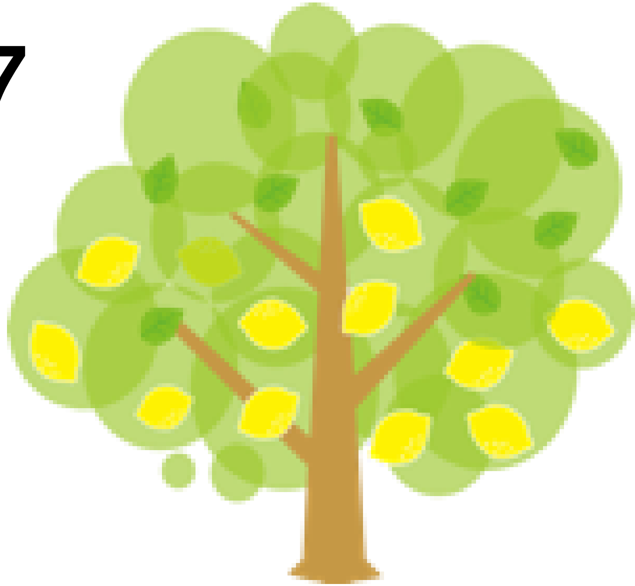
Lecture 3: Priority Queues, and a Tree Grows in 247



These slides include material originally prepared by Dr.Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- Lab 1 due Friday 2/8 at 11:59 PM
    - Turn in via Gradescope
    - Math hint:

$$a^{\log_b n} = n^{\log_b a}$$

- Lab 3 out this Wednesday
    - Has three parts: pre-lab (due 2/12), coding, and write-up parts (due 2/15)

Overview

- What is a Priority Queue
  - ADT
  - Applications
- Some not so great implementations (which you'll explore in Studio 3)
  - Lists
  - Arrays

Overview

- What is a Priority Queue
  - ADT
  - Applications
- Some not so great implementations (which you'll explore in Studio 3)
  - Lists
  - Arrays
- Trees
- Priority Queue using trees
- Using arrays to simulate trees
  - Implementation of this is Lab 3

ADTs from Last Time

- The data structures we reviewed last time (queues, stacks) track the positions of their elements.
  - “Add to the tail”/ “Remove from the head” [queues]
  - “Add to the top”/ “Remove from the top” [stacks]
- Elements themselves were completely generic – we neither knew nor cared about their properties.

Working With Ordered Data

- But let's suppose we have data that is *ordered* (e.g. integers).

- Given a collection of such data, we may want to ask questions that depend on the order of the elements.

Working With Ordered Data

- But let's suppose we have data that is *ordered* (e.g. integers).

- Given a collection of such data, we may want to ask questions that depend on the order of the elements.

- ***Challenge:*** can we efficiently answer these questions when the collection is changing dynamically?

Example: Auto Repair

- Garage receives a stream of cars needing repairs.
- Each repair job comes with a deadline.
- *Cars may not show up in order of deadlines.*

Example: Auto Repair

- Garage receives a stream of cars needing repairs.

- Each repair job comes with a deadline.

1:00 PM

Example: Auto Repair

- Garage receives a stream of cars needing repairs.
- Each repair job comes with a deadline.

1:00 PM

5:00 PM

Example: Auto Repair

- Garage receives a stream of cars needing repairs.
- Each repair job comes with a deadline.

1:00 PM

5:00 PM

11:00 AM!!!

What Do We Want?

- **Query**: at any time, which car needs to be ready first? (earliest deadline)
- **Insertion**: new cars can show up at any time, with any deadline.
- **Update**: when we repair the car with the earliest deadline, which car has next earliest deadline?

More Abstractly…

- Maintain a collection of ordered values [e.g. deadlines]
- Values can be inserted in any order
- At any time, may remove smallest value
- Want to maintain O(1) query time for smallest value

More Abstractly…

- Maintain a “min-first priority queue” PQ
- PQ.insert(v) – insert element v
- PQ.extractMin() – extract and return minimum element
- PQ.peekMin() – must be constant-time

A Few More Details

- PQ has a *fixed maximum size* [e.g. size of garage]

- An item's value *might decrease* while it is in PQ
  [e.g. a customer now wants their car back sooner]

Assumptions:

1. Items are not known until they are inserted
2. An item's value *cannot increase* while it is in PQ

What PQ Methods Might Look Like in Java

- instantiation: **`PriorityQueue<T>(int size)`**
  - The queue has a bounded size that is specified upon creation
- insertion into the PQ: **`Decreaser<T> insert(T thing)`**
  - The returned “Decreaser” object is often called a *handle*
  - **`Decreaser<T>`** allows outside activity to decrease the value of inserted **`thing`**
- is the PQ empty? **`boolean isEmpty()`**
- remove and return the currently smallest **`T: extractMin()`**
- inspect but do not remove the currently smallest **`T: peekMin()`**

What PQ Methods Might Look Like in Java

- instantiation: `PriorityQueue<T>(int size)`
  - The queue has a bounded size that is specified upon creation
- insertion into the PQ: `Decreaser<T> insert(T thing)`
  - The returned "Decreaser" object is often called a *handle*
  - `Decreaser<T>` allows outside activity to decrease the value of inserted `thing`
- is the PQ empty? `boolean isEmpty()`
- remove and return the currently smallest `T: extractMin()`
- inspect but do not remove the currently smallest `T: peekMin()`

We could just as well define a max-first PQ that maintains the largest element. The book makes this choice.

Further Notes on Java Impl (See Lab 3)

- What about **`Decreaser<T>`**?
  - **`T getValue()`** to get at the current value of this **`thing`**
  - **`void decrease(T newvalue)`**
  - We require that **`newvalue`** be no greater than the current value for the affected item
  - Why is this an operation on an object (**`Decreaser`**) outside of the PQ, instead of, say, PQ.**`decreaseItem(T which, T newvalue)`** ?

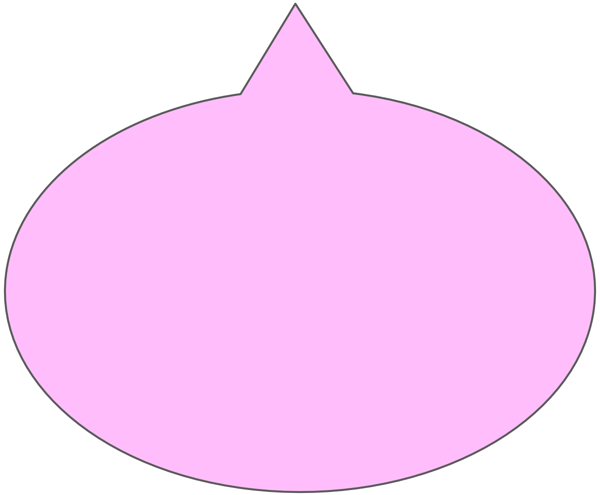
Further Notes on Java Impl (See Lab 3)

- What about **`Decreaser<T>`**?
  - **`T getValue()`** to get at the current value of this **`thing`**
  - **`void decrease(T newvalue)`**
  - We require that **`newvalue`** be no greater than the current value for the affected item
  - Why is this an operation on an object (**`Decreaser`**) outside of the PQ, instead of, say, PQ.**`decreaseItem(T which, T newvalue)`** ?

In the alternative, how long could it take to locate "which"?
Is the entry "which" unique?

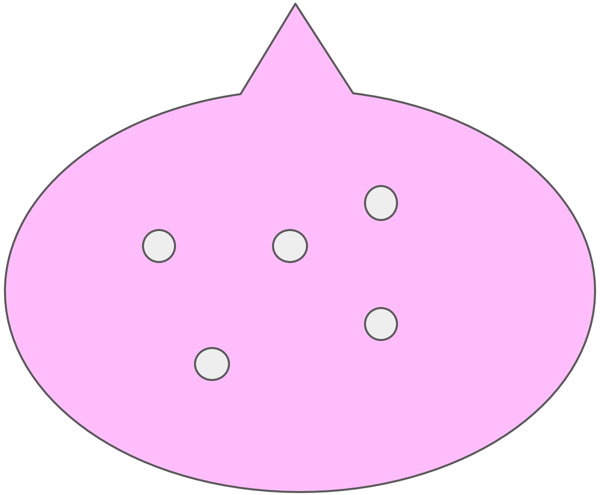
Example of a Priority Queue

```
new PriorityQueue(5)
```



Example of a Priority Queue

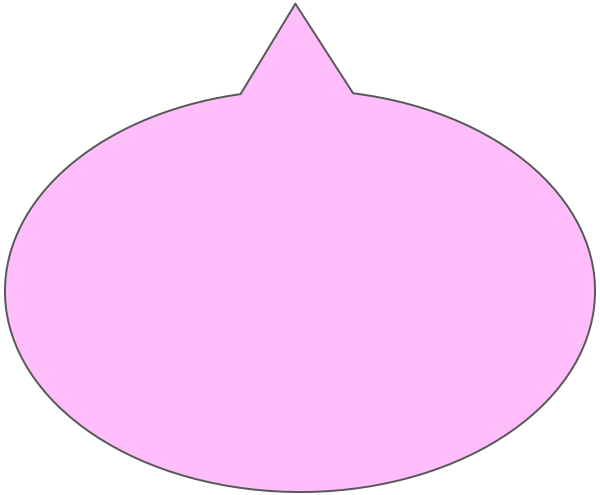
```
new PriorityQueue(5)
```



- Can hold up to 5 elements

Example of a Priority Queue

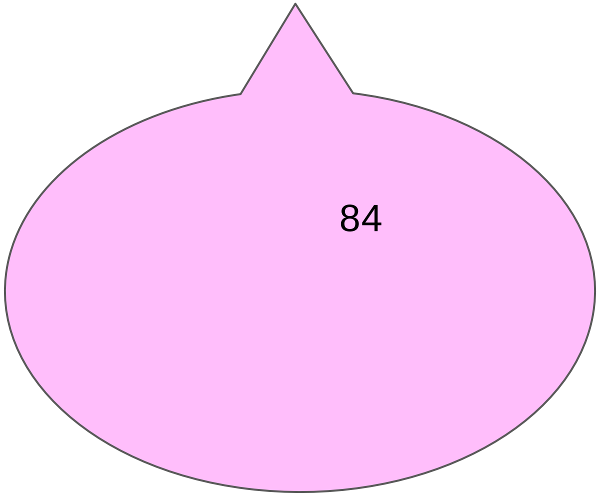
```
new PriorityQueue(5)
```



- Can hold up to 5 elements
- Is initially empty

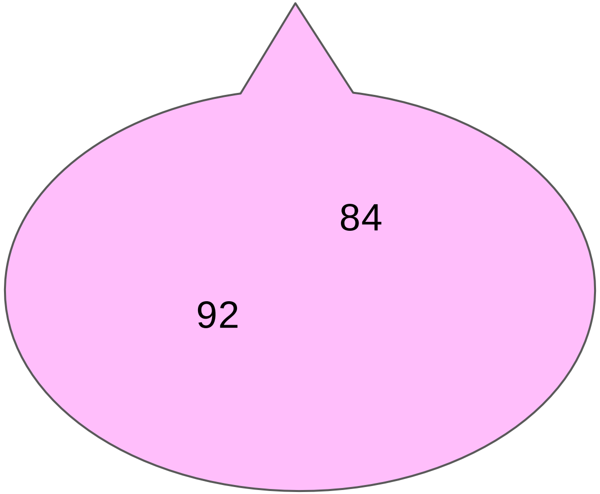
Example of a Priority Queue

```
insert(84)
```



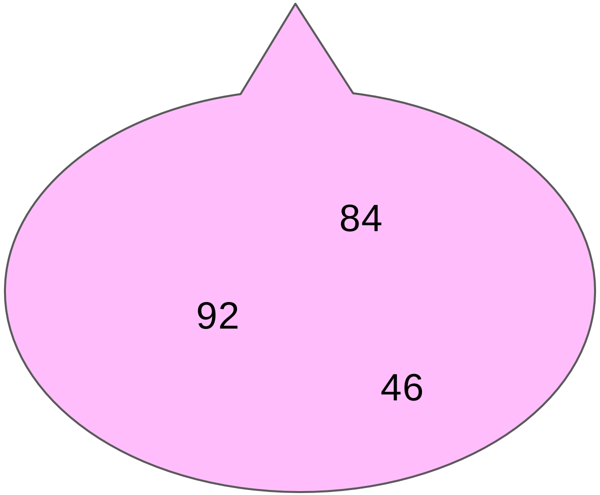
Example of a Priority Queue

```
insert(92)
```



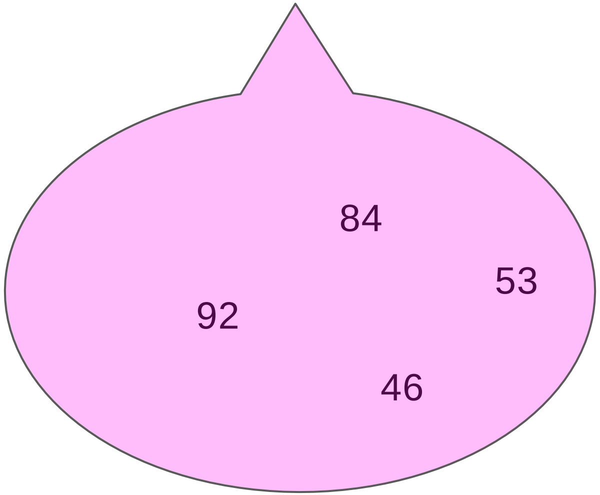
Example of a Priority Queue

```
insert(46)
```



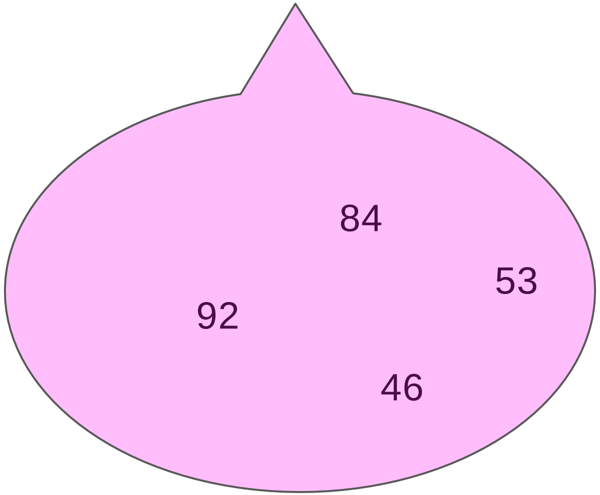
Example of a Priority Queue

`insert(53)`

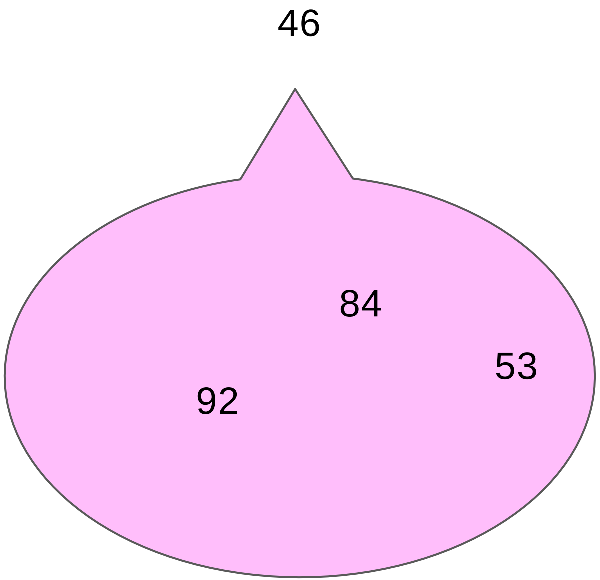


Example of a Priority Queue

```
extractMin()
```



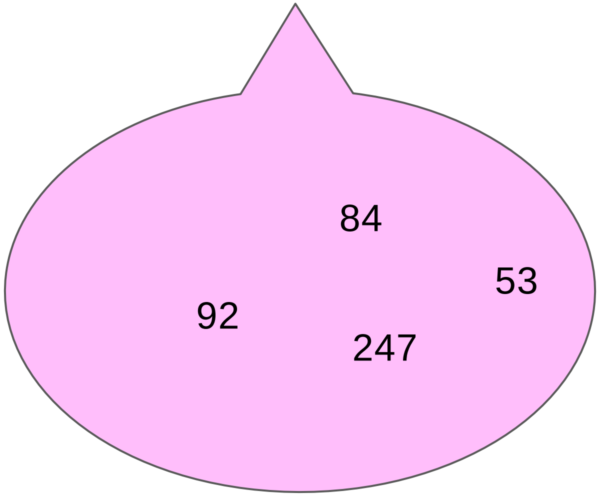
Example of a Priority Queue



`extractMin()`

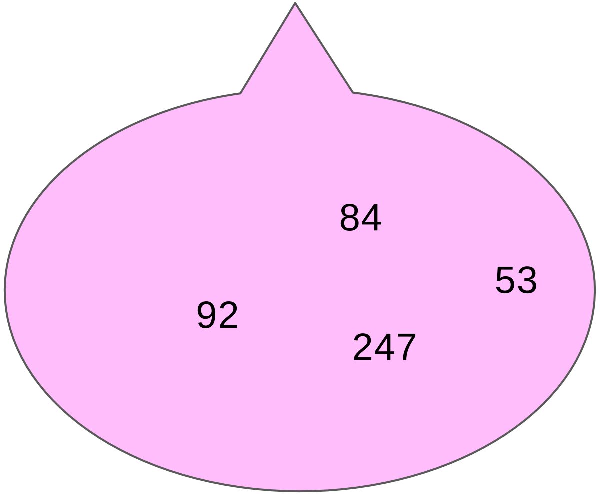
Example of a Priority Queue

```
insert(247)
```

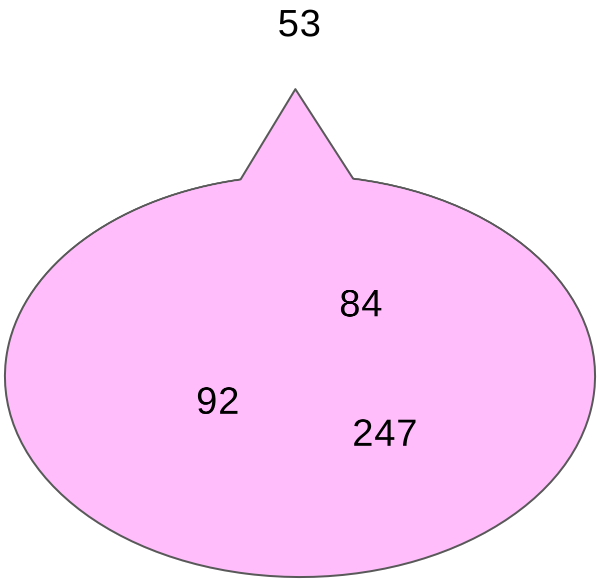


Example of a Priority Queue

```
extractMin()
```

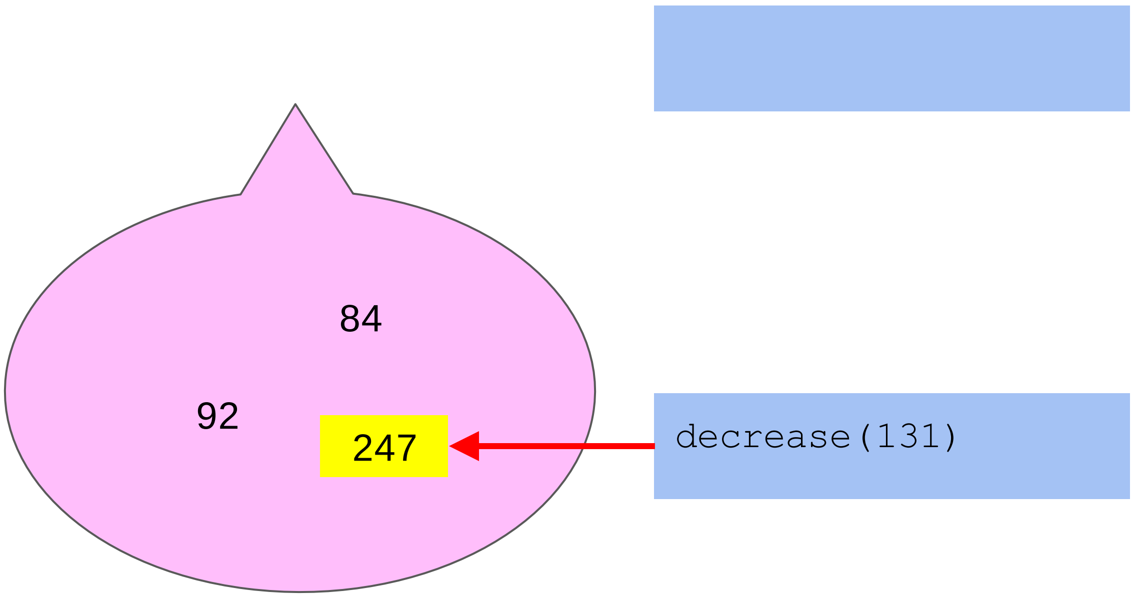


Example of a Priority Queue



`extractMin()`

Example of a Priority Queue



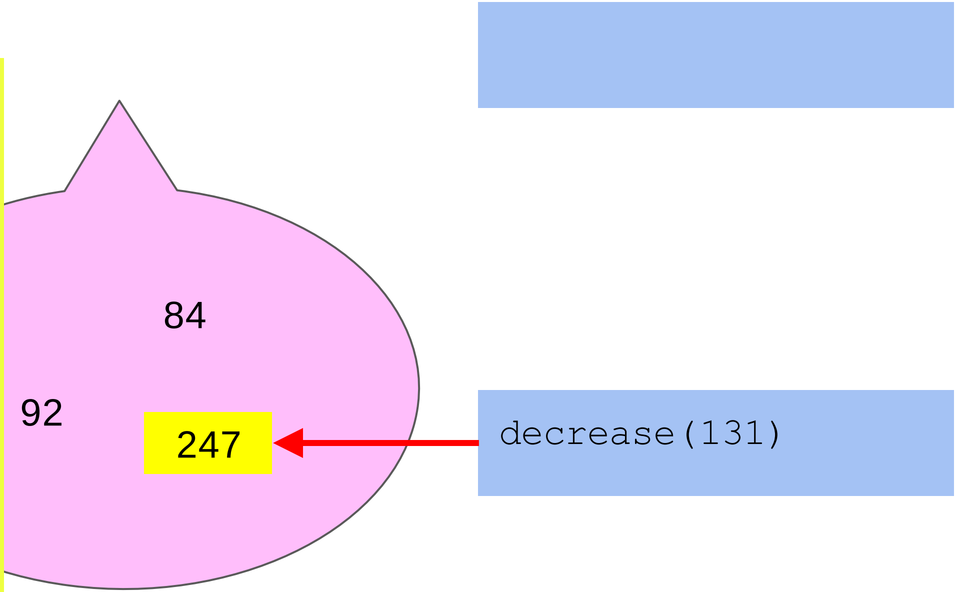
Example of a Priority Queue

We decrease 247 using its *handle*, the `Decreaser` object, which has a direct reference to the 247 entry.

Why do we need the `Decreaser`?

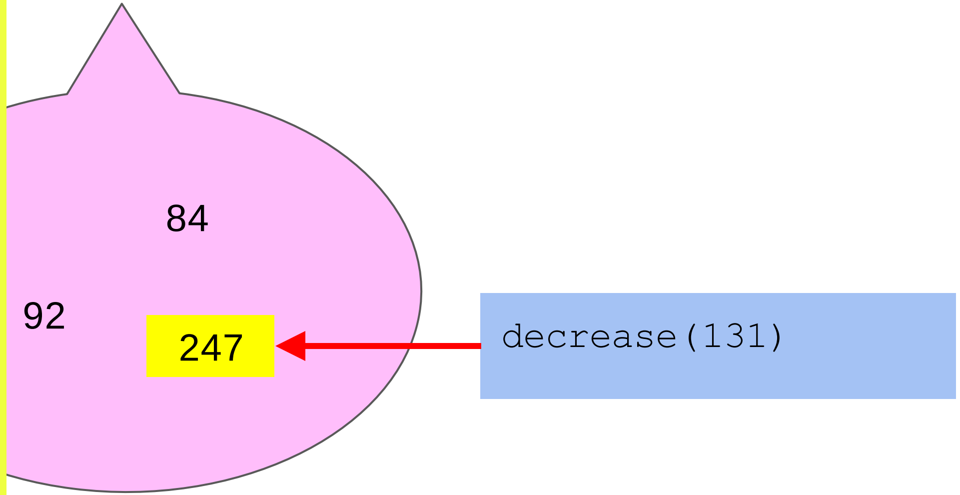
It references 247 directly, so we can decrease the value in the Priority Queue without having to *find* 247 first in the Priority Queue.

This avoids a search for 247, which might require looking at every entry, taking O(n) time for a Priority Queue of n elements.



Example of a Priority Queue

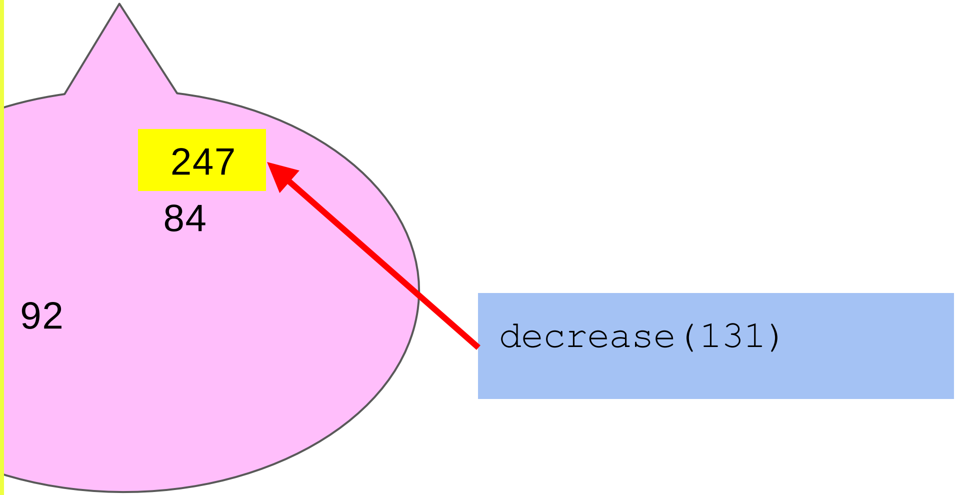
We have not yet considered the implementation, but we will soon see that the 247 entry may move around in the data structure we use.



Example of a Priority Queue

We have not yet considered the implementation, but we will soon see that the 247 entry may move around in the data structure we use.

As it does, the `Decreaser` continues to follow it, no matter where it goes.



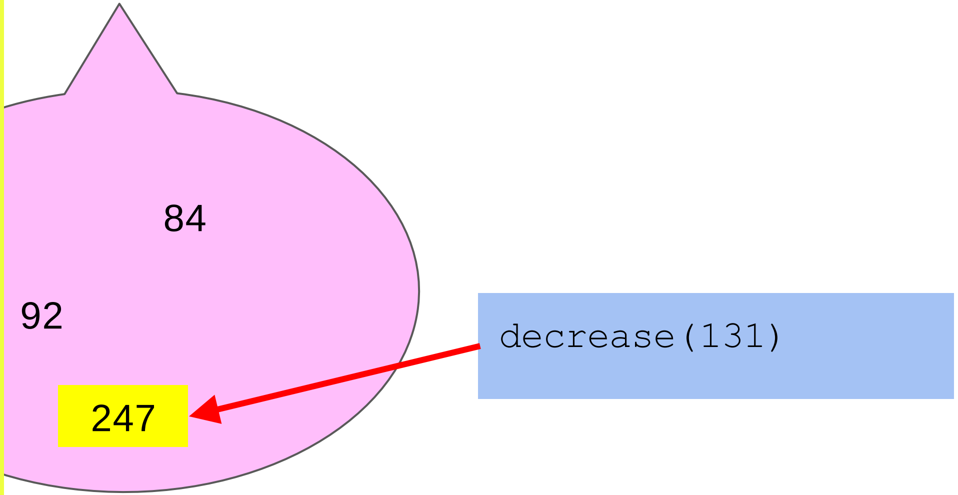
Example of a Priority Queue

We have not yet considered the implementation, but we will soon see that the 247 entry may move around in the data structure we use.

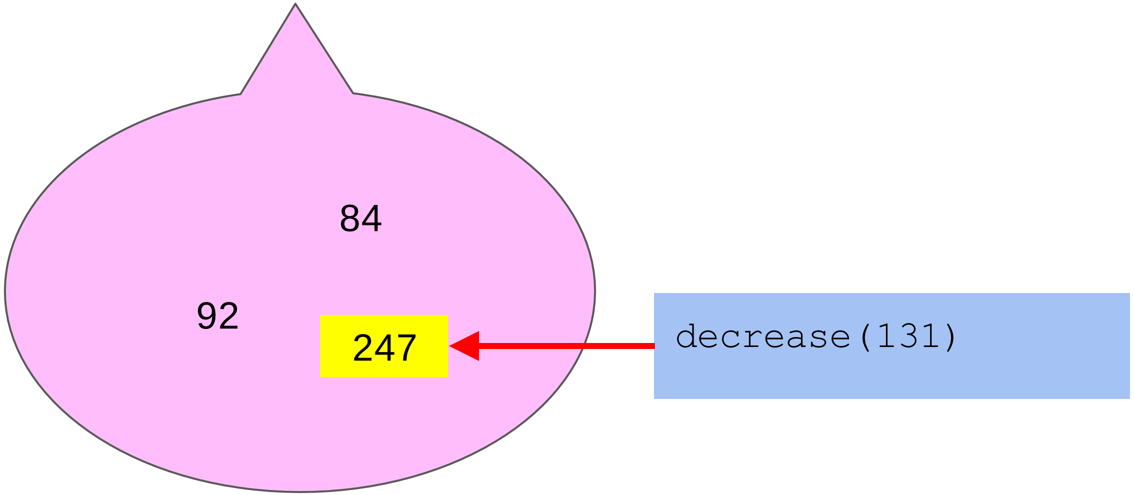
As it does, the `Decreaser` continues to follow it, no matter where it goes.

The data structure returns an entry's unique `Decreaser` object as the result of insertion.

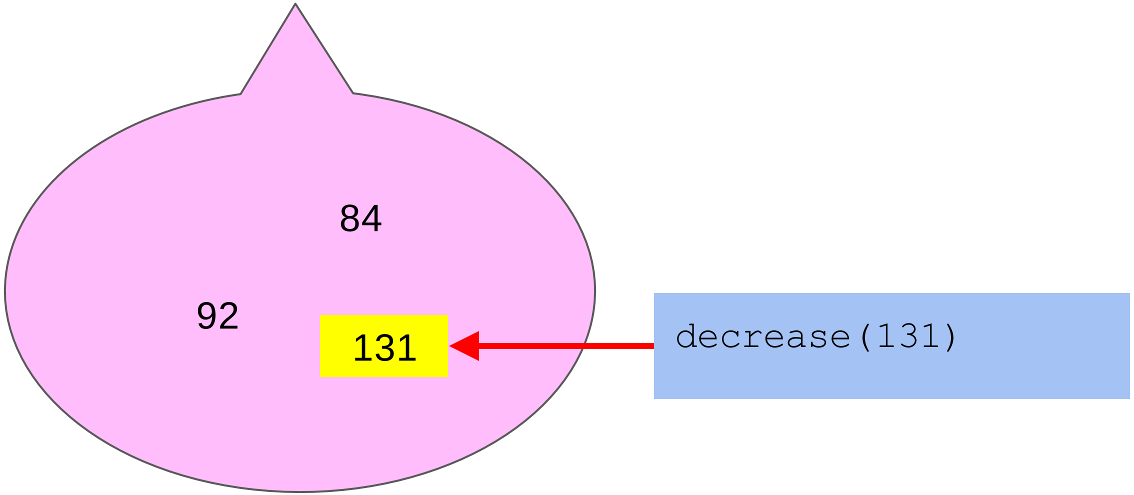
This provides a fast method for decreasing the value, as shown on the next slides.



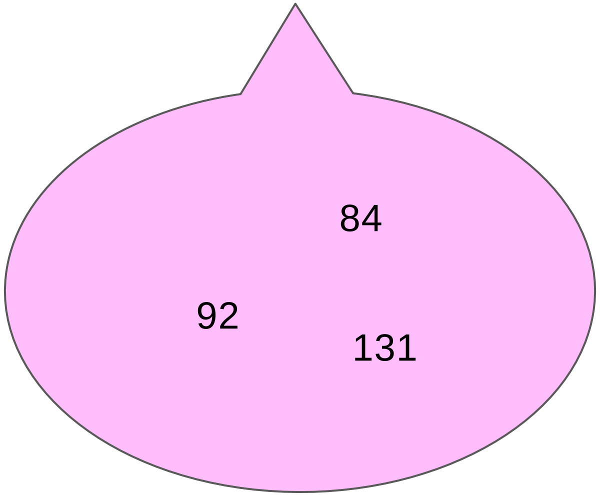
Example of a Priority Queue



Example of a Priority Queue

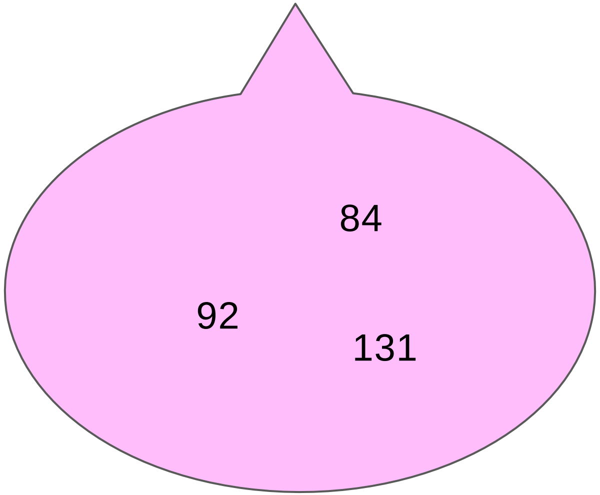


Example of a Priority Queue



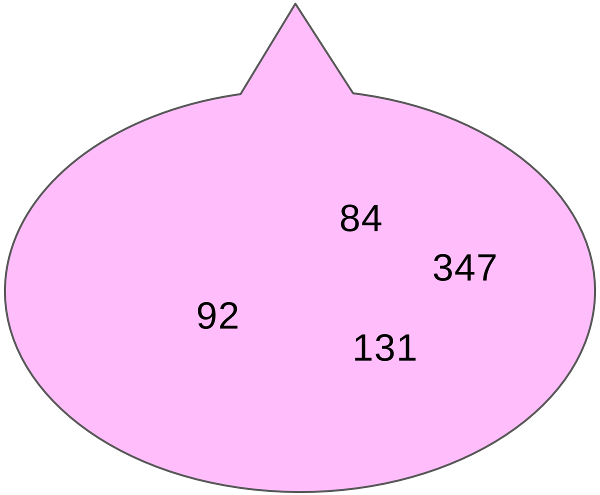
Example of a Priority Queue

`insert(347)`

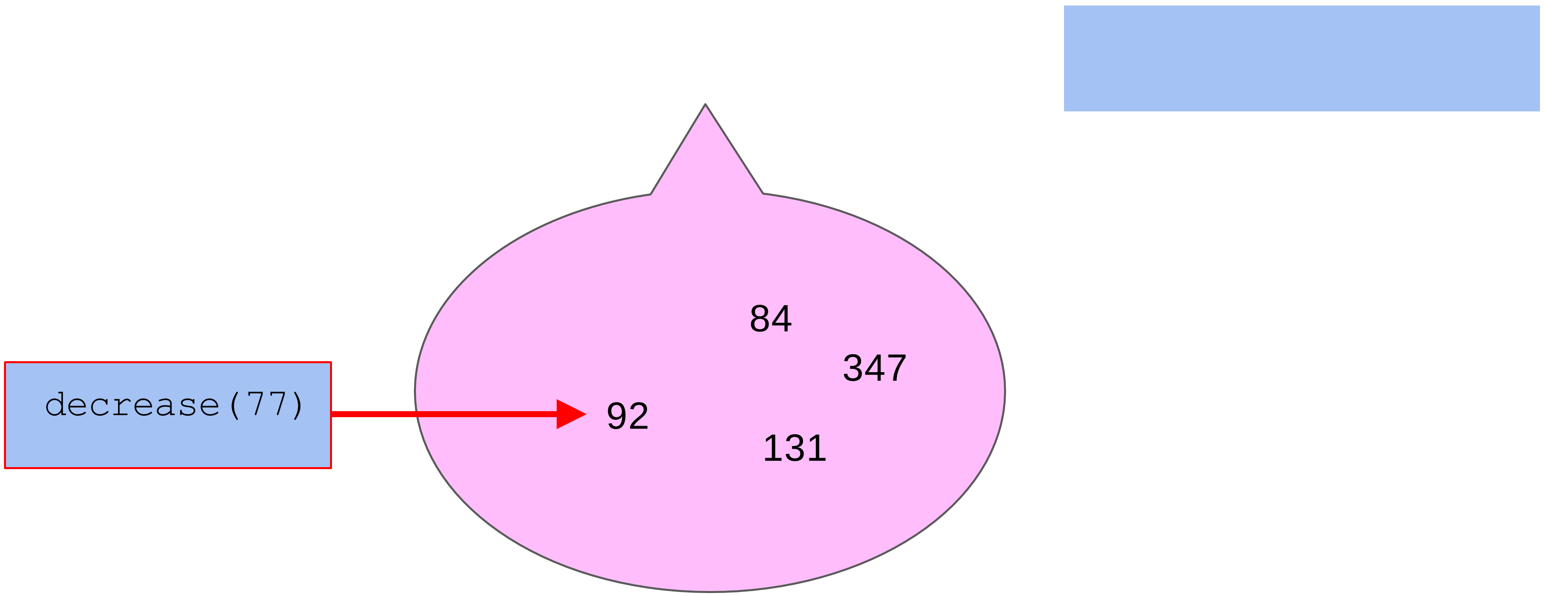


Example of a Priority Queue

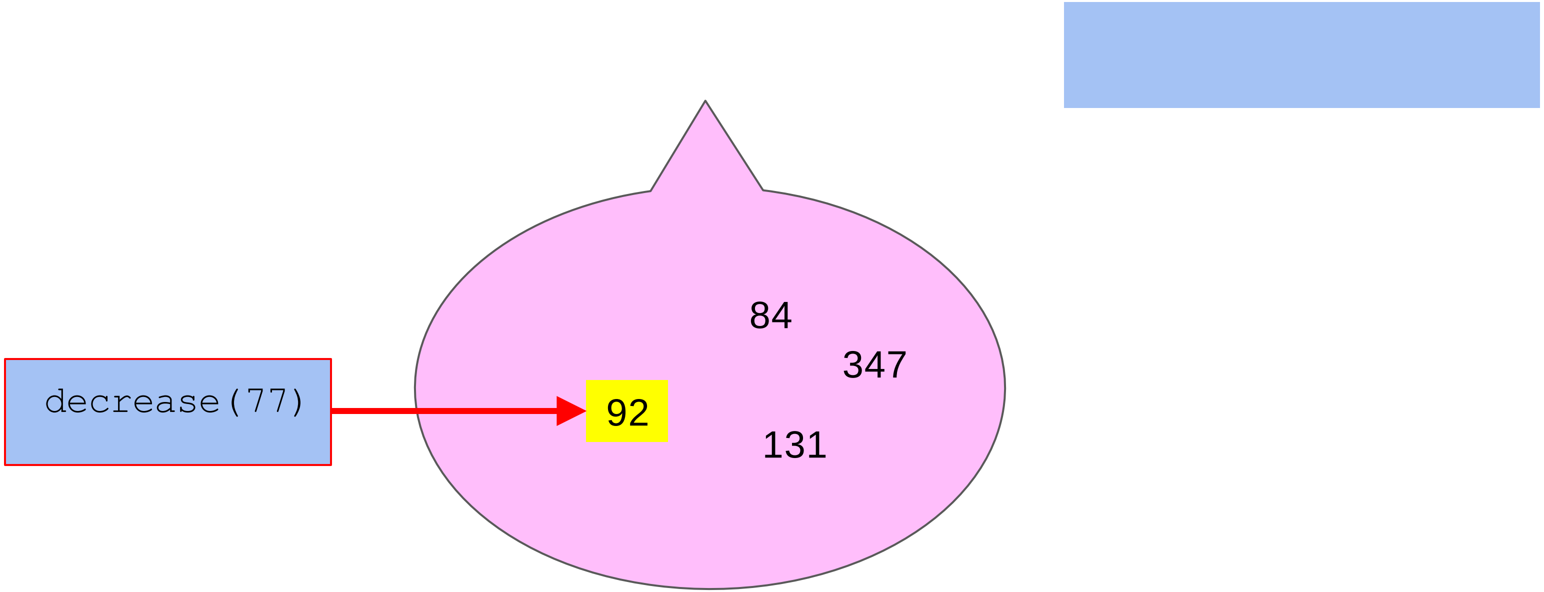
```
insert(347)
```



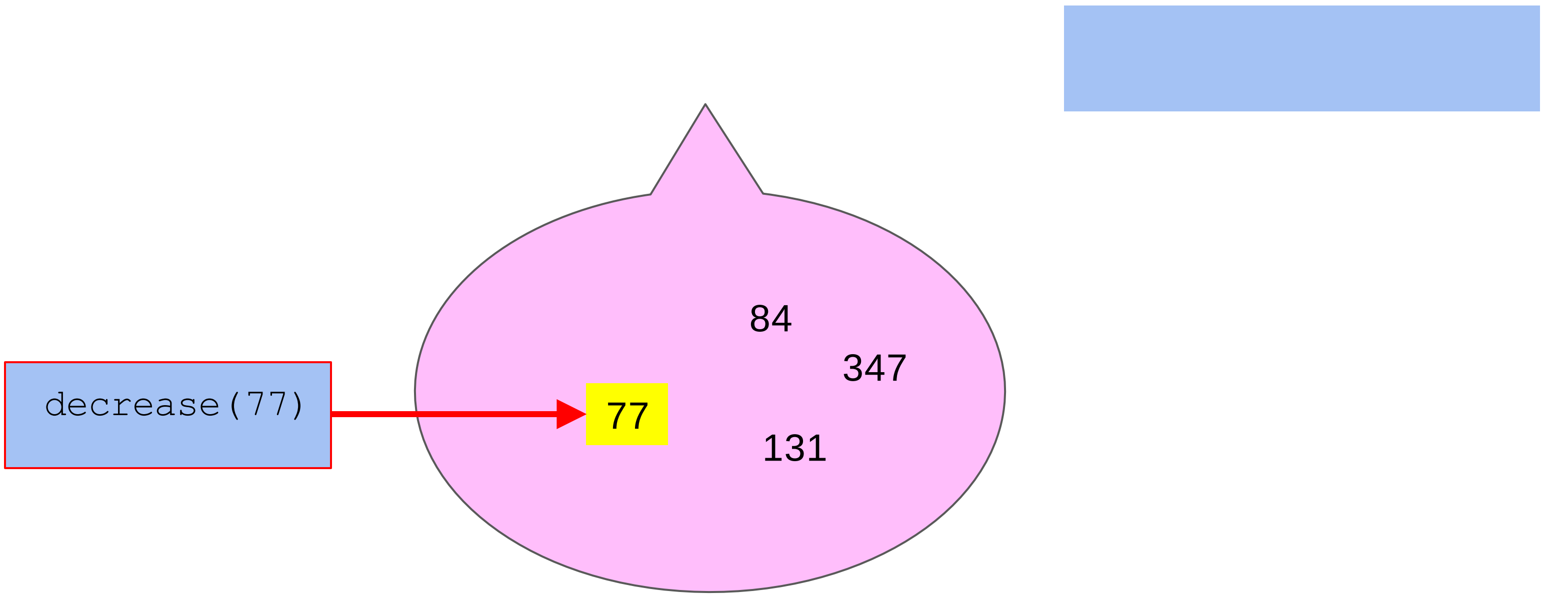
Example of a Priority Queue



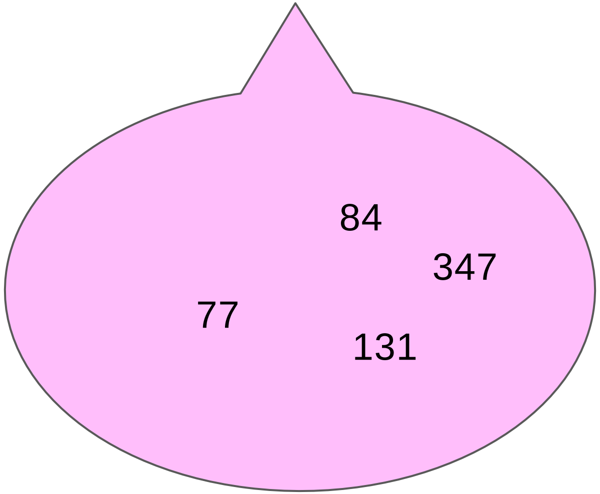
Example of a Priority Queue



Example of a Priority Queue

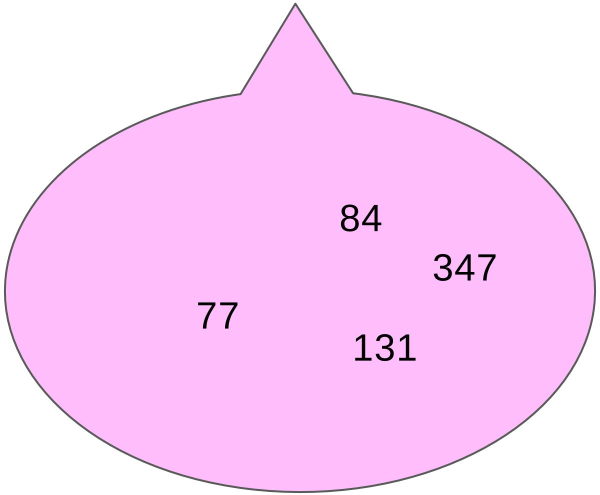


Example of a Priority Queue

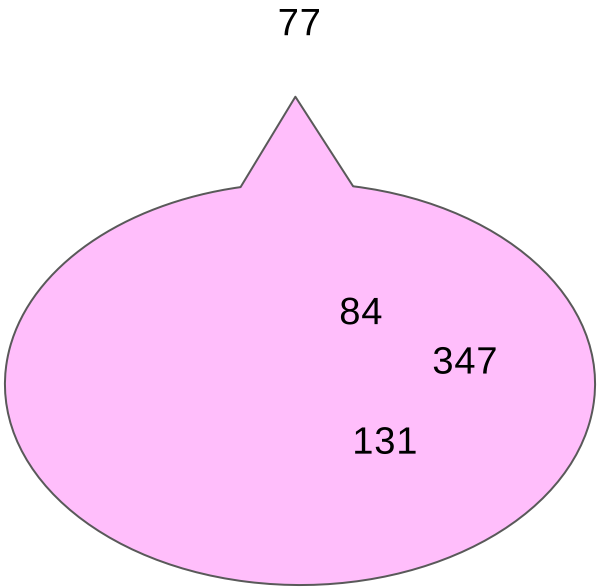


Example of a Priority Queue

```
extractMin()
```

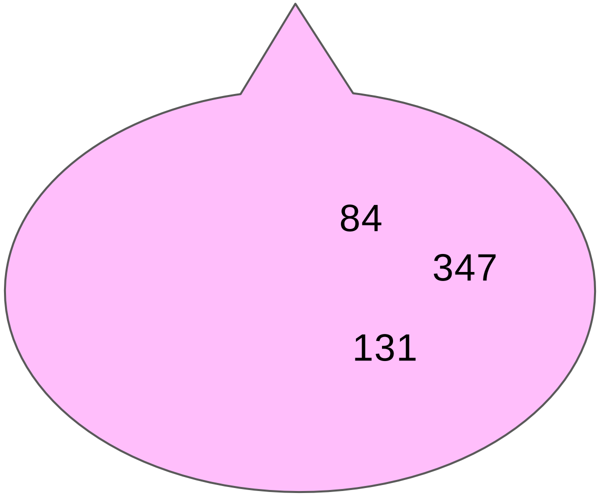


Example of a Priority Queue



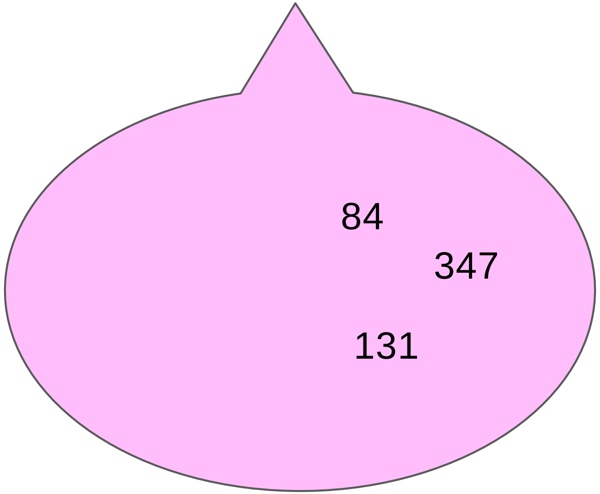

```
extractMin()
```

Example of a Priority Queue



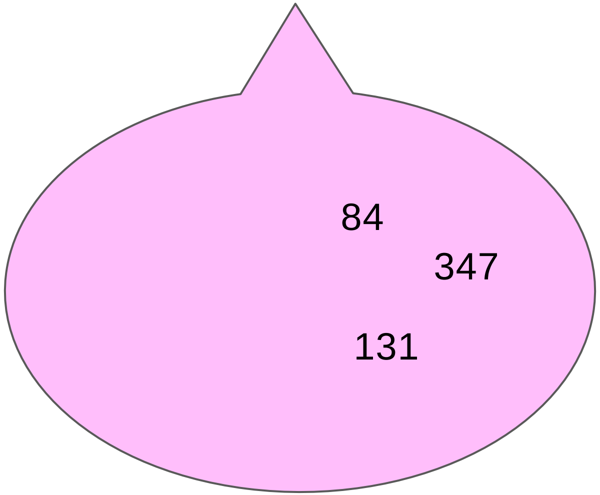
Applications

- Scheduling Tasks with Priorities
    - Find/Handle highest-priority task first
    - E.g. in computer operating systems
- Searching for the Best Solution
    - Add solutions to PQ as they are found
    - At any time, can query/remove the optimum
    - Cost of solutions may decrease over time
    - (e.g. shortest path to each node in a graph)



Applications

- Scheduling Tasks with Priorities
    - Find/Handle highest-priority task first
    - E.g. in computer operating systems
- Searching for the Best Solution
    - Add solutions to PQ as they are found
    - At any time, can query/remove the optimum
    - Cost of solutions may decrease over time
    - (e.g. shortest path to each node in a graph)
- *Can you think of others?*



Performance Goals for Priority Queue

- Let *n* be the size of the queue at a given time.
- peekMin() should be constant-time
- Want "nice" complexity for insert, decrease, extractMin as fcn of n.
- Ideally, all these operations should take time **sub-linear in *n*** (i.e. o(n))

Ideas? (Analyzed in Studio 3)

- Unsorted linked list?

PQ ops needed:

$o(n) \left\{ \begin{array}{l} \text{insert(v)} \\ \text{decrease(item, k)} \\ \text{extractMin()} \end{array} \right.$

$\Theta(1)$ peekMin()

Ideas?

- Unsorted linked list?
  - [high cost to find new min on extractMin()]
- Sorted linked list?

PQ ops needed:

$$o(n)\left\{\begin{array}{l}\text{insert(v)}\\ \text{decrease(item, k)}\\ \text{extractMin()}\end{array}\right.$$

$\Theta(1)$ peekMin()

Ideas?

- Unsorted linked list?
    - [high cost to find new min on extractMin()]
- Sorted linked list?
    - [high cost to insert]
- Unsorted array?

PQ ops needed:

$o(n)$ { insert(v)
decrease(item, k)
extractMin() }

$\Theta(1)$ peekMin()

Ideas?

PQ ops needed:

o(n) { insert(v)
decrease(item, k)
extractMin() }

Θ(1) peekMin()

- Unsorted linked list?
  - [high cost to find new min on extractMin()]
- Sorted linked list?
  - [high cost to insert]
- Unsorted array?
  - [just as bad as unsorted list]

PQ ops needed:

$o(n)\left\{\begin{array}{l}\text{insert(v)}\\ \text{decrease(item, k)}\\ \text{extractMin()}\end{array}\right.$

$\Theta(1)$ peekMin()

Ideas?

- Unsorted linked list?
- Sorted linked list?
- Unsorted array?

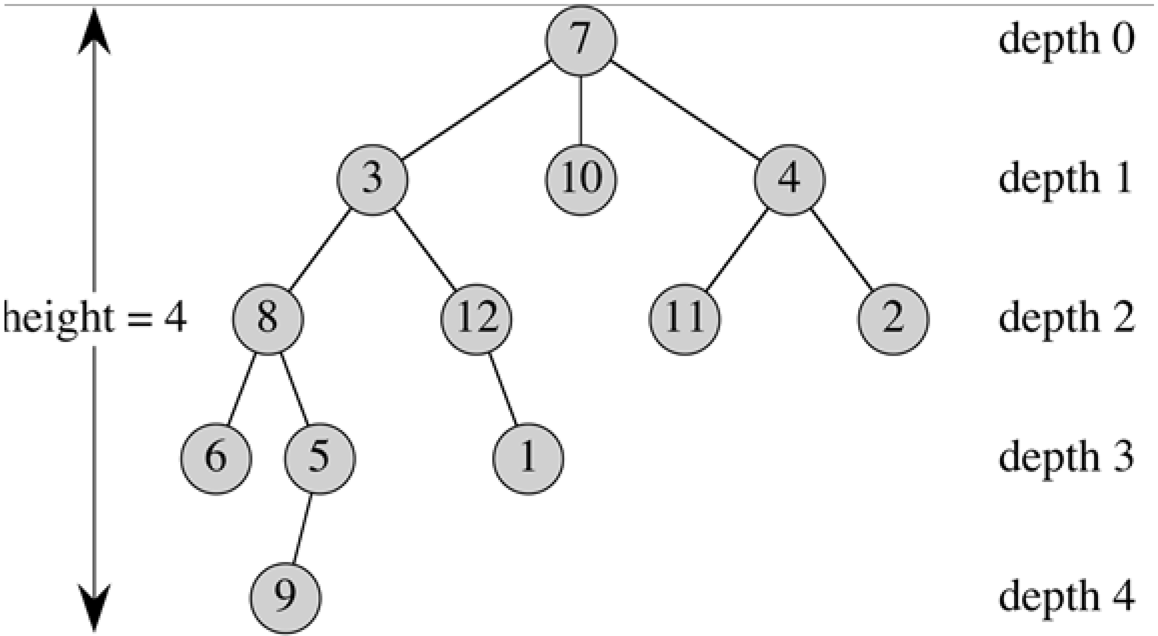
Argh – we need a new data structure to meet better-than-Θ(n) performance goals for both insertion and extraction!

A Brief Diversion: Trees

- Lists are a one-dimensional data structure
  - One-dimensional connections (forward, back)
- We can use them to implement other data structures
  - Queue
  - Stack
- We now consider trees
  - These are two-dimensional
  - Movement up or down
  - Movement left or right
- We will use trees to implement several interesting data structures

Trees

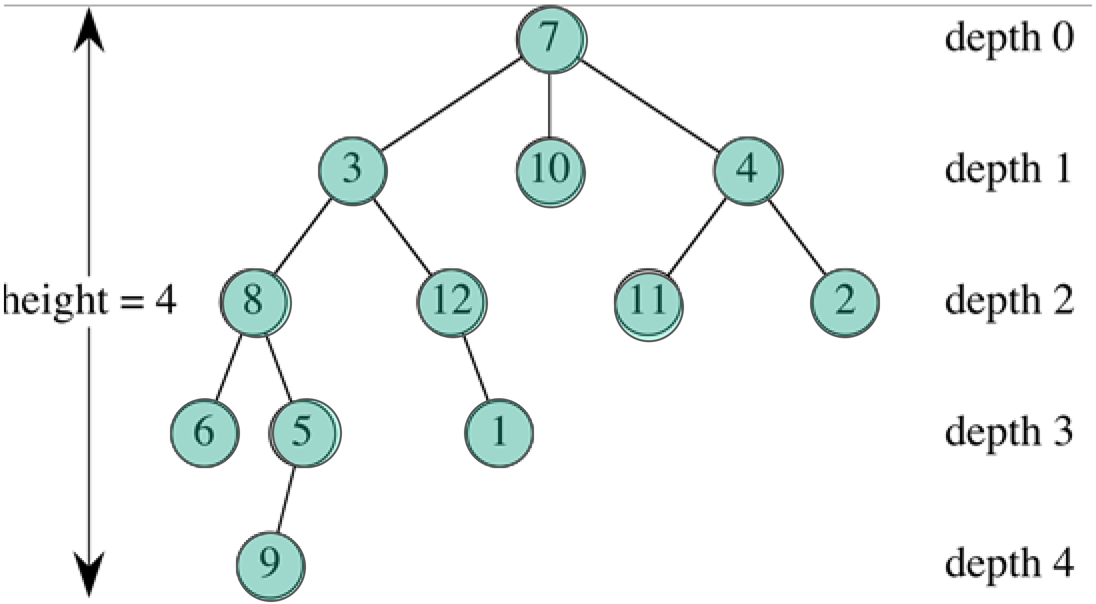
- Many definitions
- Here’s an example:



Text p. 1088 (Appendix B)

Trees

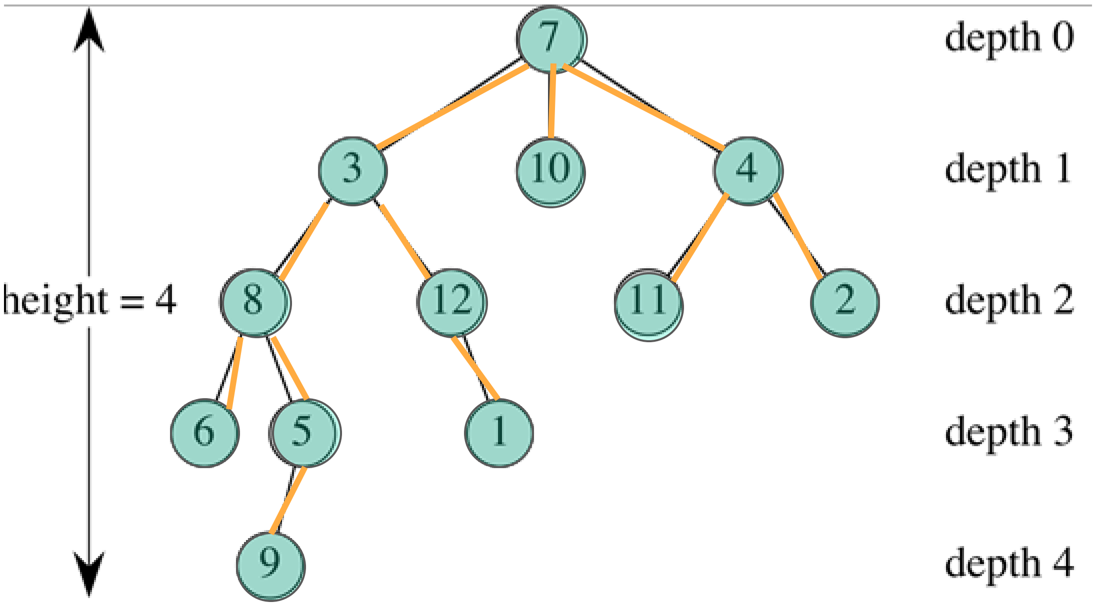
- Many definitions
- Here's an example
- Some notes:
  - Trees have
    - Nodes



Text p. 1088 (Appendix B)

Trees

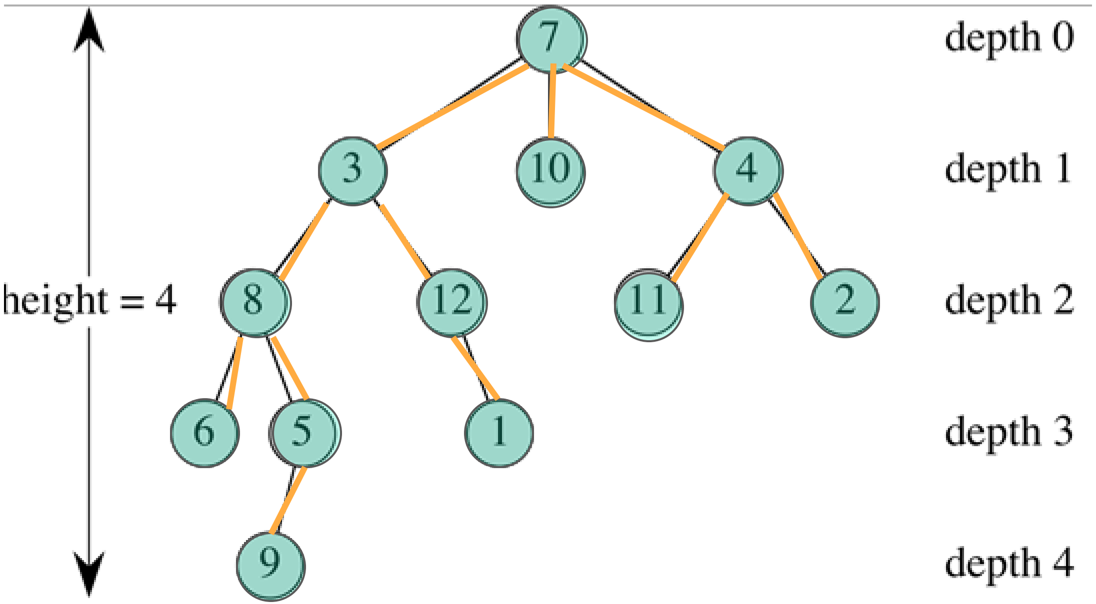
- Many definitions
- Here's an example
- Some notes:
  - Trees have
    - Nodes
    - Edges



Text p. 1088 (Appendix B)

Trees

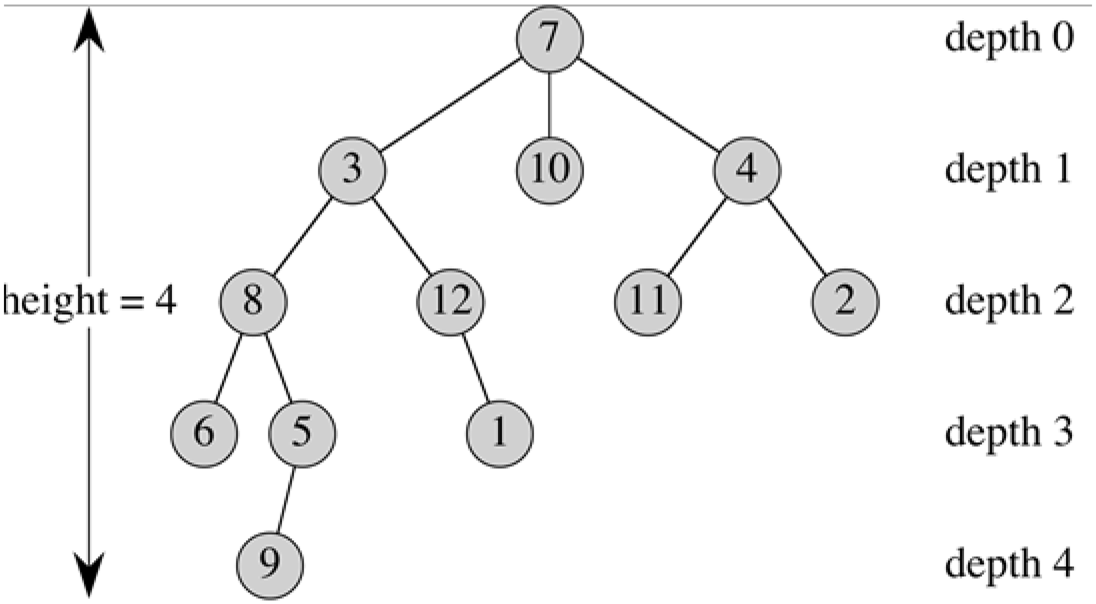
- Many definitions
- Here's an example
- Some notes:
  - Trees have
    - Nodes
    - Edges
  - The edges are undirected



Text p. 1088 (Appendix B)

Trees

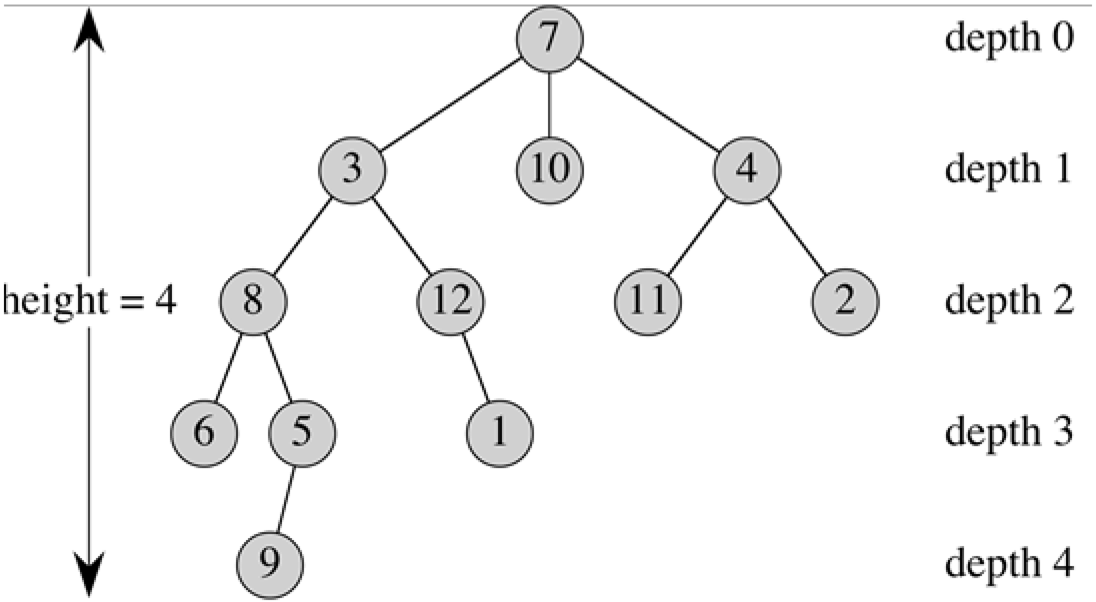
- Many definitions
- Here's an example
- Some notes:
    - Tree is upside down!



