Announcement

Exercise sessions: one room instead of two

- Monday exercises: **EC**
- Thursday exercises: **ML1**

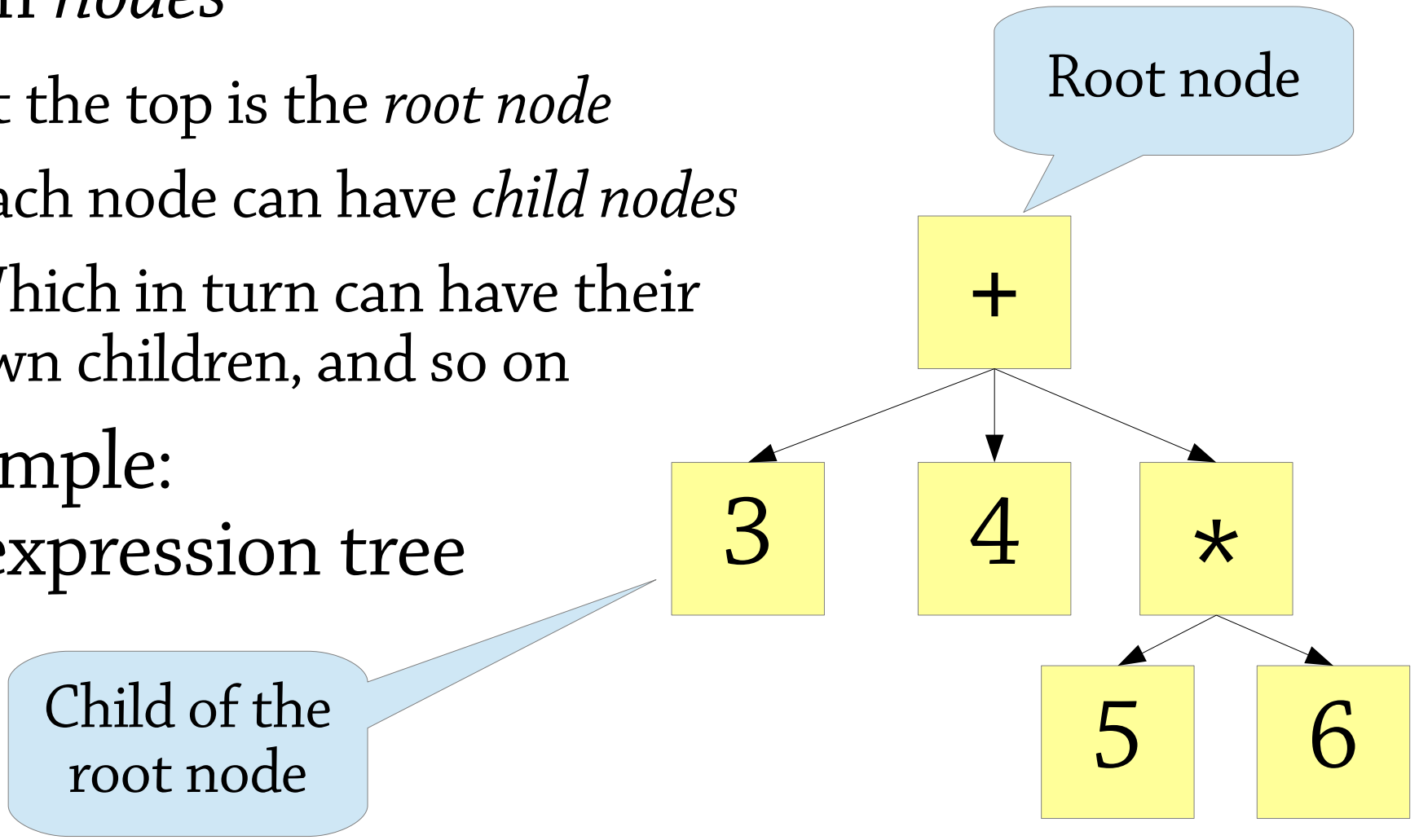
Trees
(Weiss 4.1-4.2)

Trees

A *tree* is a hierarchical data structure built up from *nodes*

- At the top is the *root node*
- Each node can have *child nodes*
- Which in turn can have their own children, and so on

Example:
an expression tree

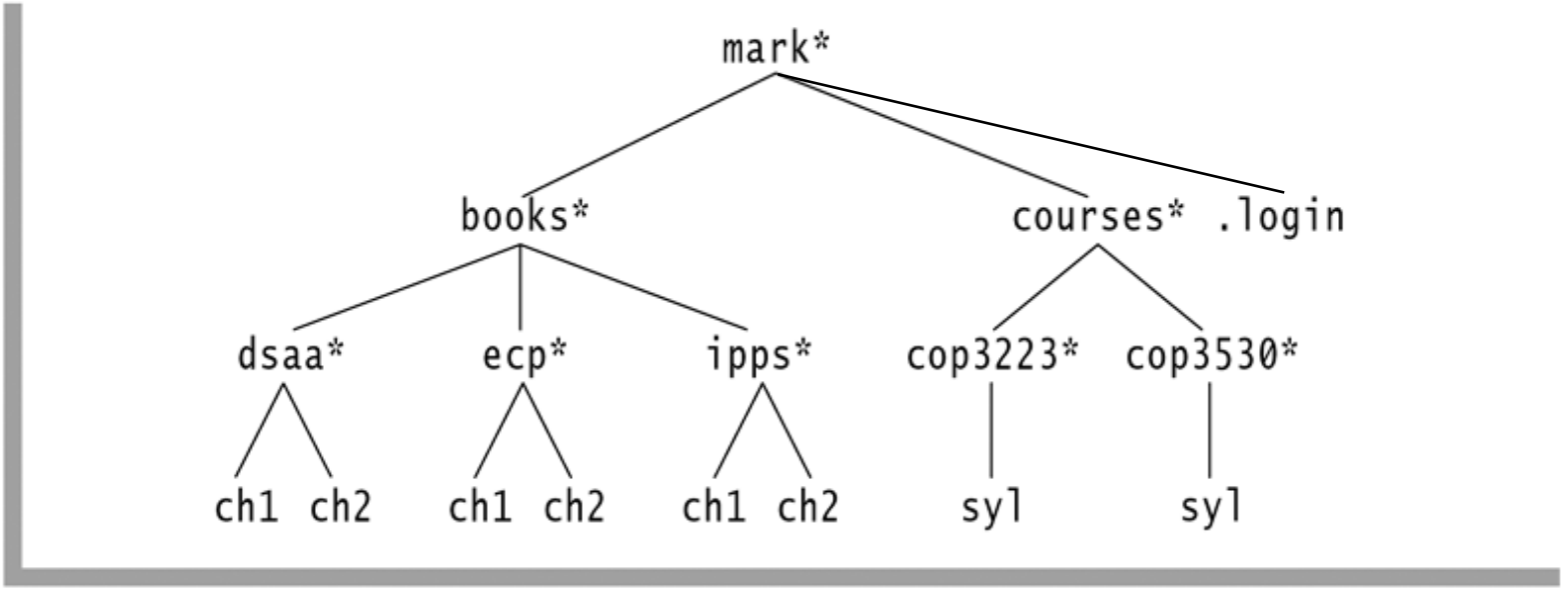


Trees

For a collection of nodes to be a tree:

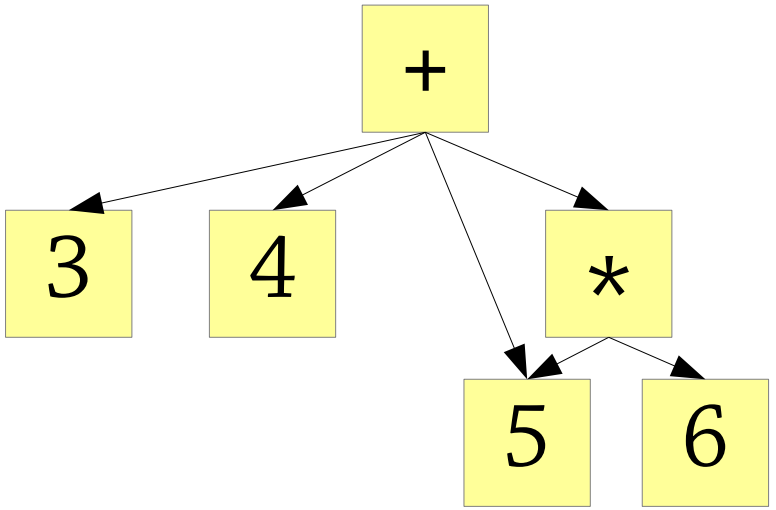
- There must be exactly one root node
- Two nodes cannot share a child
- Every node must be descended from the root

figure 18.4
A Unix directory

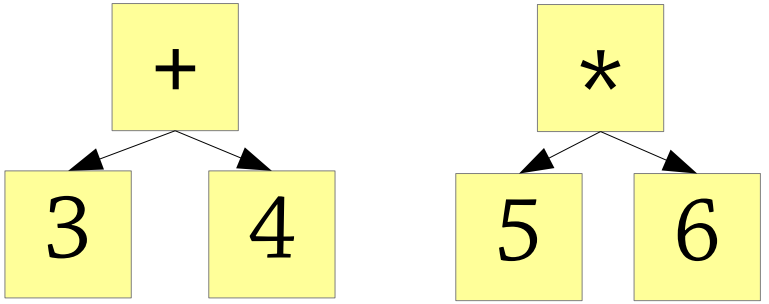


Not trees

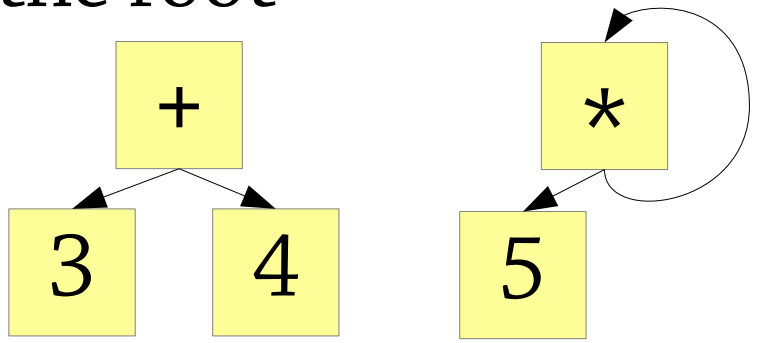
Not a tree: 5 is the child of two nodes



Not a tree: multiple roots



Not a tree: some nodes are not descended from the root

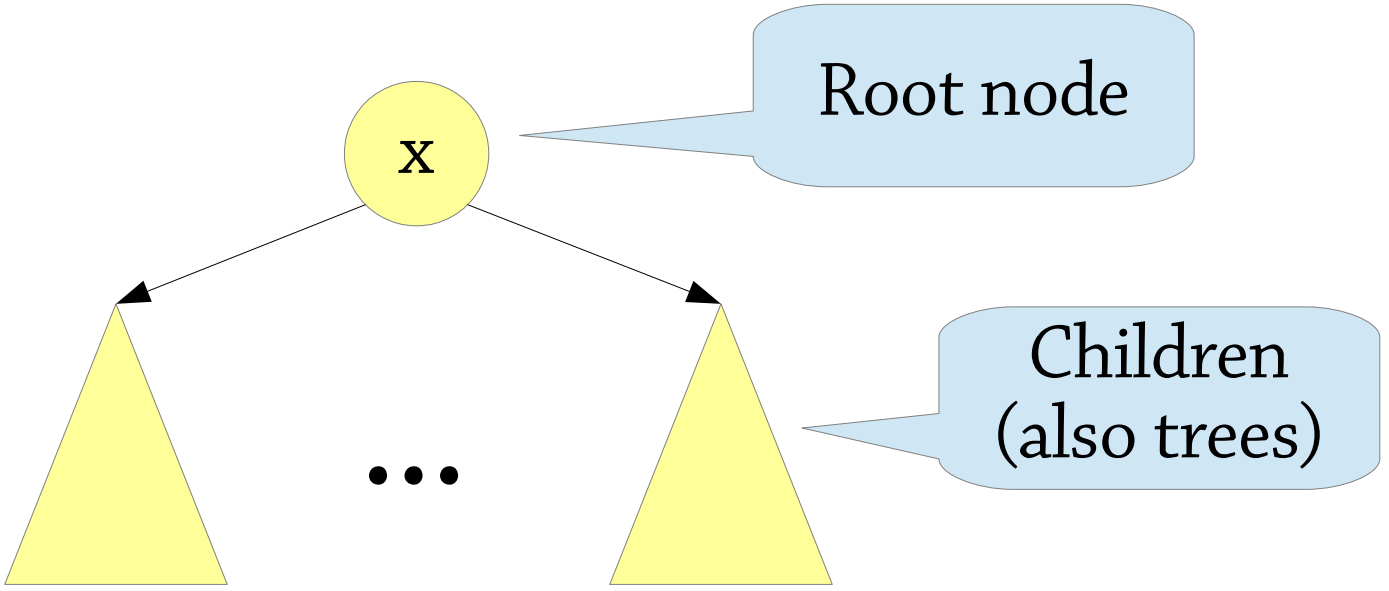


Trees recursively

Trees can be defined recursively:

- A tree can be empty
- A non-empty tree consists of a root node together with its children, which are trees themselves and must not share any nodes in common

This definition is useful for programming



Binary trees

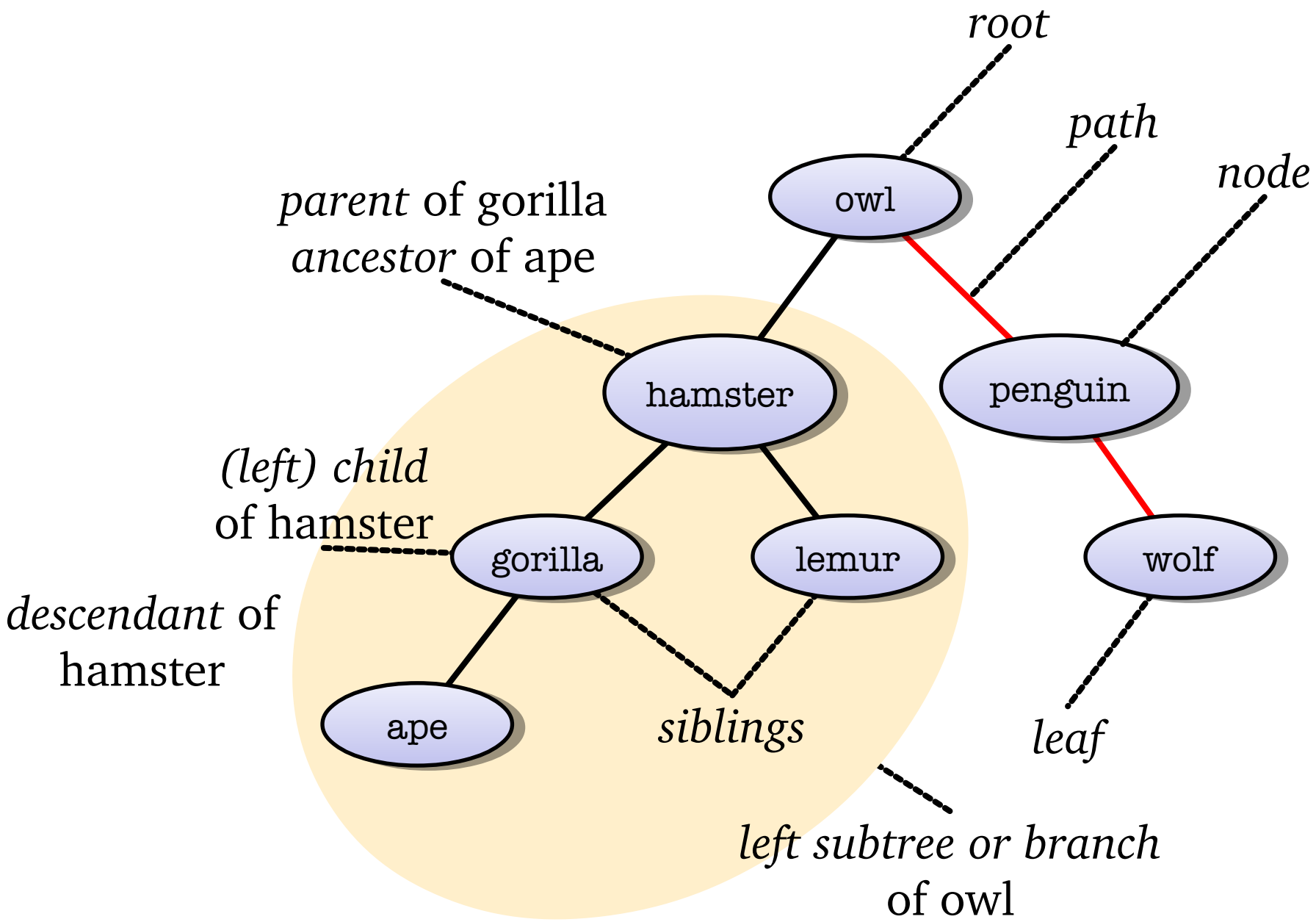
Very often we use *binary trees*, where each node has two children, called the *left* and *right* child

```
class Node<E> {
  E value;
  Node<E> left, right;
  (optionally) Node<E> parent;
}
class Tree<E> {
  E root;
}
```

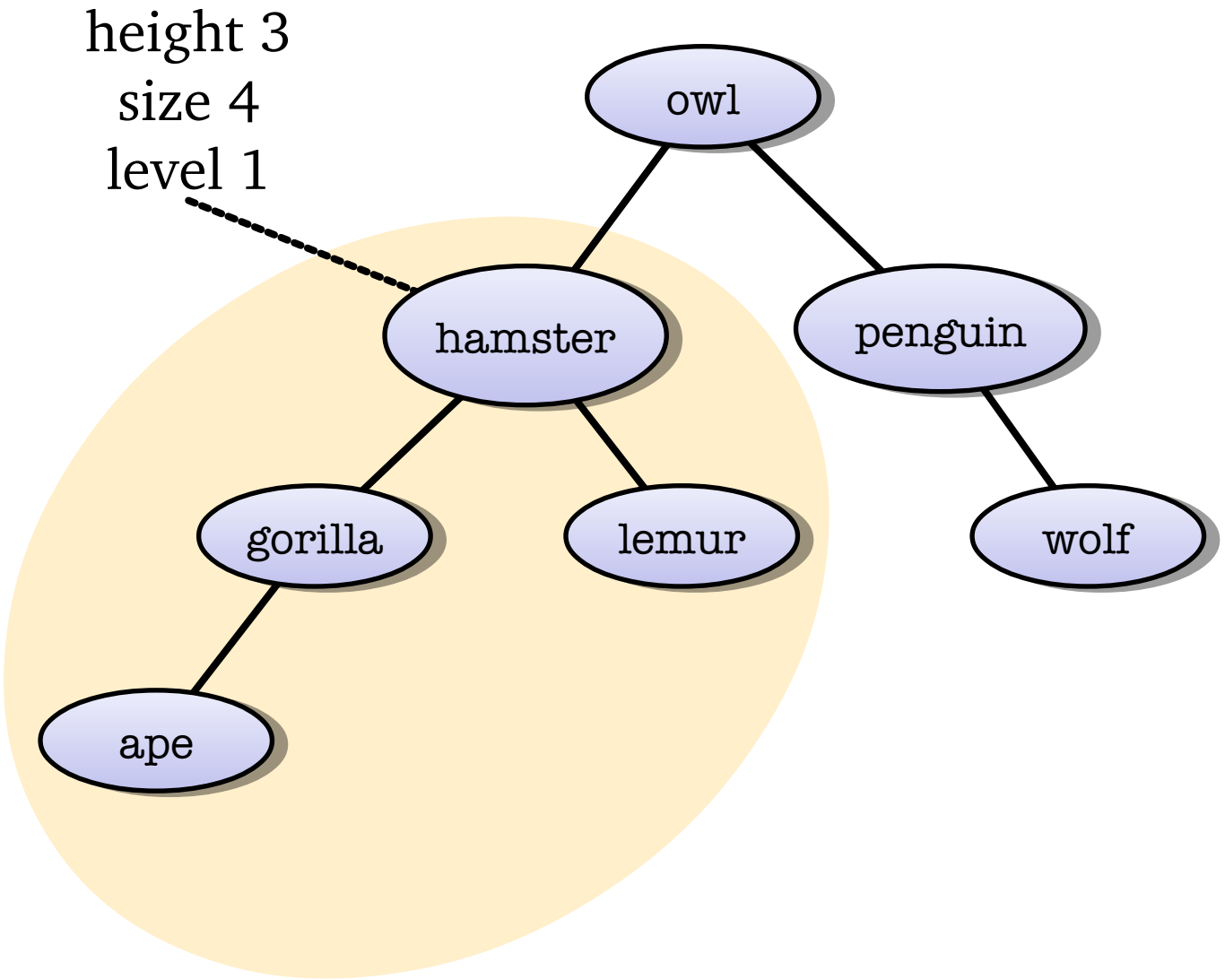
Can be `null`

```
-- If you know functional programming:
data Tree a
  = Node a (Tree a) (Tree a)
  | Nil
```

Terminology

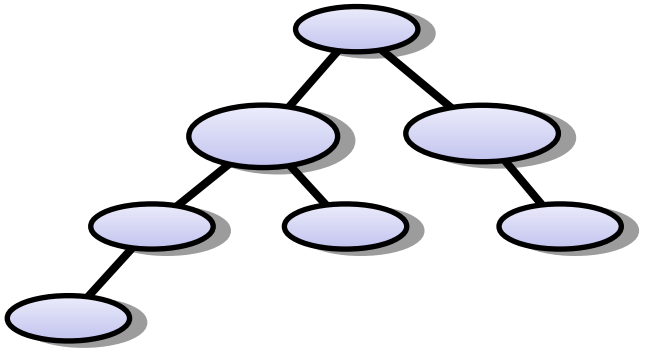


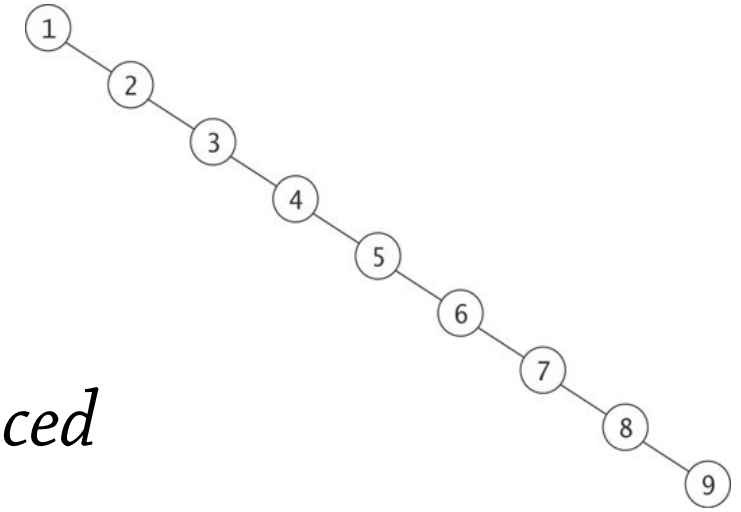
height = number of levels in tree
size = number of nodes in tree
level = distance to root



Balanced trees

A binary tree of size n has a height of between $log_2\ n$ and n:





A tree of size n is called *balanced*
if its height is O(log n)

Many tree algorithms have complexity O(height of tree), so are efficient on balanced trees and less so on unbalanced trees

Normally: balanced trees good, unbalanced bad!

Priority queues
(Weiss 6.1-6.4, 6.9)

Priority queues

A *priority queue* has three operations:

- *insert*: add a new element
- *find minimum*: return the smallest element
- *delete minimum*: remove the smallest element

Similar idea to a stack or a queue, but:

- you get the *smallest* element out

Alternatively, you give each element a *priority* when you insert it; you get out the smallest-priority element

Applications

A printer queue where certain jobs have higher priority

- the boss's documents get highest priority
- after that, shorter documents are printed first
- if two documents have the same priority, the oldest one is printed first
- or some combination of age and size?

Just need to be able to compute a priority when adding the job!

Similarly, the scheduler in an operating system – decide which process to run next

Applications

Sorting a list:

- Start with an empty priority queue
- Add each element of the input list in turn
- Repeatedly find and remove the smallest element
- You get all elements out in ascending order!

