Priority queues

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(highest)-priority element.

Applications

A natural application for priority queues is to handle access to a limited resource where the resource users can be assigned different degree of urgency.

- An electronic queueing system for an emergency room where the patients' conditions varies in severity.
- Processes with priority levels queueing for resources (such as computation time).

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: find and remove minimum element
  O(n)

Finding and removing the minimum is quite expensive.

An inefficient priority queue

Idea 2: use a *array sorted in descending order*

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

Finding and removing the minimum is cheap! 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*)

Pre-, postconditions and invariants

- Preconditions – requirements on a function's input (not expressed by types) that must hold when it's called.
- Postcondition – requirements on a function's output that will hold when it returns.
- Invariants – requirements on data that exists in between function calls. In Java this typically means requirements on an object's instance variables. Invariants can be seen being pre- and postconditions that are added to all instance methods of a class.

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!

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

- Queues: at first we tried a dynamic array, but there was no way to efficiently remove items, so we switched to a circular array
- 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.

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 in Java

Define a method

```
boolean invariant()
```

that returns *true* if the invariant holds

- in this case, if the array is reverse-sorted

Then, in the implementation of every operation, do

```
assert invariant();
```

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

(Note: must run program with `-ea`)

Invariants in Haskell

Define a function

```
invariant :: ReprType → Bool
```

Then add an extra case to all operations:

```
anOperation x =
 assert (invariant x) $
 theActualCode
```

For ghc, checking assertions is enabled by default. Disable it by giving option `-O` or `-fignore-asserts`.

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

Example: Implementing priority queue

- Implement prio queue using inefficient idea 2.
- Both in Java and Haskell
- Uses assertions with invariant
- Also example of generics and `Comparator`/`Ord`.
- See code in lecture material.

Implementing priority queues using binary heaps

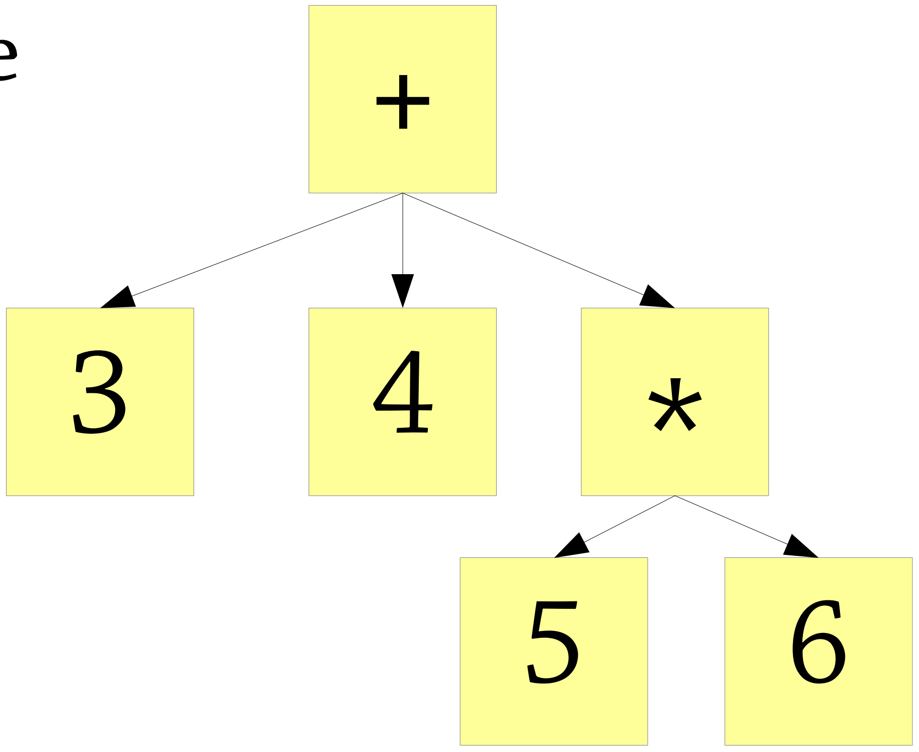
Trees

A *tree* is a hierarchical data structure

- Each node can have several *children* but only has one *parent*
- The *root* has no parents; there is only one root

Example: expression tree

Example:
directory hierarchy



Binary trees

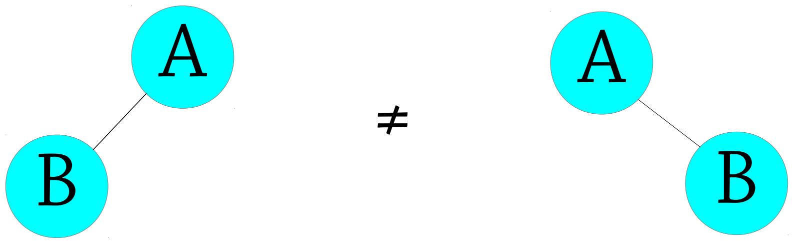
Very often we use *binary trees*, where each node has at most two children

```
class Node<E> {
  E value;
  Node<E> left, right;
}
```

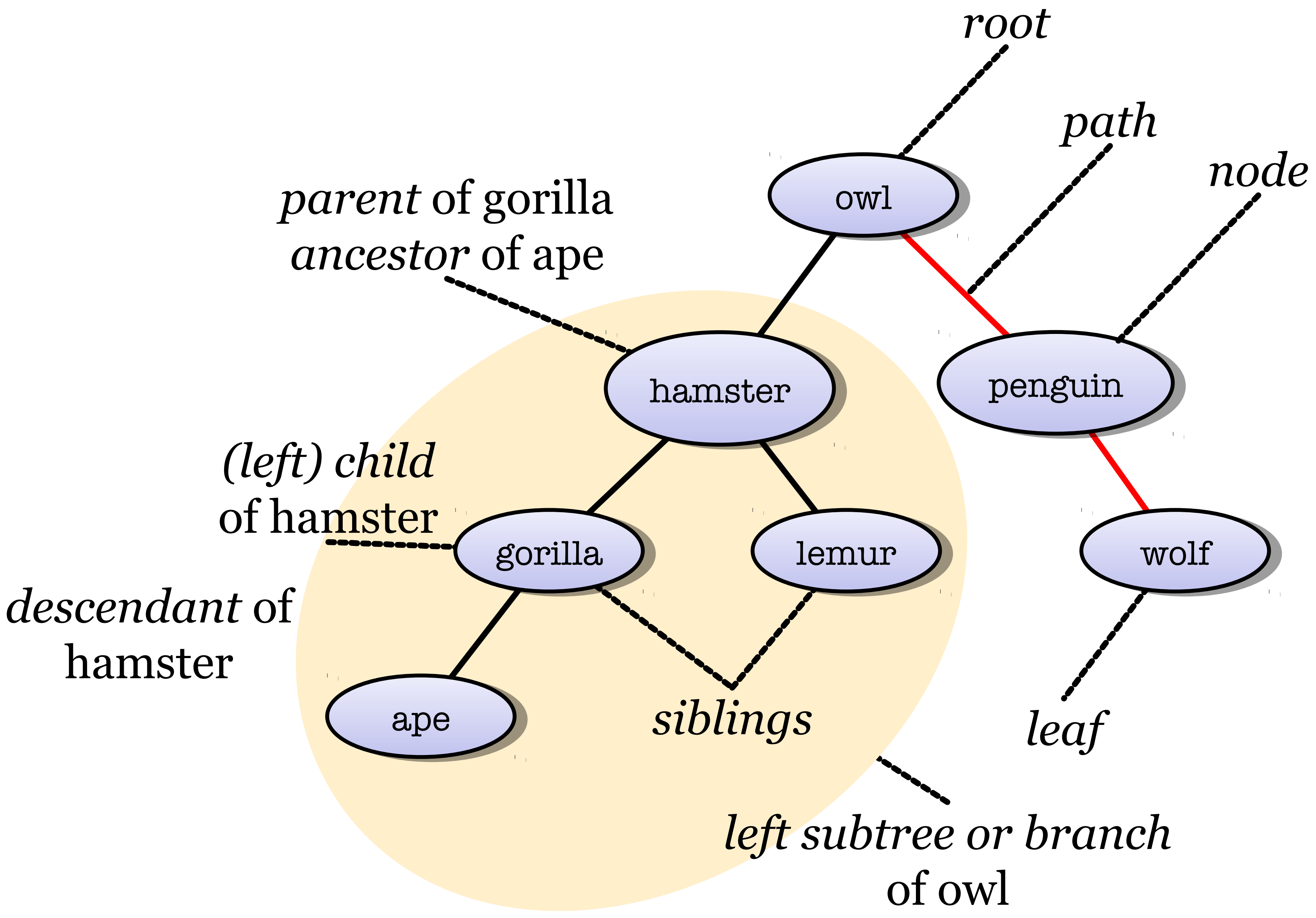
Can be `null`

```
data Tree a
  = Node a (Tree a) (Tree a)
  | Nil
```

`Note that these two binary trees are not considered the same:`

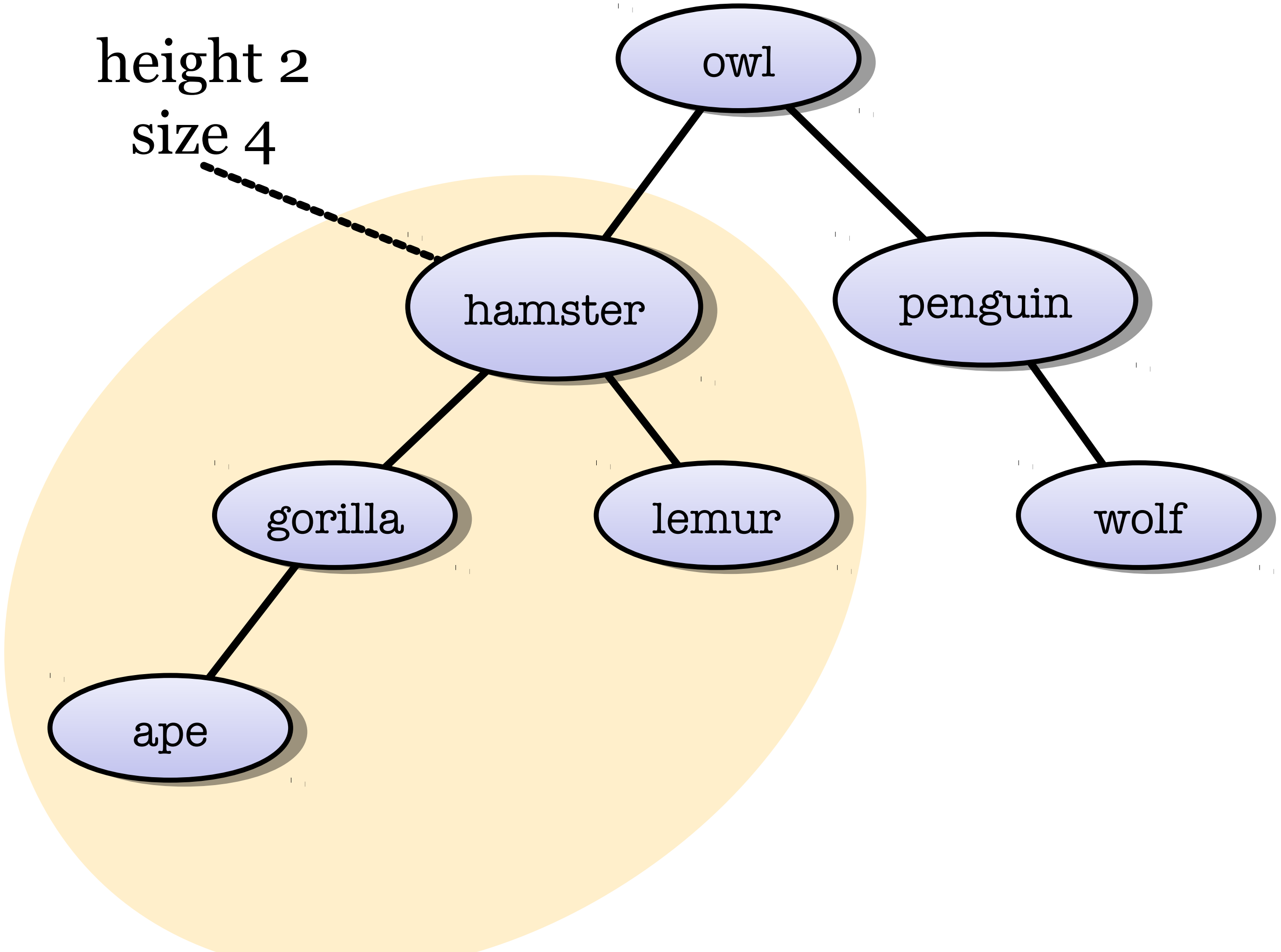


Terminology



Terminology

height = length of longest path from root to leaf
size = number of nodes in tree
depth/level of node = length of path to root