Text p. 1088 (Appendix B)

Trees

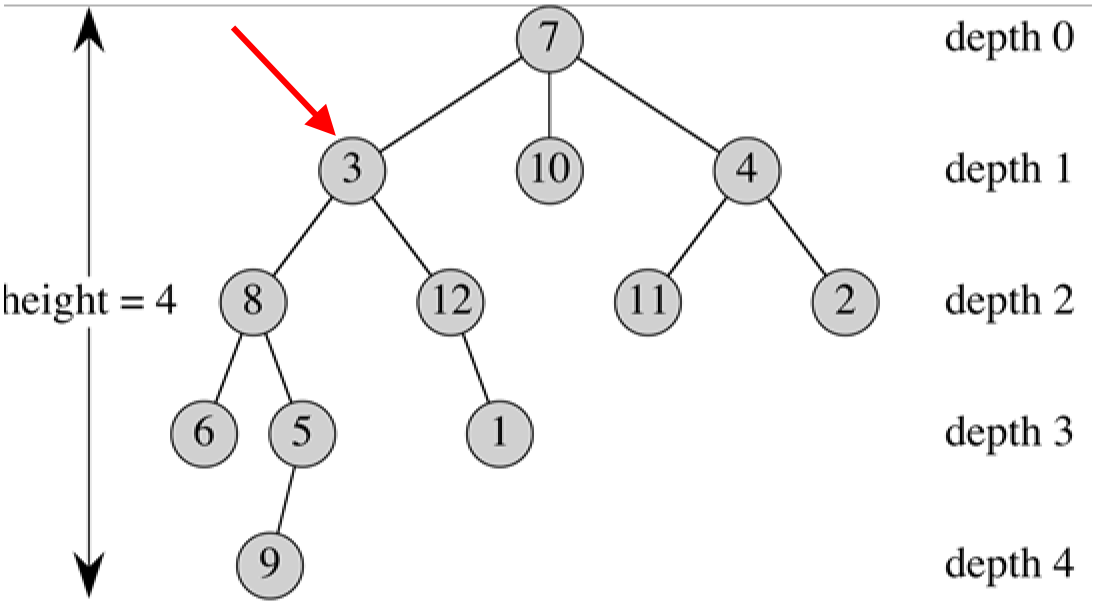
- Many definitions
- Here’s an example
- Some notes:
  - Tree is upside down!



Text p. 1088 (Appendix B)

Trees

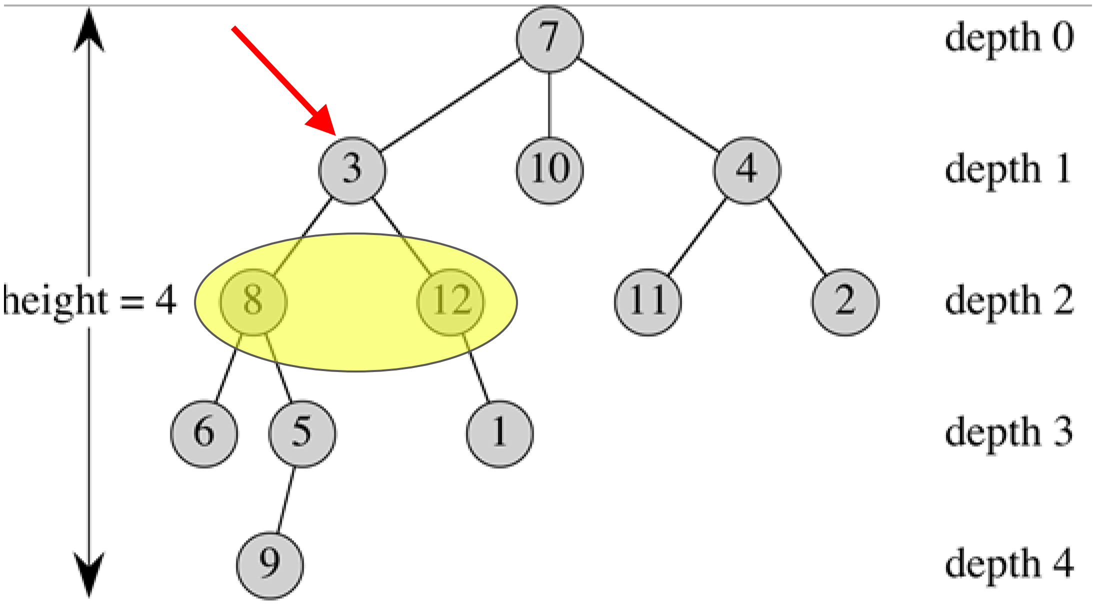
- Many definitions
- Here's an example
- Some notes:
  - Tree is upside down!
  - We speak of a parent



Text p. 1088 (Appendix B)

Trees

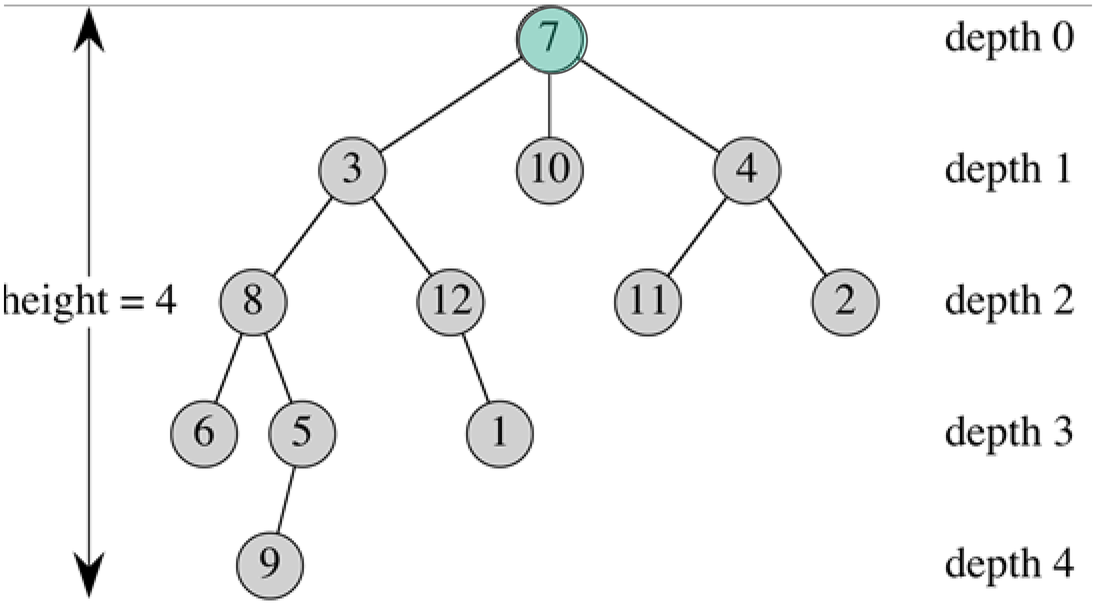
- Many definitions
- Here’s an example
- Some notes:
  - Tree is upside down!
  - We speak of a parent
    - And its children
    - They are siblings



Text p. 1088 (Appendix B)

Trees

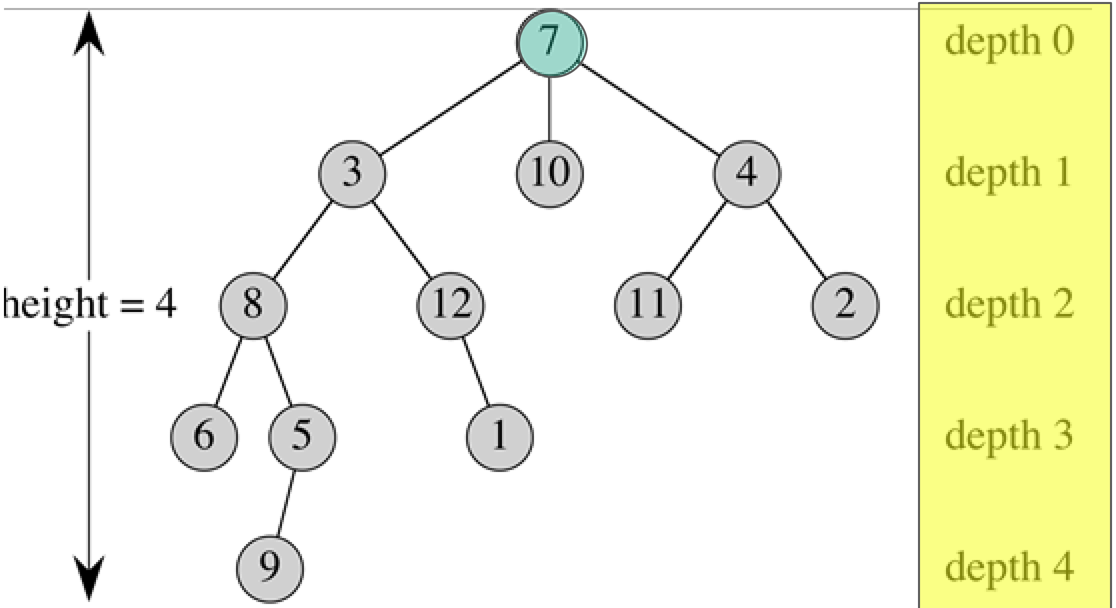
- Many definitions
- Here's an example
- Some notes:
  - Tree is upside down!
  - We speak of a parent
    - And its children
- For now
  - A tree is *rooted*
    - Root is orphan node



Text p. 1088 (Appendix B)

Trees

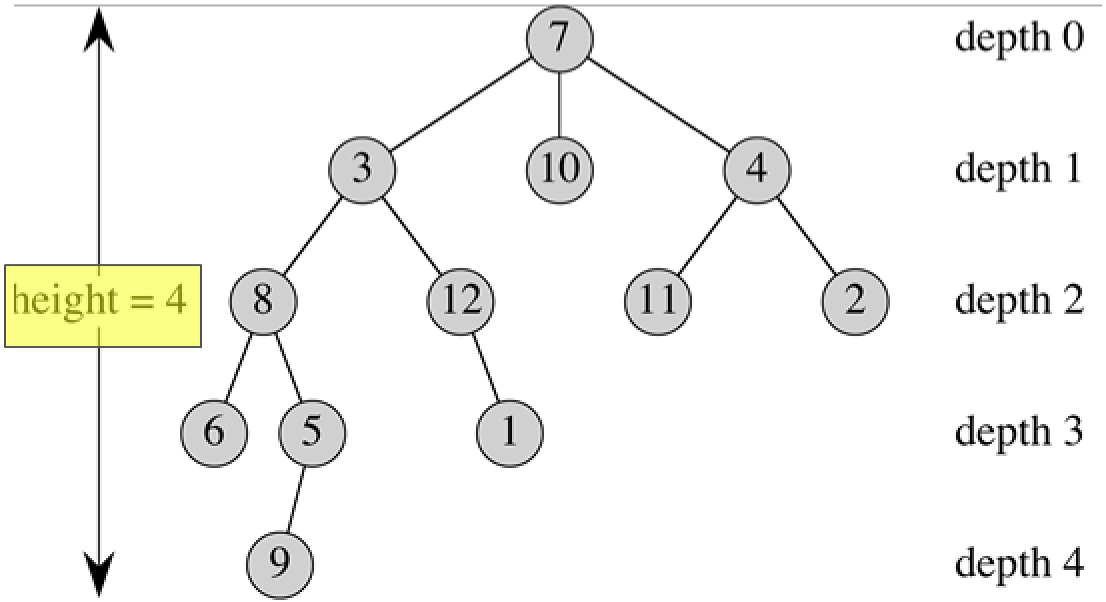
- Each node occurs at some *depth* from the root
  - The root is at depth 0



Text p. 1088 (Appendix B)

Trees

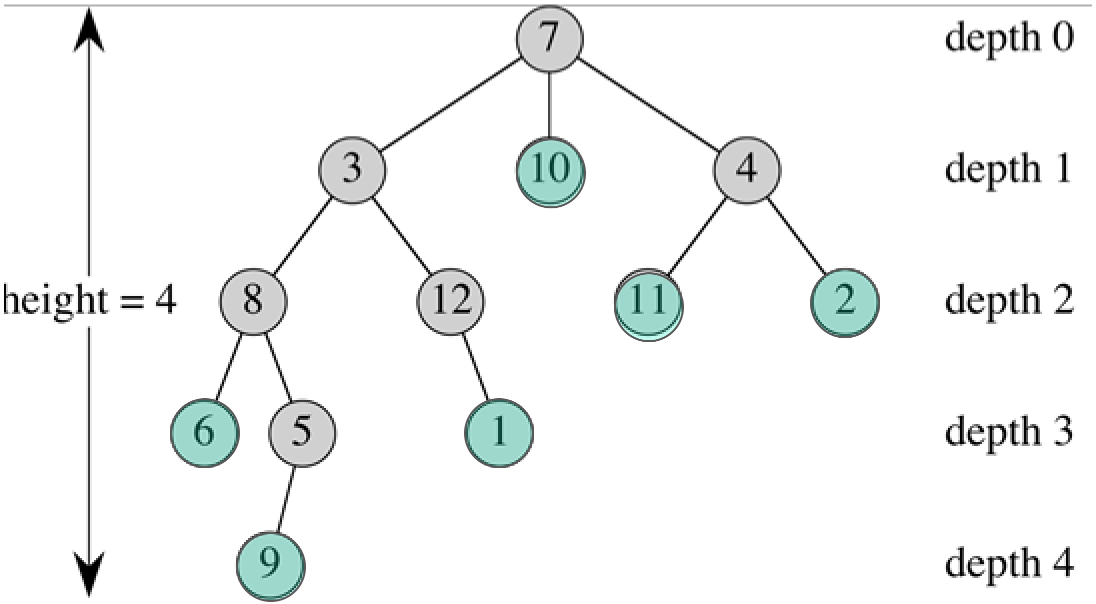
- Each node occurs at some *depth* from the root
  - The root is at depth 0
- The height of a tree is the maximum depth among all of the tree's nodes



Text p. 1088 (Appendix B)

Trees

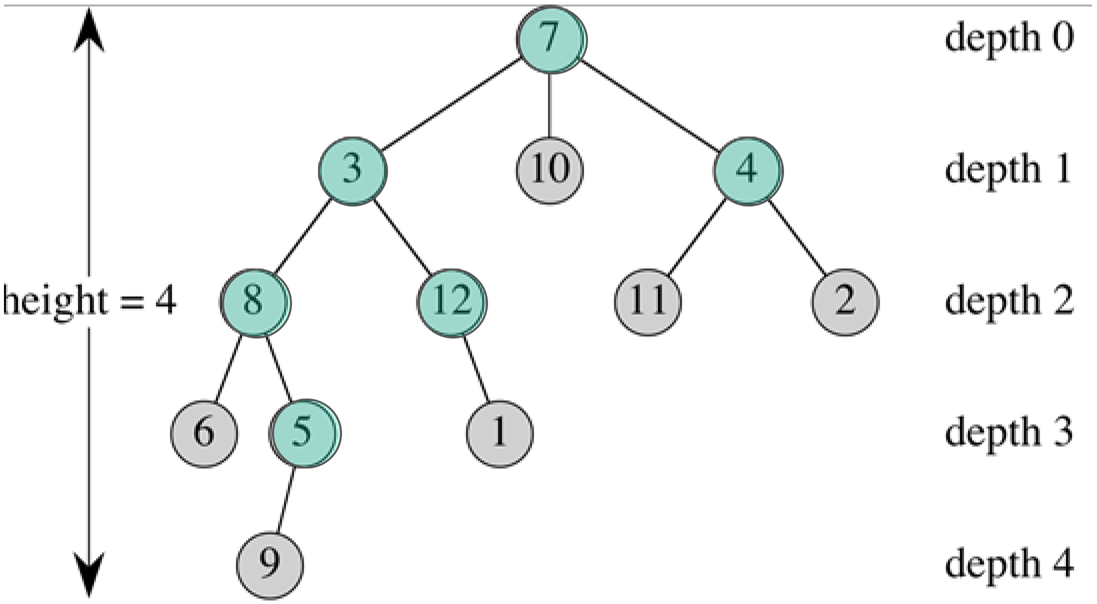
- Each node occurs at some *depth* from the root
    - The root is at depth 0
- The height of a tree is the maximum depth among all of the tree's nodes
- Some nodes are *leaves*



Text p. 1088 (Appendix B)

Trees

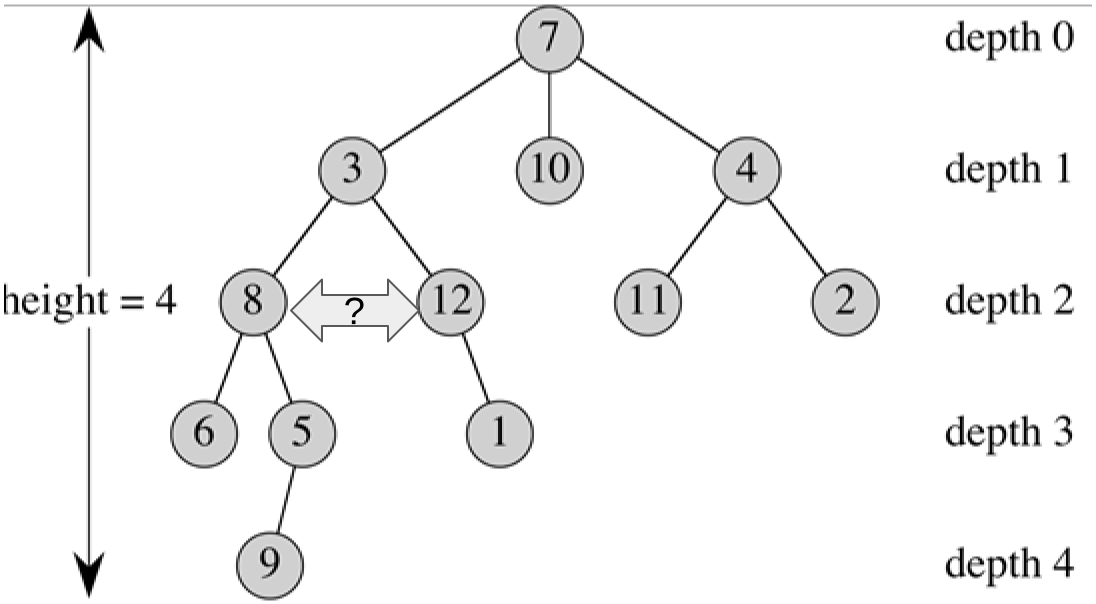
- Each node occurs at some *depth* from the root
  - The root is at depth 0
- The height of a tree is the maximum depth among all of the tree's nodes
- Some nodes are *leaves*
- Others are *internal nodes*



Text p. 1088 (Appendix B)

Trees

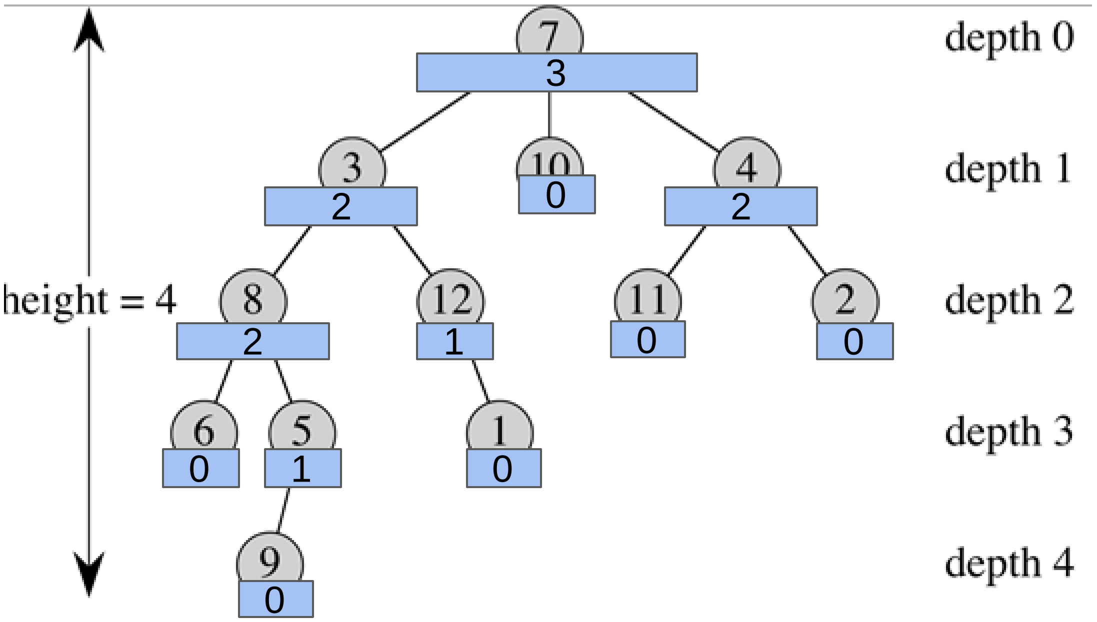
- If the left-to-right orientation of the children matters, tree is **ordered**



Text p. 1088 (Appendix B)

Trees

- If the left-to-right orientation of the children matters, tree is **ordered**
- The *degree* of a node is the count of its children



Text p. 1088 (Appendix B)

OK, Back to Priority Queues…

Specializing Trees for Our Needs

- We'll focus on binary trees – every node has at most two children.

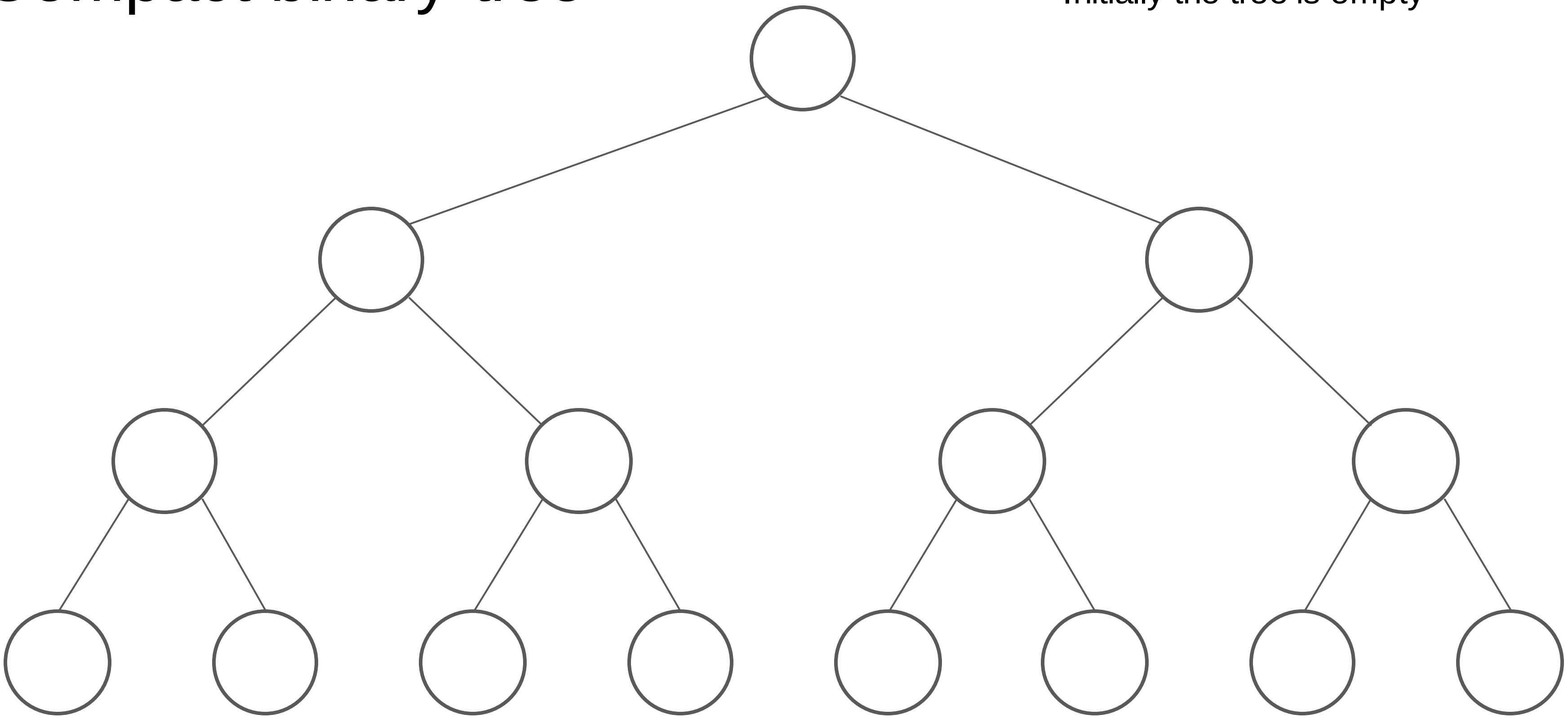
- We'll focus on compact binary trees – nodes are always added top-to-bottom and left-to-right

Specializing Trees for Our Needs

- **We'll focus on binary trees – every node has at most two children.**
- **We'll focus on compact binary trees – nodes are always added top-to-bottom and left-to-right**
- **(*There is a unique compact binary tree with n nodes*)**

Compact binary tree

Initially the tree is empty

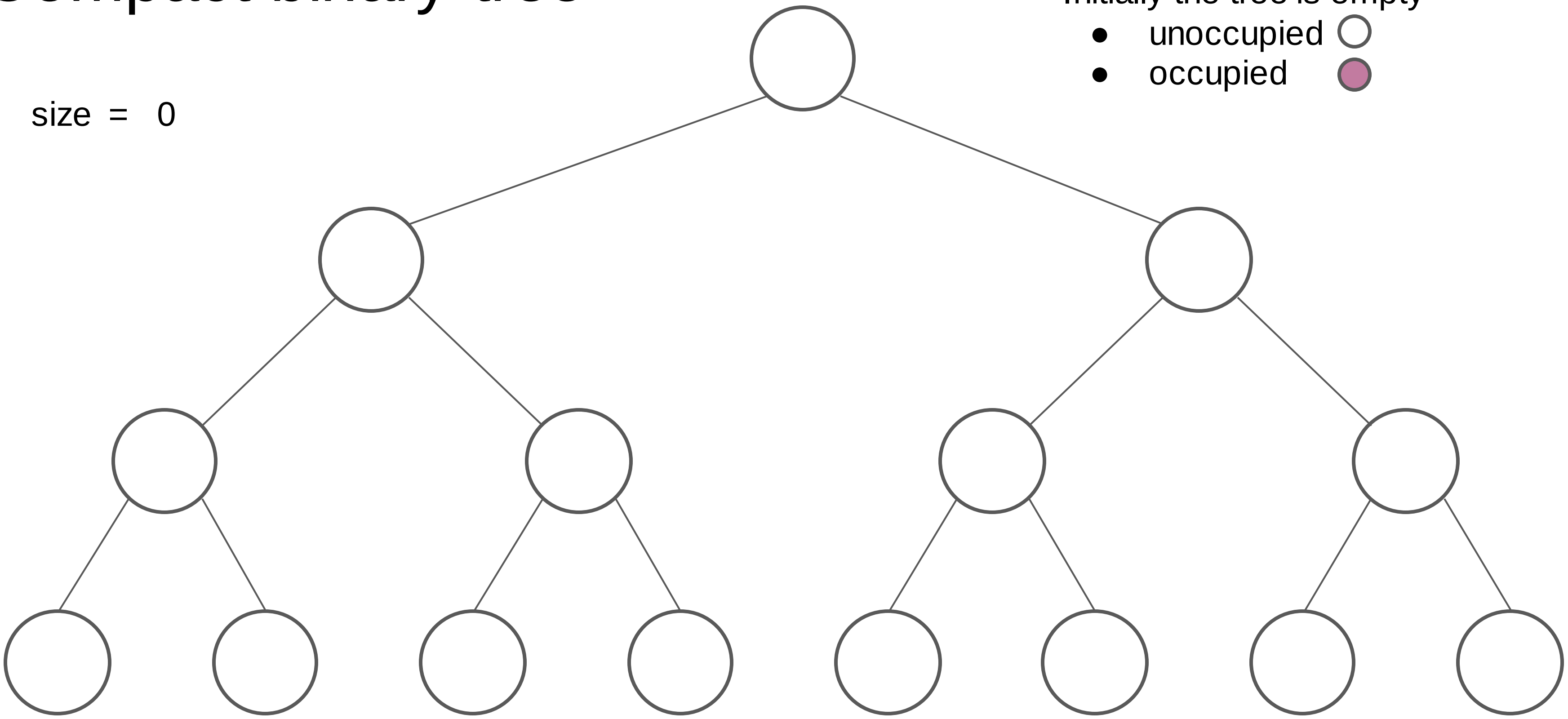


Compact binary tree

size = 0

Initially the tree is empty

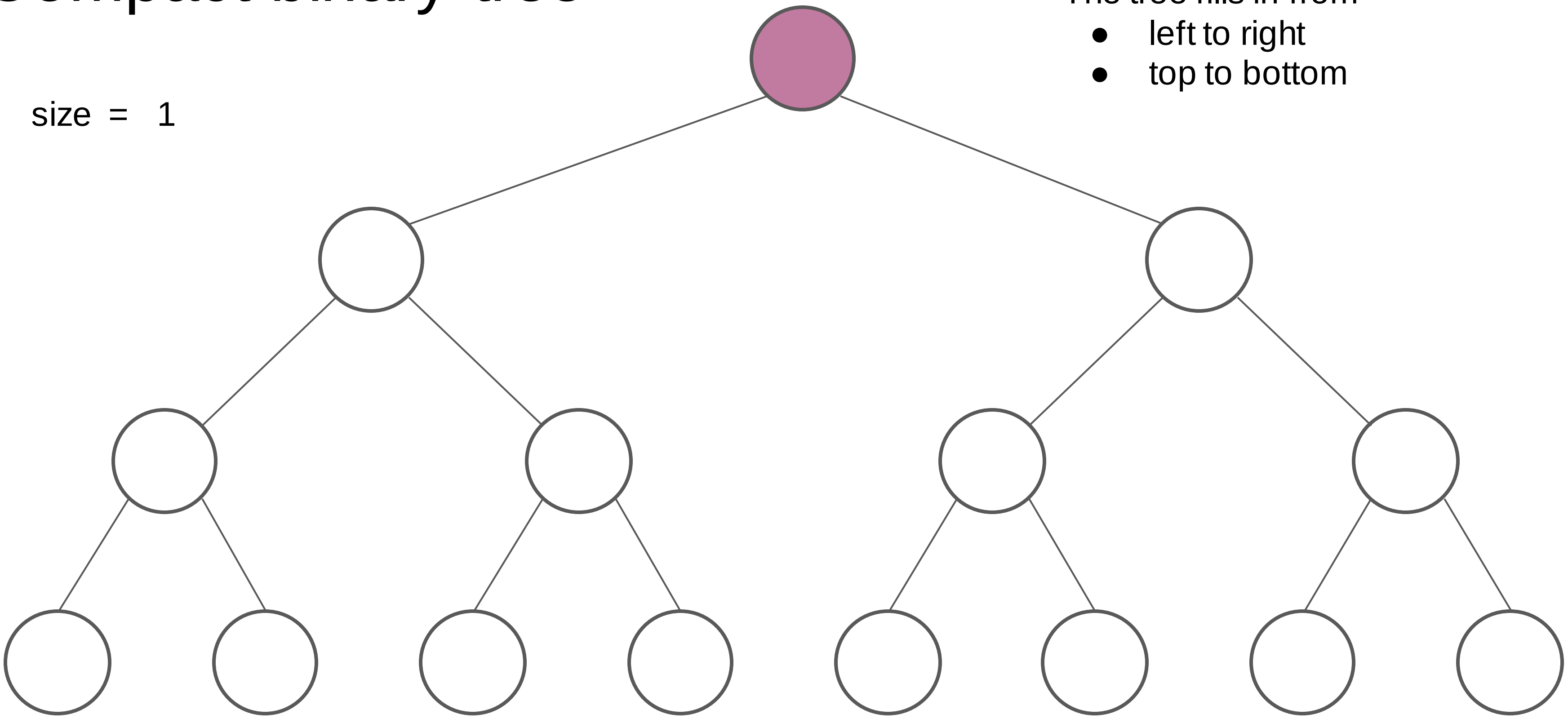
- unoccupied
- occupied



Compact binary tree

The tree fills in from
- left to right
- top to bottom

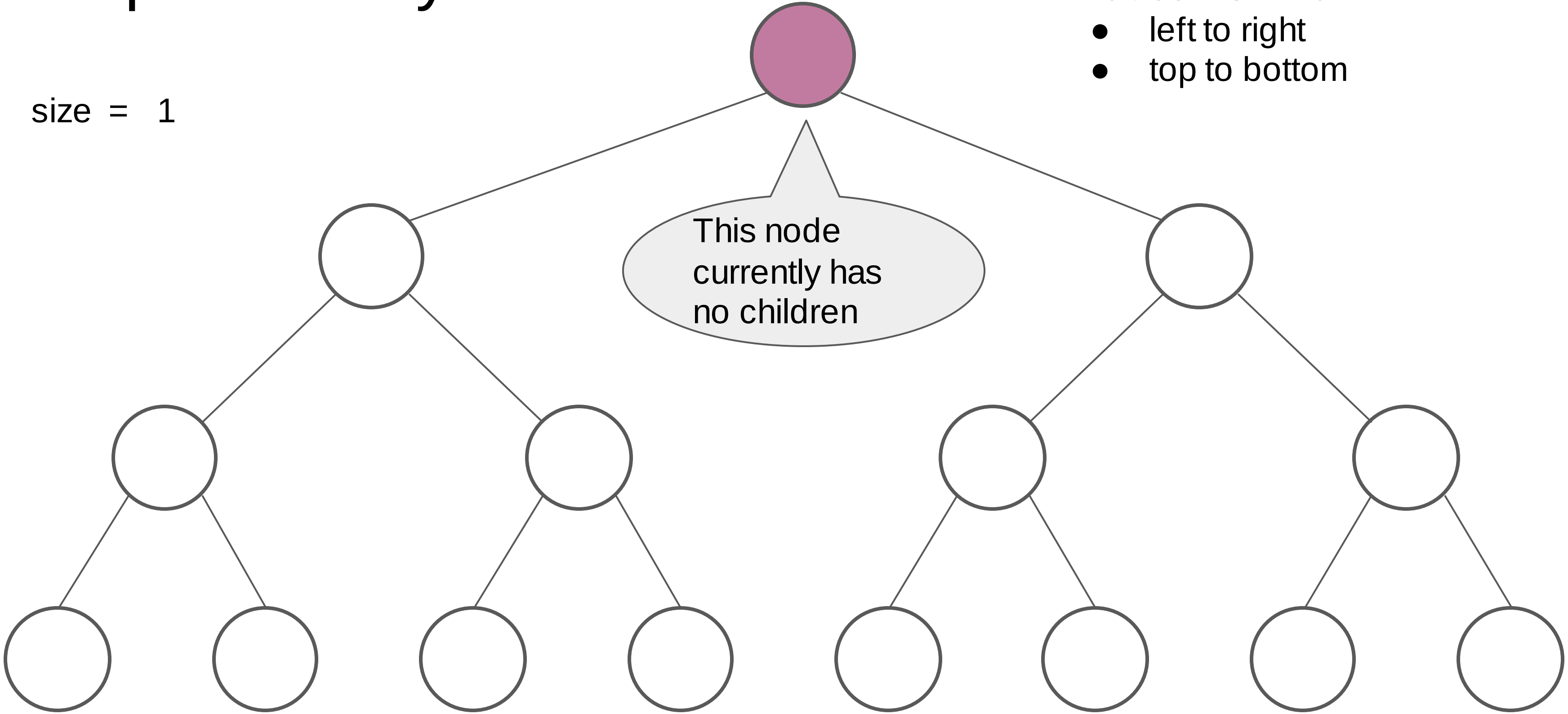
size = 1



Compact binary tree

The tree fills in from
- left to right
- top to bottom

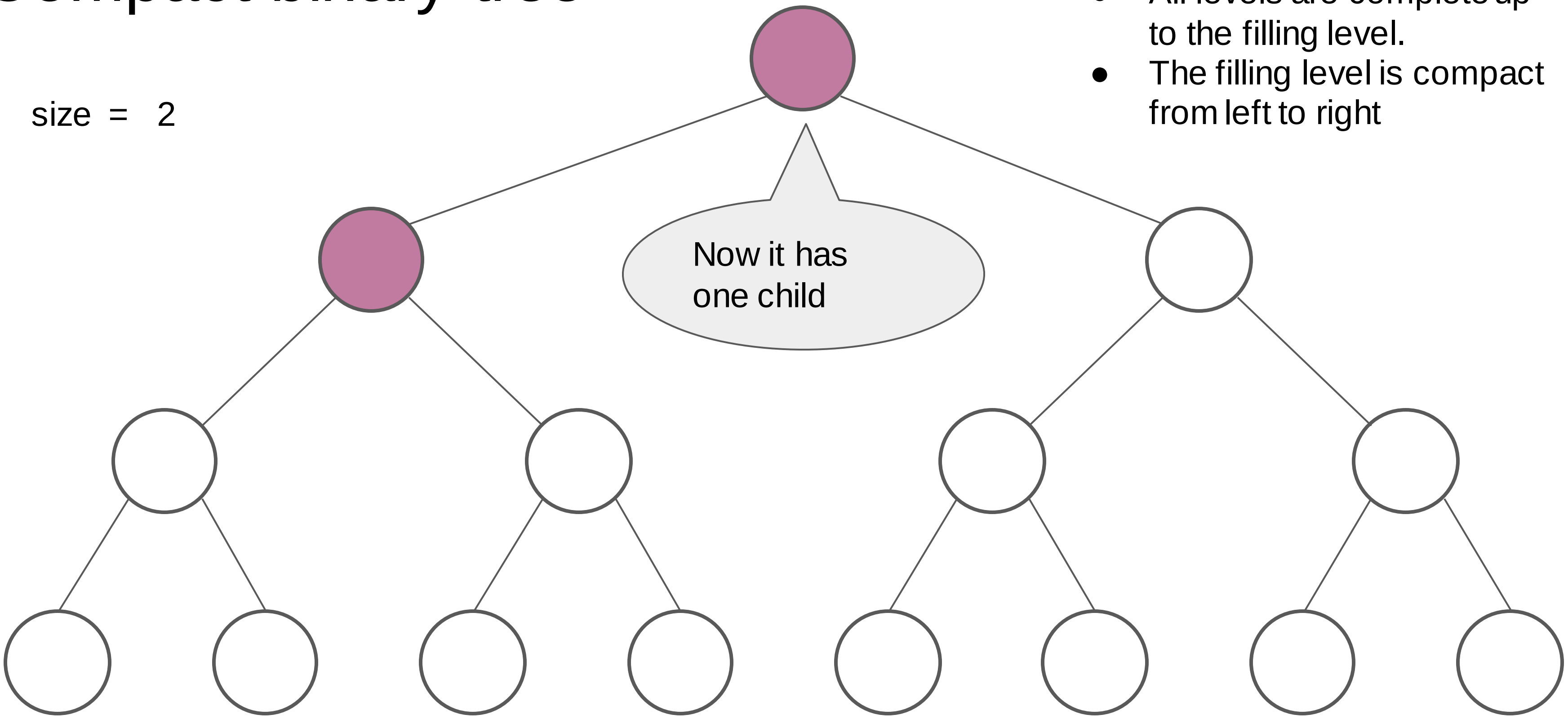
size = 1



Compact binary tree

size = 2

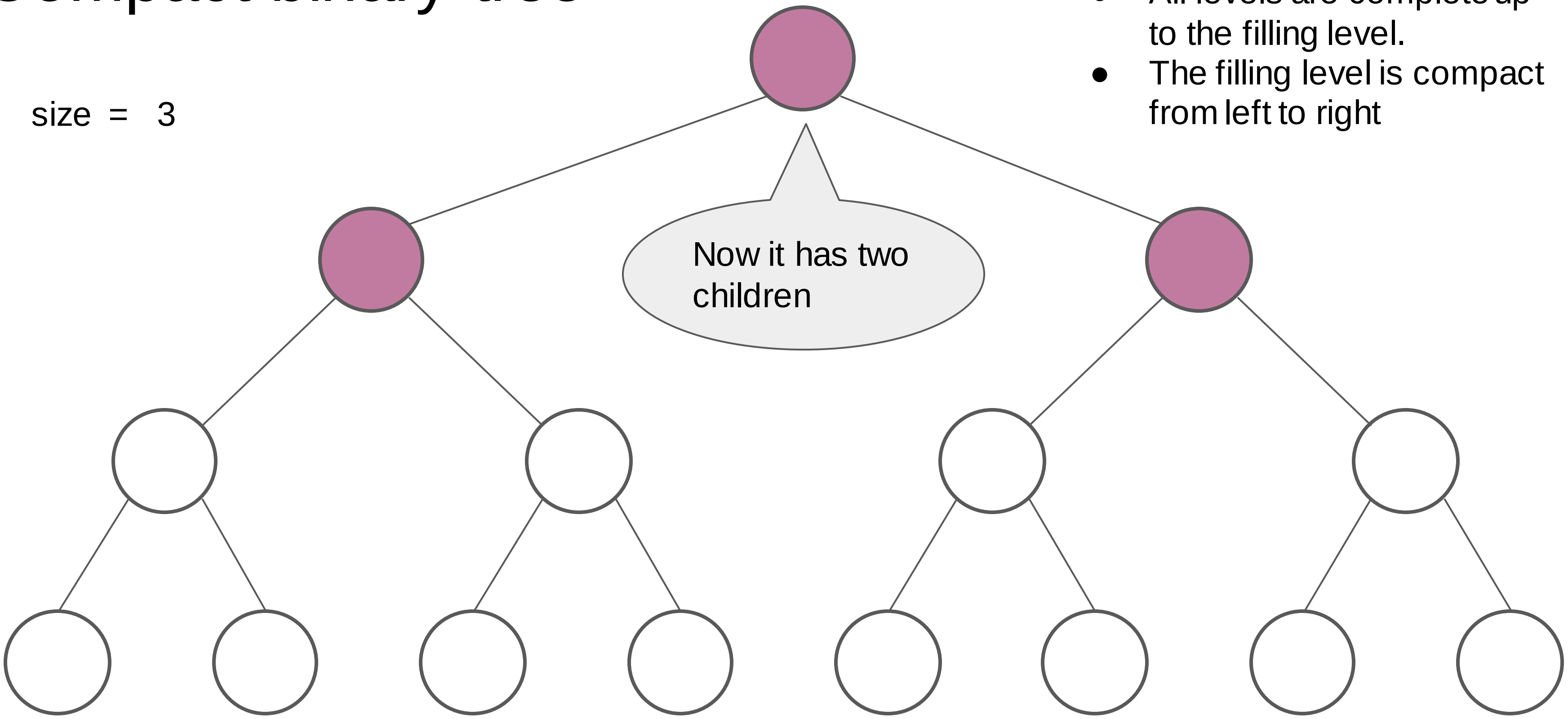
- All levels are complete up to the filling level.
- The filling level is compact from left to right



Compact binary tree

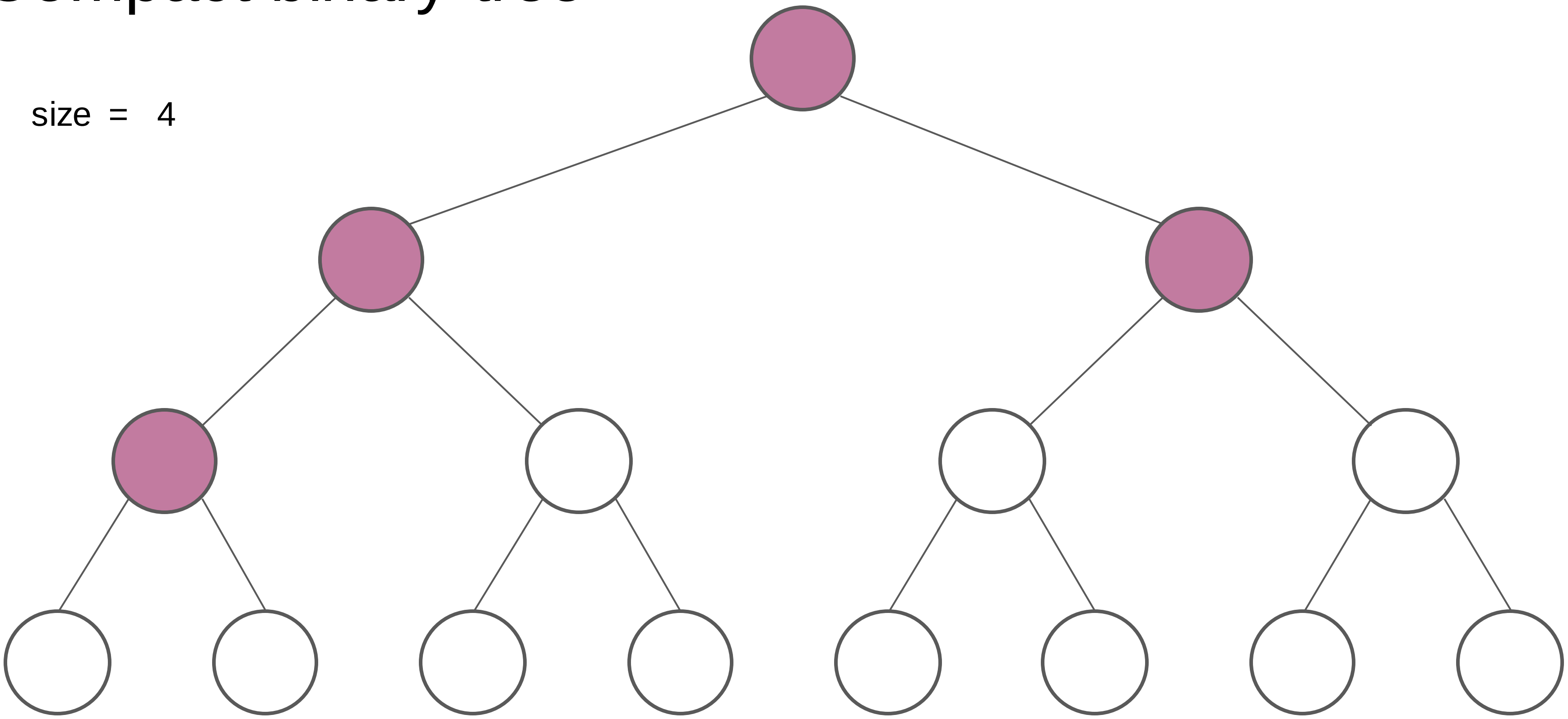
size = 3

- All levels are complete up to the filling level.
- The filling level is compact from left to right



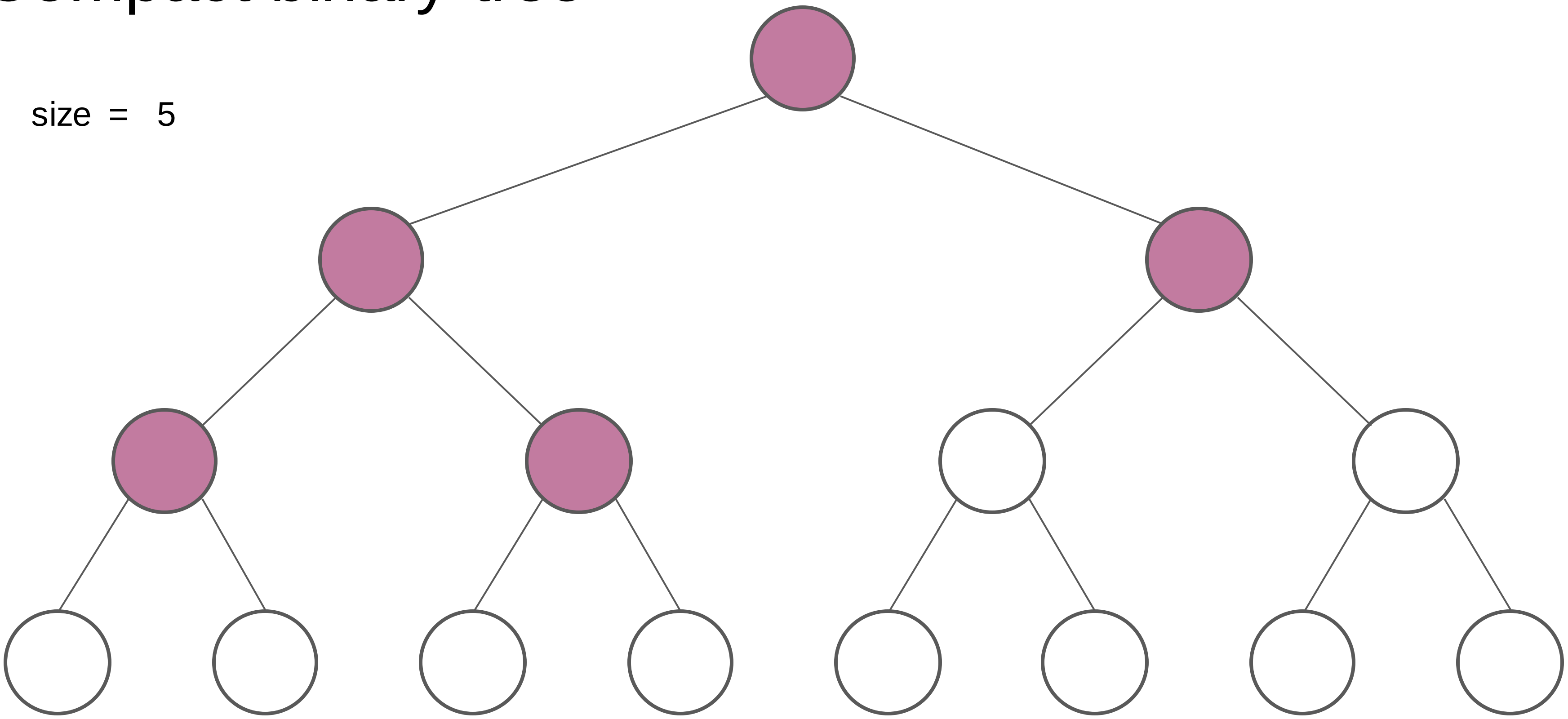
Compact binary tree

size = 4



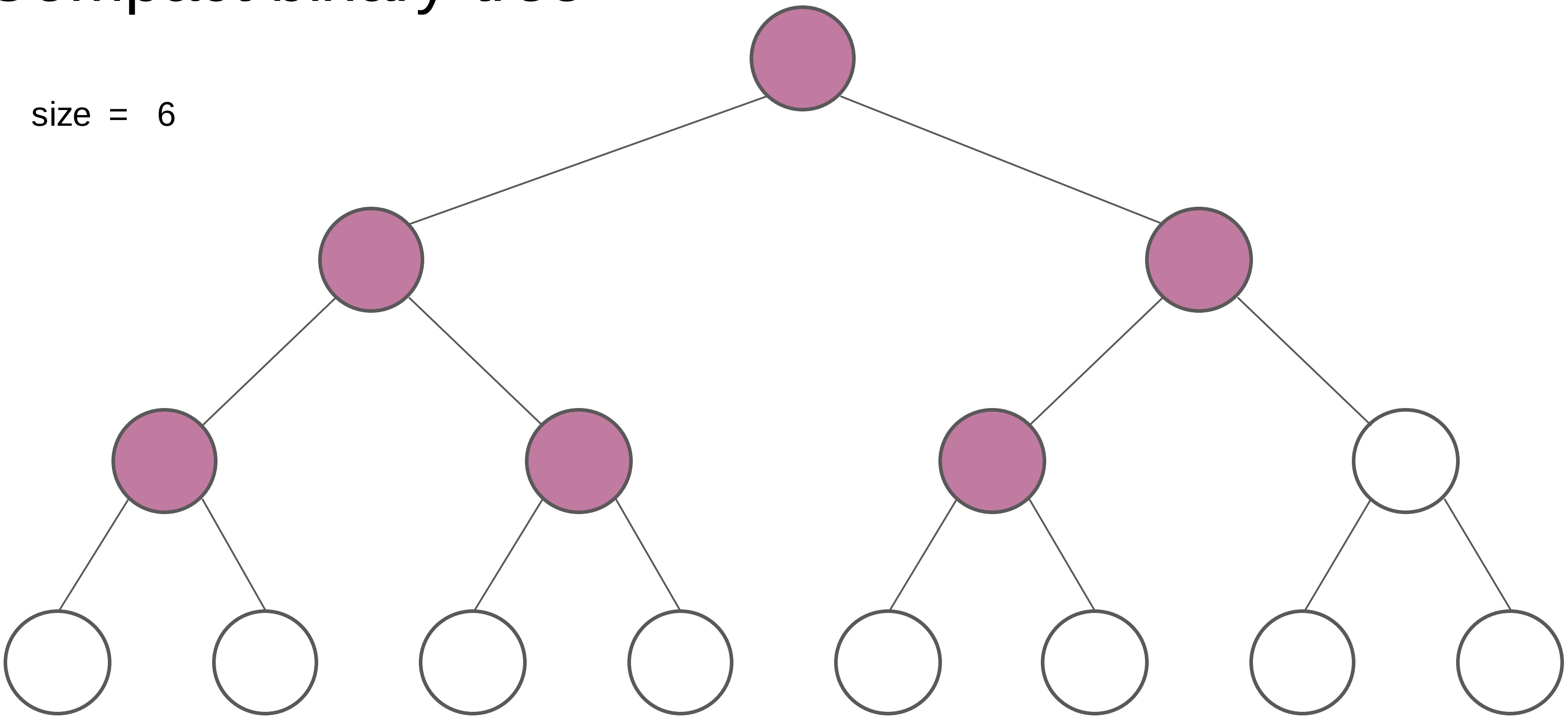
Compact binary tree

size = 5



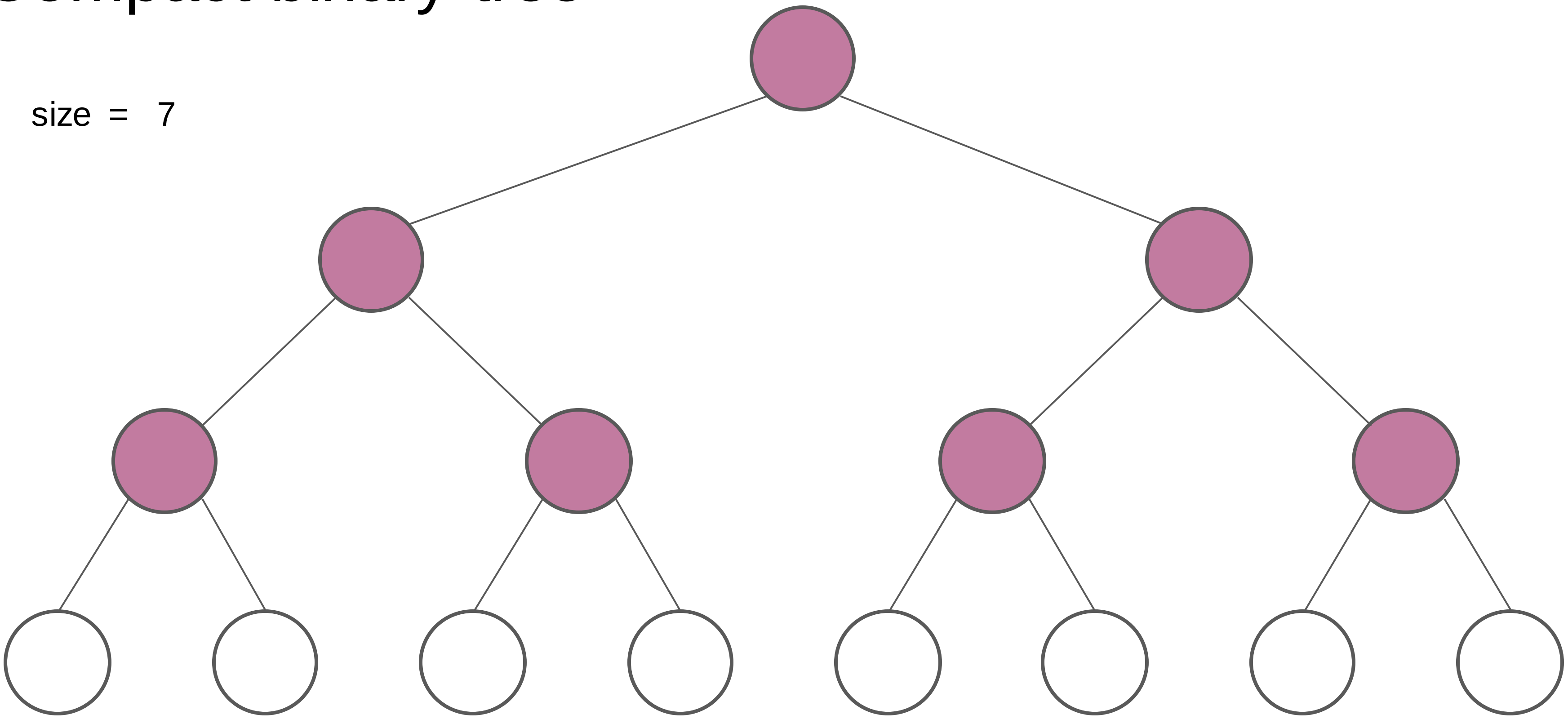
Compact binary tree

size = 6



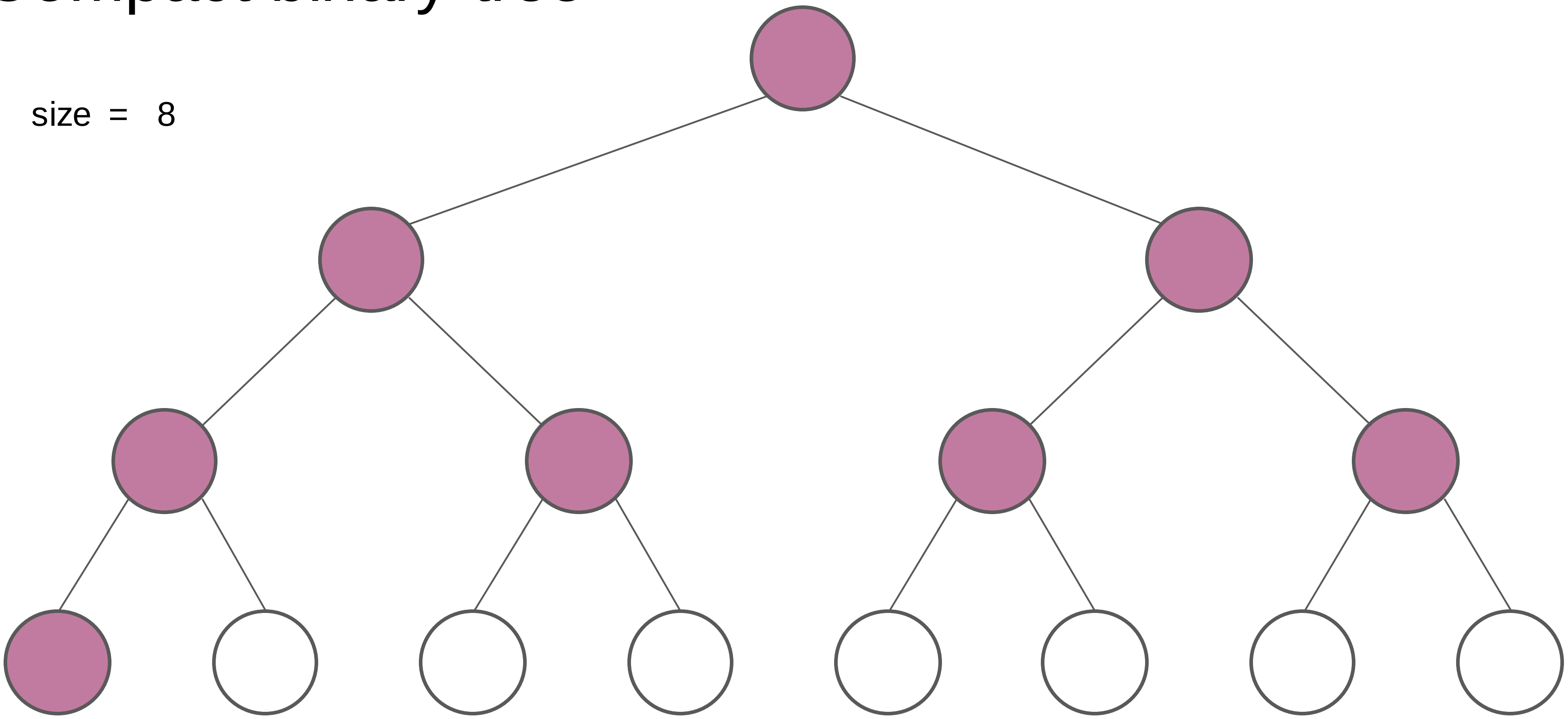
Compact binary tree

size = 7



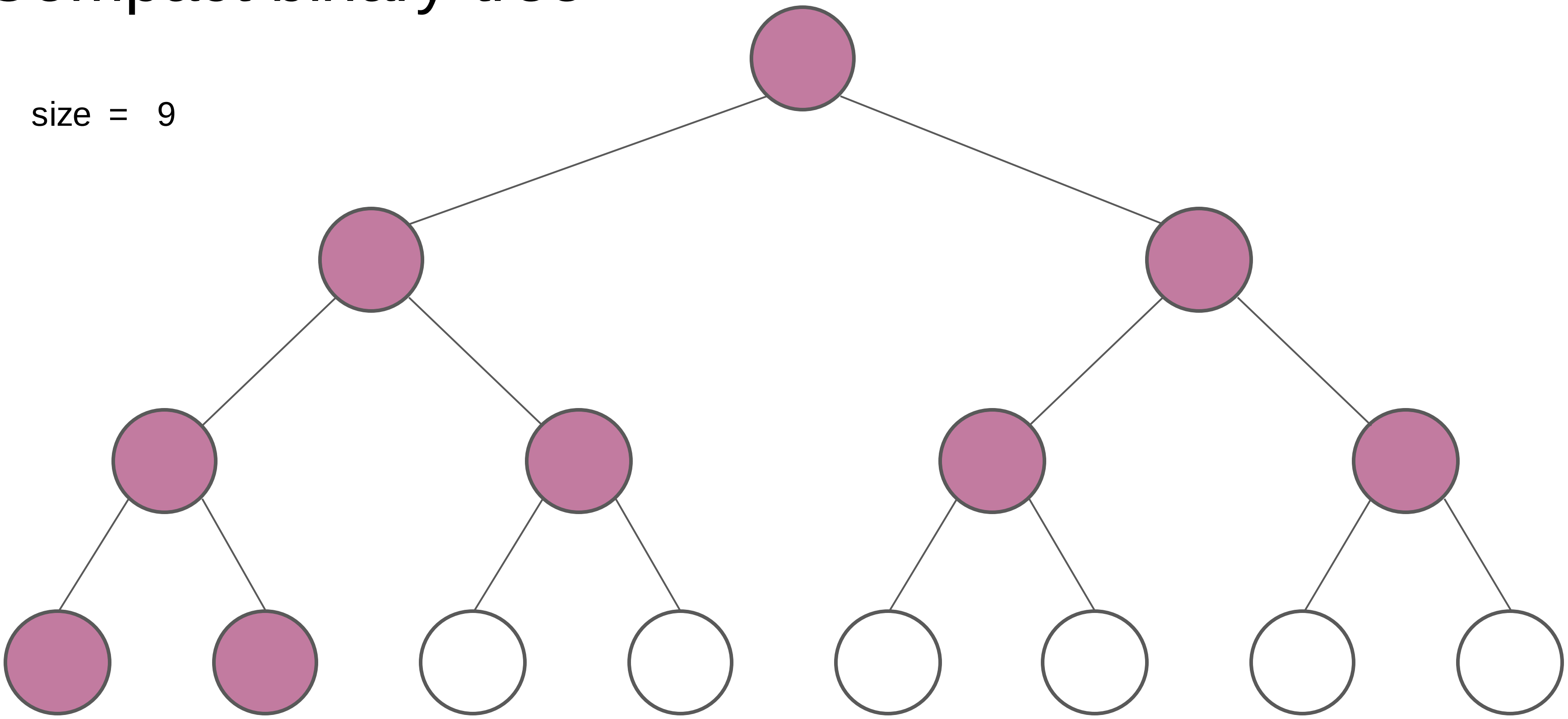
Compact binary tree

size = 8

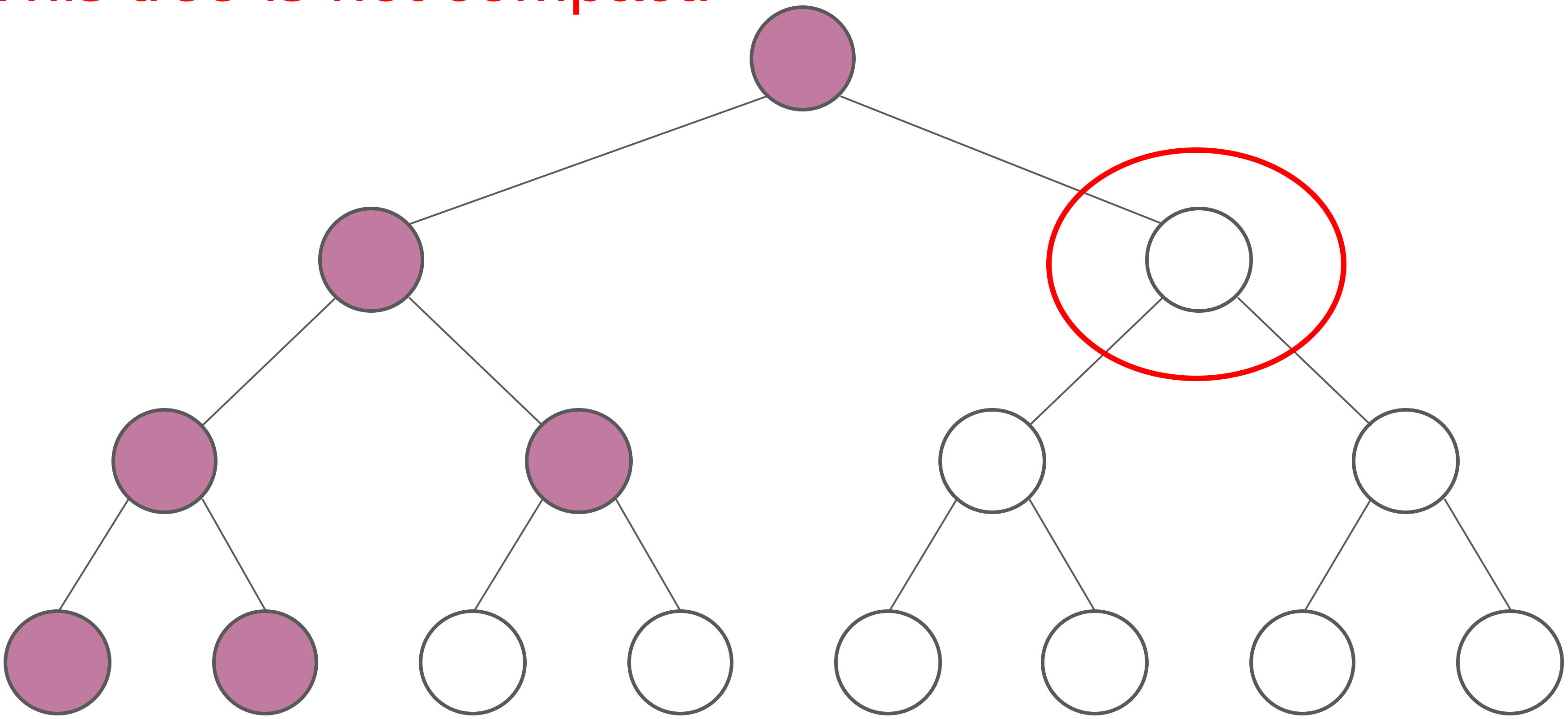


Compact binary tree

size = 9

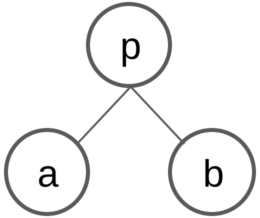


This tree is not compact!