If all priority queue operations are O(log n), this sorting algorithm takes O(n log n) time

Applications

A simulation – models *events* happening at a particular *time*

- "At 10:00:03 the customer entered the shop"
- "At 10:08:05 the customer got to the checkout"

When an event happens, it can cause more events to happen in the future

- "When a customer enters the shop, 1 minute later they pick up some bread"
- "When the cashier finishes scanning the items, 30 seconds later the customer finishes paying"

Applications – simulation

Keep a priority queue of *future events*

- "At 10:00:03 a person will enter the shop"

Simulator's job: remove *earliest* event and run it, then repeat

- In the priority queue, earlier events will be counted as "smaller" than later events

When we run that event, it can in turn add more events to the priority queue

- When a customer enters the shop, add an event "the customer picks up some bread" to the priority queue at a time of 1 minute later

This lecture

1.

How to make an efficient priority queue

2.

How to design data structures

An inefficient priority queue

Idea 1: implement a priority queue as a *dynamic array*

- Insert: add new element to end of array
  O(1)
- Find minimum: linear search through array
  O(n)
- Delete minimum: remove minimum element
  O(n)

Finding the minimum is quite expensive though.

An inefficient priority queue

Idea 2: use a *sorted array*

- Insert: insert new element in right place
  O(n)
- Find minimum: minimum is first element
  O(1)
- Delete minimum: remove first element
  O(n)

Finding the minimum is cheap! Yay!
But... insertion got expensive :(

Invariants

By making the array sorted...

- Finding the minimum got easier
- But insertion got harder

"The array is sorted" is an example of a data structure *invariant*

- A property picked by the data structure designer, that always holds
- *Insert*, *find minimum and delete minimum* can assume that the invariant holds (the array is sorted)
- ...but they must make sure it remains sorted afterwards (*preserve the invariant*)

More on invariants

Choosing the right invariant is *the most important step* in data structure design!

A good invariant adds some *extra structure* that:

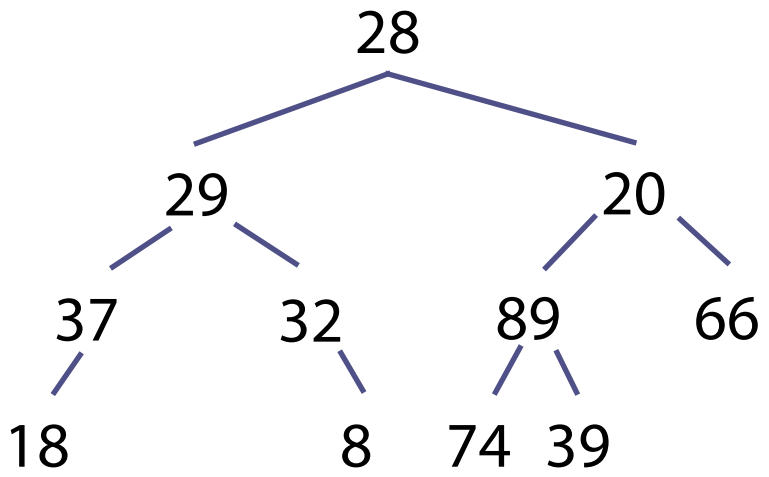
- makes it easy to *get* at the data
  (the invariant is useful)
- without making it hard to *update* the data
  (it's not too hard to preserve the invariant)

Finding the right invariant takes a lot of practice!

Binary heaps: priority queues implemented using trees

Heaps – representation

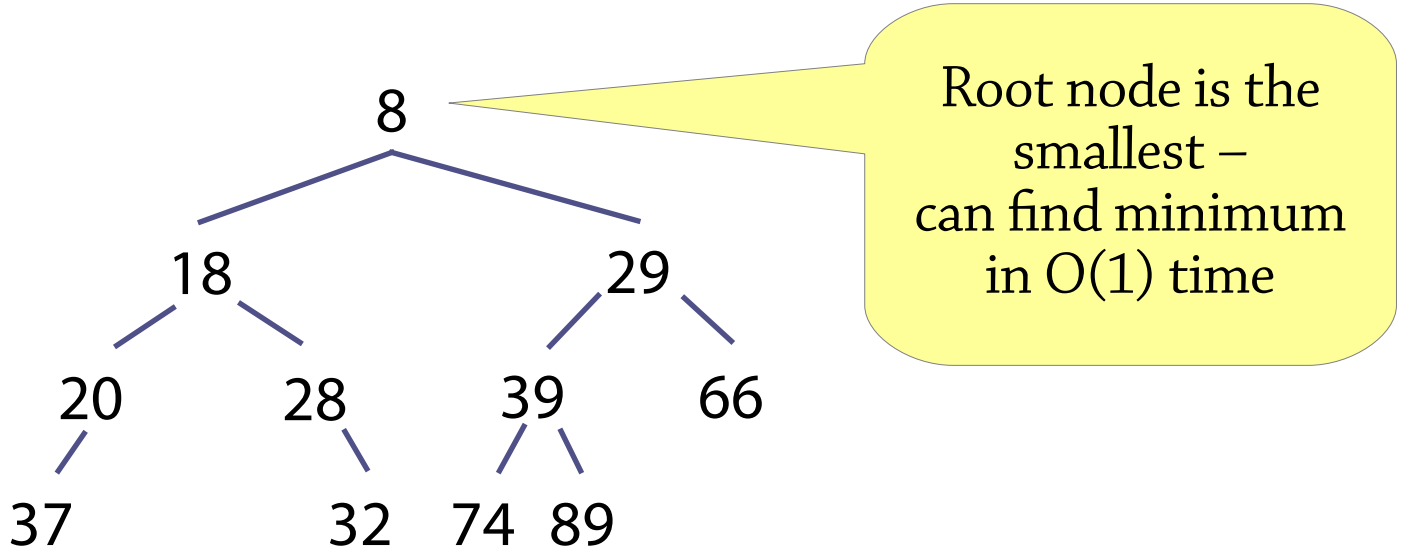
A heap implements a priority queue as a tree. Here is a tree:



This is not yet a heap. We need to add an invariant that makes it easy to find the minimum element.

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:



Where can we find the smallest element?

Why the heap property

Why did we pick this invariant? One reason:

- It puts the smallest element at the root of the tree, so we can find it in O(1) time

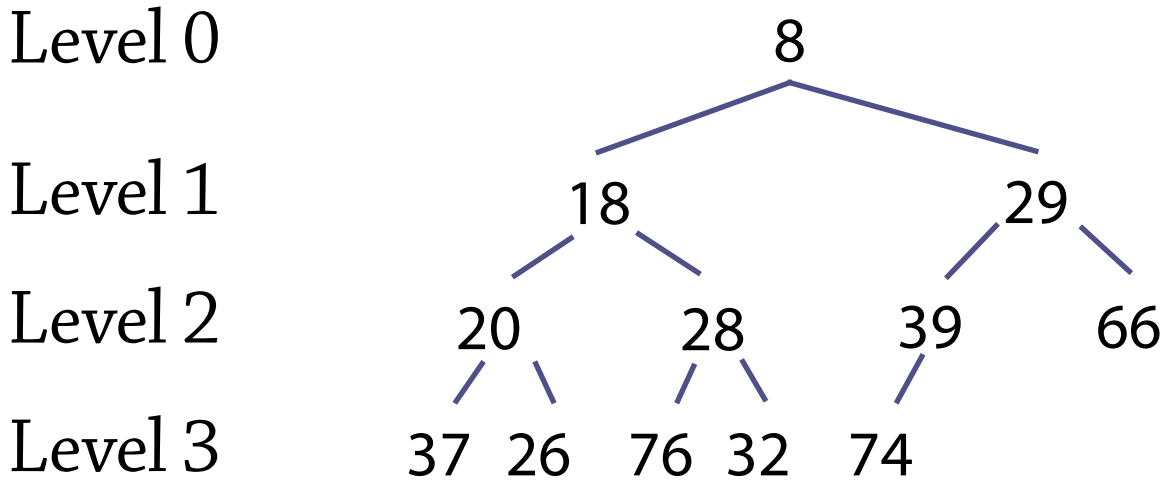
Why not just have the invariant "the root node is the smallest"? Because:

- Trees are a *recursive* structure – the children of a node are also trees
- It's then a good rule of thumb to have a recursive invariant – each node of the tree should satisfy the same sort of property
- In this case, instead of "the root node is smaller than its descendants", we pick "each node is smaller than its descendants"

General hint: when using a tree data structure, make each node have the same invariant

Binary heap

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



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

Why completeness?

There are a couple of reasons why we choose to have a complete tree:

- It makes sure the tree is balanced
- When we insert a new element, it means there is only one place the element can go – this is one less design decision we have to make

There's a third one which we will see a bit later!

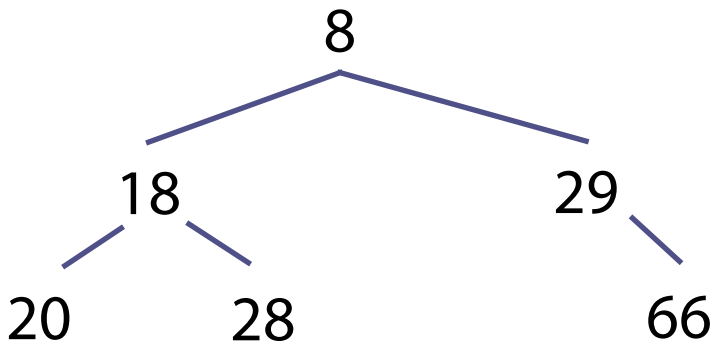
Binary heap invariant

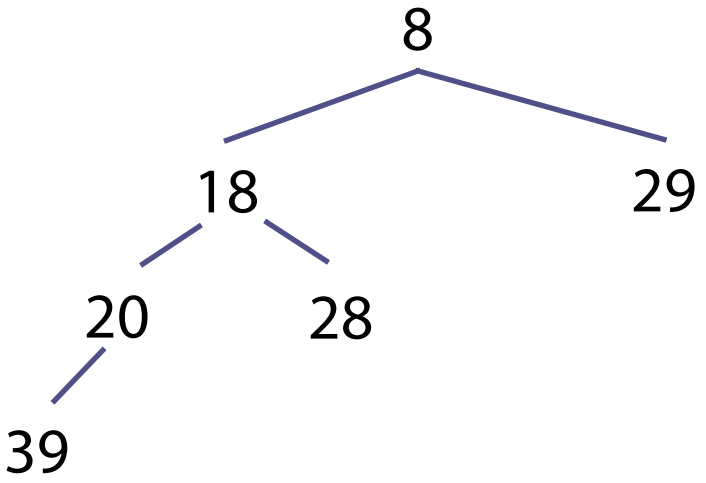
The binary heap invariant:

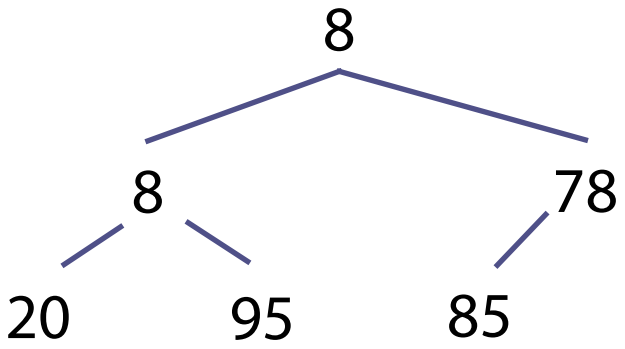
- The tree must be *complete*
- It must have the *heap property* (each node is less than or equal to its children)

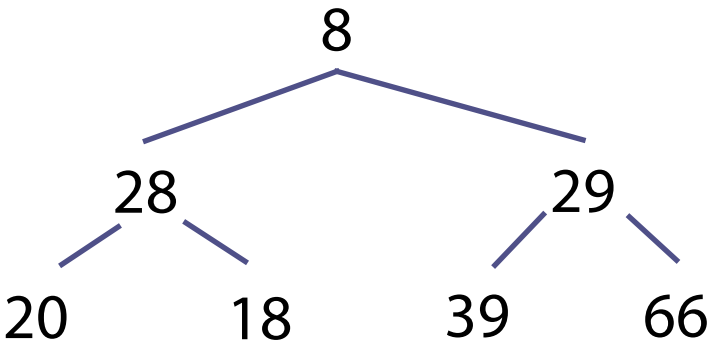
Remember, all our operations must preserve this invariant

Binary heap or not?