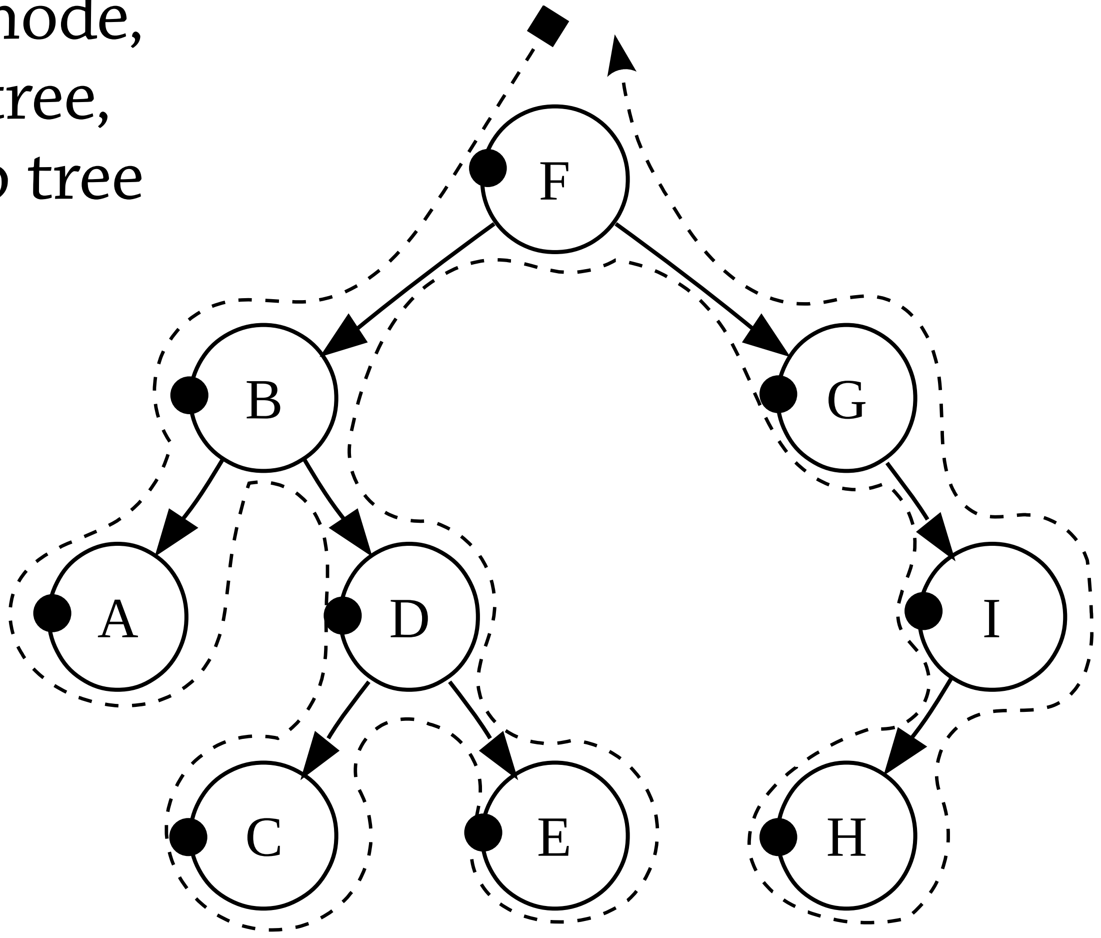


Tree traversal

- Two major modes of tree traversal (visiting all nodes):
  - DFS, depth-first search. For binary trees there are three variants:
    - Pre-order
    - In-order
    - Post-order
  - BFS, breadth-first search.
- See the following slides for visualisations.