A **binary heap** is a compact binary tree with an ordering invariant – the heap property.

Heap property: a special relationship between each parent and its children



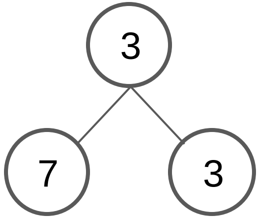
- **$p.value \leq min(a.value, b.value)$**

property for
min-first heaps!

- Says *nothing* about how a.value and b.value compare

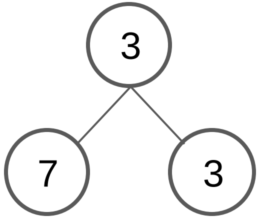
Examples

- Has heap property

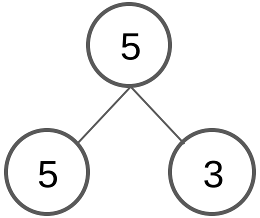


Examples

- Has heap property

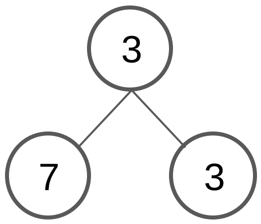


- Lacks heap property (what do you see that is wrong?)

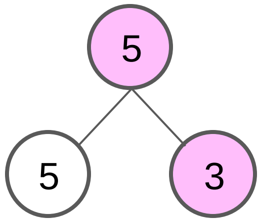


Examples

- Has heap property



- Lacks heap property



Heap Property Implies Fast peekMin()

- In a heap, the heap property applies between every node and its children (if any).

- So where is the smallest element in a binary heap?

Heap Property Implies Fast peekMin()

- In a heap, the heap property applies between every node and its children (if any).

- So where is the smallest element in a binary heap?

- At the root, of course…

Heap Property Implies Fast peekMin()

- In a heap, the heap property applies between every node and its children (if any).
- So where is the smallest element in a binary heap?
- At the root, of course?
- Better prove it…

Theorem

- If a heap is not empty → a minimum element is found at its root

Theorem

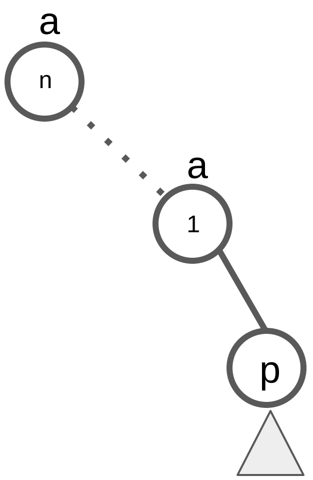
- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):

Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
    - Suppose we have a nonempty heap and the root node is not a minimal element
    - Then a minimal element must exist somewhere else, say at node p

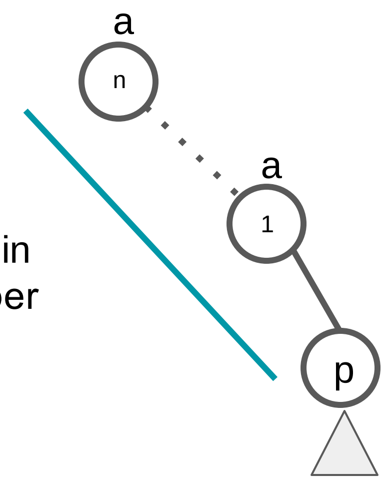
Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
  - Suppose we have a nonempty heap and the root node is not a minimal element
  - Then a minimal element must exist somewhere else, say at node p



Theorem

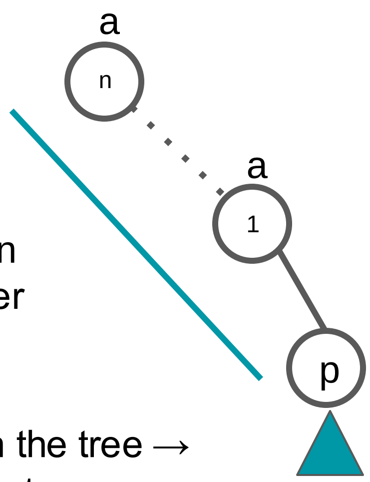
- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
    - Suppose we have a nonempty heap and the root node is not a minimal element
    - Then a minimal element must exist somewhere else, say at node p



Nodes above p in the tree → proper ancestors of p

Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
  - Suppose we have a nonempty heap and the root node is not a minimal element
  - Then a minimal element must exist somewhere else, say at node p

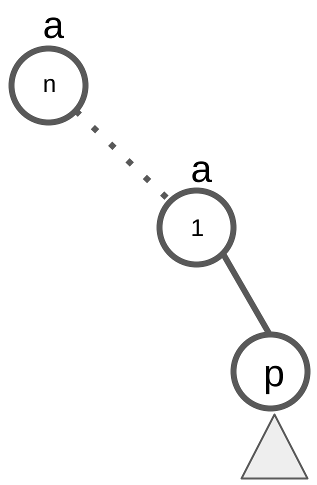


Nodes above p in the tree → proper ancestors of p

Nodes below p in the tree → proper descendants

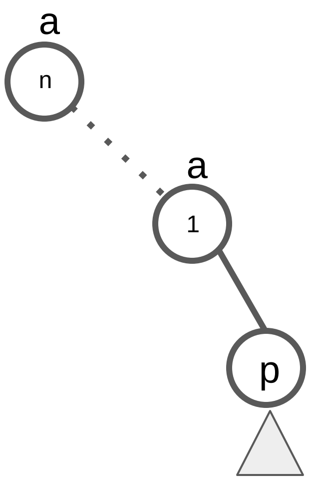
Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
    - Suppose we have a nonempty heap and the root node is not a minimal element
    - Then a minimal element must exist somewhere else, say at node p
    - Here let's say the name and value of a node are synonymous
        - so p contains value p, a_1 contains value a_1 , and so on



Theorem

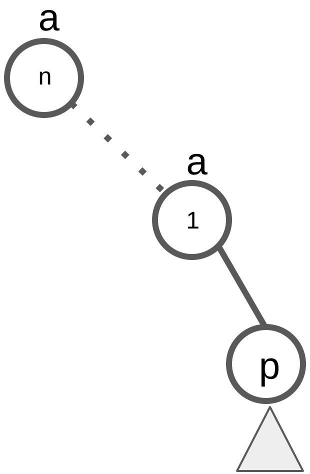
- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
  - Suppose we have a nonempty heap and the root node is not a minimal element
  - Then a minimal element must exist somewhere else, say at node p
  - Here let's say the name and value of a node are synonymous
    - so p contains value p, a_1 contains value a_1 , and so on
  - Consider the sequence of proper ancestors of p
    - a_1 a_2 ….a_n



Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
  - Suppose we have a nonempty heap and the root node is not a minimal element
  - Then a minimal element must exist somewhere else, say at node p
  - Here let's say the name and value of a node are synonymous
    - so p contains value p, a_1 contains value a_1 , and so on
  - Consider the sequence of proper ancestors of p
    - a_1 a_2 ….a_n
  - Applying the **heap property** we obtain:

$$a_n \leq \ldots \leq a_1 \leq p$$

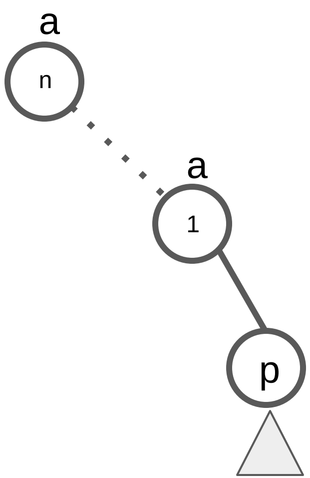


Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
  - Suppose we have a nonempty heap and the root node is not a minimal element
  - Then a minimal element must exist somewhere else, say at node p
  - Here let's say the name and value of a node are synonymous
    - so p contains value p, a_1 contains value a_1 , and so on
  - Consider the sequence of proper ancestors of p
    - a_1 a_2 ….a_n
  - Applying the **heap property** we obtain:

$$a_n \leq \ldots \leq a_1 \leq p$$

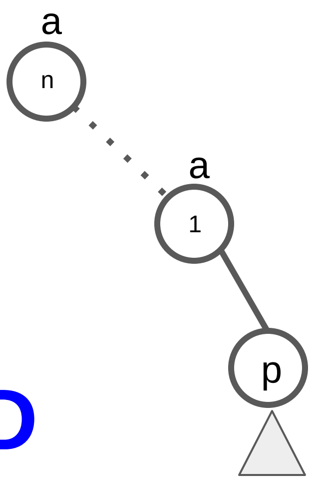
  - This contradicts the claim that a_n is not a minimal element



Theorem

- If a heap is not empty → a minimum element is found at its root
- Proof (by contradiction):
    - Suppose we have a nonempty heap and the root node is not a minimal element
    - Then a minimal element must exist somewhere else, say at node p
    - Here let's say the name and value of a node are synonymous
        - so p contains value p, a_1 contains value a_1 , and so on
    - Consider the sequence of proper ancestors of p
        - a_1 a_2 ….a_n
    - Applying the **heap property** we obtain:

$$a_n \leq \ldots \leq a_1 \leq p$$



- This contradicts the claim that a_n is not a minimal element **QED**

Binary Heap Operations

- How do we implement
  - insert
  - extractMin
  - decrease

Binary Heap Operations

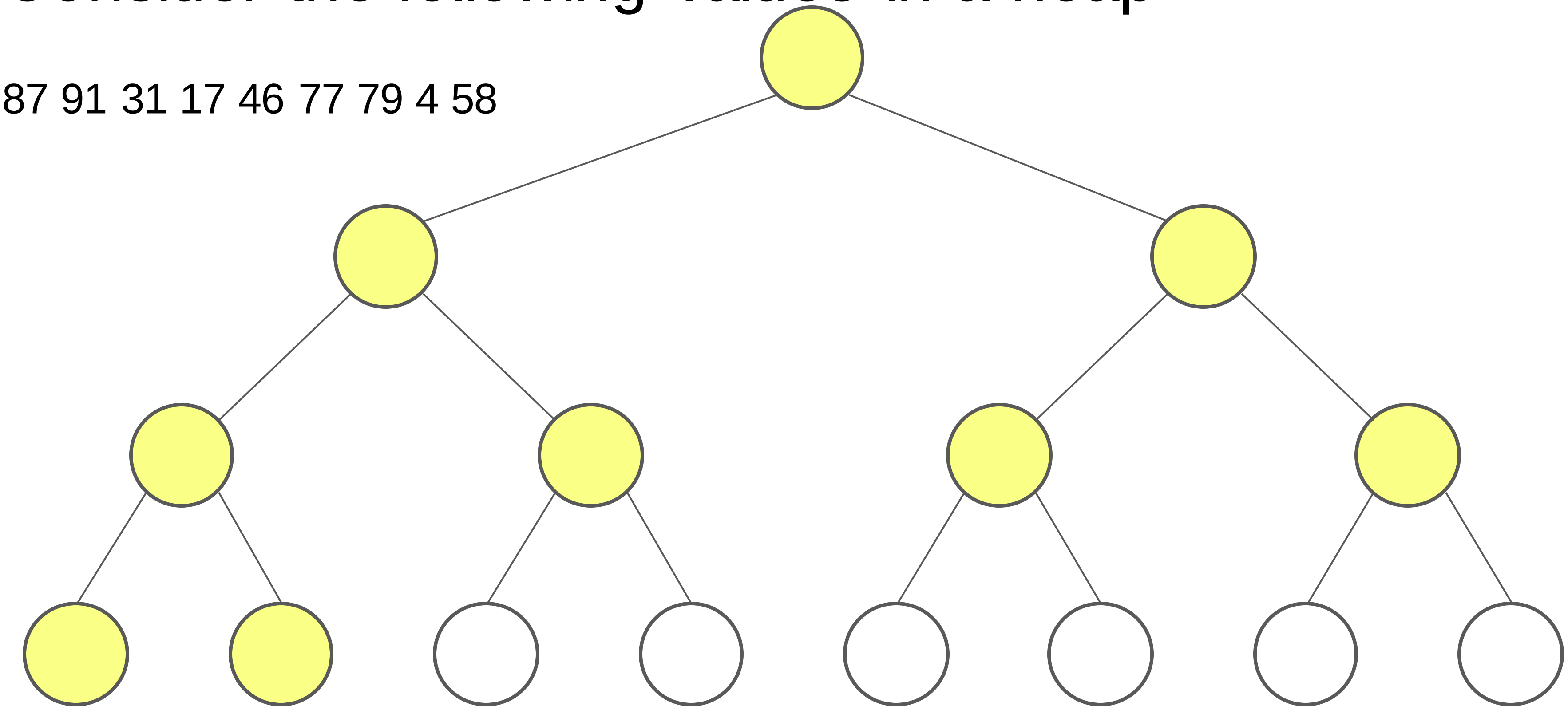
- How do we implement
  - insert
  - extractMin
  - decrease ⇨ We'll do this one first

Consider the following values in a heap

87 91 31 17 46 77 79 4 58

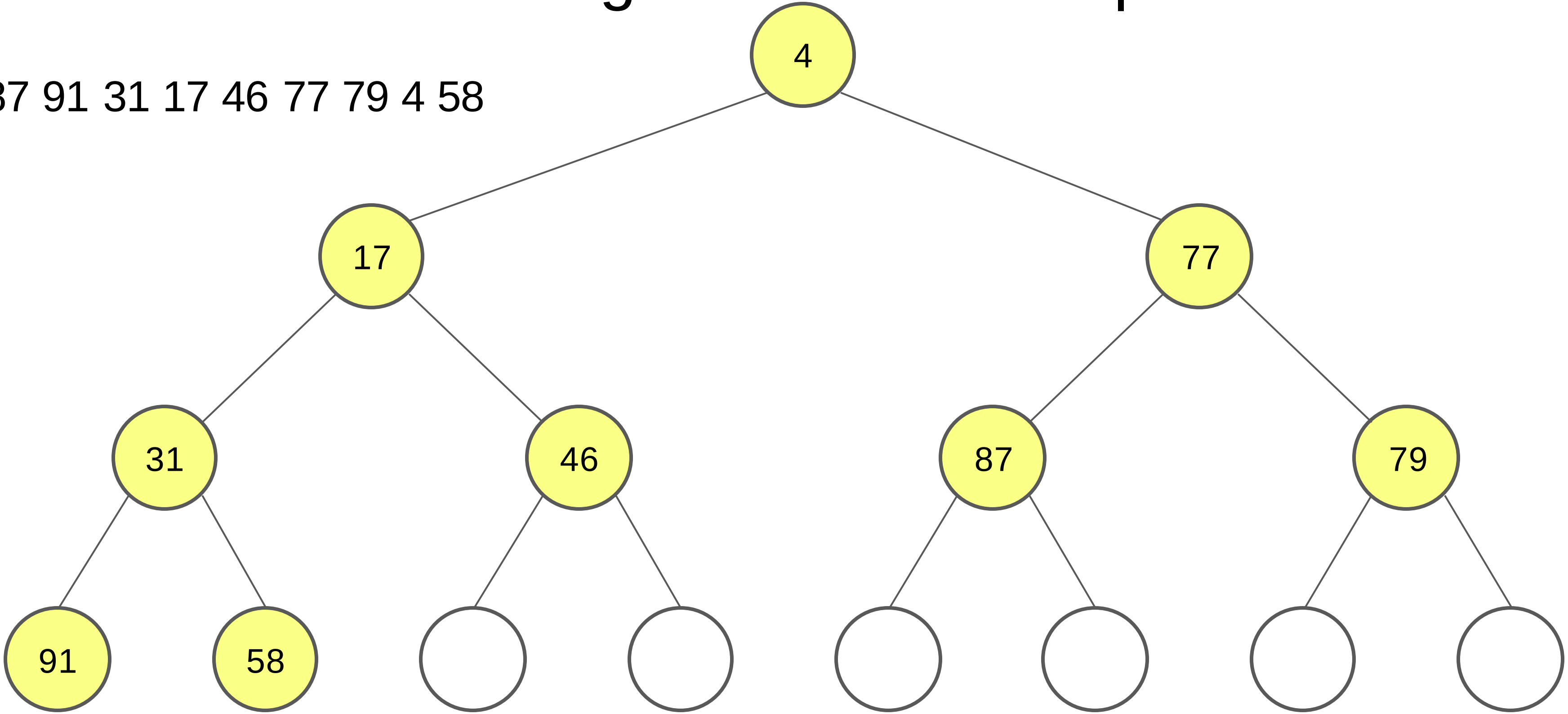
Consider the following values in a heap

87 91 31 17 46 77 79 4 58

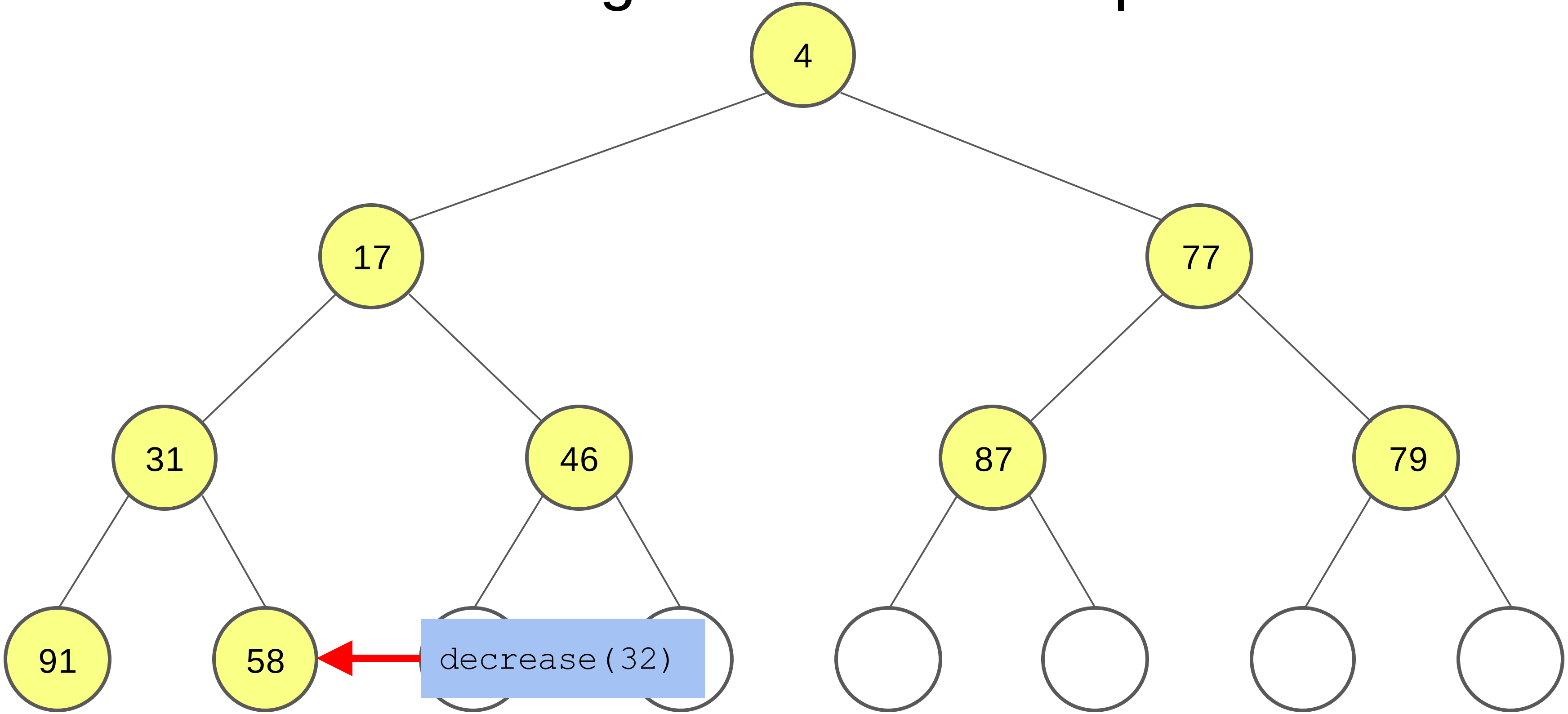


Consider the following values in a heap

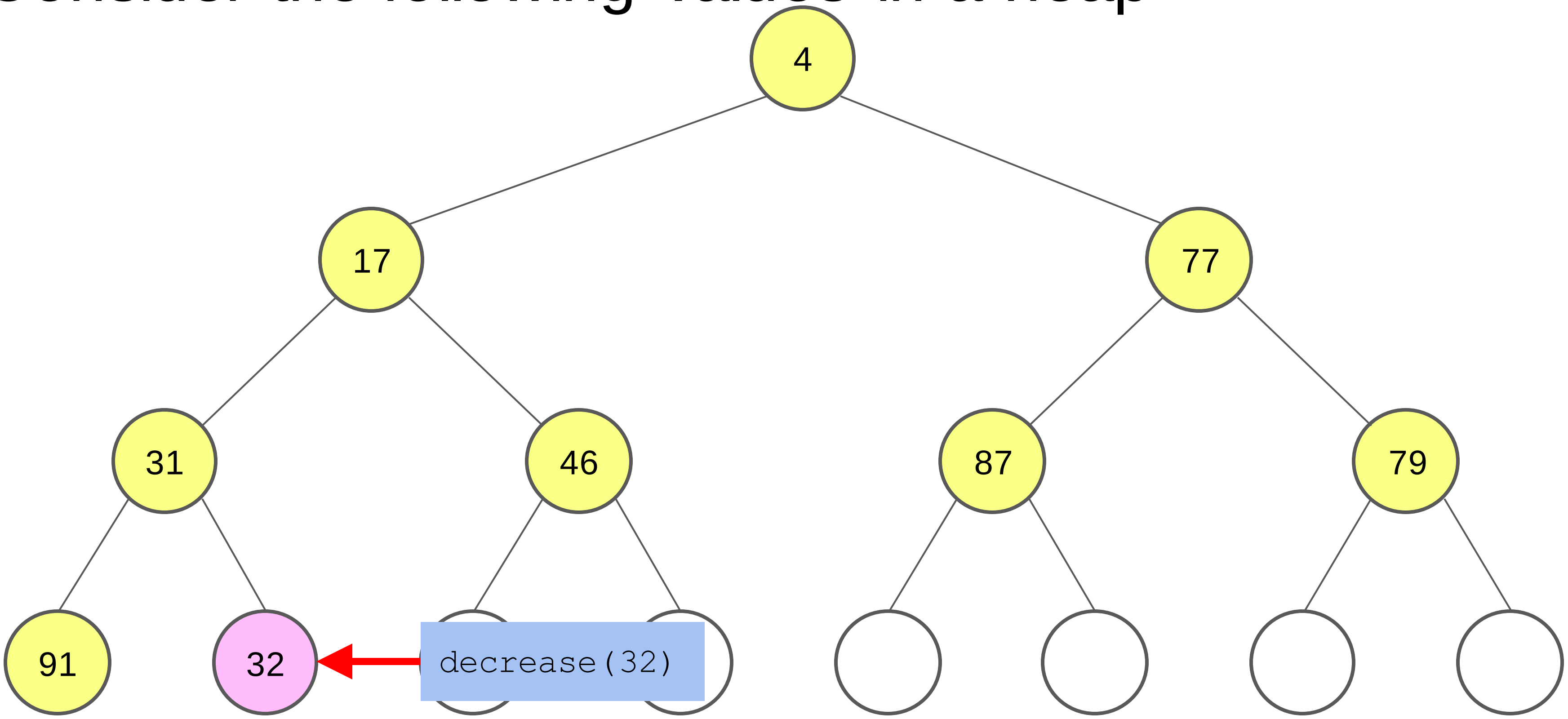
87 91 31 17 46 77 79 4 58



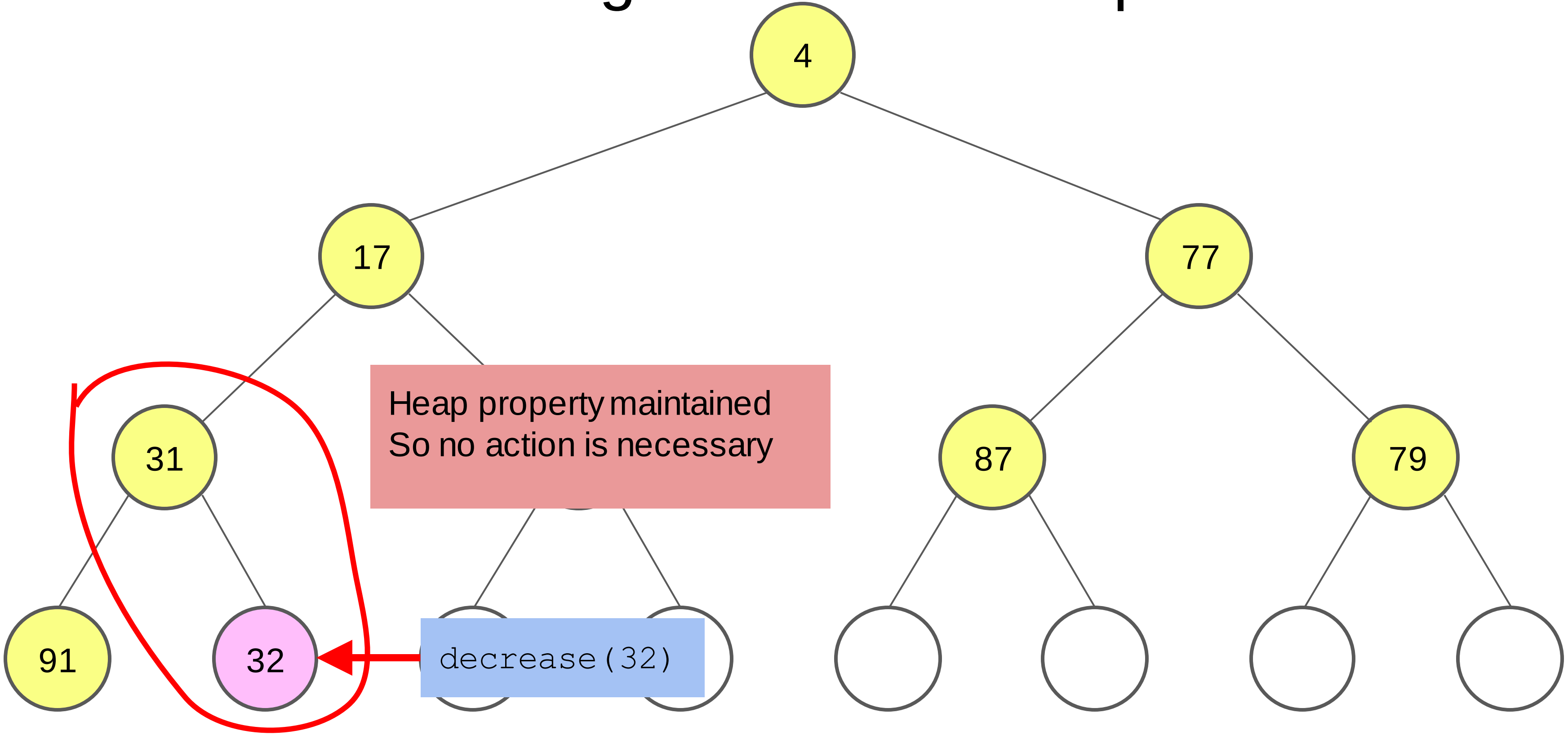
Consider the following values in a heap



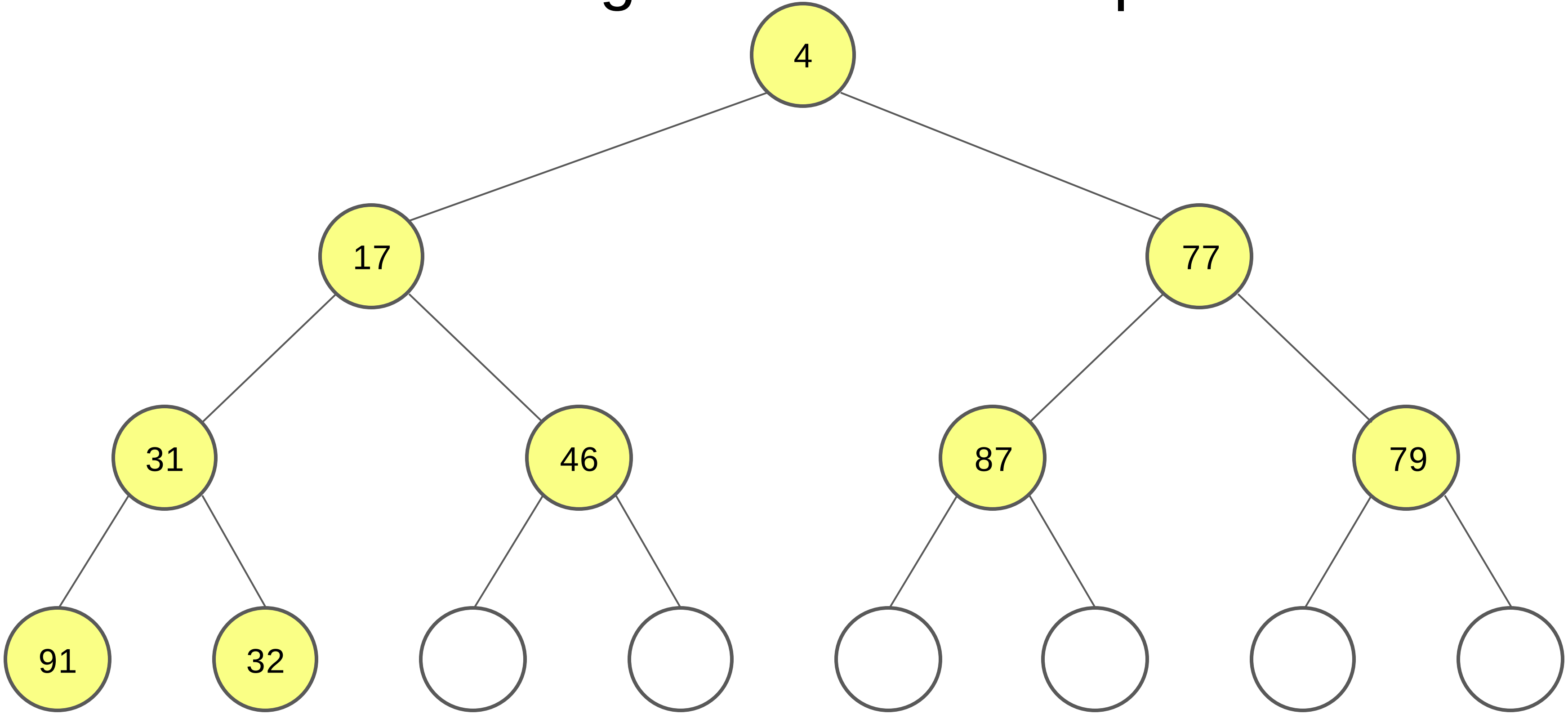
Consider the following values in a heap



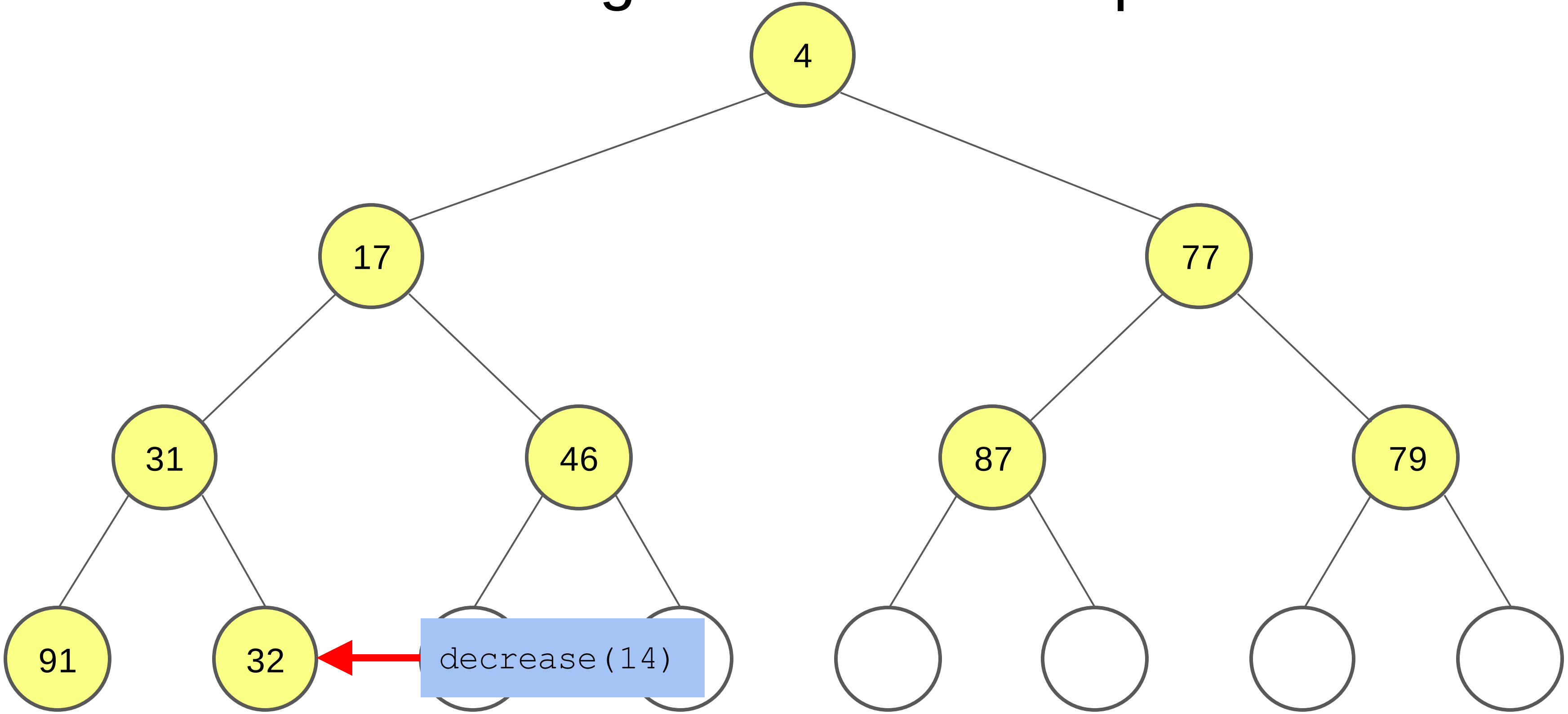
Consider the following values in a heap



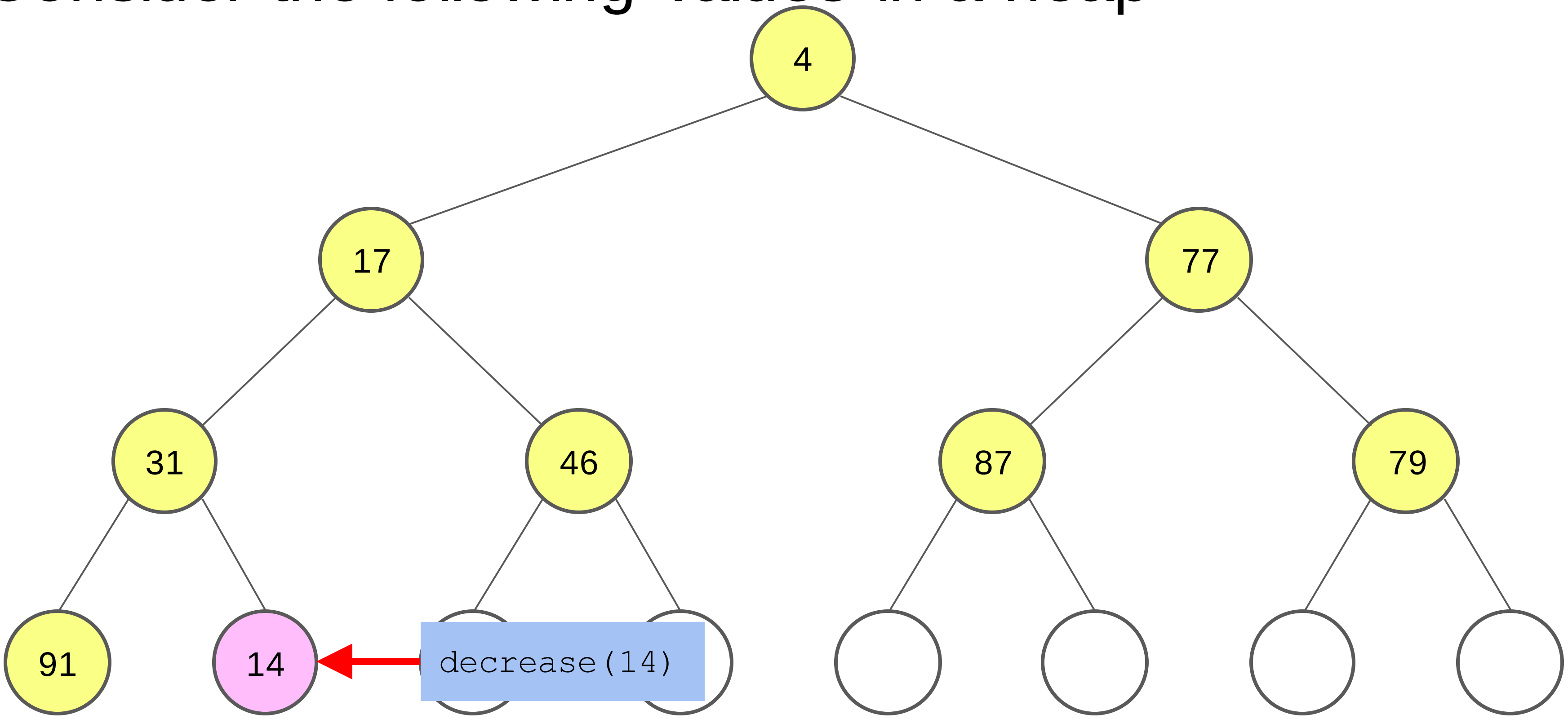
Consider the following values in a heap



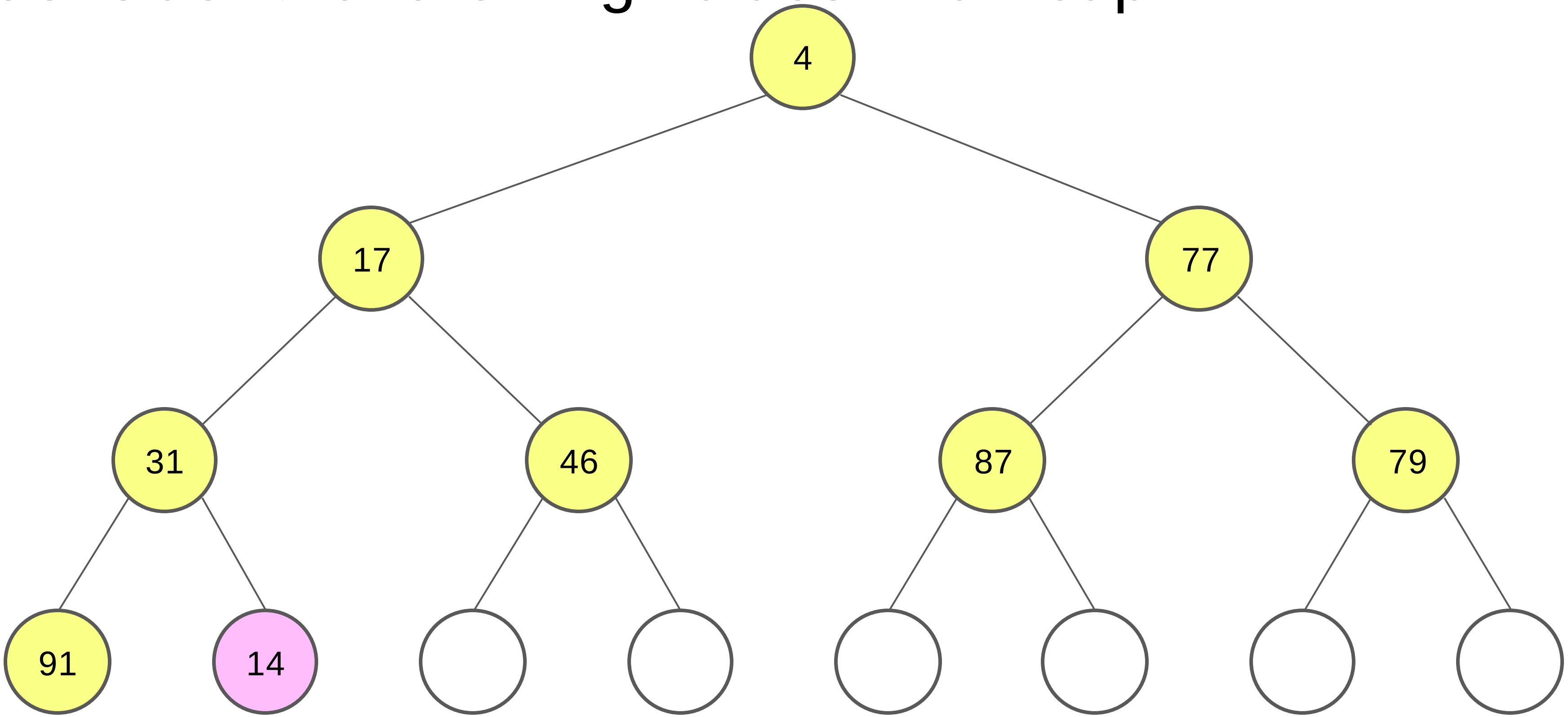
Consider the following values in a heap



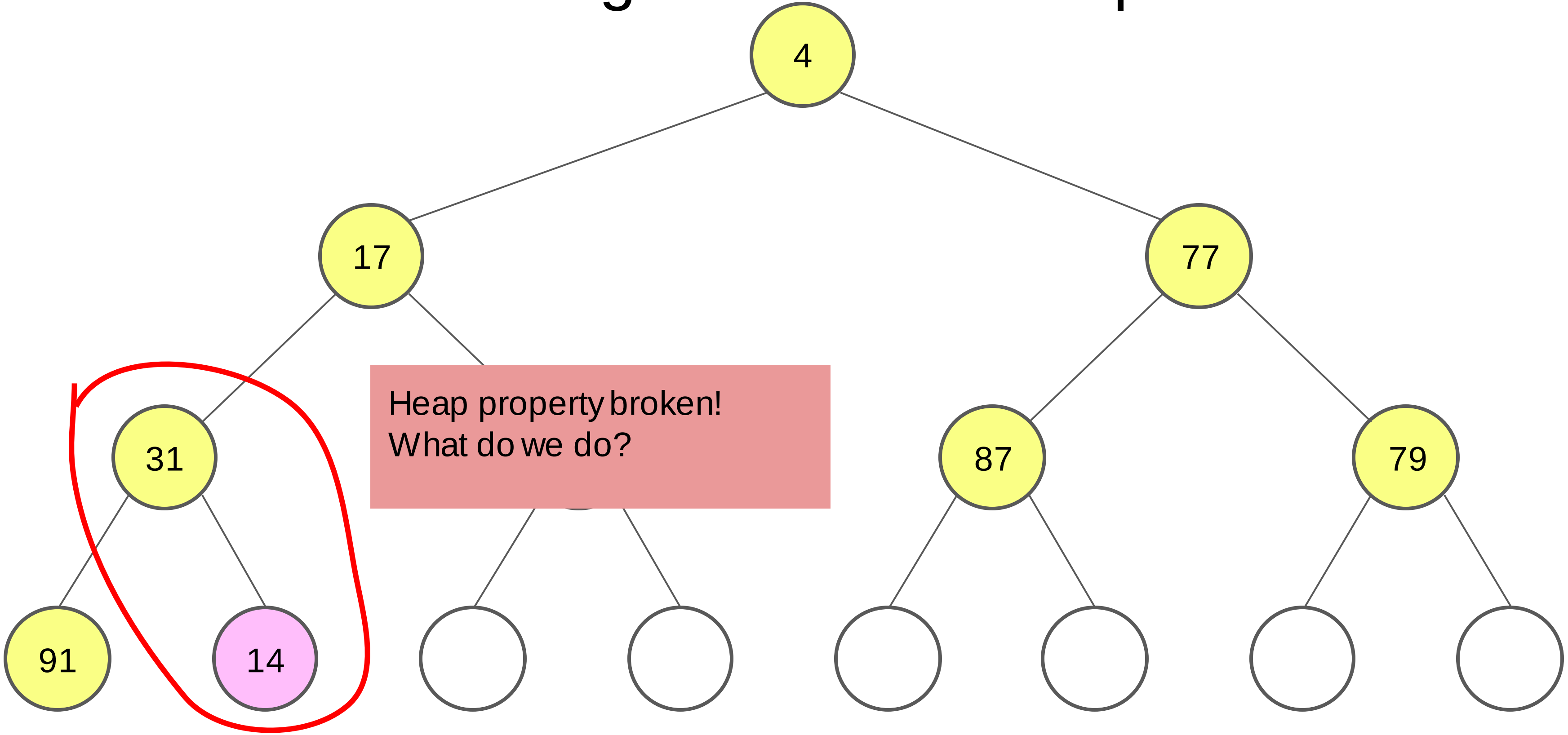
Consider the following values in a heap



Consider the following values in a heap

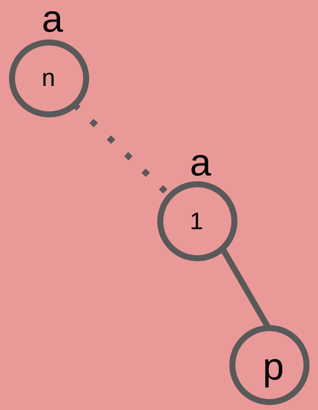


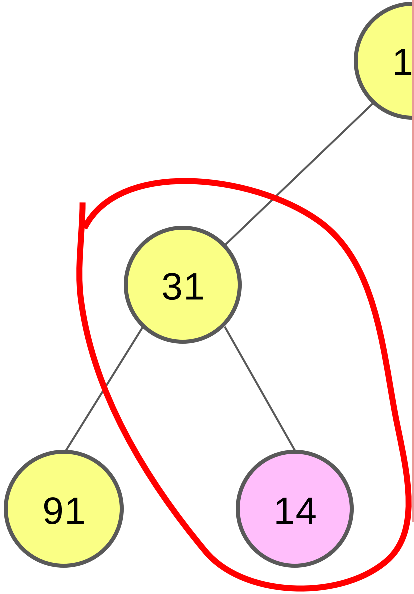
Consider the following values in a heap



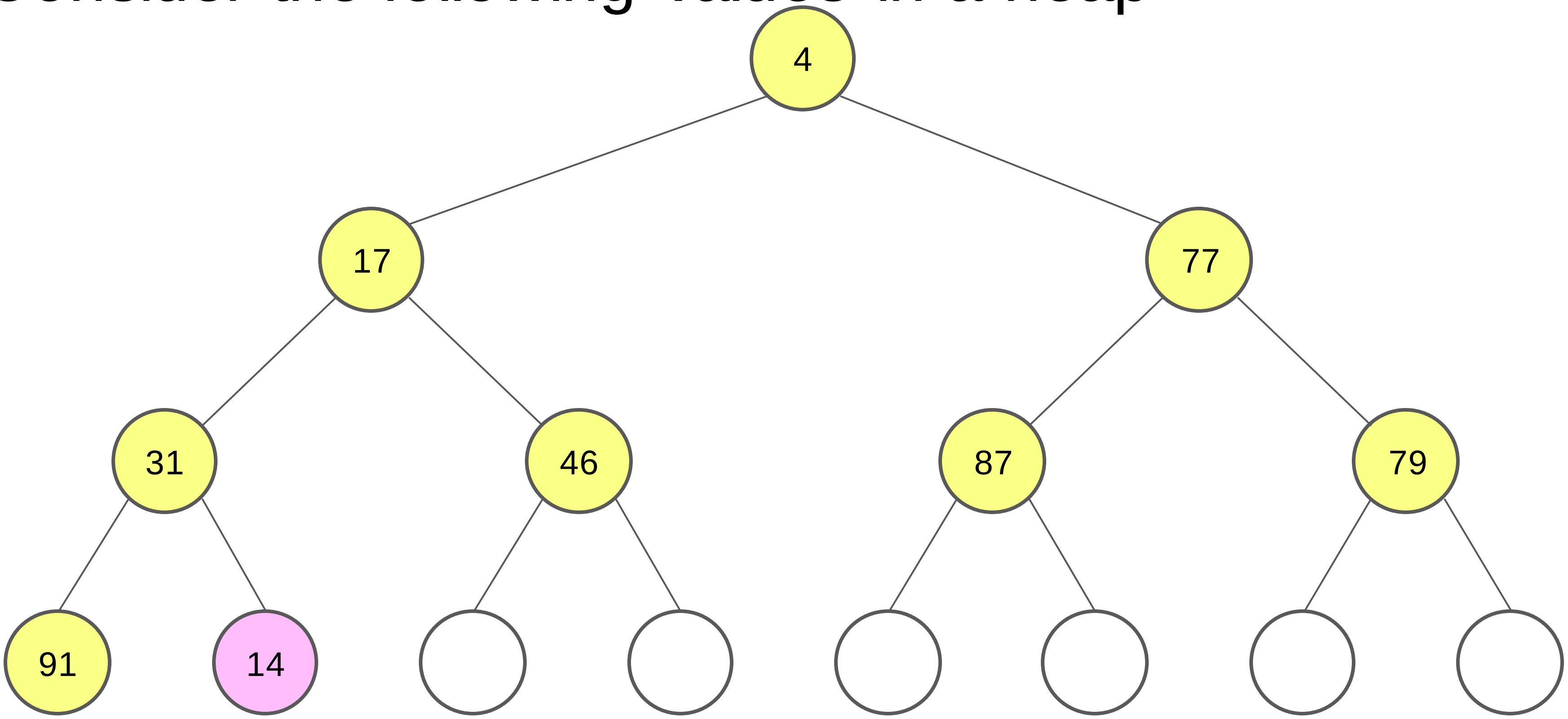
Consider the following values in a heap

- If $a_1 > p$
  - then heap property is broken
  - ...but swapping them makes those two values OK,
  - so we can keep swapping up the tree until the heap property is restored.

