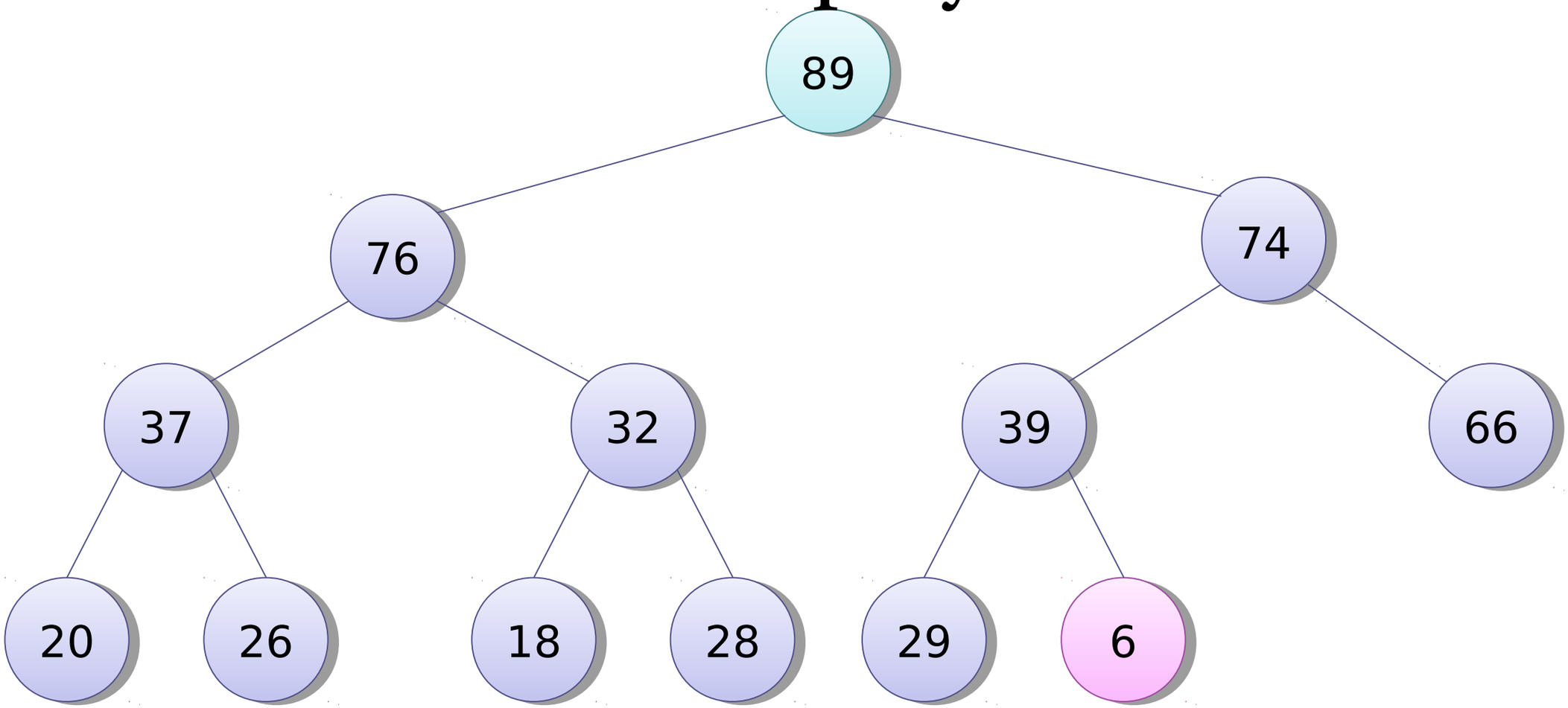
Trace of heapsort

First build a heap (not shown)