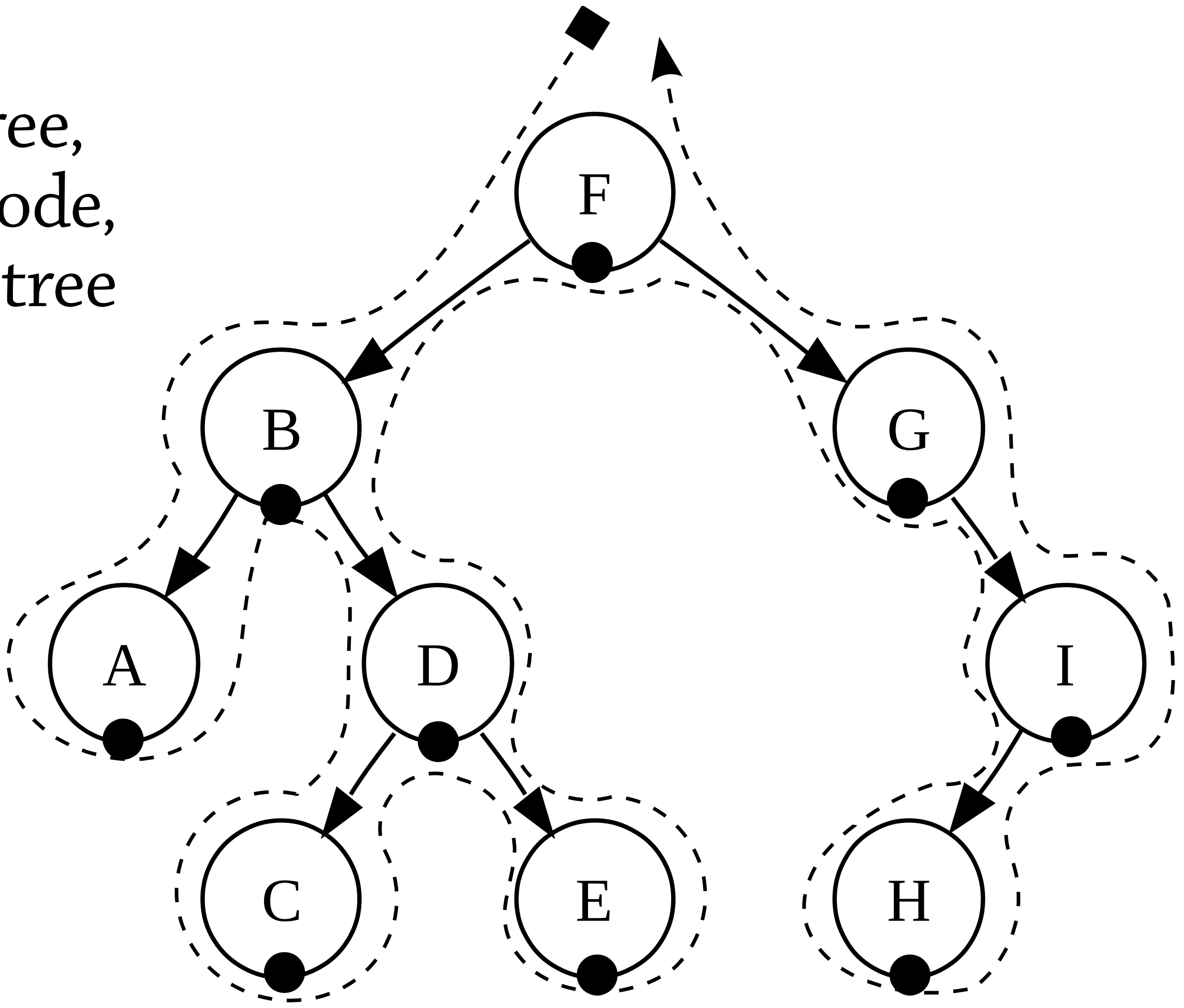
Pre-order traversal

First current node,
then left sub tree,
then right sub tree



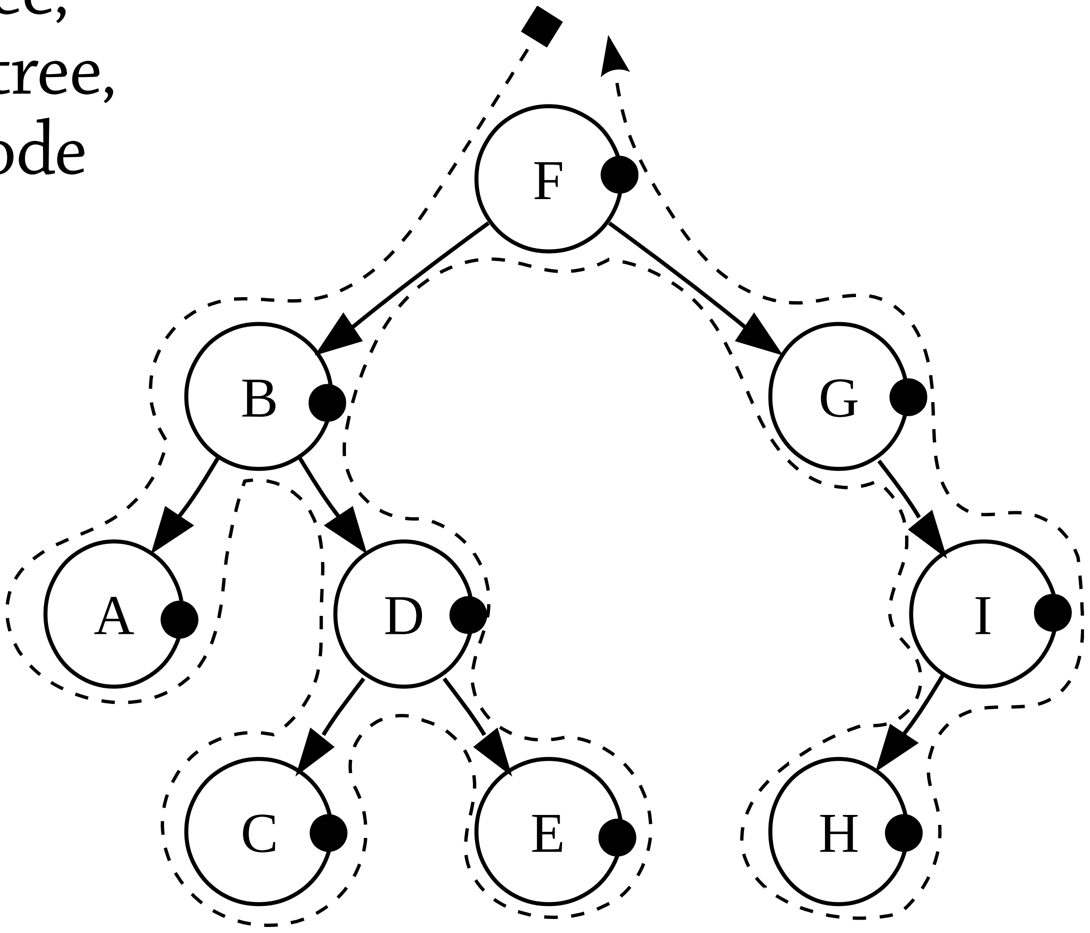
In-order traversal

First left sub tree,
then current node,
then right sub tree



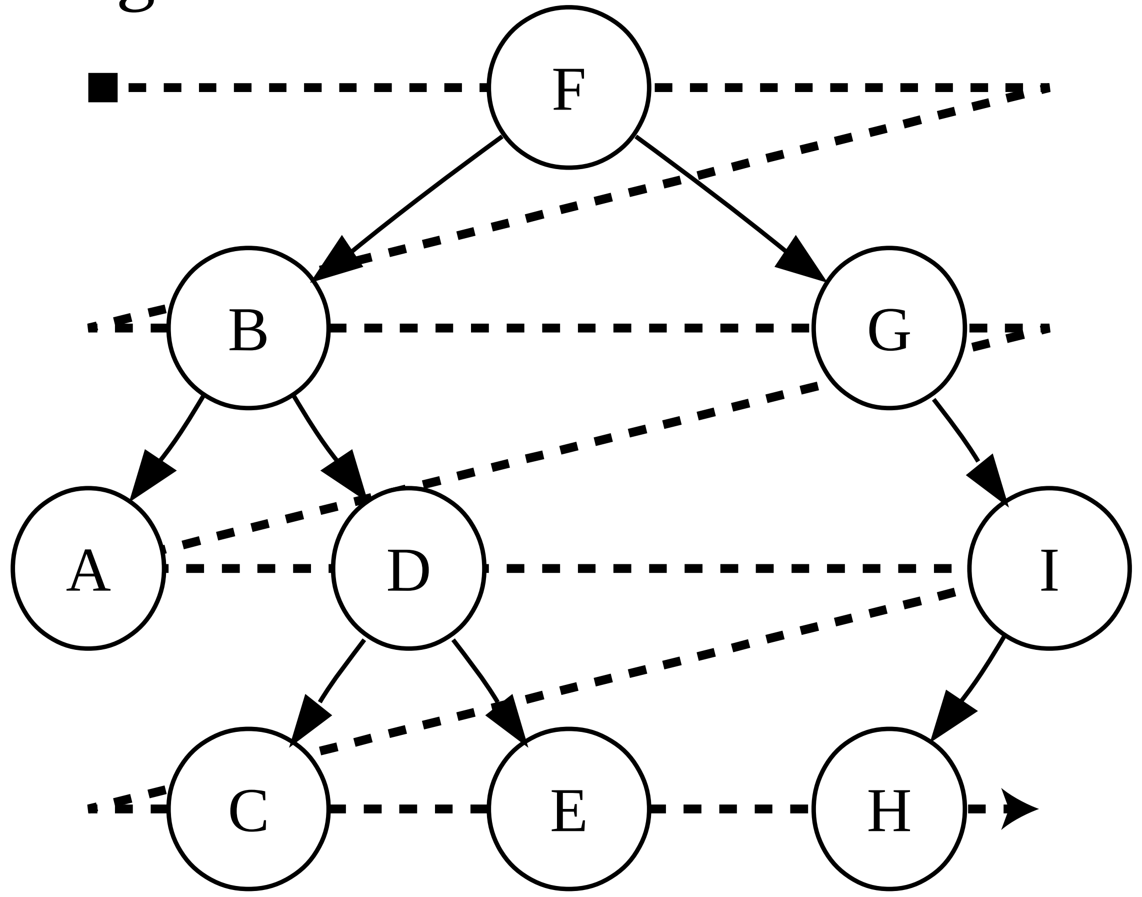
Post-order traversal

First left sub tree,
then right sub tree,
then current node



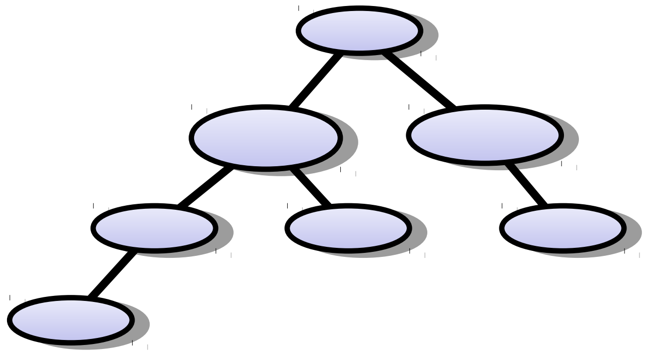
Breadth-first search

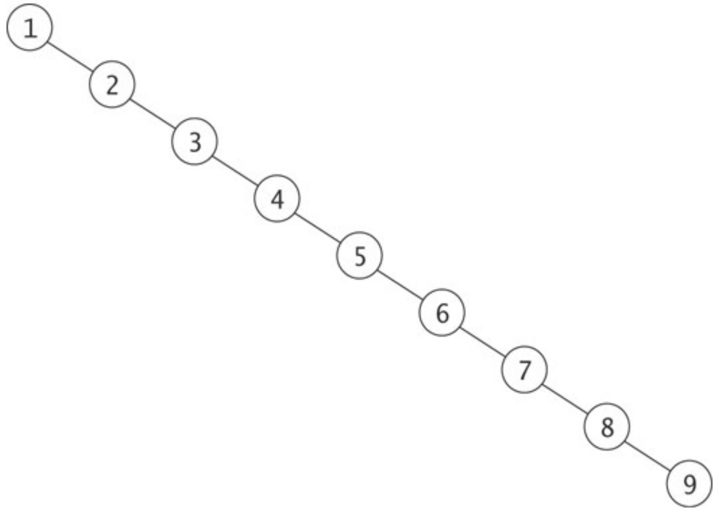
Each level from left to right



Balanced trees

A tree can be *balanced* or *unbalanced*. A balanced tree has restrictions on its shape so that it's not too high. A perfect binary tree (all leaves on the same level) has size $2^{h+1} - 1$.





If a tree of size *n* is

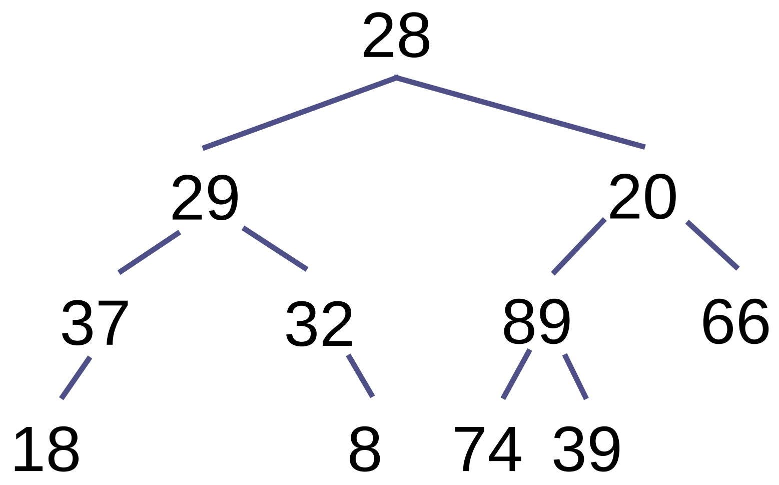
- balanced, its height is O(log n)
- unbalanced, its height could be O(n)

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!

Heaps – representation

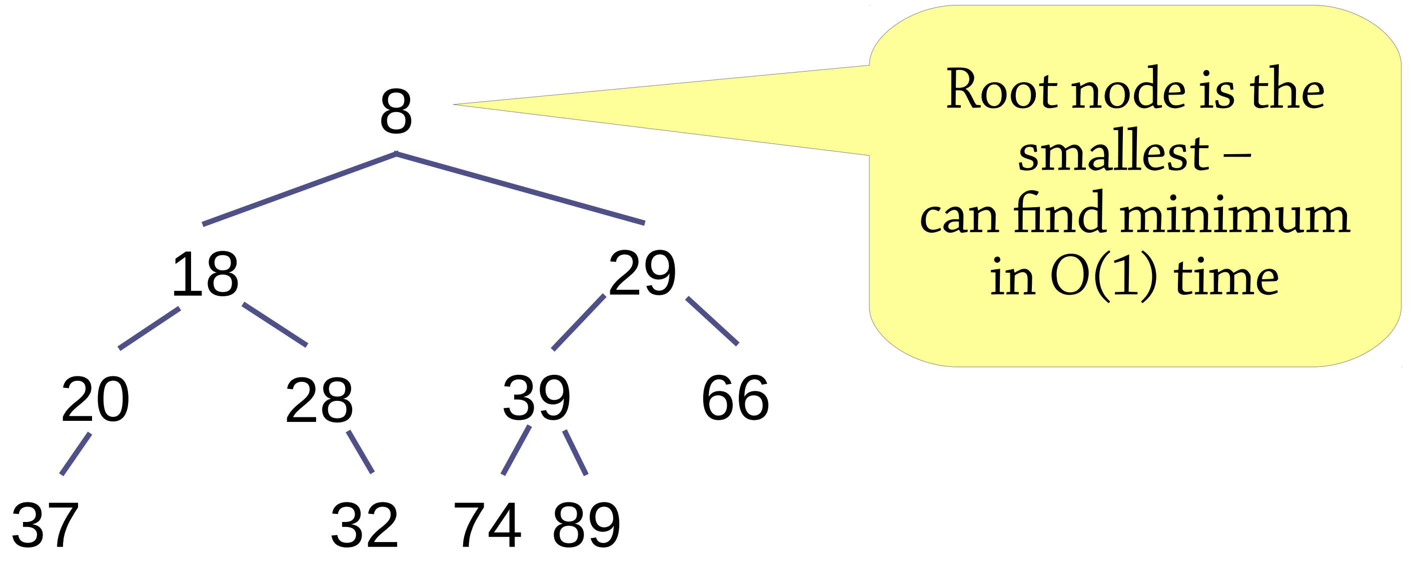
A heap implements a priority queue as abinary tree. Here is a tree:



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

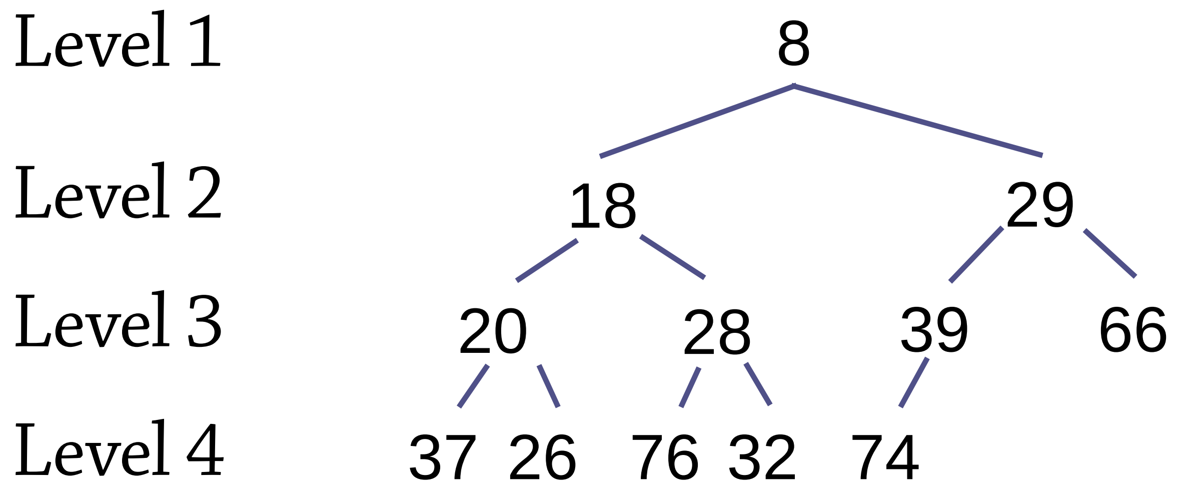
Why not just have the invariant "the root node is the smallest"? Because:

- 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)

Complete binary tree

- The height is O(log *n*) since $2^h \le n \le 2^{h+1} - 1$
- So complete trees are balanced.
- If we manage to implement operations with complexity O(*h*) then they will be O(log *n*)

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!