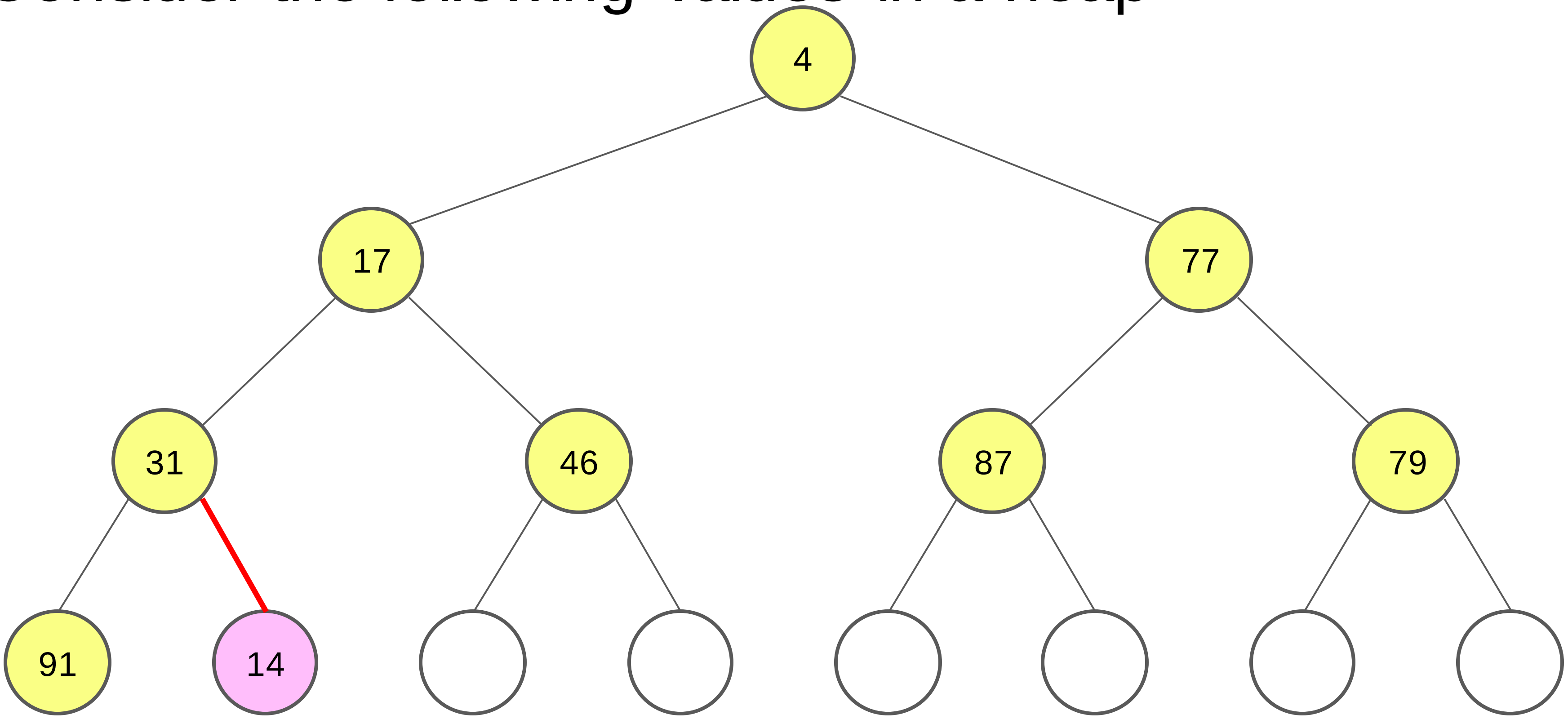




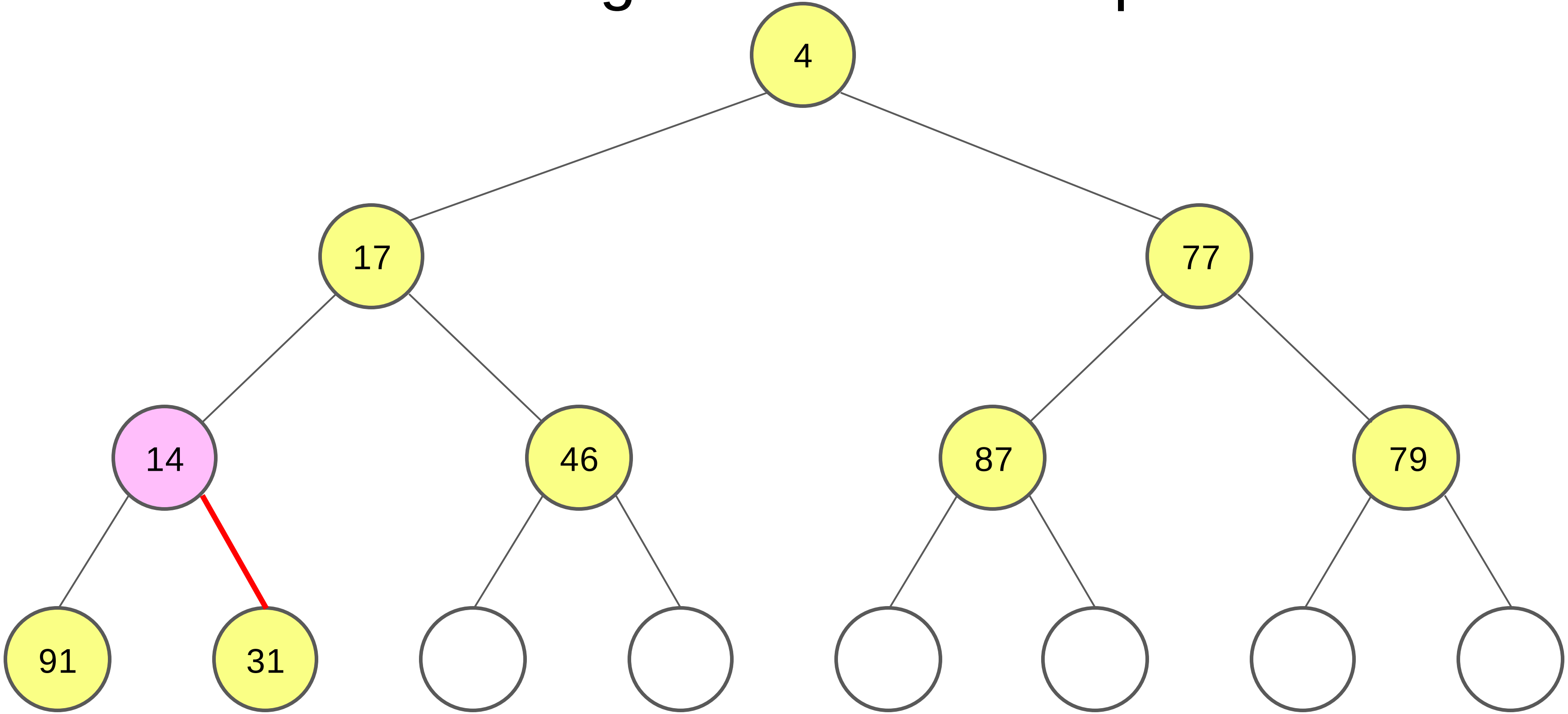
Consider the following values in a heap



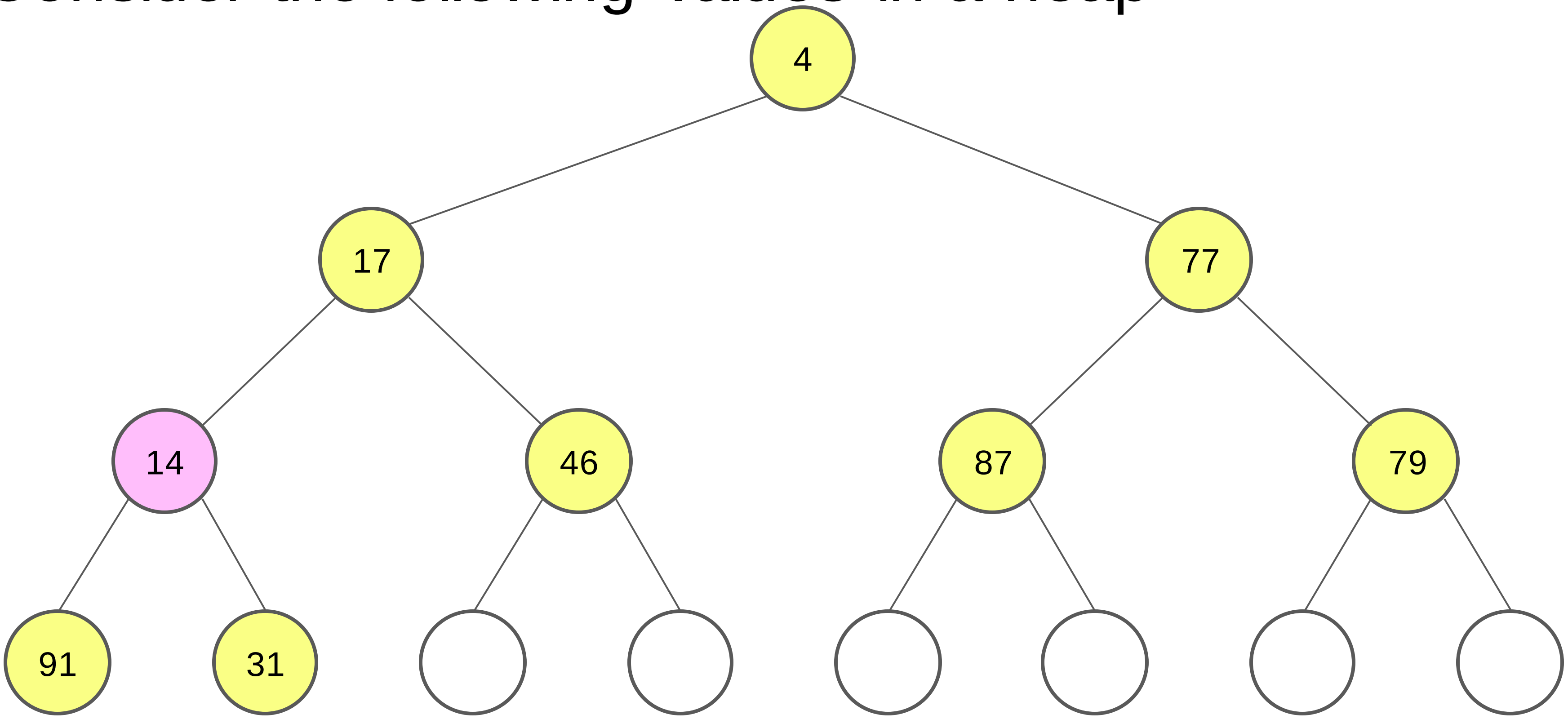
Consider the following values in a heap



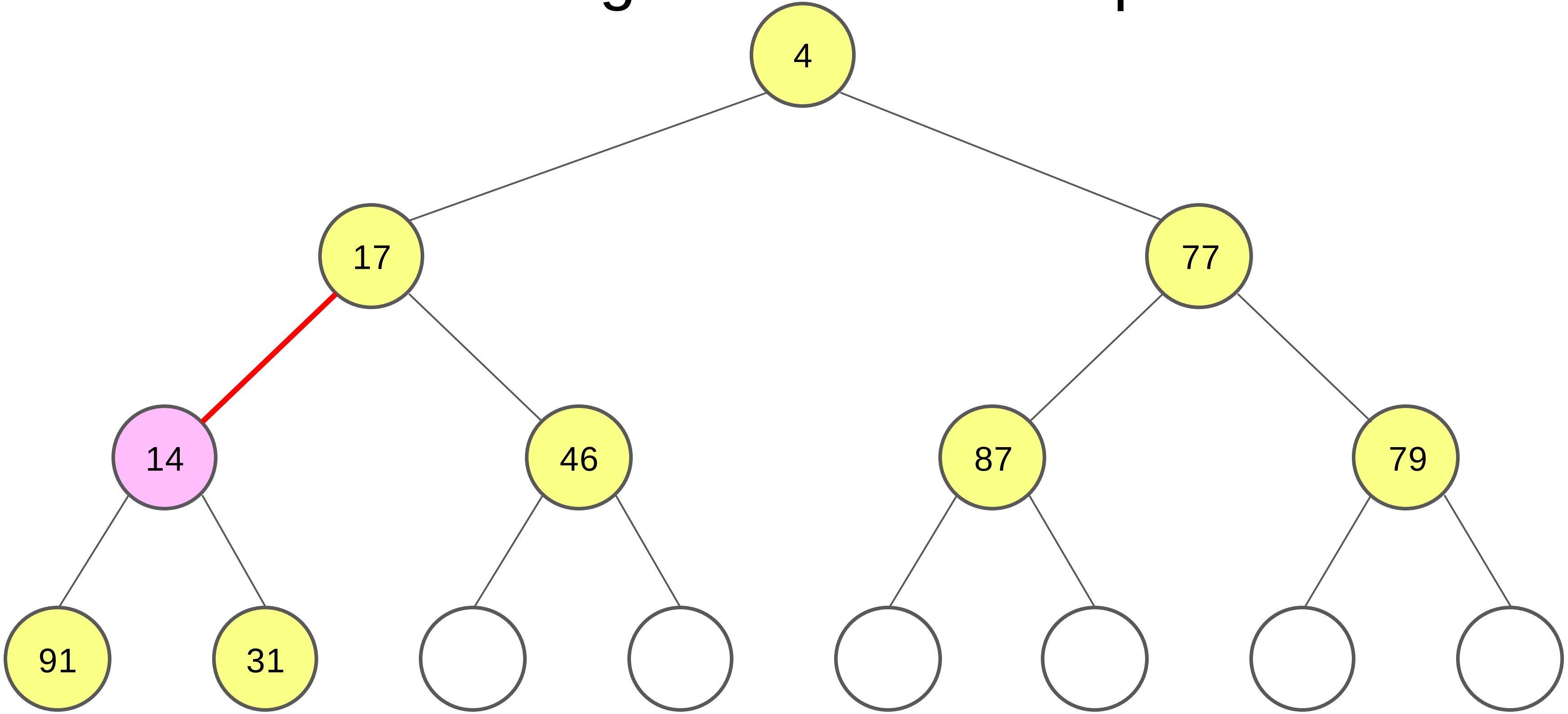
Consider the following values in a heap



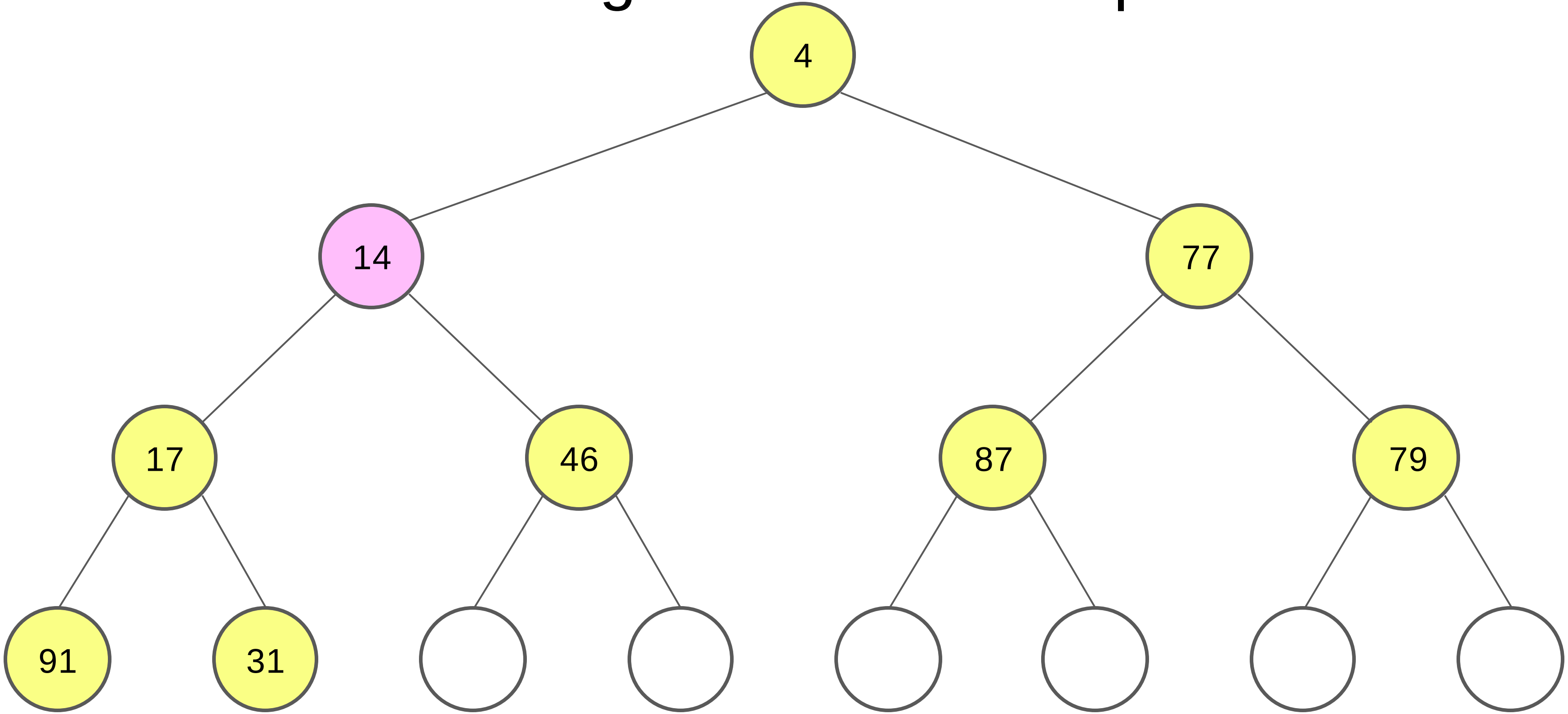
Consider the following values in a heap



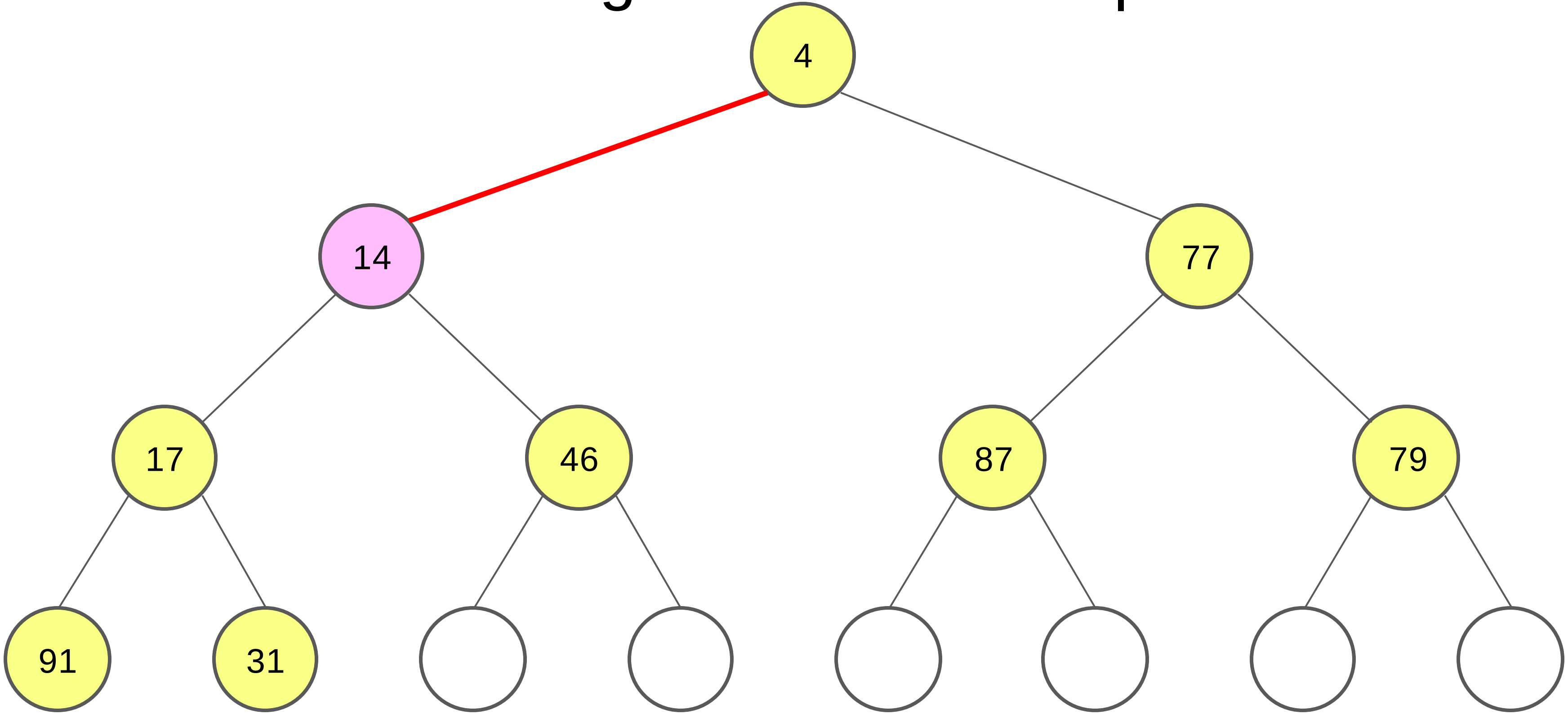
Consider the following values in a heap



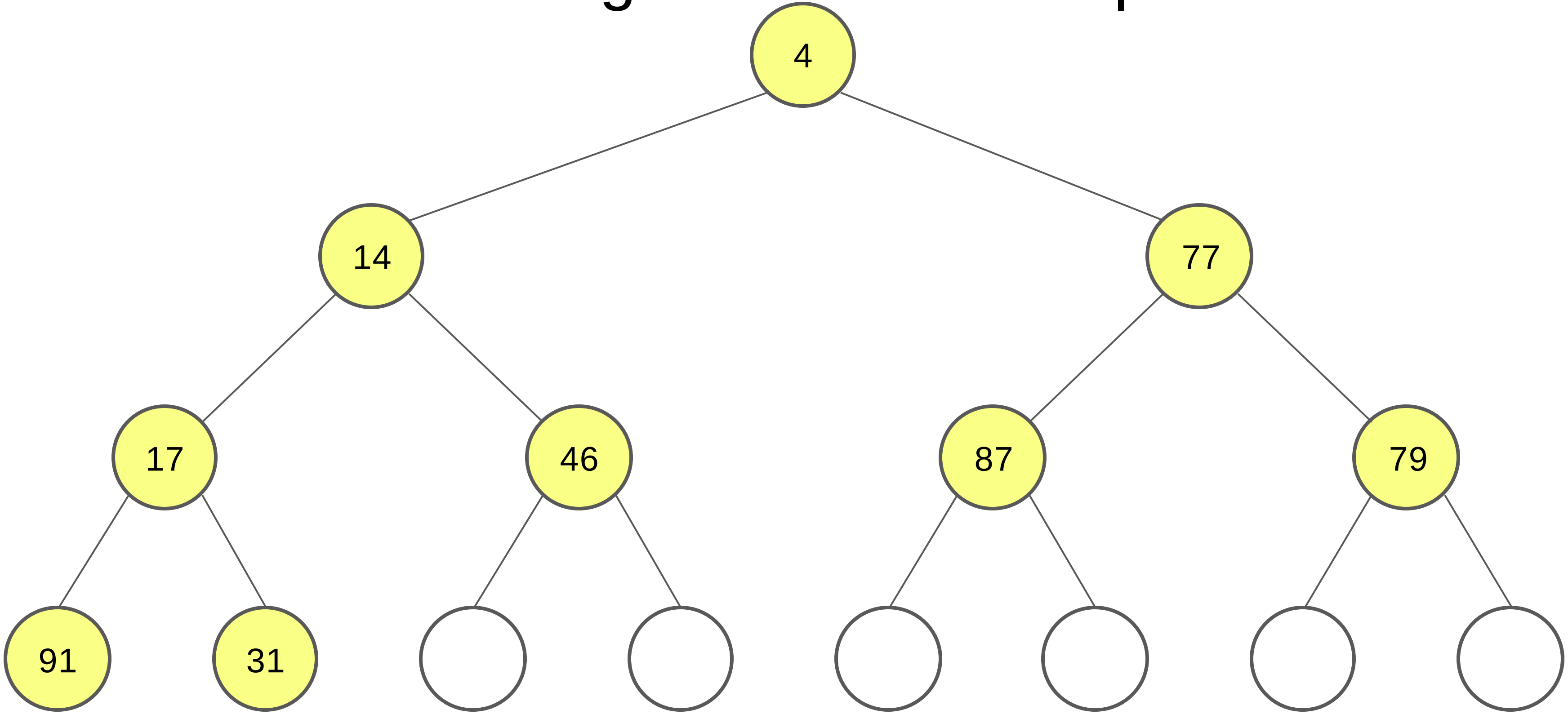
Consider the following values in a heap



Consider the following values in a heap



Consider the following values in a heap



OK, Now For Insertion

- **Claim**: if you can do decrease(), you can do insert()!

OK, Now For Insertion

- **Claim**: if you can do decrease(), you can do insert()!

Algorithm for insert?

OK, Now For Insertion

- **Claim**: if you can do decrease(), you can do insert()!

Algorithm for insert?

Algorithm for decrease

OK, Now For Insertion

- **Claim**: if you can do decrease(), you can do insert()!

OK, Now For Insertion

- **Claim**: if you can do decrease(), you can do insert()!

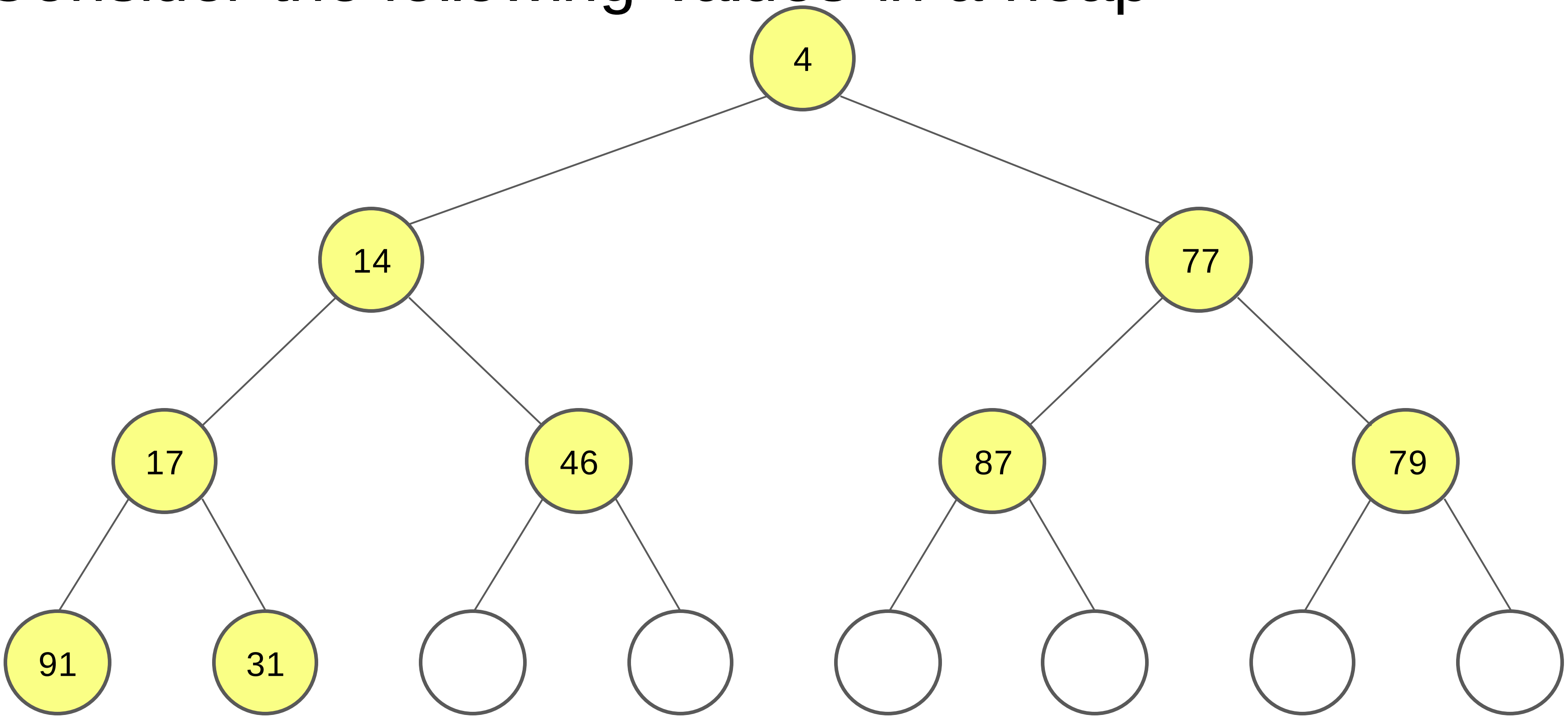
- Will argue that insert() reduces to decrease()

OK, Now For Insertion

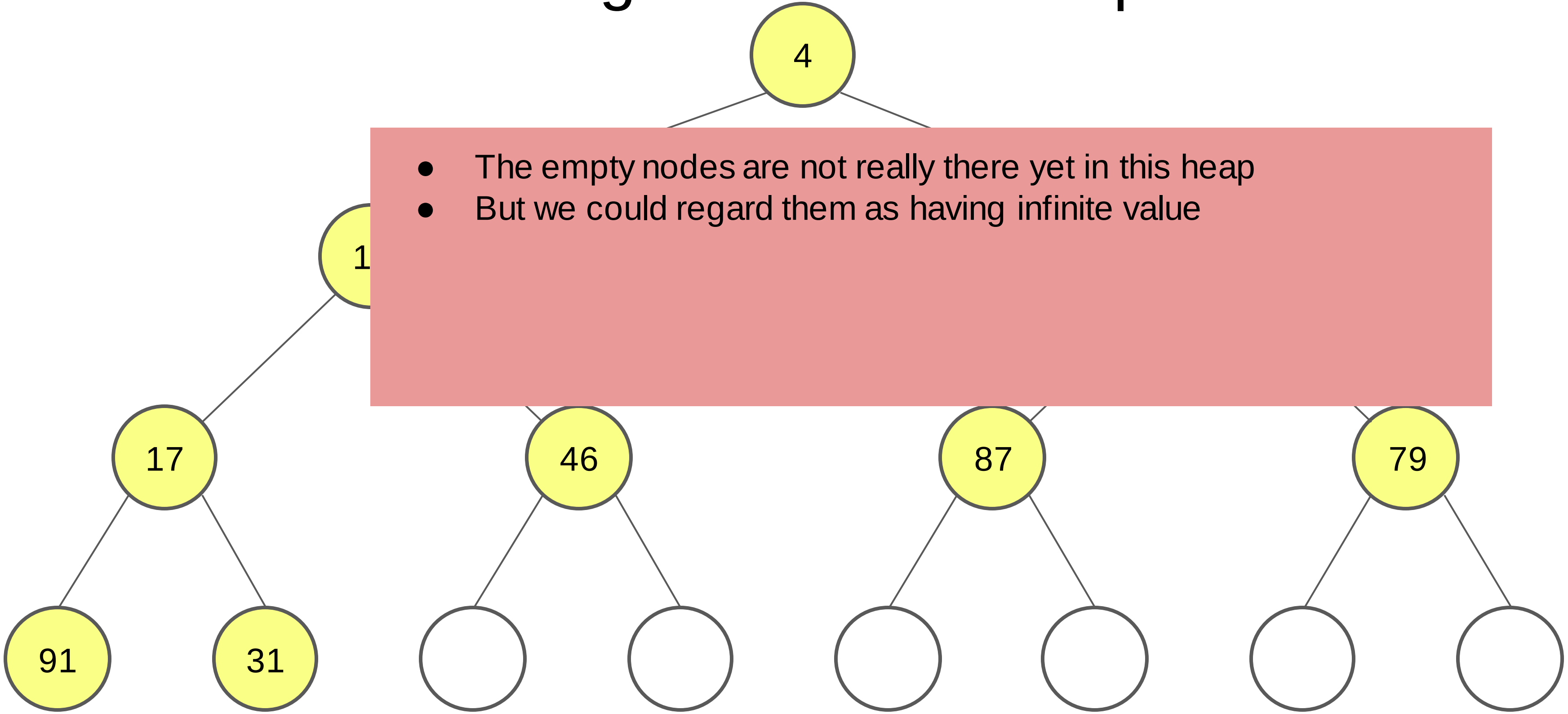
- **Claim**: if you can do decrease(), you can do insert()!

- Will argue that insert() reduces to decrease()
- [“*If you can do decrease, here’s how to use it for insert*”]

Consider the following values in a heap

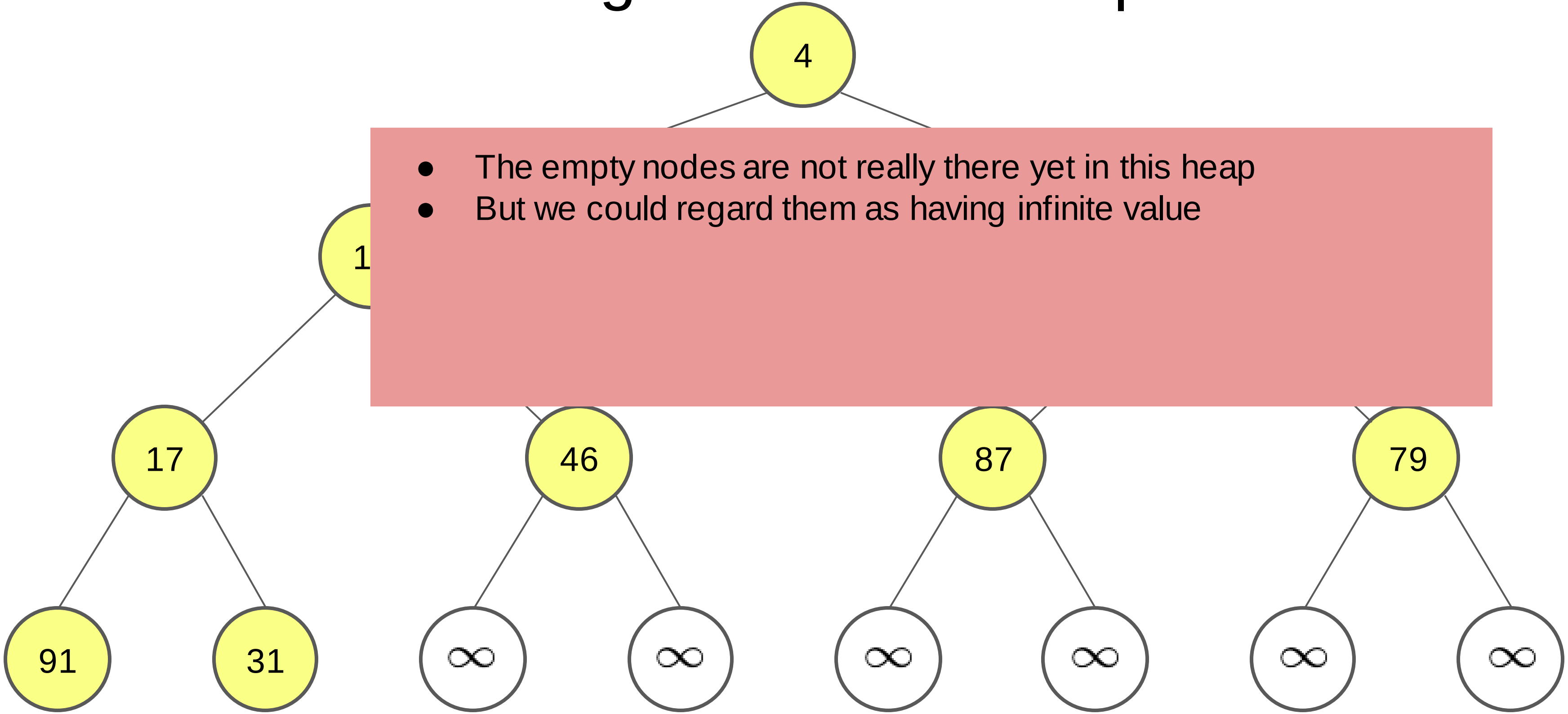


Consider the following values in a heap



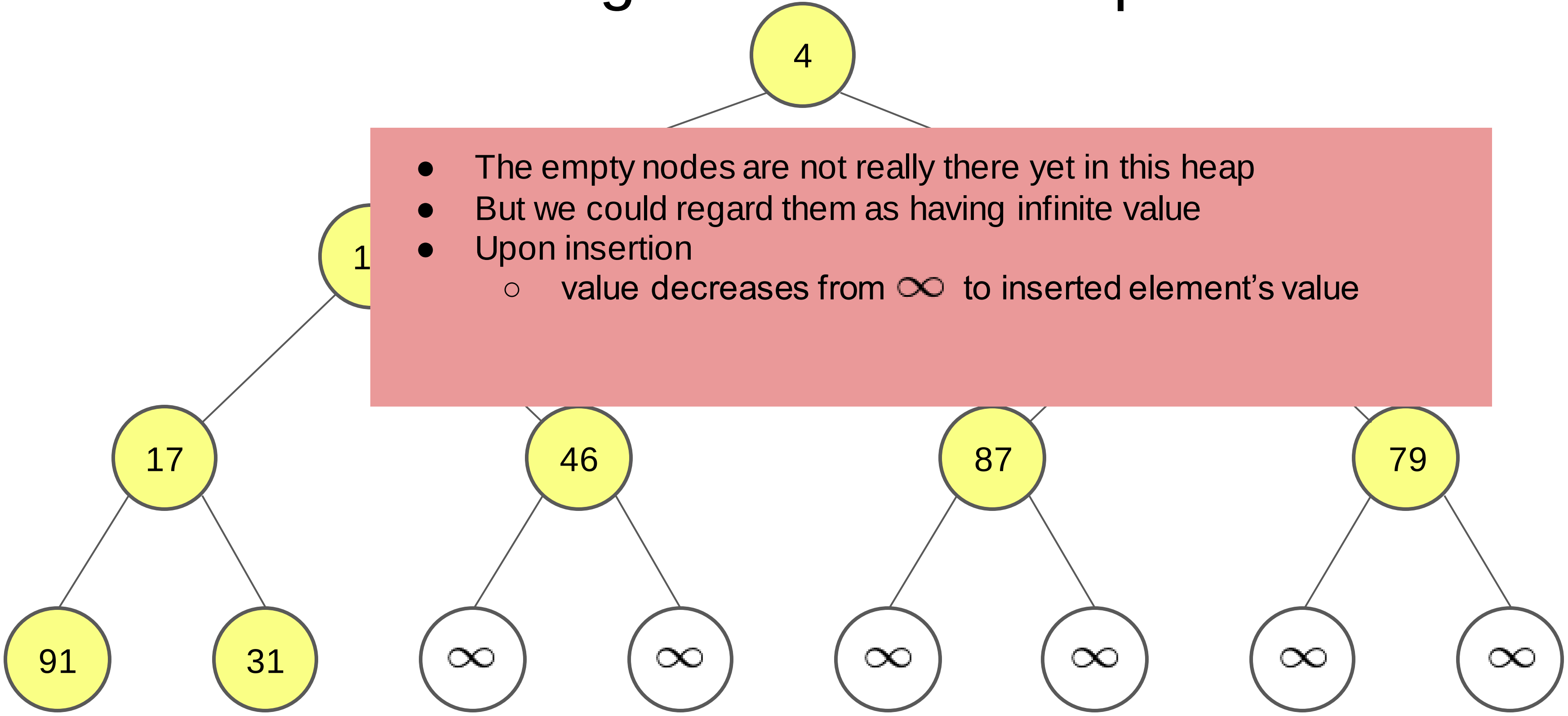
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value

Consider the following values in a heap



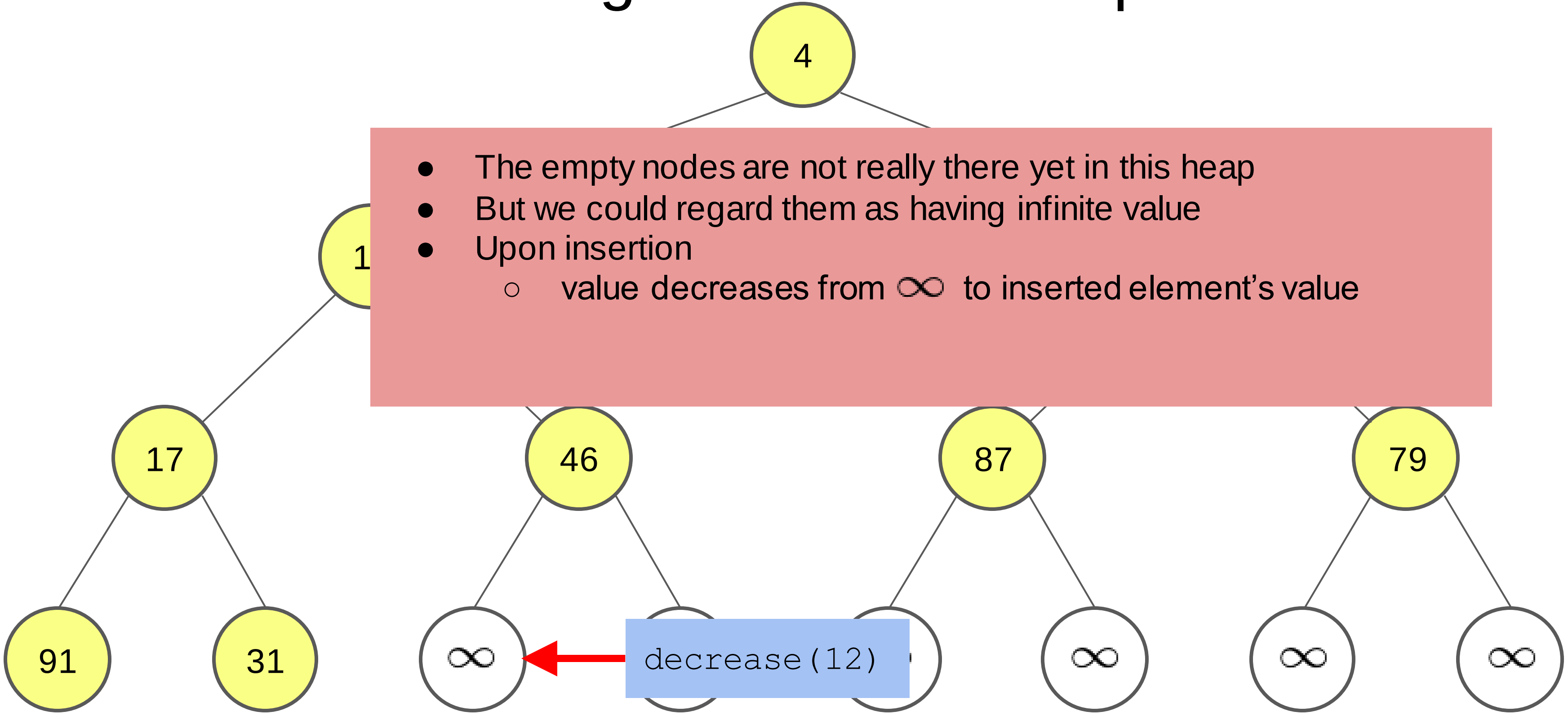
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value

Consider the following values in a heap



- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value

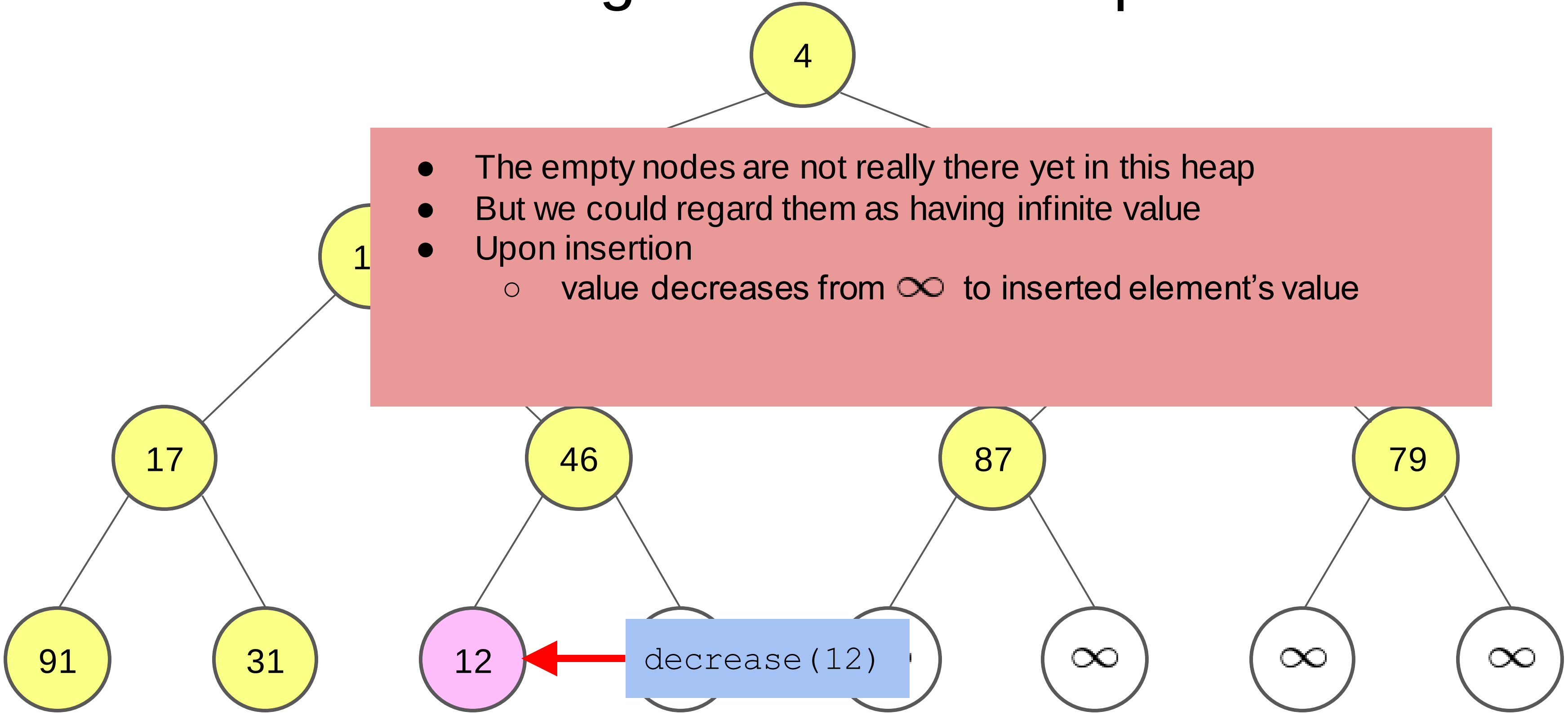
Consider the following values in a heap



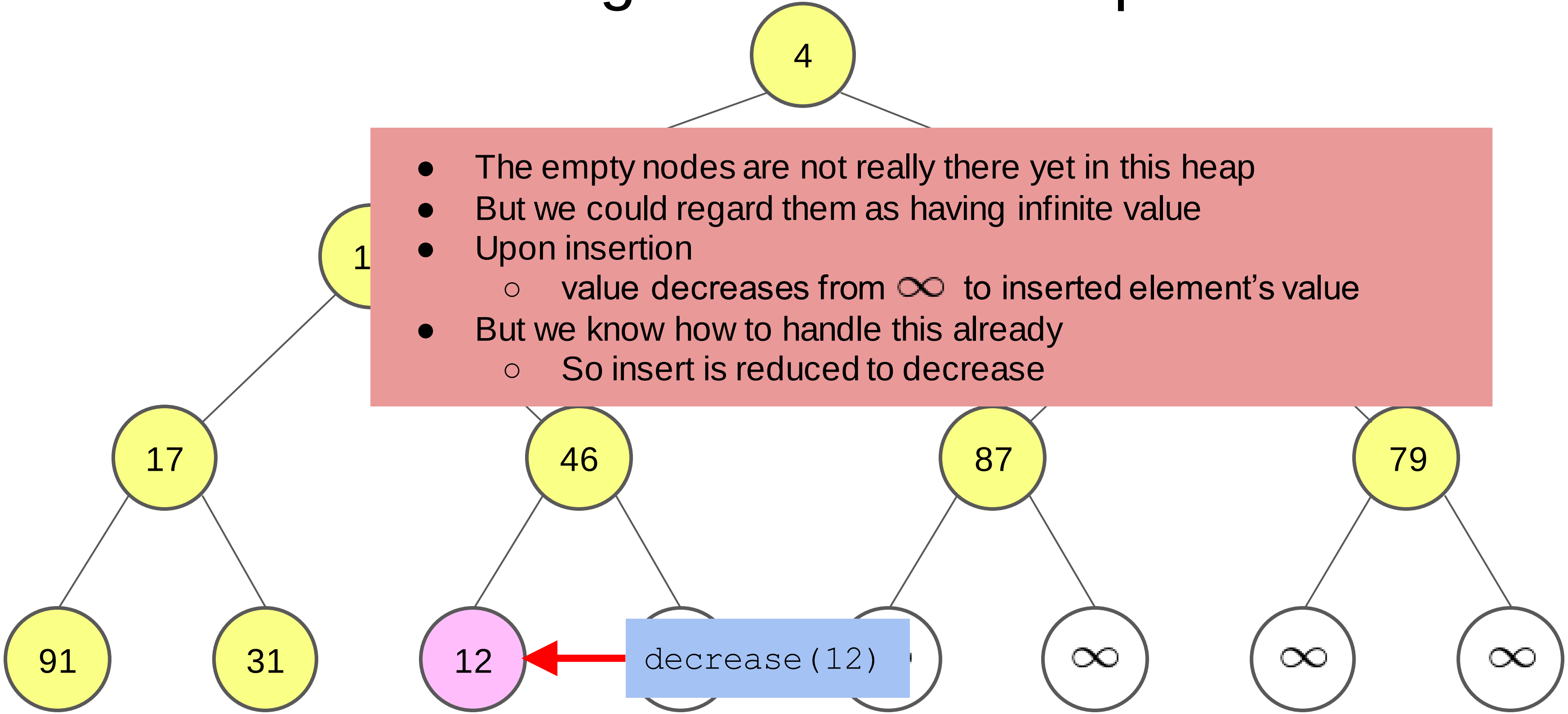
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value

Consider the following values in a heap

- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value

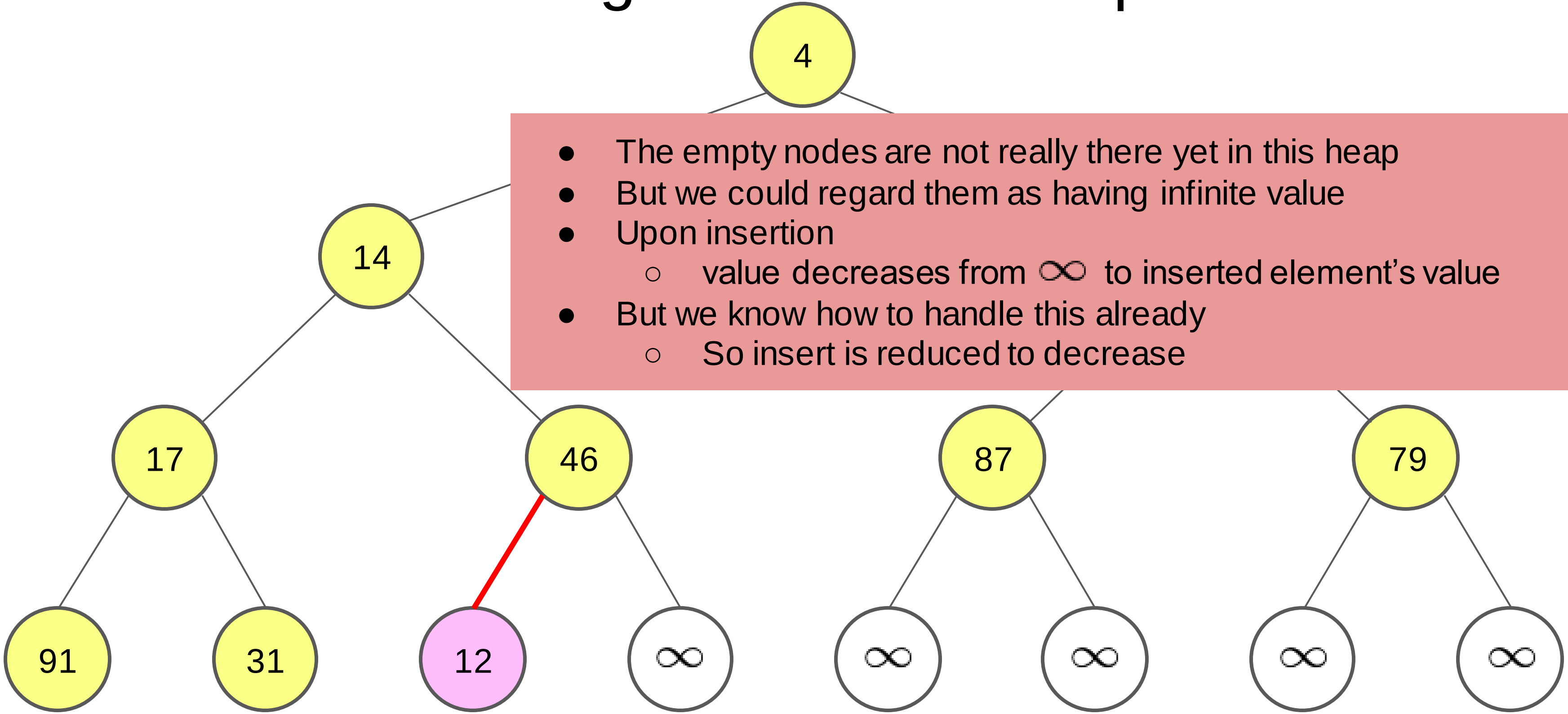


Consider the following values in a heap



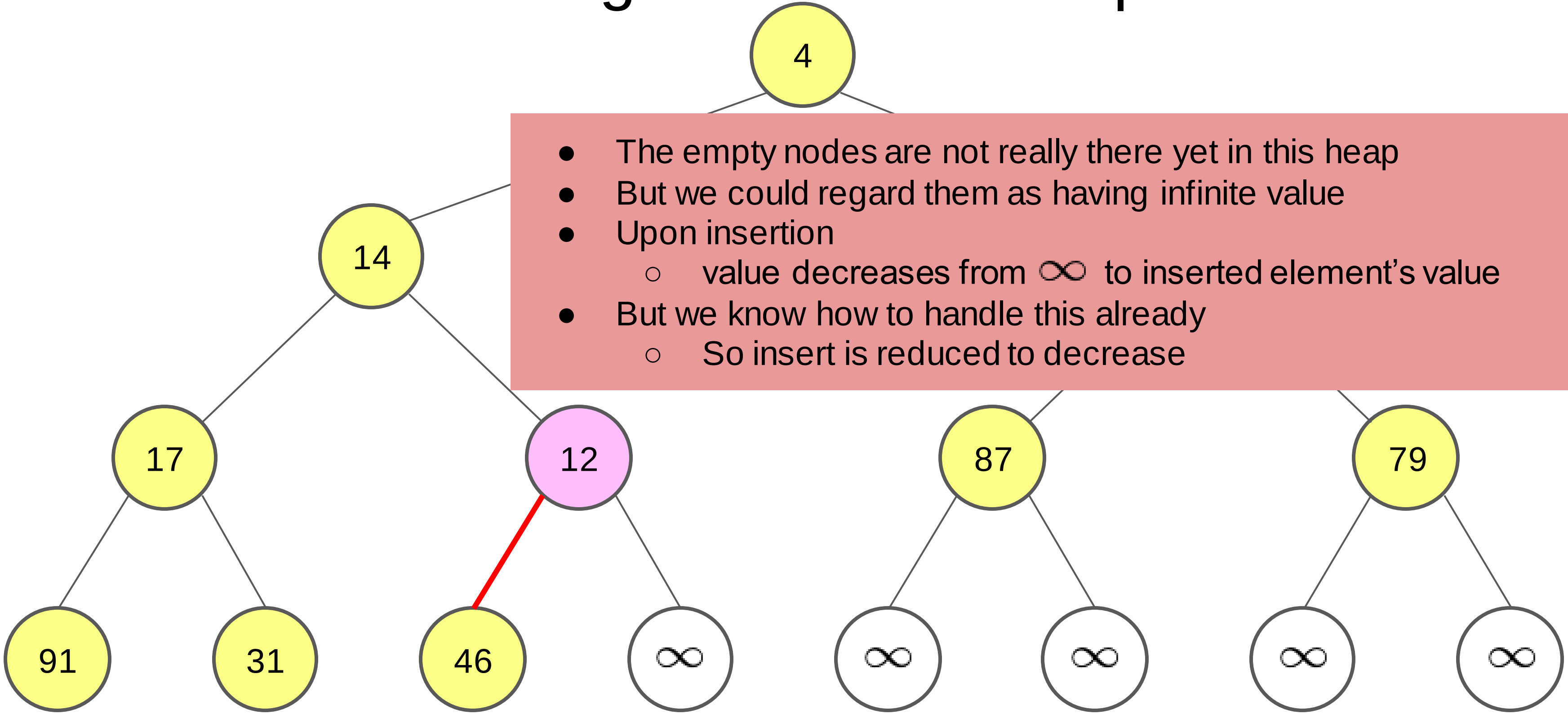
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



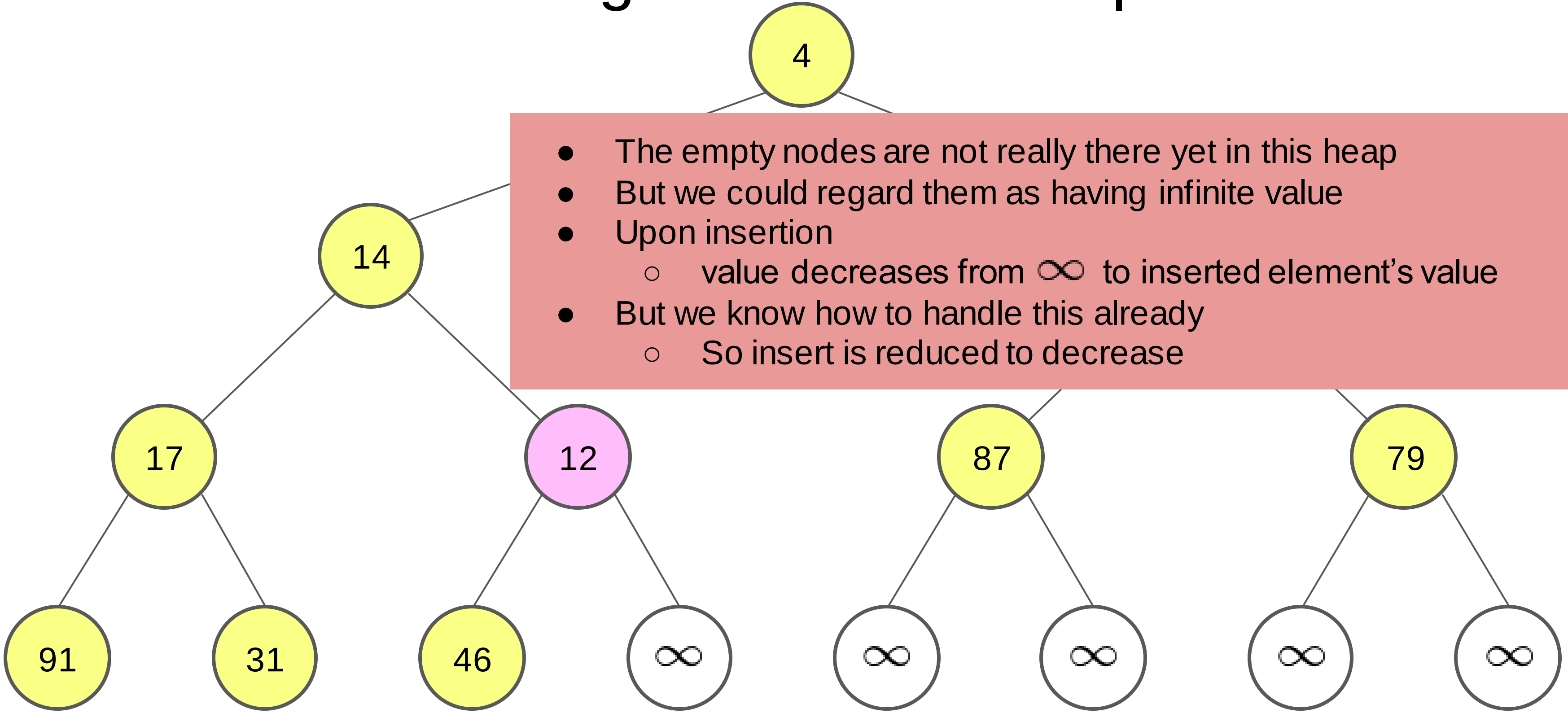
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



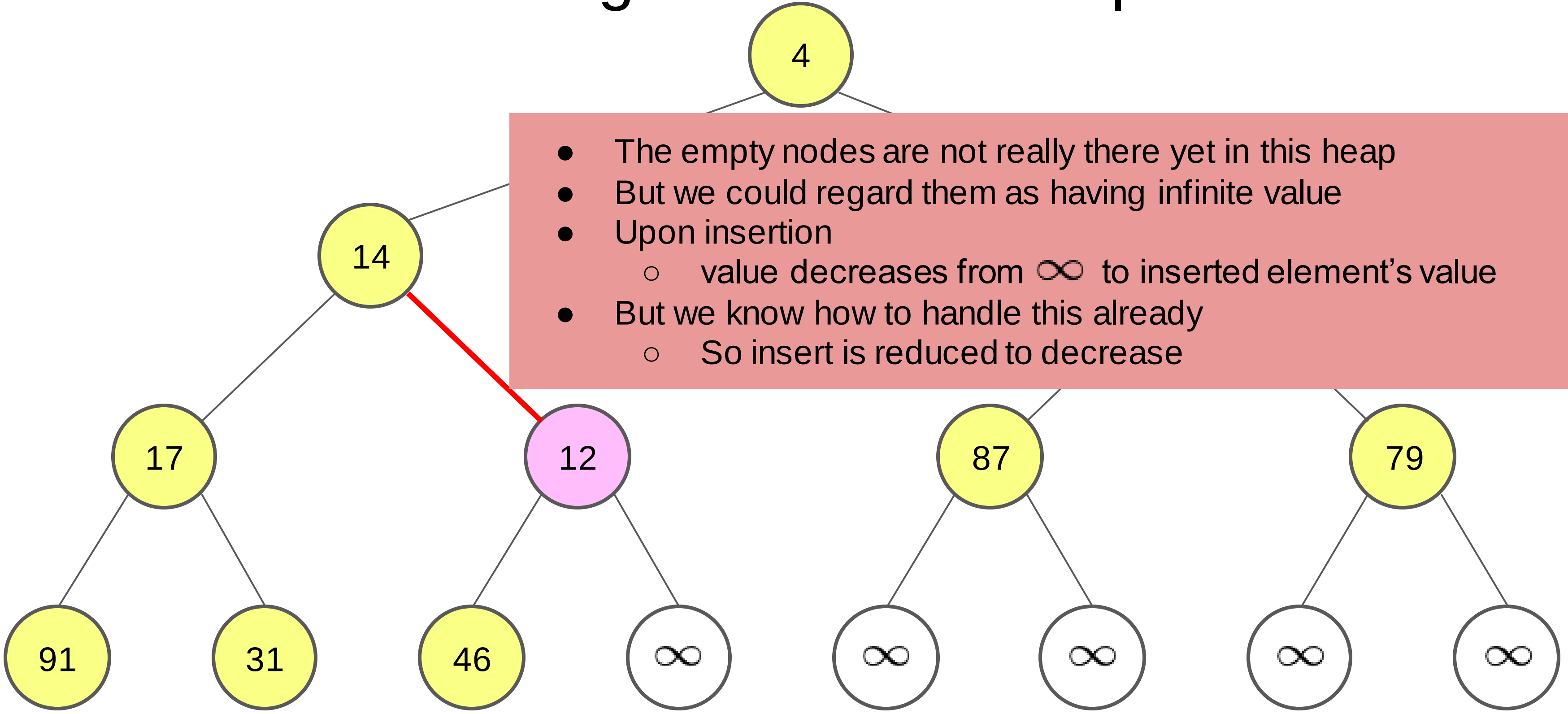
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



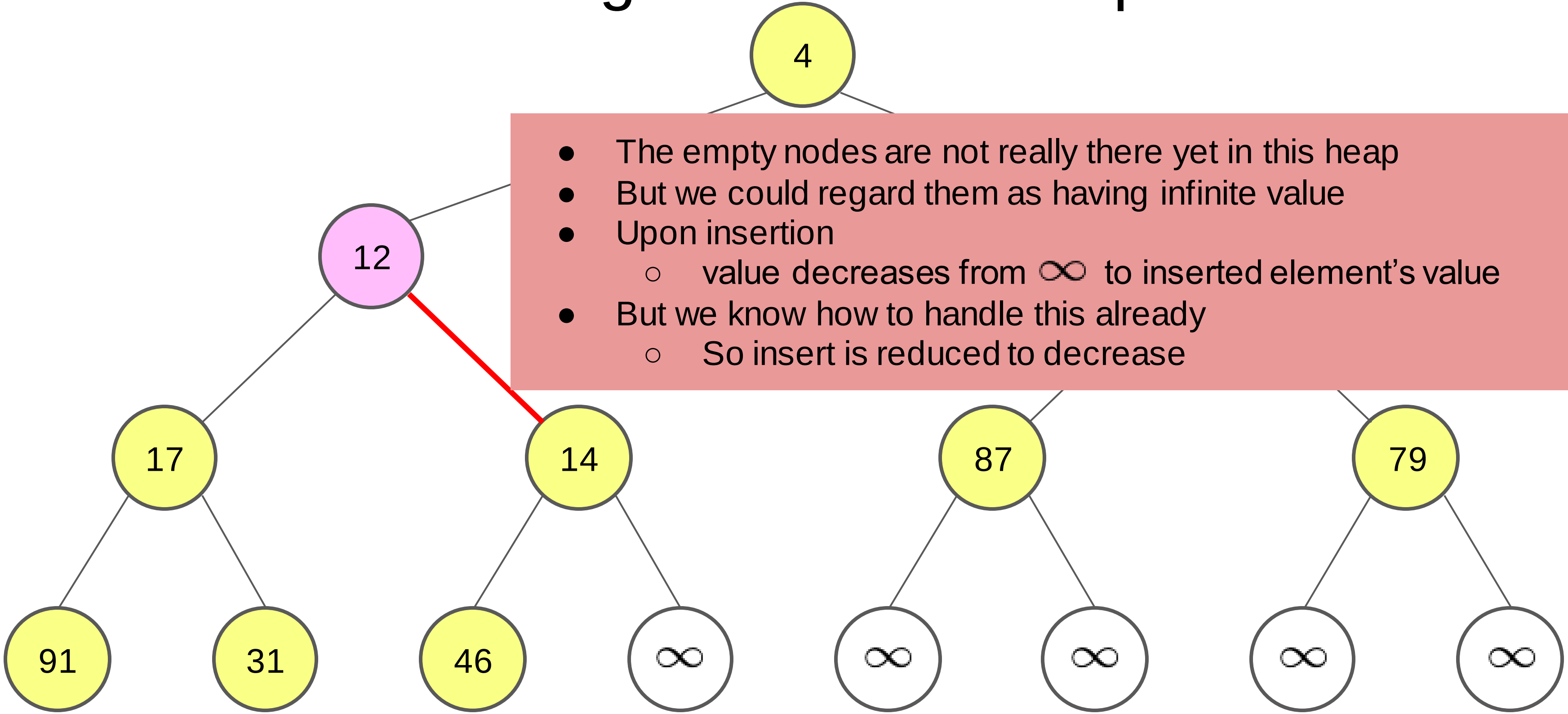
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



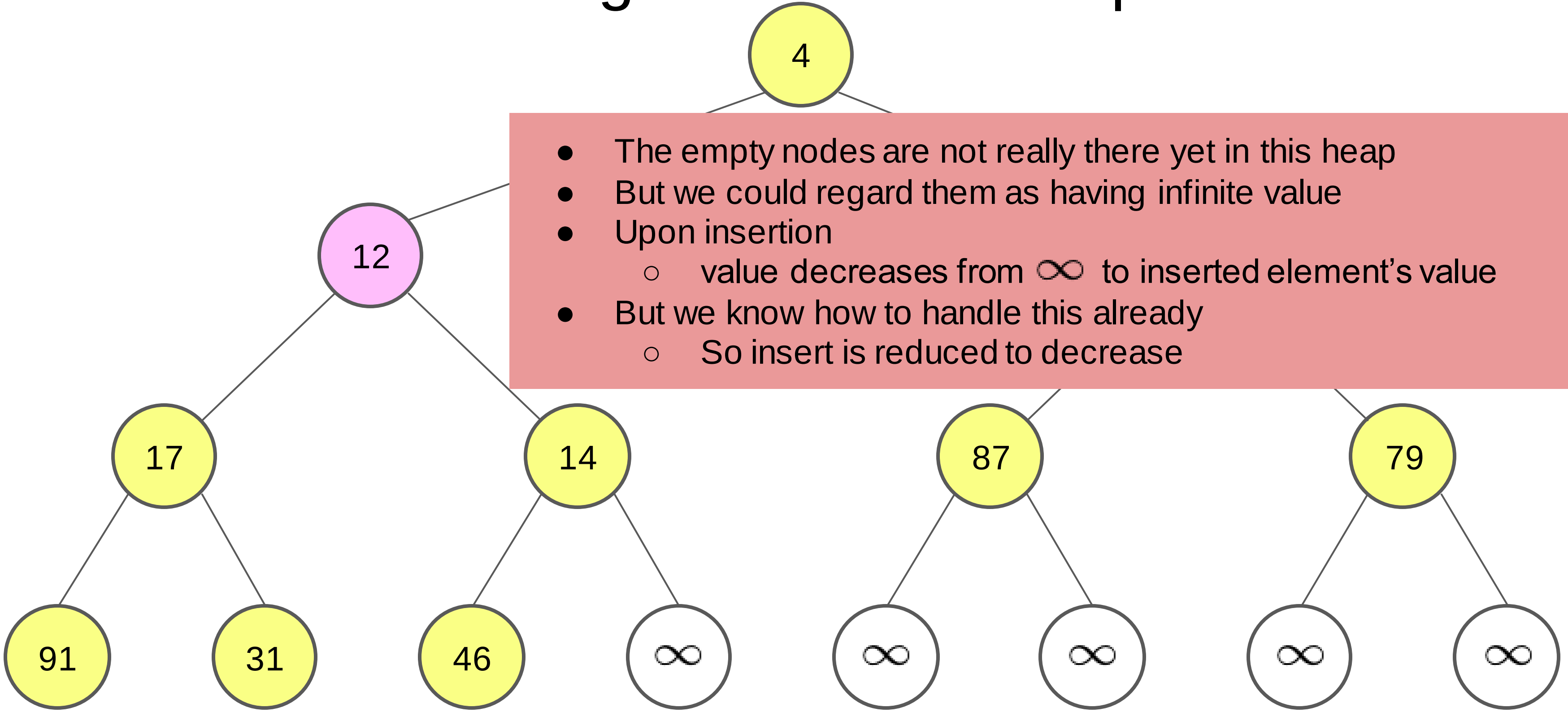
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



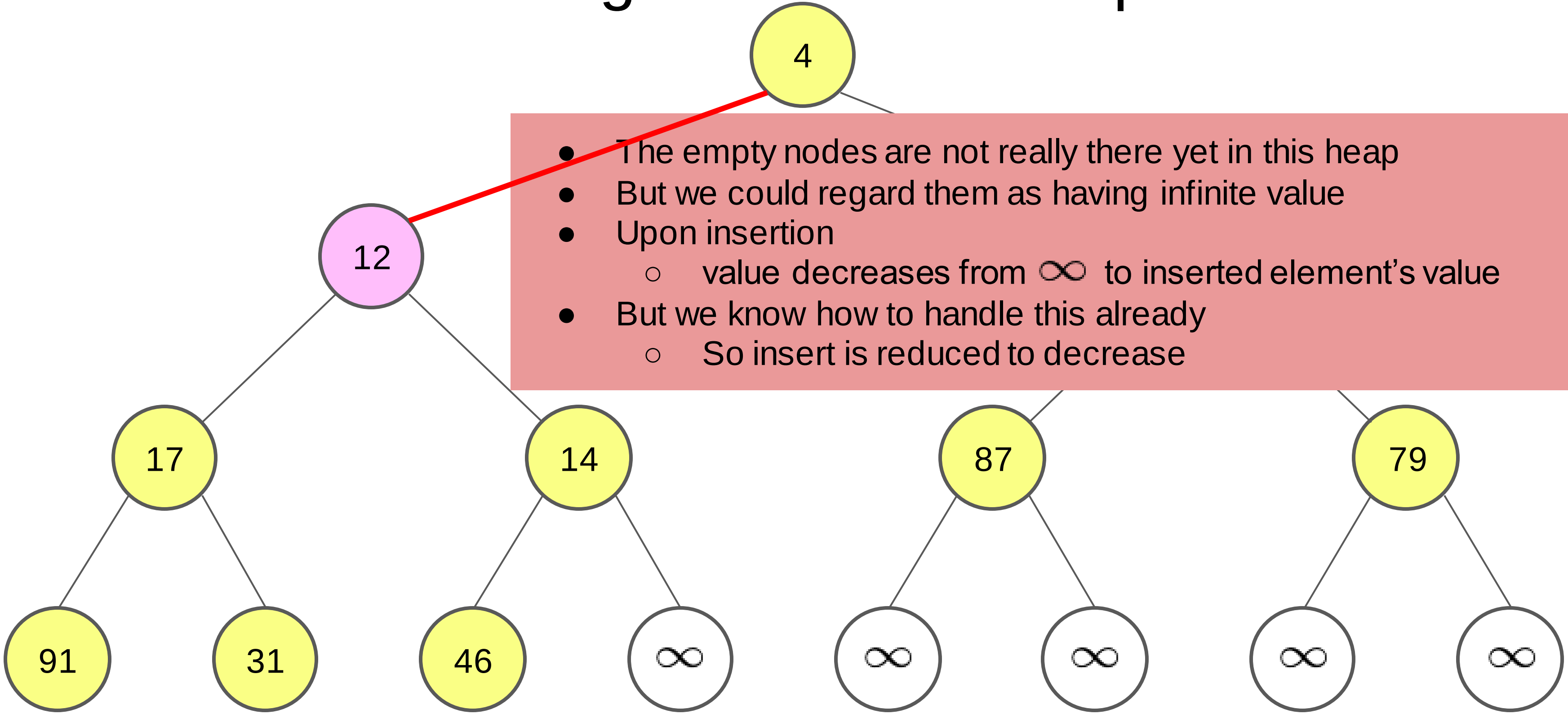
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



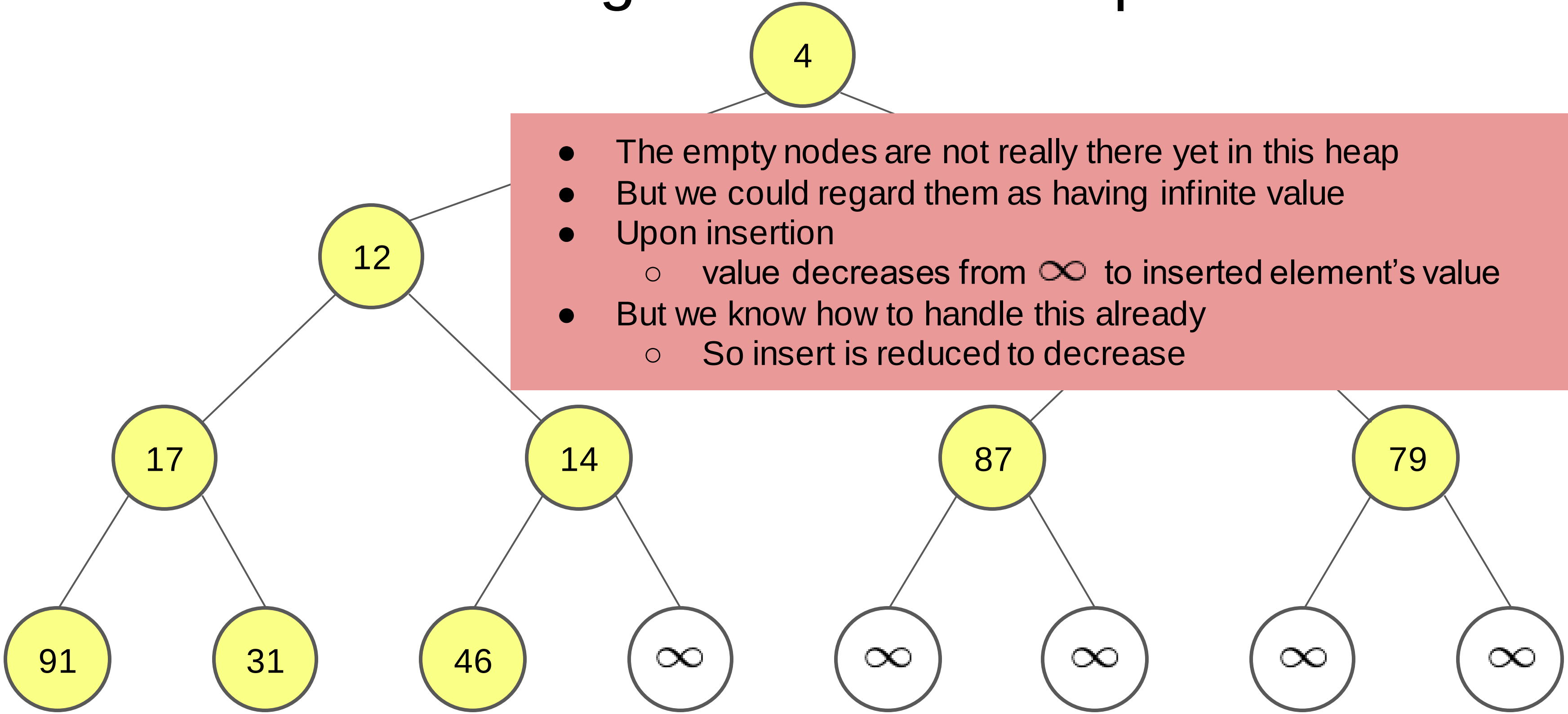
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