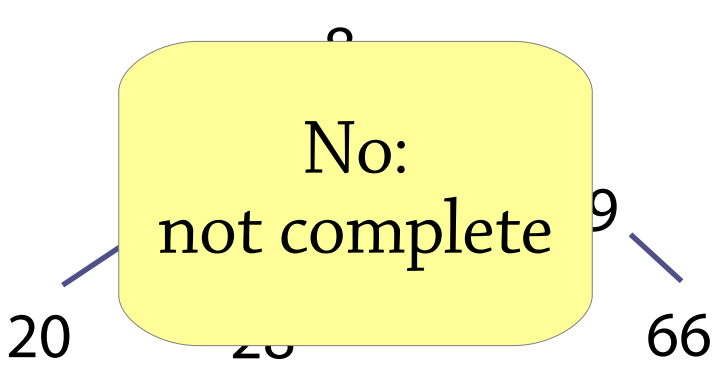


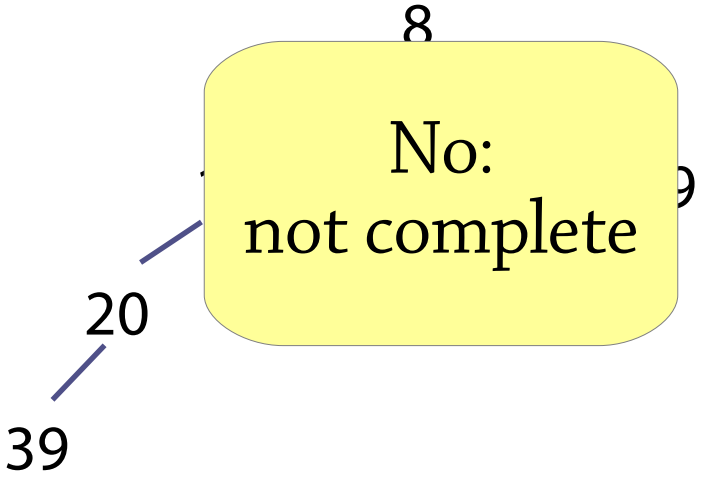


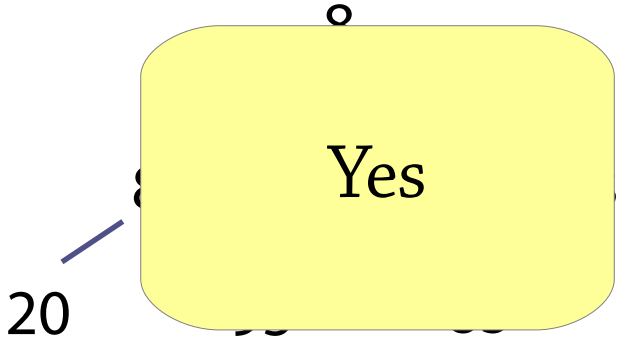


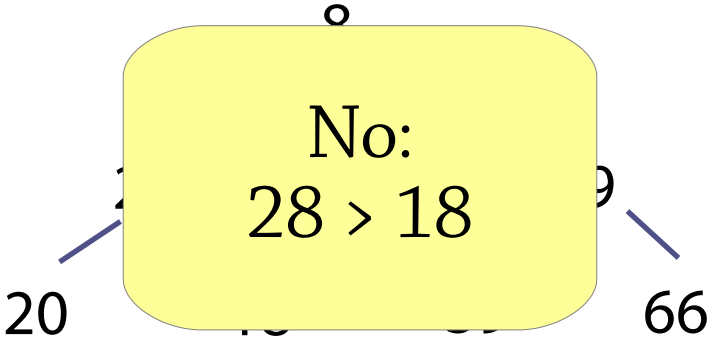


Binary heap or not?



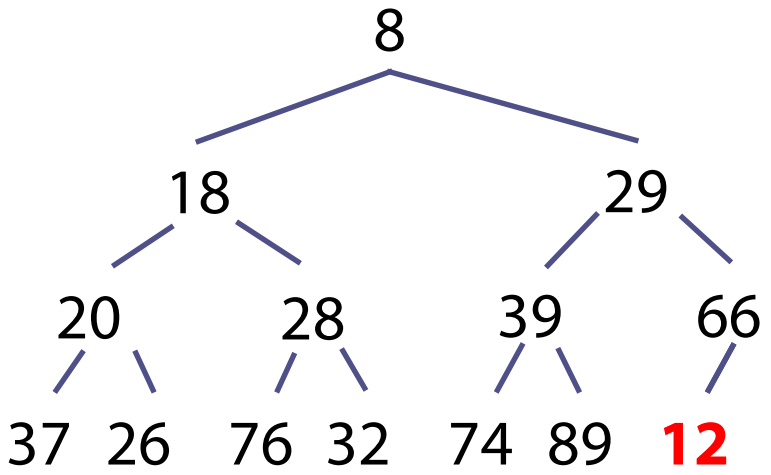






Adding an element to a binary heap

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.

An aside

To modify a data structure with an invariant, we have to

- modify it,
- while preserving the invariant

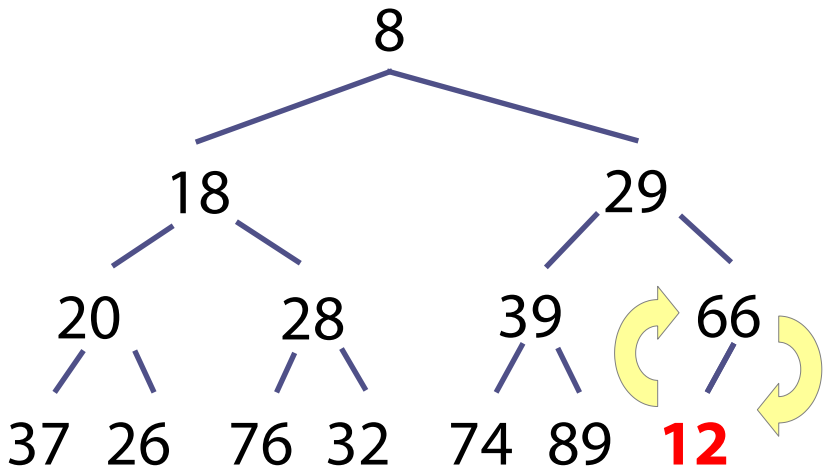
Often it's easier to separate these:

- first modify the data structure, possibly breaking the invariant in the process
- then “repair” the data structure, making the invariant true again

This is what we are going to do here

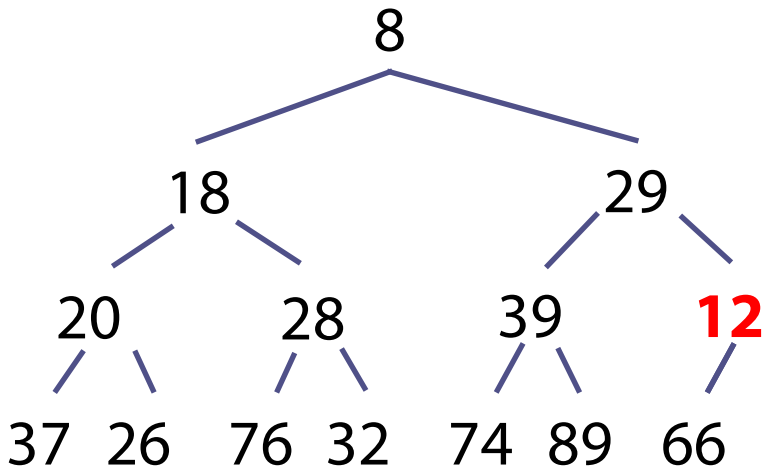
Adding an element to a binary heap

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



Adding an element to a binary heap

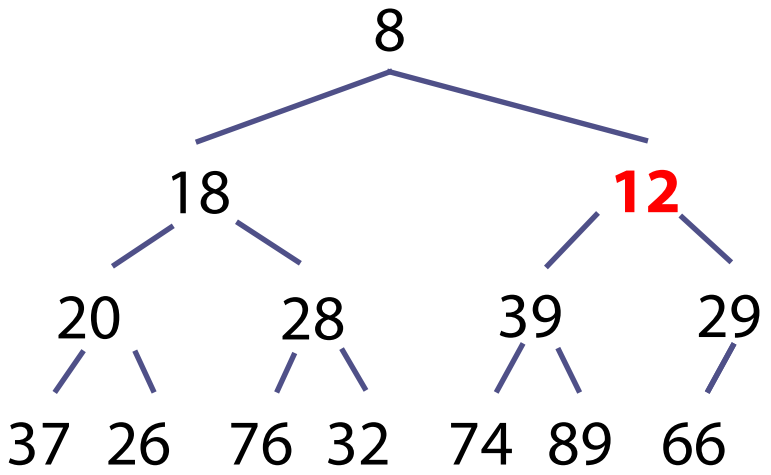
Step 2: if the new element is less than its parent, swap it with its parent



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.

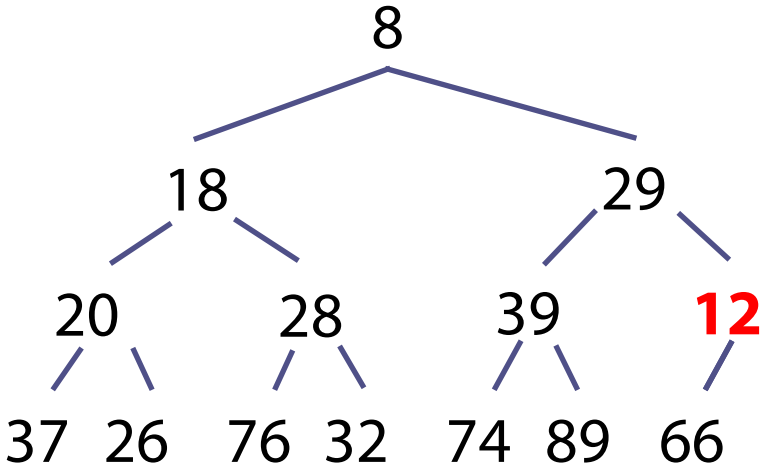


Now 12 is in its right place, and the invariant is restored. (Think about why this algorithm restores the invariant.)

Why this works

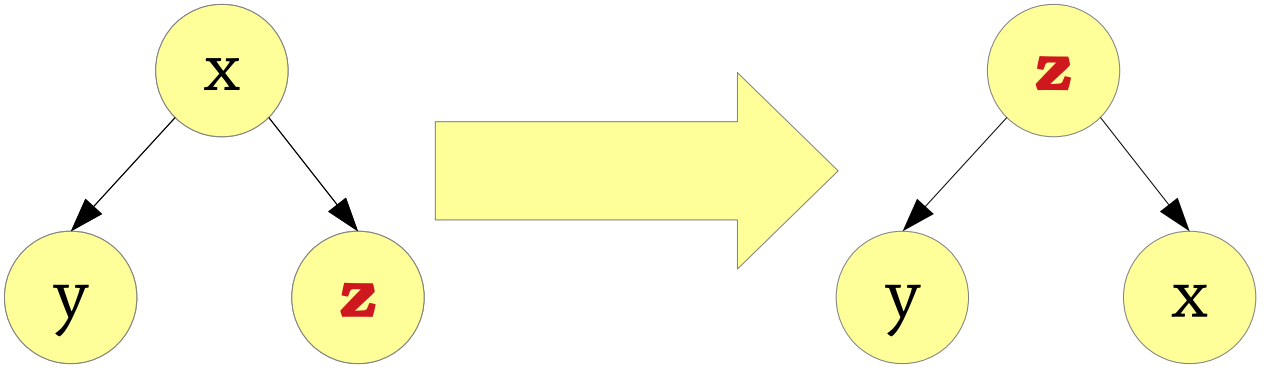
At every step, the heap property almost holds *except* that the new element might be less than its parent

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



Why this works

Suppose that *z* is the new node and we swap it with its parent:



How do we know that the invariant for y isn't broken in the right-hand diagram?