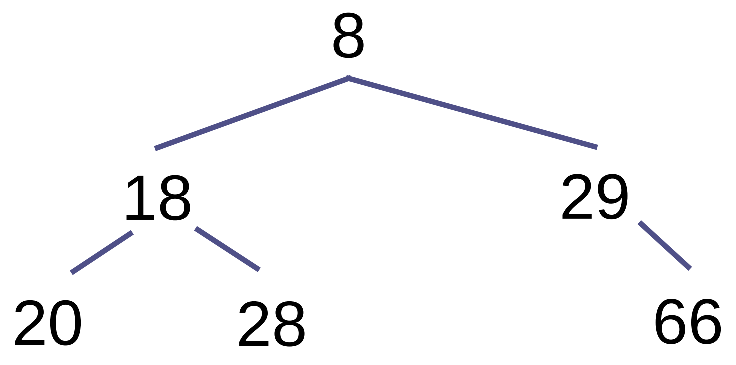
Binary heap invariant

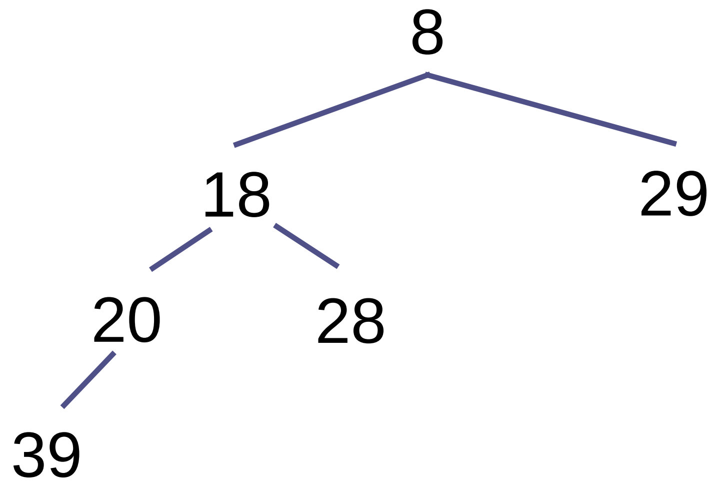
The binary heap invariant:

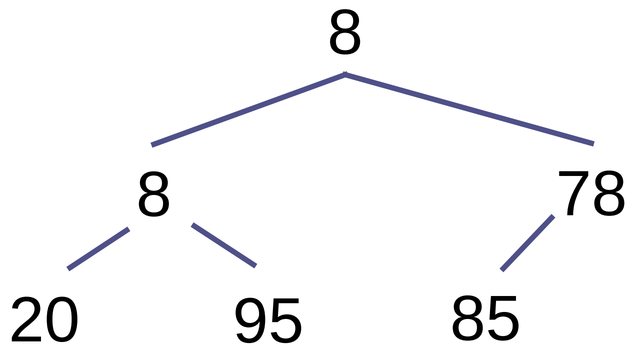
- The tree must be *complete*
- It must have the *heap property* (each node is less than or equal to its children)

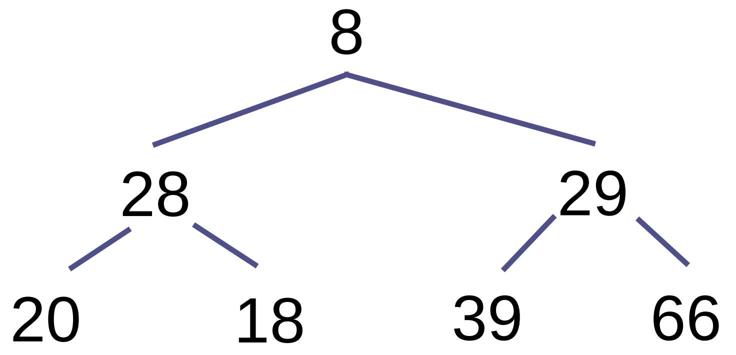
Remember, all our operations must preserve this invariant

Heap or not?

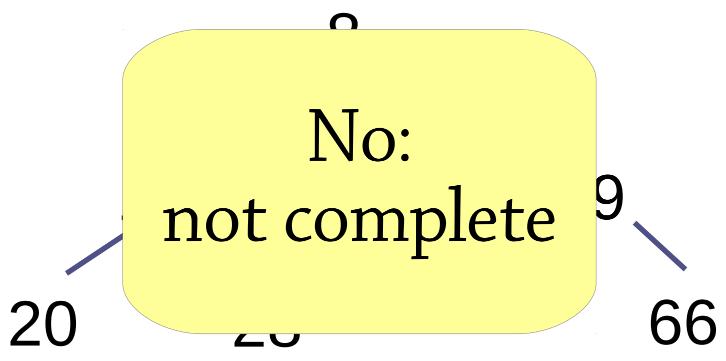




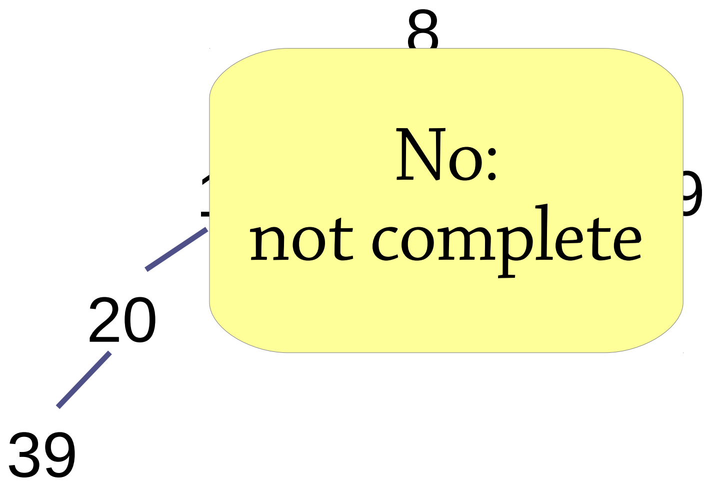




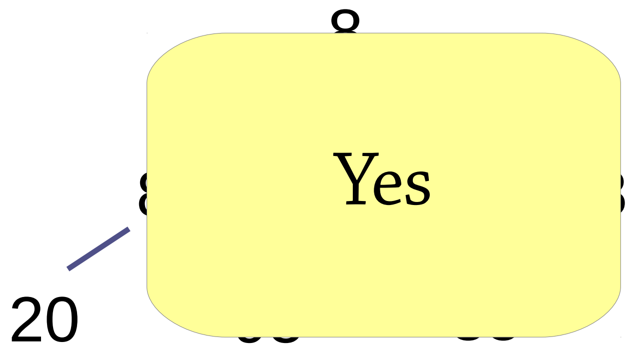
Heap or not?



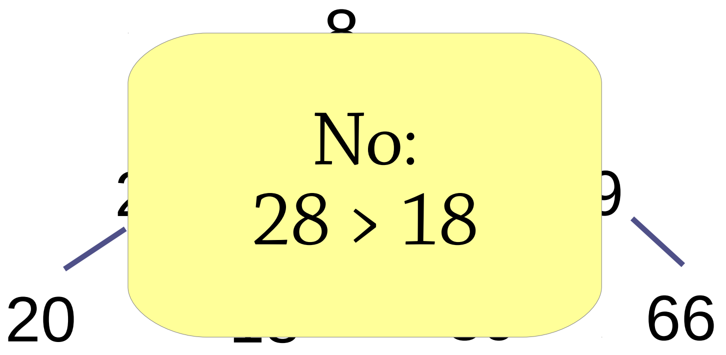
8
9
20
66
No:
not complete



8
9
20
39
No:
not complete



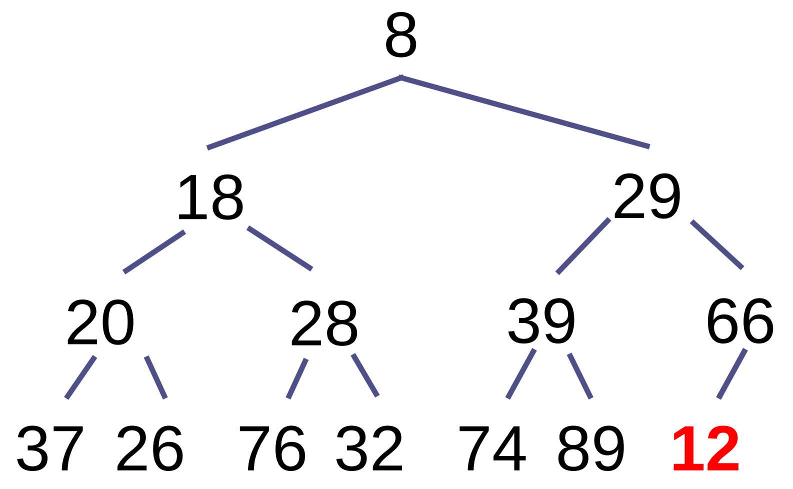
8
20
Yes



8
9
20
66
No:
28 > 18

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

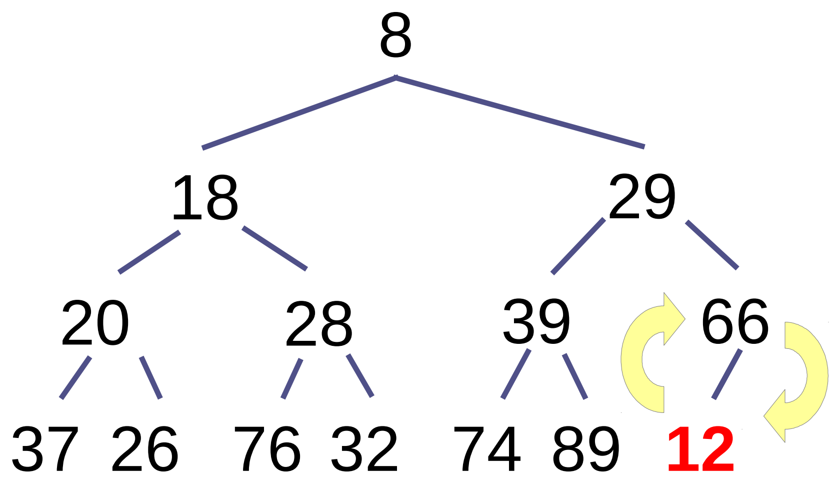
Often it's easier to separate these:

- 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

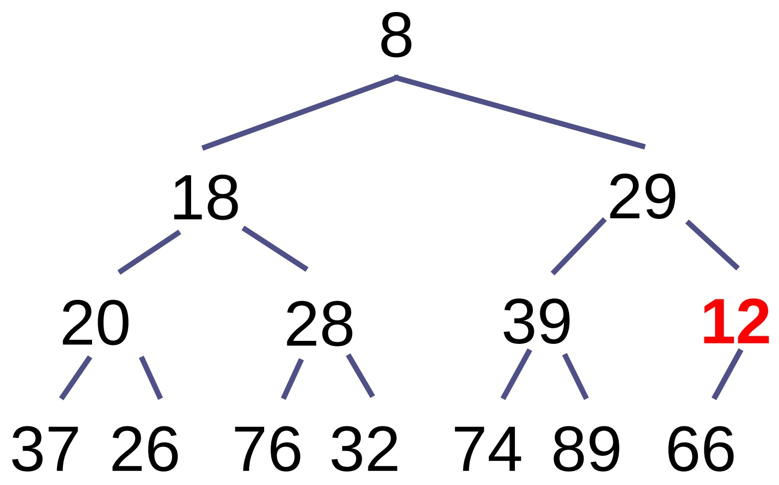
Adding an element to a binary heap

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



Adding an element to a binary heap

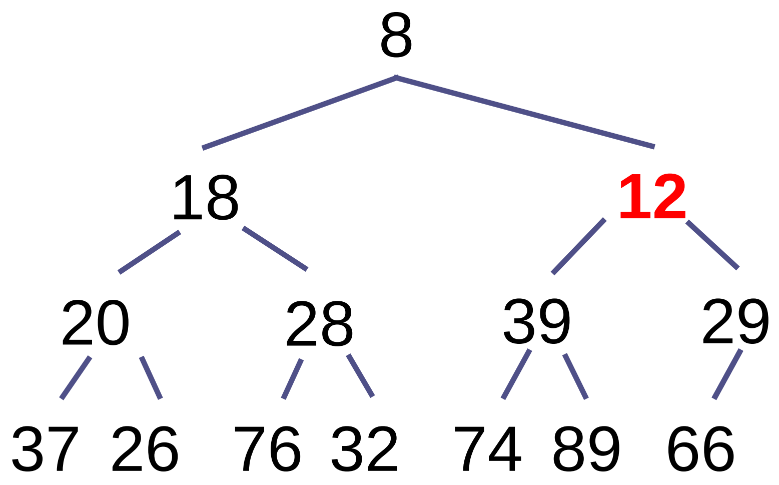
Step 2: if the new element is less than its parent, swap it with its parent



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.