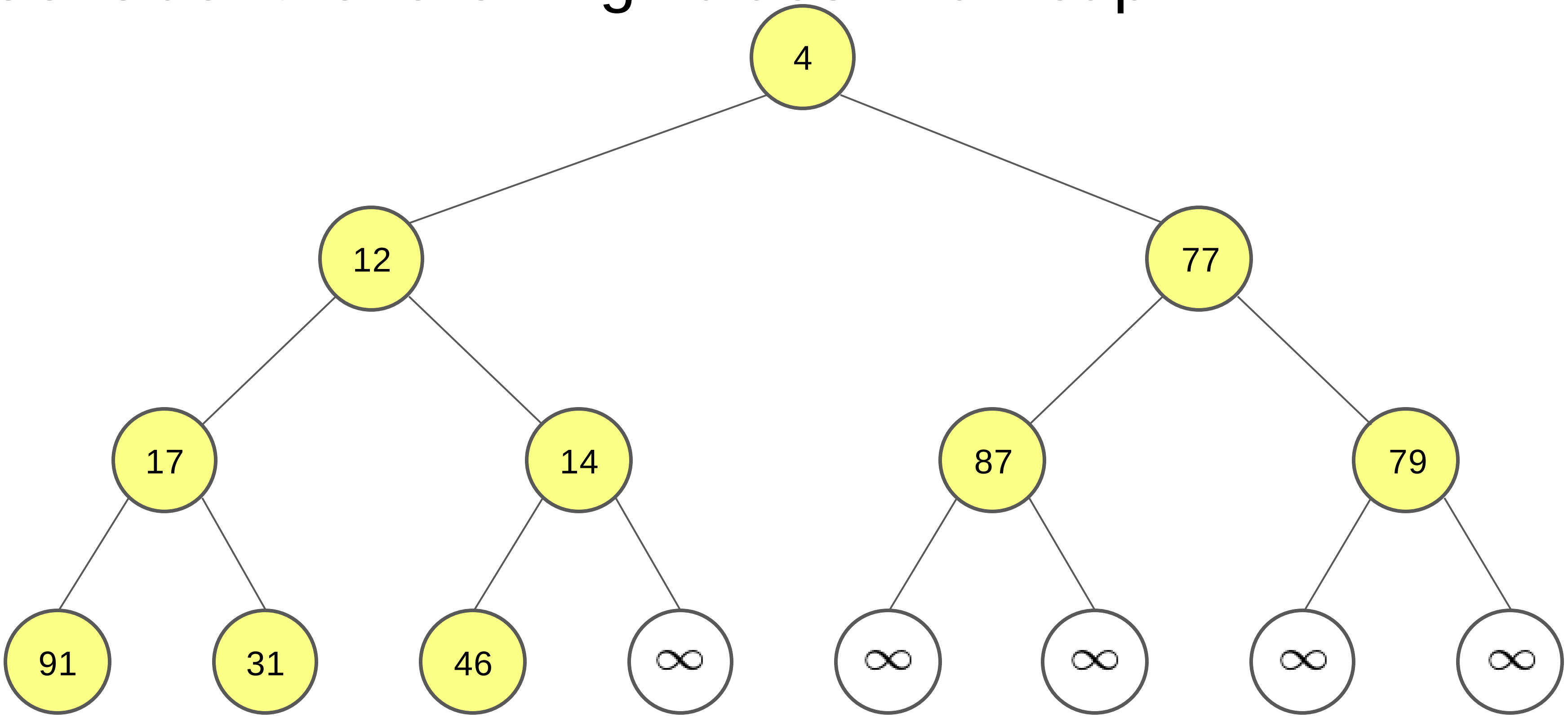
- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

Consider the following values in a heap



- The empty nodes are not really there yet in this heap
- But we could regard them as having infinite value
- Upon insertion
  - value decreases from ∞ to inserted element's value
- But we know how to handle this already
  - So insert is reduced to decrease

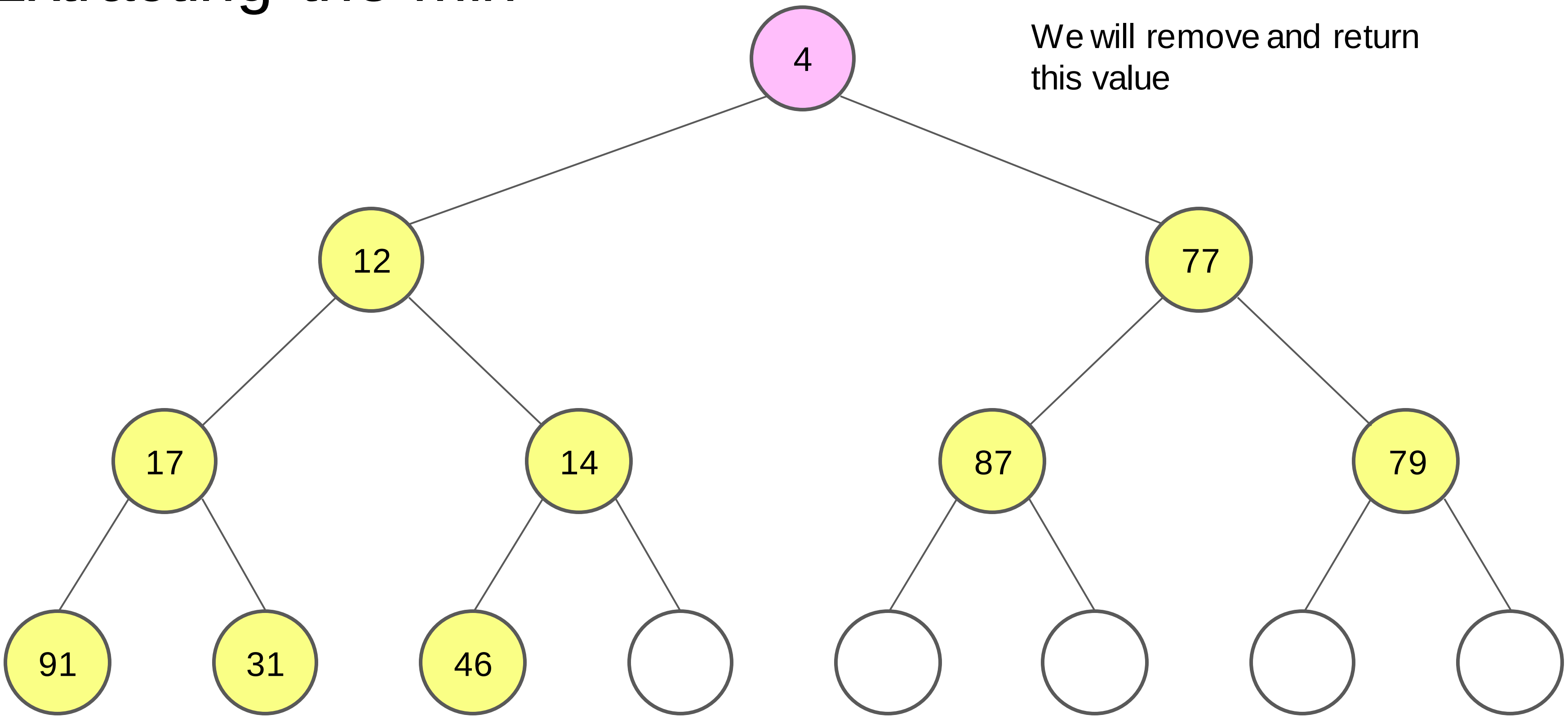
Consider the following values in a heap



One More Operation

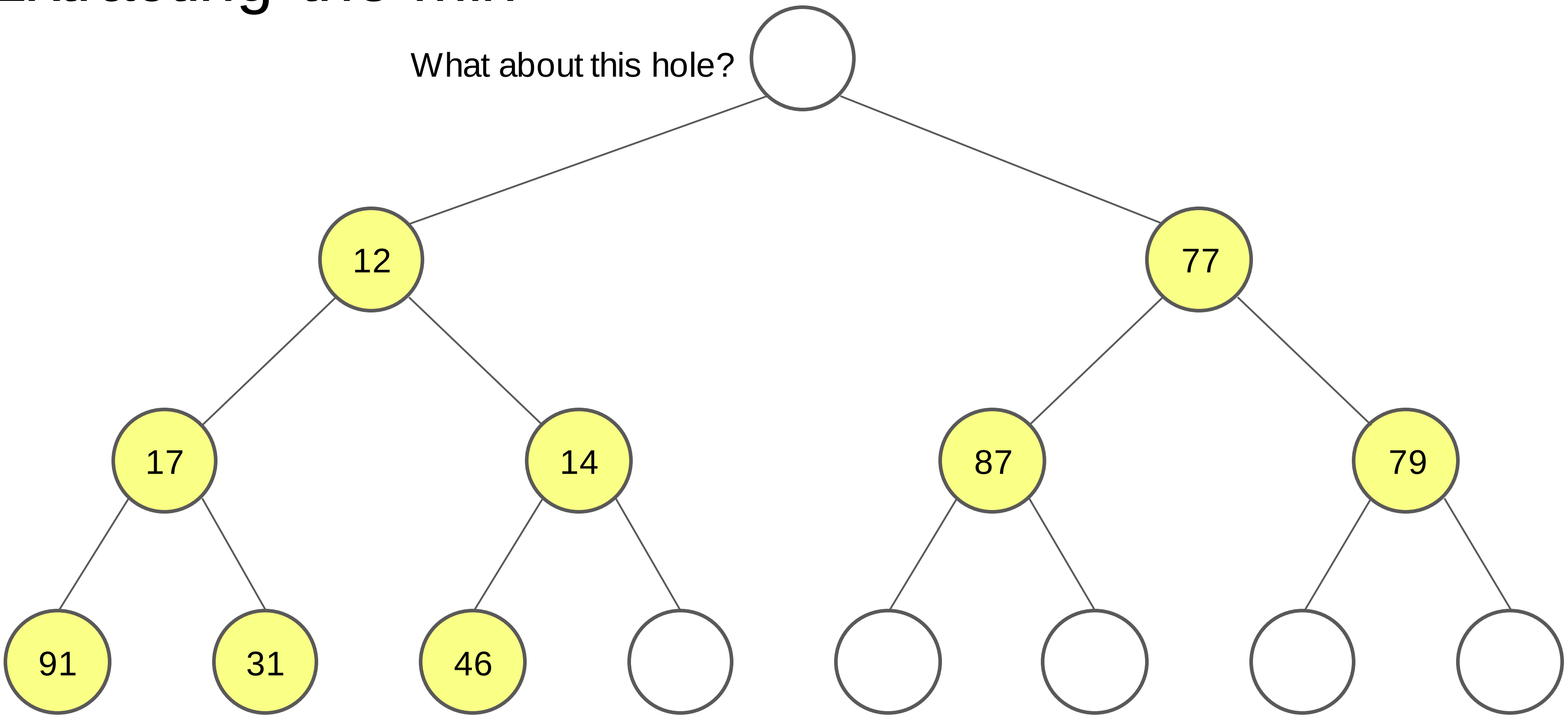
- extractMin – remove smallest element of heap
- We know where the smallest element is… [root]
- But once we remove it, tree is no longer compact!

Extracting the min



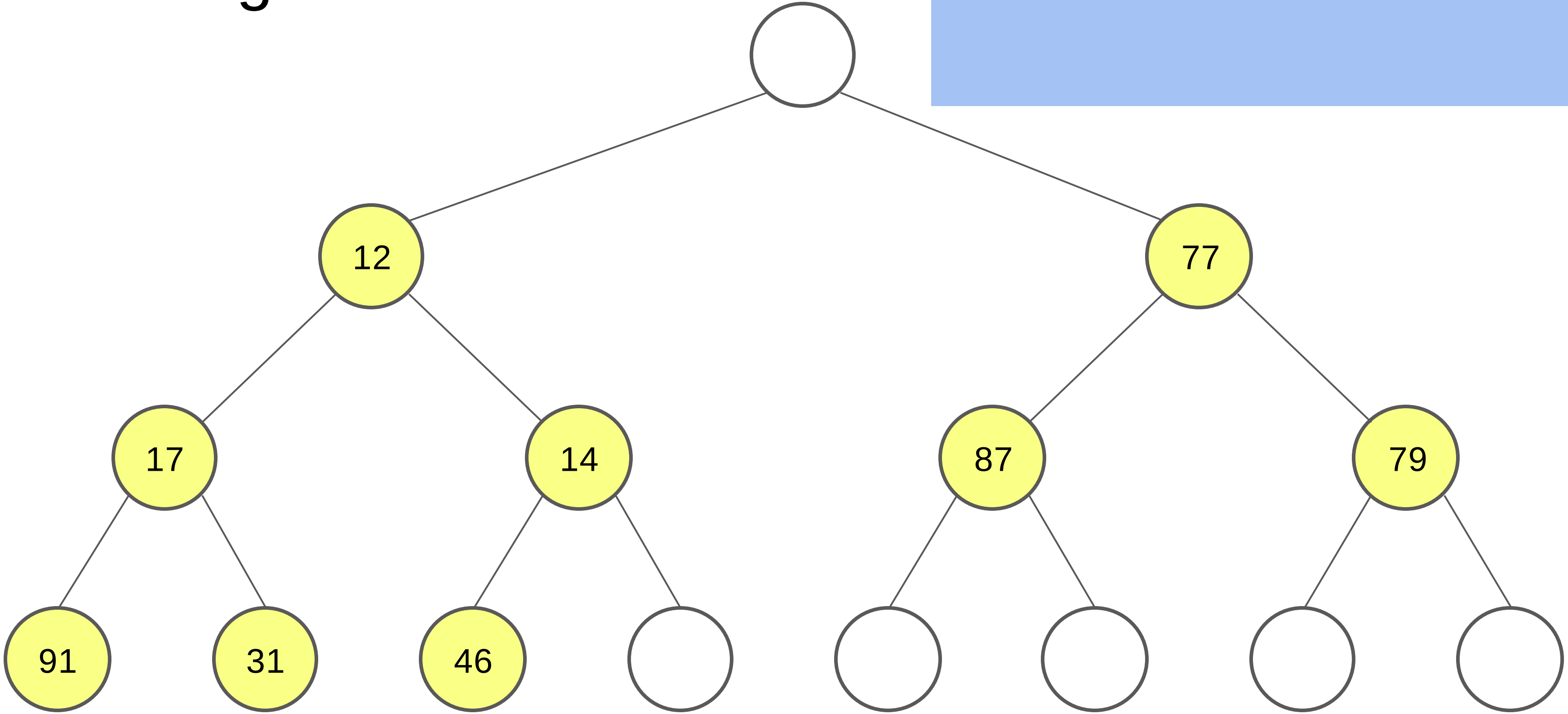
We will remove and return this value

Extracting the min



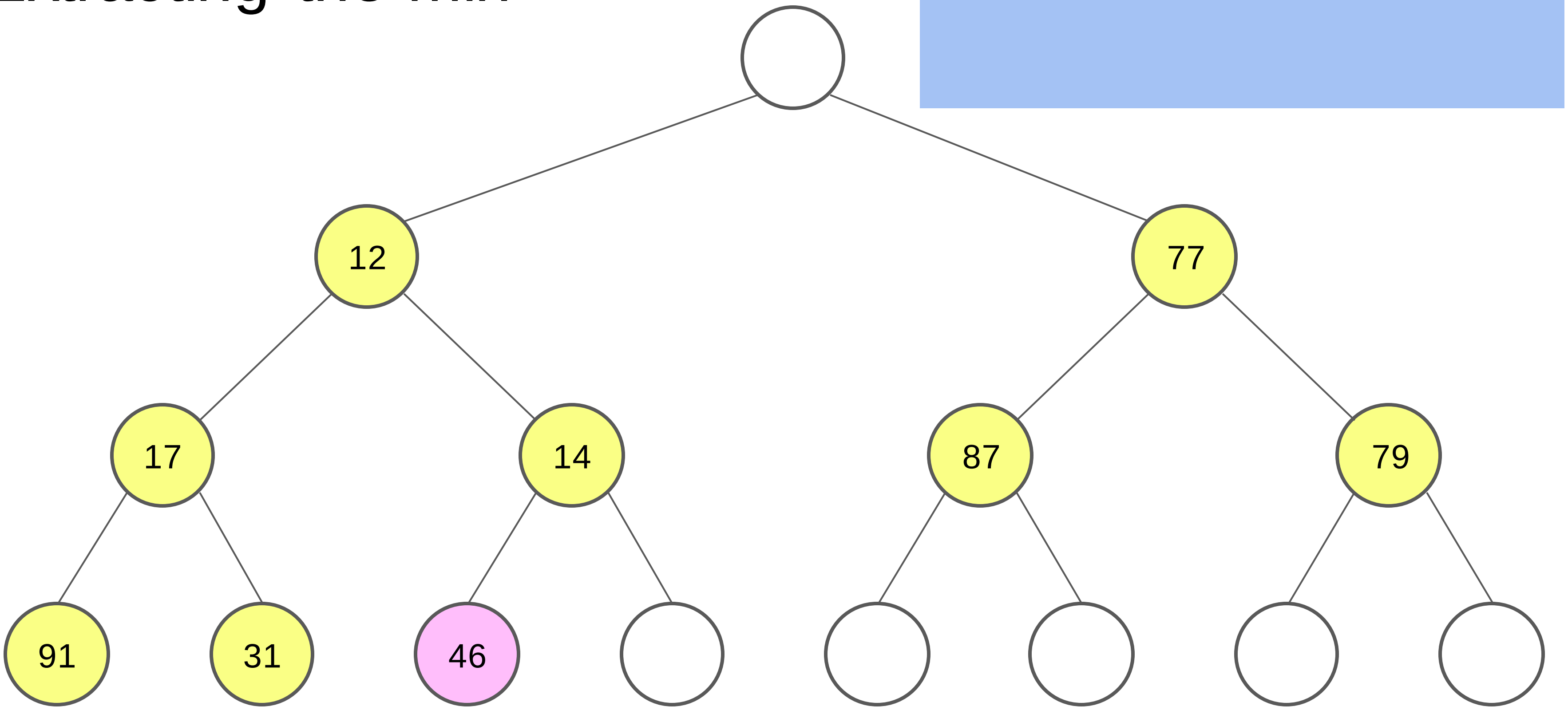
Extracting the min

- Move the last value to the root



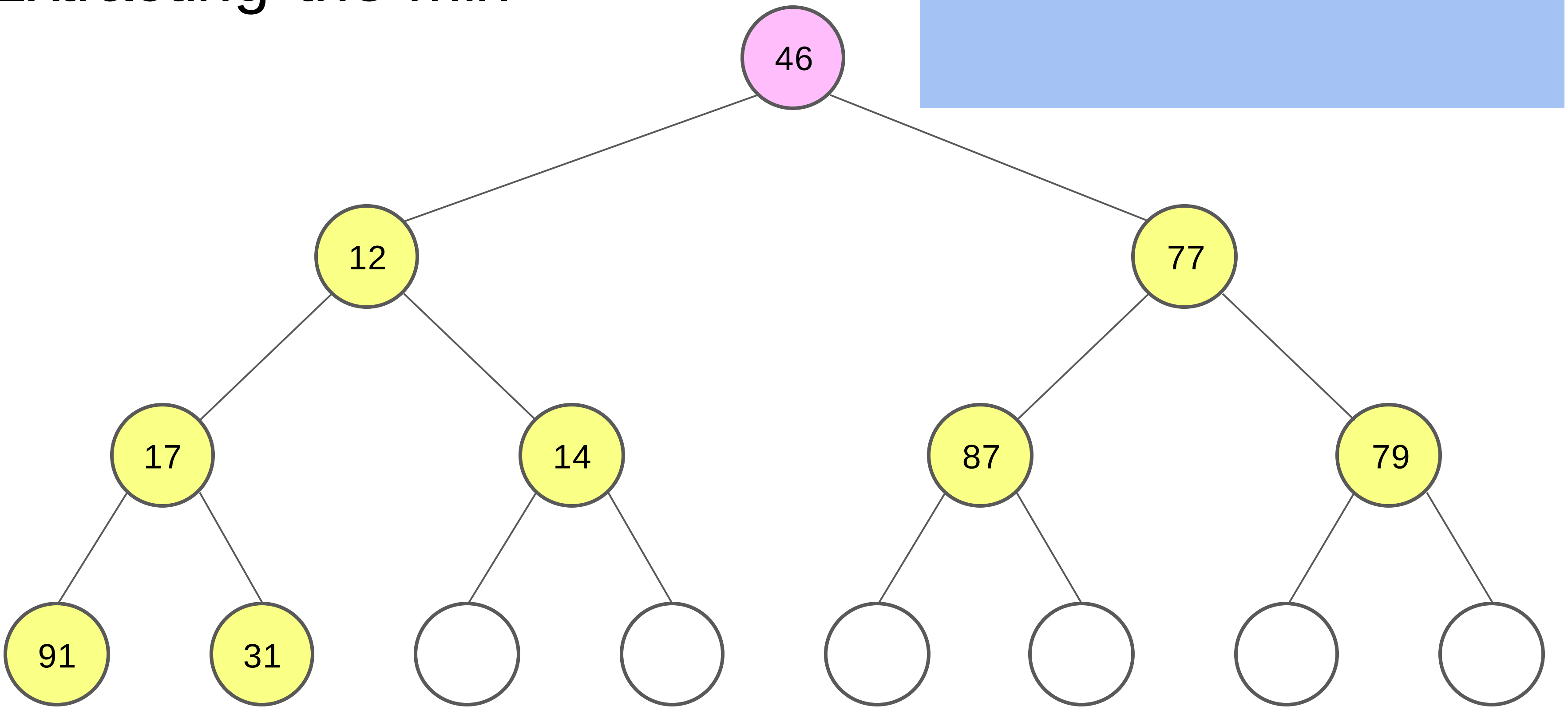
Extracting the min

- Move the last value to the root



Extracting the min

- Move the last value to the root

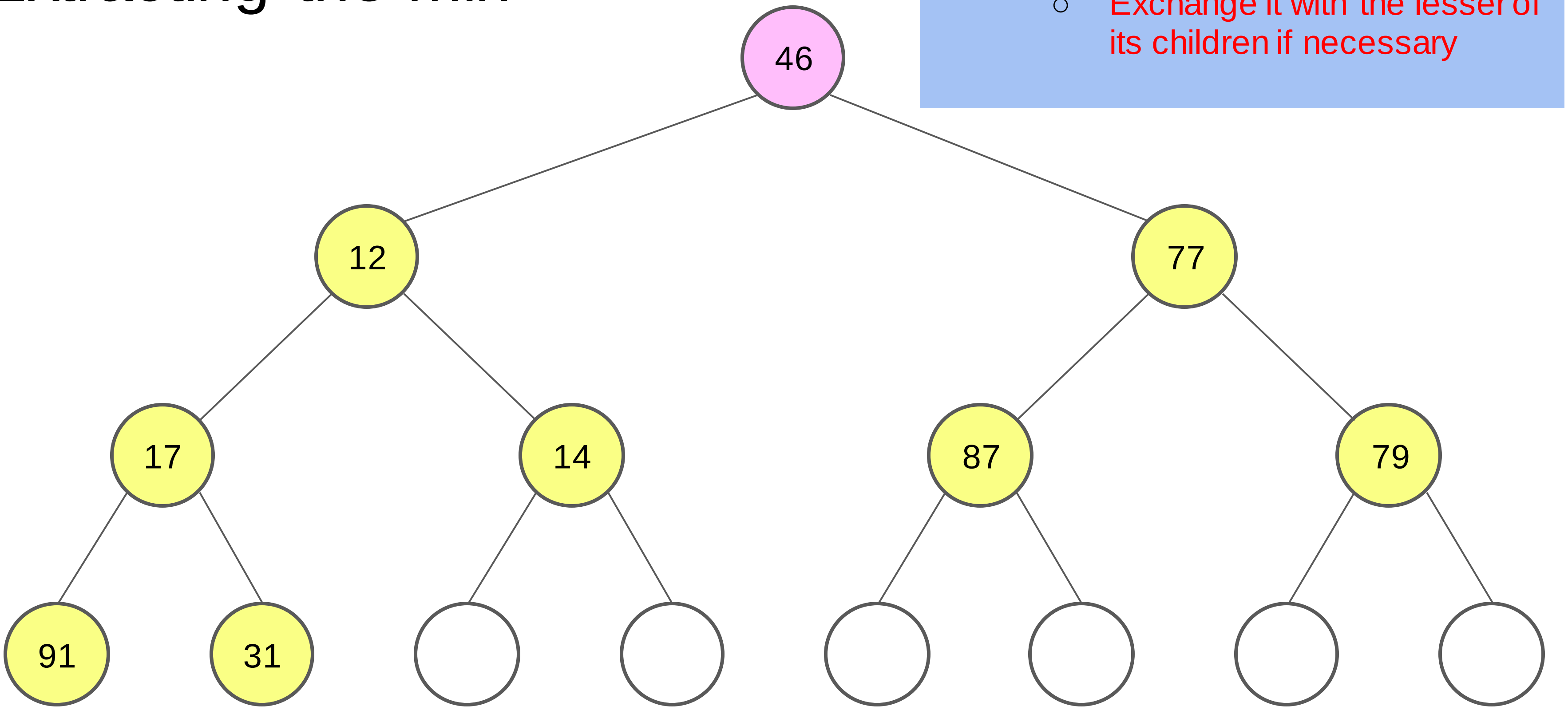


Wait, what?????

- The tree is compact again – hooray!
- But *heap property* at root may now be violated – boo!
- How can we we fix up the tree to be a heap again?
- Will use another swapping procedure: **heapify**

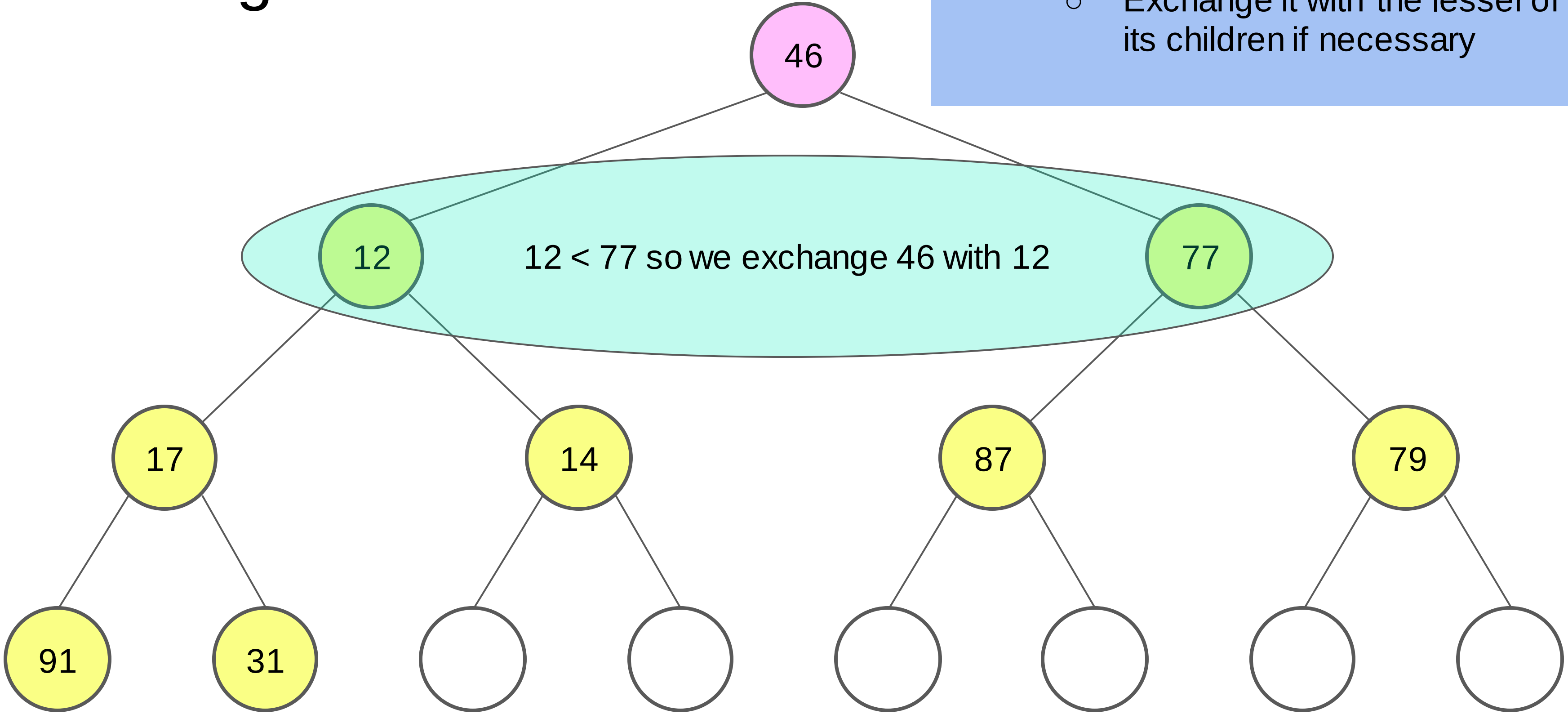
Extracting the min

- Move the last value to the root
- Heapify at that node
    - Exchange it with the lesser of its children if necessary



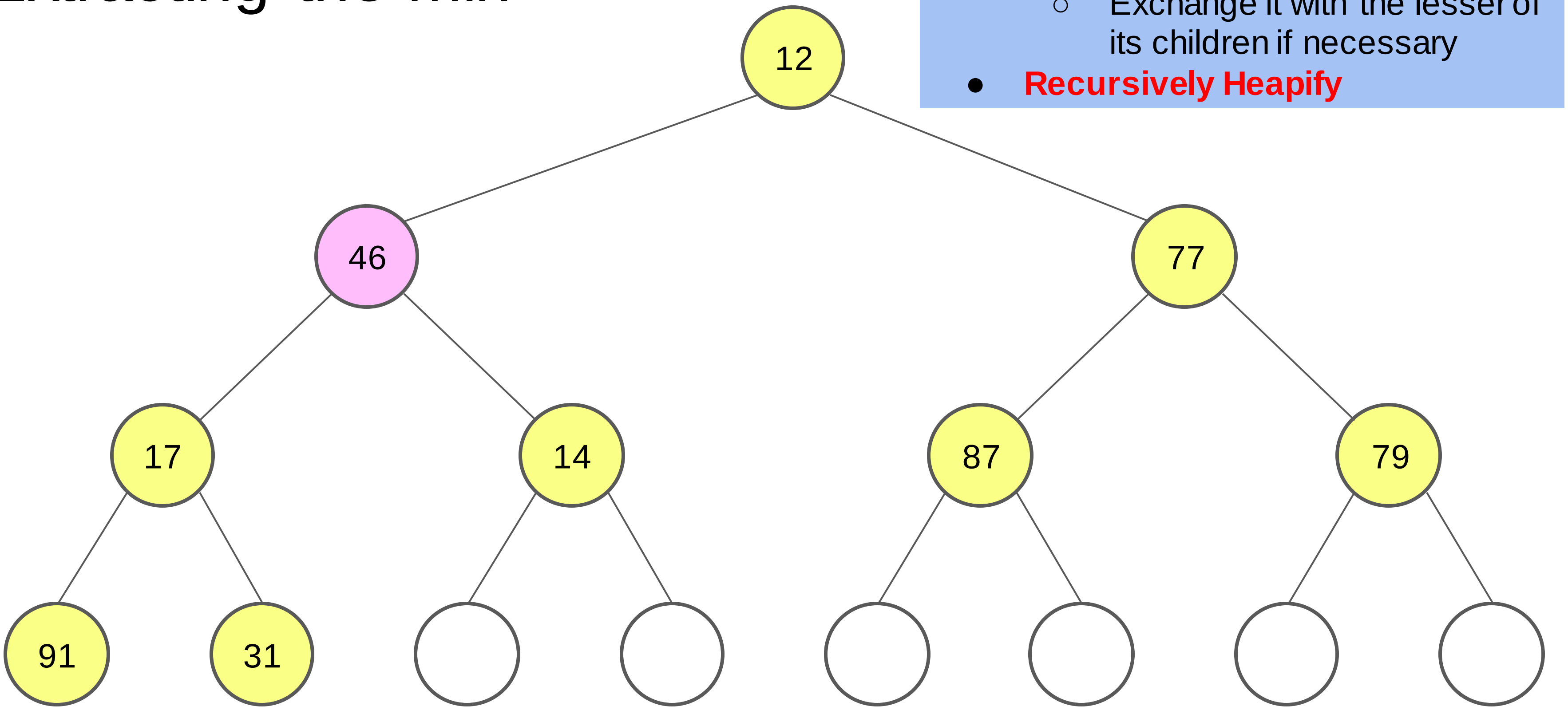
Extracting the min

- Move the last value to the root
- Heapify at that node
    - Exchange it with the lesser of its children if necessary



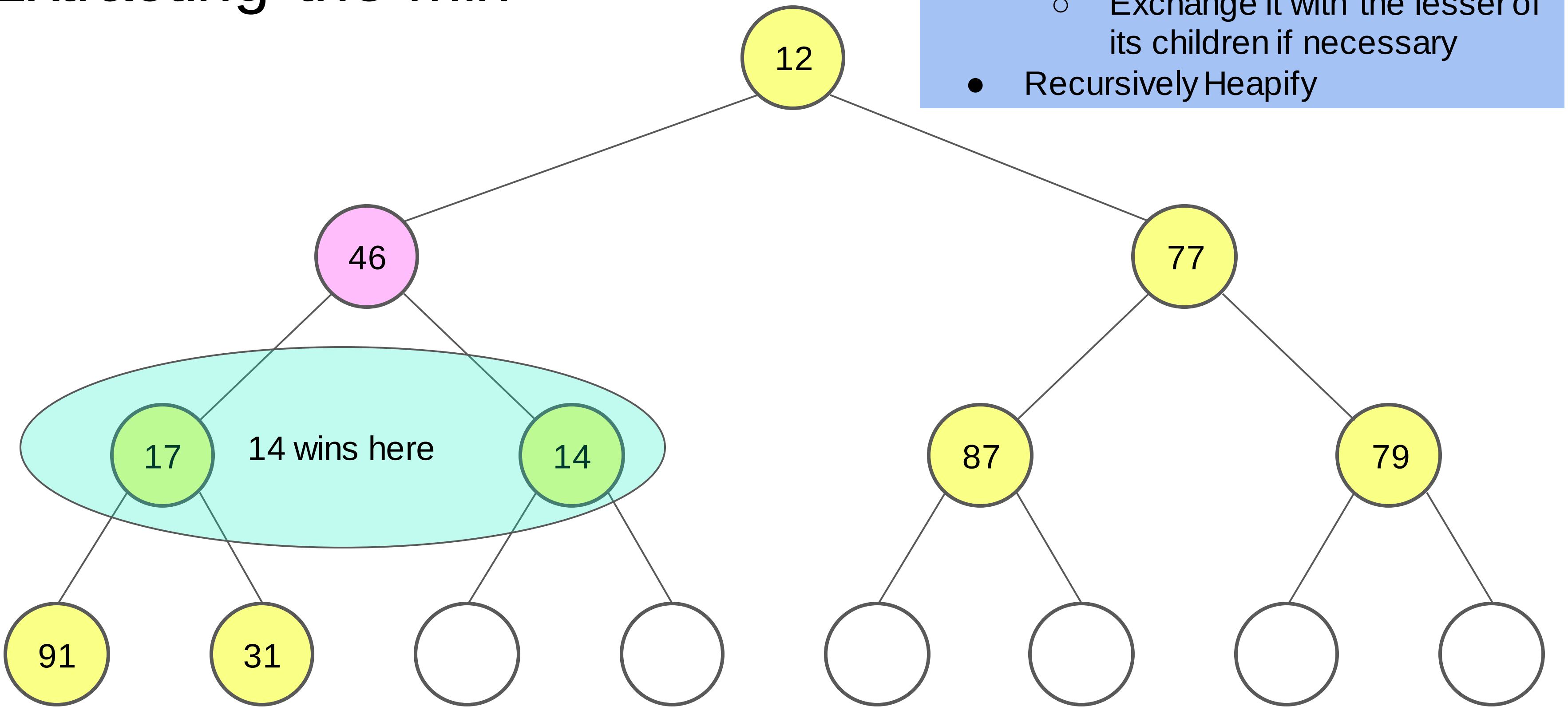
Extracting the min

- Move the last value to the root
- Heapify at that node
    - Exchange it with the lesser of its children if necessary
- **Recursively Heapify**



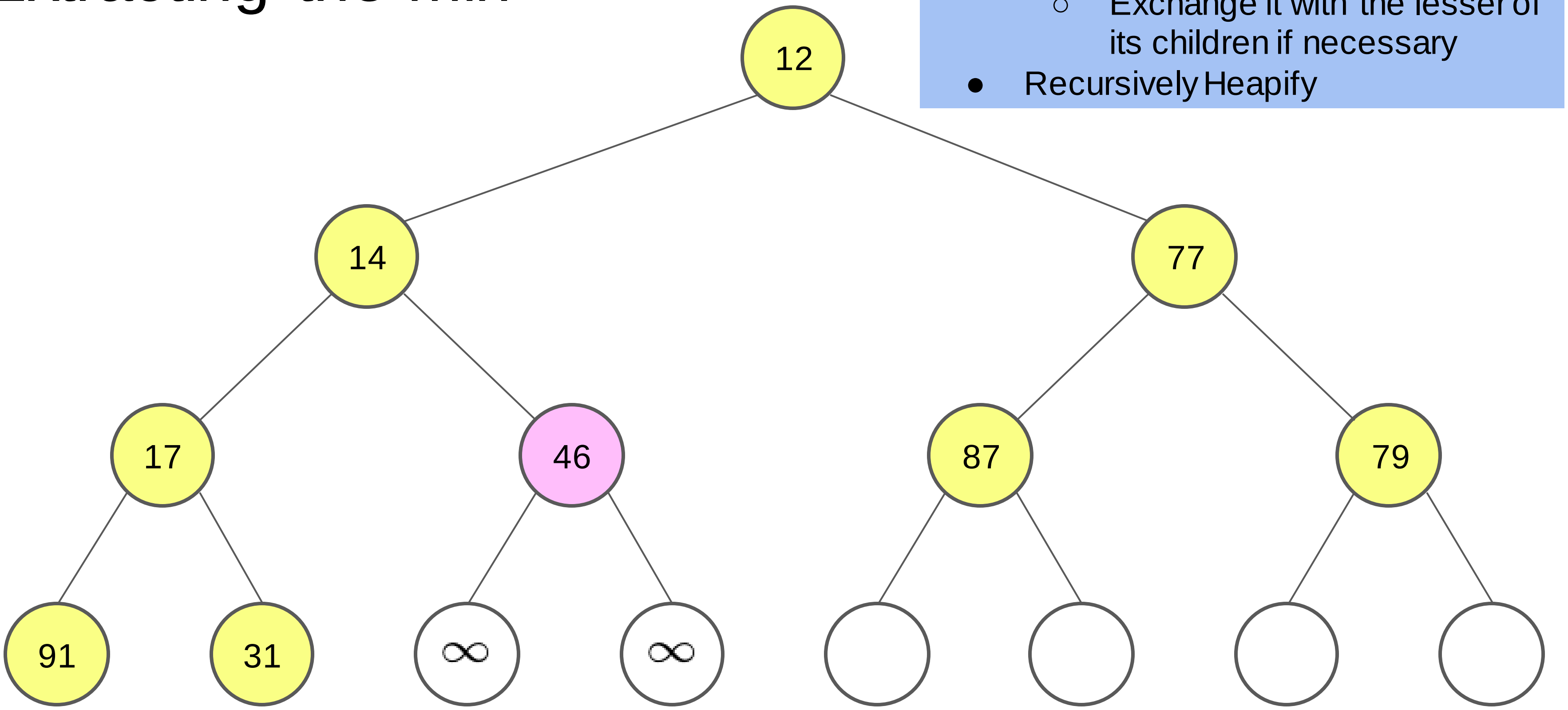
Extracting the min

- Move the last value to the root
- Heapify at that node
    - Exchange it with the lesser of its children if necessary
- Recursively Heapify



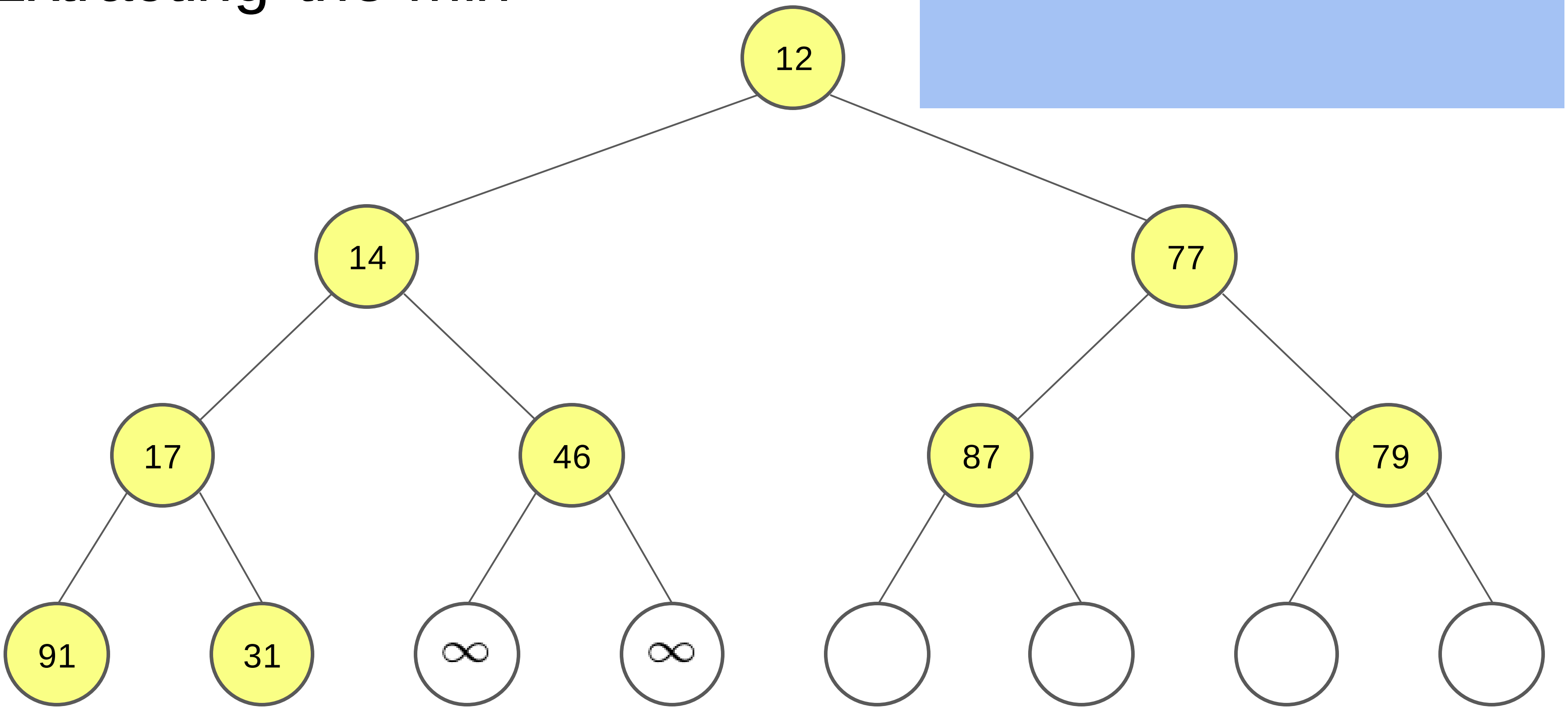
Extracting the min

- Move the last value to the root
- Heapify at that node
  - Exchange it with the lesser of its children if necessary
- Recursively Heapify