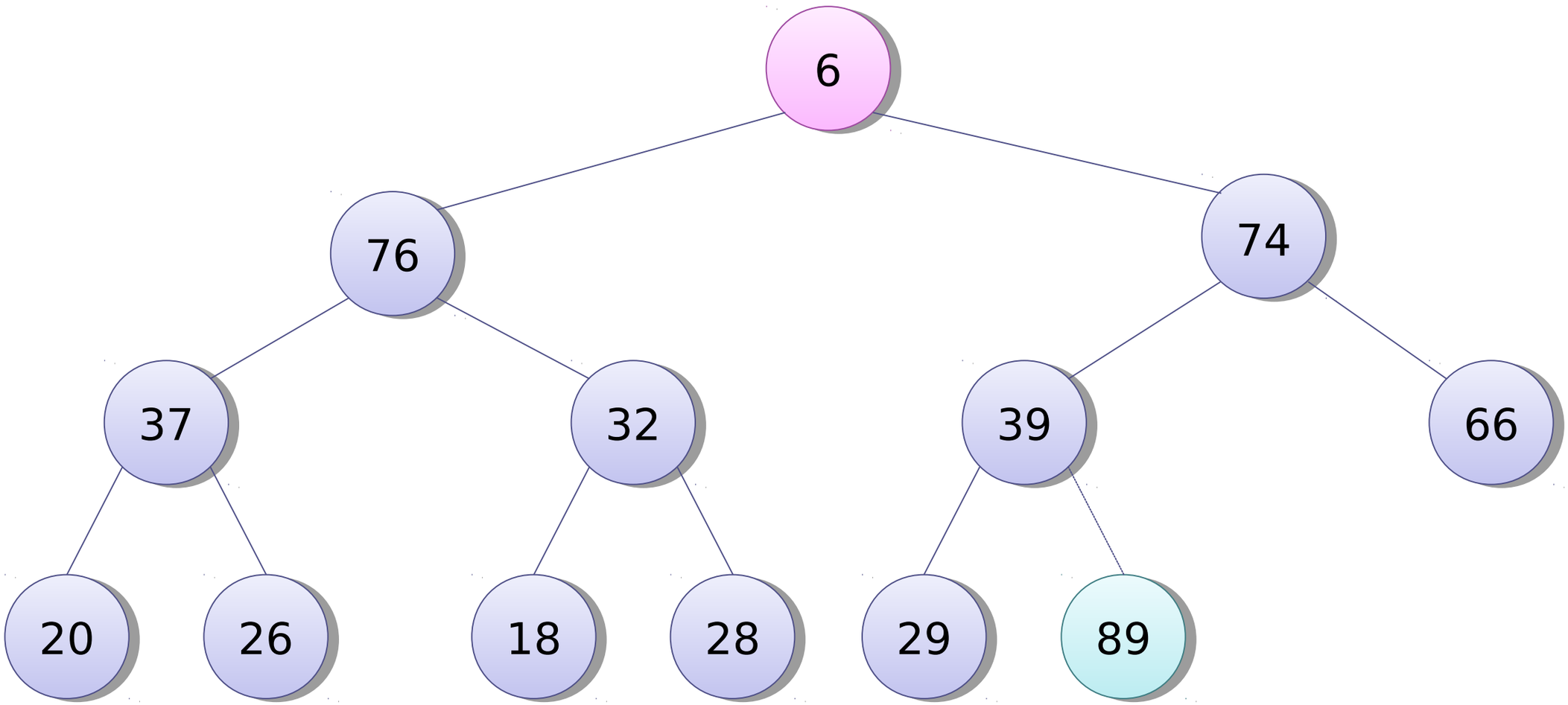
Trace of heapsort

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



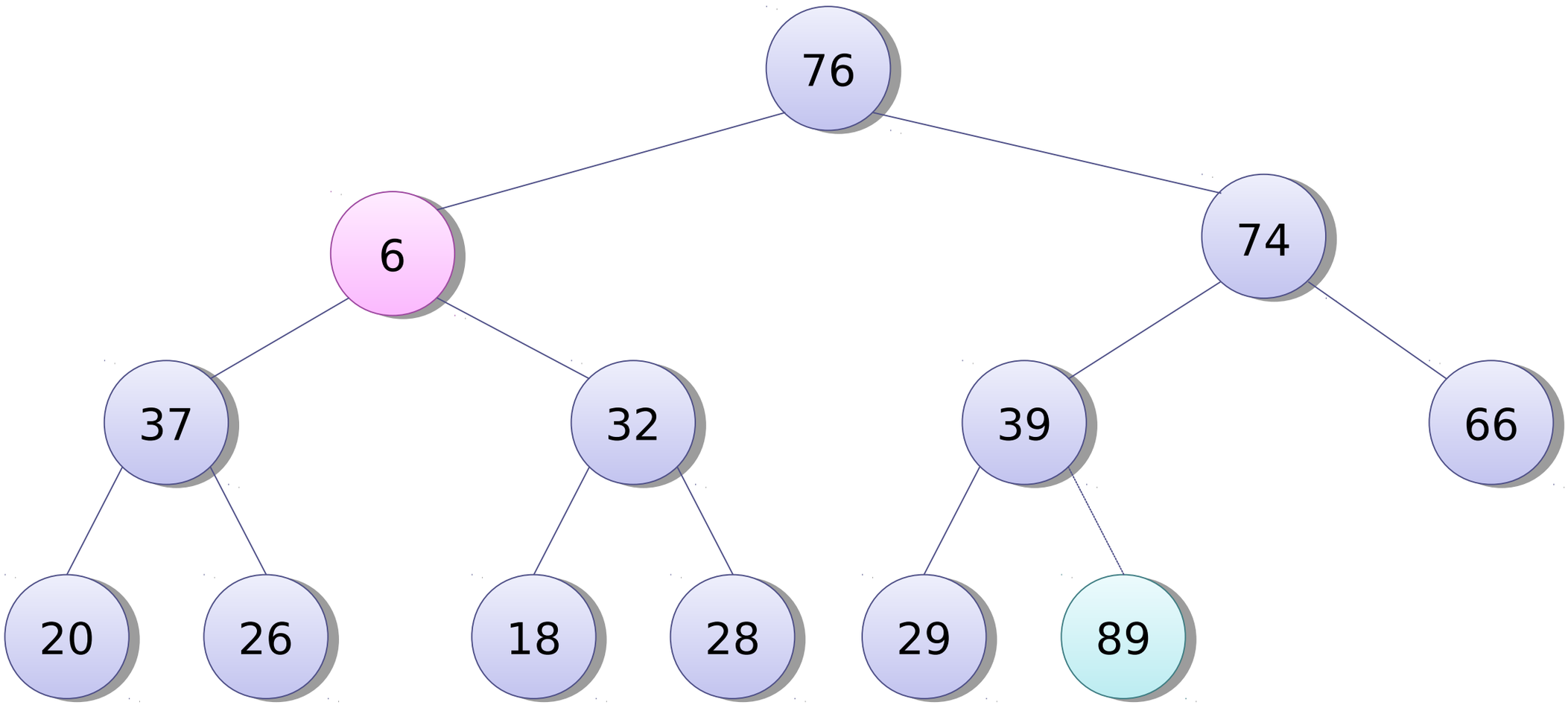
Trace of heapsort

Step 2: sift first element down



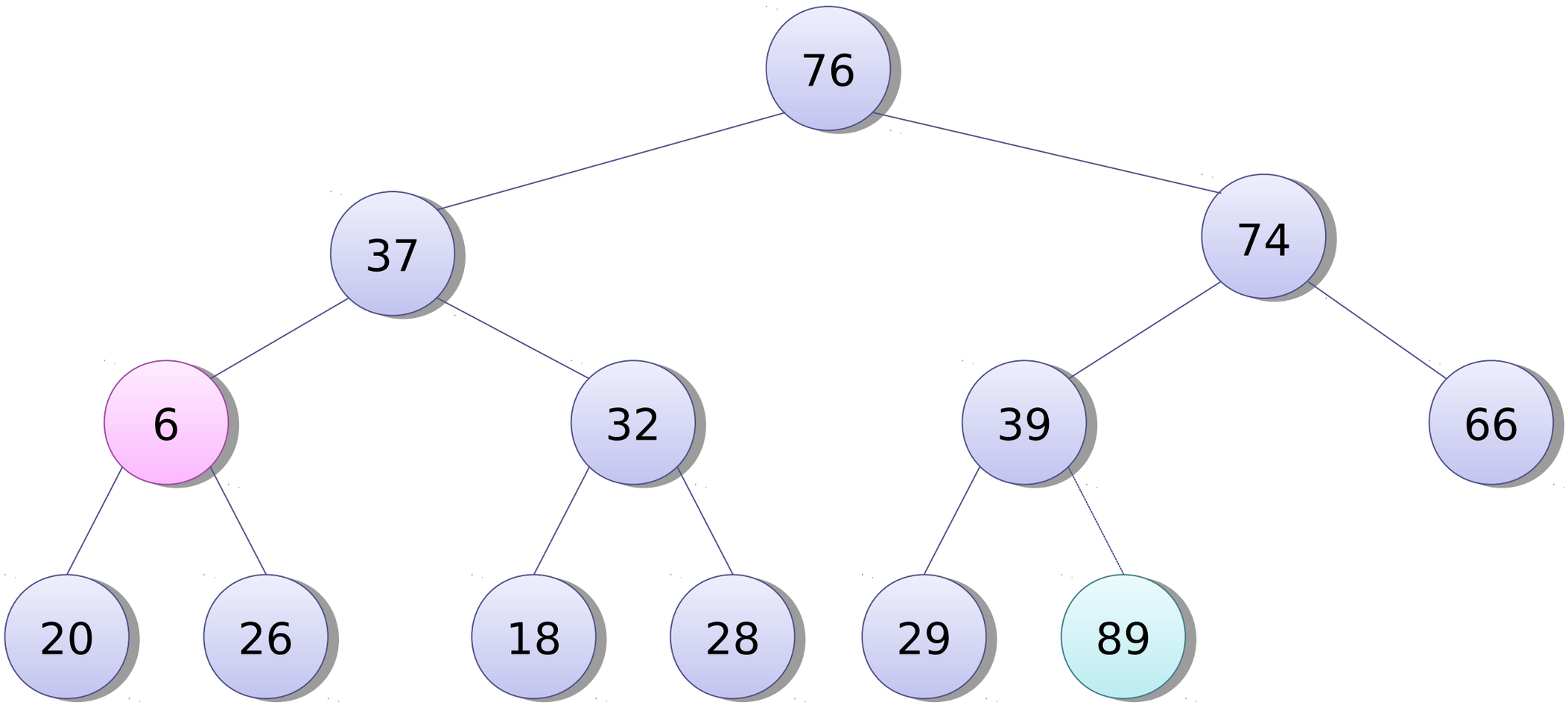
Trace of heapsort

Step 2: sift first element down



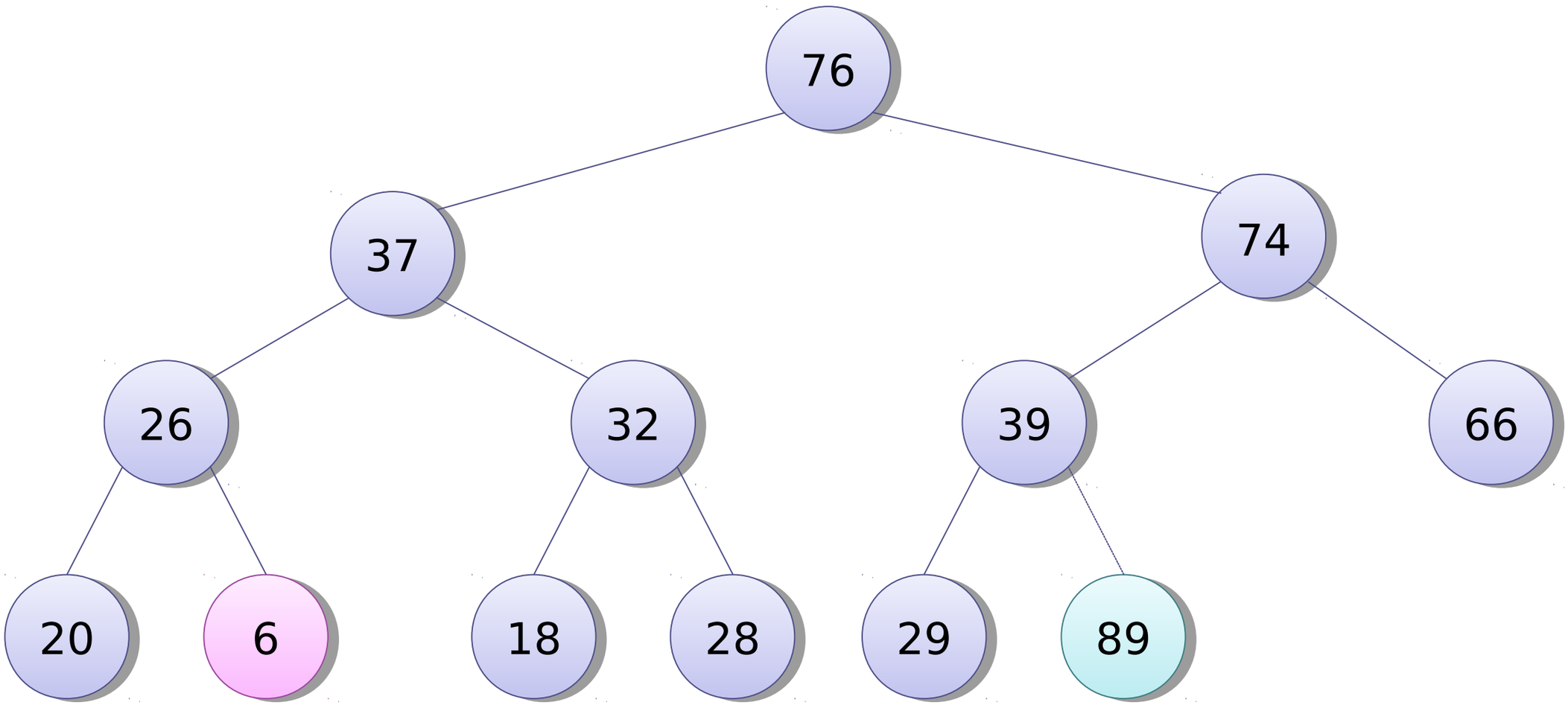
Trace of heapsort

Step 2: sift first element down

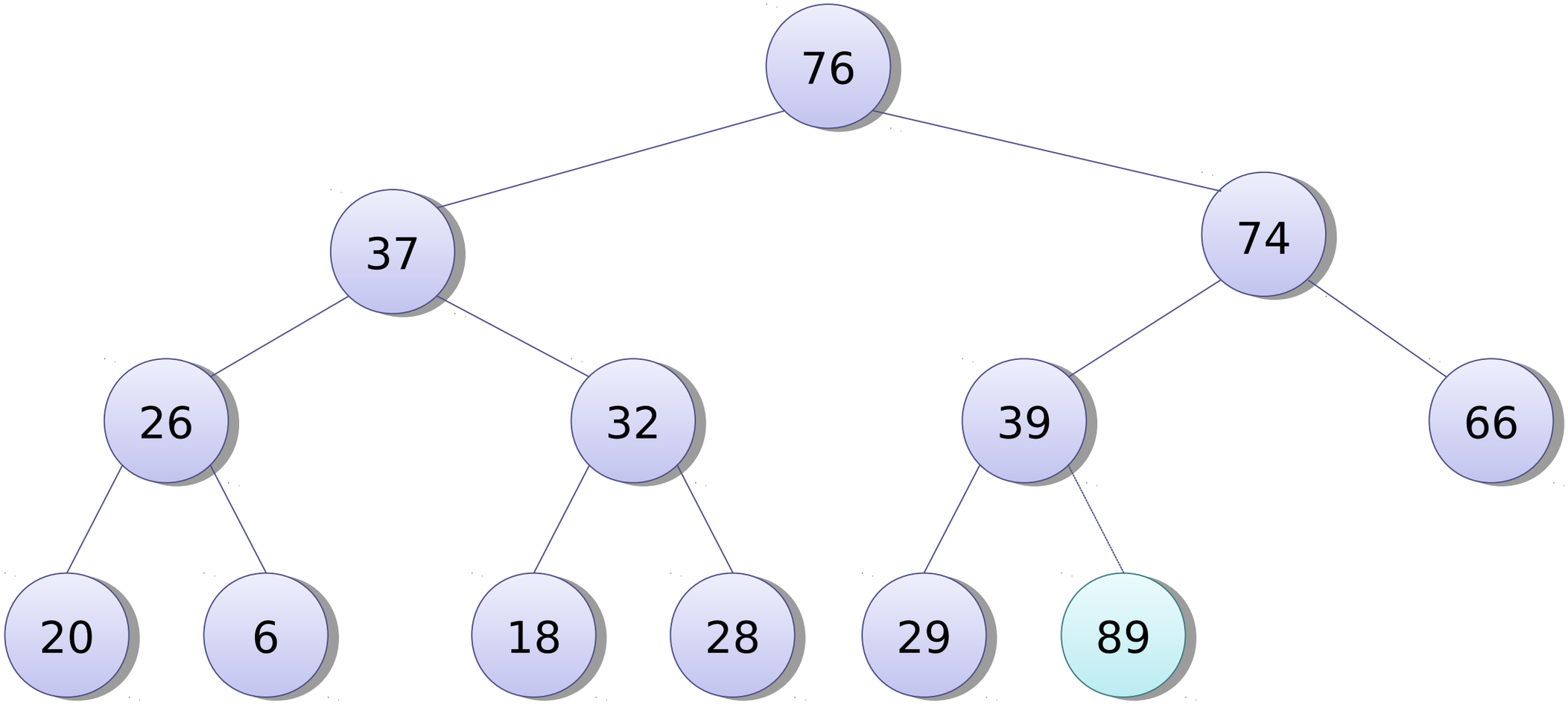


Trace of heapsort

Step 2: sift first element down

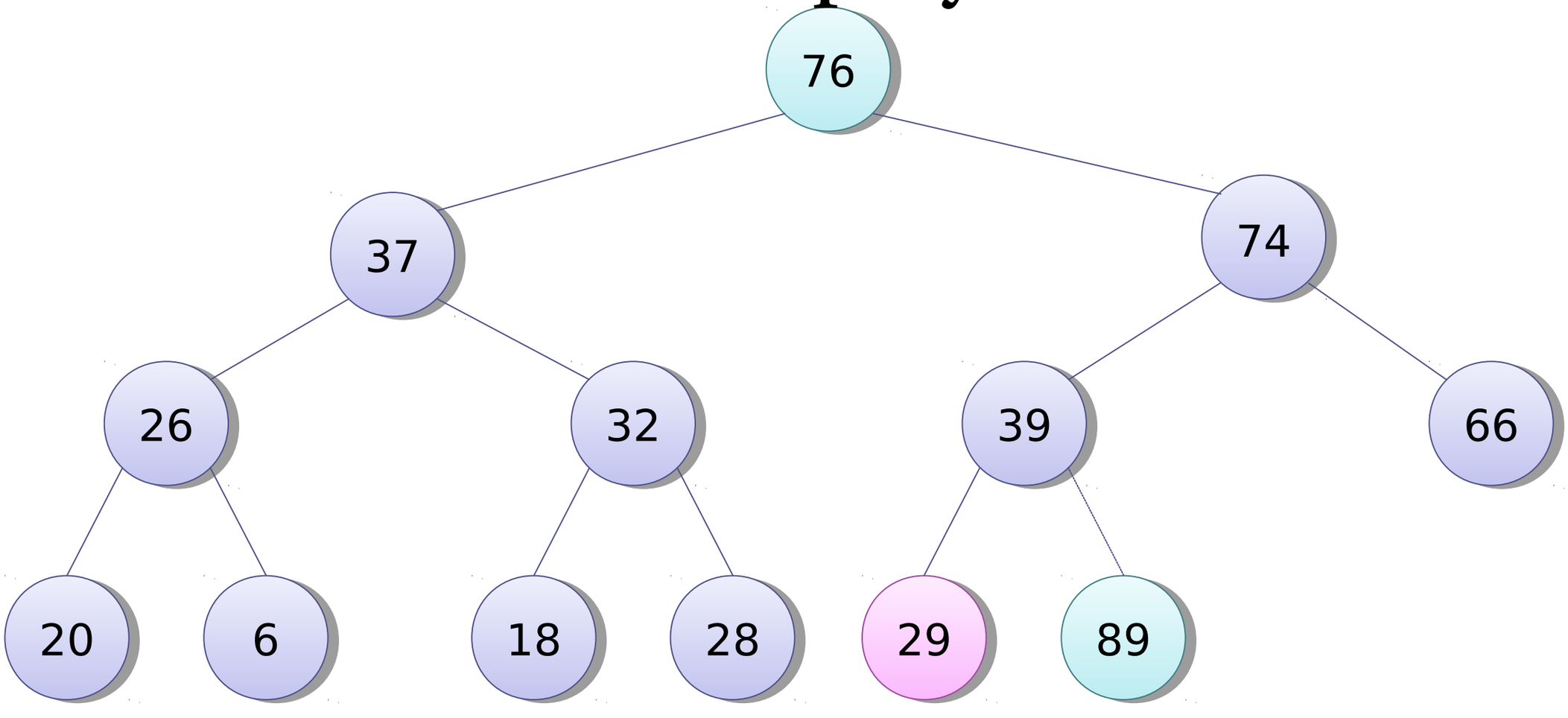


Trace of heapsort



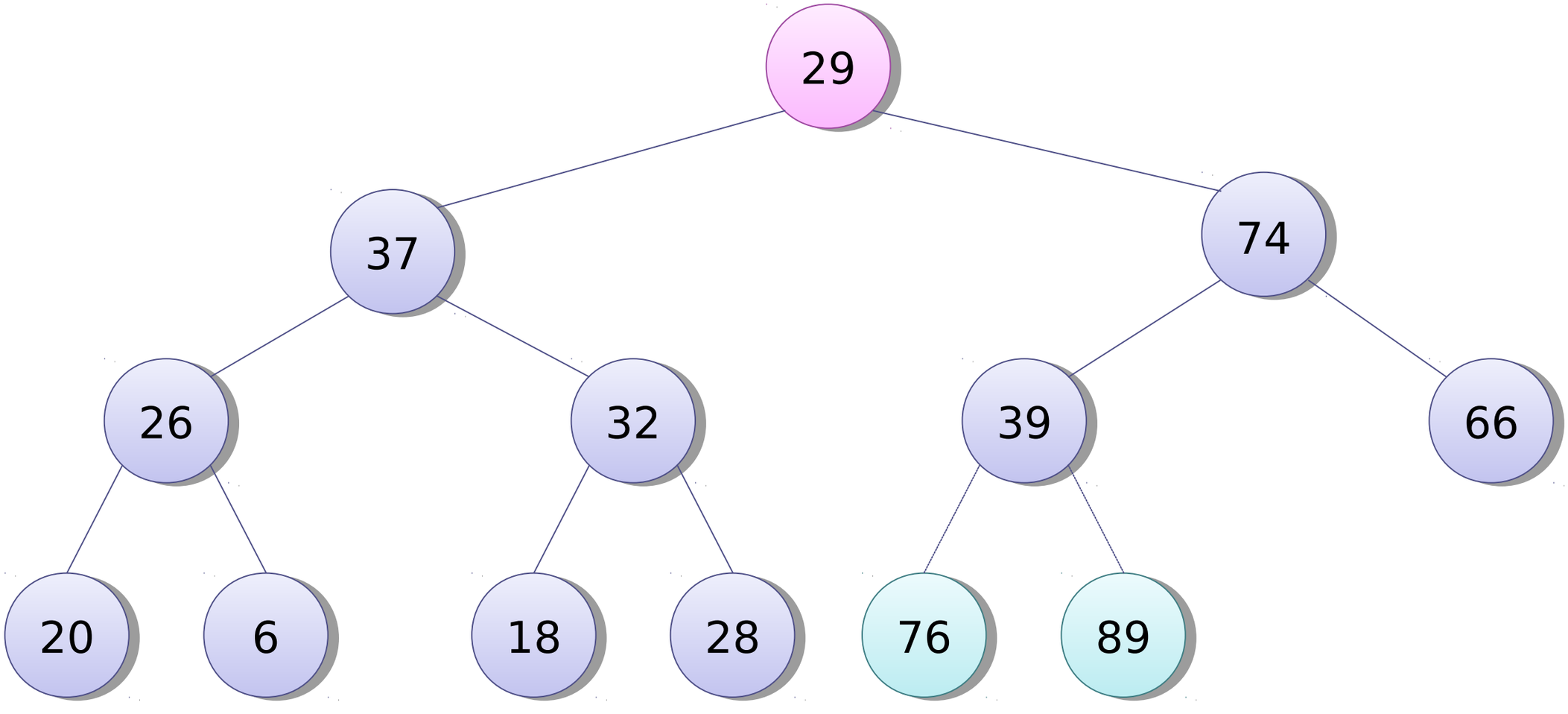
Trace of heapsort

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



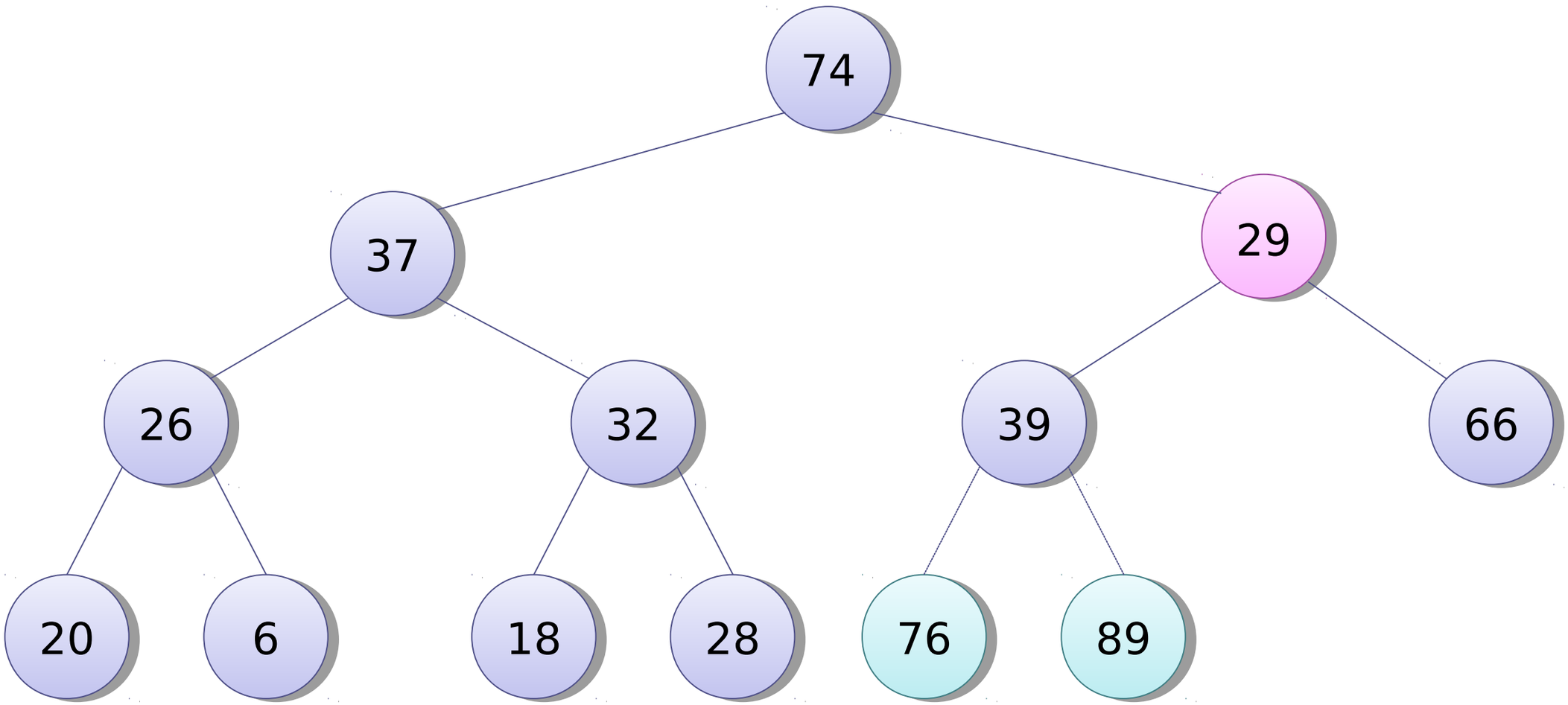
Trace of heapsort

Step 2: sift first element down



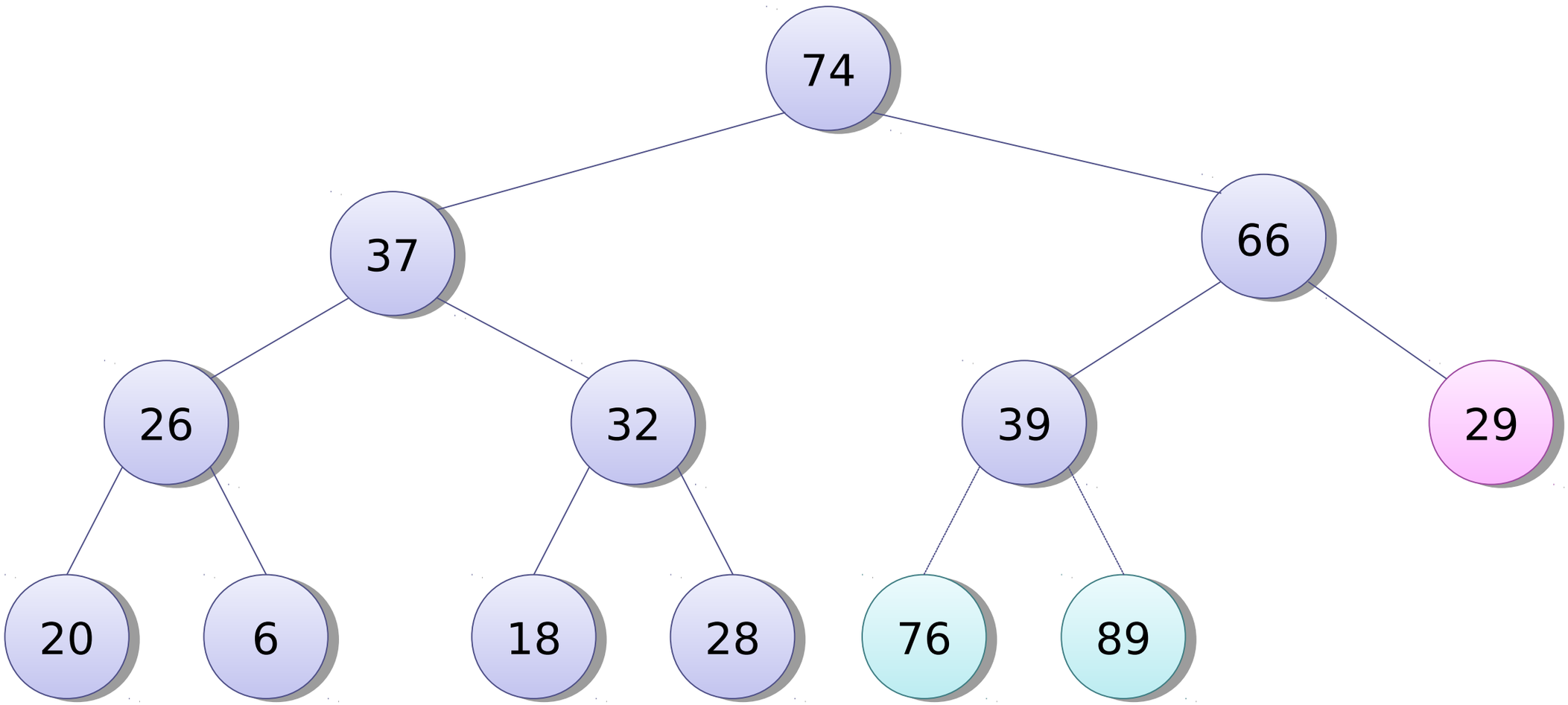
Trace of heapsort

Step 2: sift first element down