From the left-hand diagram, we must have $z < x$ (otherwise we wouldn't do the swap) and $x \leq y$ (because the invariant is only broken for *z* at this point) – therefore $z < y$.

Removing the minimum element

To remove the minimum element, we are going to follow a similar scheme as for insertion:

- First remove the minimum (root) element from the tree somehow, breaking the invariant in the process
- Then repair the invariant

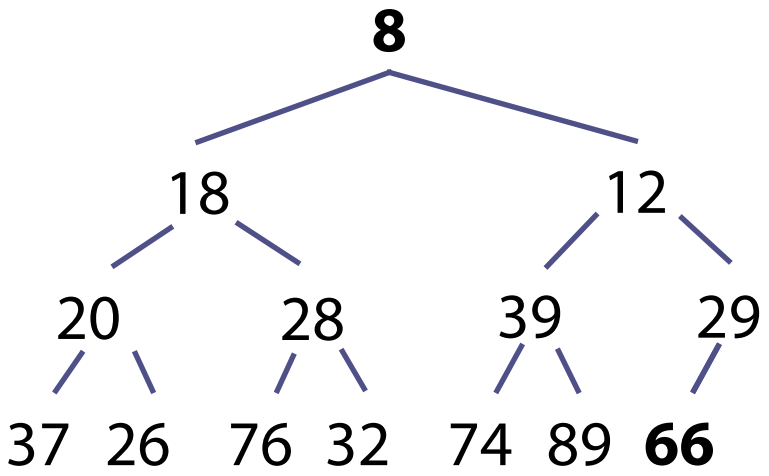
Because of *completeness*, we can only really remove the *last* (bottom-right) element from the tree

- Solution: first *swap* the root element with the last element, then remove the last element

Removing the minimum element

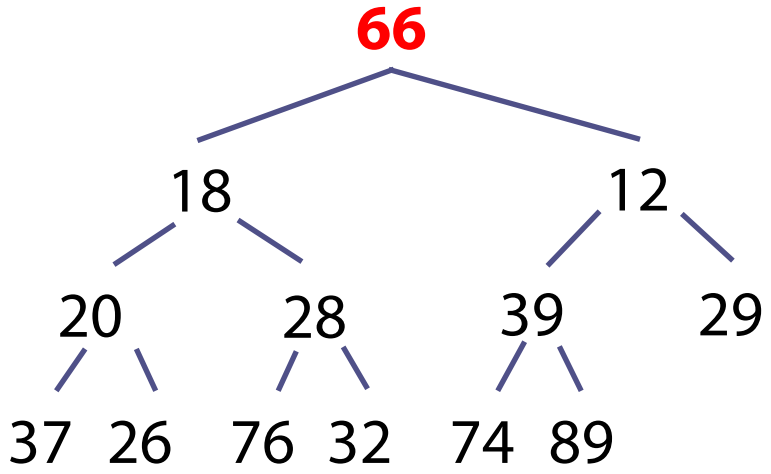
The goal: remove the minimum element

First: swap the root and the last element (66), and then remove the last element



Removing the minimum element

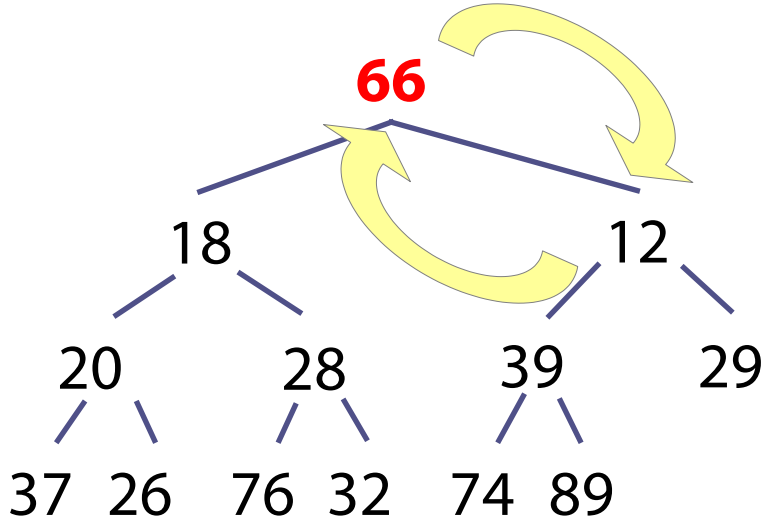
Step 1: swap the root element and the *last element* in the tree, and remove the last element



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

Removing the minimum element

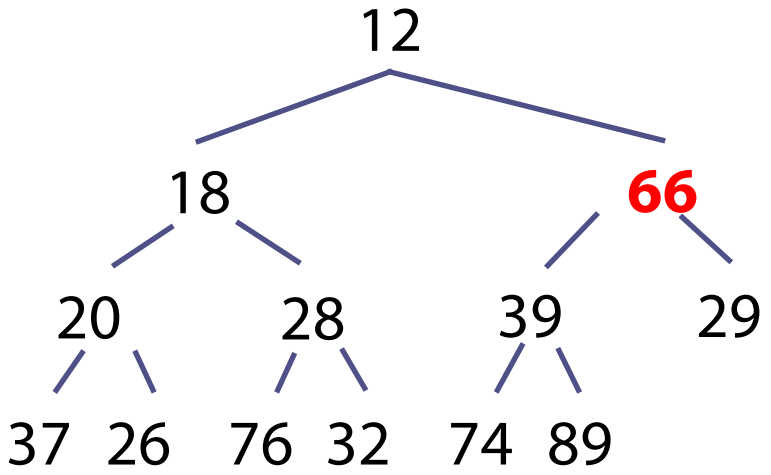
Step 2: if the moved element is greater than its children, swap it with its *least child*



(Why the least child in particular?)

Removing the minimum element

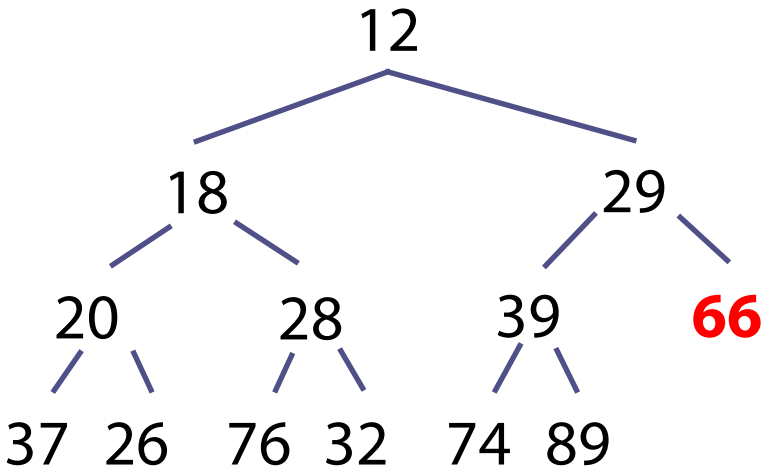
Step 2: if the moved element is greater than its children, swap it with its *least child*



(Why the least child in particular?)

Removing the minimum element

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



Sifting

Two useful operations we can extract from all this

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

- Repeatedly swap the element with its parent

Sift down: if an element might be greater than its children, i.e. needs "moving down" (used in removing the minimum element)

- Repeatedly swap the element with its least child

Binary heaps – summary so far

Implementation of priority queues

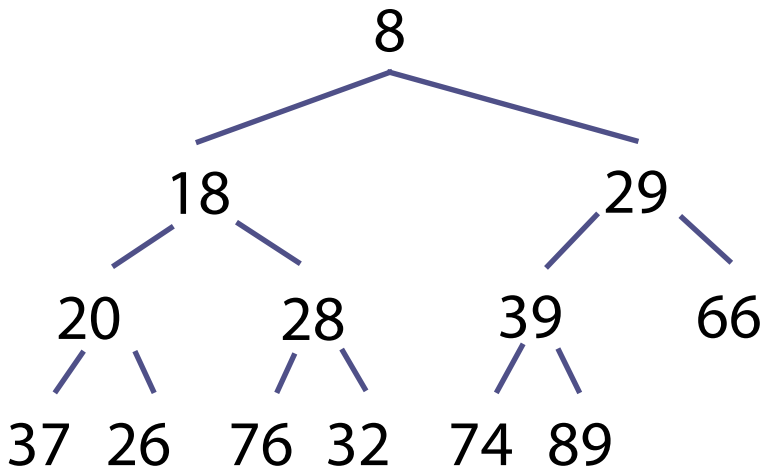
- *Heap property* – means smallest value is always at root
- *Completeness* – means tree is always balanced

Complexity:

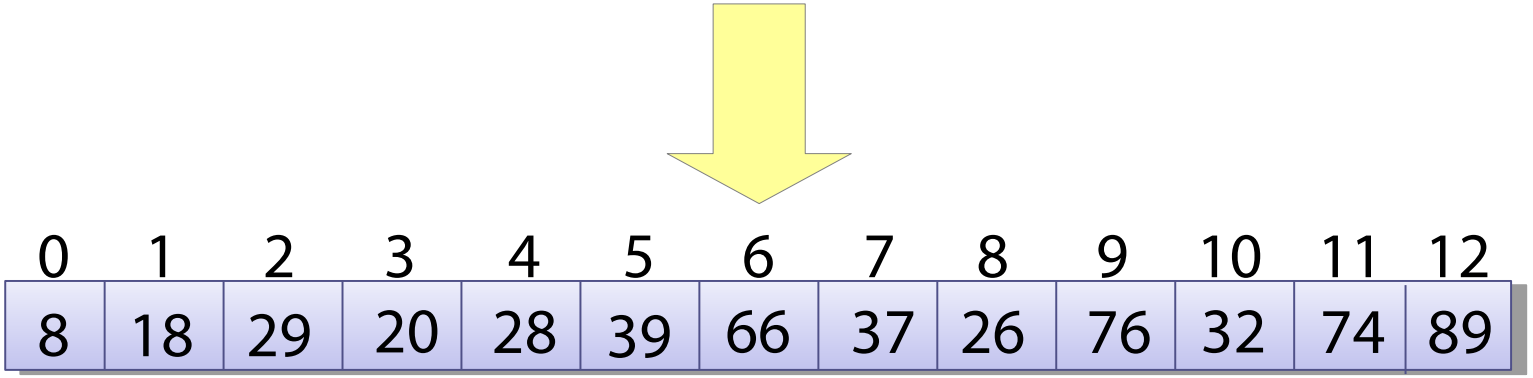
- *find minimum* – **O(1)**
- *insert, delete minimum* –
  O(height of tree), **O(log n)** because tree is balanced

Binary heaps are arrays!