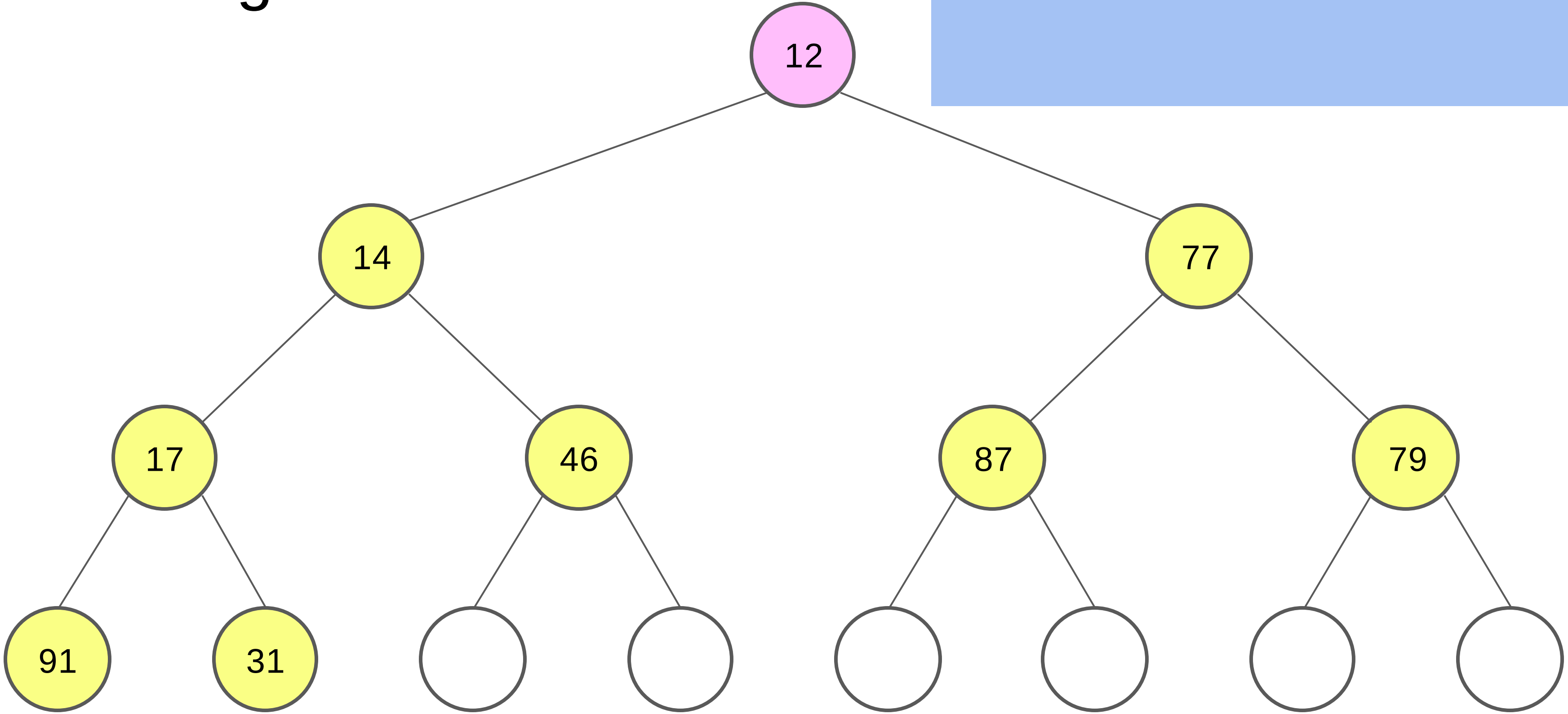
Extracting the min

- Done!
- But let's do that again



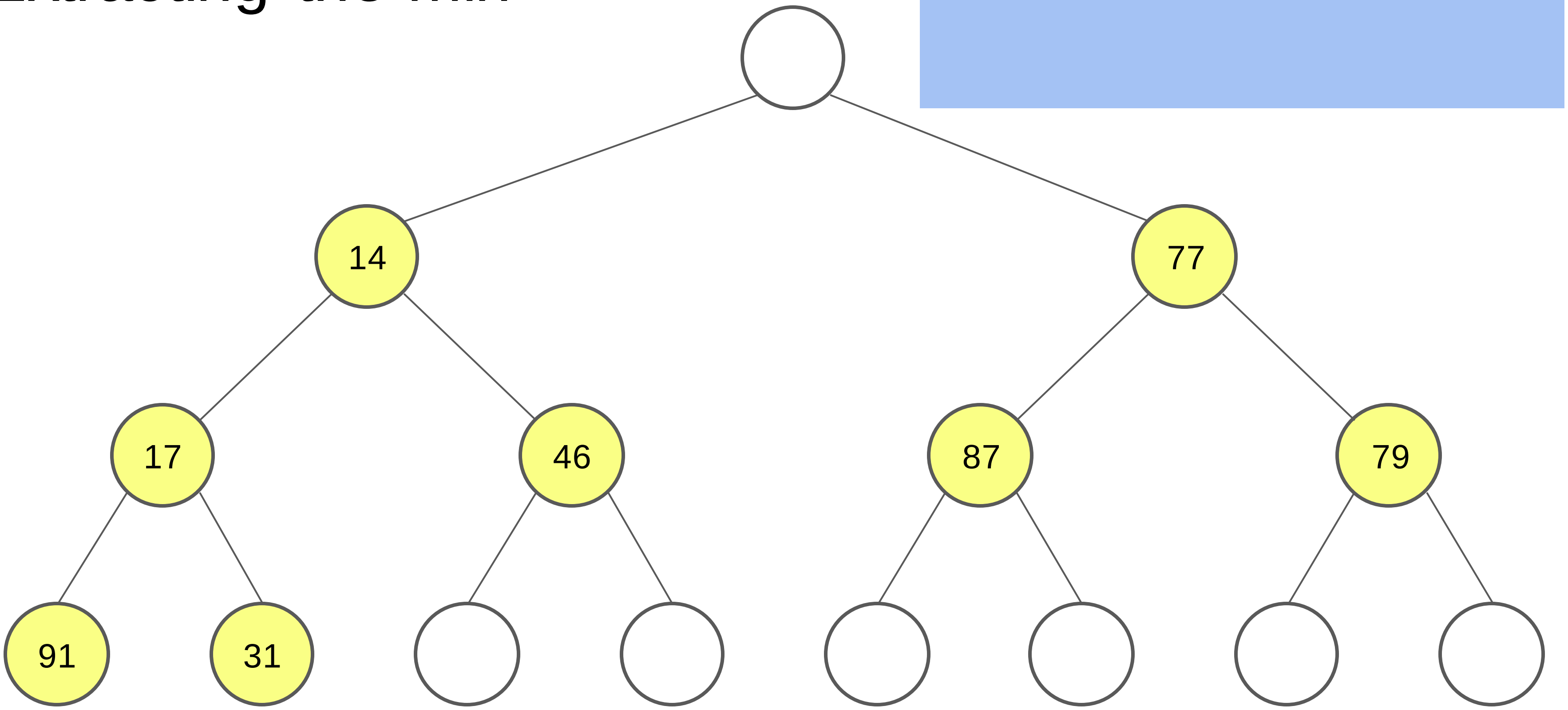
Extracting the min

- We will return 12



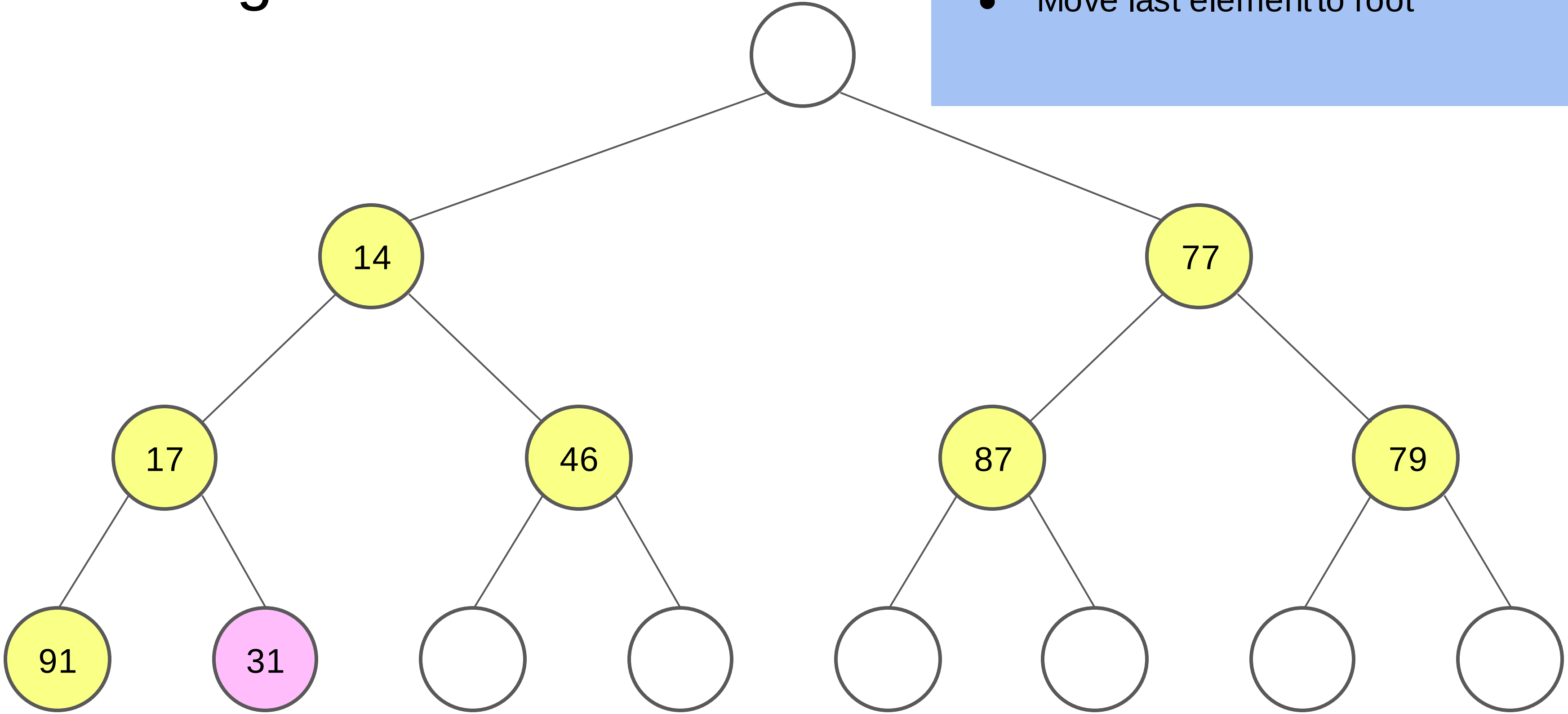
Extracting the min

- We will return 12
- Creates a hole at root



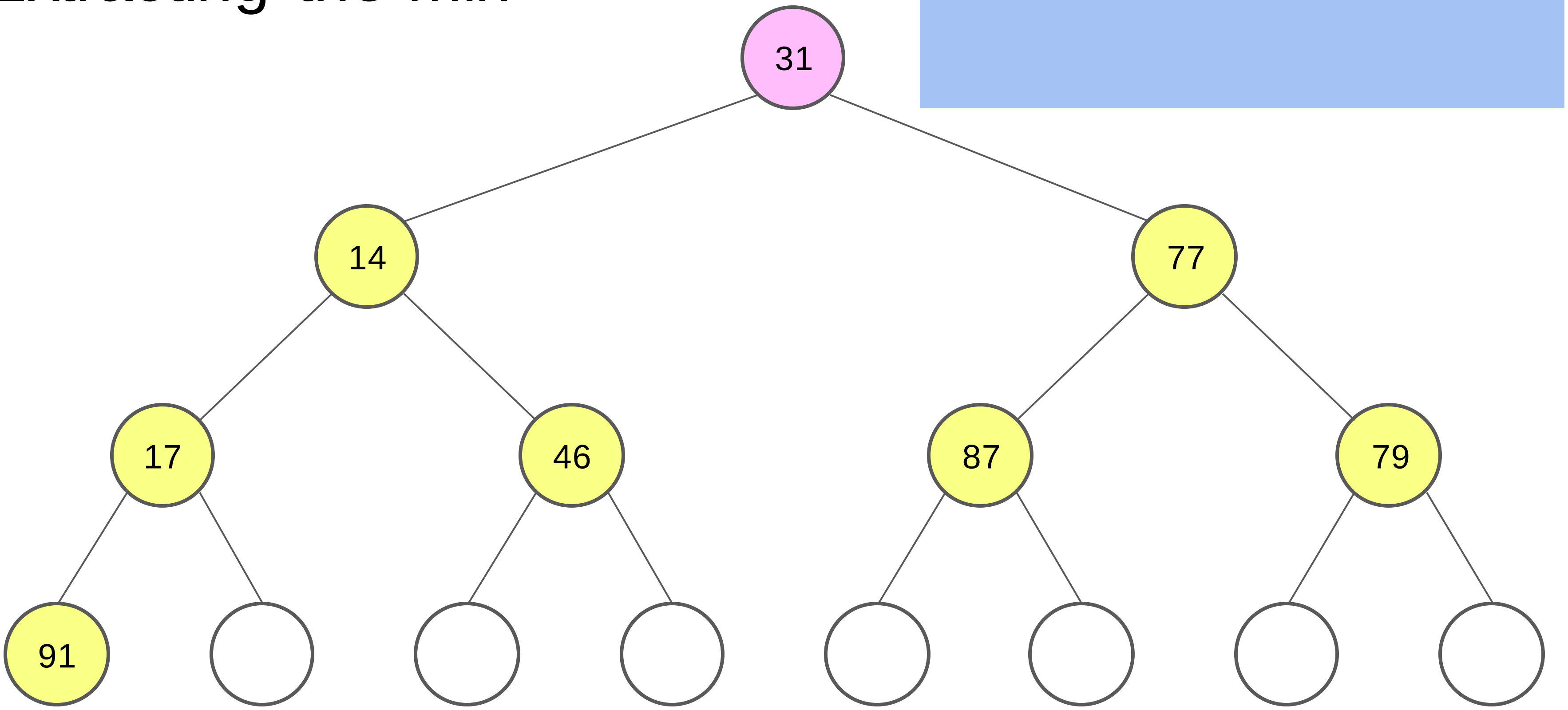
Extracting the min

- We will return 12
- Creates a hole at root
- Move last element to root



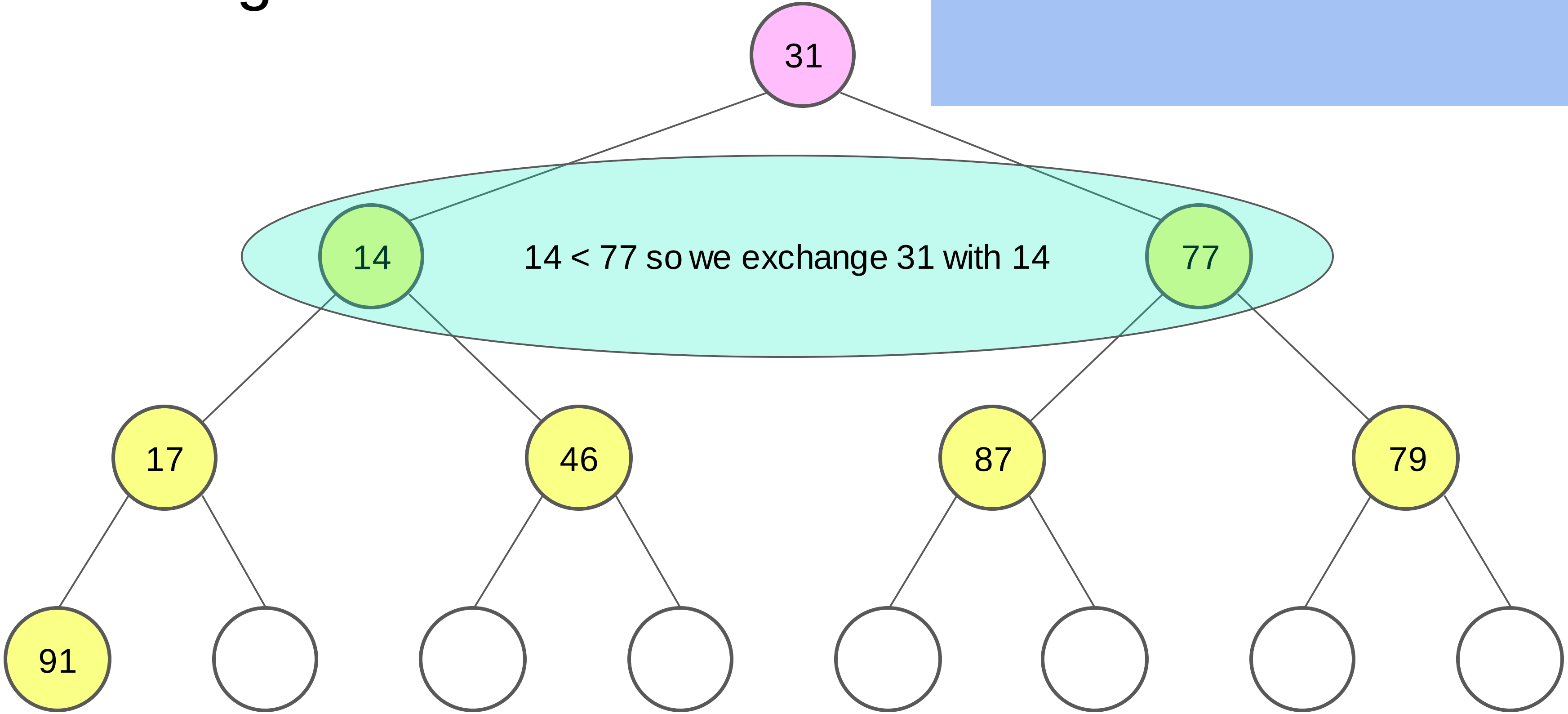
Extracting the min

- Heapify



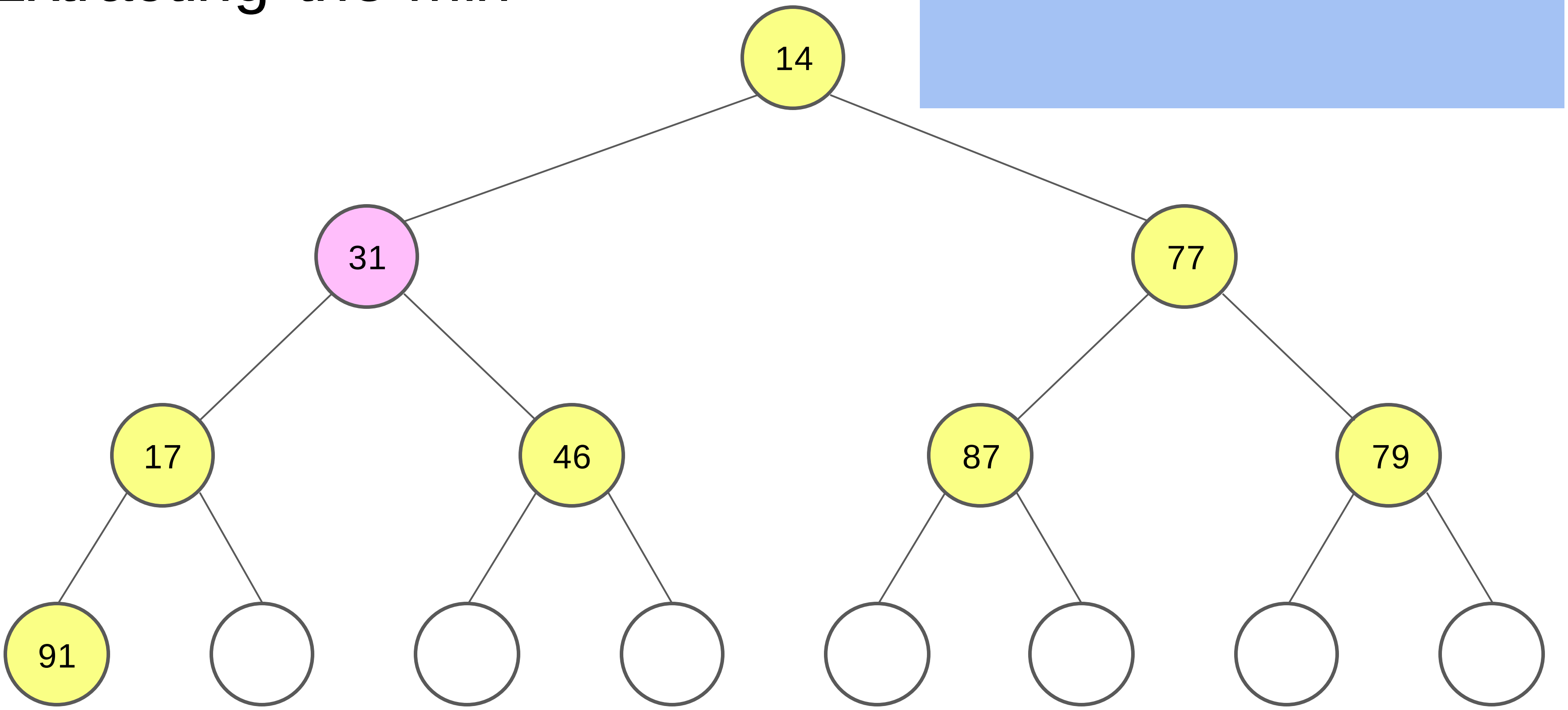
Extracting the min

- Heapify



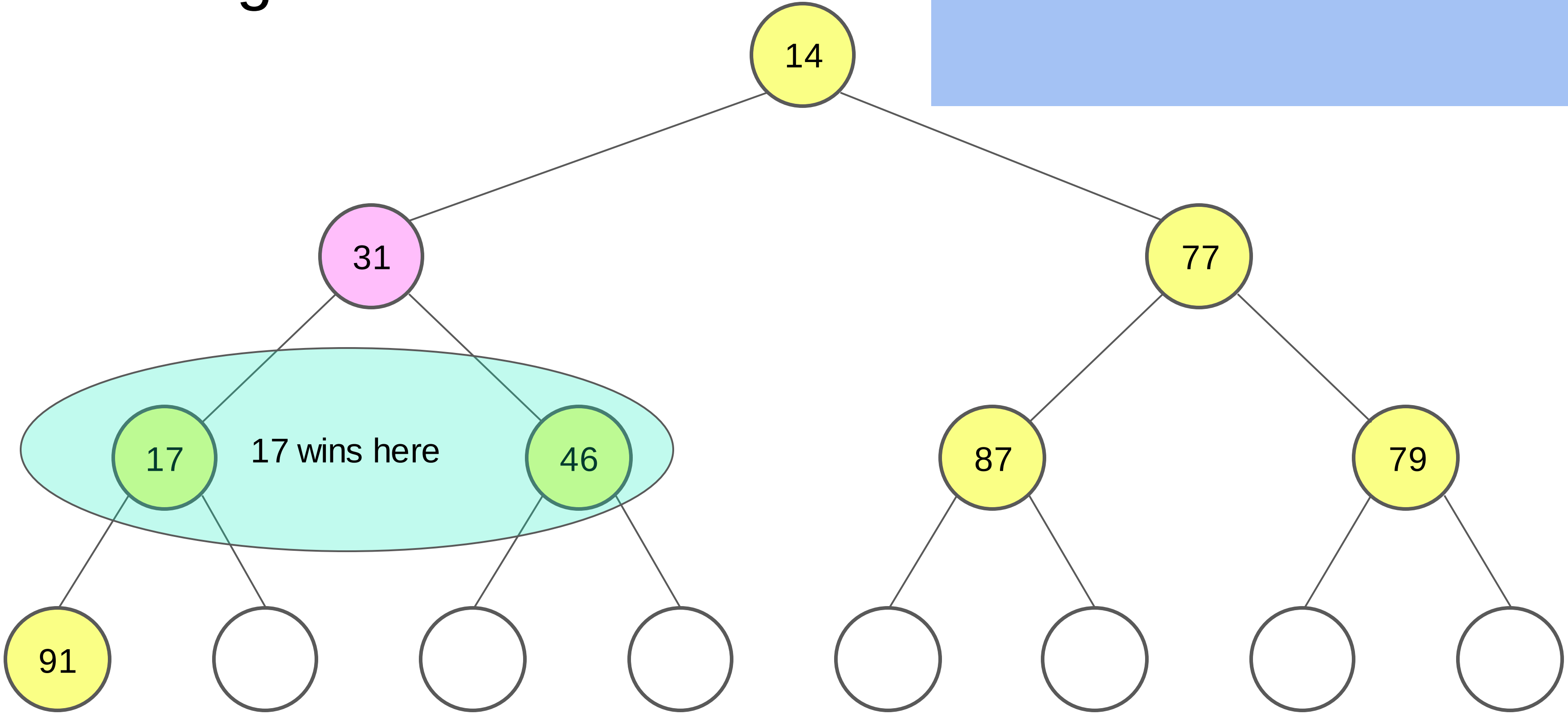
Extracting the min

- Heapify



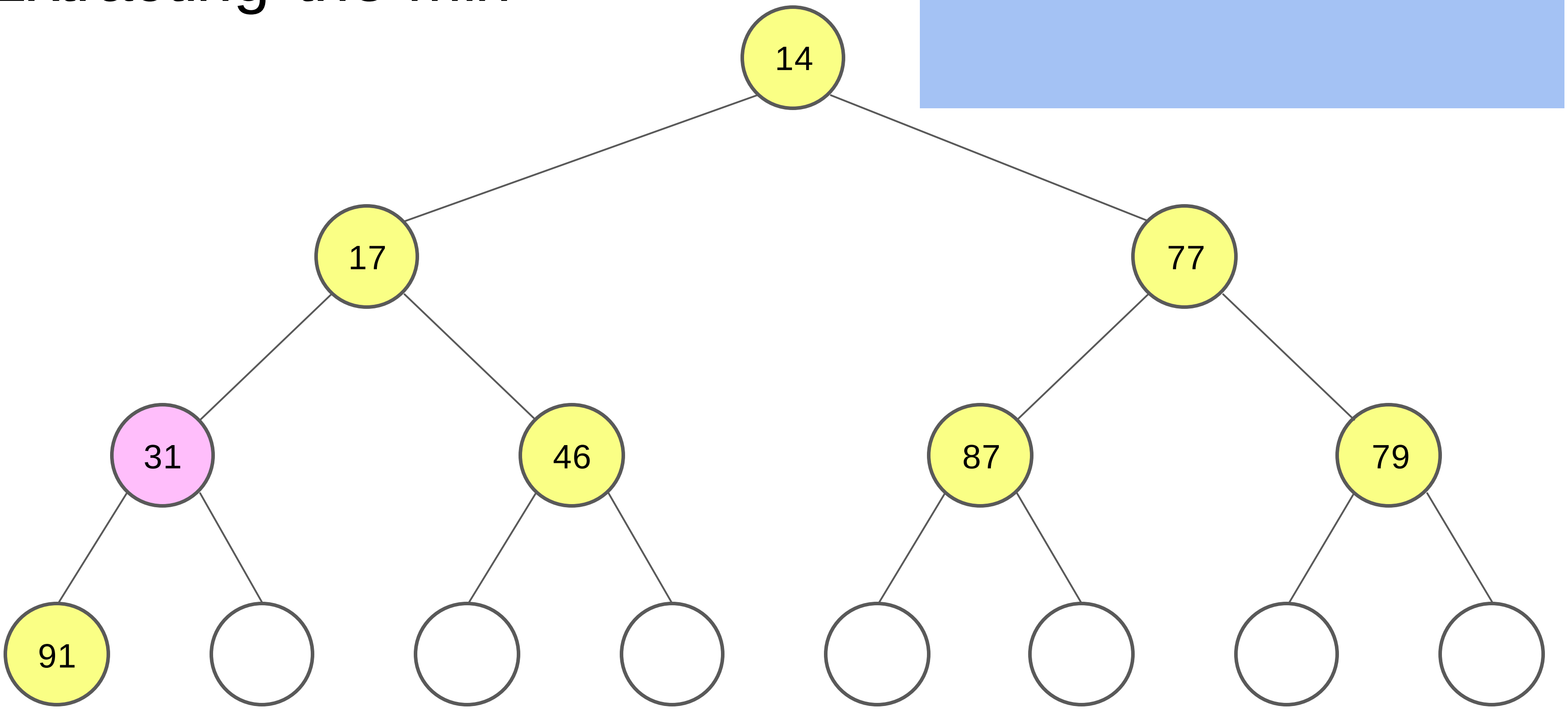
Extracting the min

- Heapify



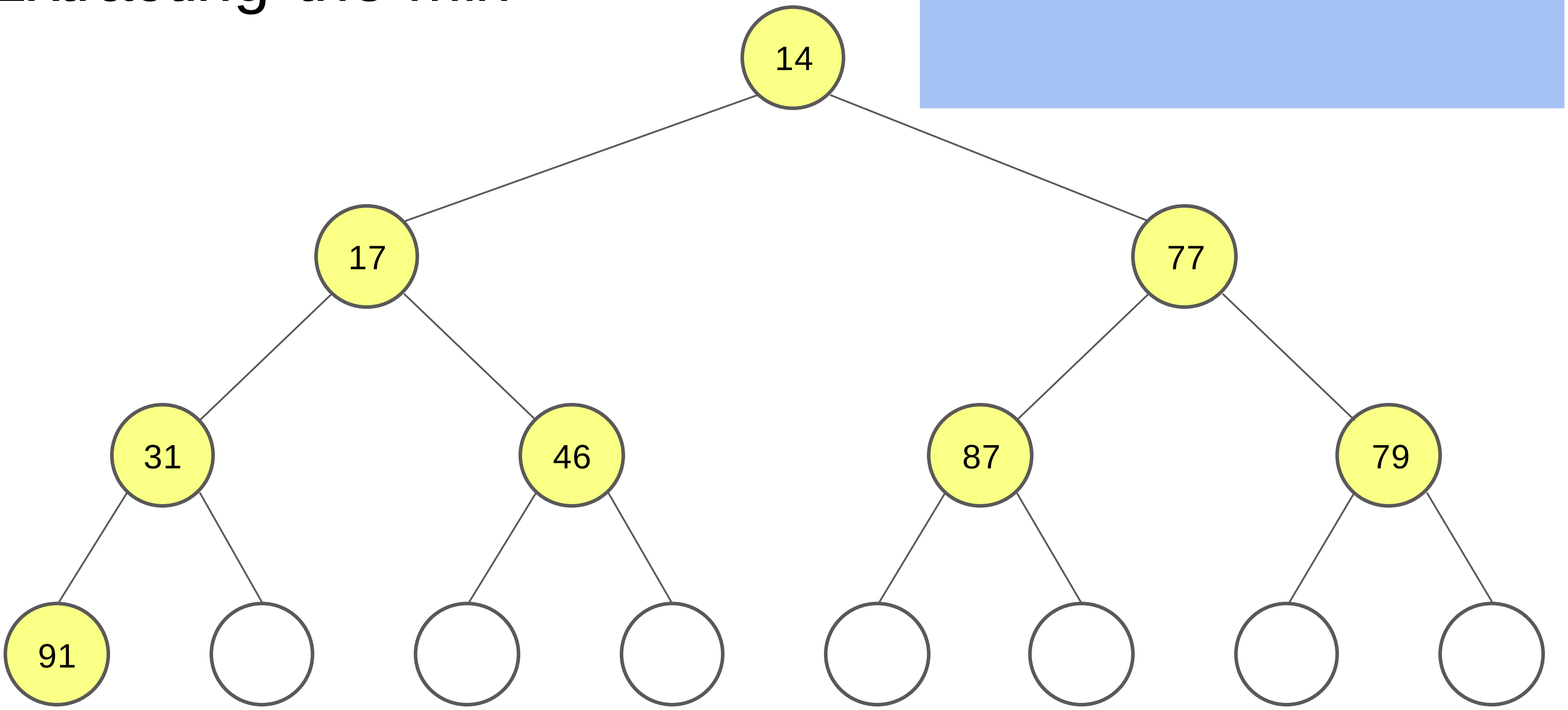
Extracting the min

- Heapify



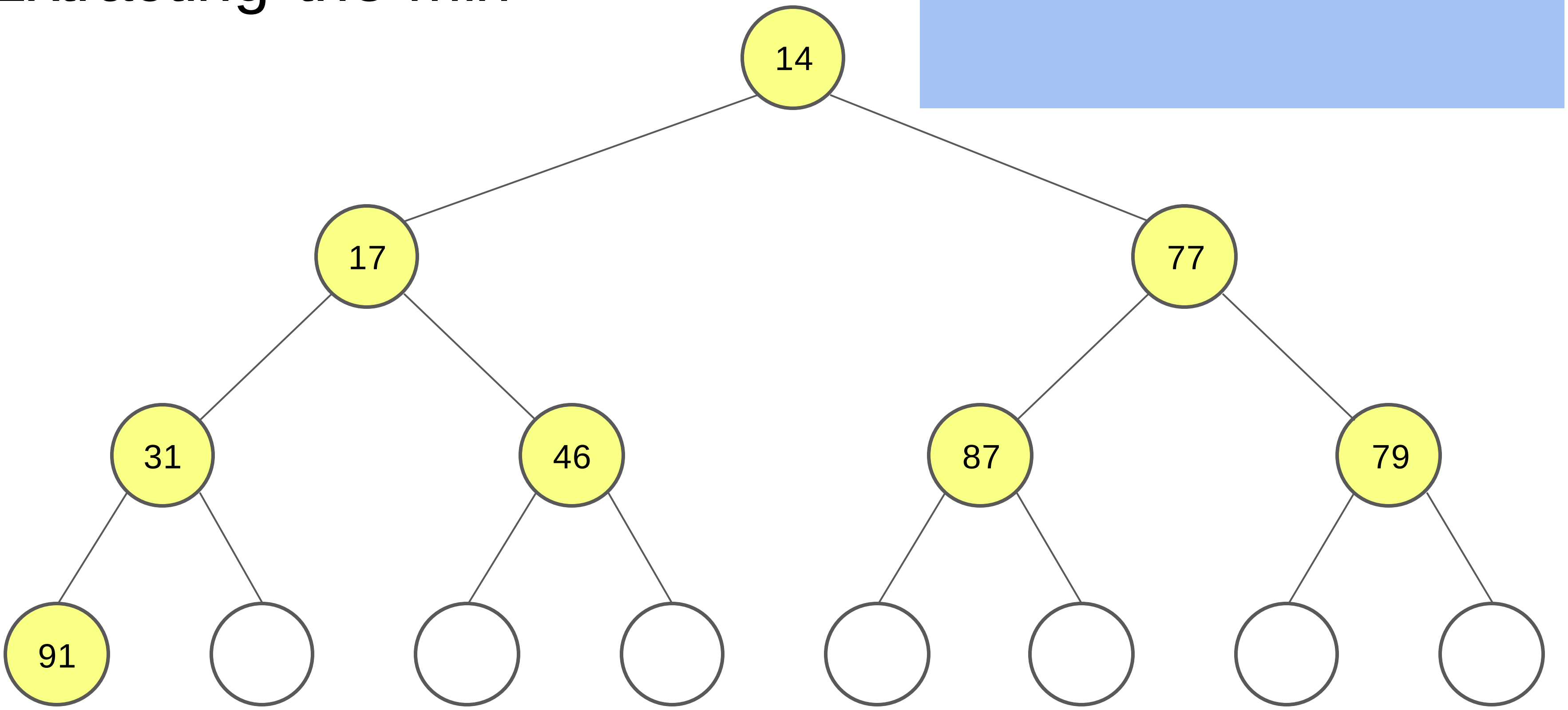
Extracting the min

- Heapify
- Done



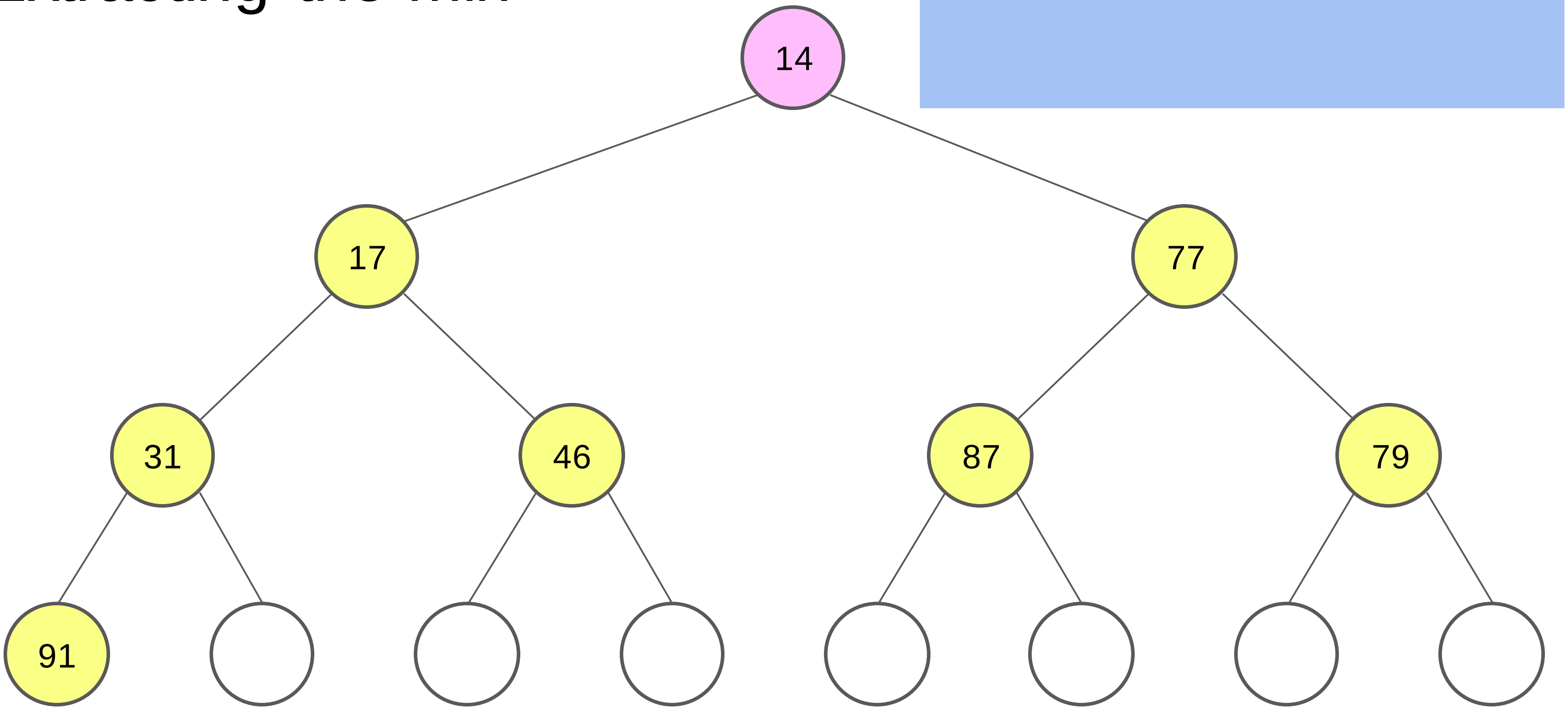
Extracting the min

- Again



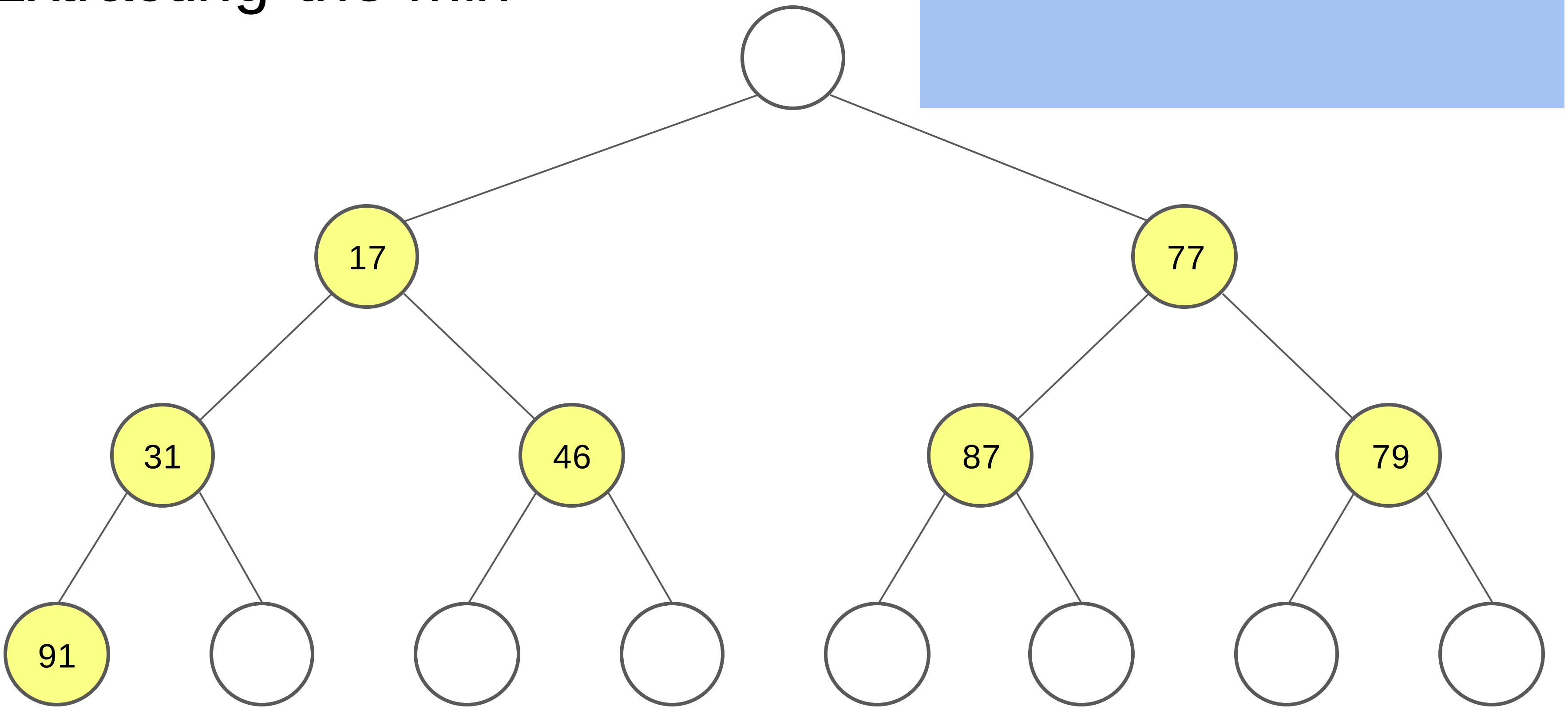
Extracting the min

- Again
- 14 will be returned



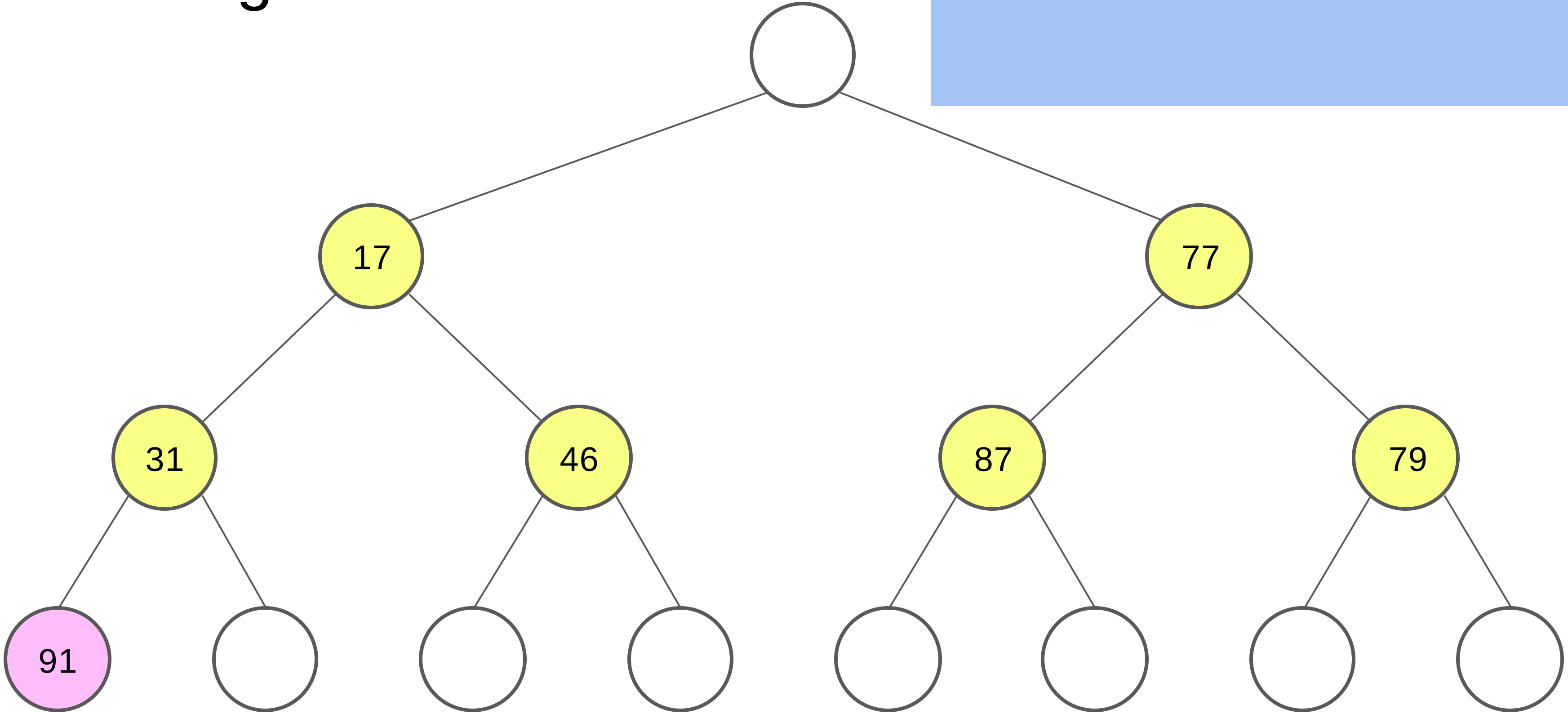
Extracting the min

- Again
- 14 will be returned



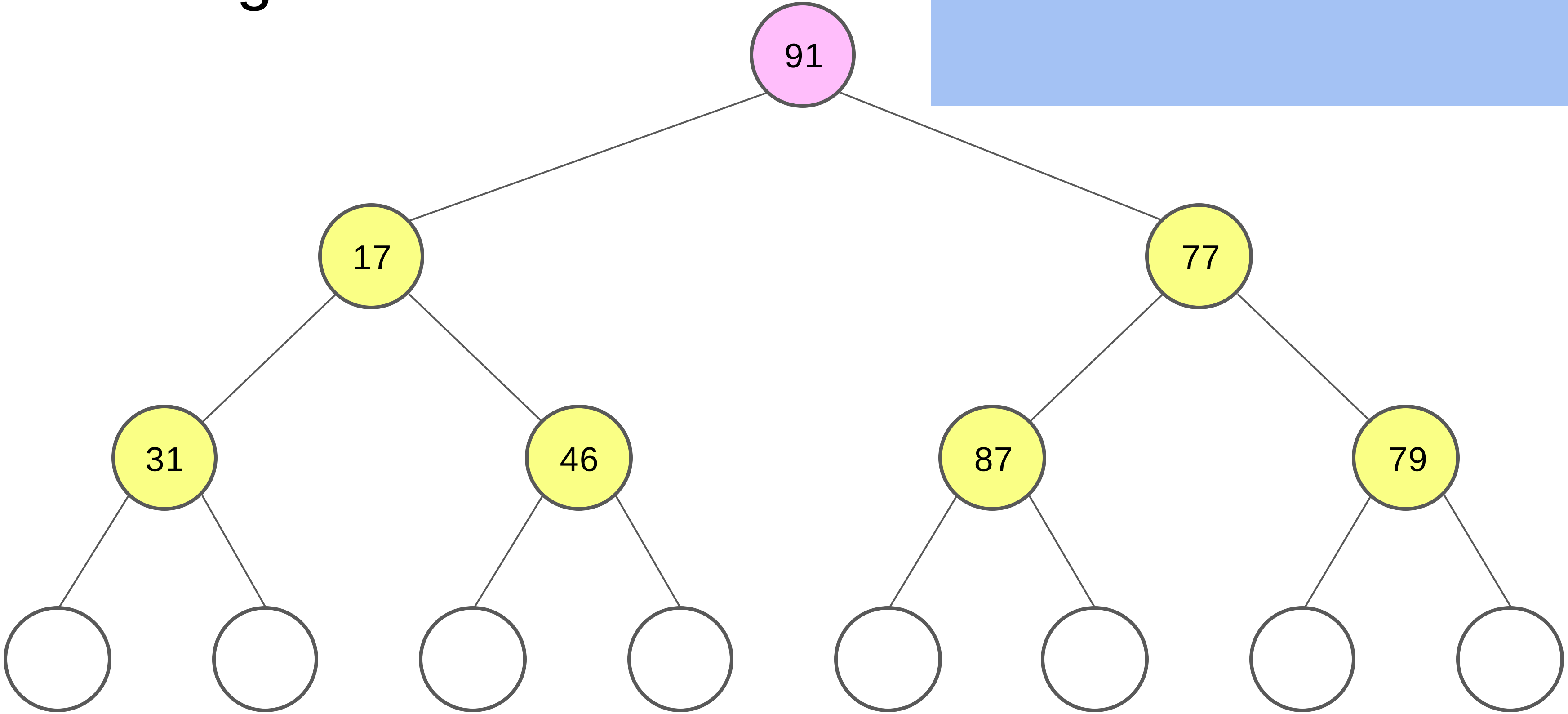
Extracting the min

- Again
- 14 will be returned



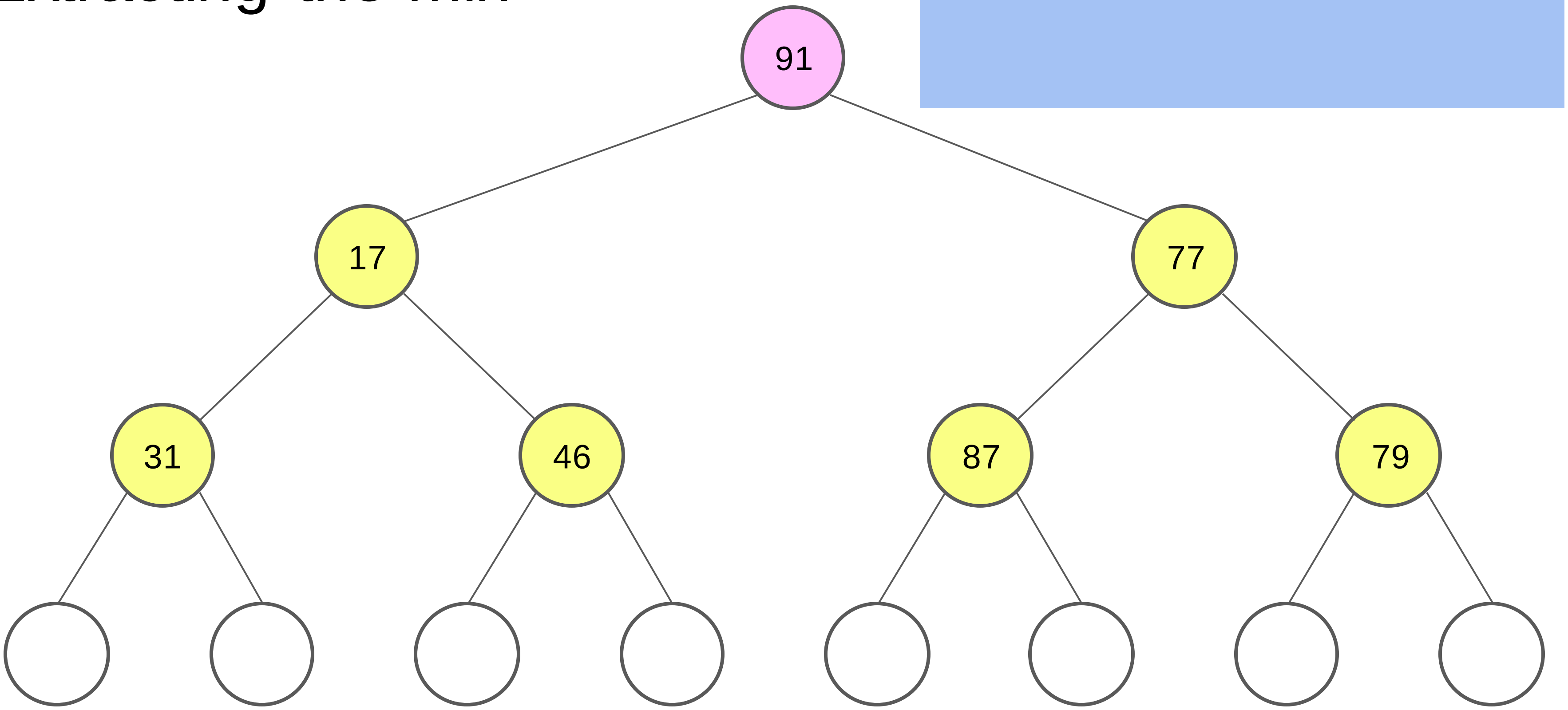
Extracting the min

- Move last element to root



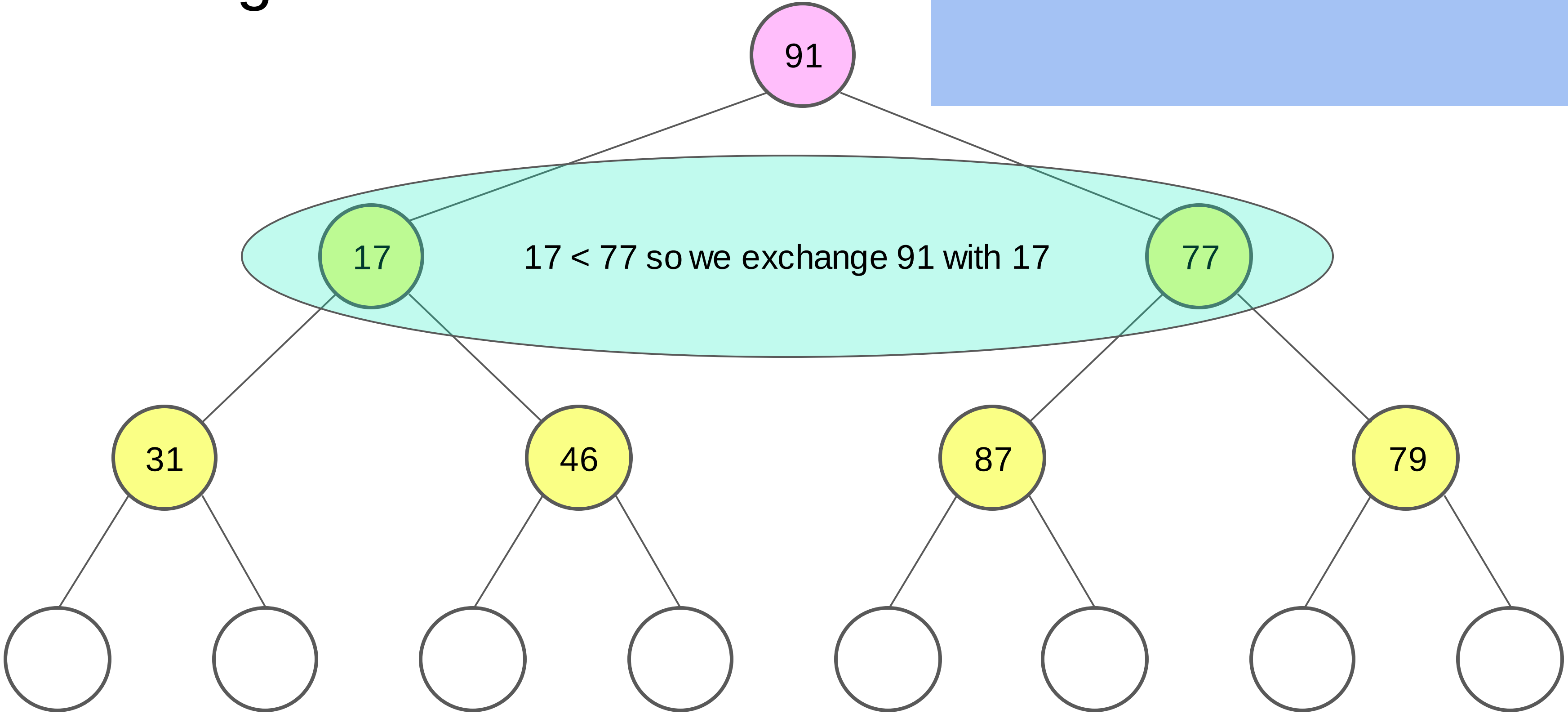
Extracting the min

- Heapify



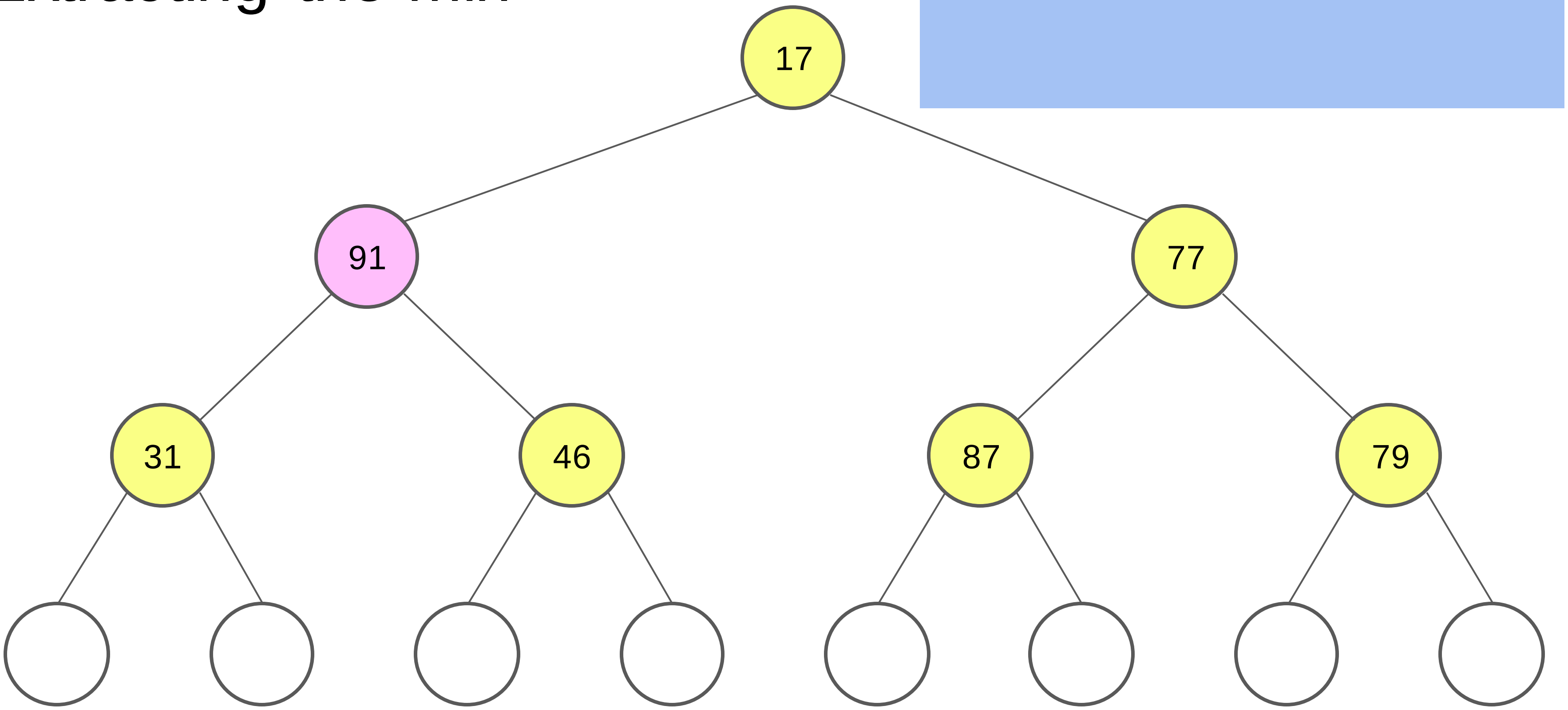
Extracting the min

- Heapify



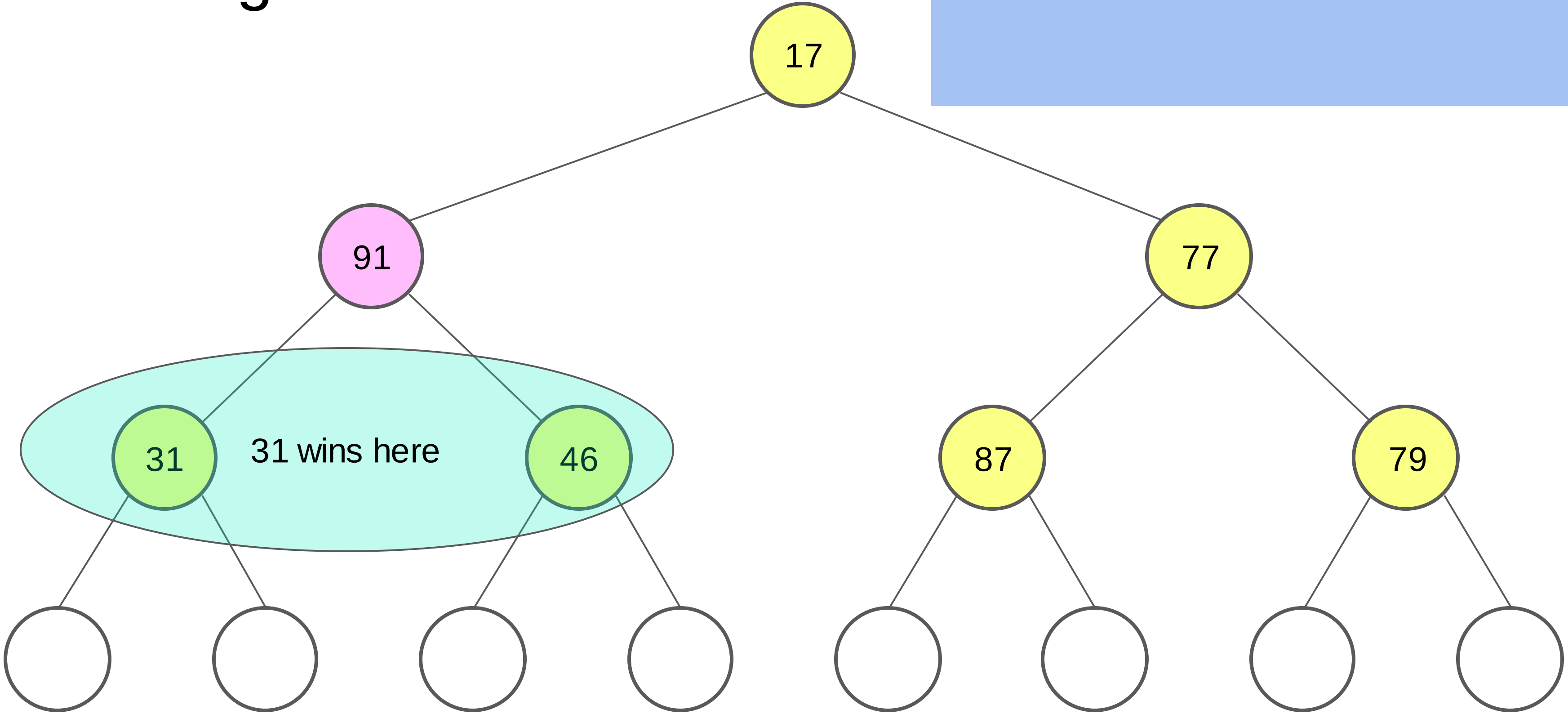
Extracting the min

- Heapify



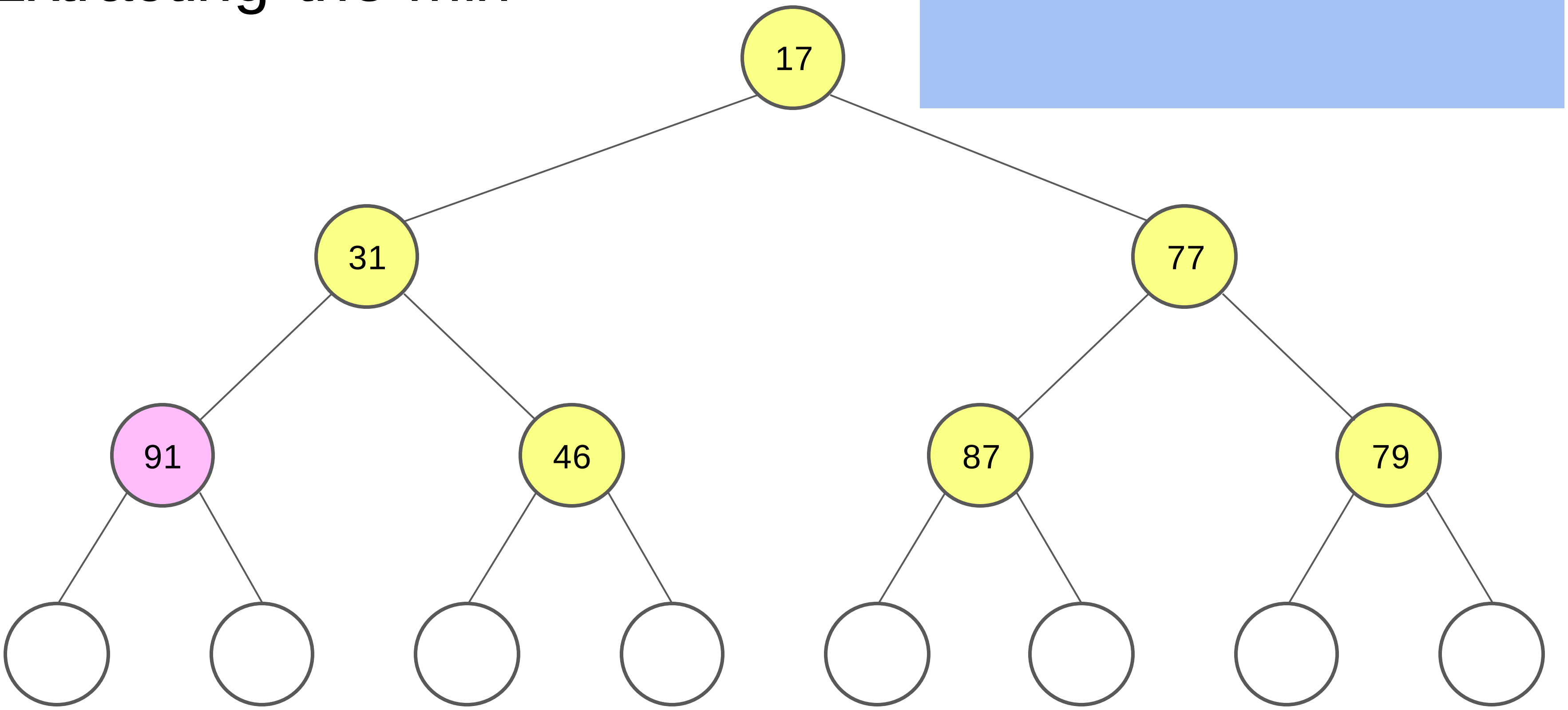
Extracting the min

- Heapify



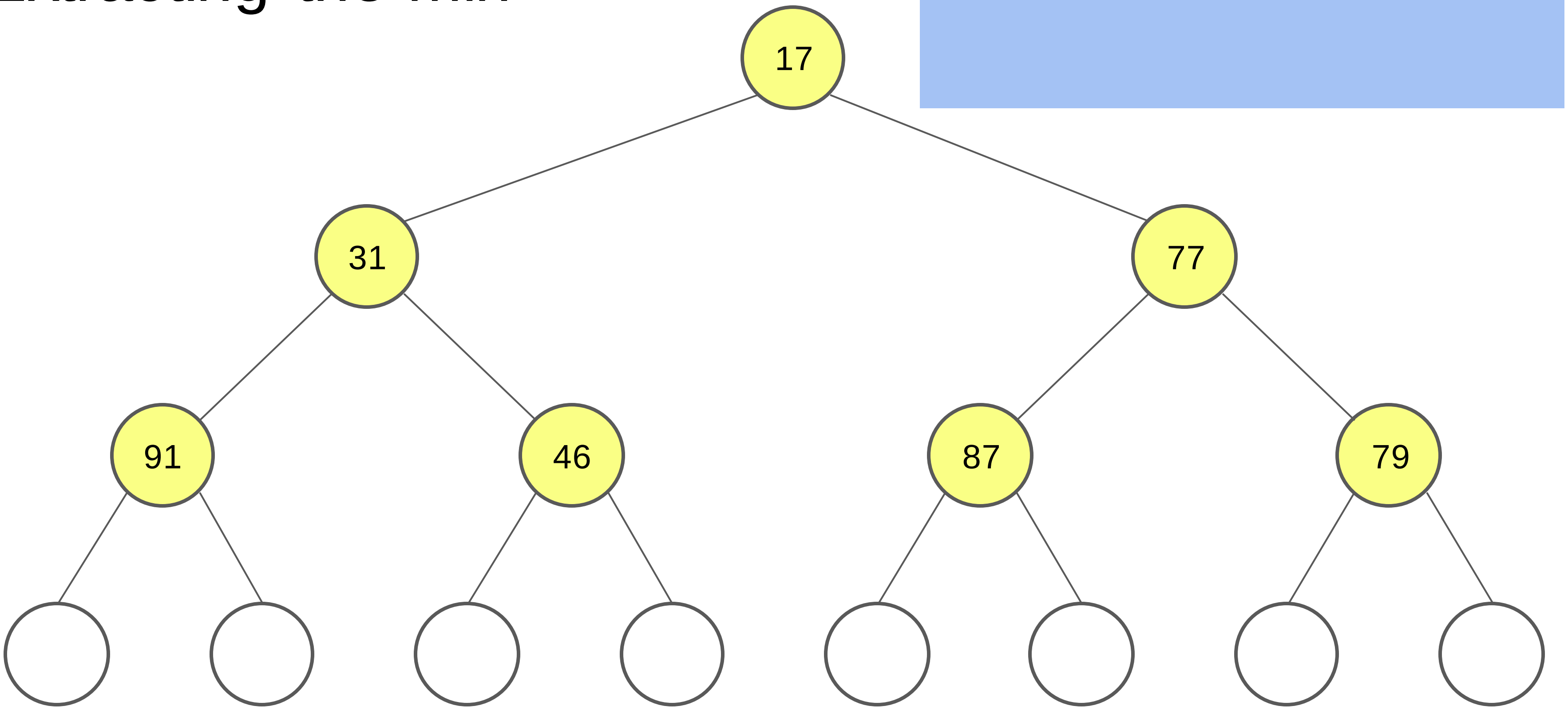
Extracting the min

- Heapify



Extracting the min

- Again, 17 will be returned

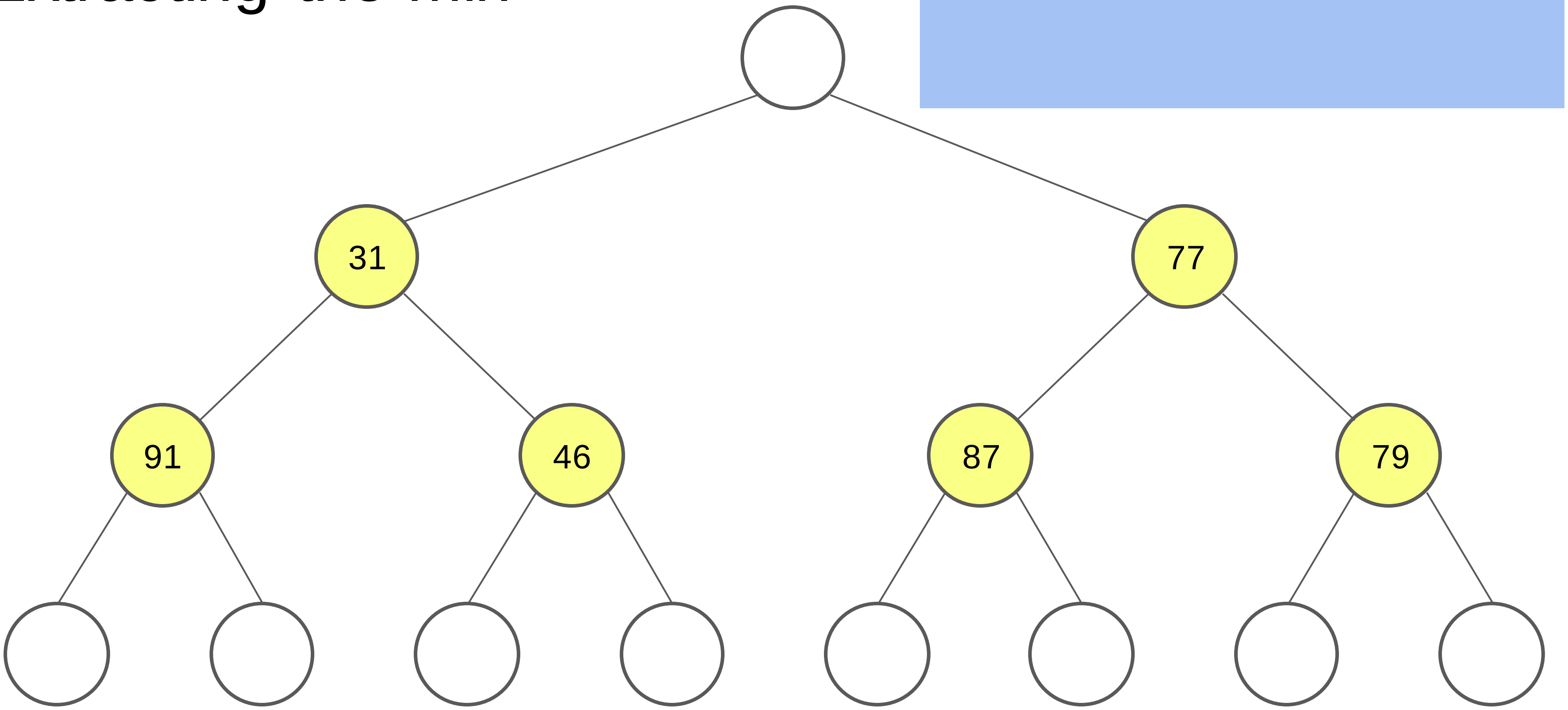


Pause… you try it

Take a minute to work through the next couple of extractions yourself…