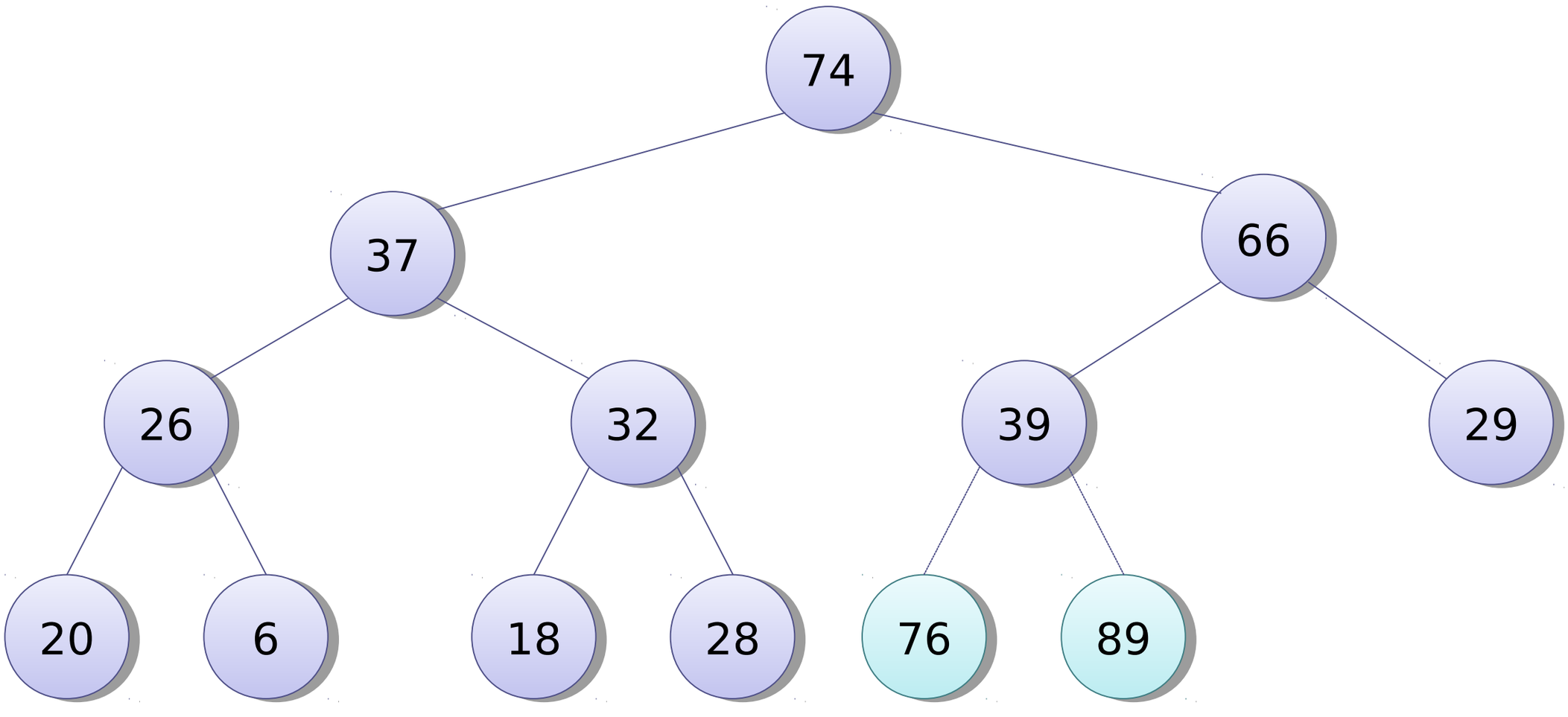


Trace of heapsort

Step 2: sift first element down

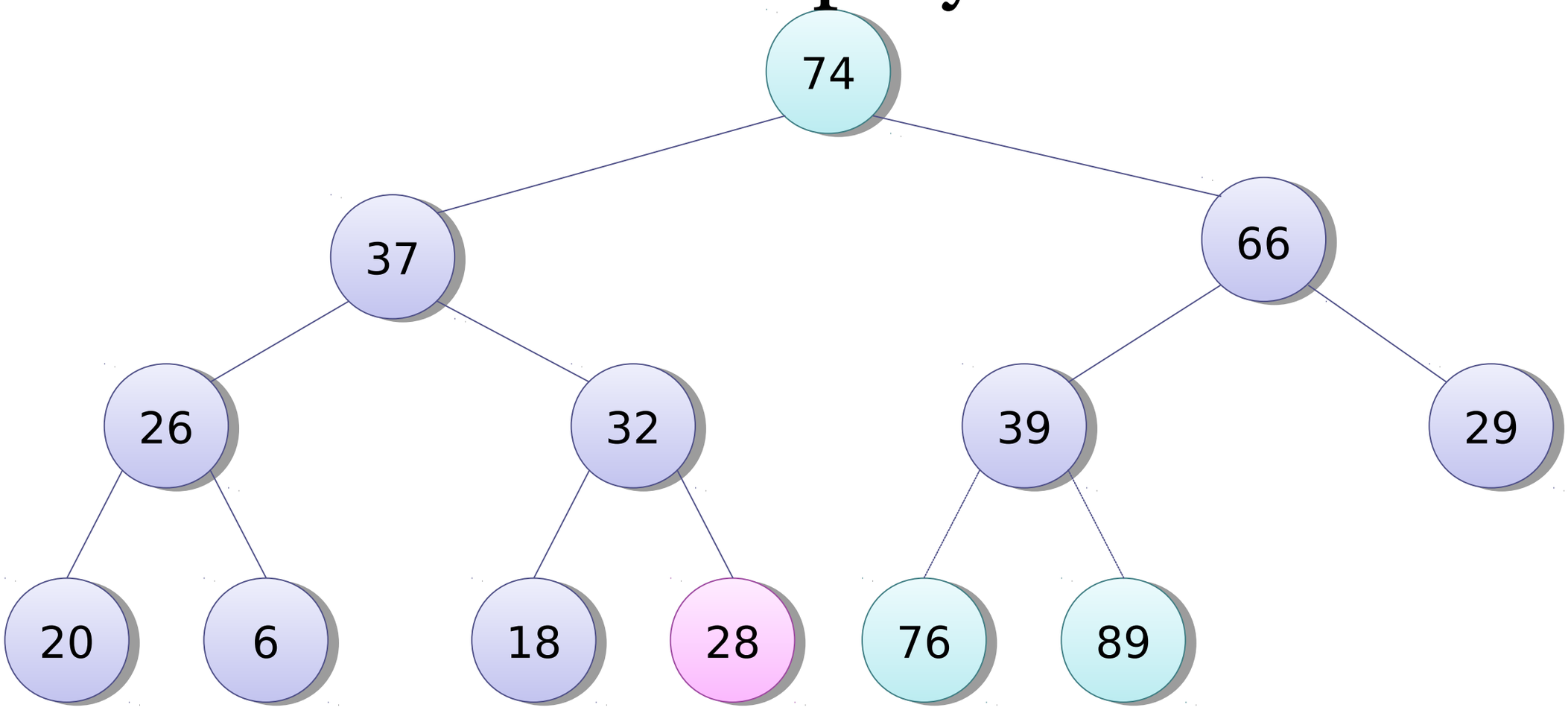


Trace of heapsort



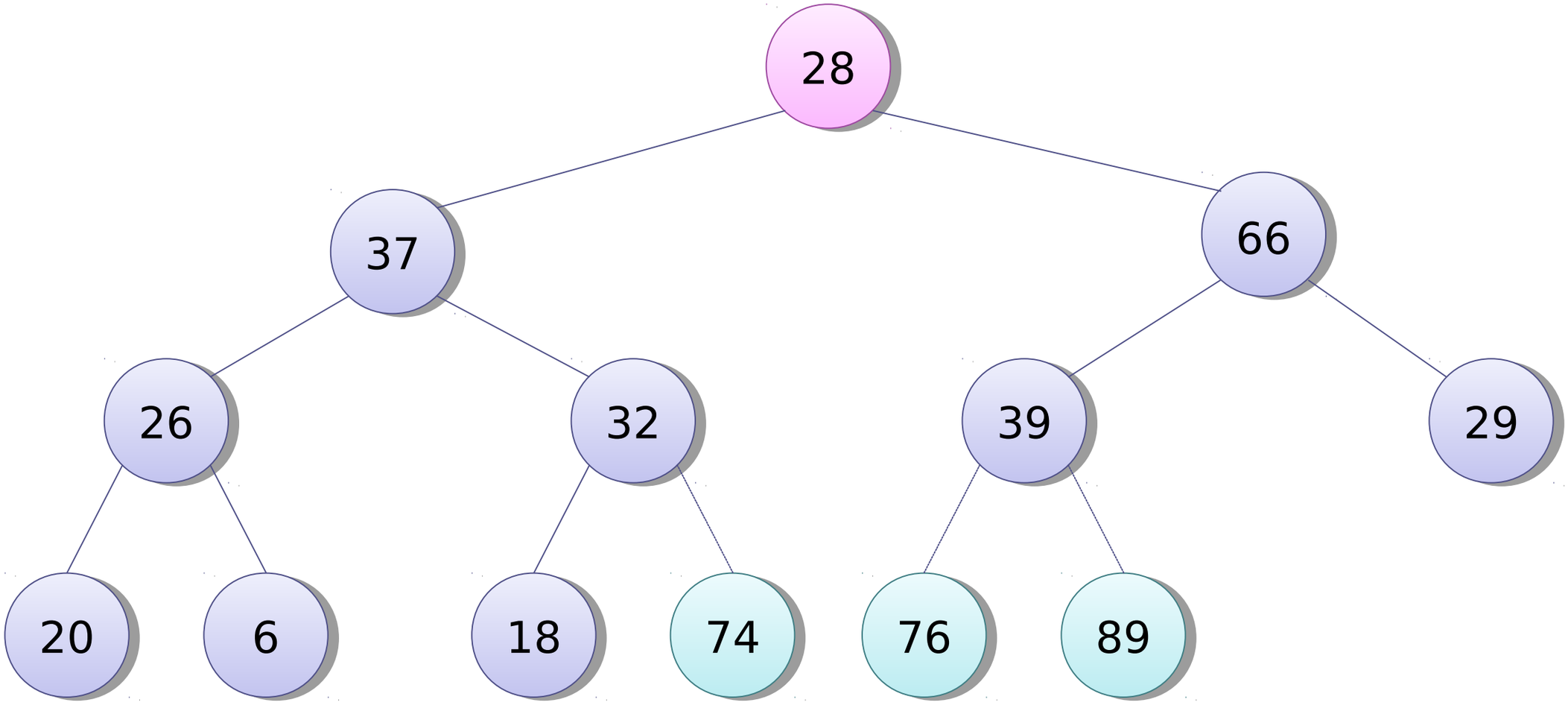
Trace of heapsort

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



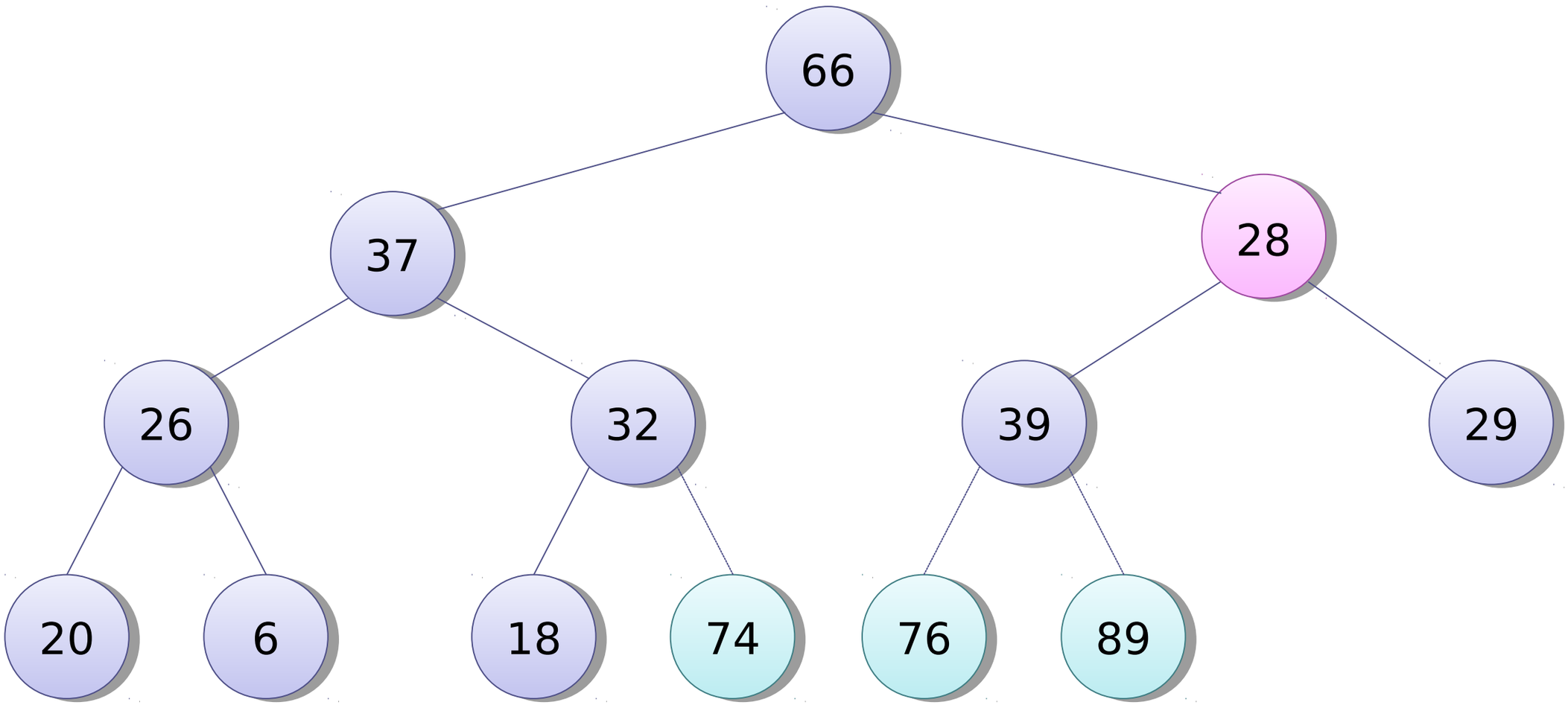
Trace of heapsort

Step 2: sift first element down



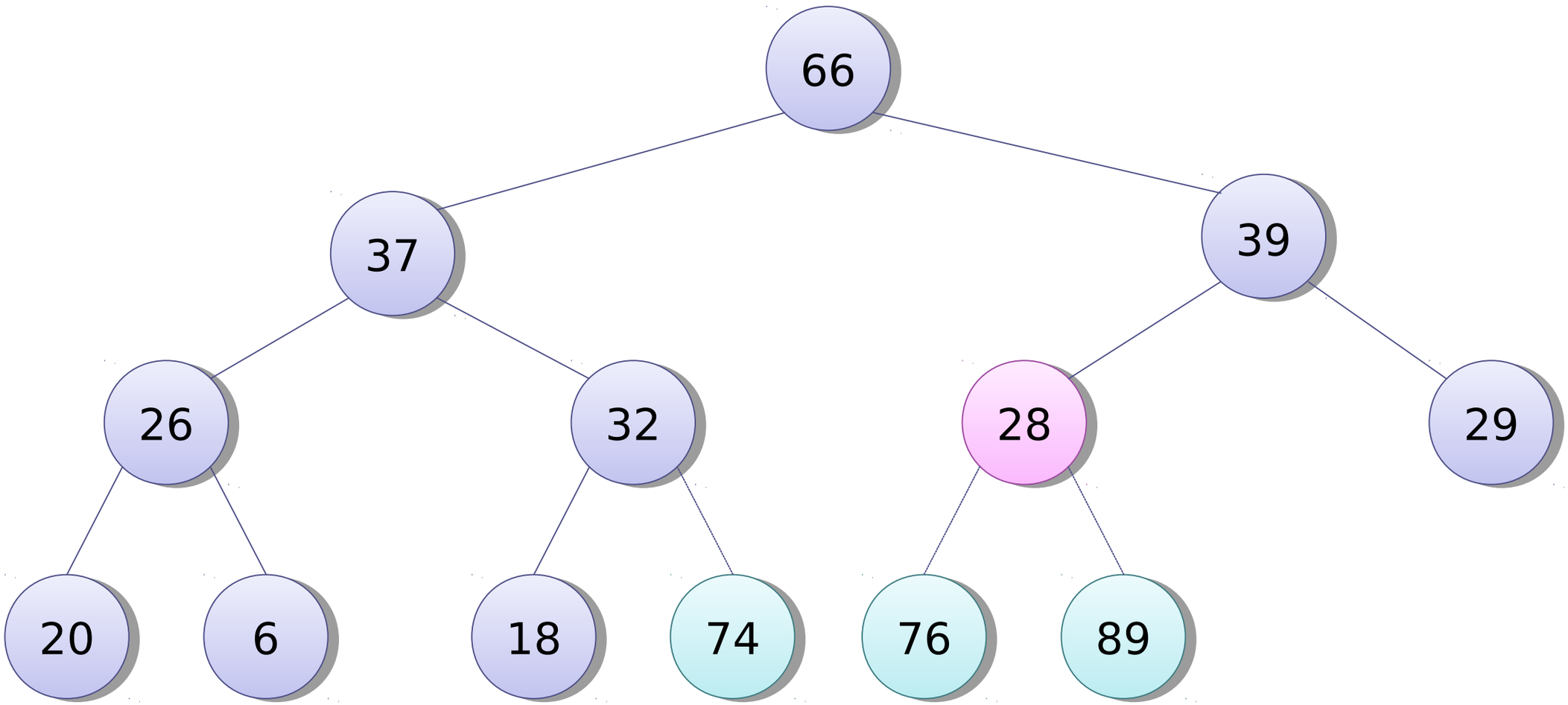
Trace of heapsort

Step 2: sift first element down

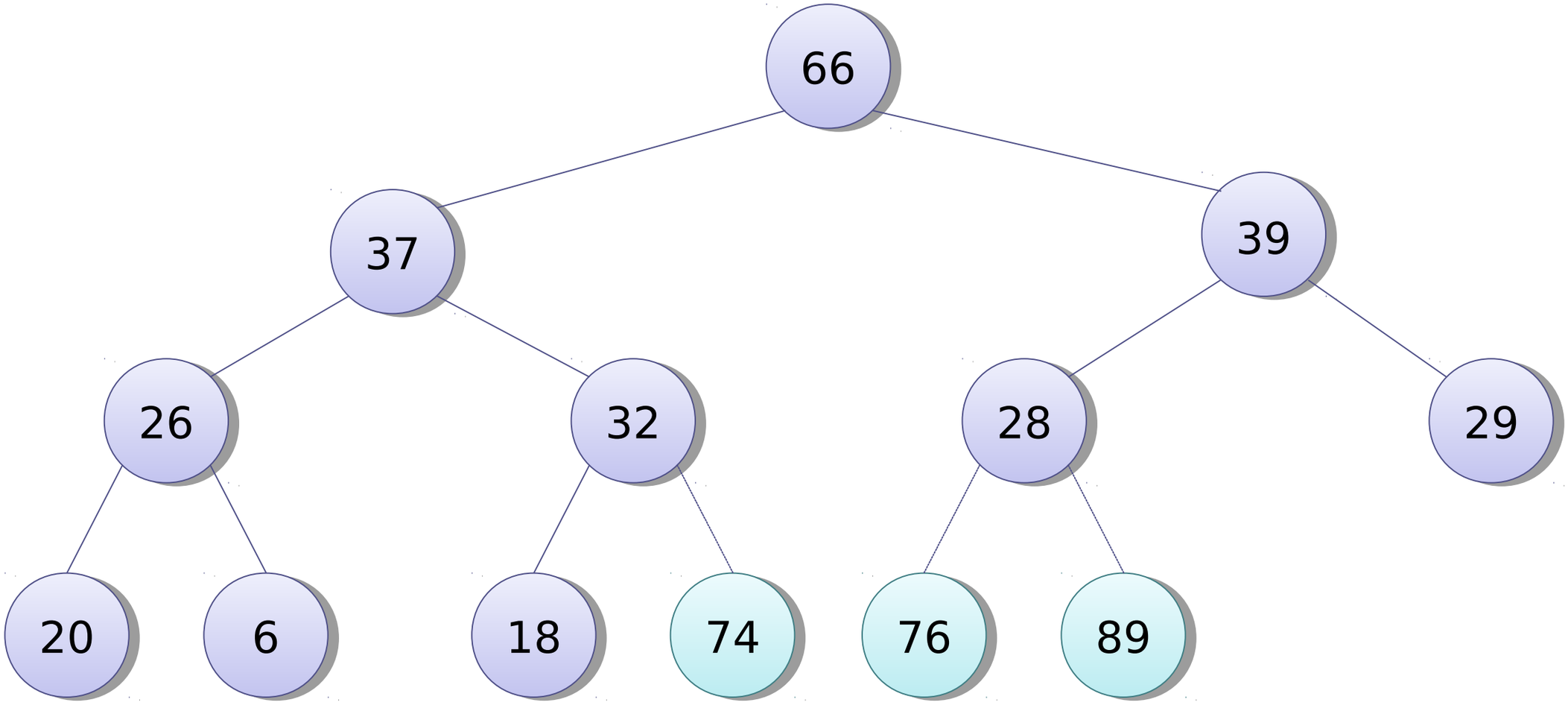


Trace of heapsort

Step 2: sift first element down

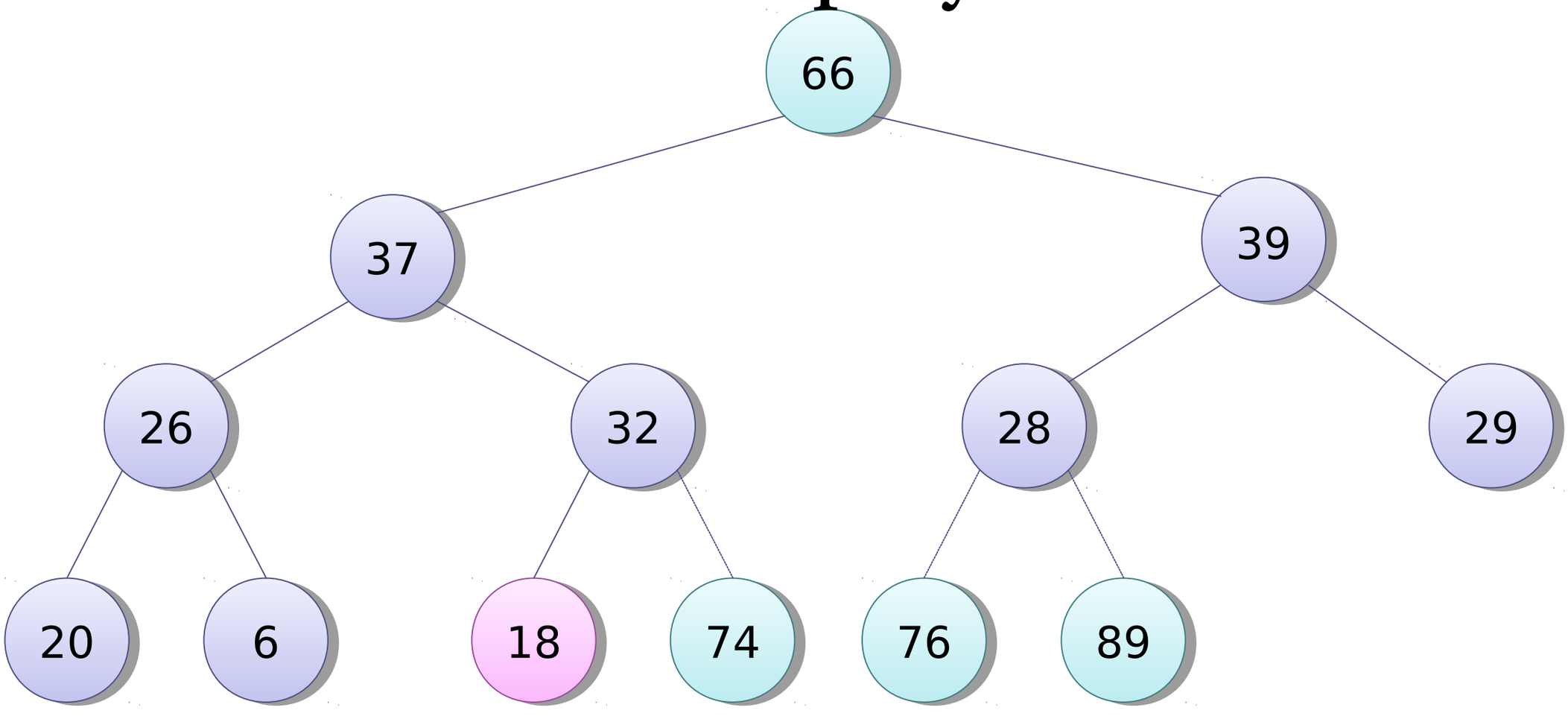


Trace of heapsort



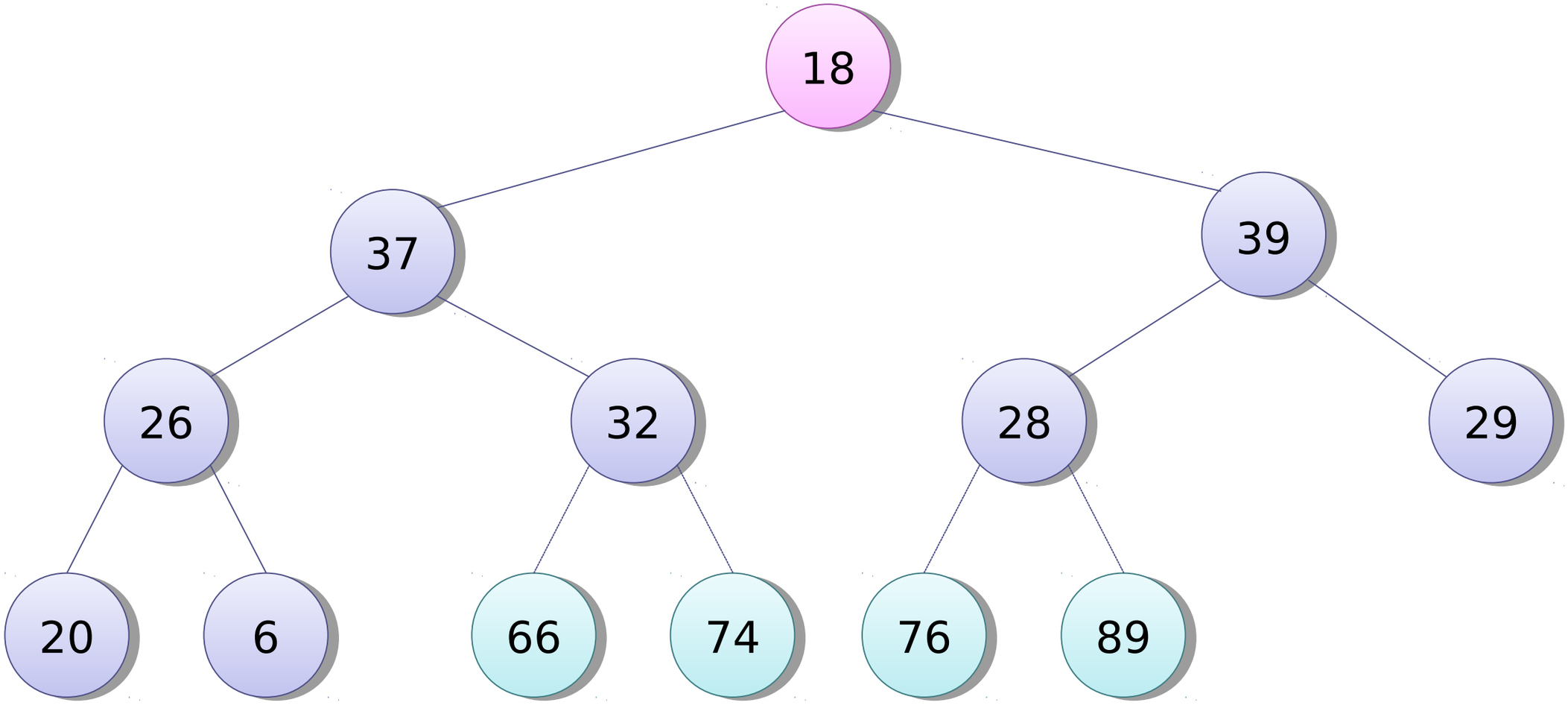
Trace of heapsort

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



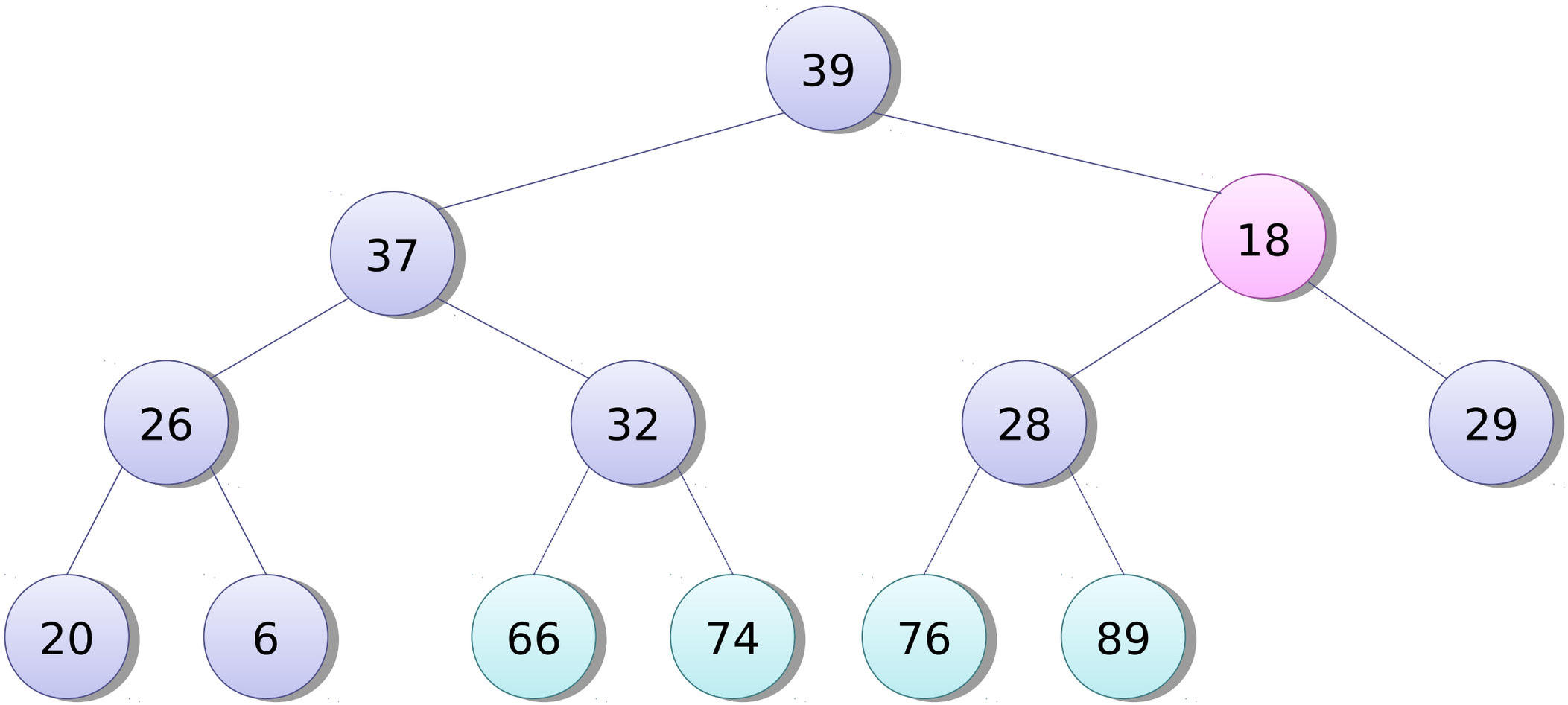
Trace of heapsort

Step 2: sift first element down