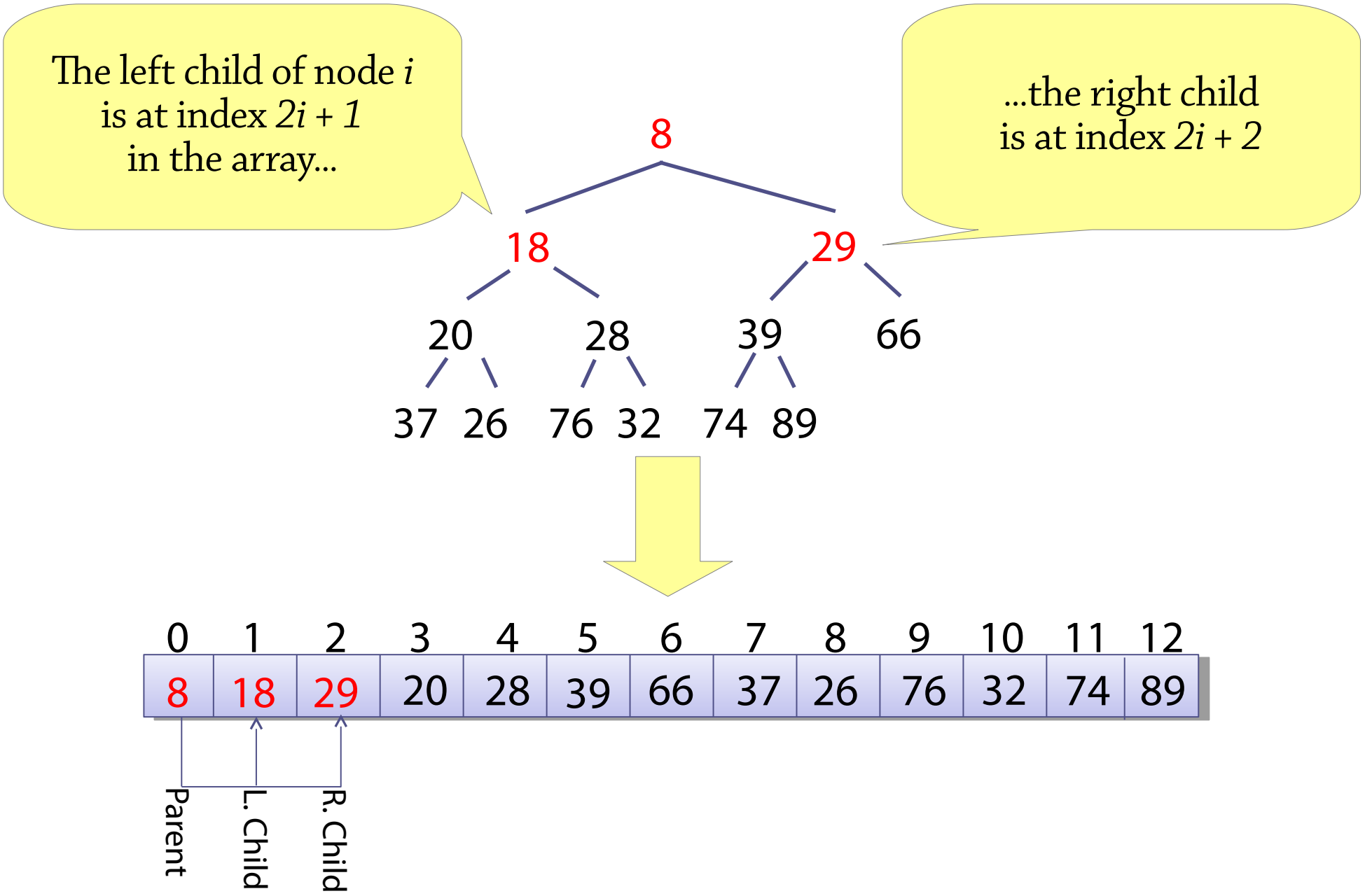
A binary heap is really implemented using an array!



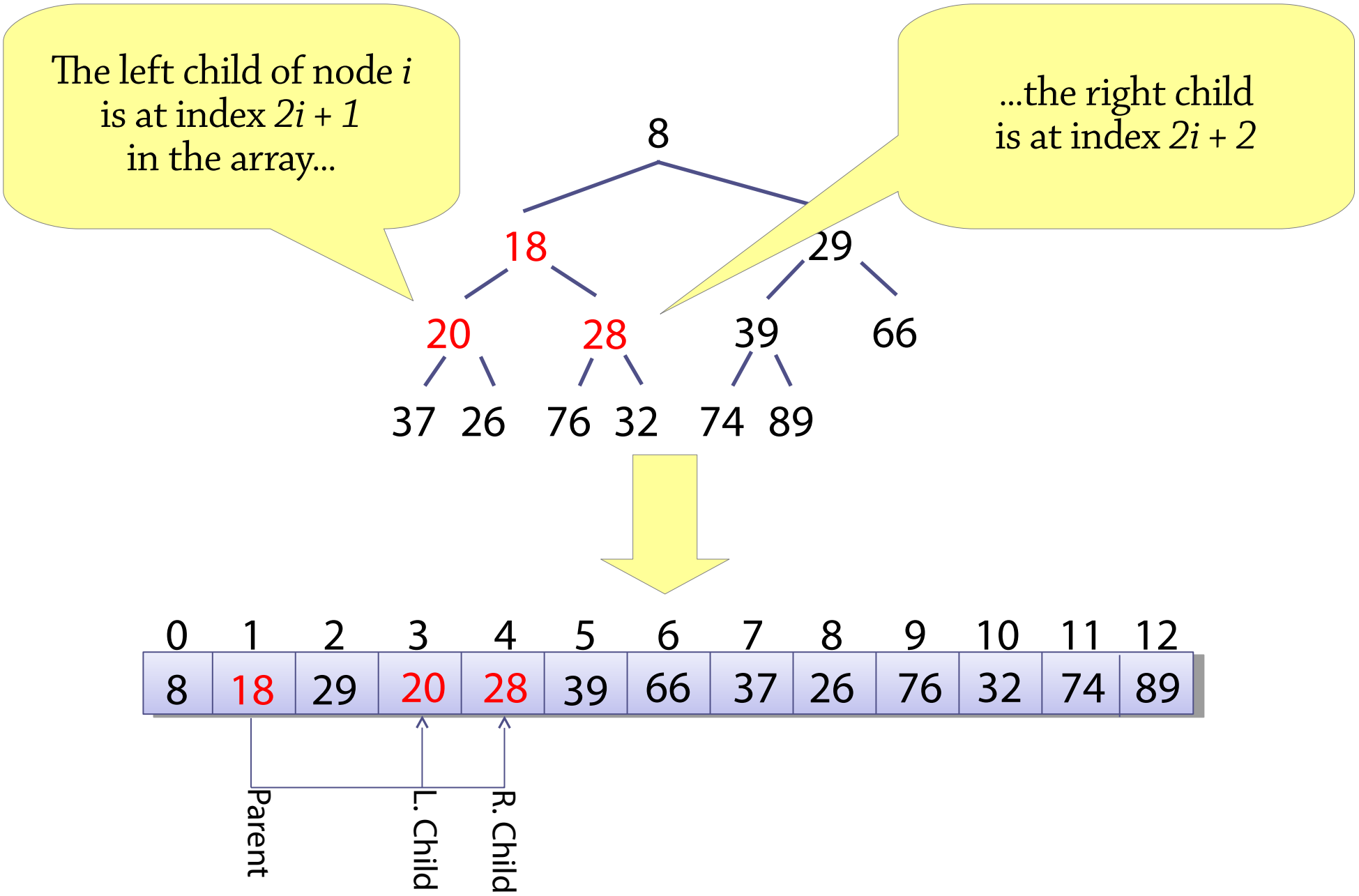
Possible because of completeness property