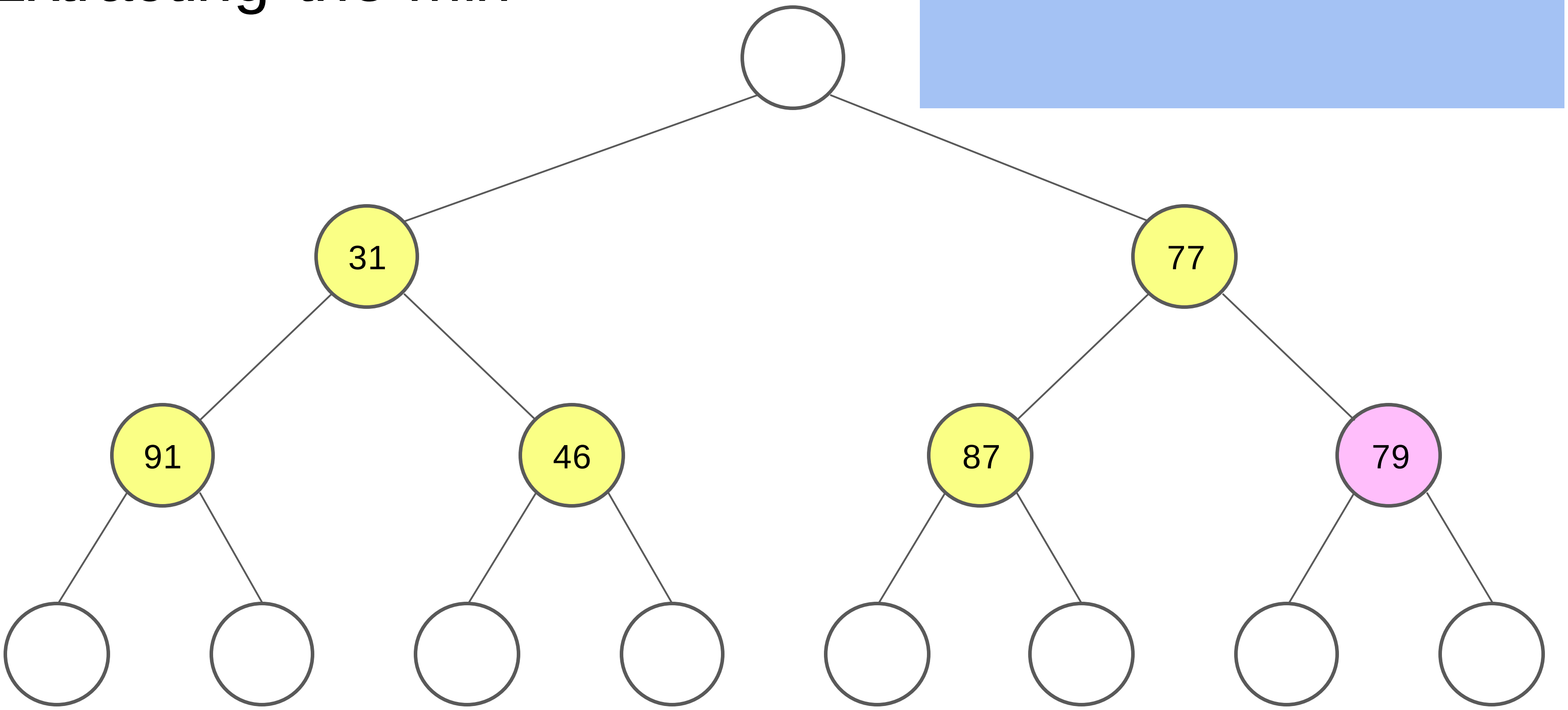
Extracting the min

- Again, 17 will be returned



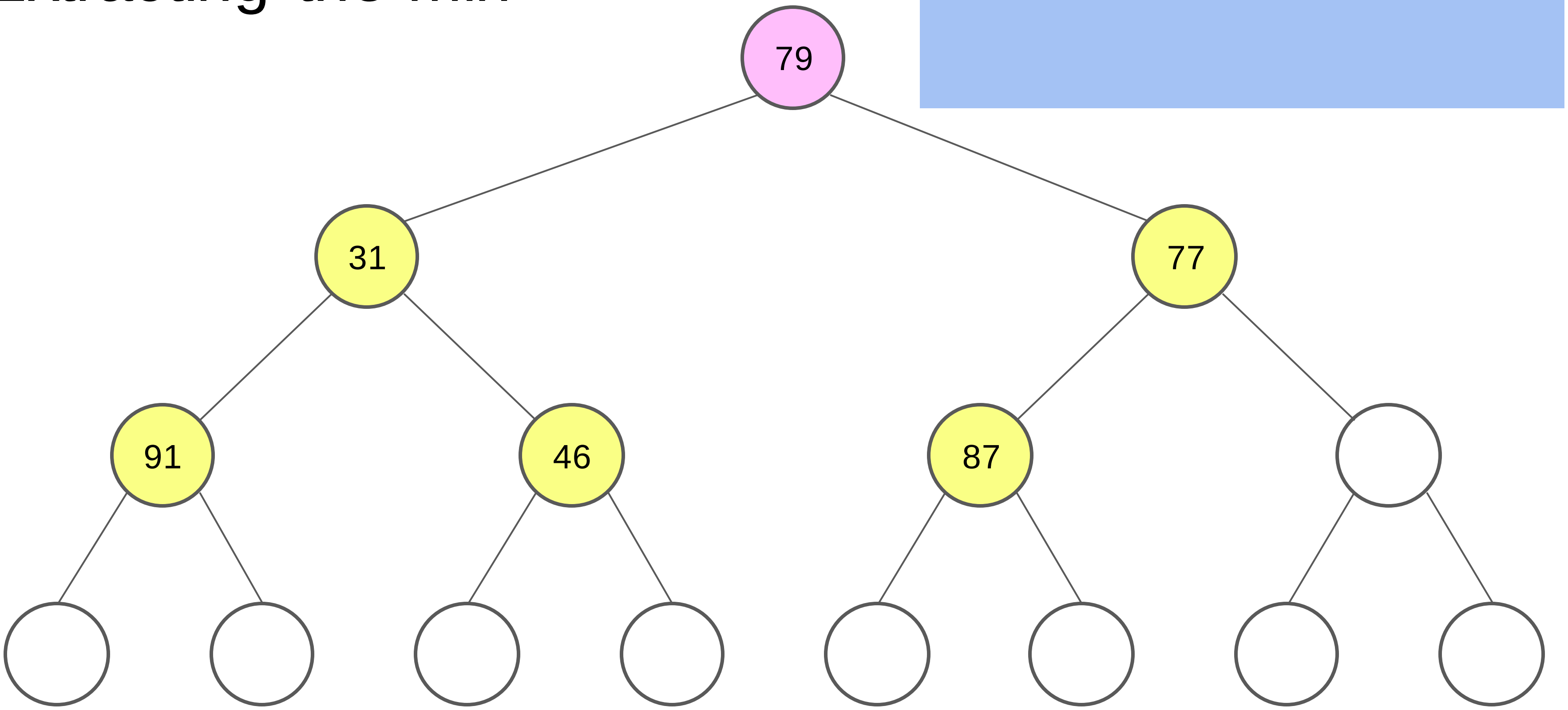
Extracting the min

- Move last element to root



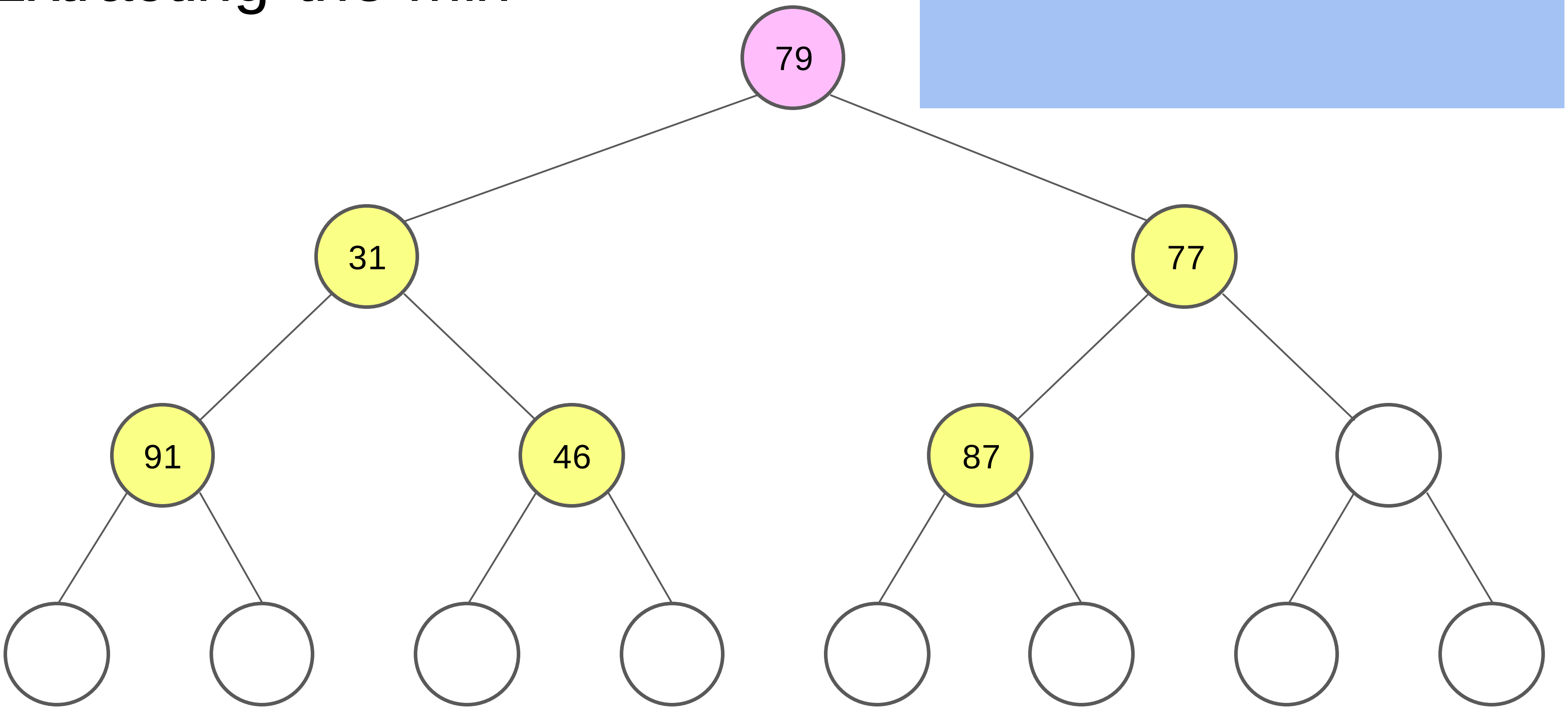
Extracting the min

- Move last element to root



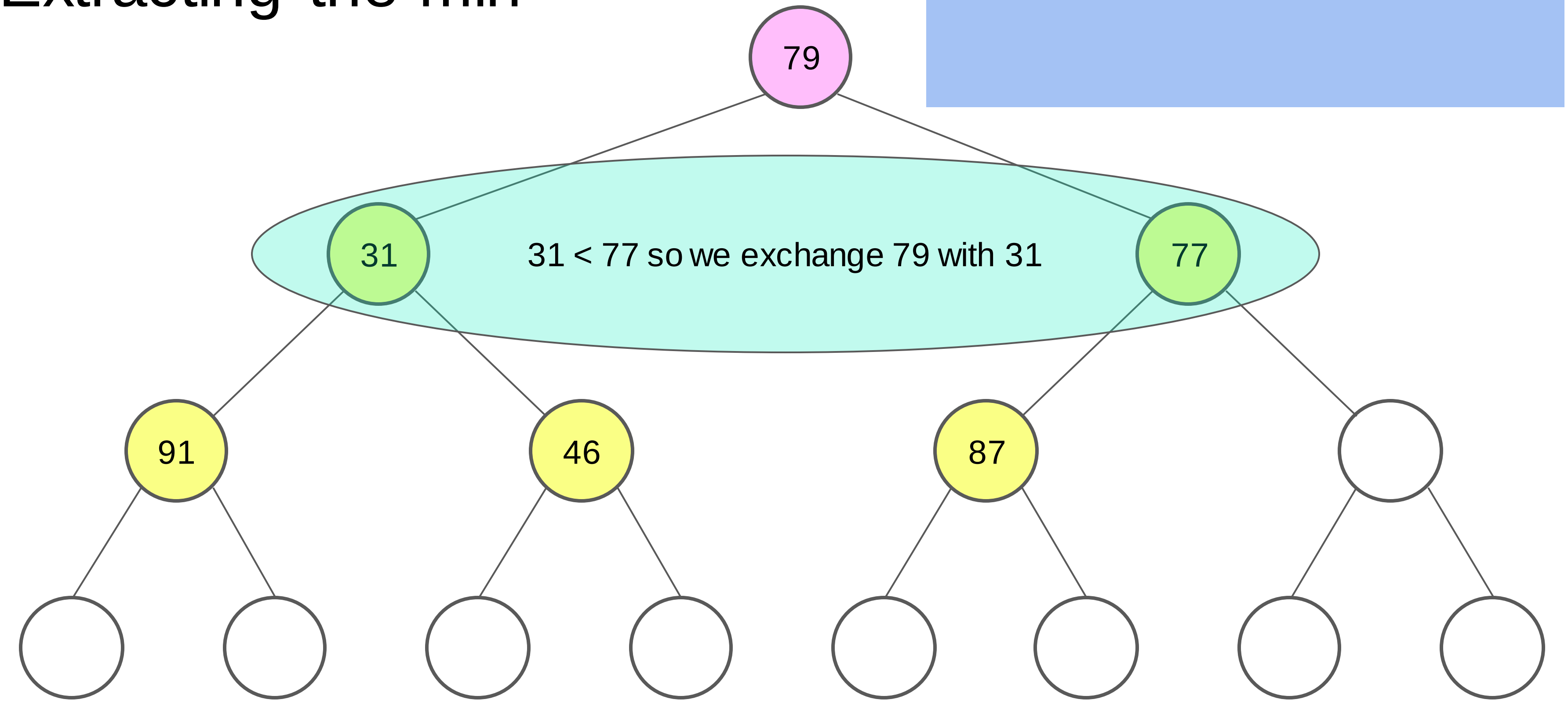
Extracting the min

- Heapify



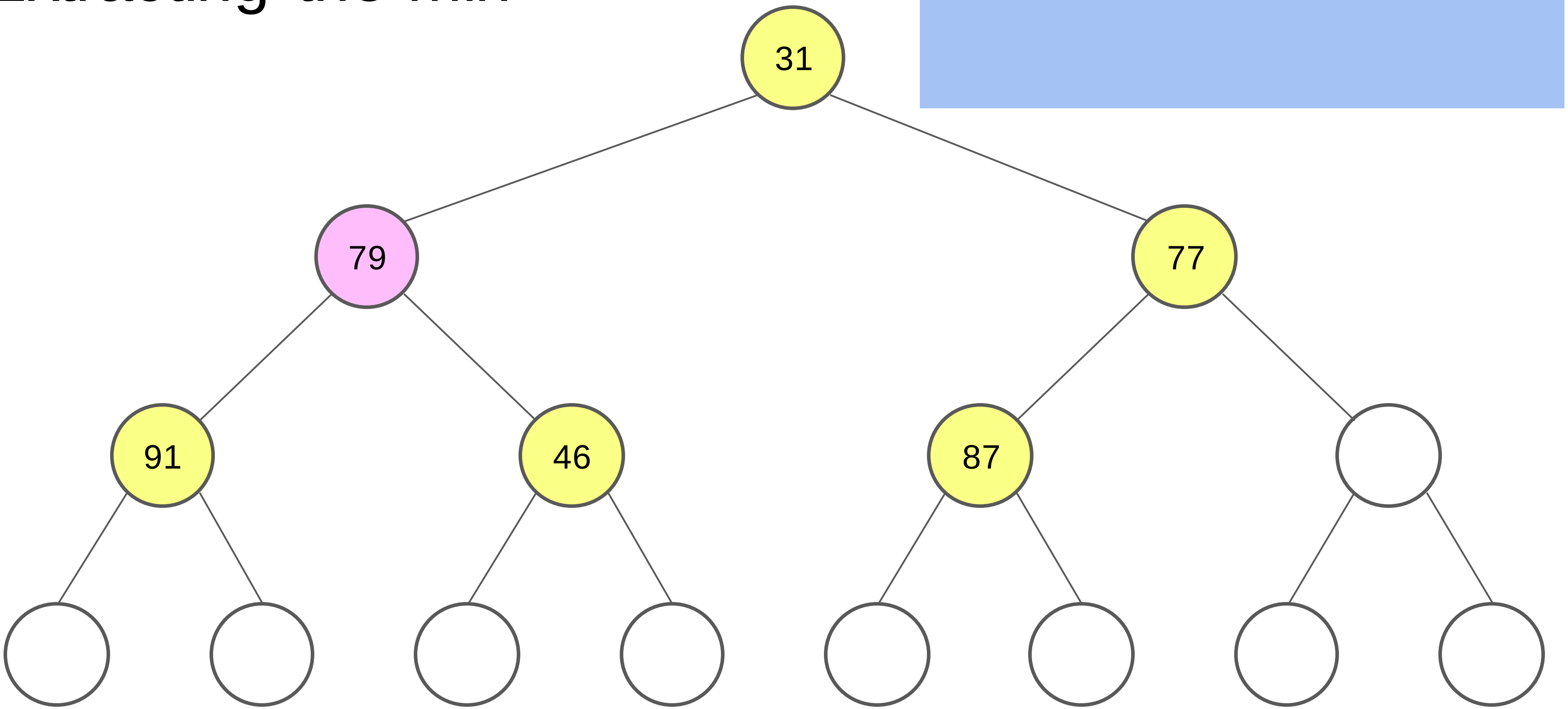
Extracting the min

- Heapify



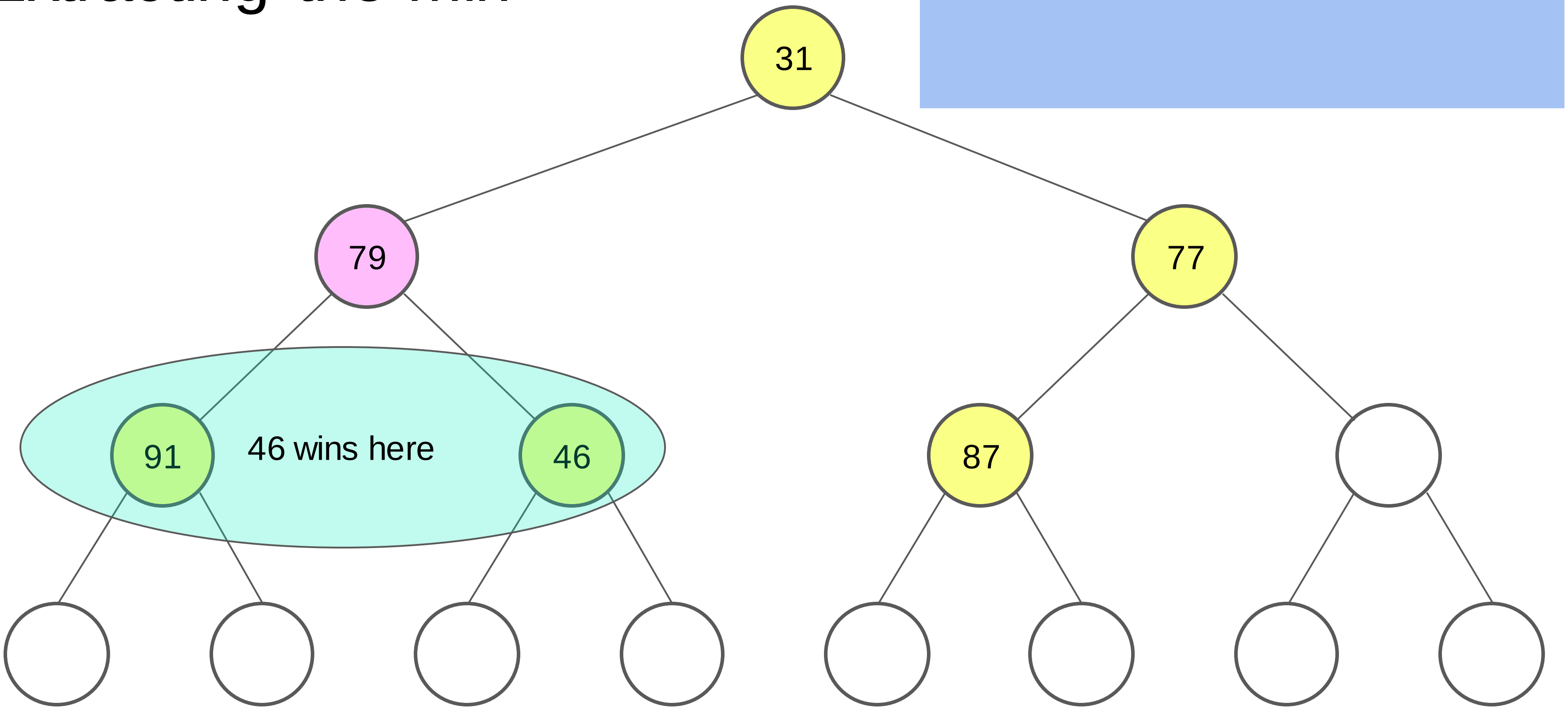
Extracting the min

- Heapify



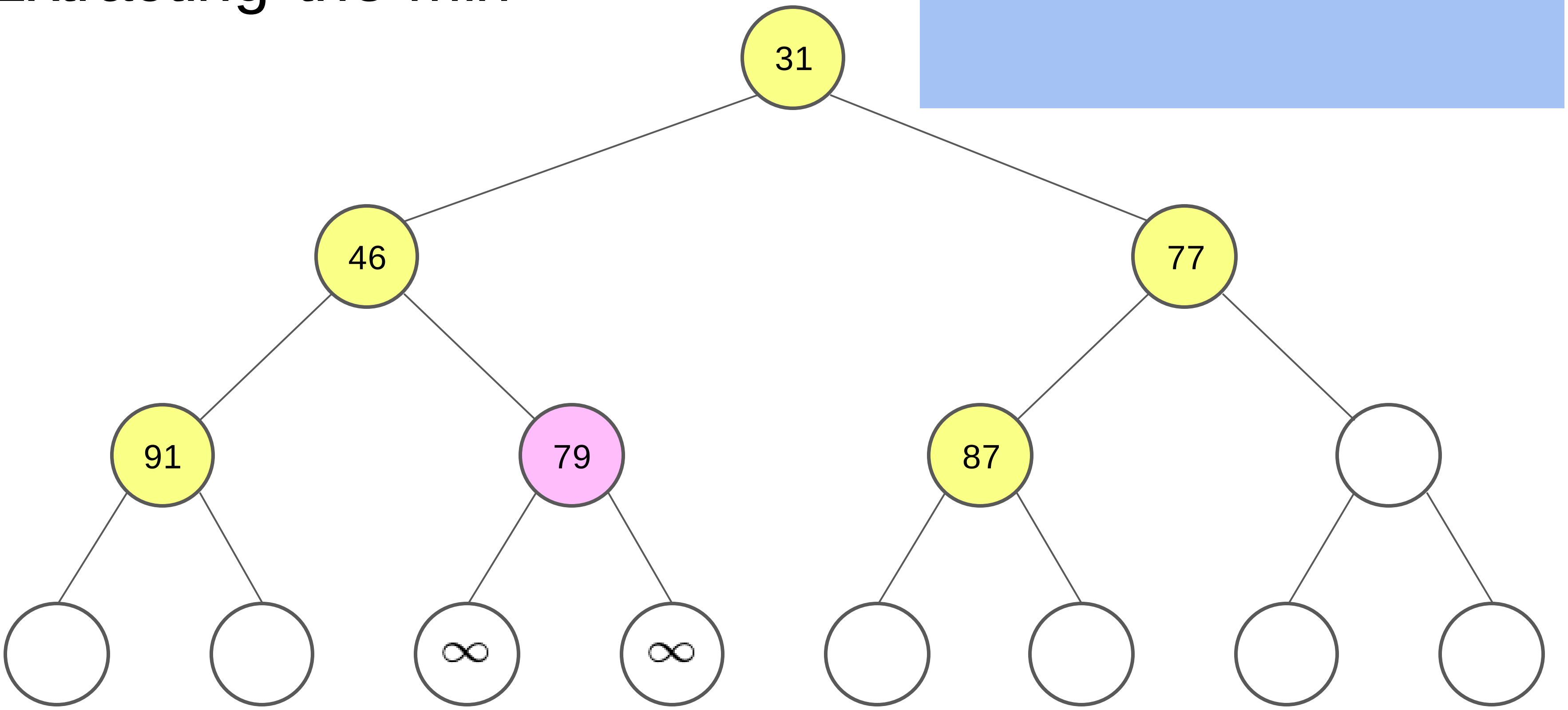
Extracting the min

- Heapify



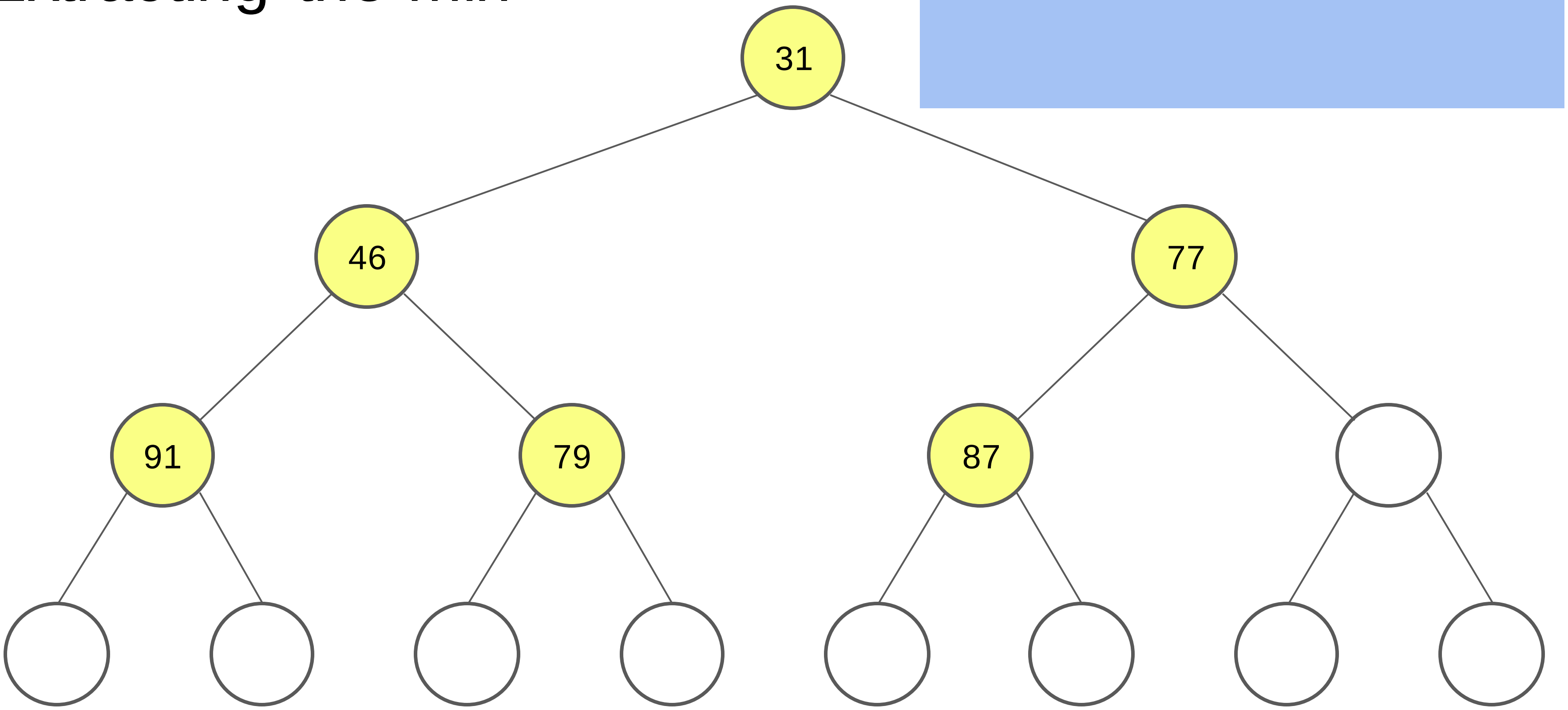
Extracting the min

- Heapify



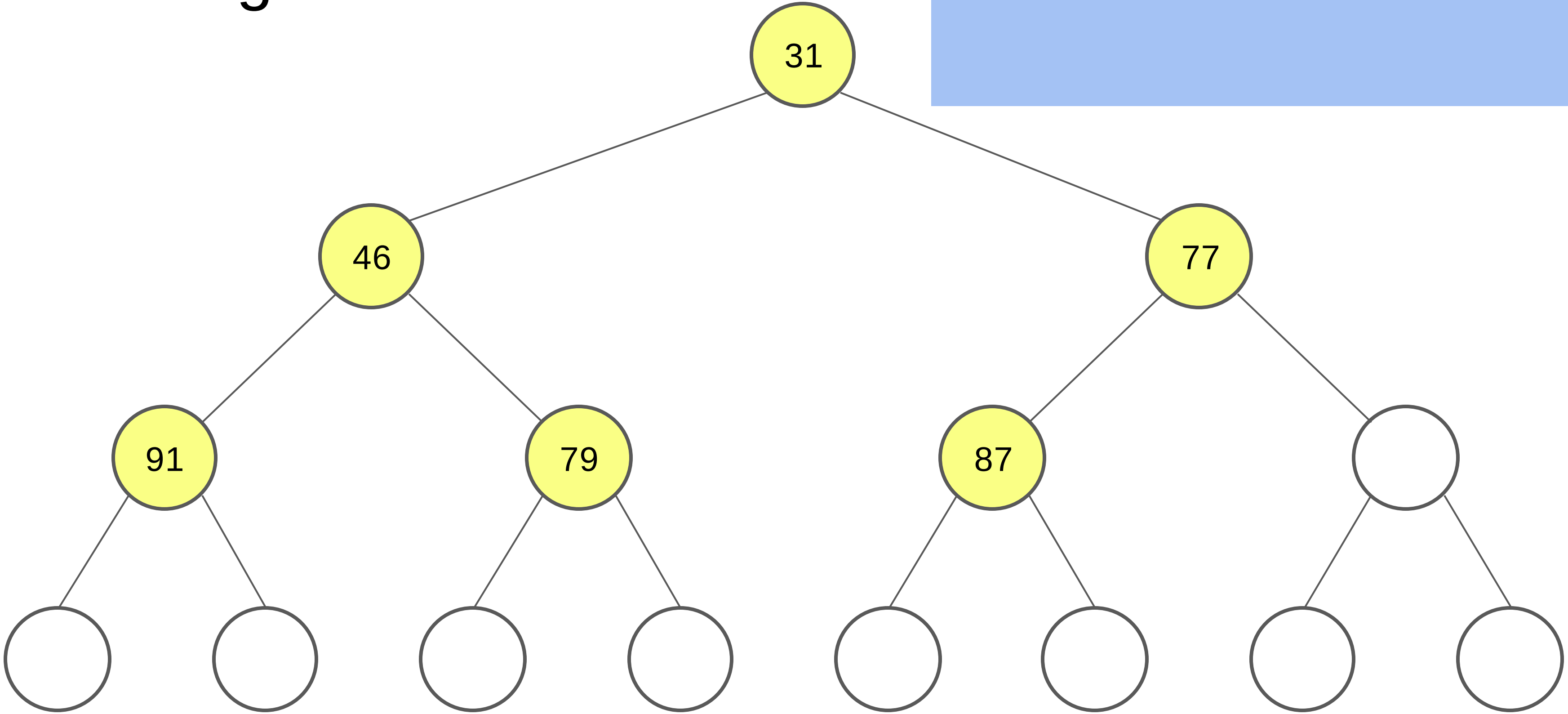
Extracting the min

- Done



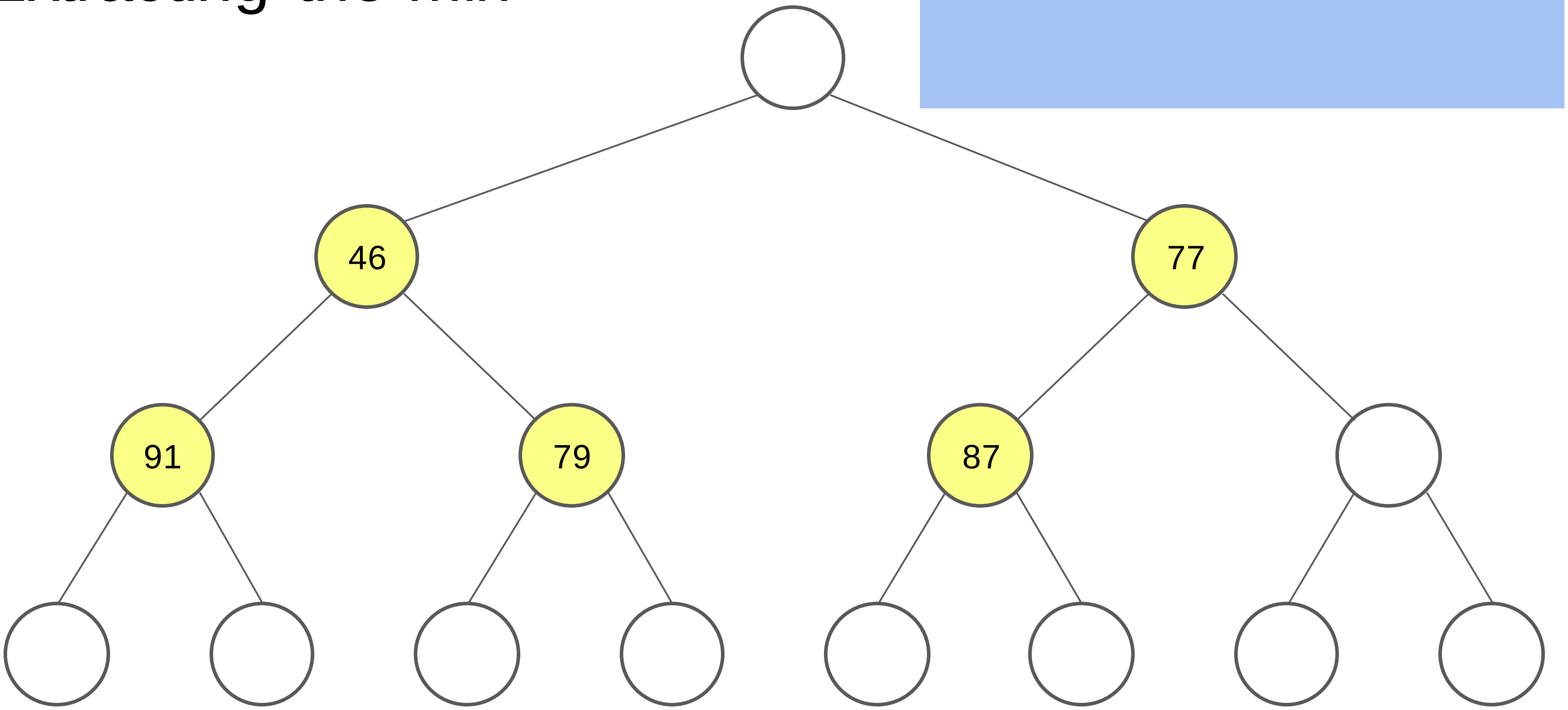
Extracting the min

- Again, 31 will be returned



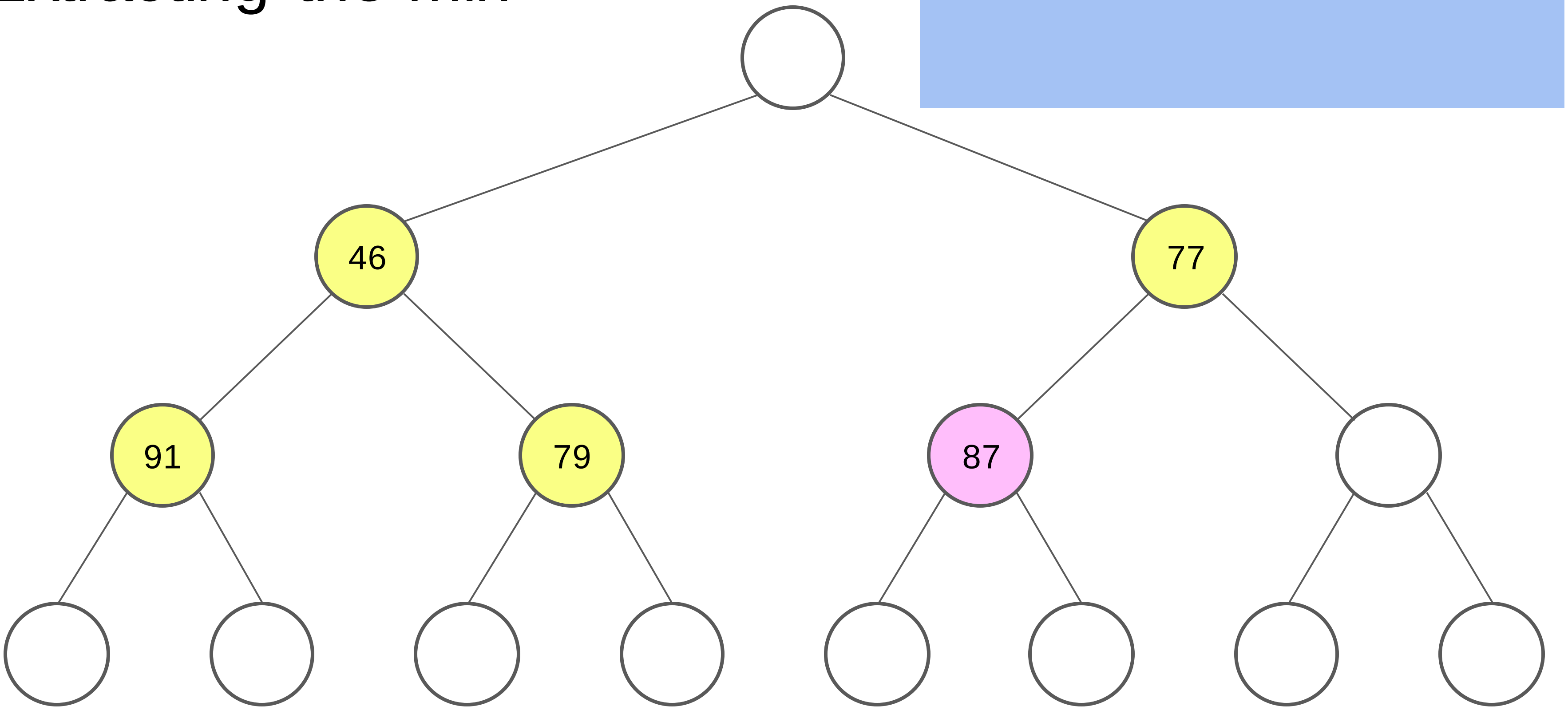
Extracting the min

- Again, 31 will be returned
- Tell me what to do !!



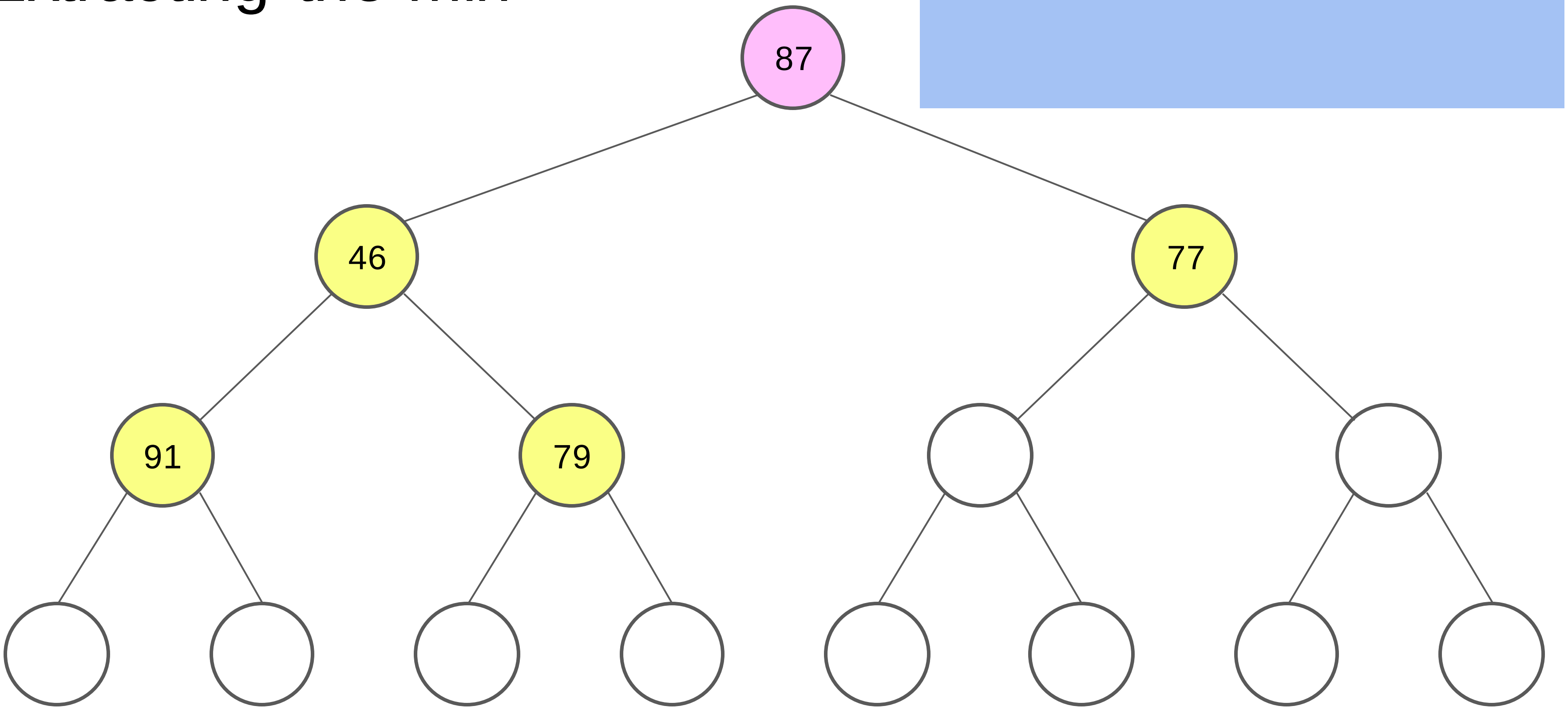
Extracting the min

- Tell me what to do



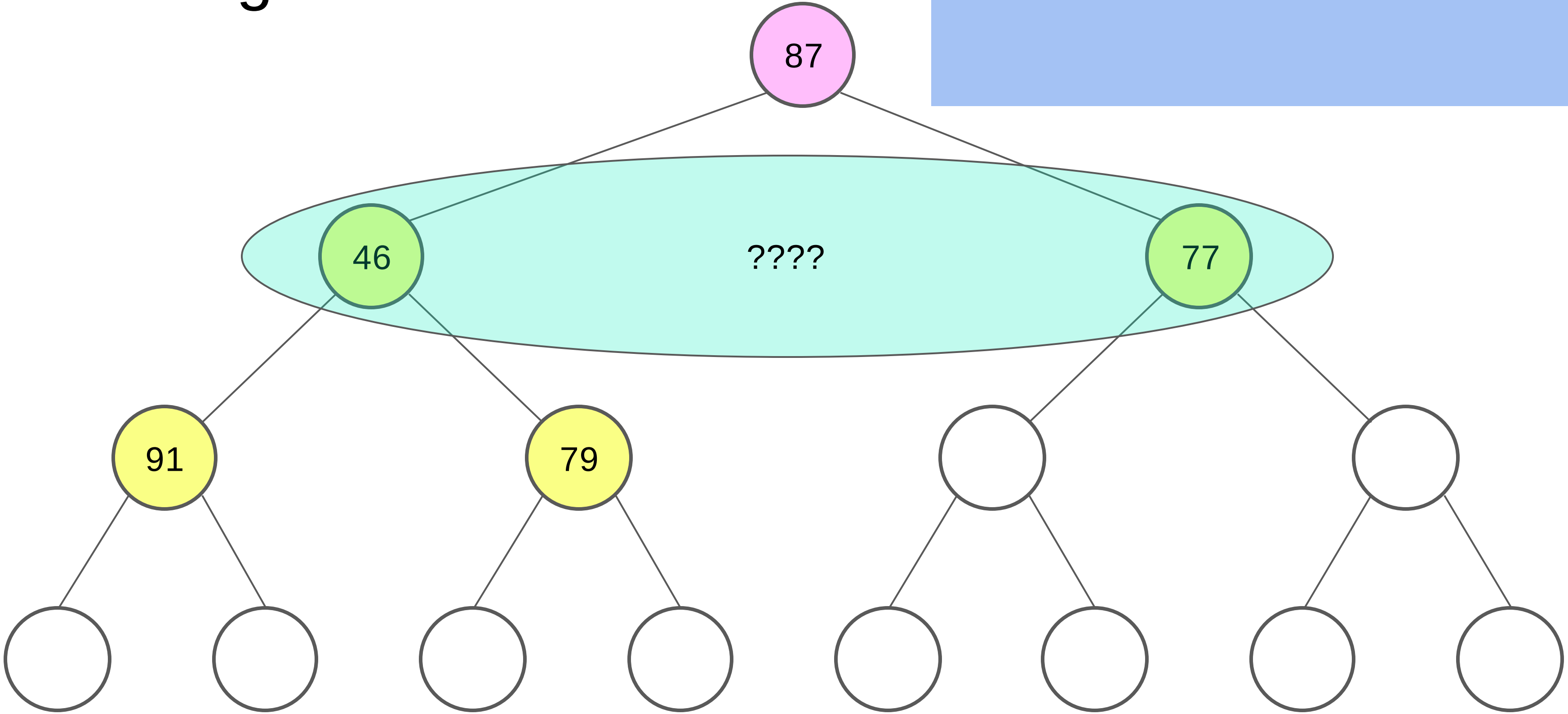
Extracting the min

- Tell me what to do



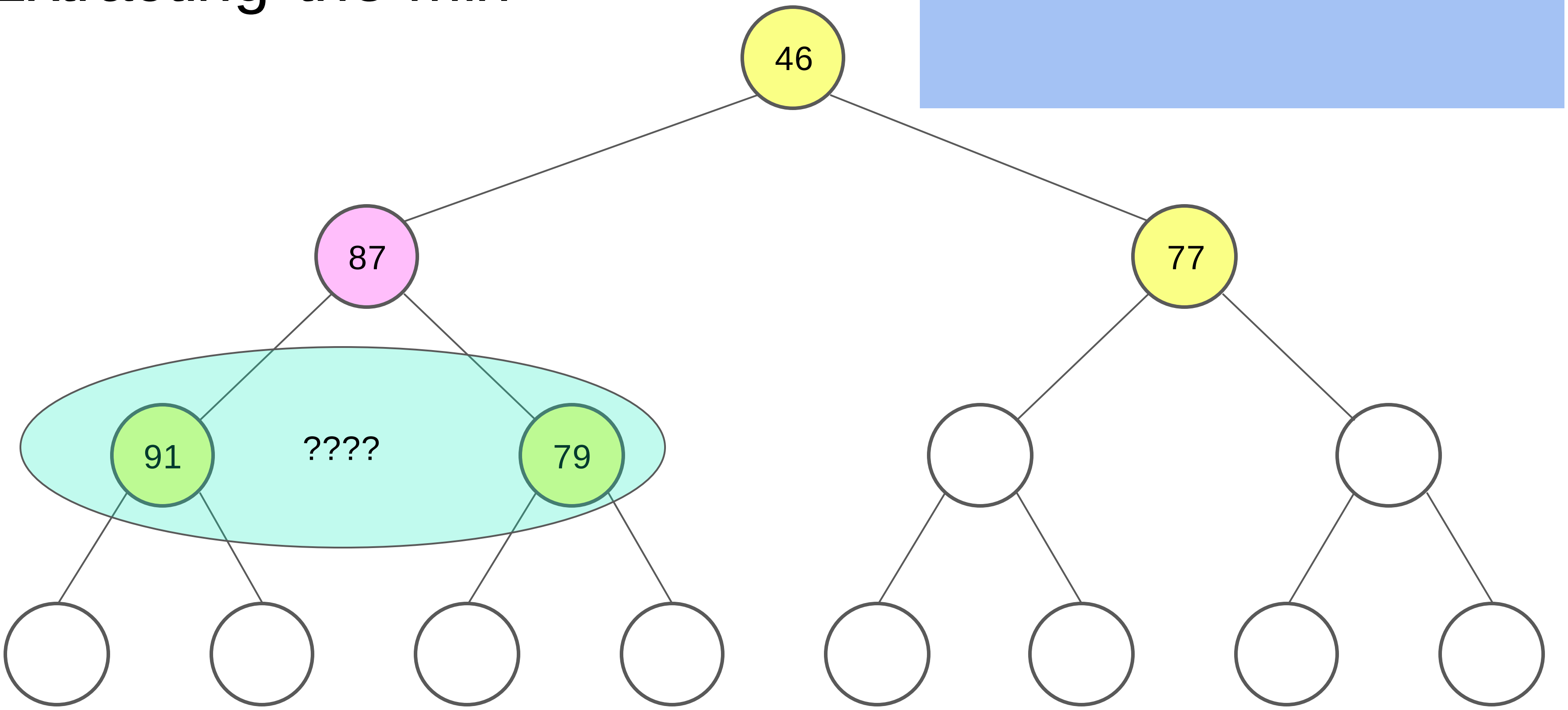
Extracting the min

- Tell me what to do



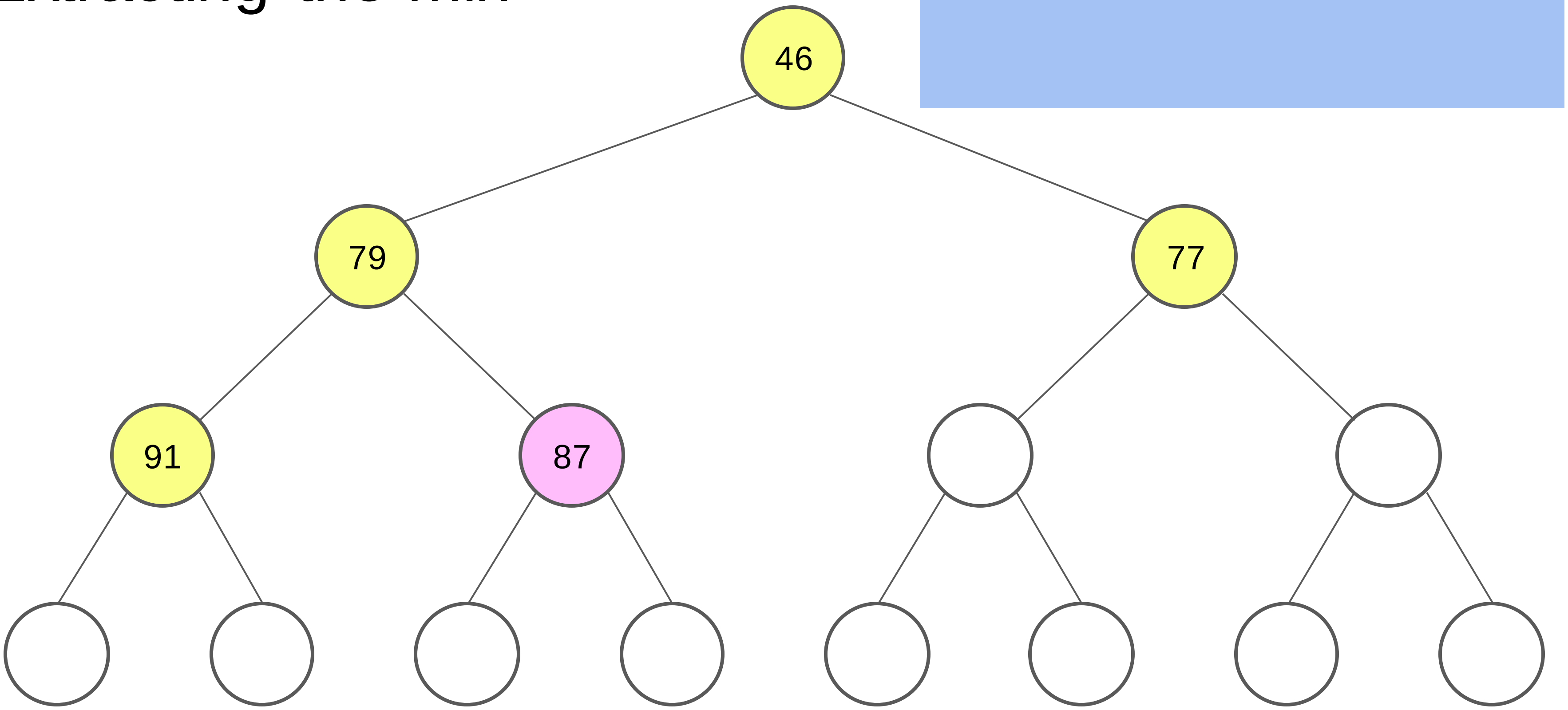
Extracting the min

- Tell me what to do



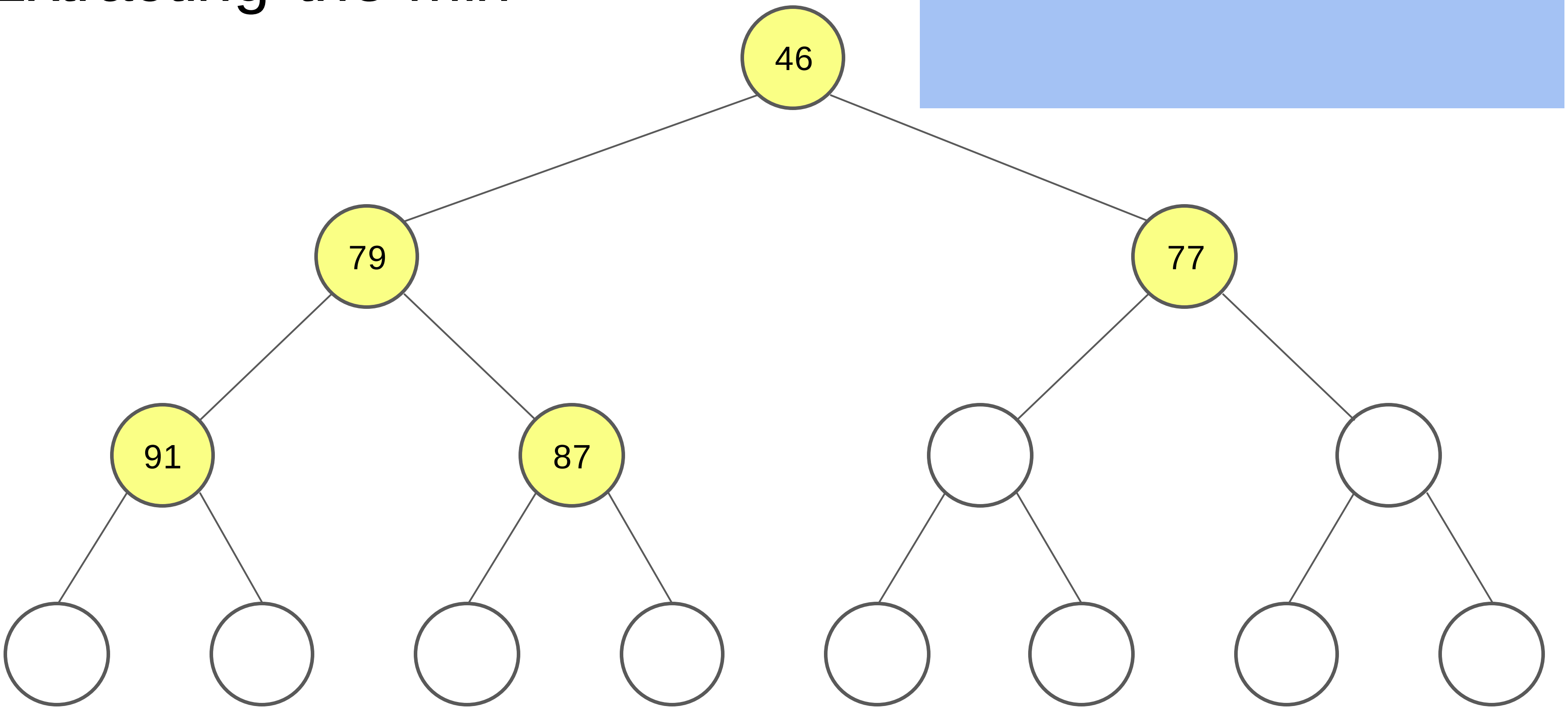
Extracting the min

- Tell me what to do



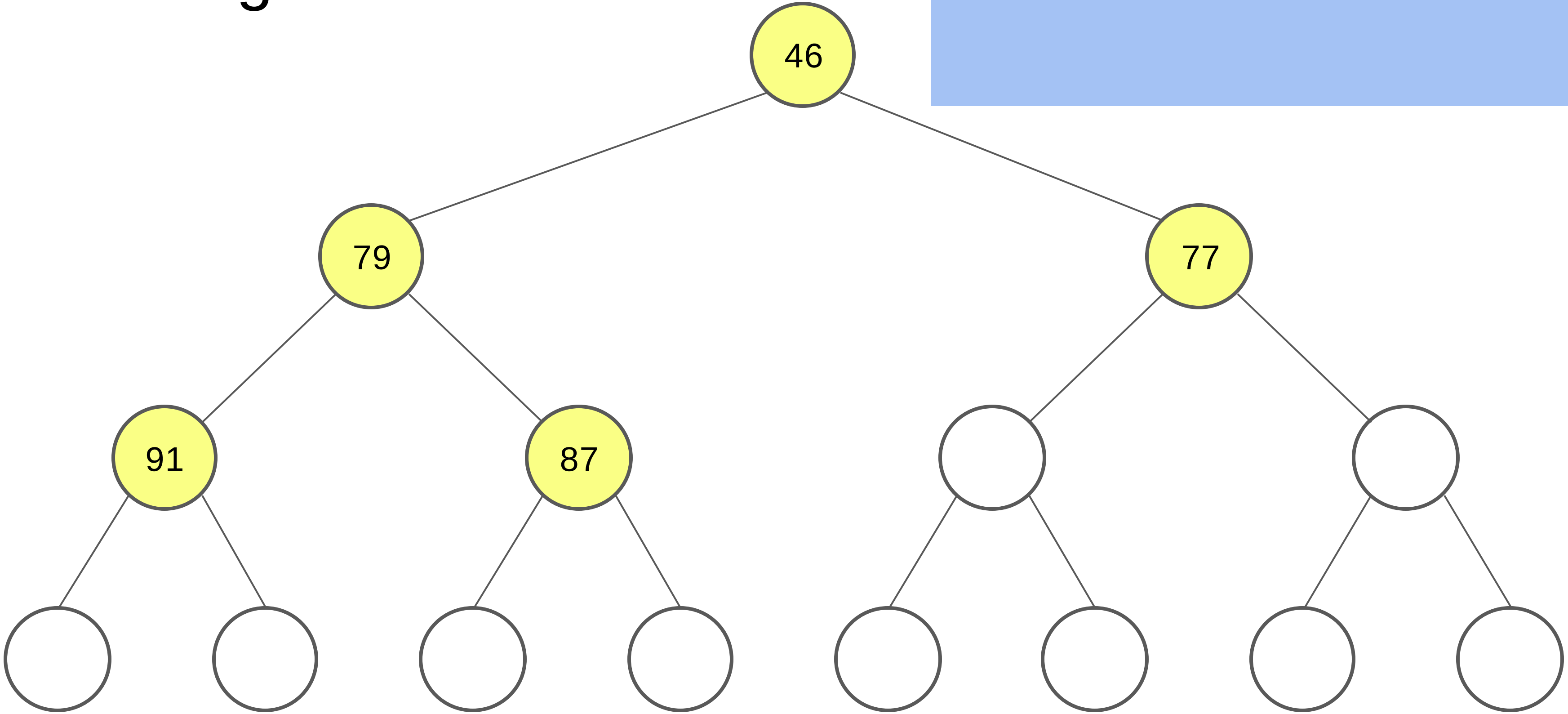
Extracting the min

- Tell me what to do



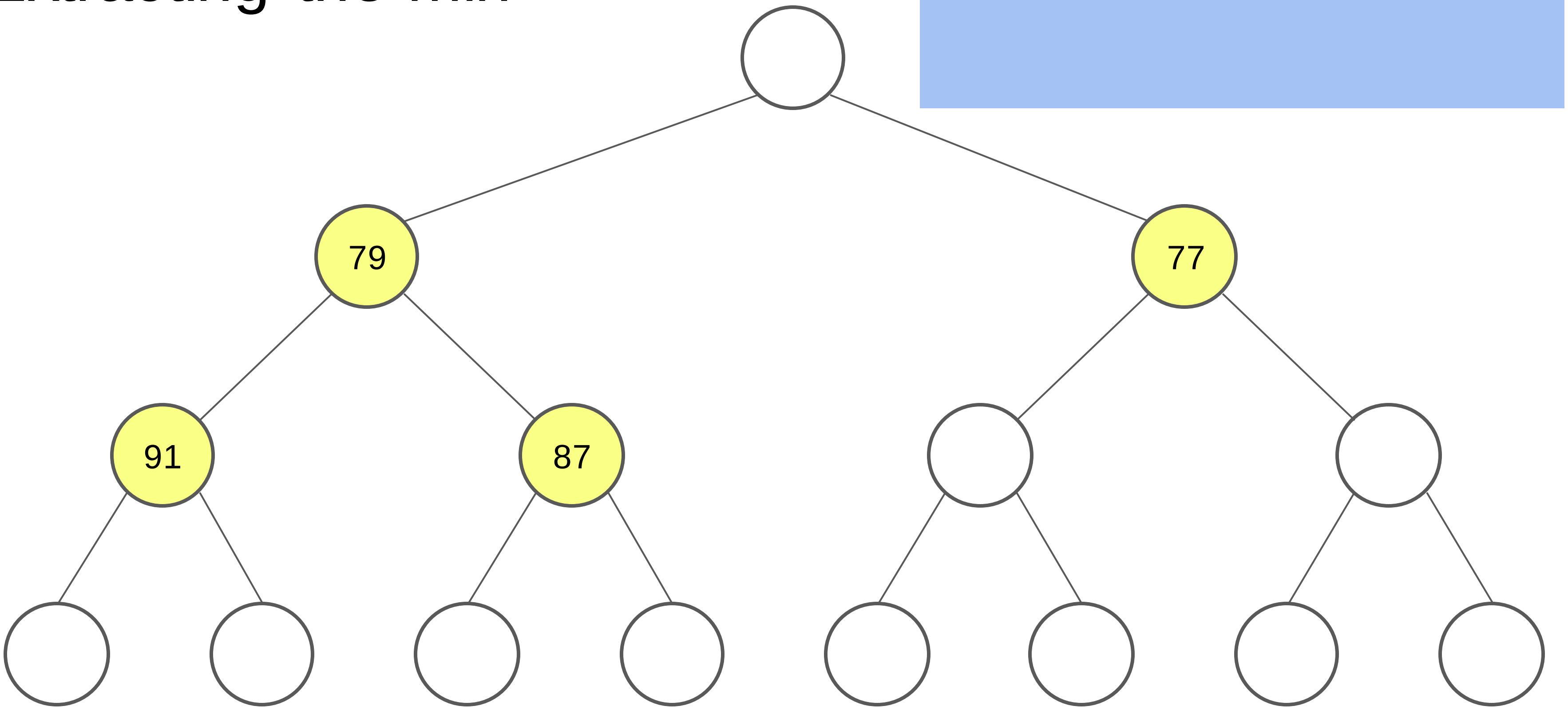
Extracting the min

- Again, 46 will be returned



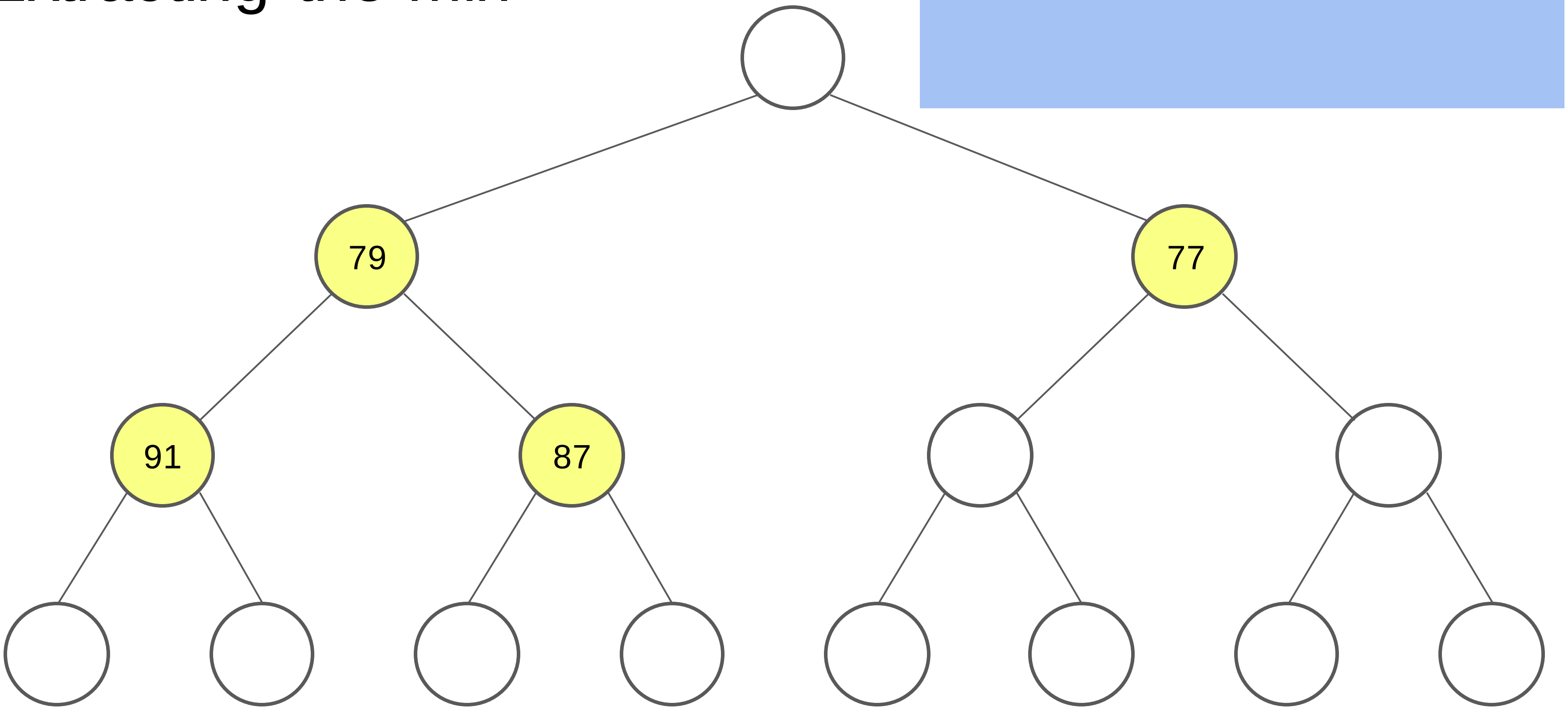
Extracting the min

- Again, 46 will be returned



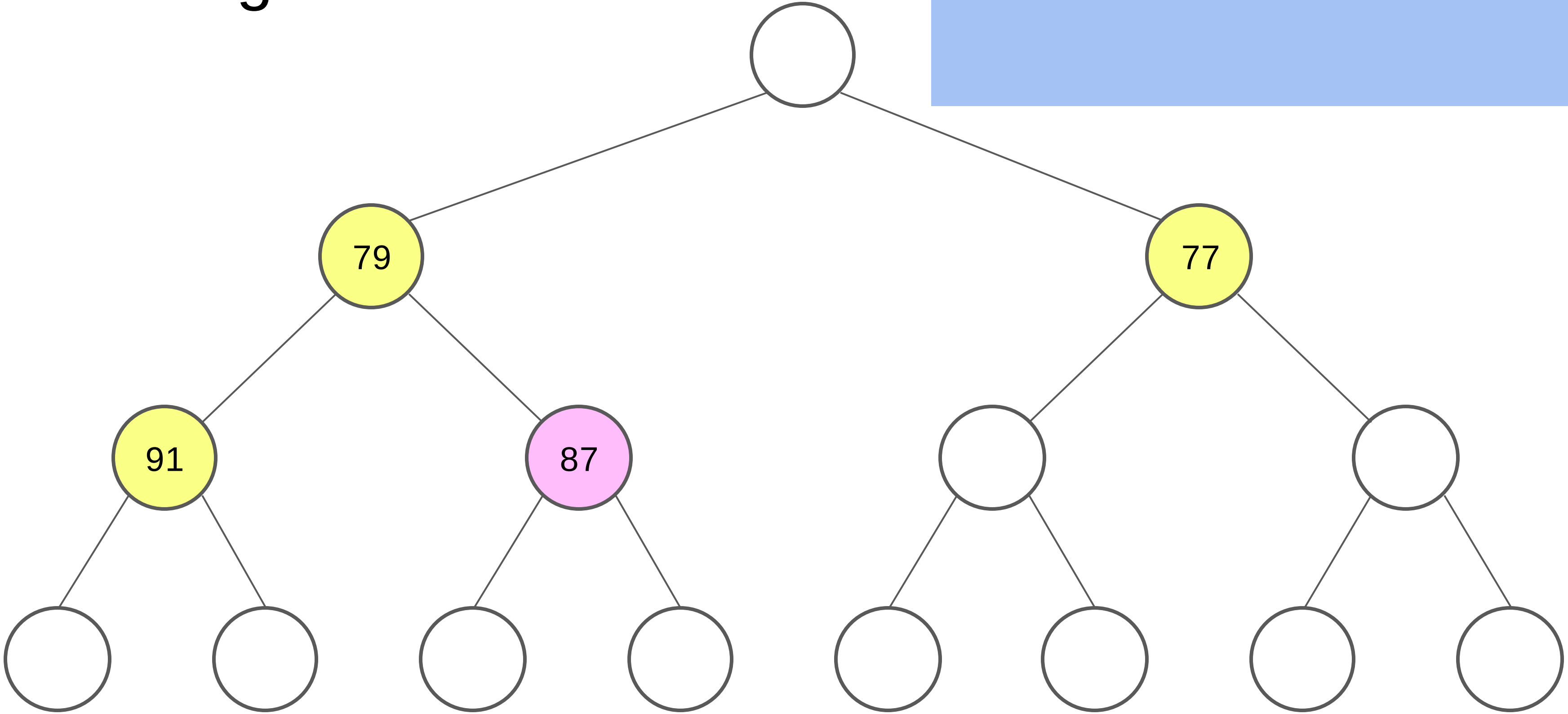
Extracting the min

- What should I do?



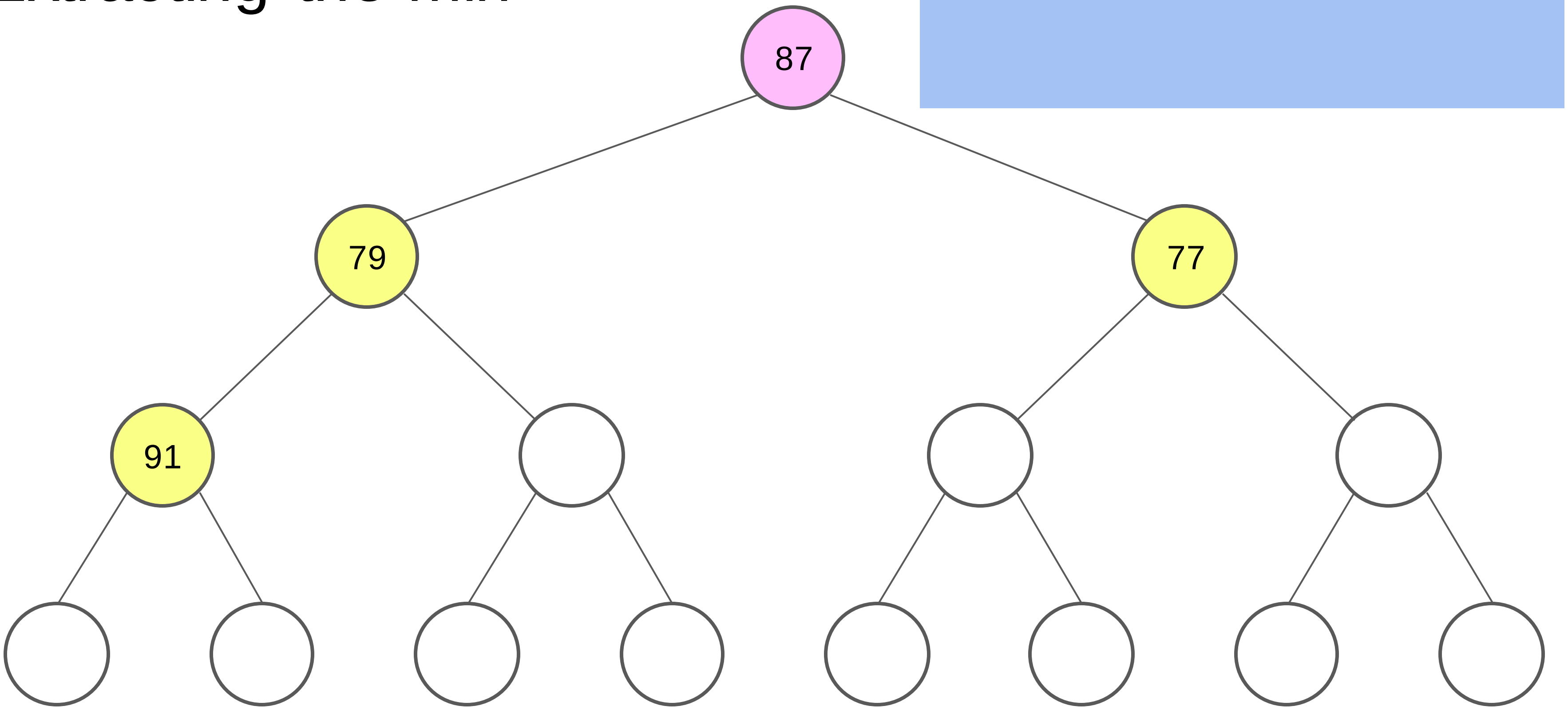
Extracting the min

- What should I do?



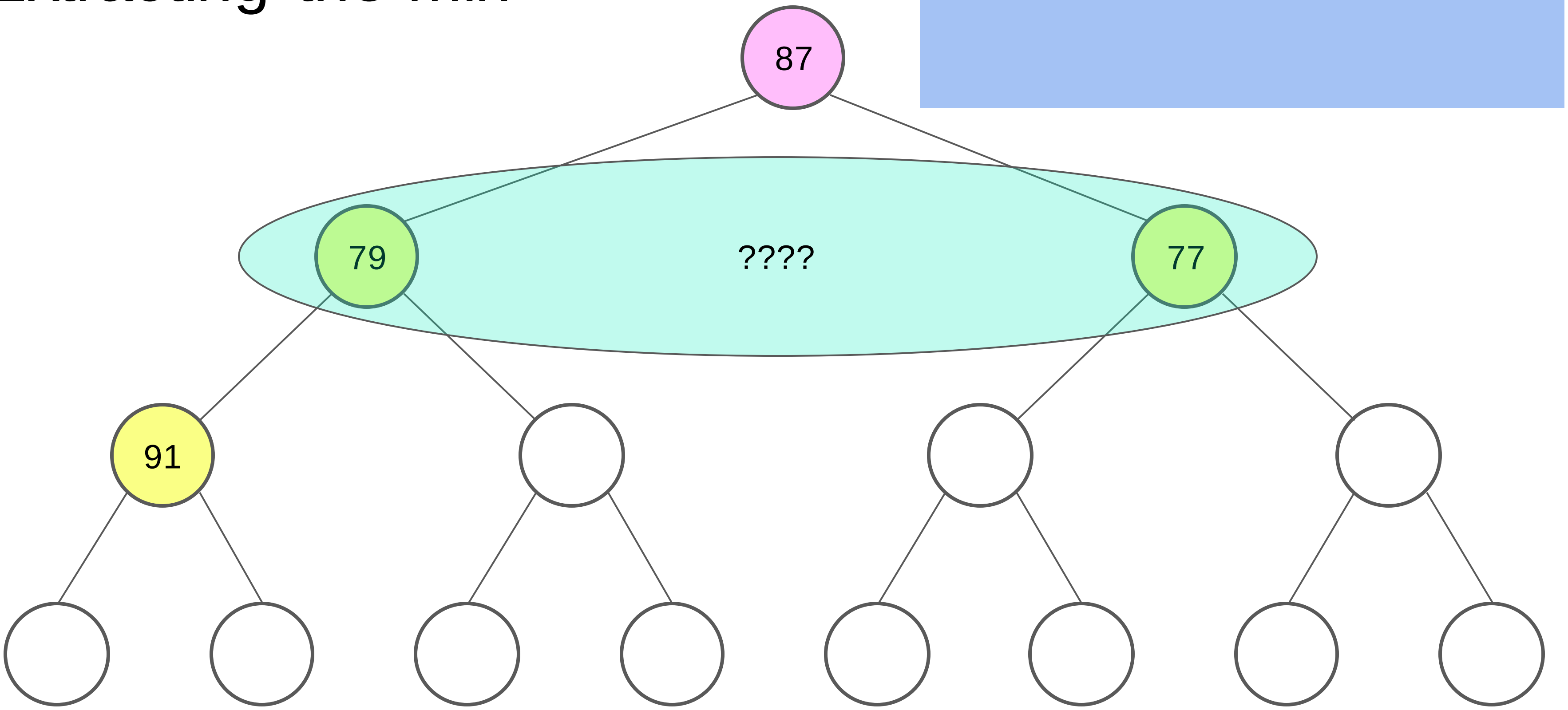
Extracting the min

- What should I do?