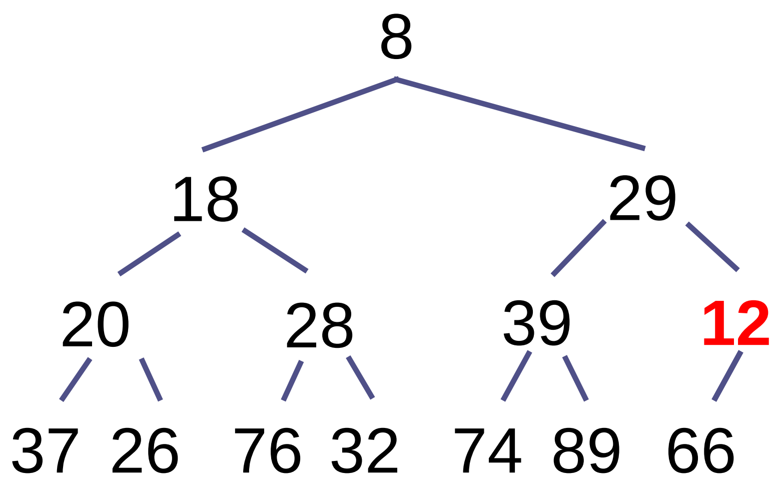


Now 12 is in its right place, and the invariant is restored. (Think about why this algorithm restores the invariant.)

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?)



Removing the minimum element

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

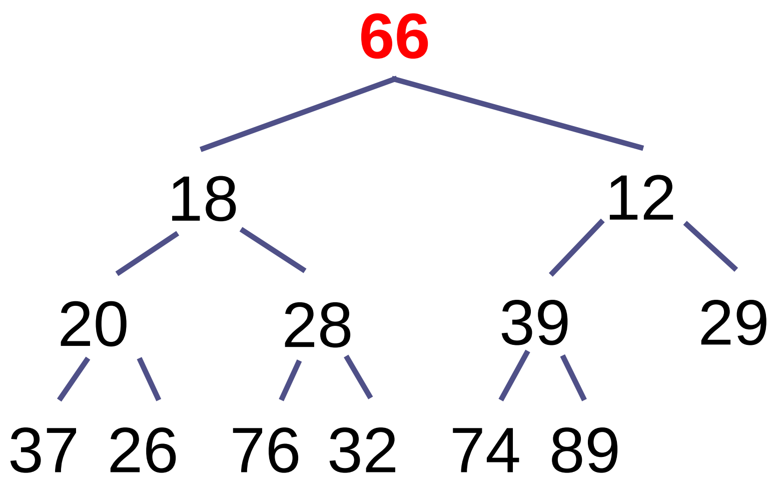
- First remove the minimum (root) element from the tree somehow, breaking the invariant in the process
- Then repair the invariant

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

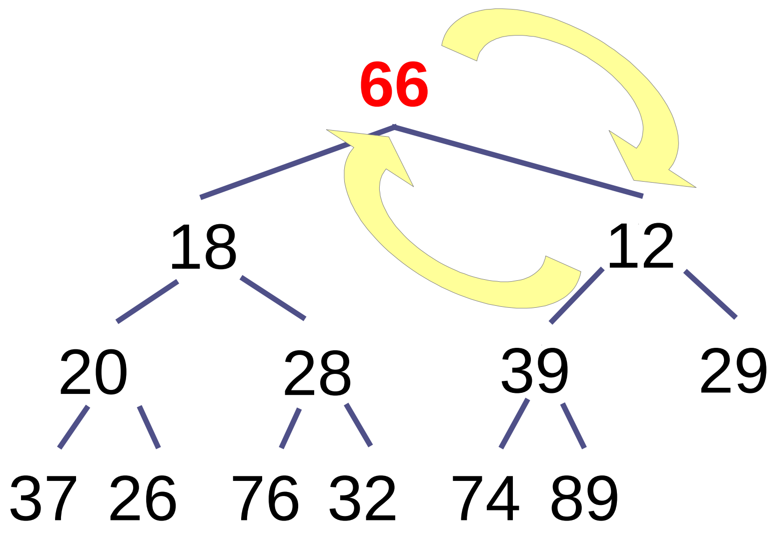
Step 1: replace the root element with the *last element* in the tree, and remove the last element



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

Removing the minimum element

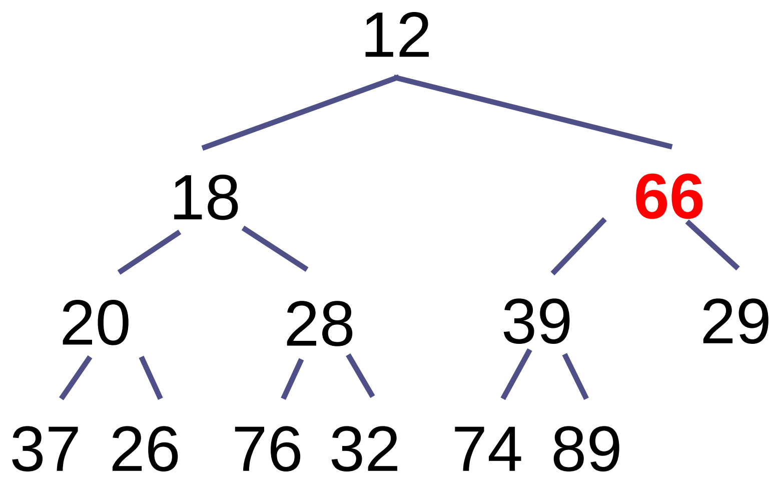
Step 2: if the moved element is greater than its children, swap it with its *least child*



(Why the least child in particular?)

Removing the minimum element

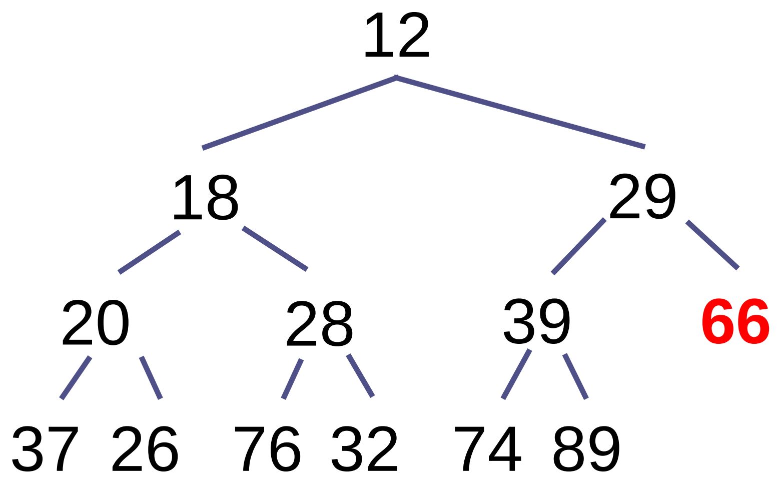
Step 2: if the moved element is greater than its children, swap it with its *least child*



(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

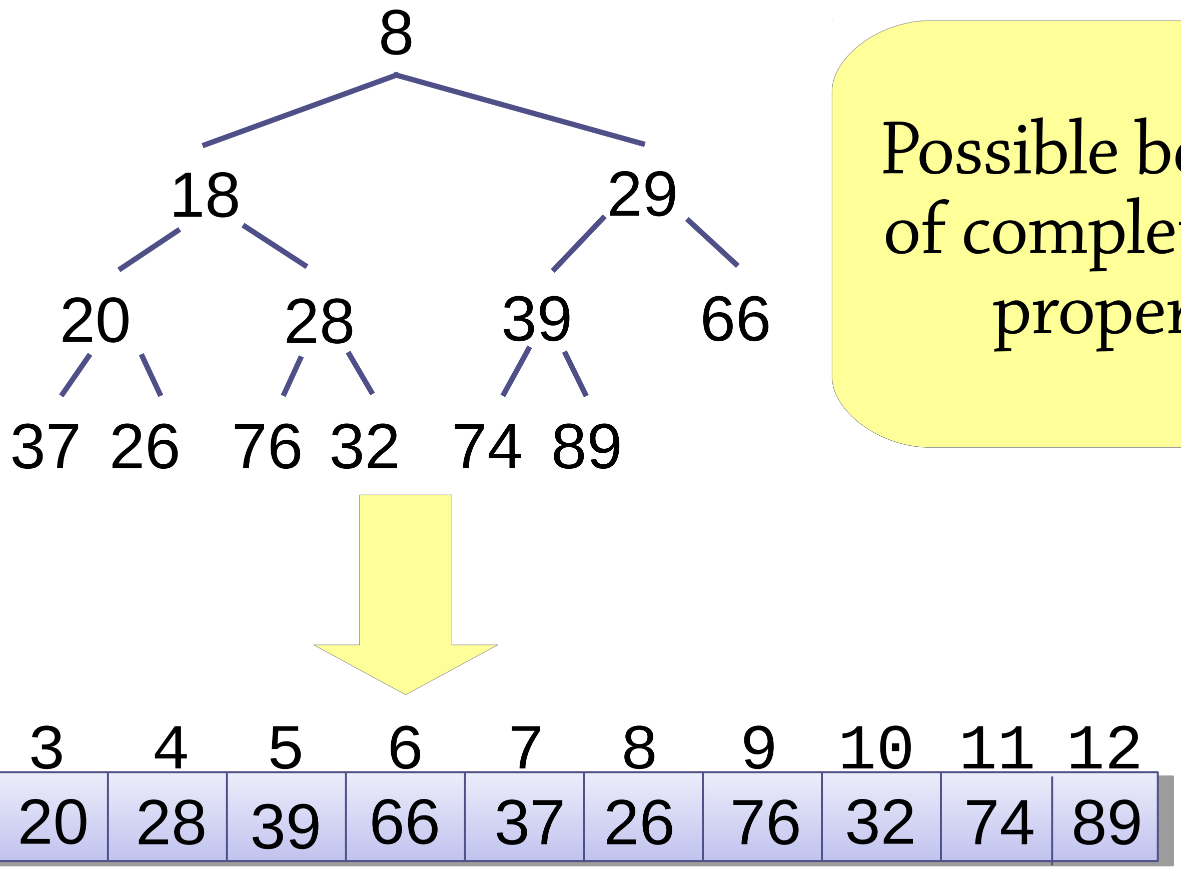
- *Heap property* – means smallest value is always at root
- *Completeness* – means tree is always balanced