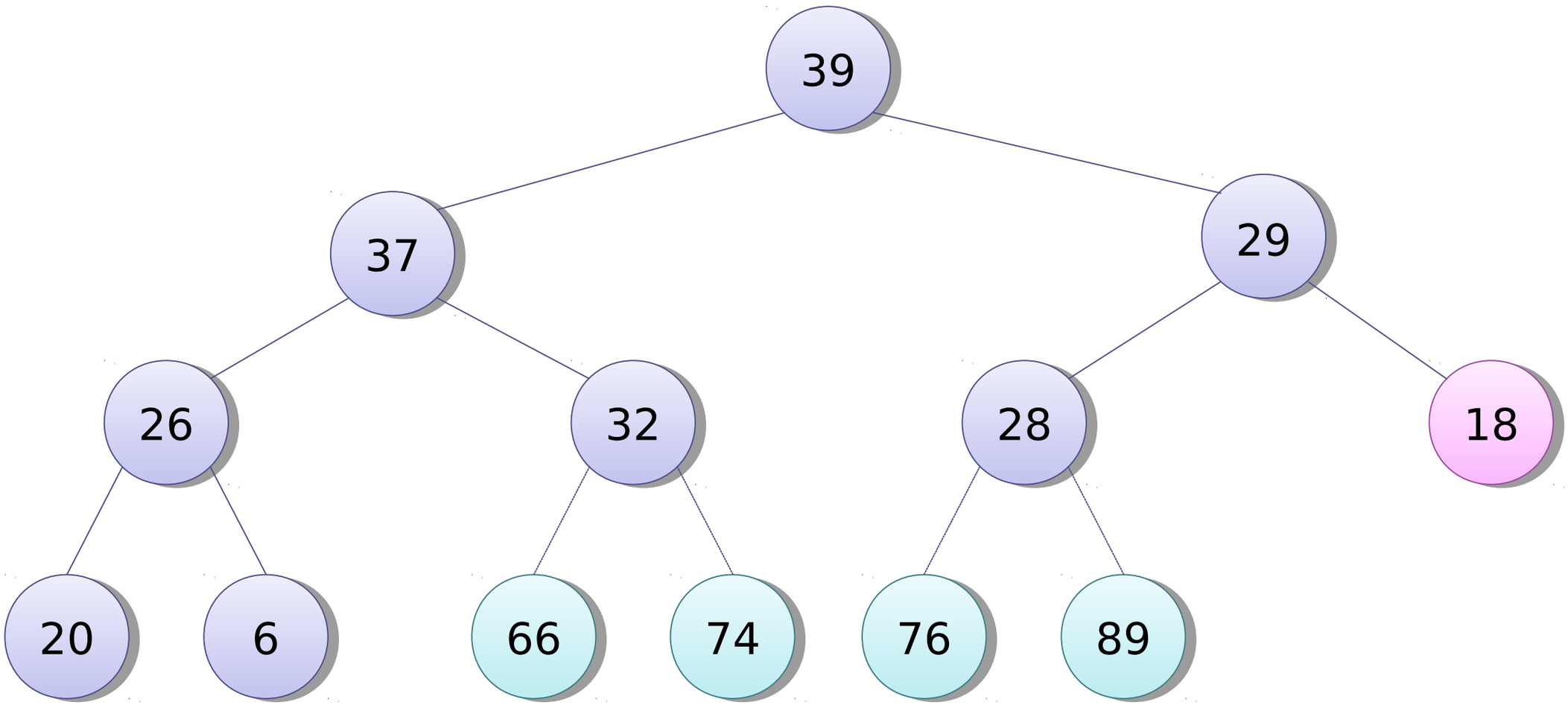
Trace of heapsort

Step 2: sift first element down

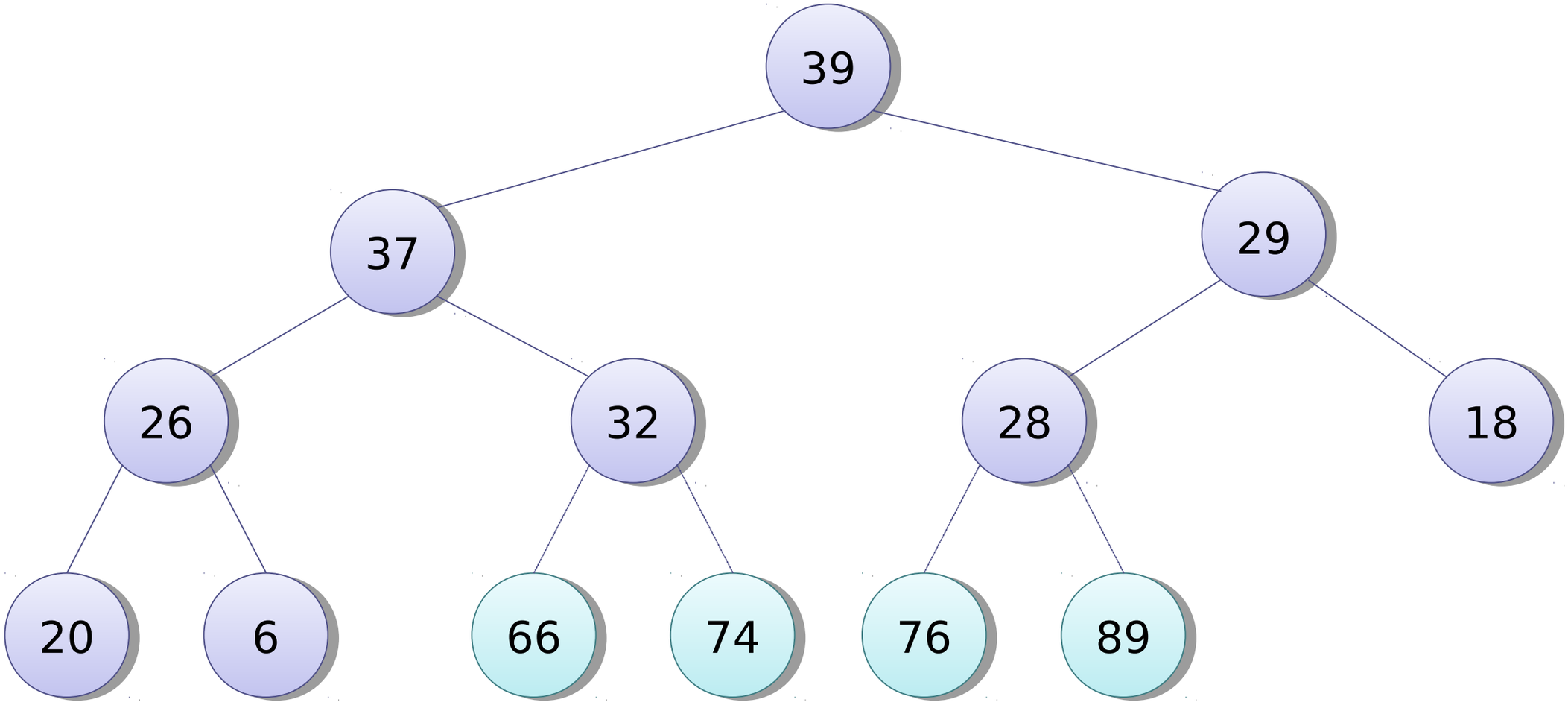


Trace of heapsort

Step 2: sift first element down

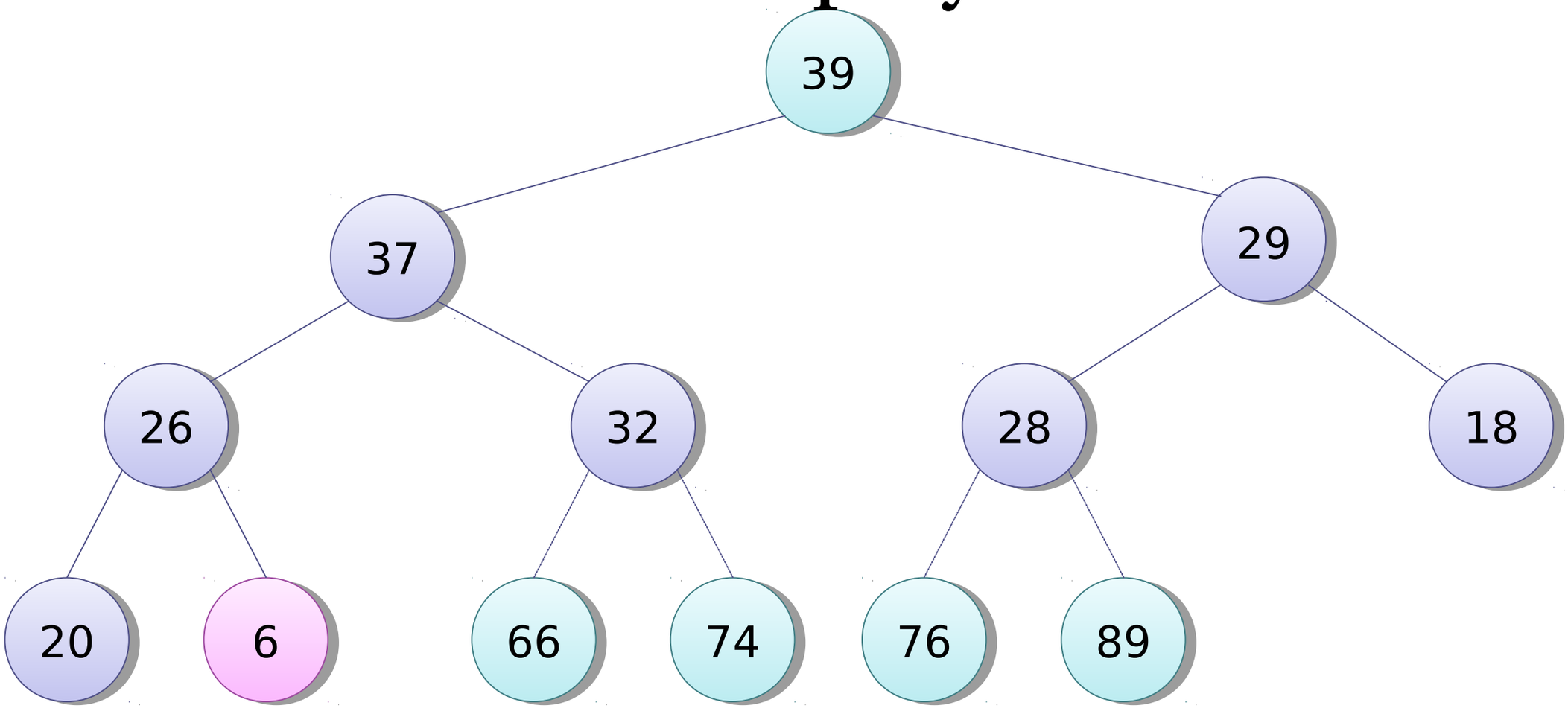


Trace of heapsort



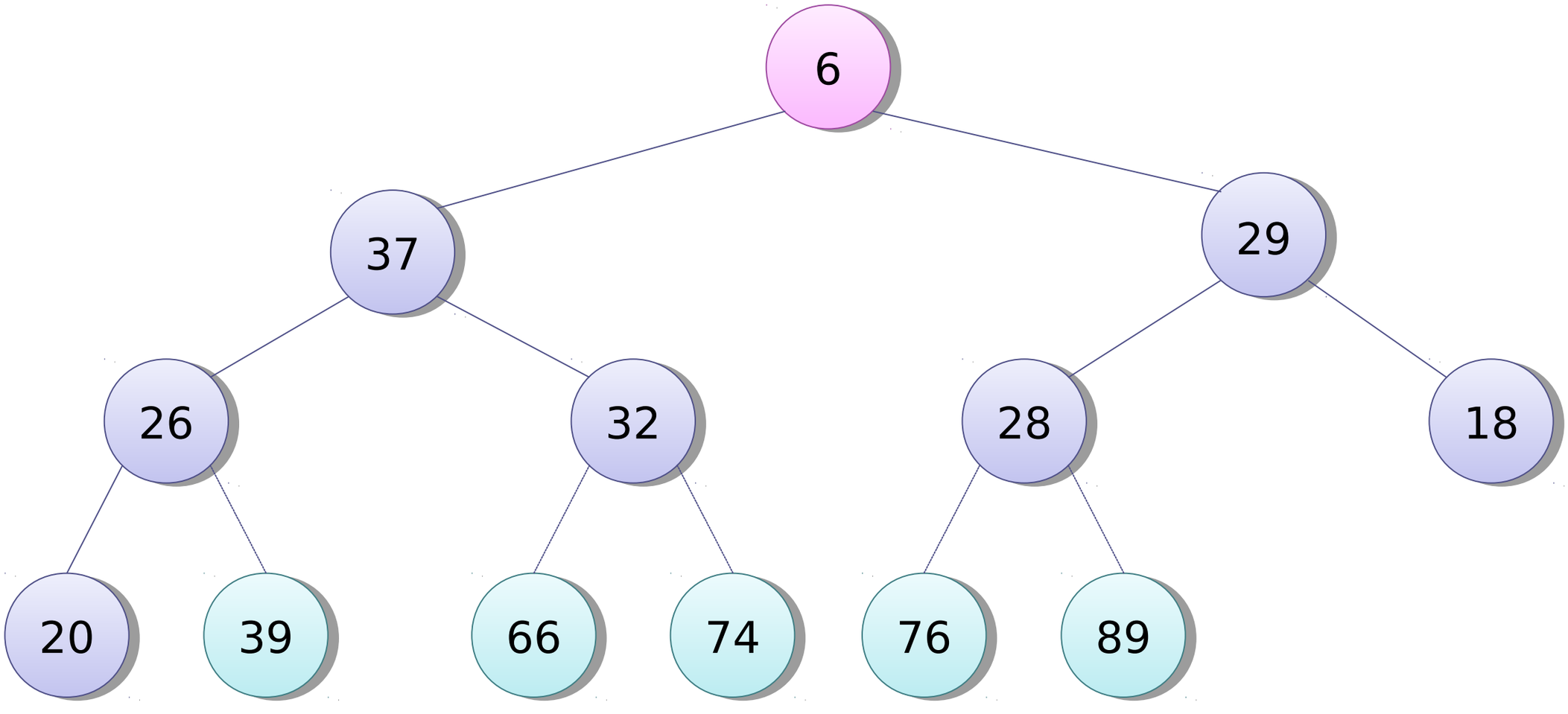
Trace of heapsort

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



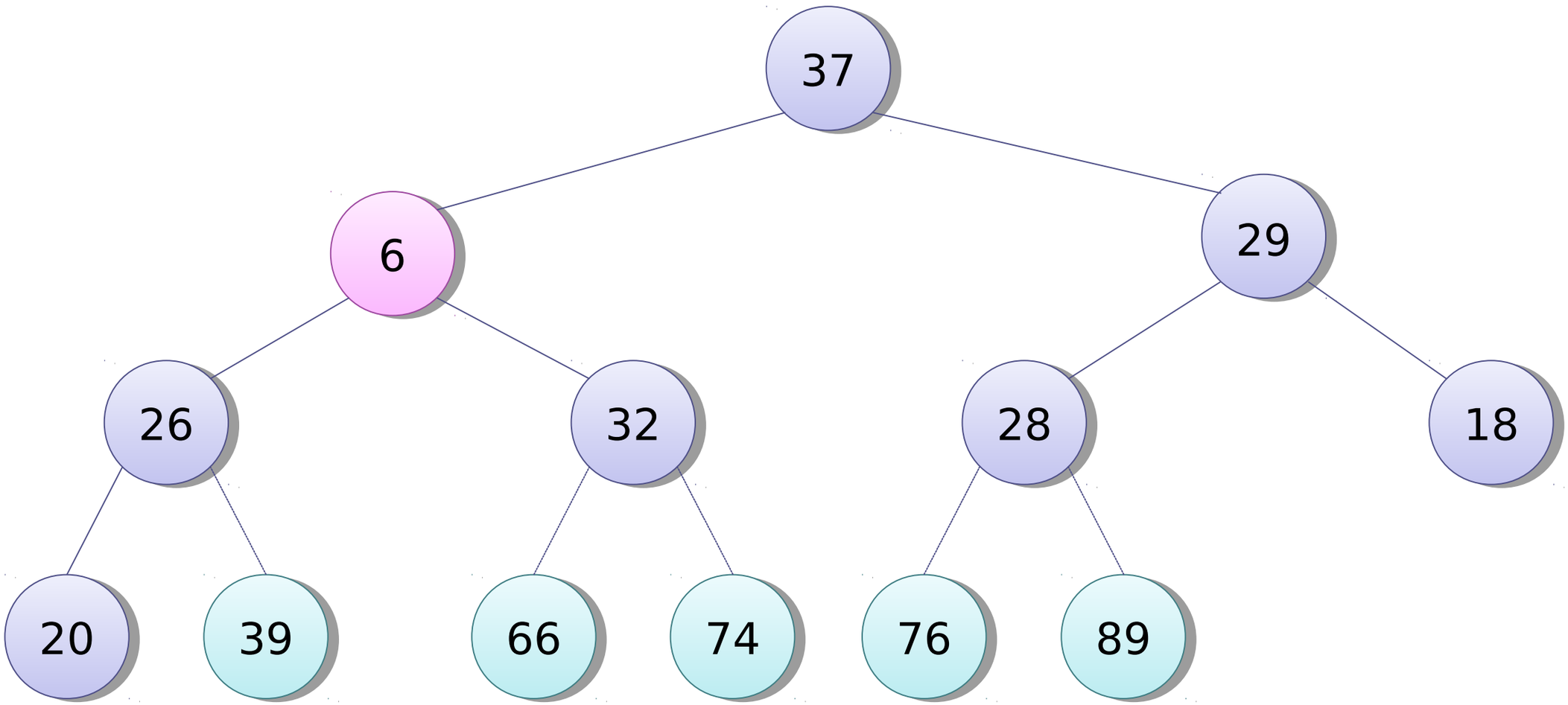
Trace of heapsort

Step 2: sift first element down



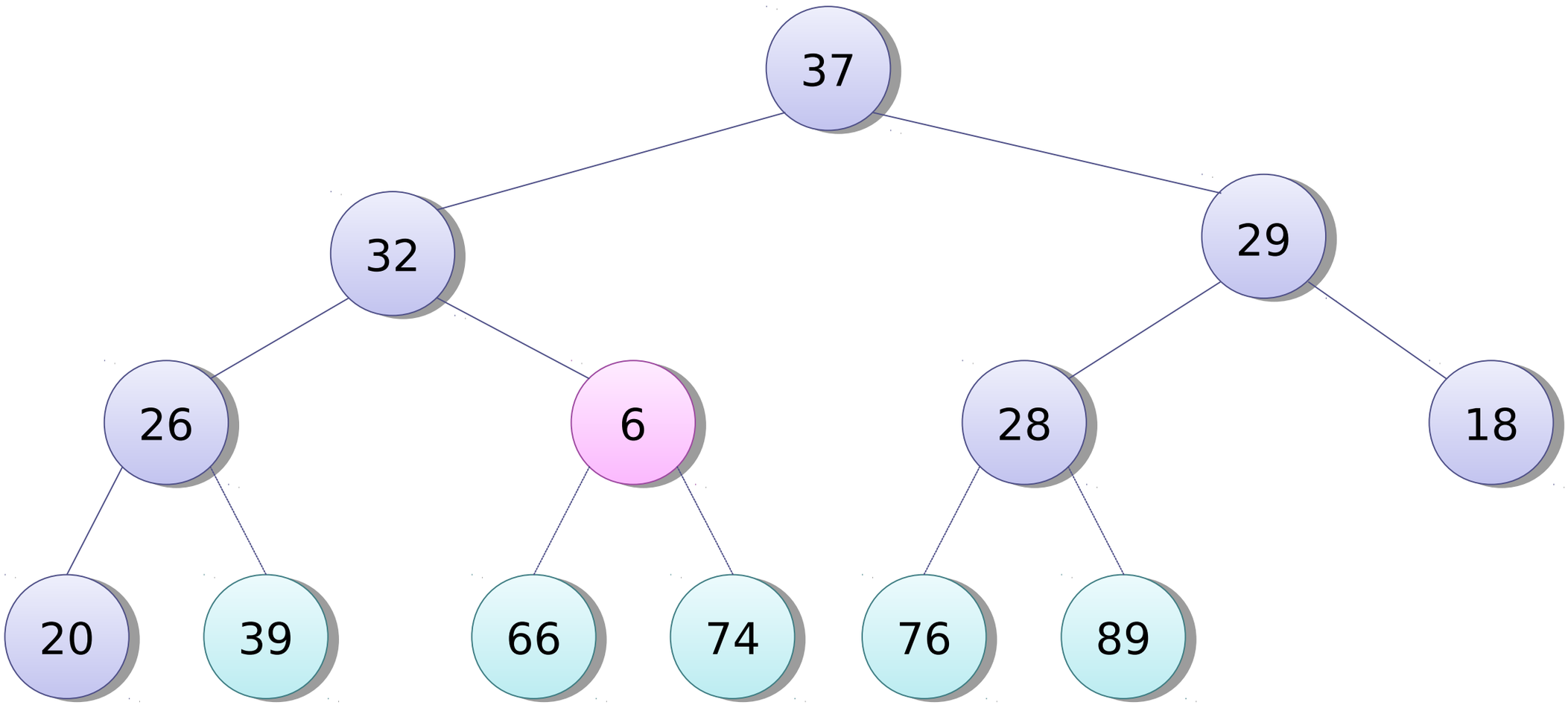
Trace of heapsort

Step 2: sift first element down

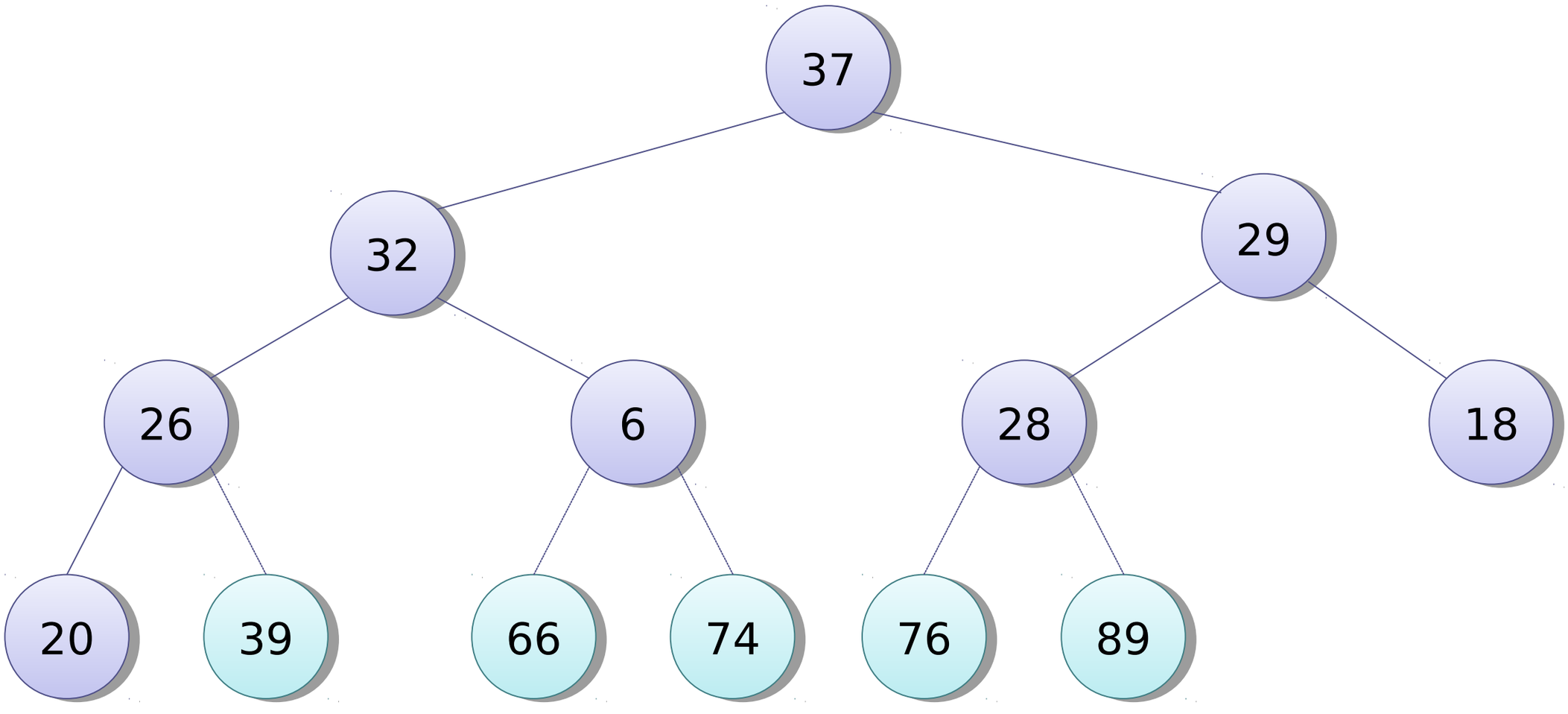


Trace of heapsort

Step 2: sift first element down

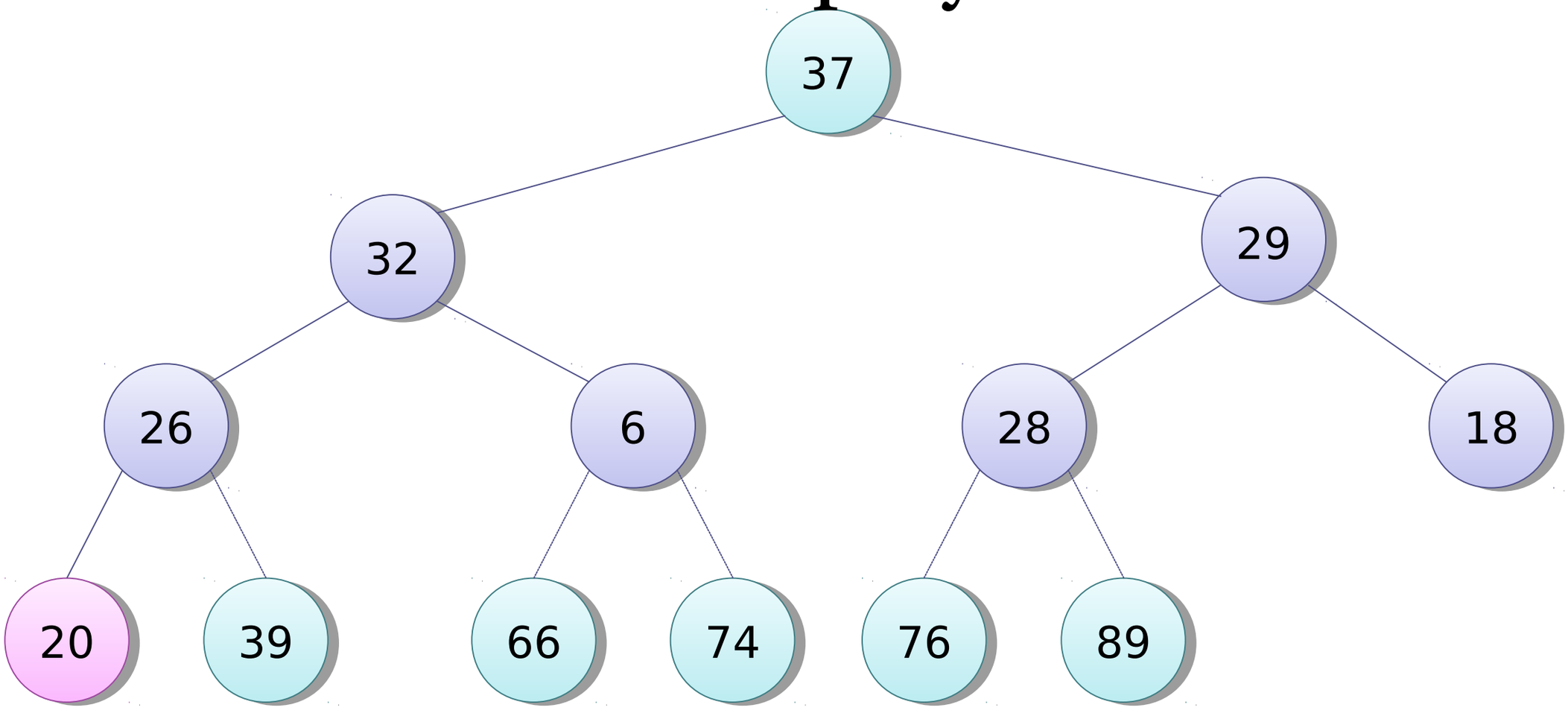


Trace of heapsort



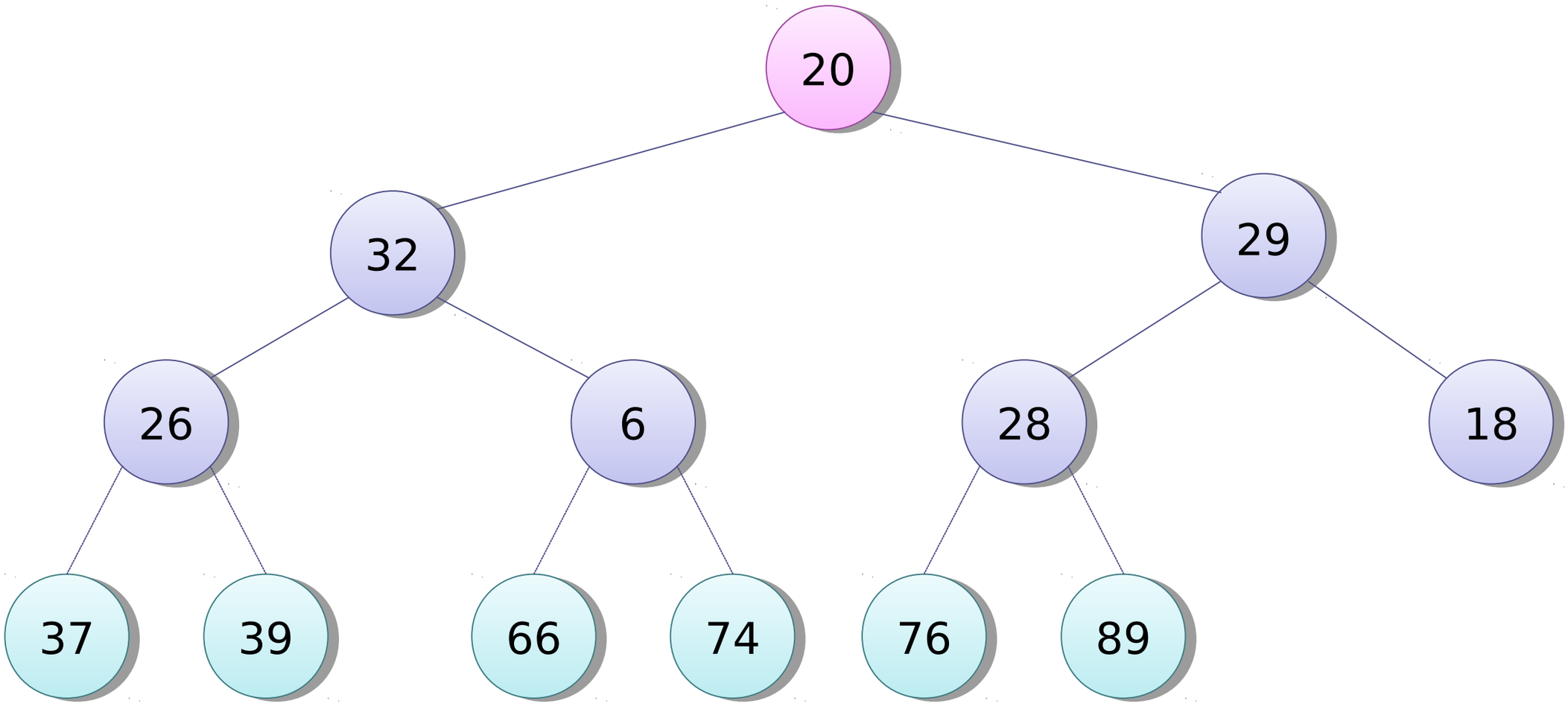
Trace of heapsort

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