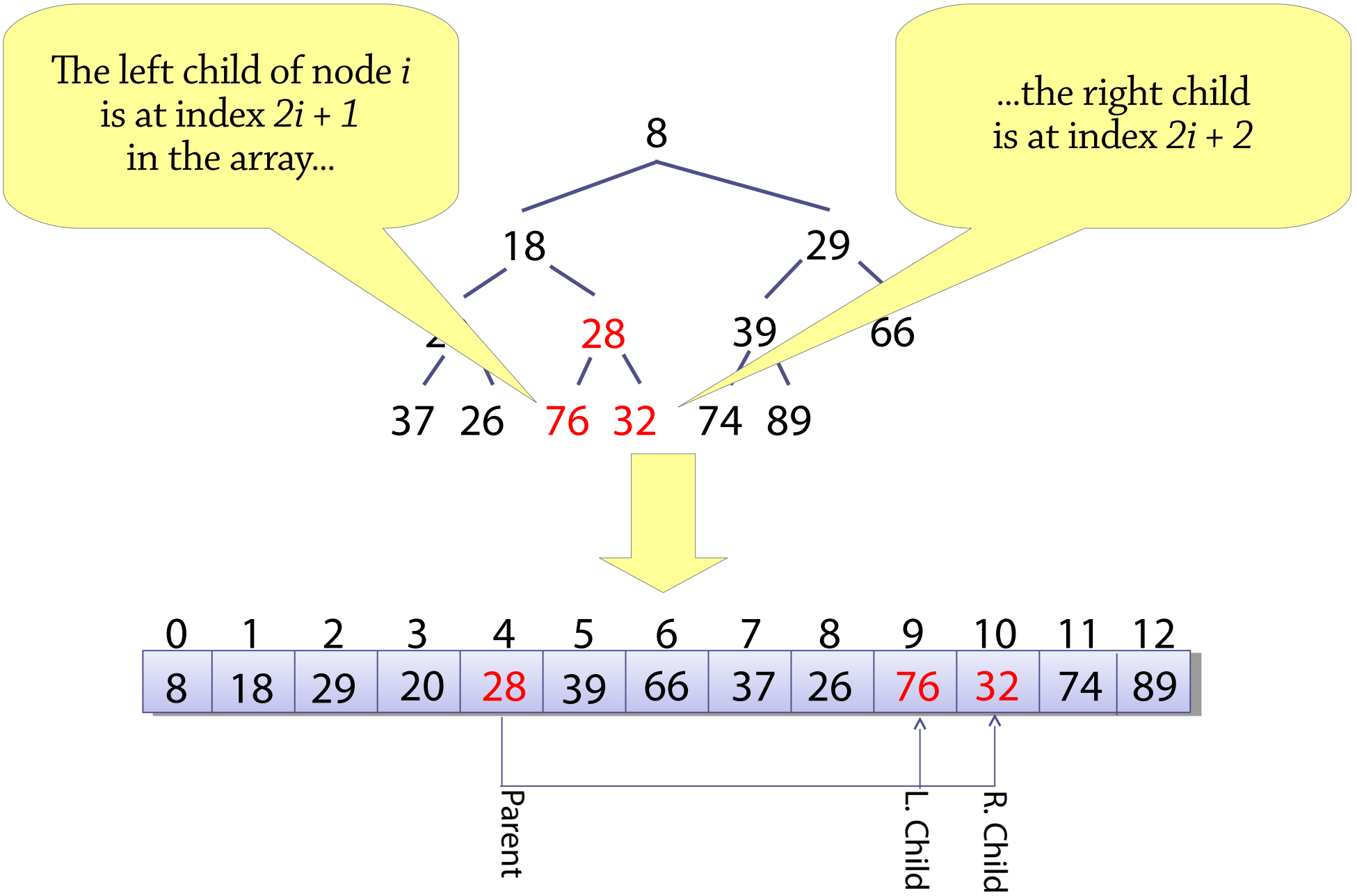
Child positions



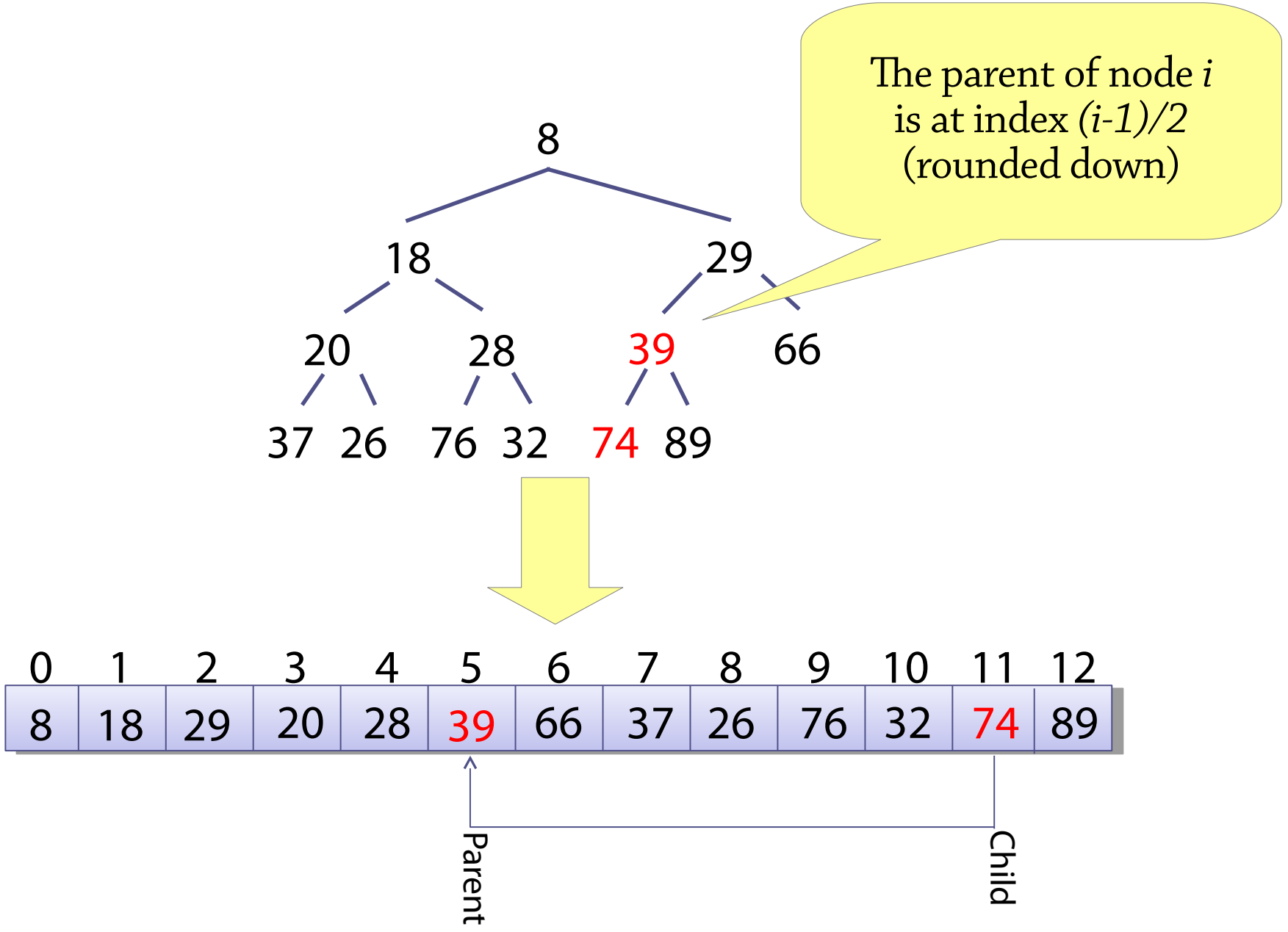
Child positions



Child positions



Parent position



Reminder: inserting into a binary heap

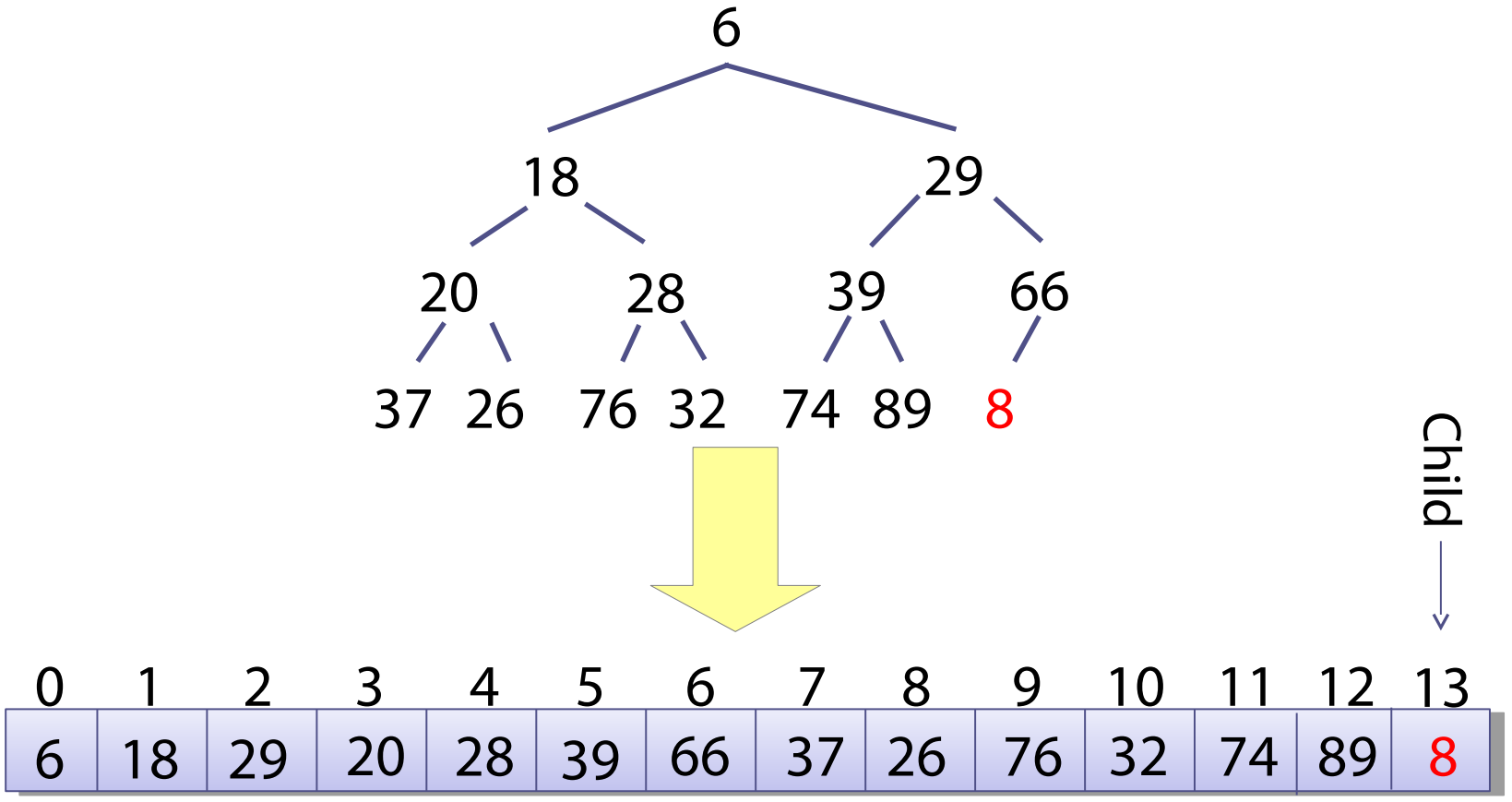
To insert an element into a binary heap:

- 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!