Complexity:

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

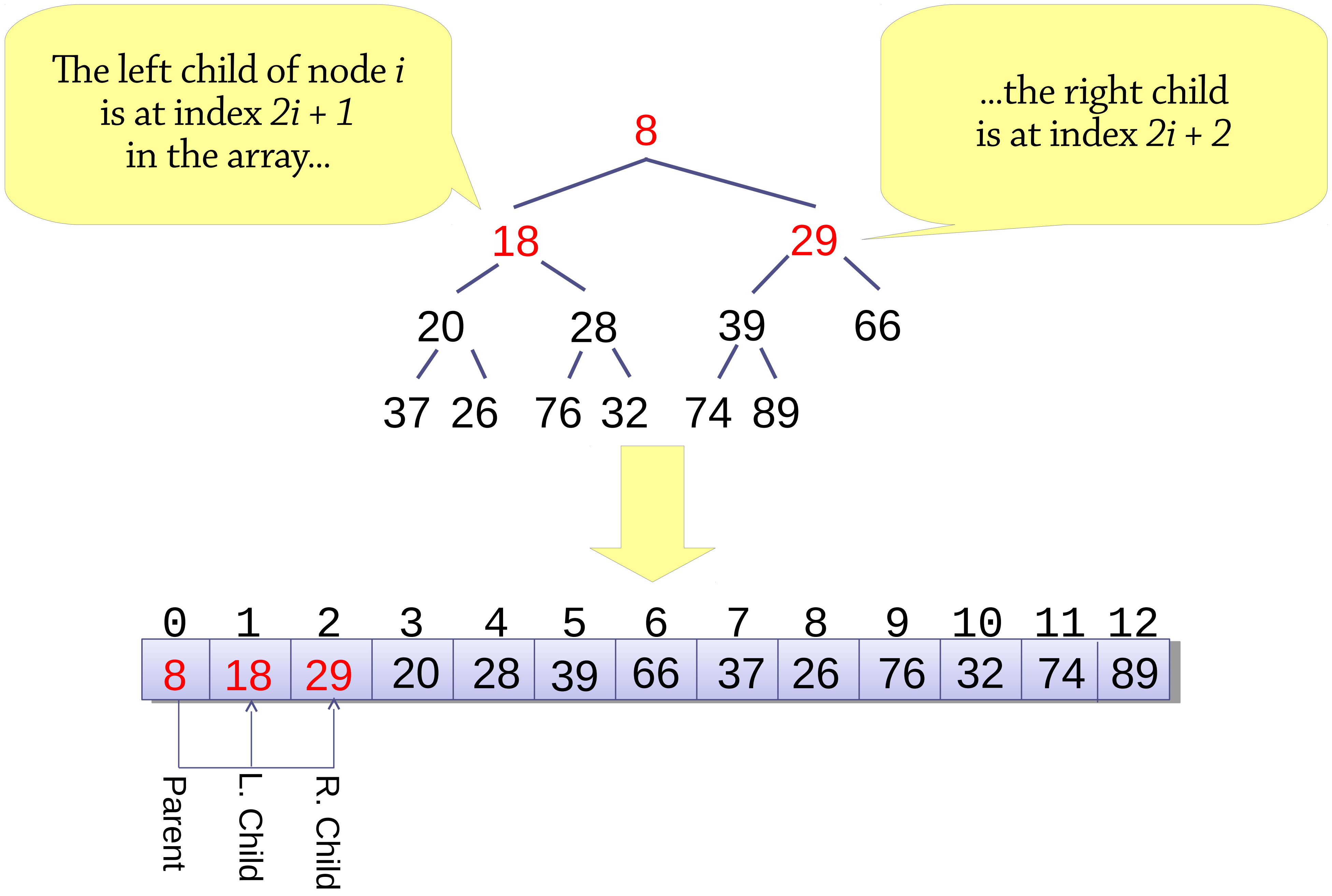
Binary heaps are arrays!

A binary heap is really implemented using an array!