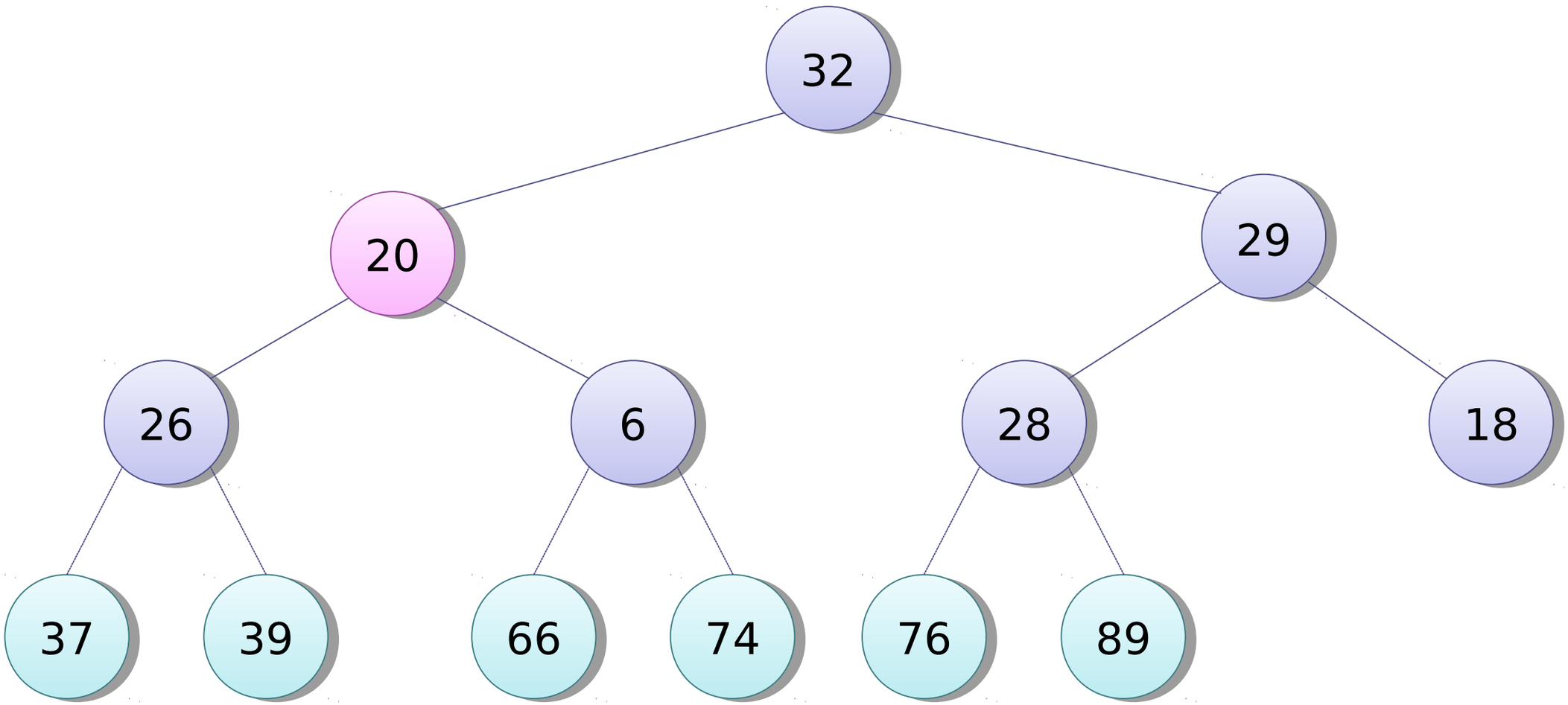
Trace of heapsort

Step 2: sift first element down



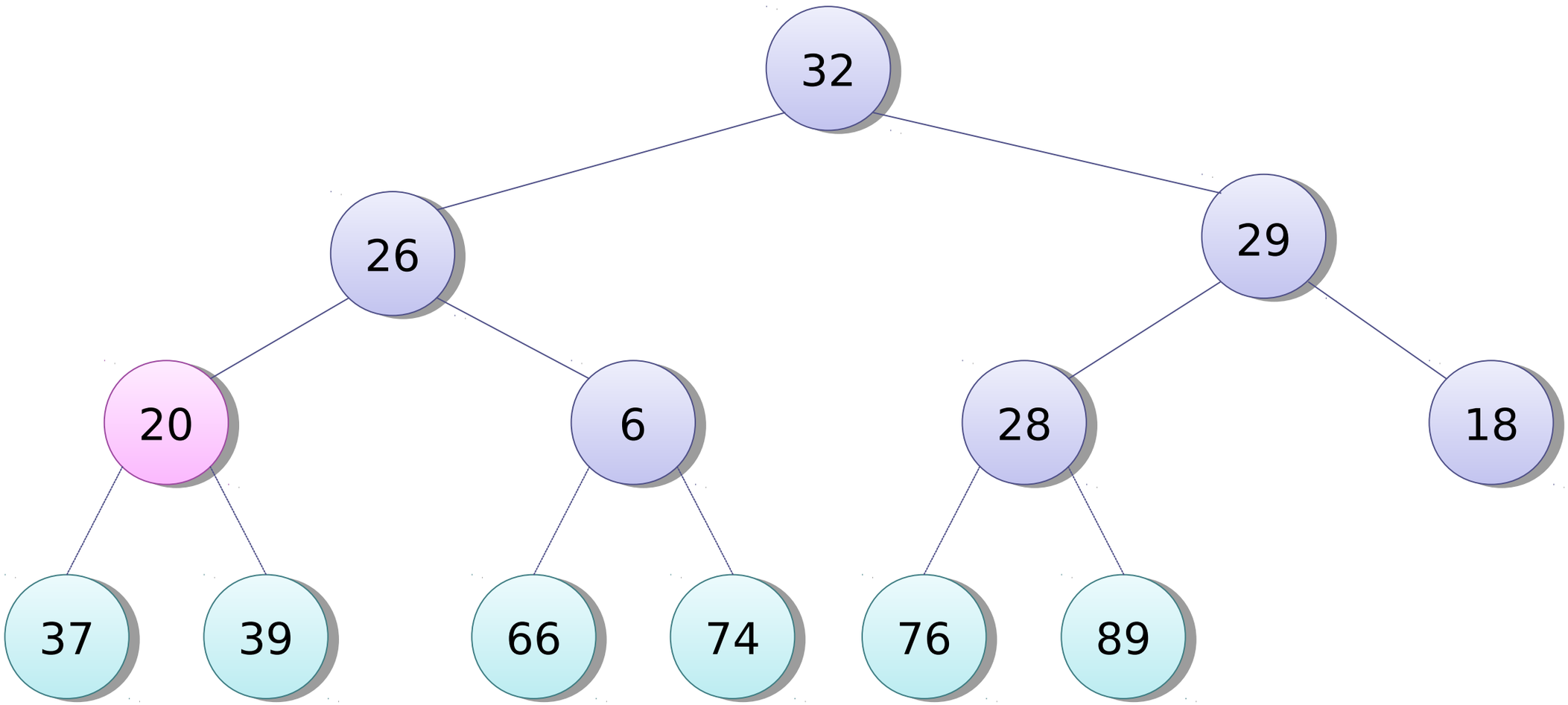
Trace of heapsort

Step 2: sift first element down

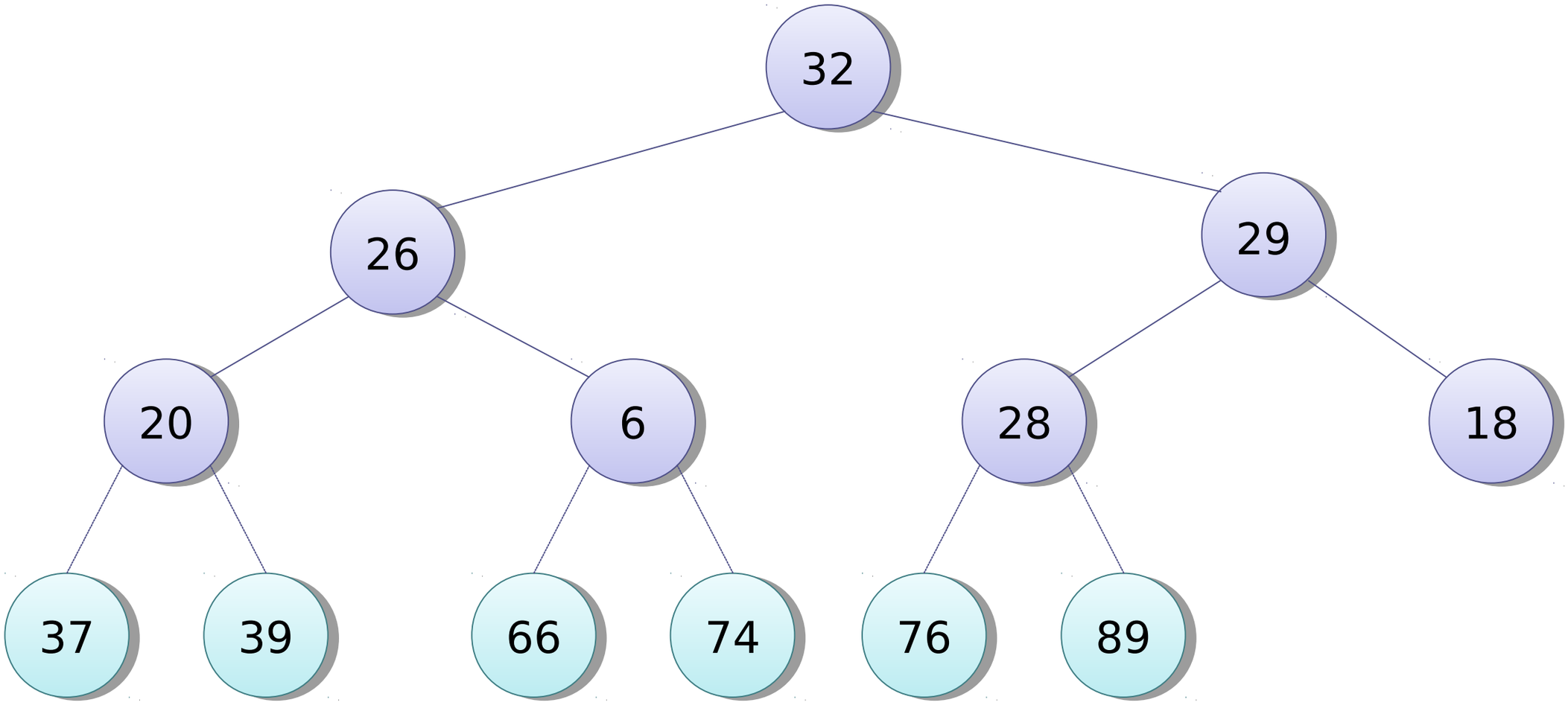


Trace of heapsort

Step 2: sift first element down

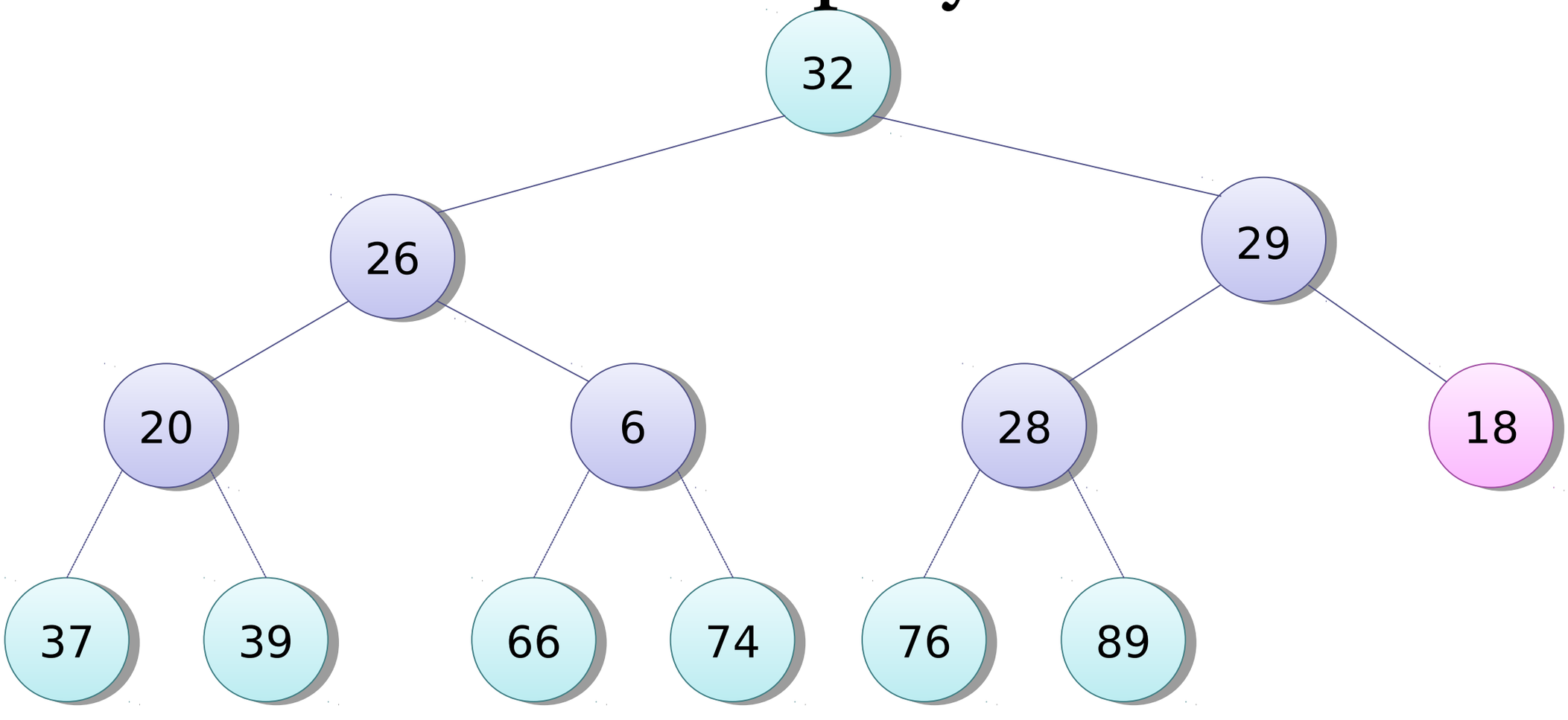


Trace of heapsort



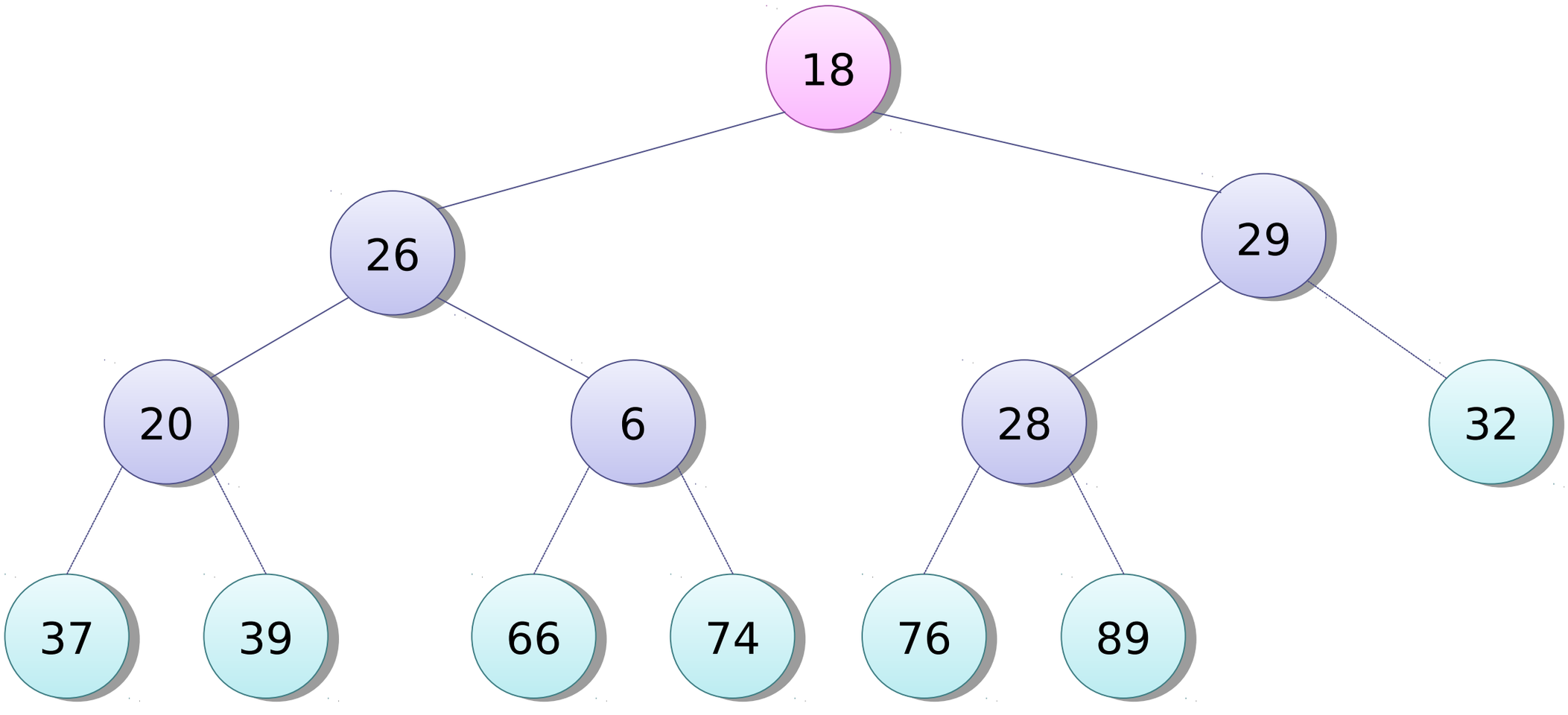
Trace of heapsort

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



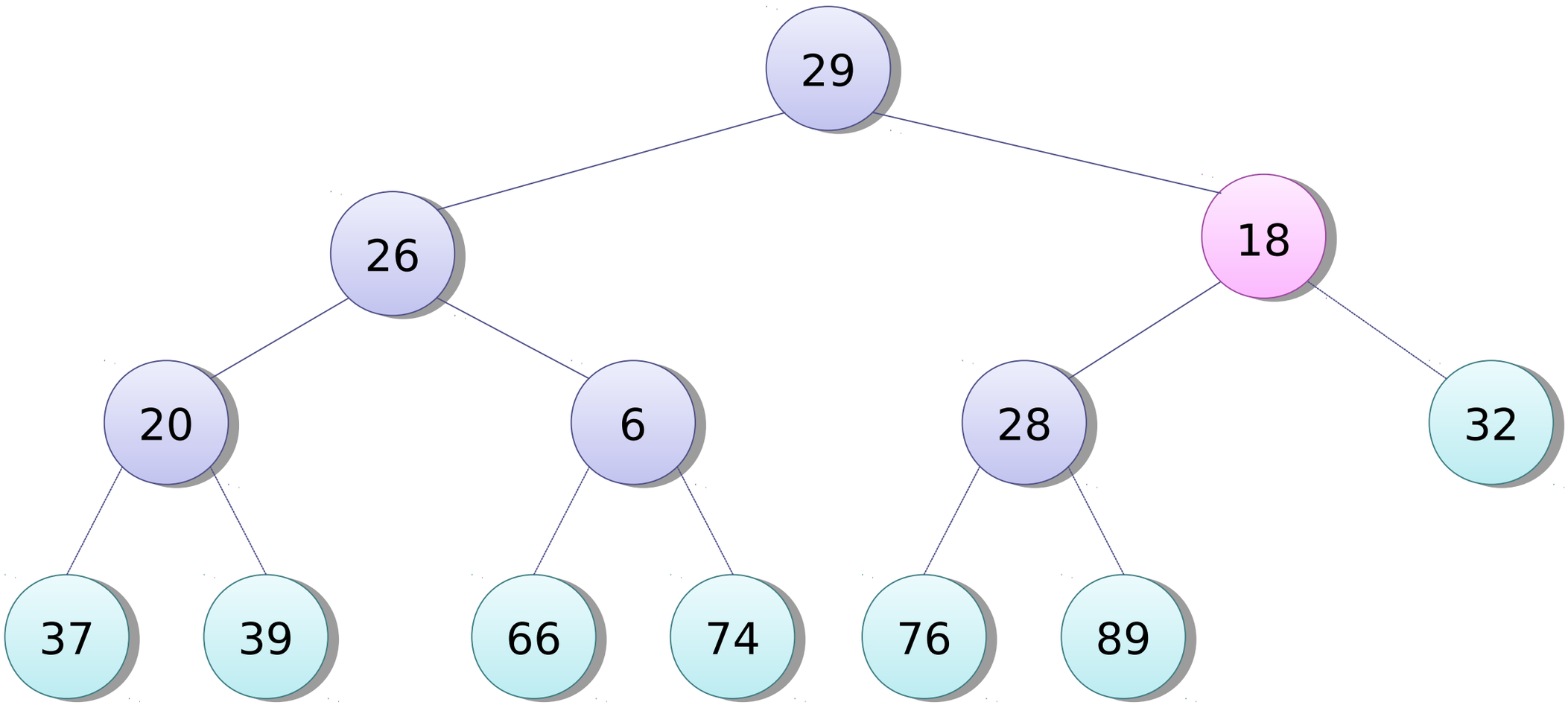
Trace of heapsort

Step 2: sift first element down



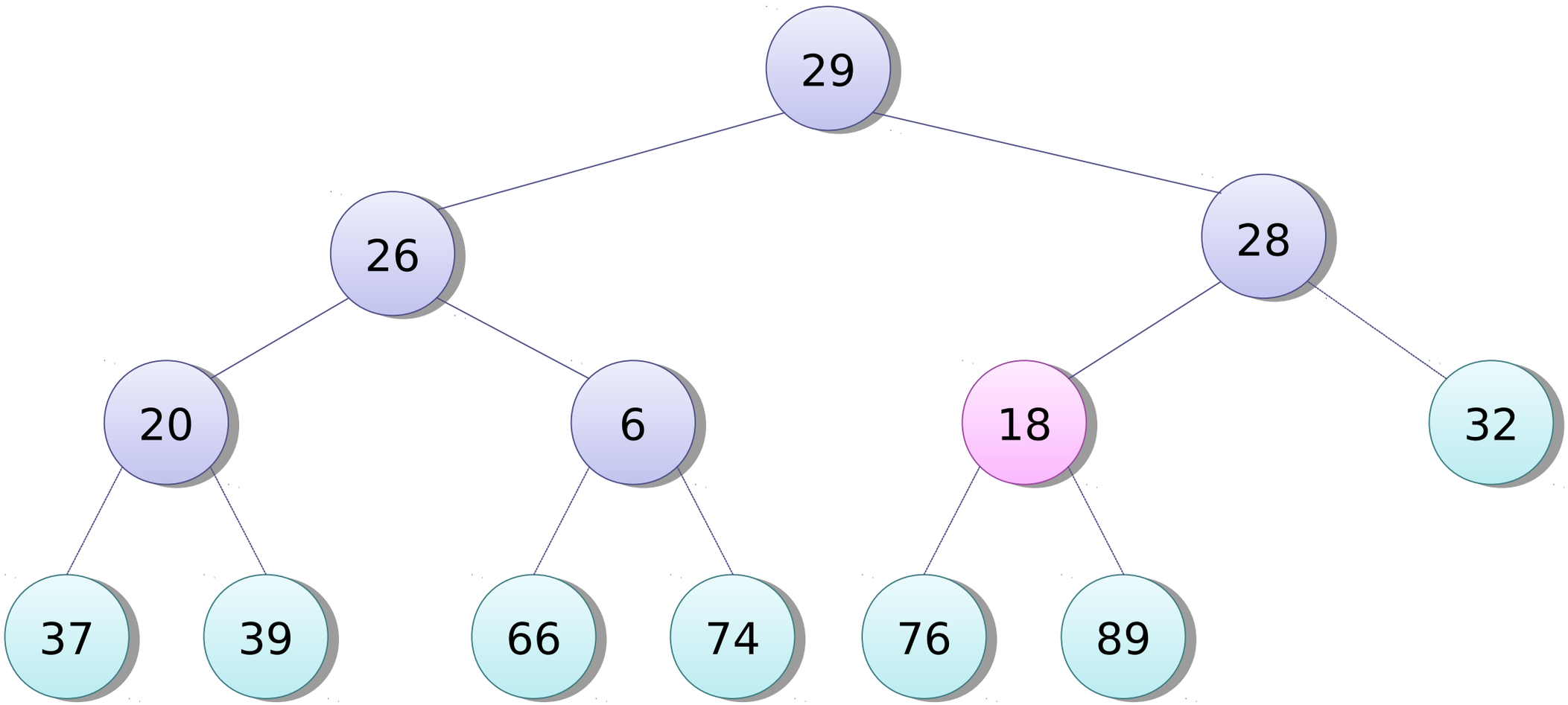
Trace of heapsort

Step 2: sift first element down

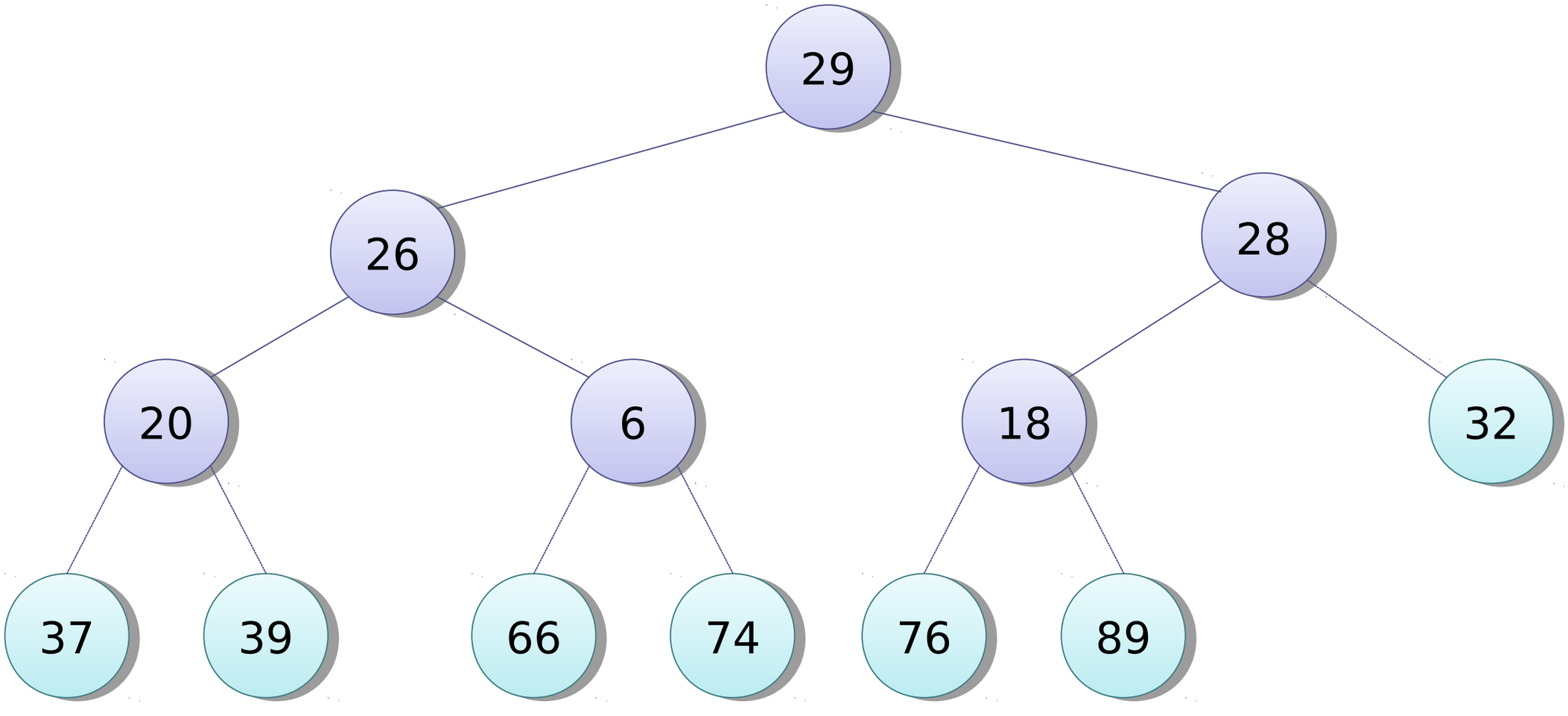


Trace of heapsort

Step 2: sift first element down

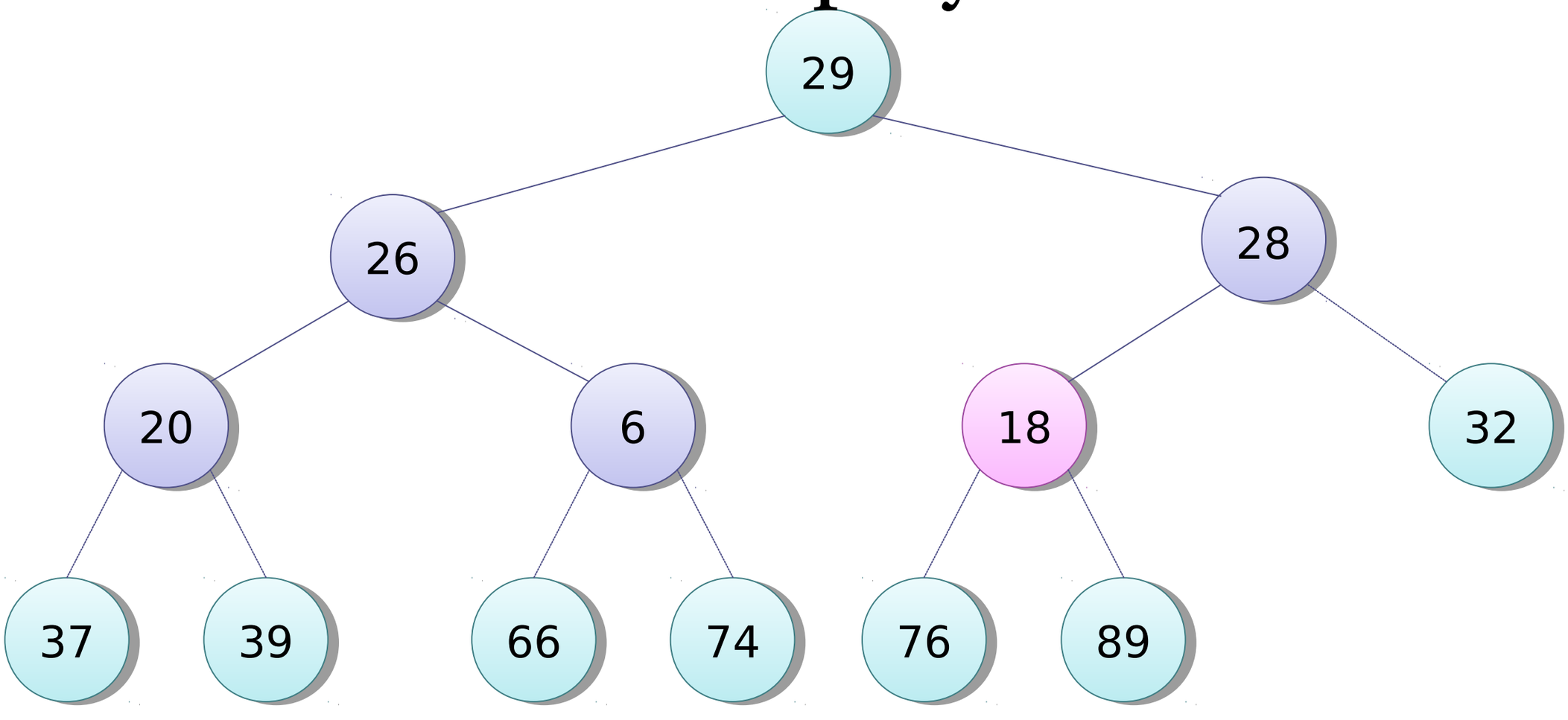


Trace of heapsort



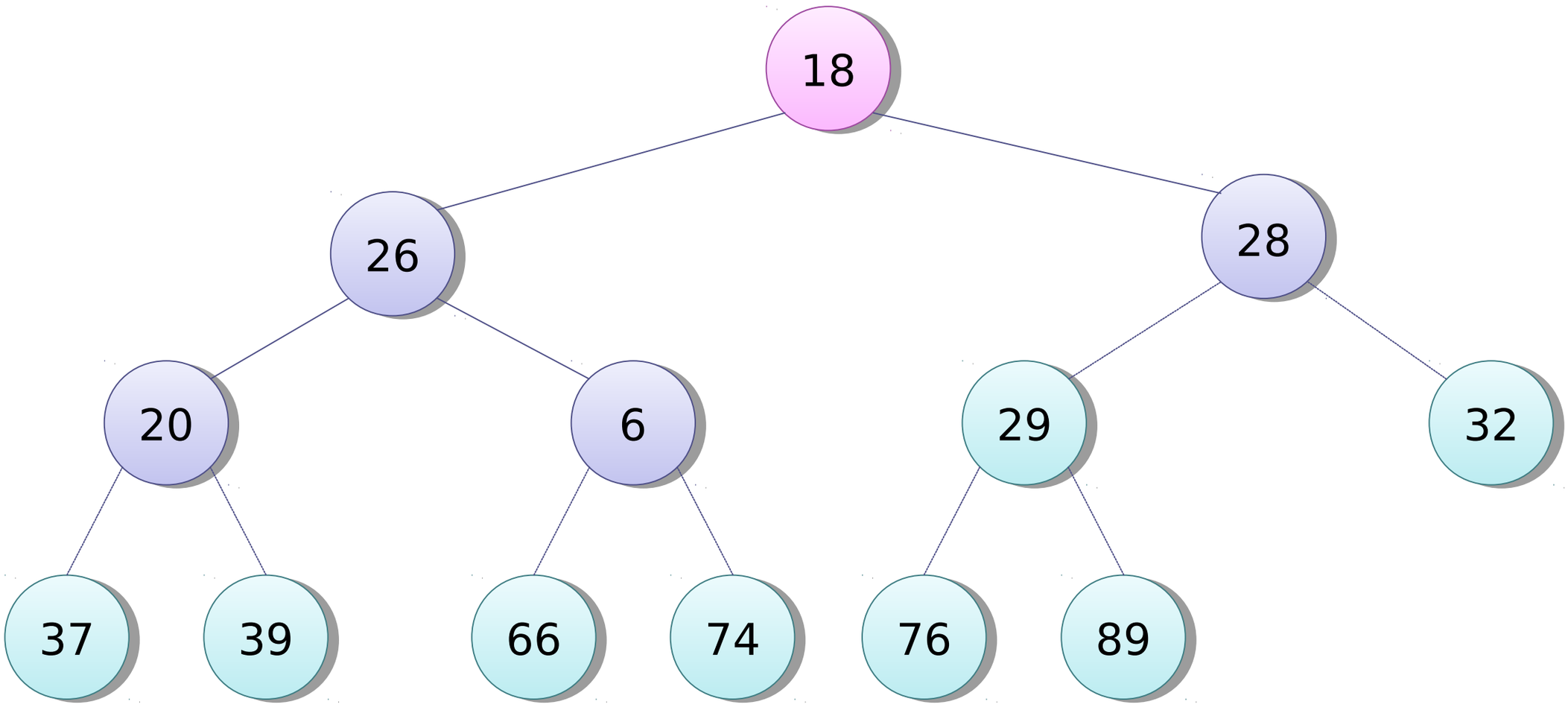