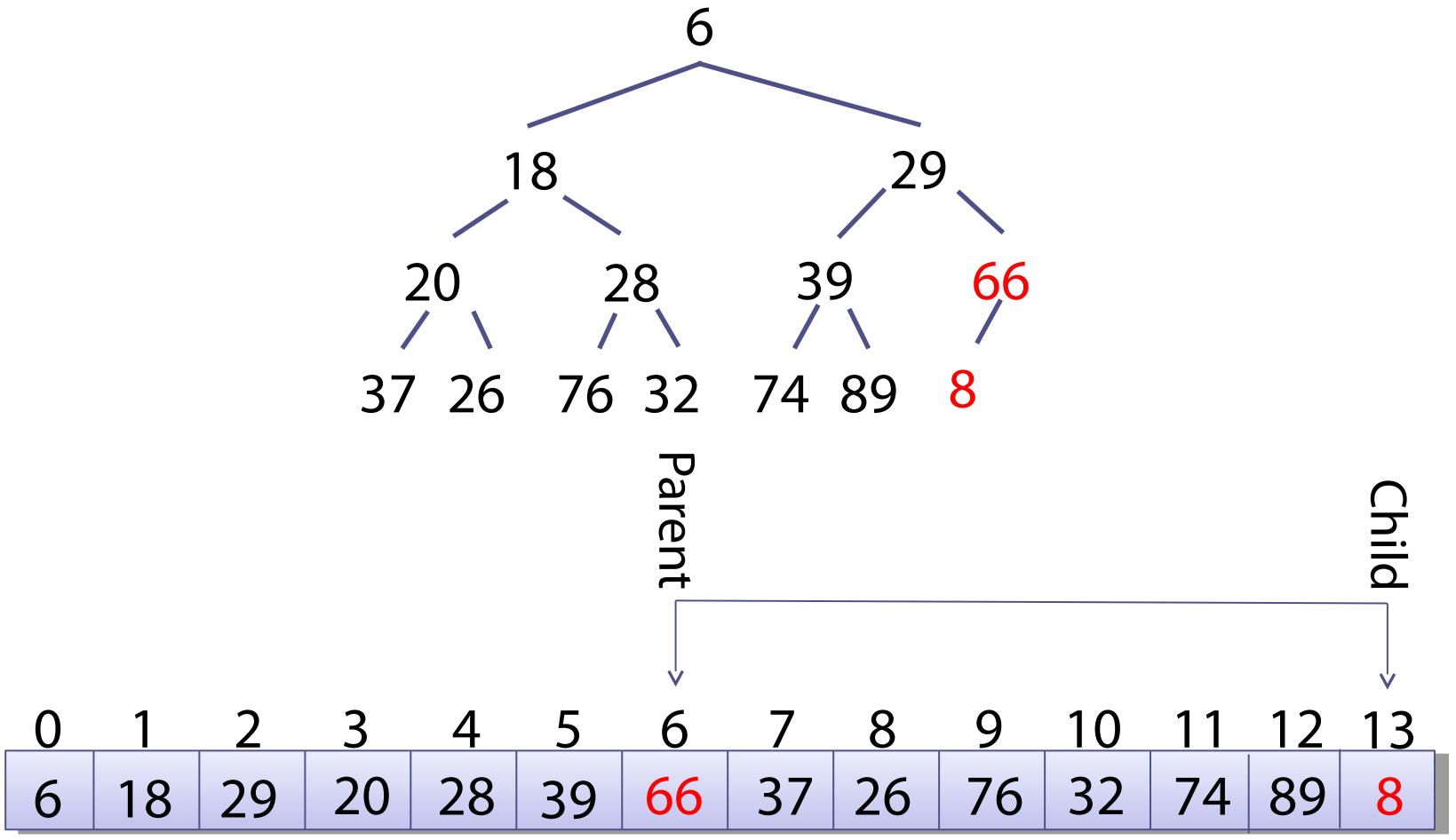
Inserting into a binary heap

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



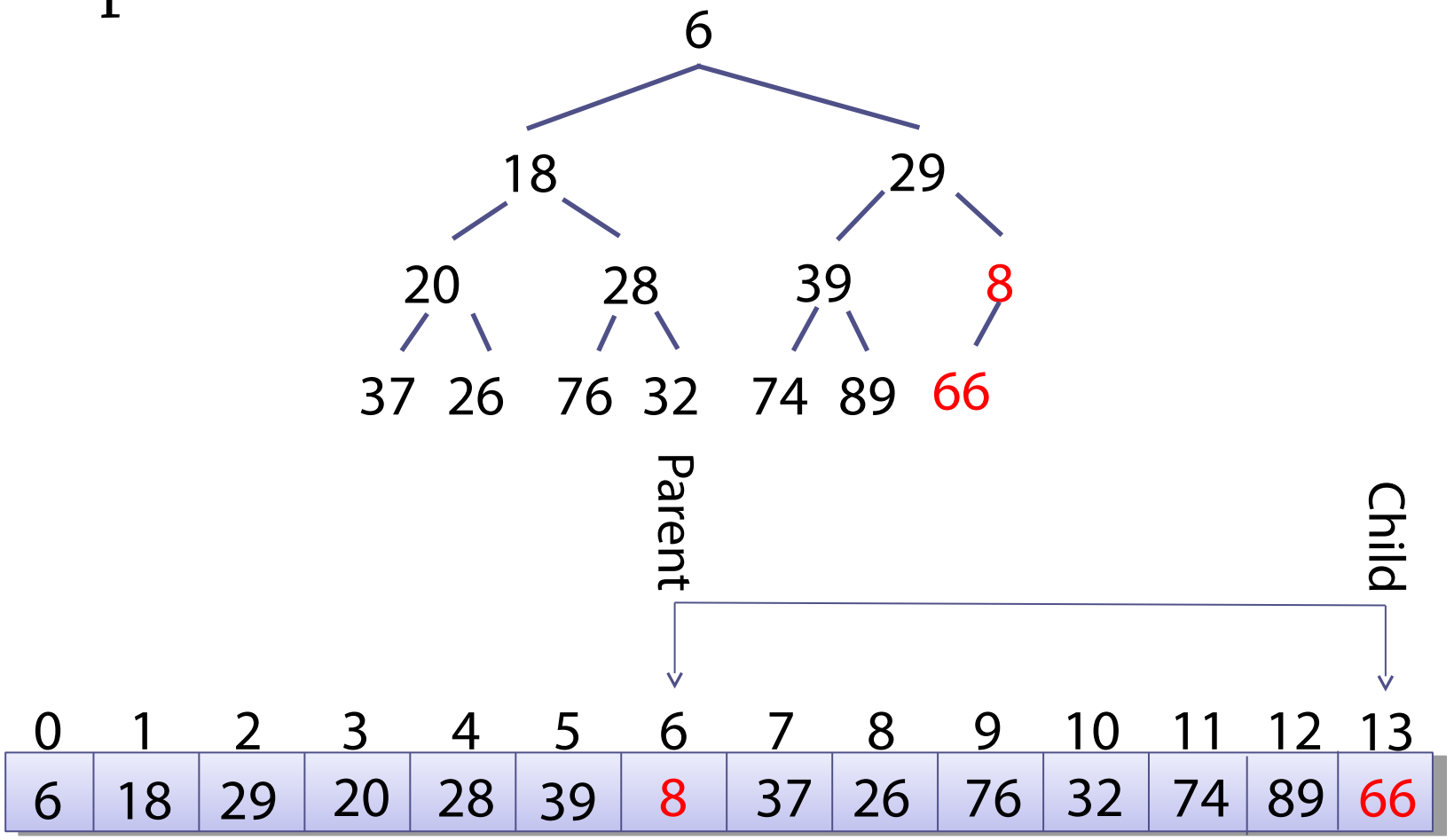
Inserting into a binary heap

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



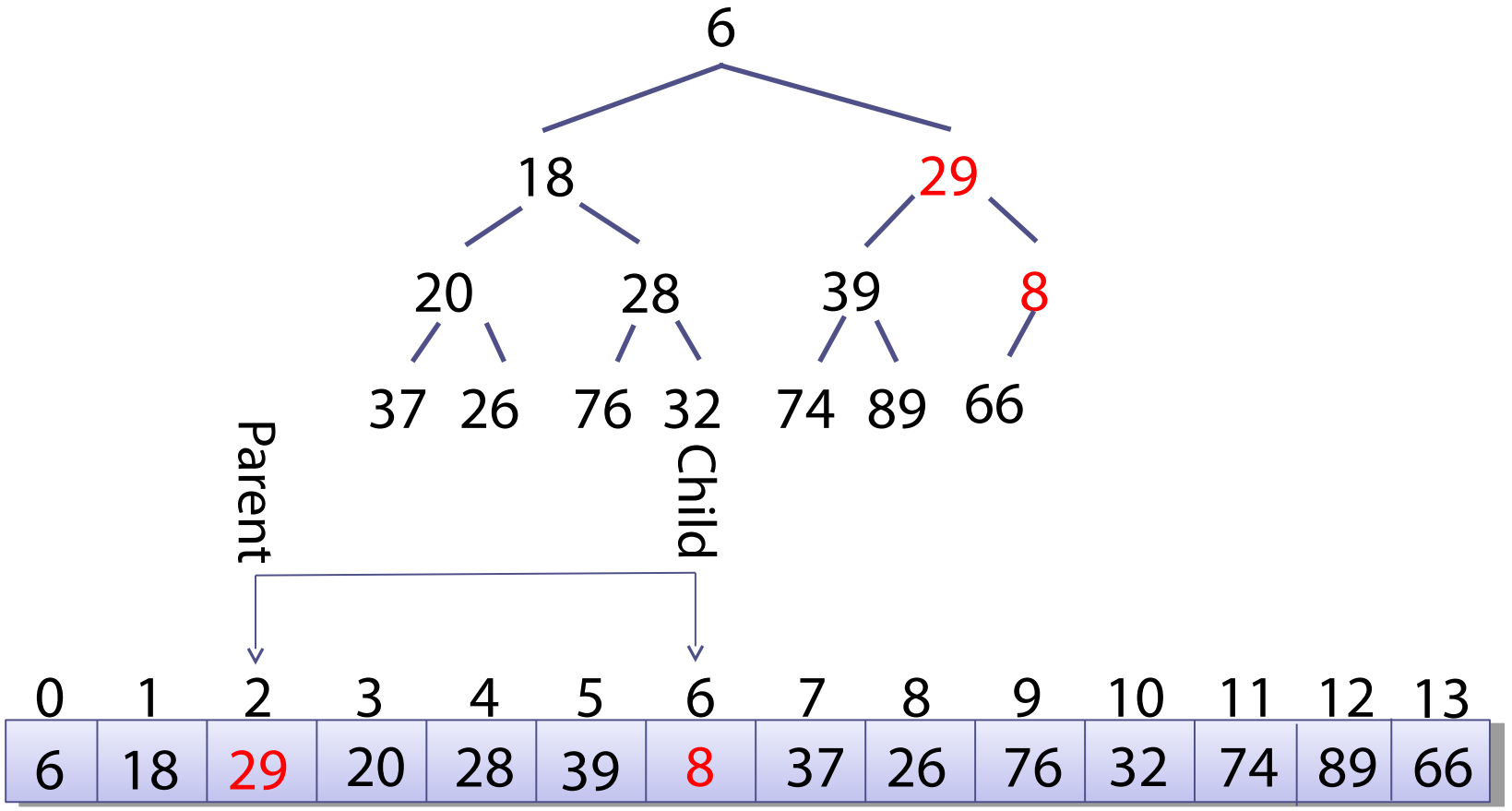
Inserting into a binary heap

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



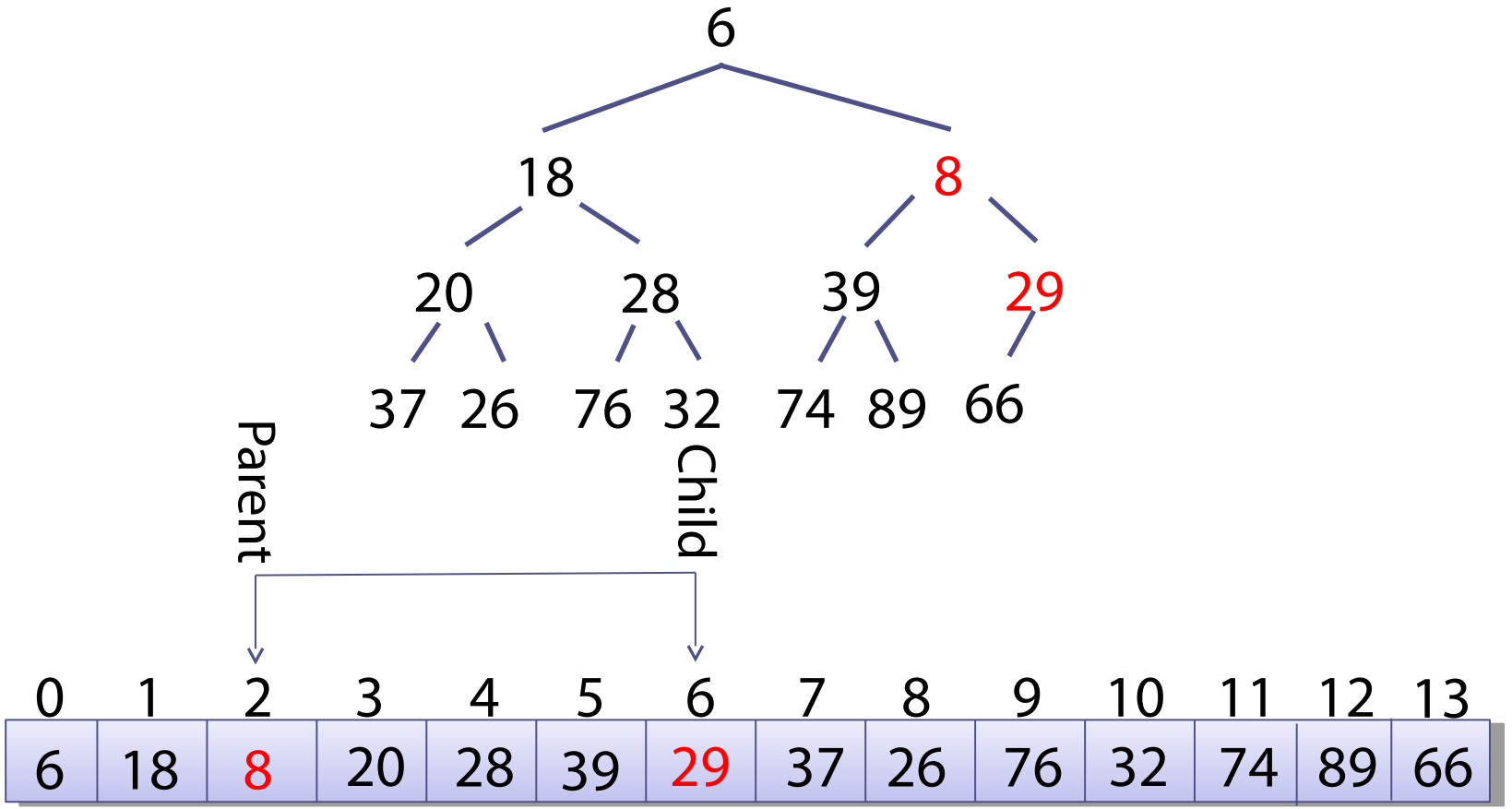
Inserting into a binary heap

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

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

Min vs max heaps

What we have seen is called a *min heap*

- can find, delete minimum element

There is a variant called a *max heap*

- can find, delete maximum element (but not minimum)

The implementation is totally symmetric

- In fact, if the min heap implementation takes the *comparator* as an argument, you can use it as a max heap just by changing the comparator (as you will discover doing the lab)

Heapsort

Heapsort uses a heap to sort an array, in-place!

- Convert the array into a max heap, which can be achieved by "sifting" each element in turn (in linear time!)
- Find and remove the maximum element, store it in the last element of the array (which is no longer used by the heap)
- Repeat this process until the heap is empty – the elements have been removed in decreasing order of size and stored starting from the end of the array and working backwards – i.e., they are now sorted

Not in the course, but historically important

- First O(n log n) sorting algorithm
- Introsort – quicksort but switches to heapsort if the recursion depth gets too high (to avoid $O(n^2)$ behaviour)

Data structure design

How not to do it

Here is how *not* to design a data structure:

1. Take the operations you have to implement
2. Think very hard about how to implement them
3. Bash something together that seems to work

Because:

- You will probably have lots of bugs
- You will probably miss the best solution

Data structure design

How to design a data structure:

- Pick a representation
  Here: we represent the priority queue by a binary tree
- Pick an invariant
  Here: the heap property and completeness

Once you have the right representation and invariant, *the operations often almost "design themselves"!*

- There is often only one way to implement them

You could say...
data structure = representation + invariant

Picking a representation and invariant

How do you know which representation and invariant to go for?

Good plan: have a first guess, see if the operations work out, then tweak it

- Priority queues: at first we tried a sorted array, but then *remove minimum* needed to delete the first element (inefficient). Then we tried a tree instead. Putting the smallest element at the root led us to the heap property. Completeness allows us to represent the heap as an array for extra efficiency.
- Queues: at first we tried a dynamic array, but there was no way to efficiently remove items, so we switched to a circular array

Takes practice!

Checking the invariant

What happens if you *break the invariant*?

- e.g., insert simply adds the new element to the end of the heap

Answer: nothing goes wrong straight away, but later operations might fail

- A later *find minimum* might return the wrong answer!

These kind of bugs are a nightmare to track down!

Solution: *check the invariant*

Checking the invariant

Define a method

```
bool invariant()
```

that returns *true* if the invariant holds

in this case, if the heap property holds

Then, in the implementation of every operation, do

```
assert invariant();
```

This will *throw an exception* if the invariant doesn't hold!

(Note: in Java, must run program with `-ea`)

Checking invariants

Writing down and checking invariants will help you *find bugs much more easily*

- Very many data structure bugs involve breaking an invariant
- Even if you don't think about an invariant, if your data structure is at all fancy there is probably one hiding there!
- Almost all programming languages support assertions – **use them** to check invariants and make your life easier

Looking back on older designs

We implemented bounded queues by an array and a pair of indices *front* and *back*

- The *contents of the queue* is the elements between index *front* and index *back*

Once we decide on this representation, there is only one way to implement the queue!

- Here, "representation" means – what datatype we use, plus what an instance of that datatype *means* as a queue (in this case, what the queue contains)

Today

Main topic was binary heaps, but it was also about *how to design data structures*

- The main task is not *how to implement the operations*, but choosing the right representation and invariant
- These are the main design decisions – once you choose them, lots of stuff falls into place
- Understanding them is the best way to understand a data structure, and checking invariants is a very good way of avoiding bugs!

But you also need lots of existing data structures to get inspiration from!

- Many of these in the rest of the course