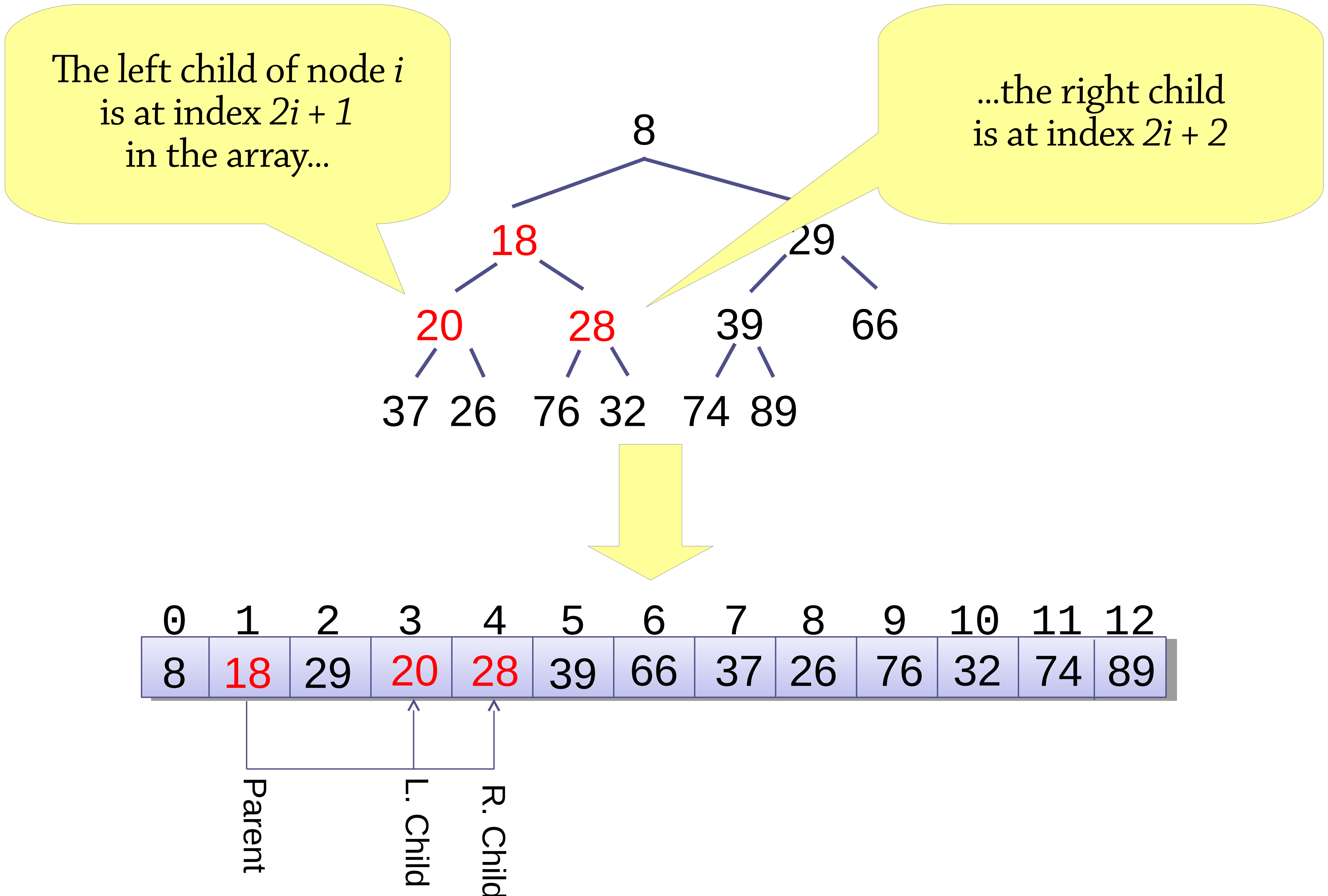


Possible because of completeness property

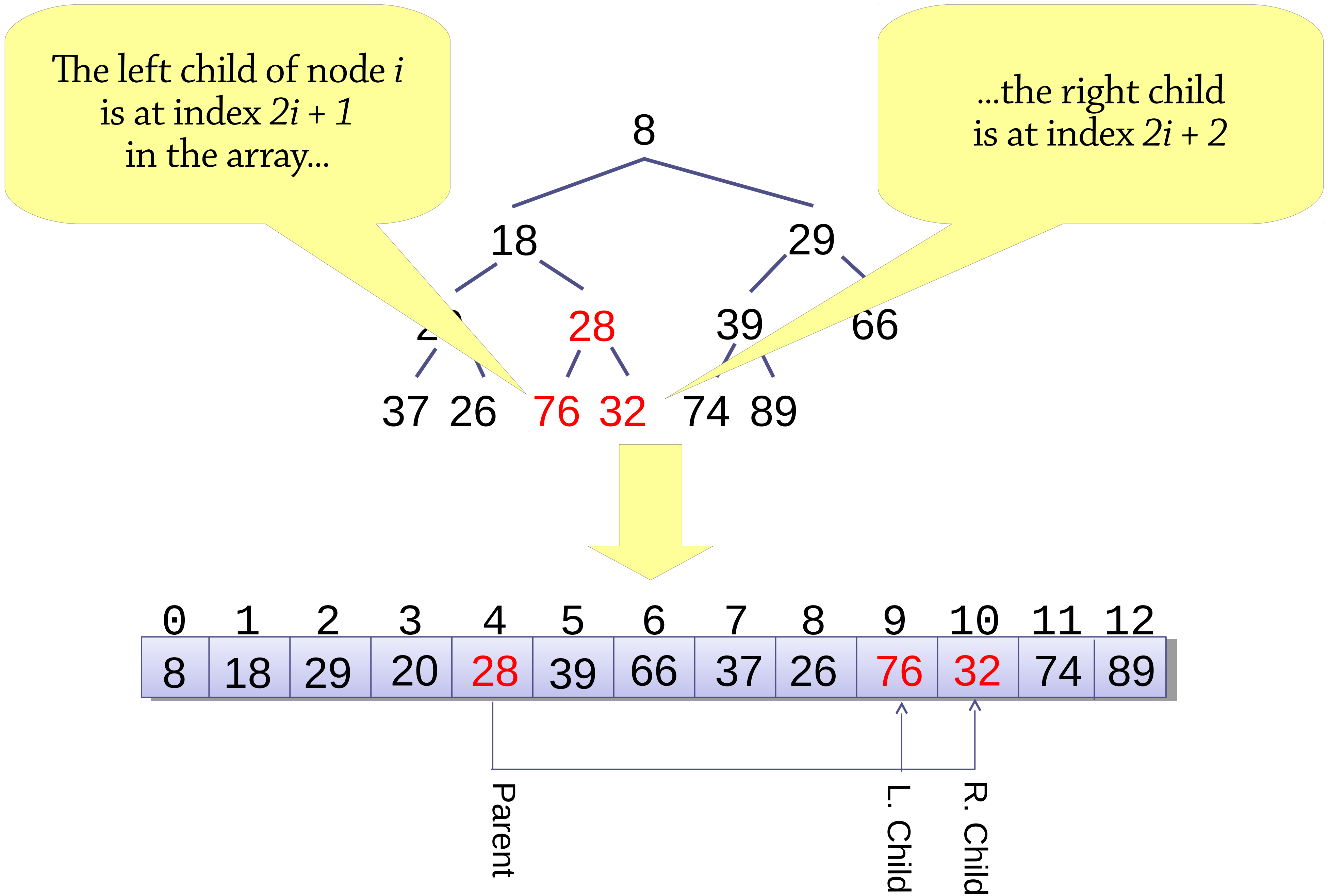
Child positions



Child positions

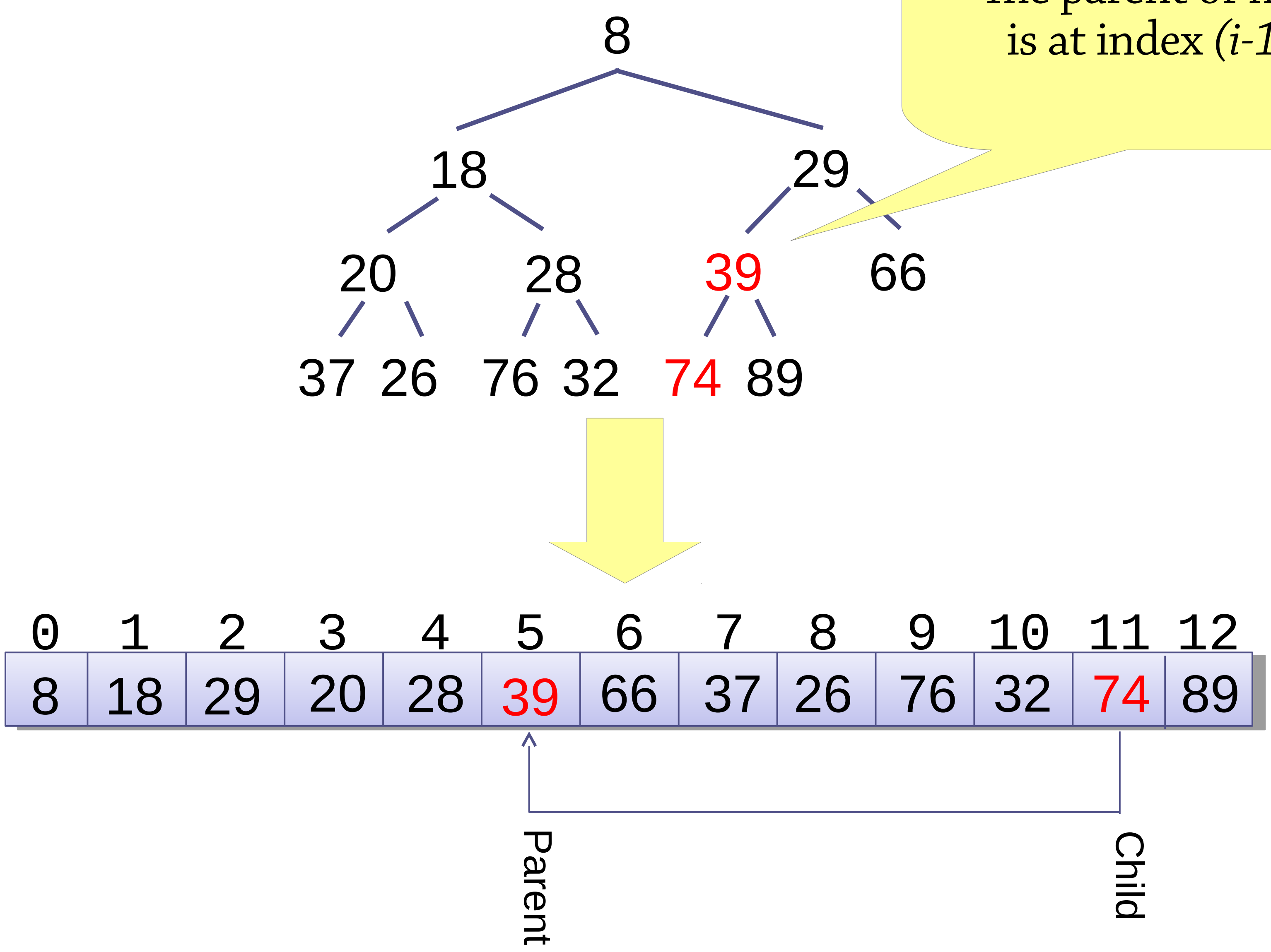


Child positions



Parent position

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



Reminder: inserting into a binary heap

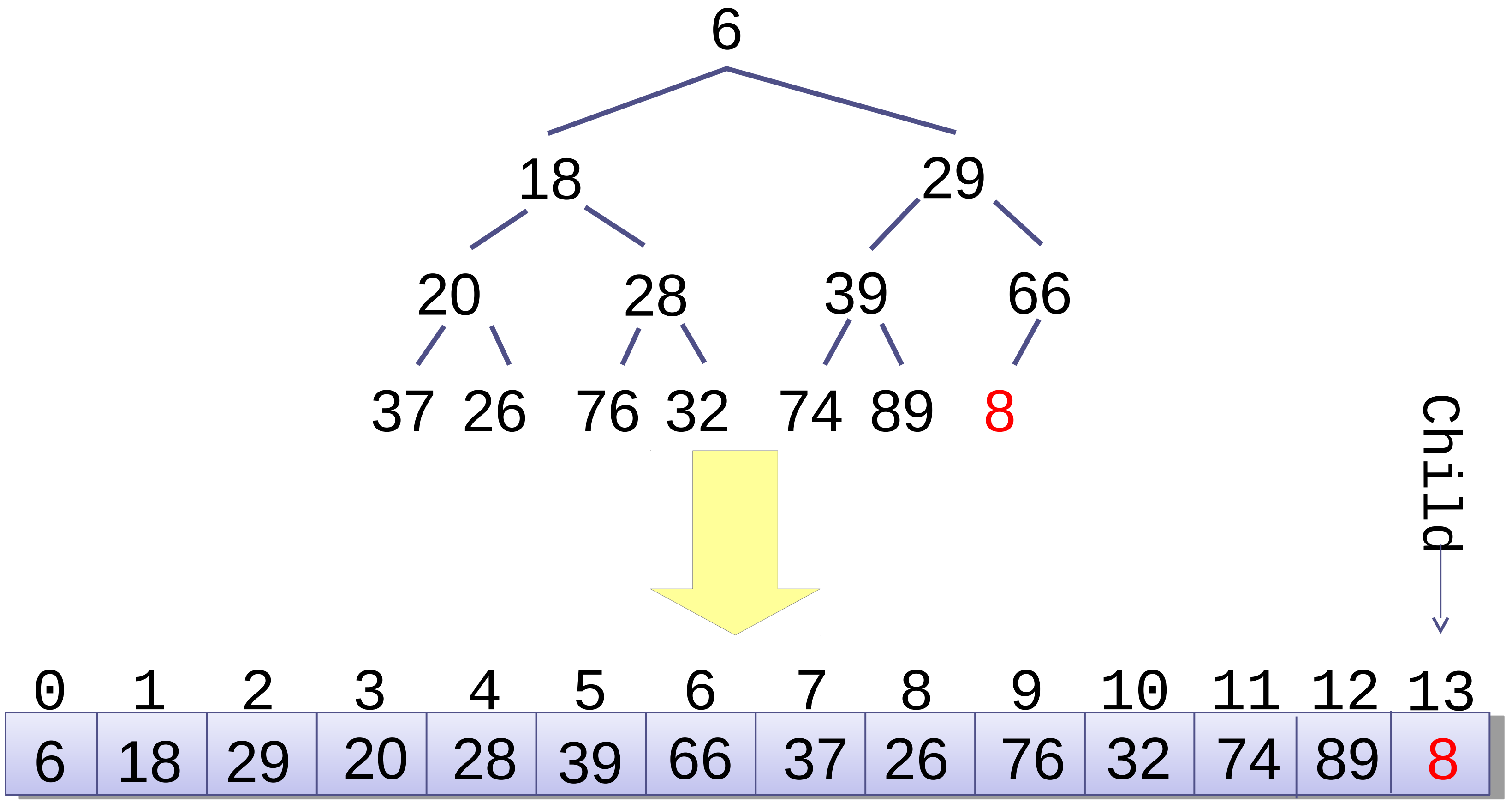
To insert an element into a binary heap:

- 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!