Trace of heapsort

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



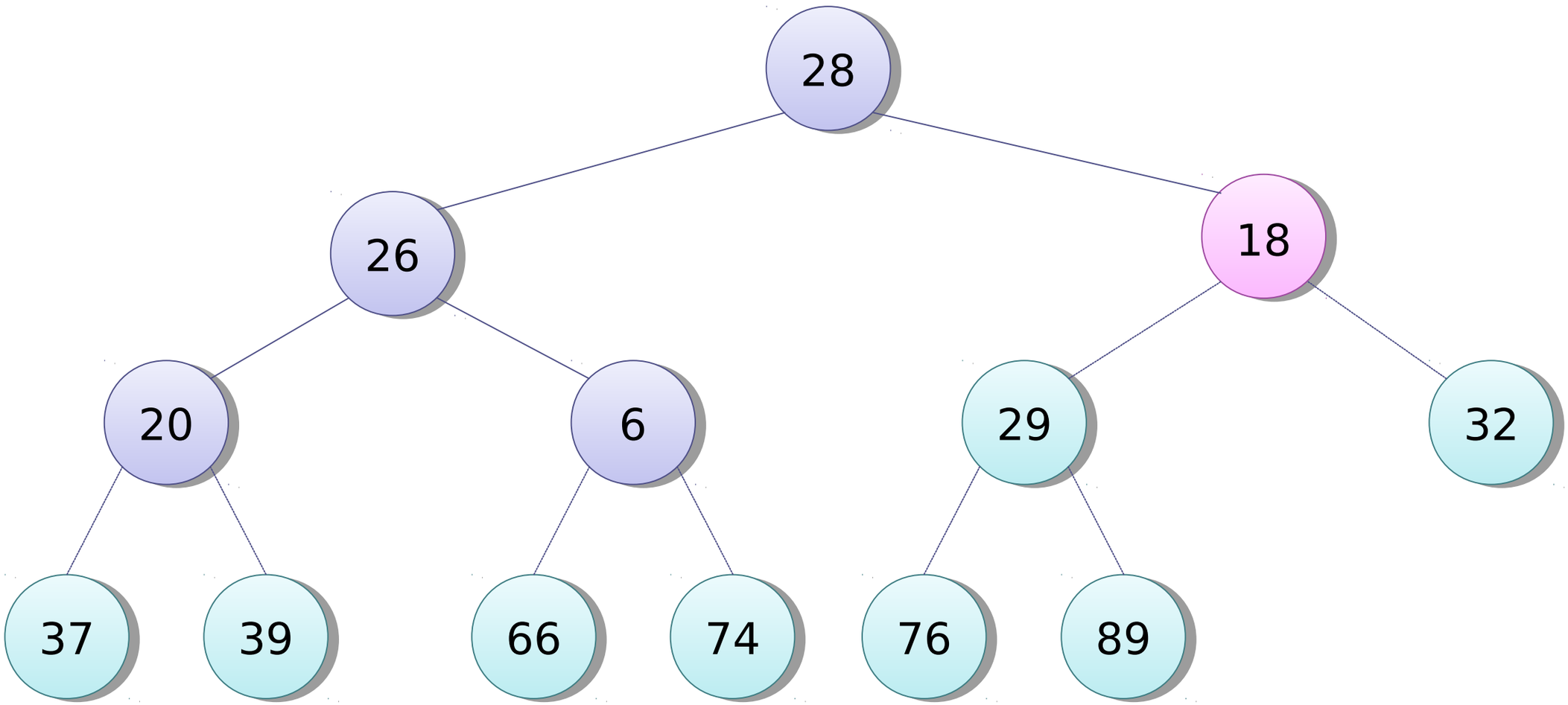
Trace of heapsort

Step 2: sift first element down

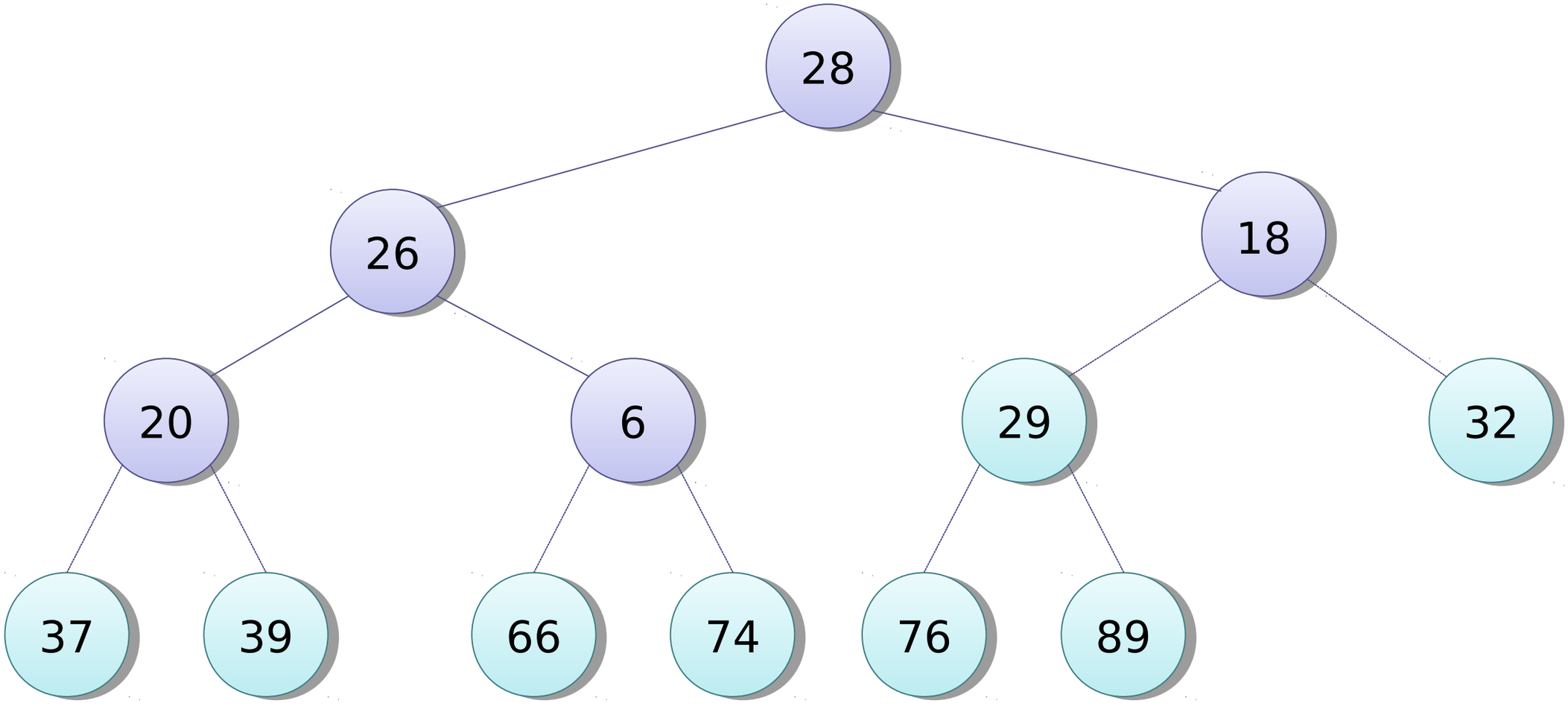


Trace of heapsort

Step 2: sift first element down

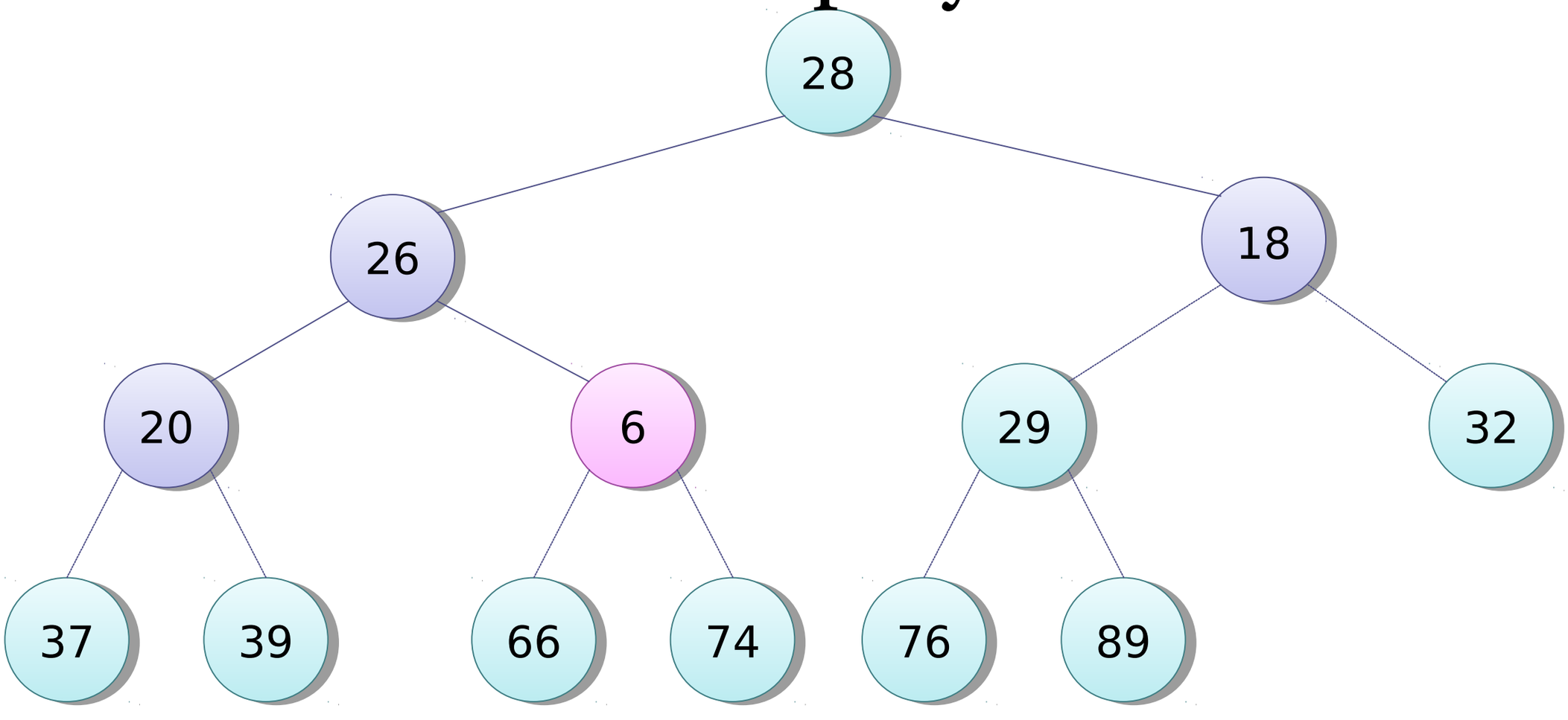


Trace of heapsort



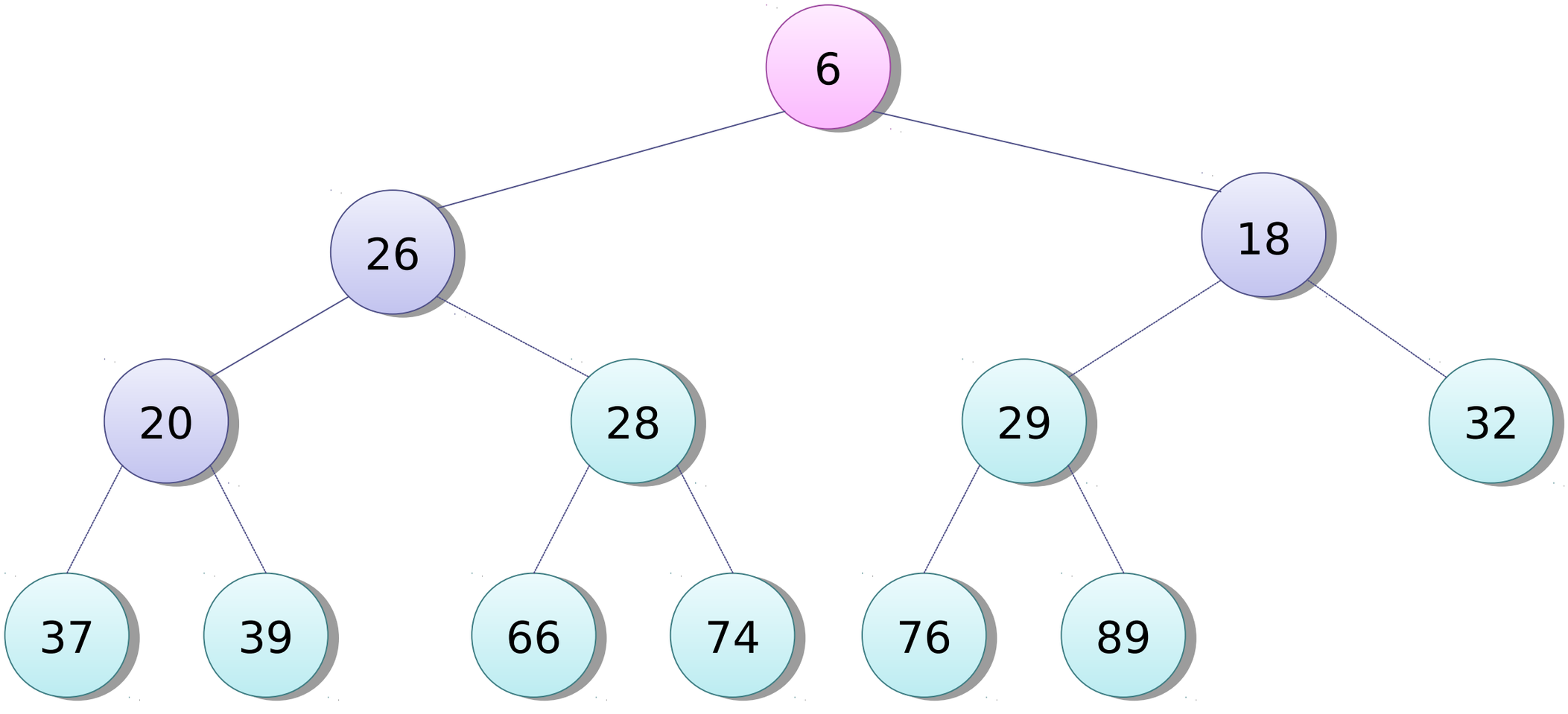
Trace of heapsort

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



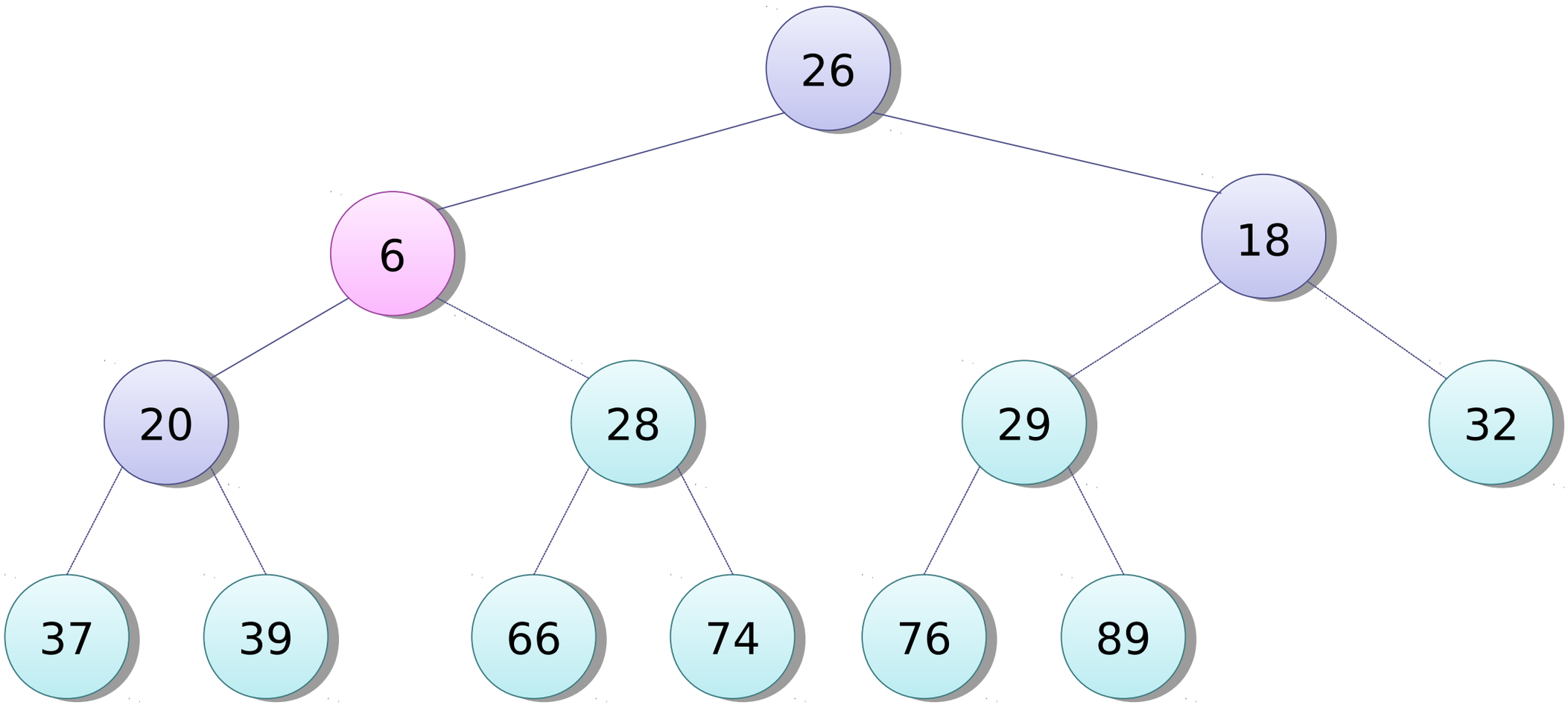
Trace of heapsort

Step 2: sift first element down



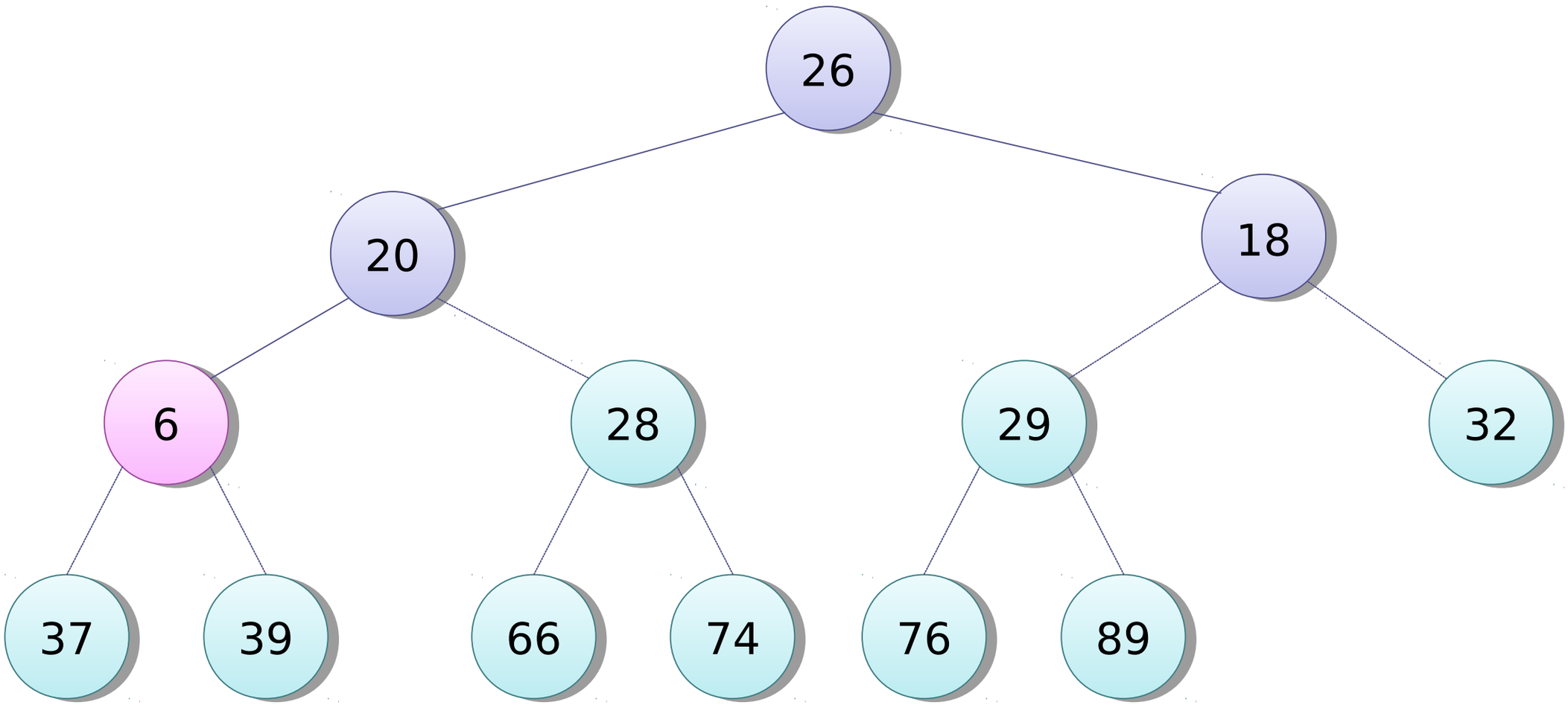
Trace of heapsort

Step 2: sift first element down

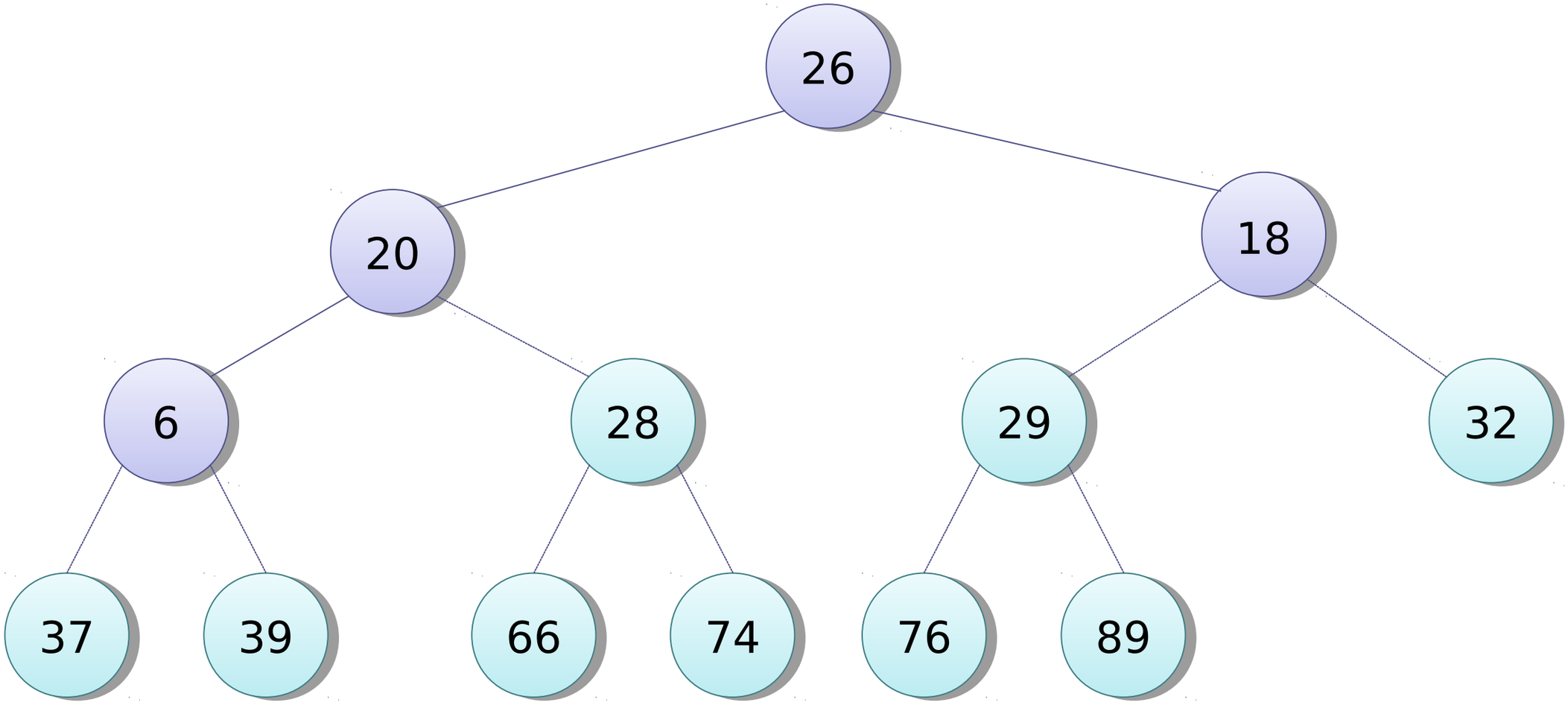


Trace of heapsort

Step 2: sift first element down

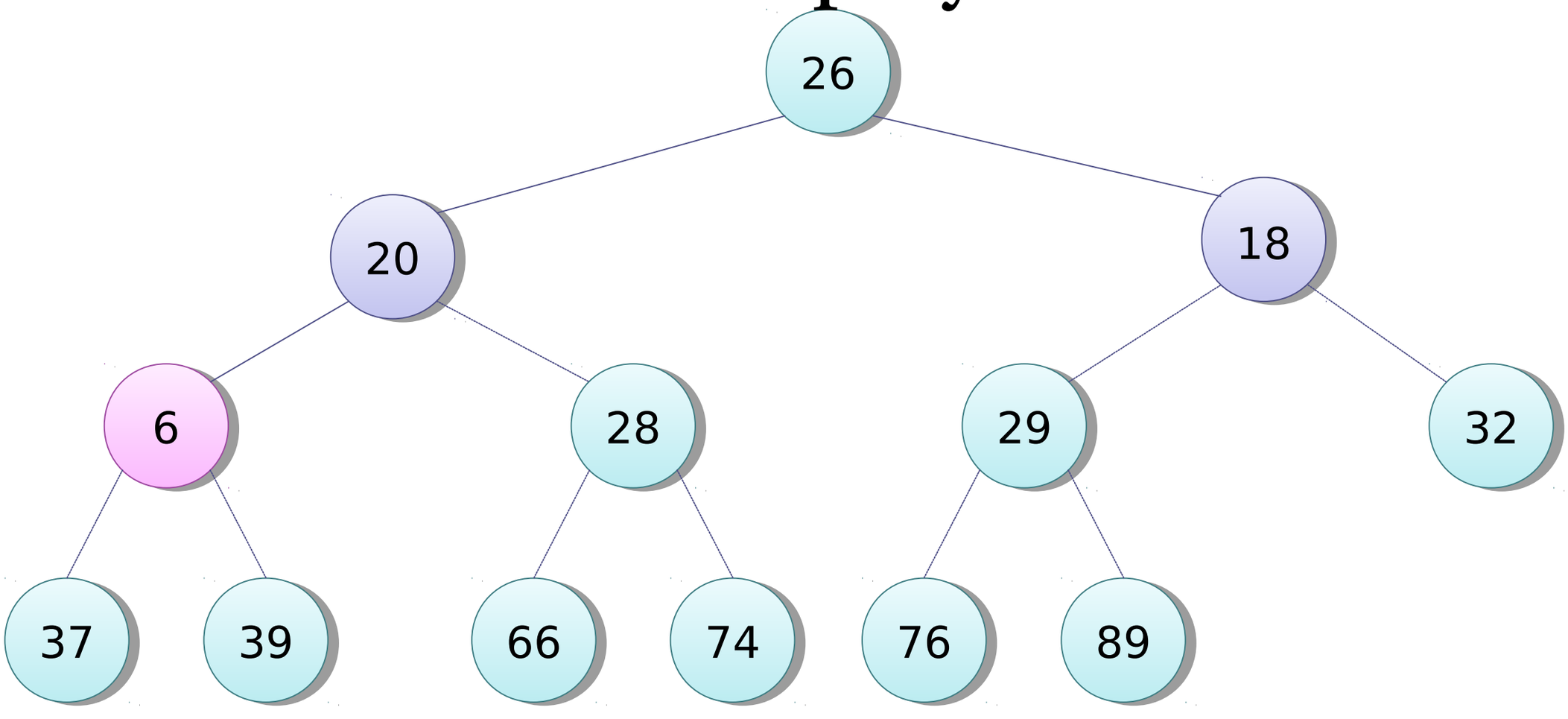


Trace of heapsort



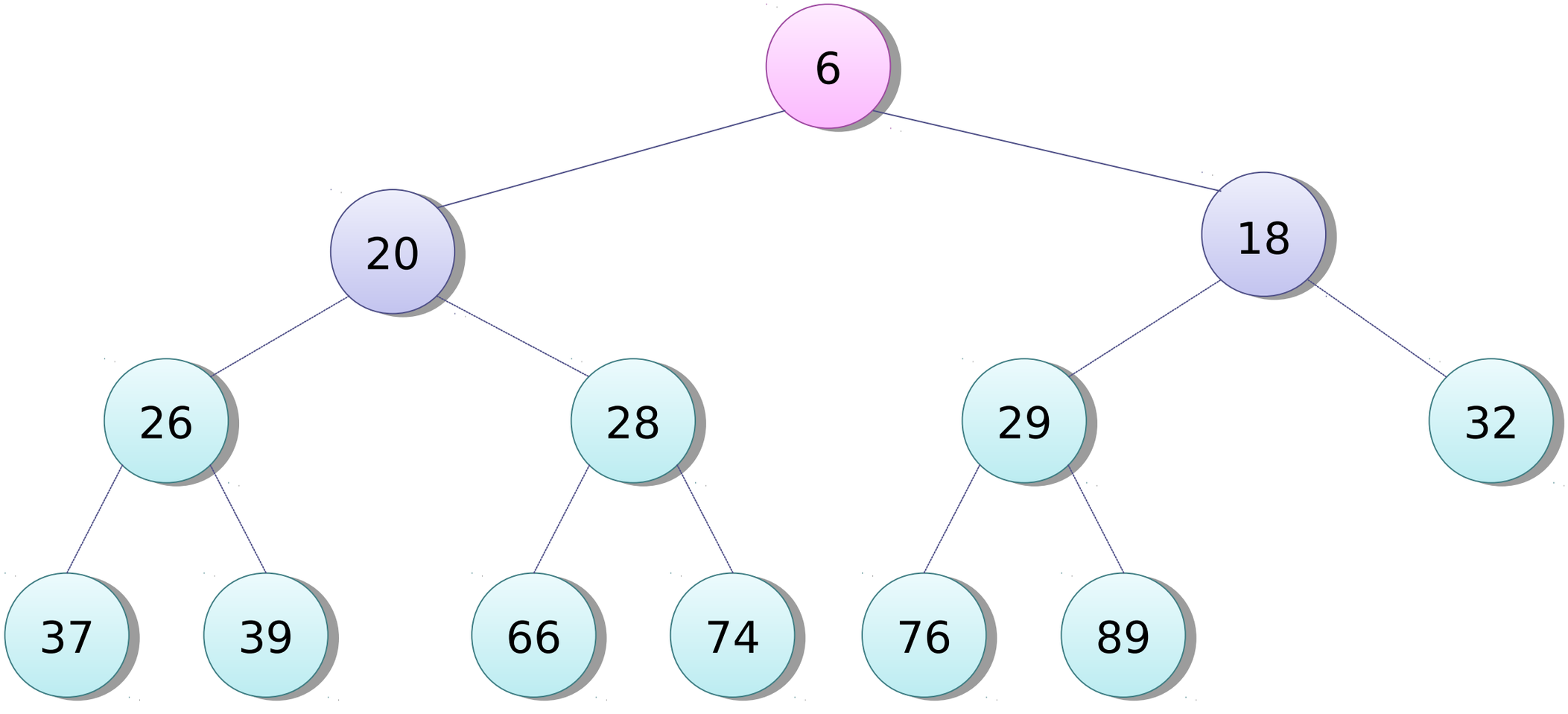
Trace of heapsort

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



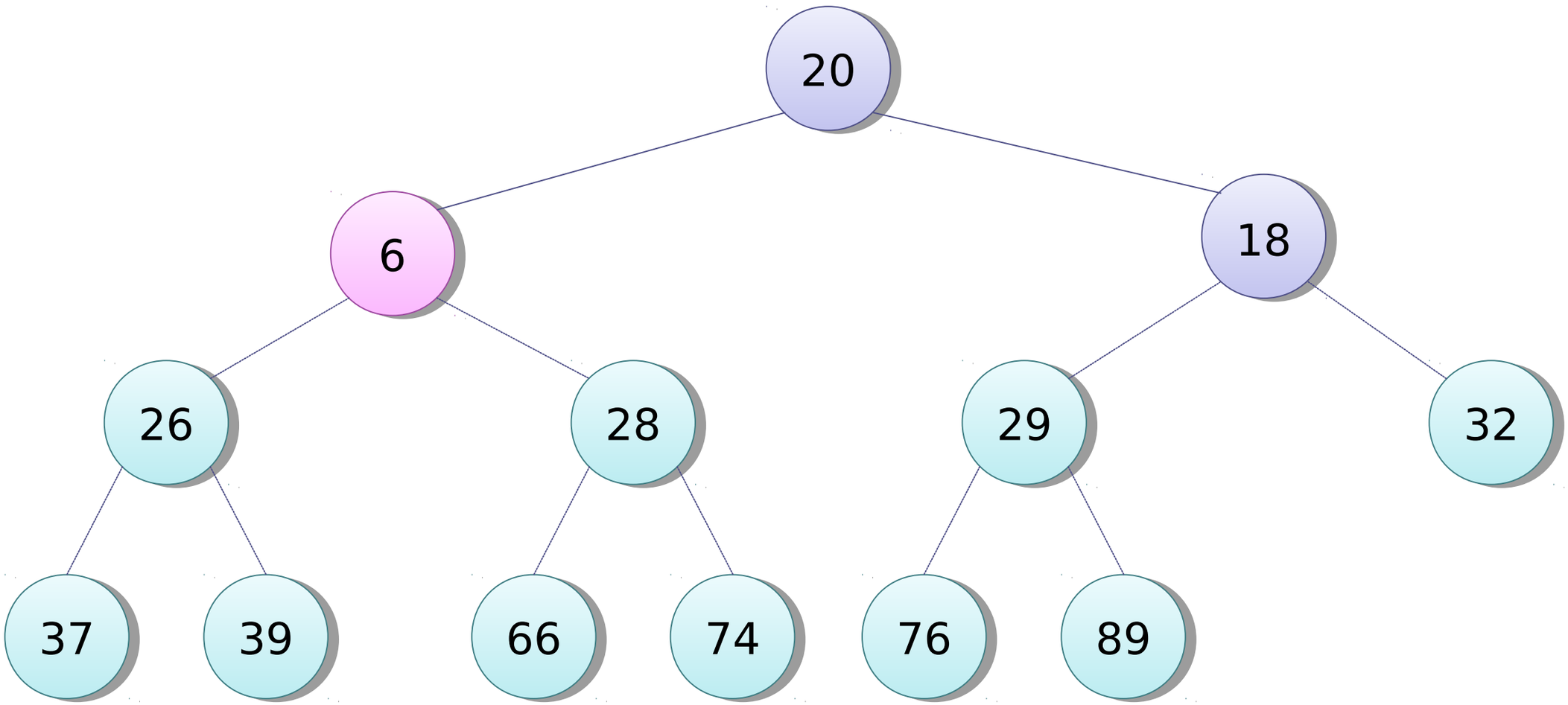
Trace of heapsort

Step 2: sift first element down

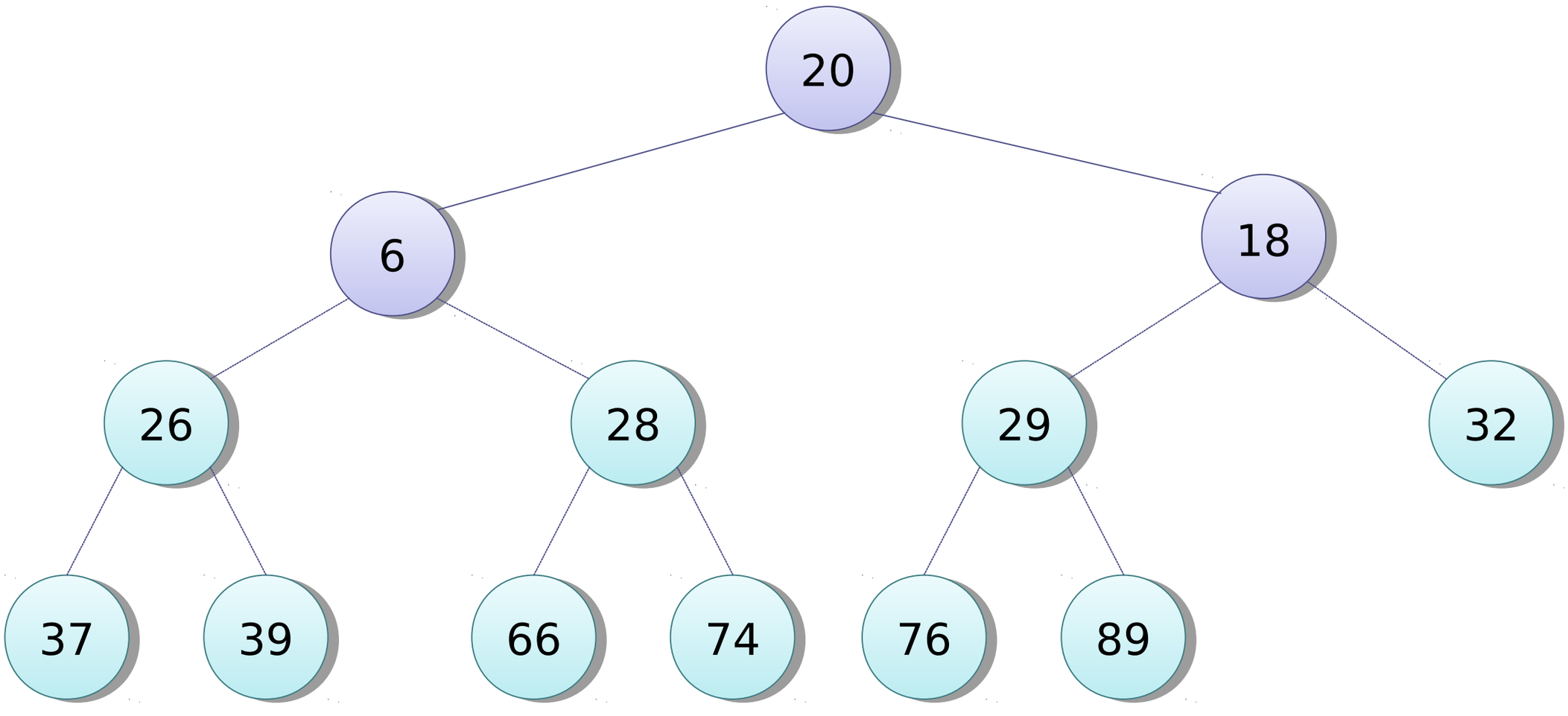


Trace of heapsort

Step 2: sift first element down

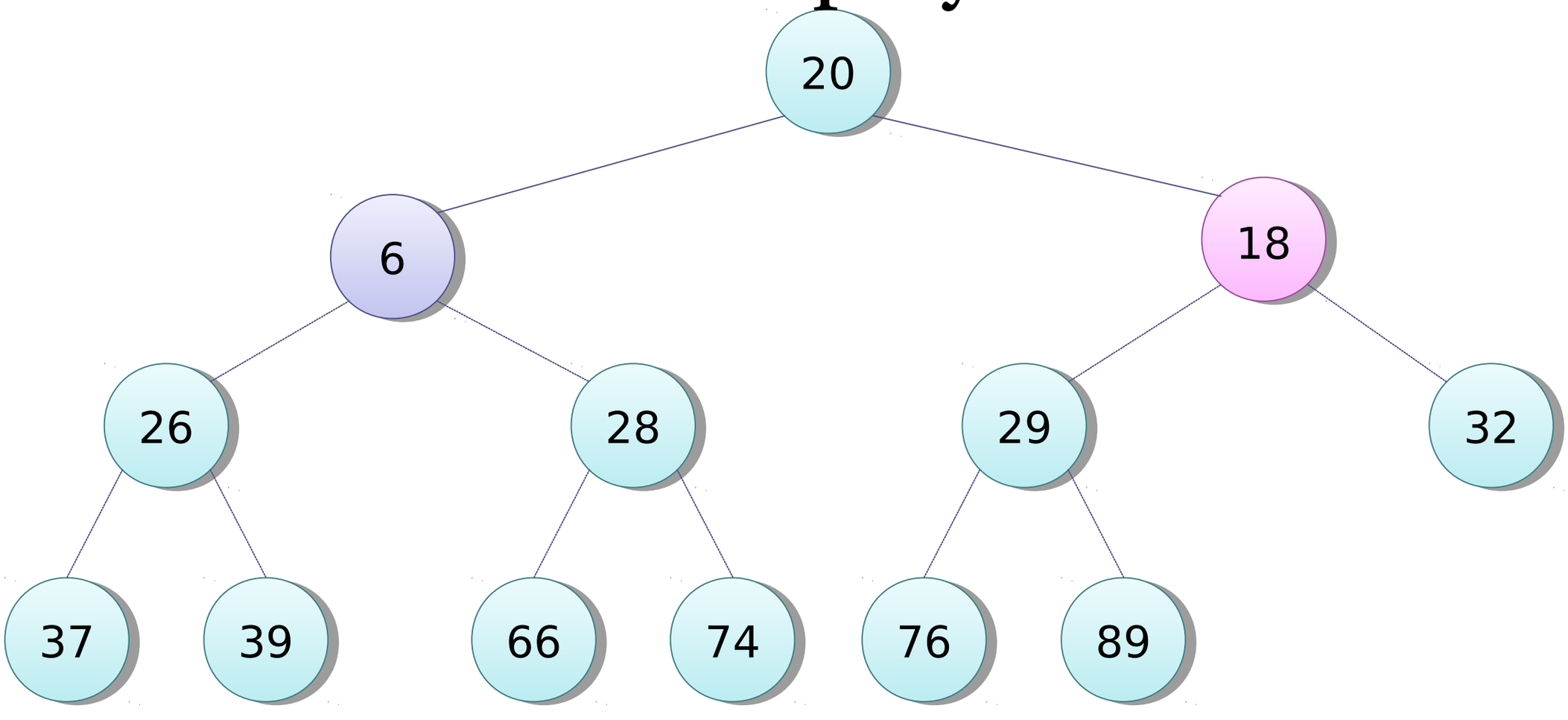


Trace of heapsort



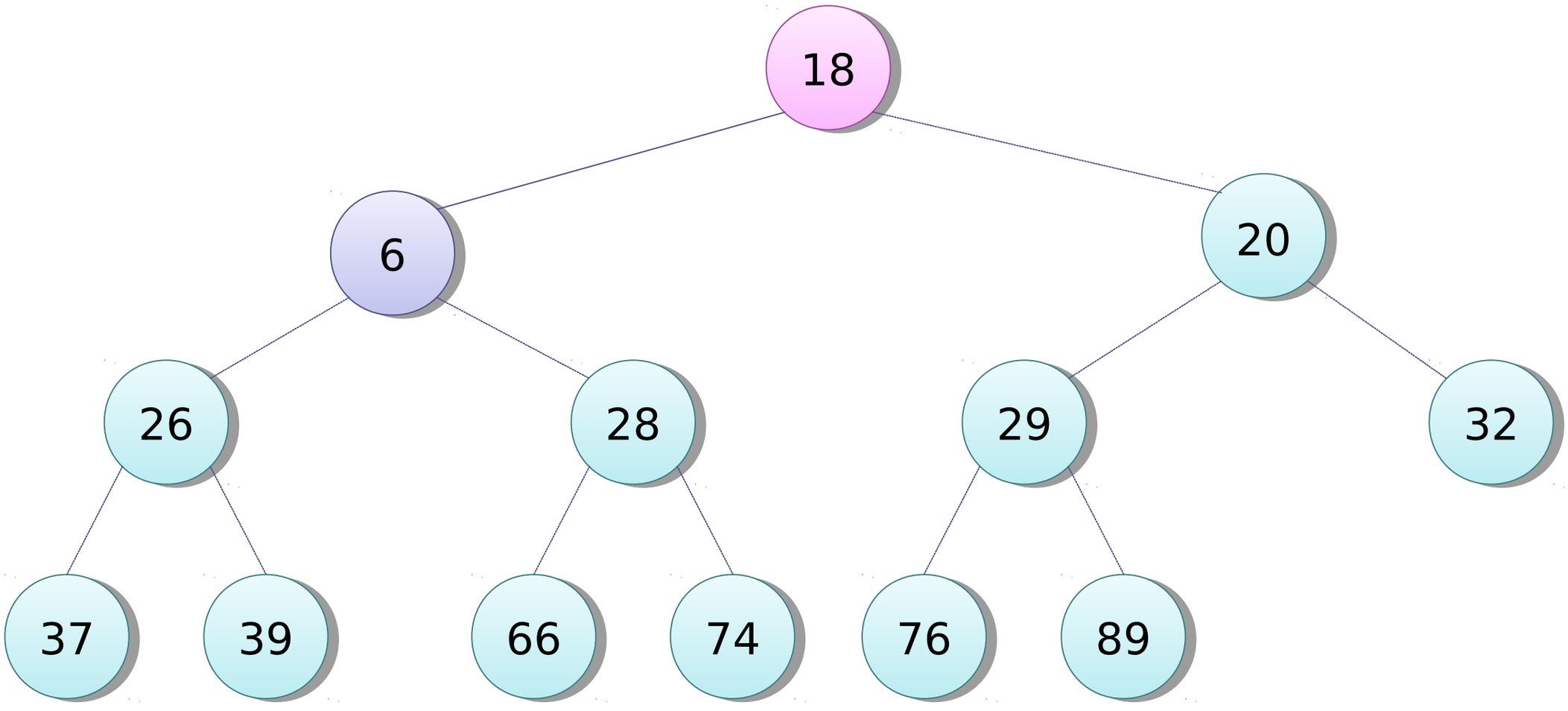
Trace of heapsort

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

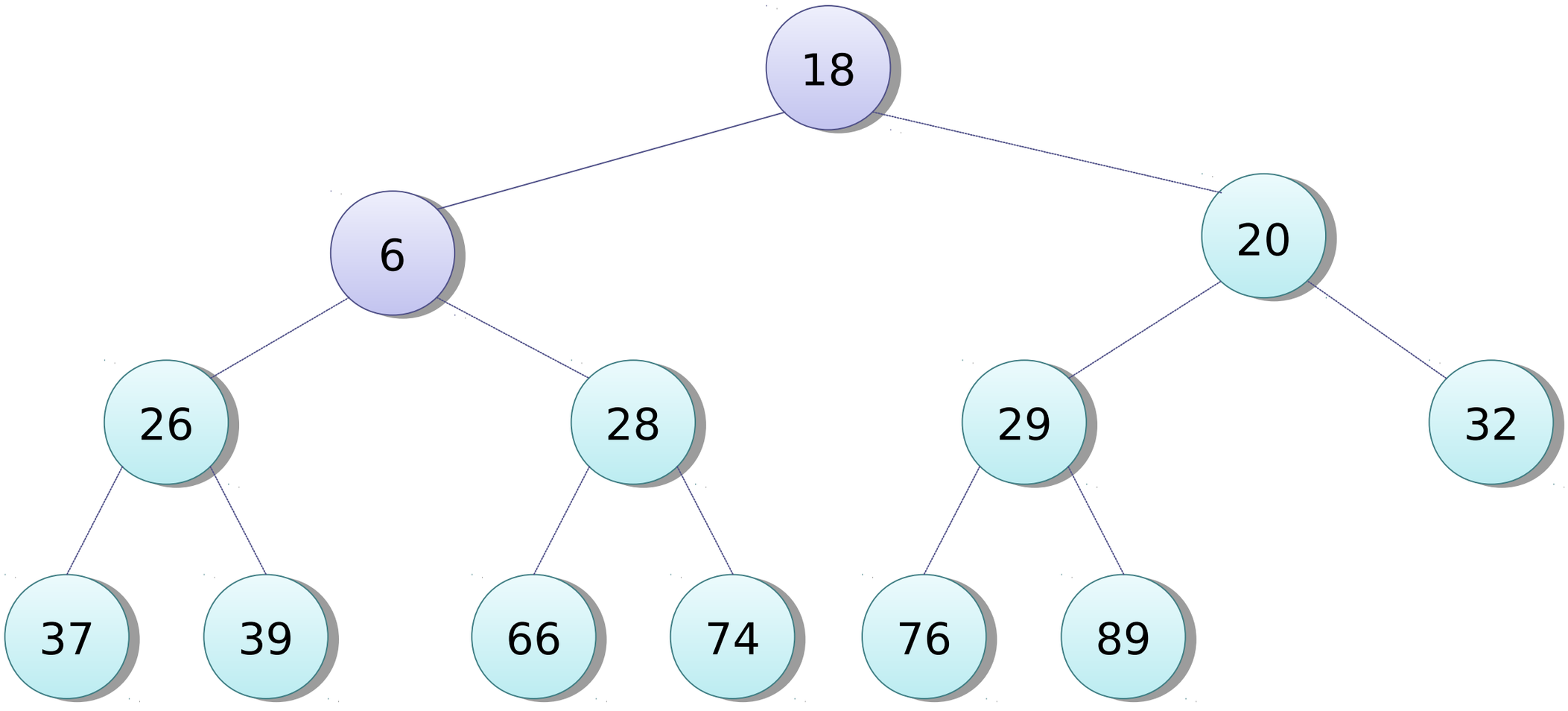


Trace of heapsort

Step 2: sift first element down

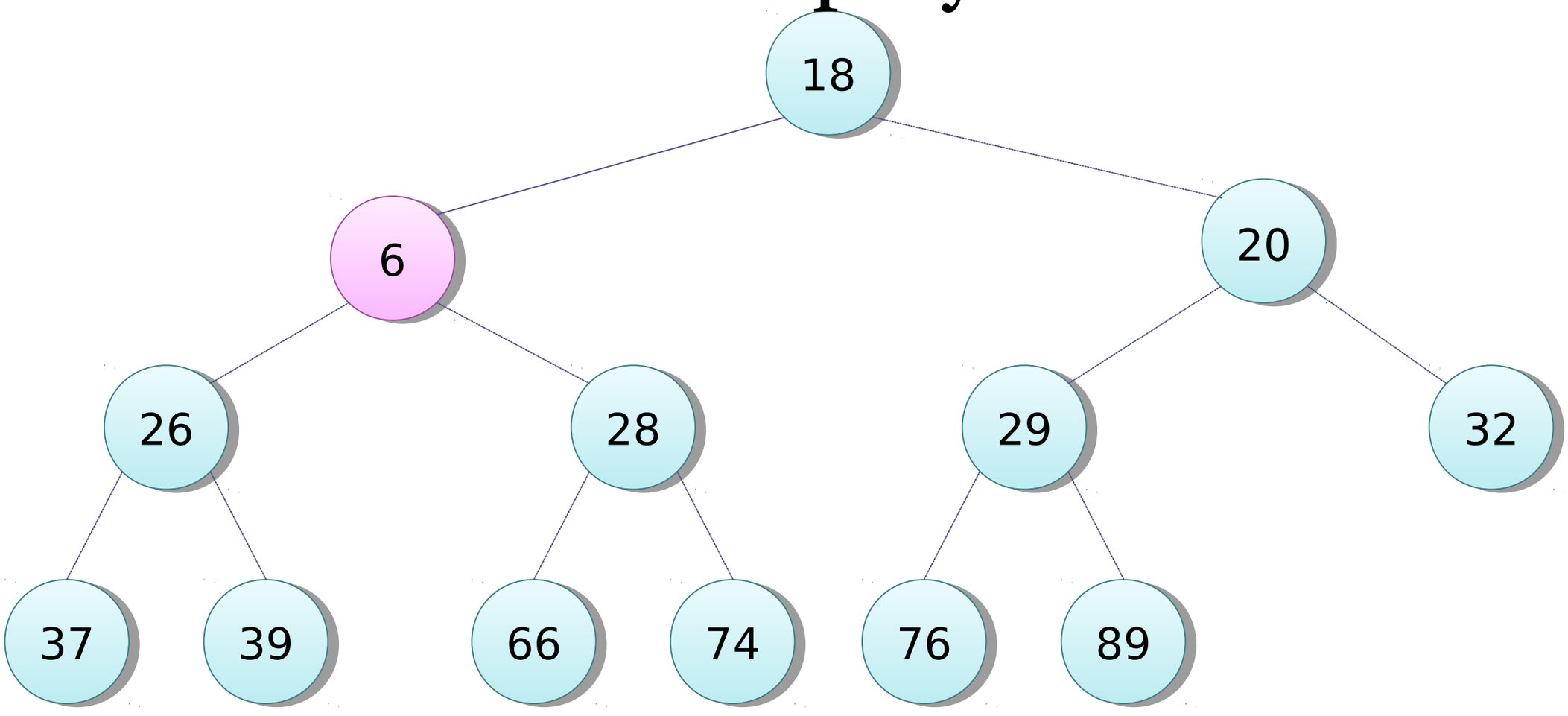


Trace of heapsort



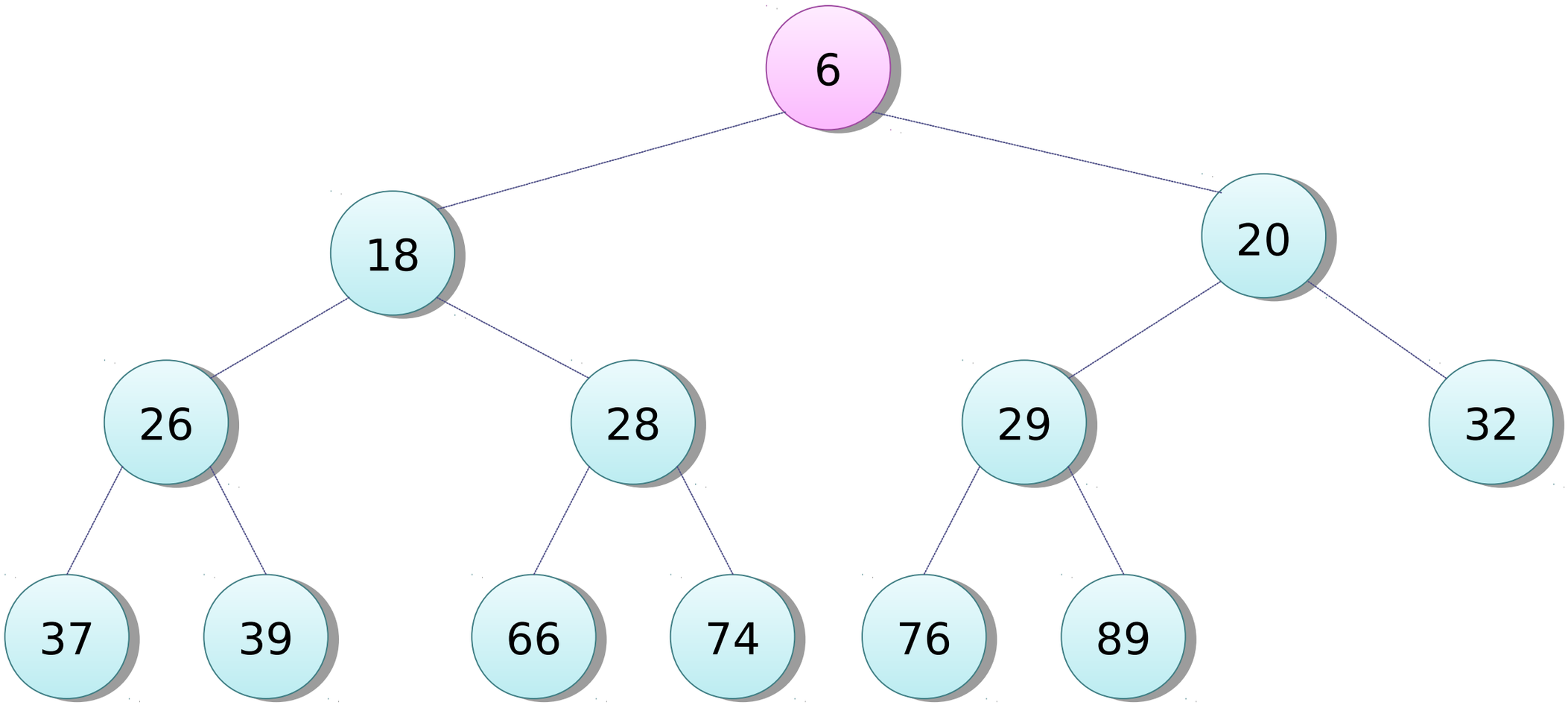
Trace of heapsort

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



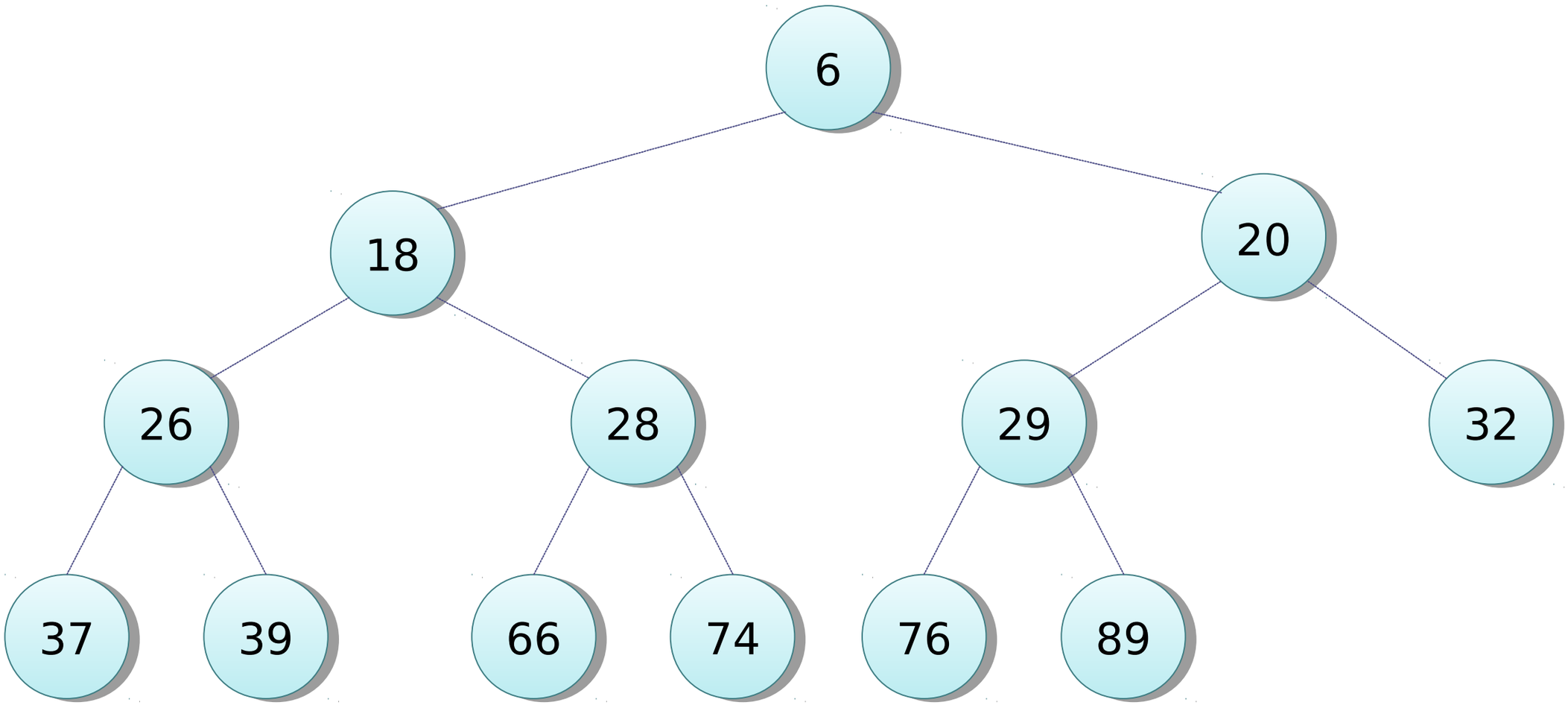
Trace of heapsort

Step 2: sift first element down



Trace of heapsort

Done!



Building a heap

The first element is already a one-element heap

- Sift element two up – now elements one and two form a valid heap
- Sift element three up – now the first three elements form a valid heap
- …and so on

Each sift is O(log n) – so total O(n log n)

In code:

```
for (int i = 1; i < n; i++)
  siftUp(i);
```

Building a heap

Better approach: instead of looping *forwards* through the array and sifting *up*, loop *backwards* and sift *down*:

```
for (int i = n / 2; i > 0; i--)
    siftDown(i-1);
```

Gives O(n) instead of O(n log n) complexity! (See book 21.3)

Complexity of heapsort

Building the heap: O(n)

Then does n deleteMins, each O(log n) complexity

Total: O(n log n)

Summary

Priority queues: insert, findMin, deleteMin

Binary heaps: O(log n) insert, O(1) findMin, O(log n) deleteMin

- A 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!