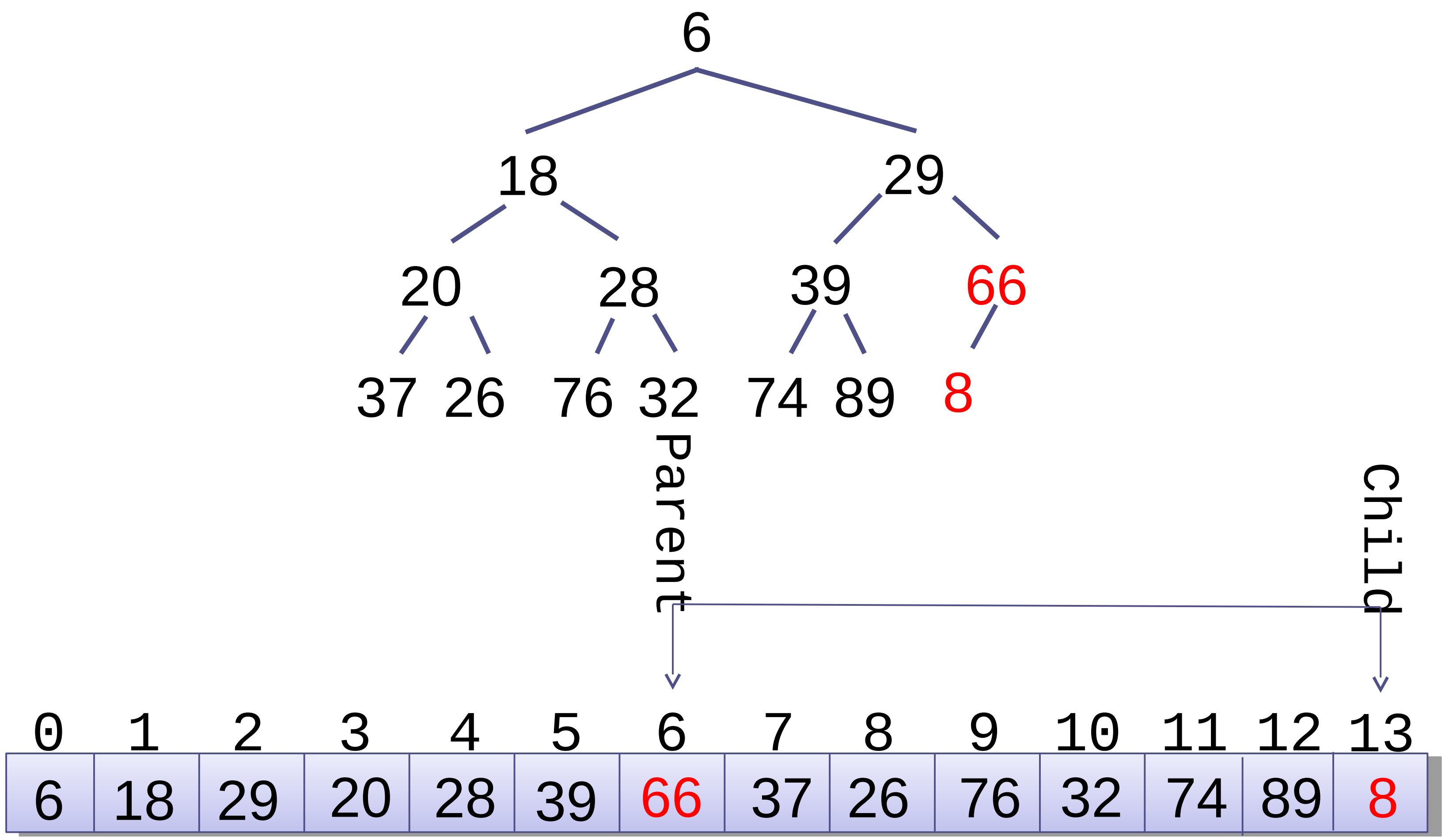
Inserting into a binary heap

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



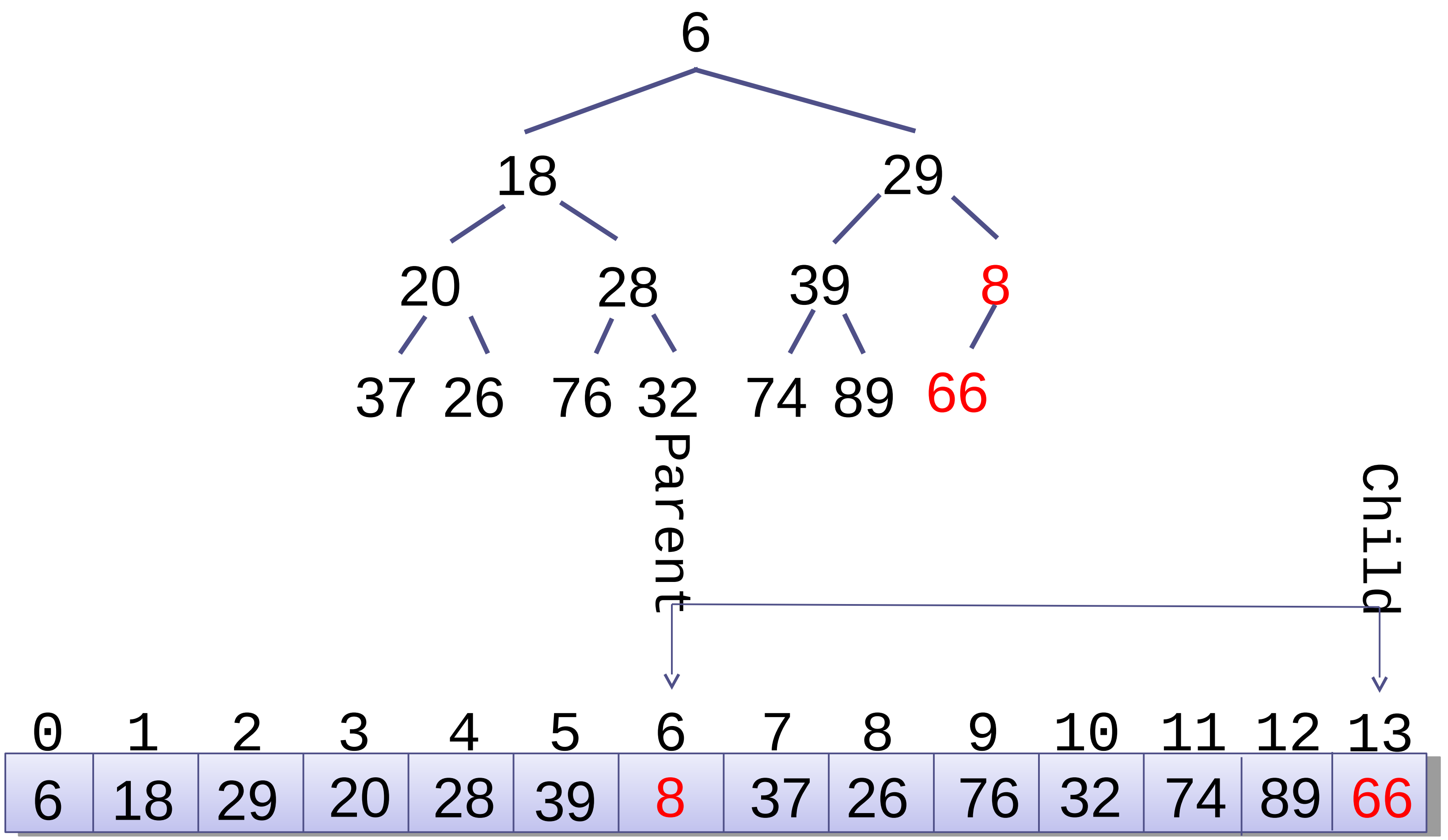
Inserting into a binary heap

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



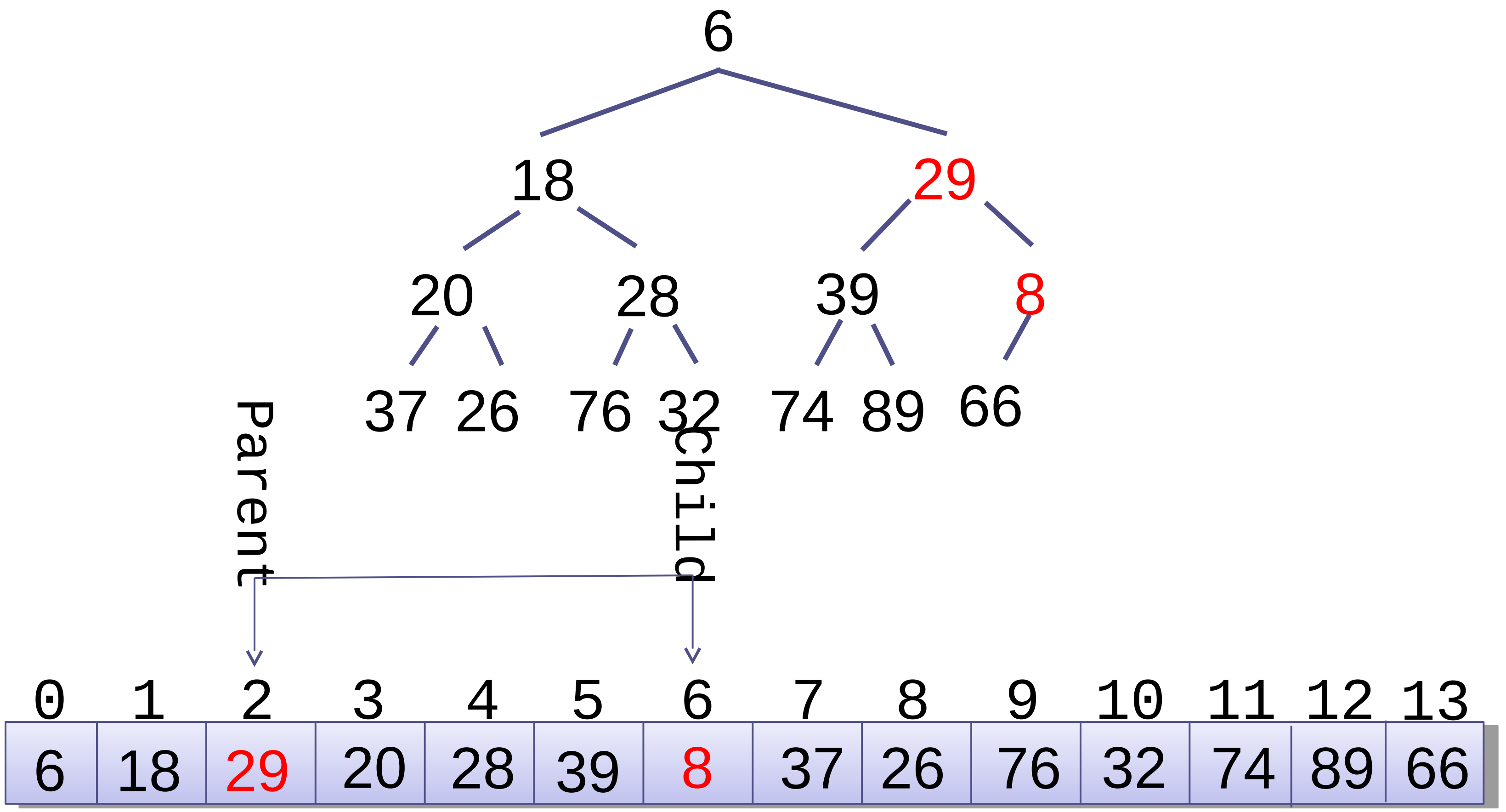
Inserting into a binary heap

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



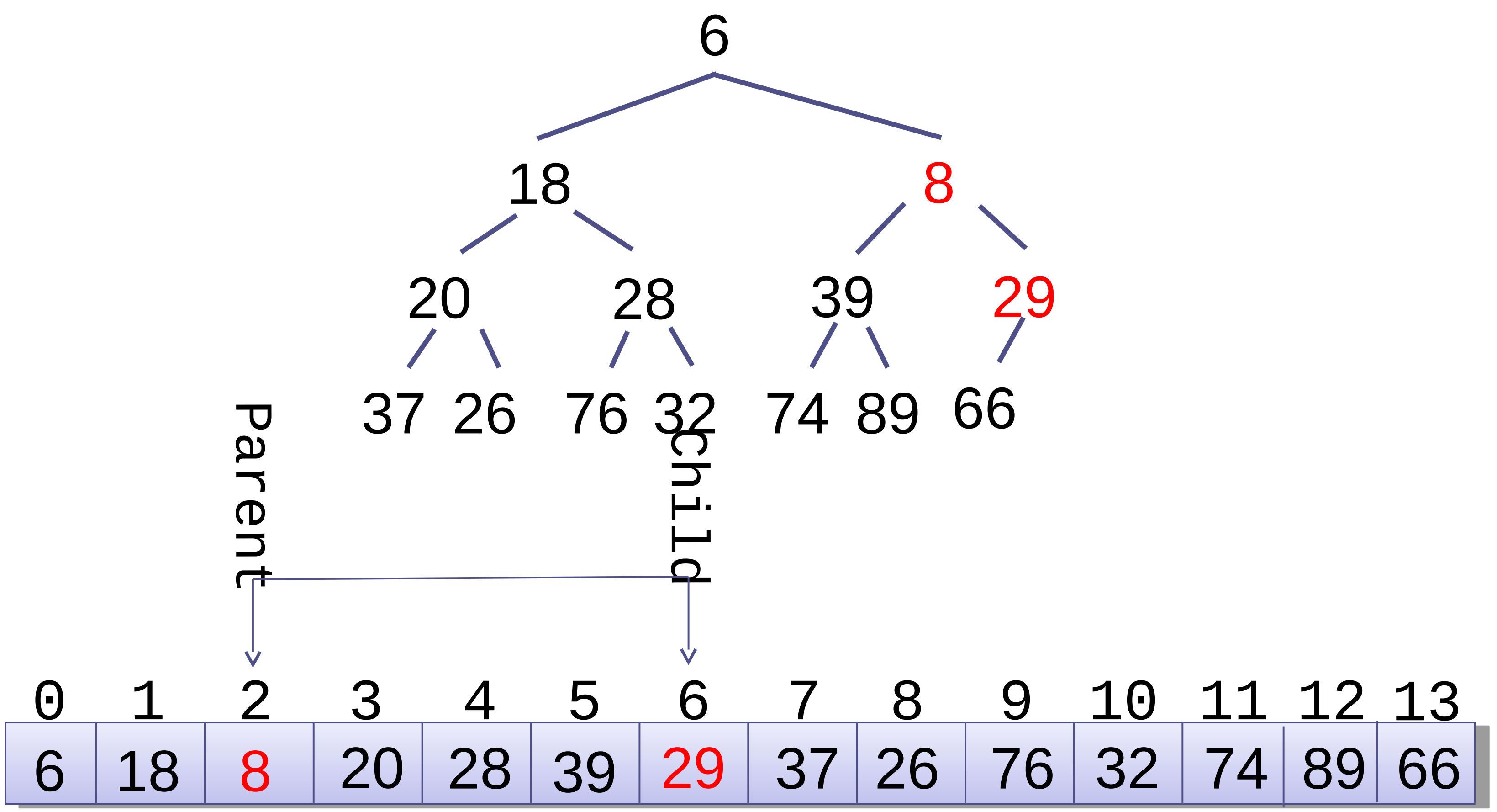
Inserting into a binary heap

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!)

Heap sort

Binary heaps can be used to sort arrays.

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!

Using a binary heap like this is called Heap sort. It's complexity is O(n log n).

Heap sort

- Heap sort can be done by using the original array as starting point and apply the *build-heap* algorithm.
- Build-heap actually runs in O(n), but repeatedly extracting the minimum element makes the overall complexity O(n log n) anyway.
- If a max-heap is built instead of a min-heap, the minum element can be extracted and moved to the end in each step. This makes heap sort in-place.

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