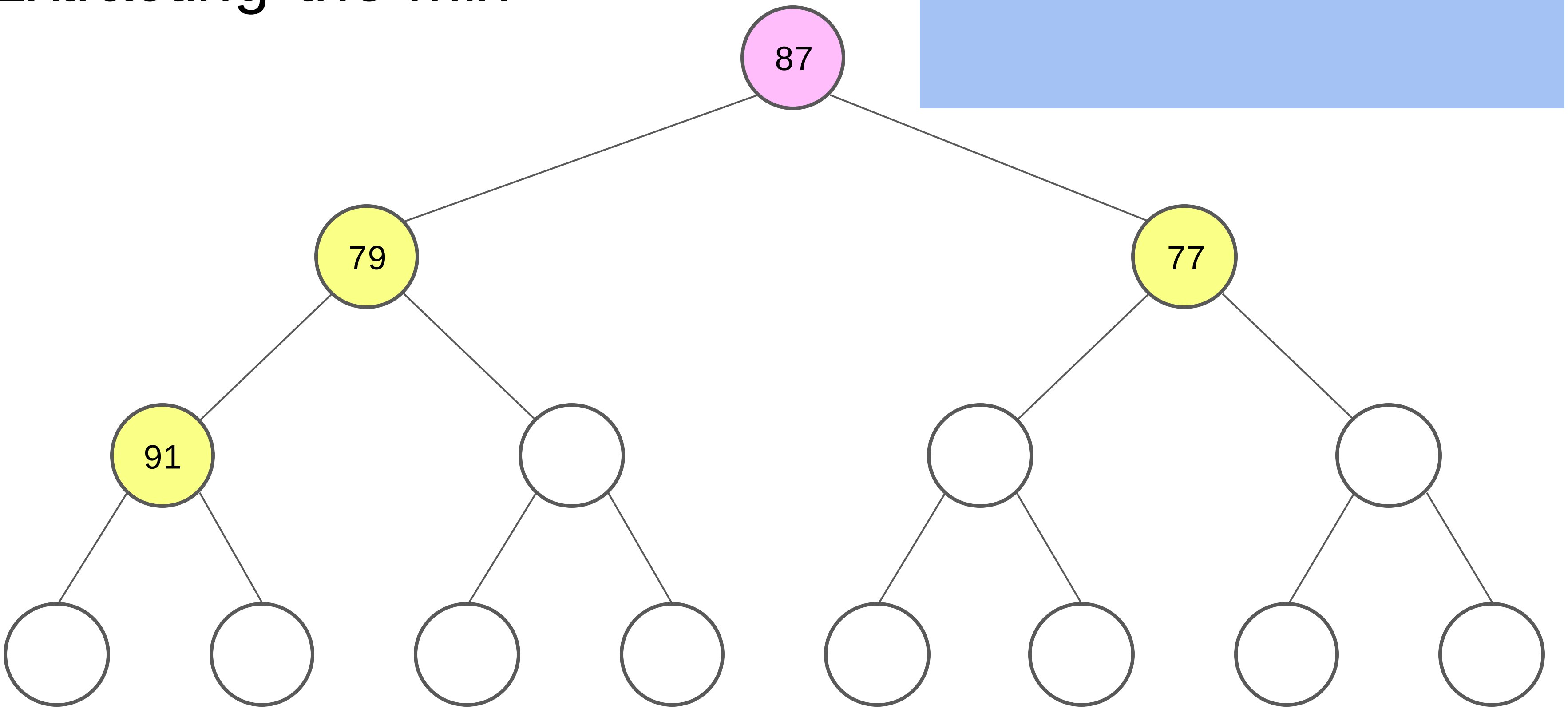
Extracting the min

- What should I do?



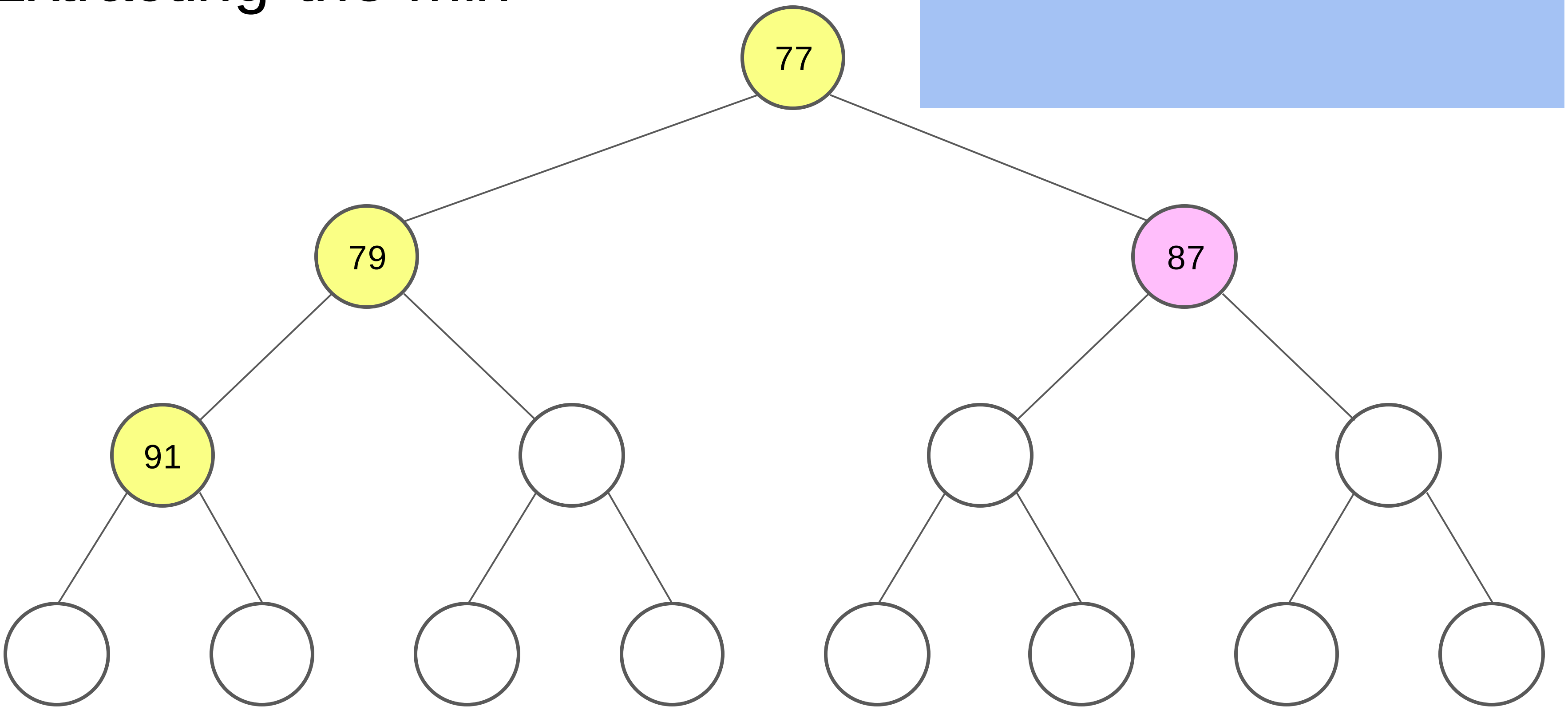
Extracting the min

- What should I do?



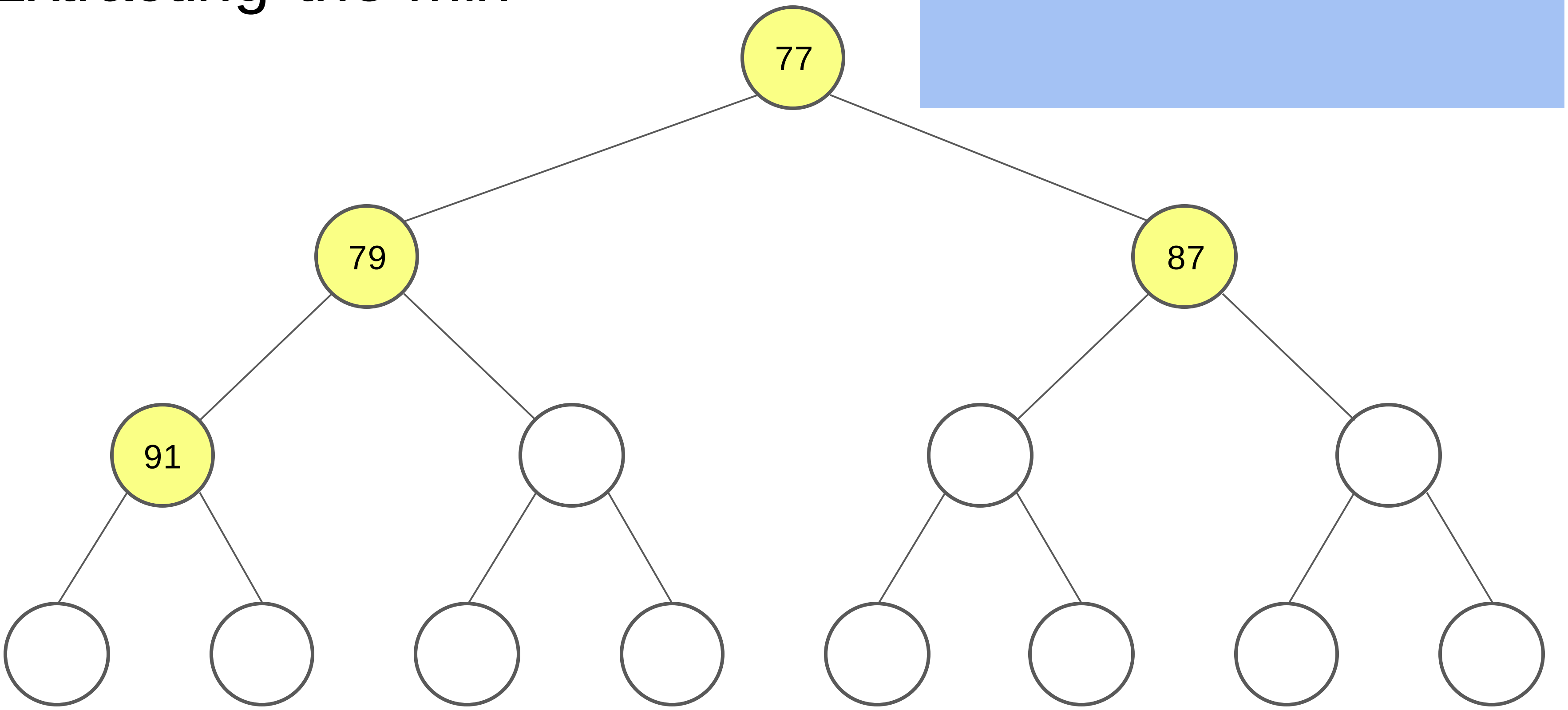
Extracting the min

- What should I do?



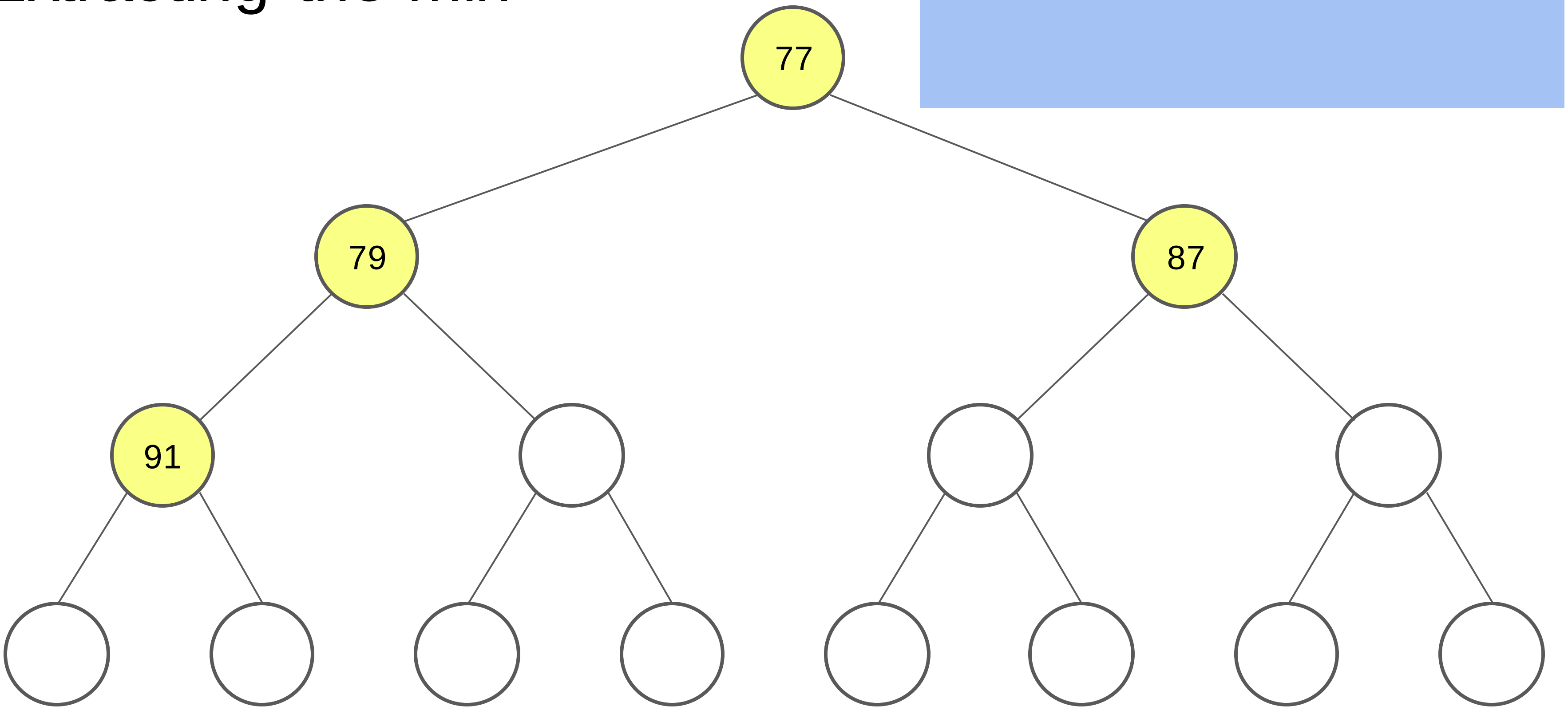
Extracting the min

- What should I do?



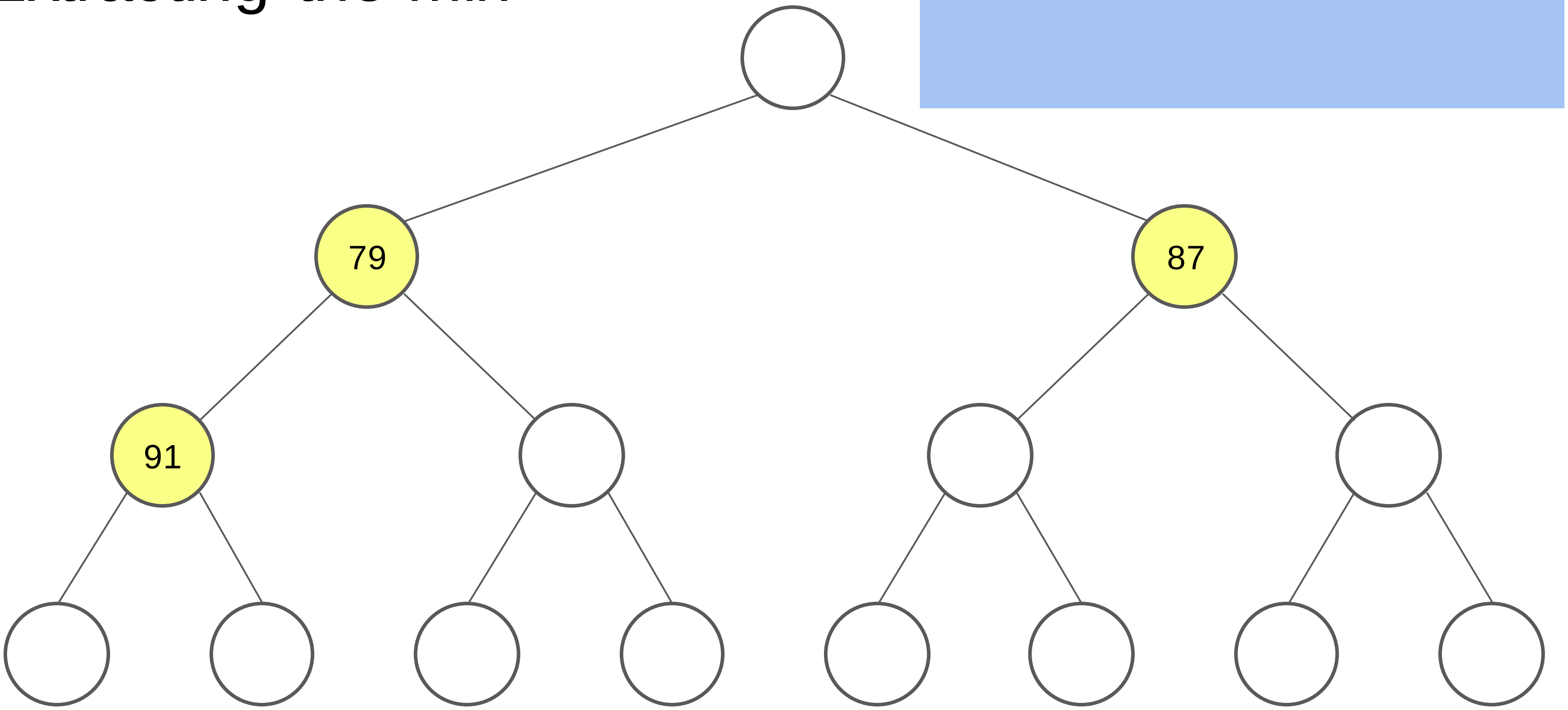
Extracting the min

- Again, 77 will be returned



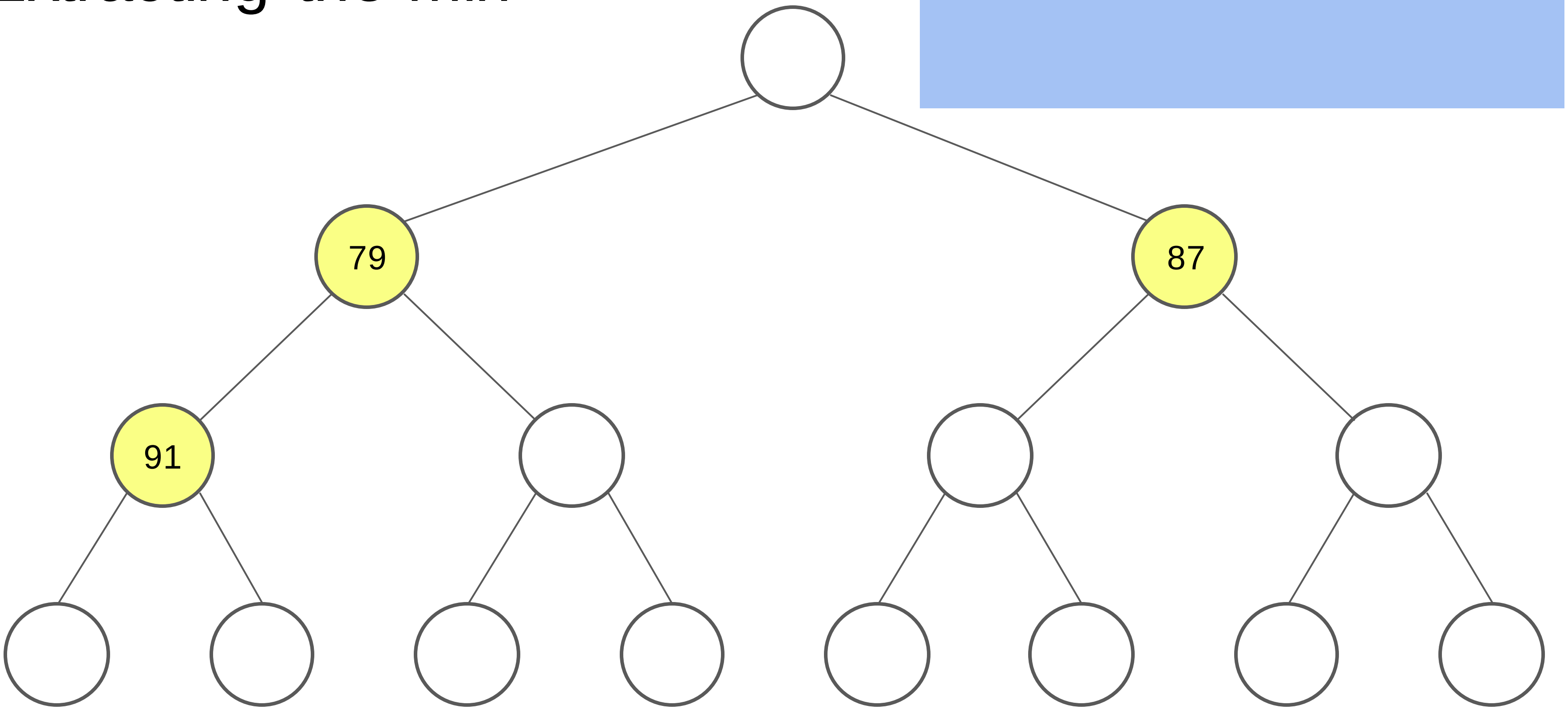
Extracting the min

- Again, 77 will be returned



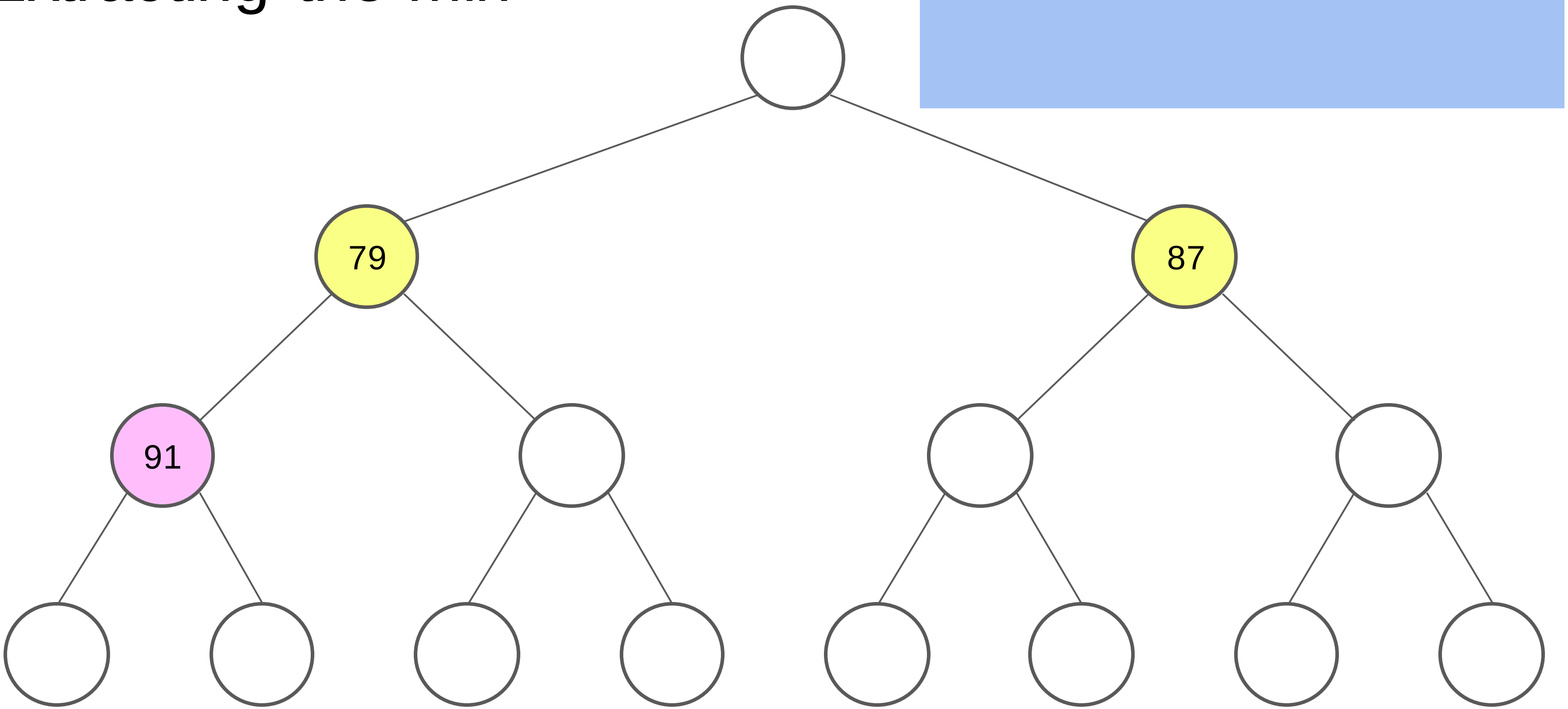
Extracting the min

- What to do?



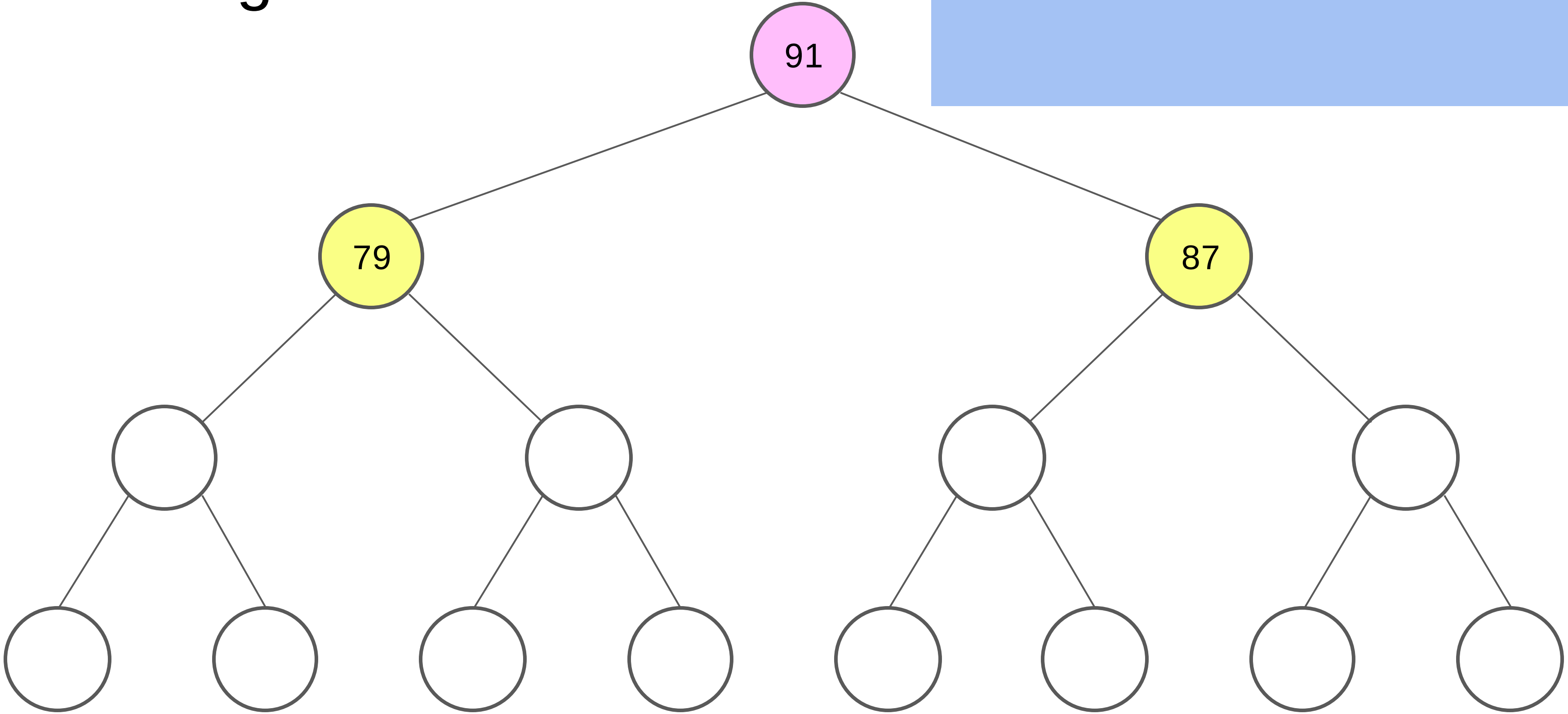
Extracting the min

- What to do?



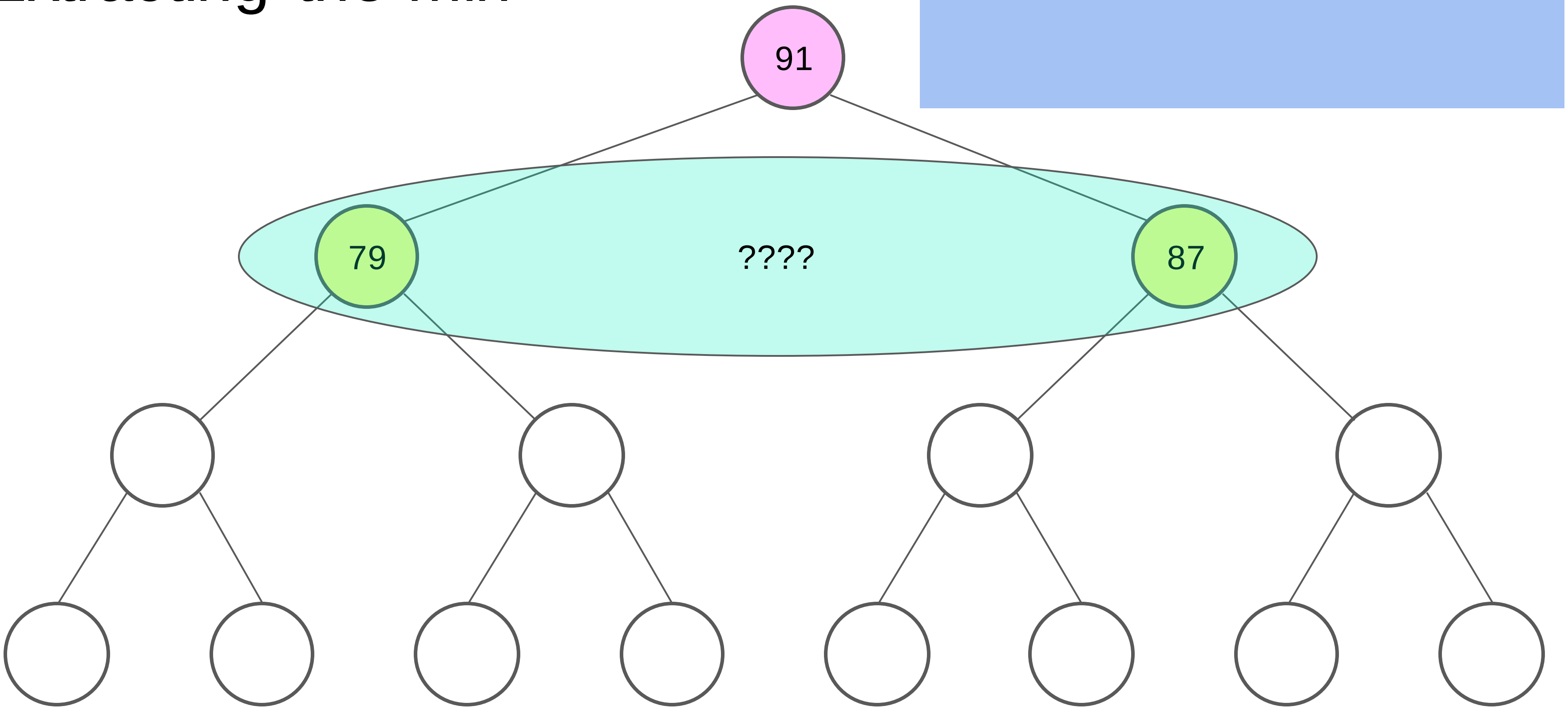
Extracting the min

- What to do?



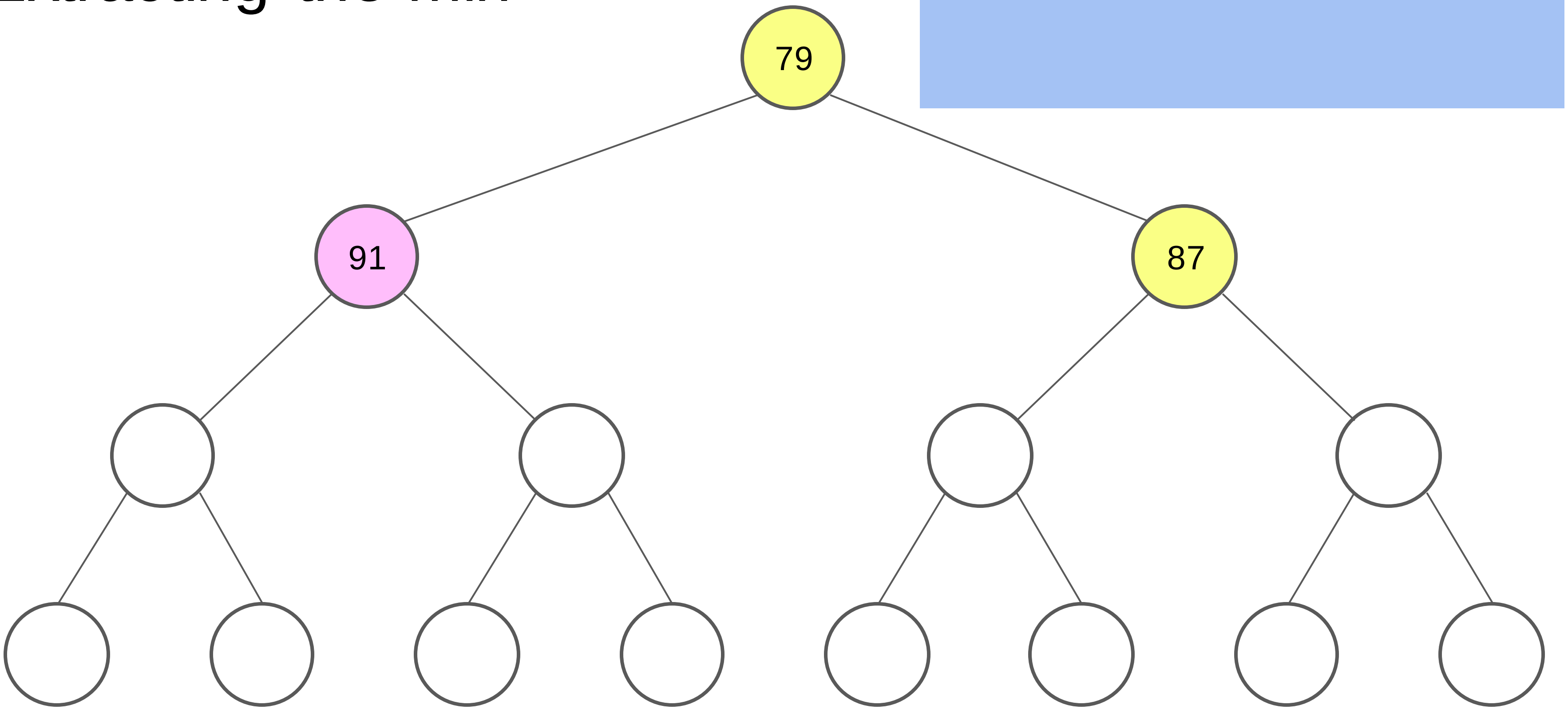
Extracting the min

- What to do?



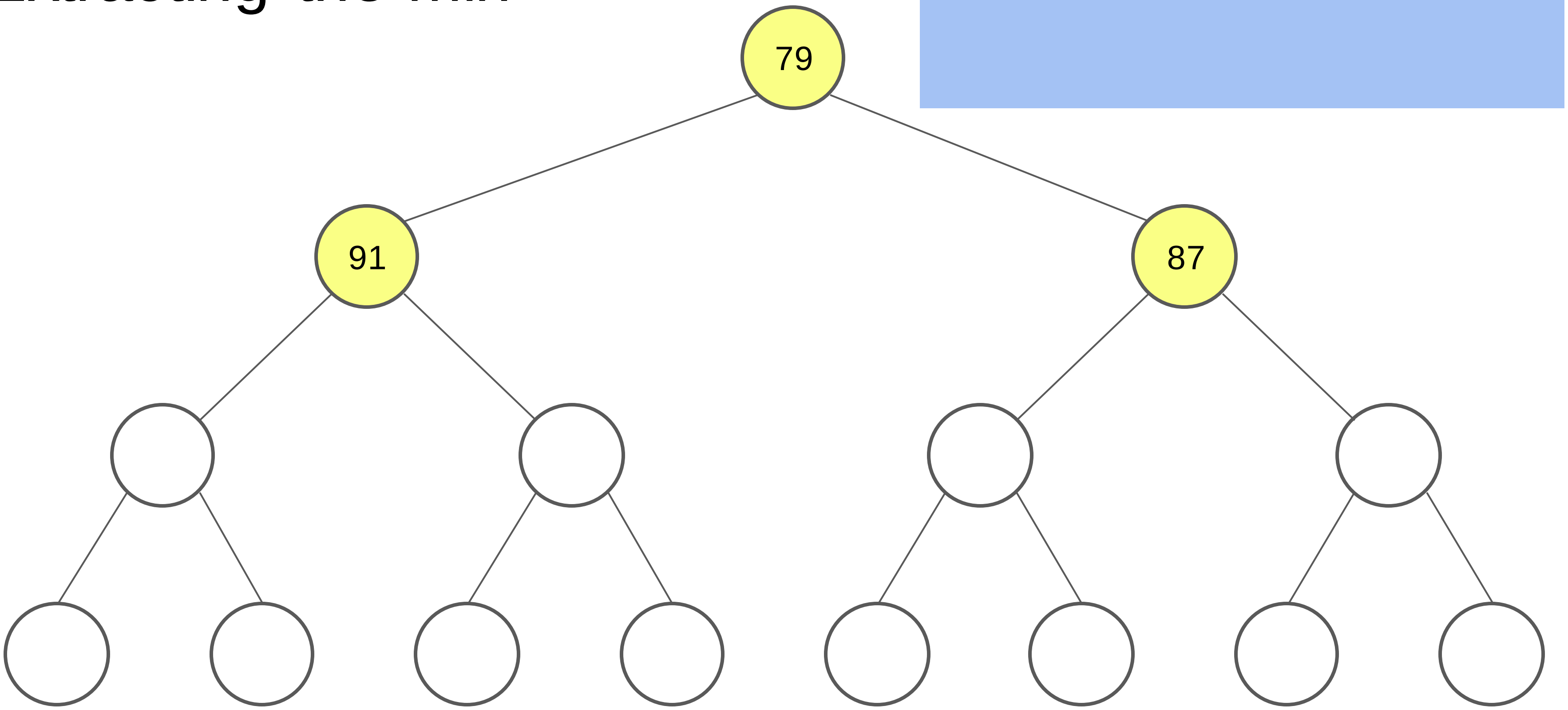
Extracting the min

- What to do?



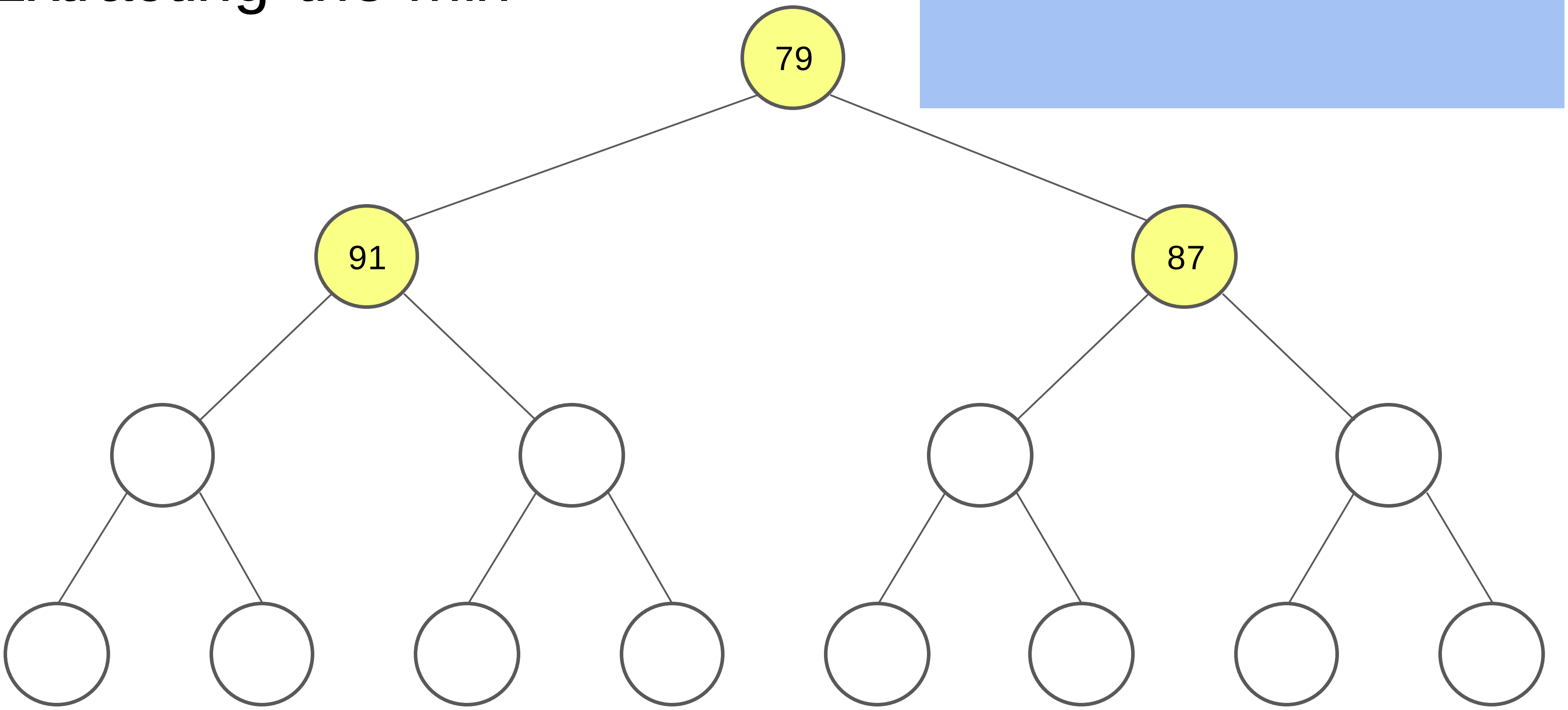
Extracting the min

- What to do?



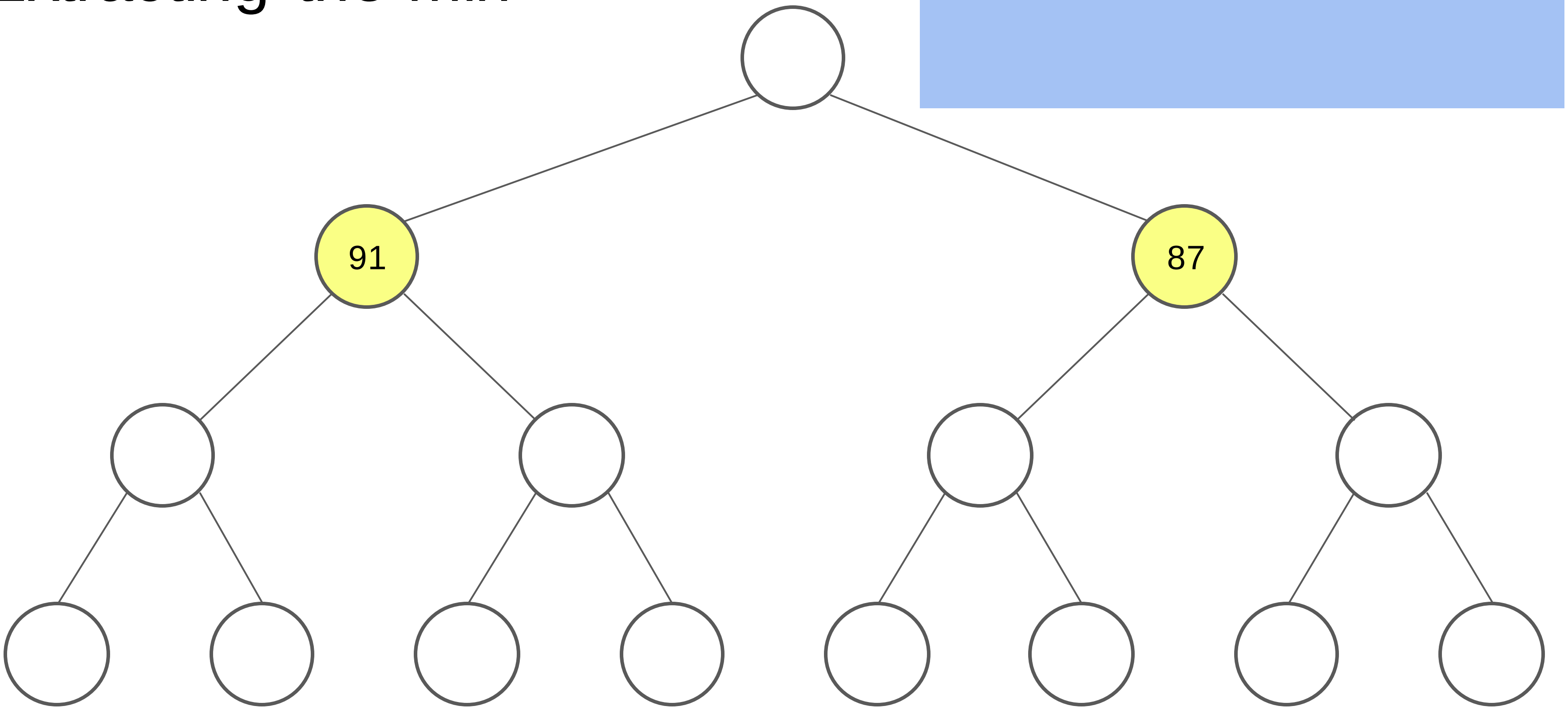
Extracting the min

- 79 is returned next



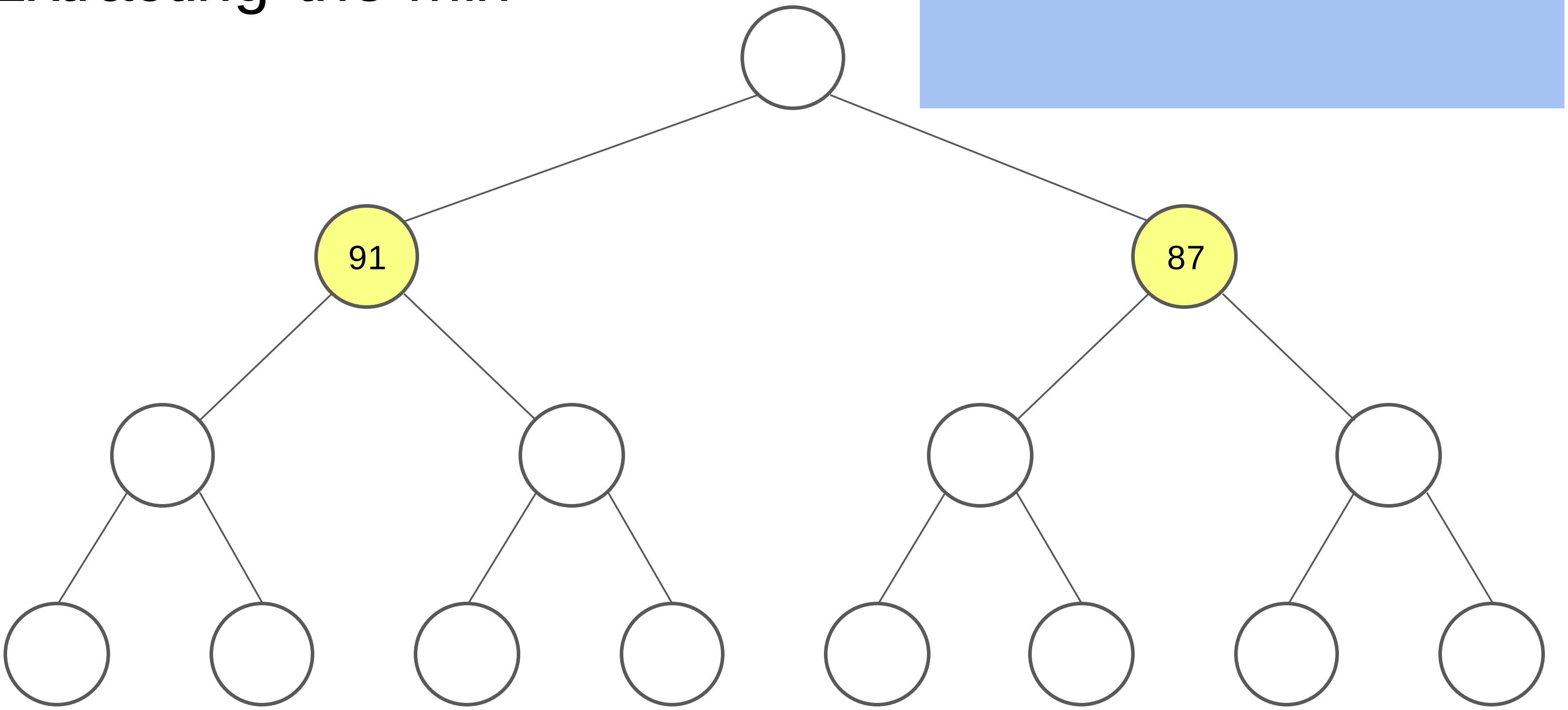
Extracting the min

- 79 is returned next



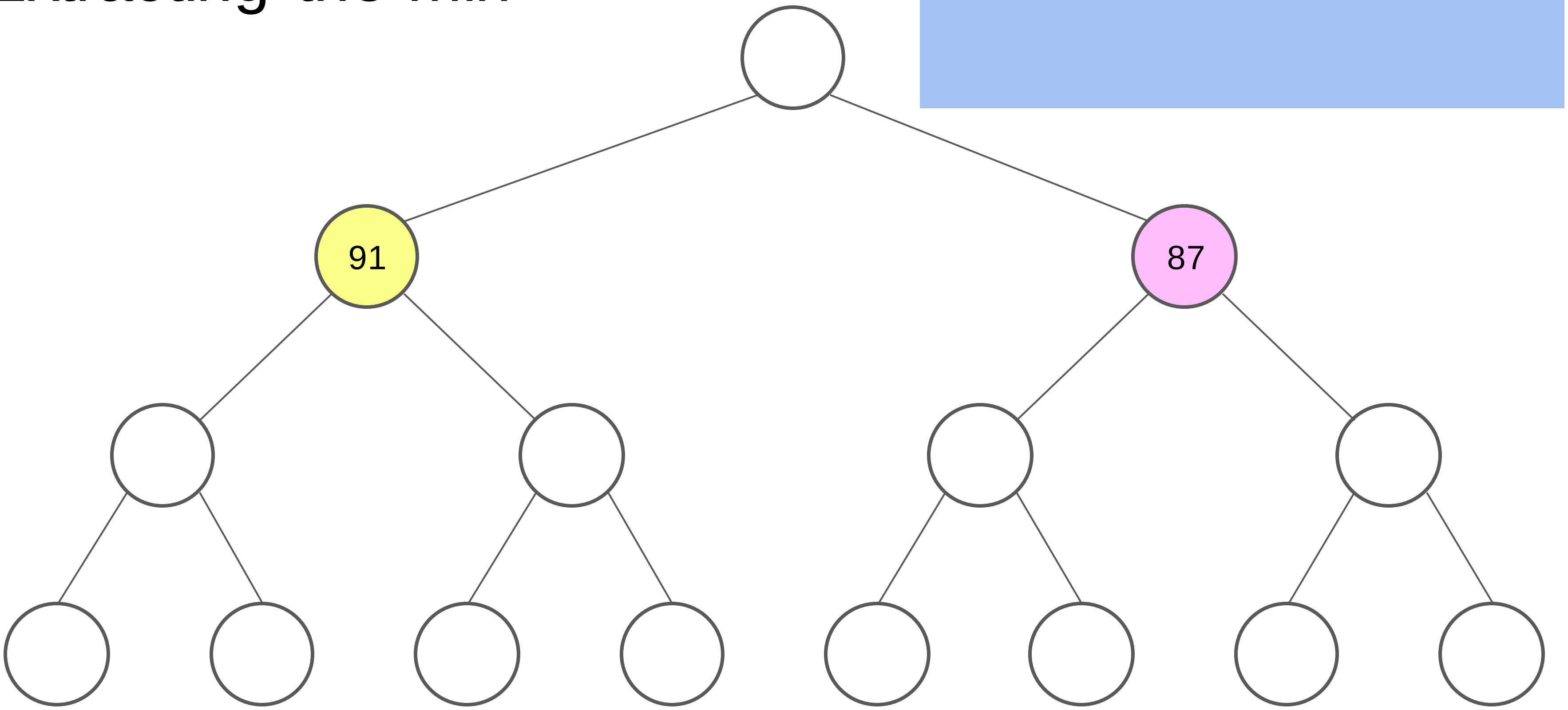
Extracting the min

- And…?



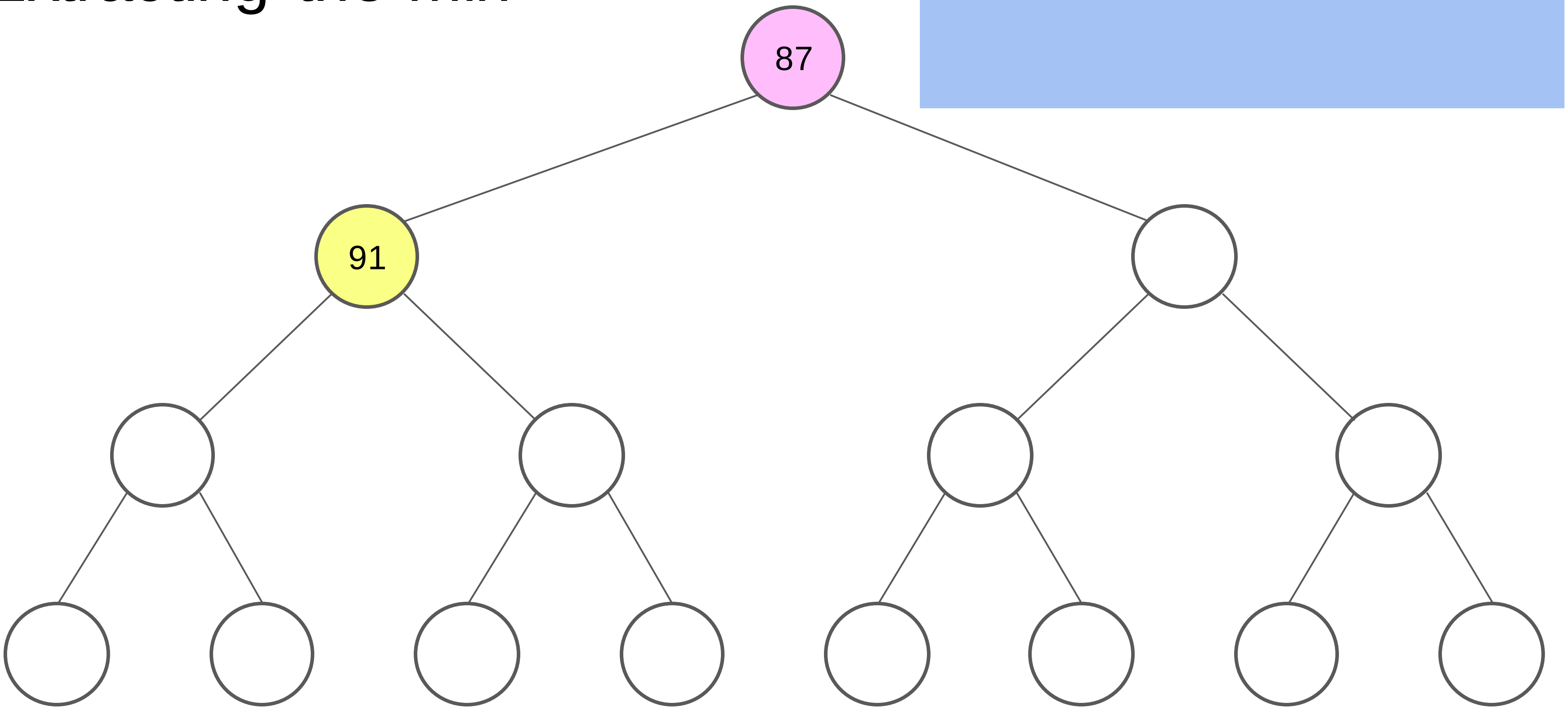
Extracting the min

- And…?



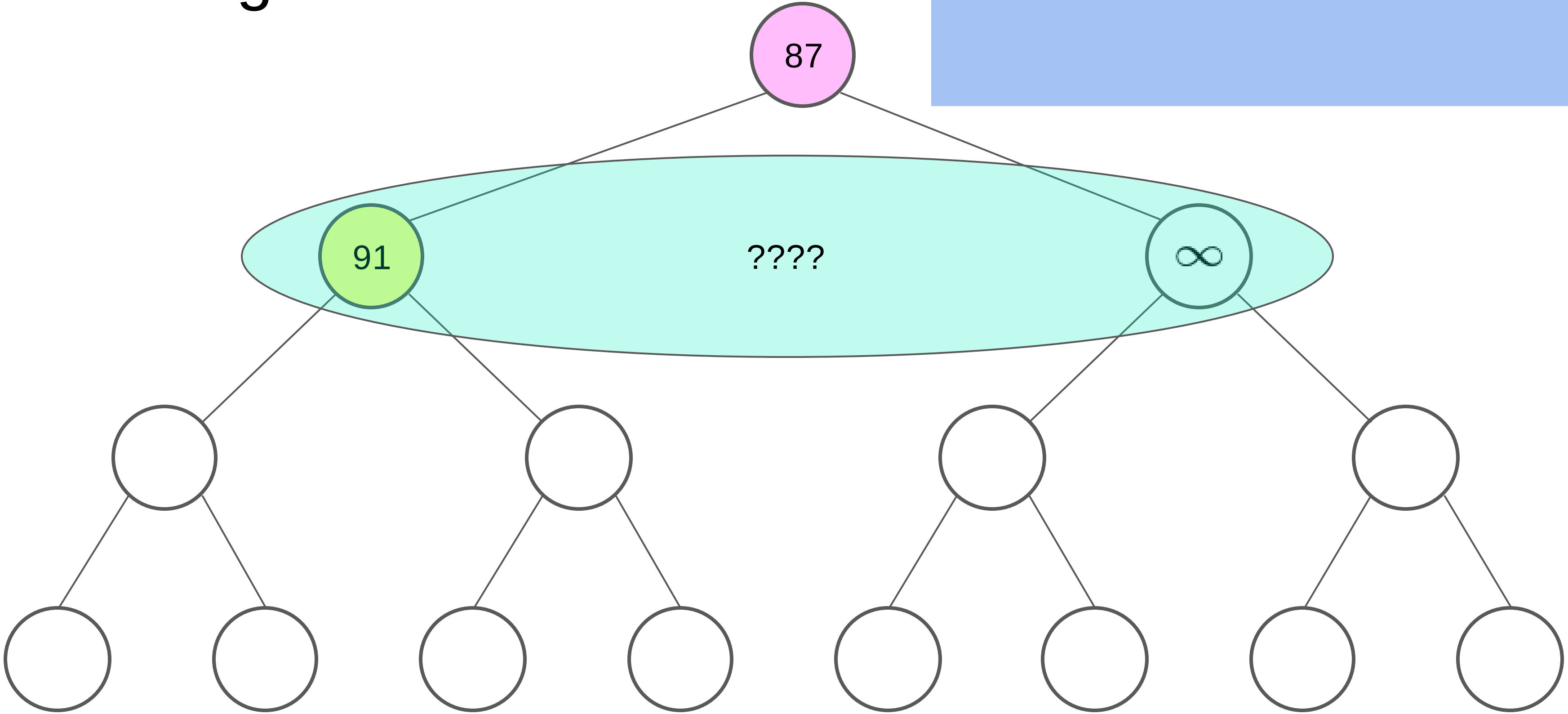
Extracting the min

- And…?



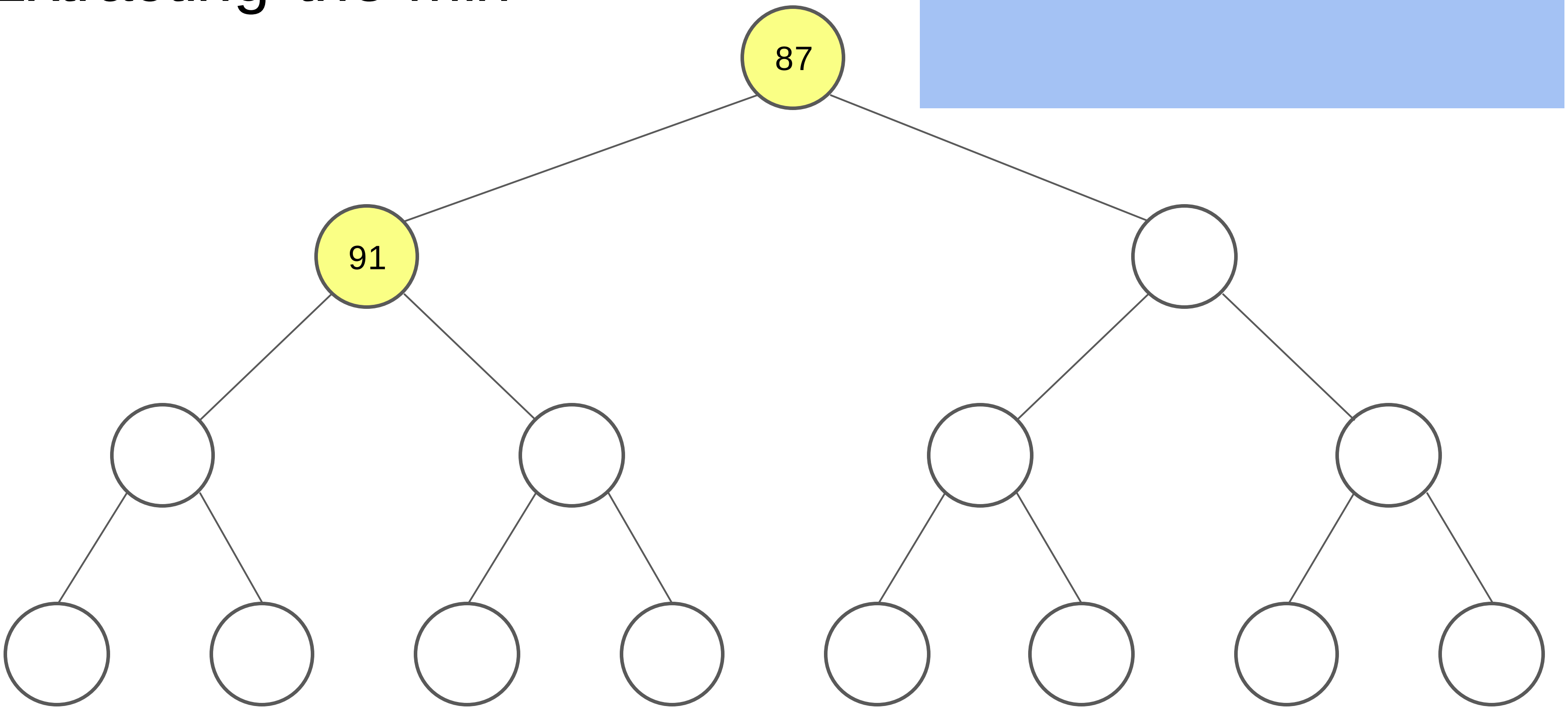
Extracting the min

- And…?



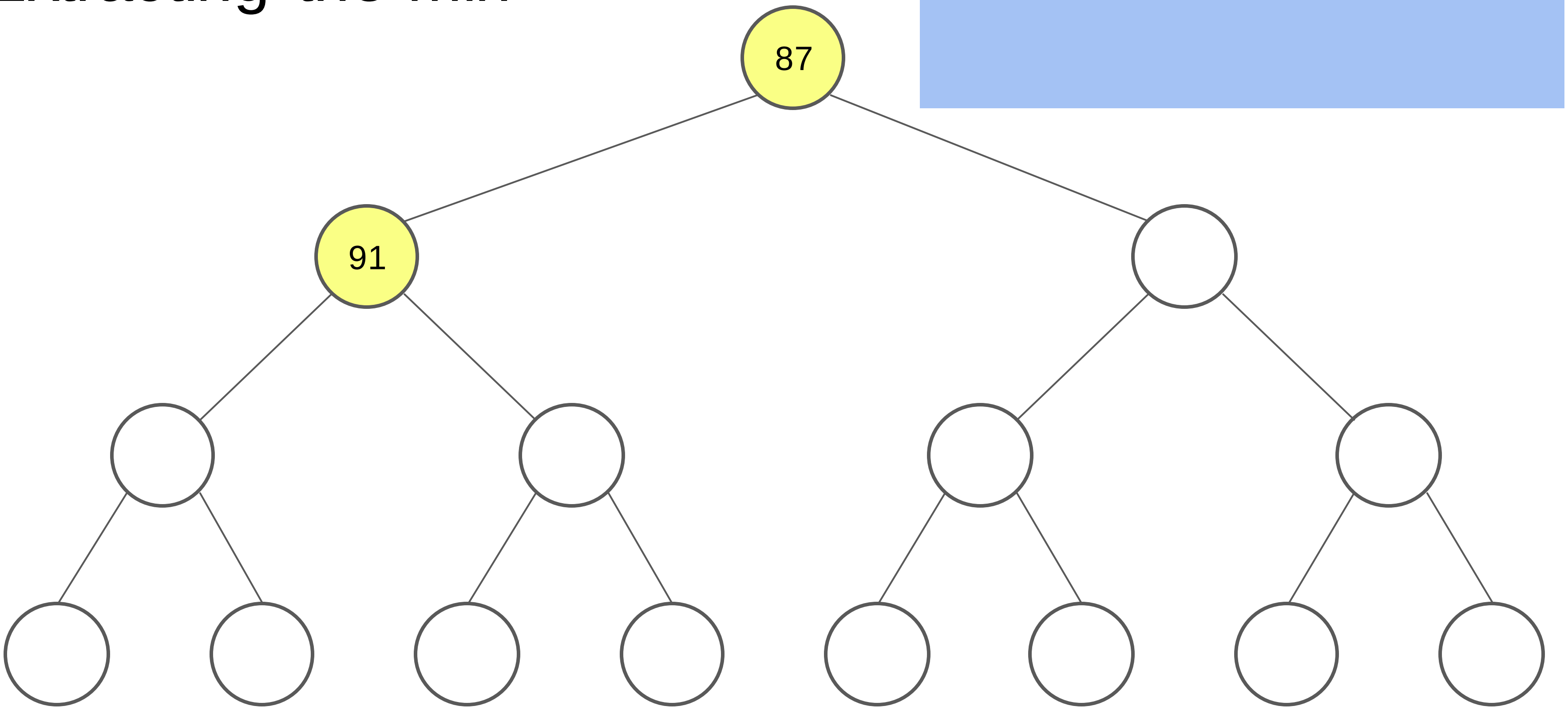
Extracting the min

- And…?



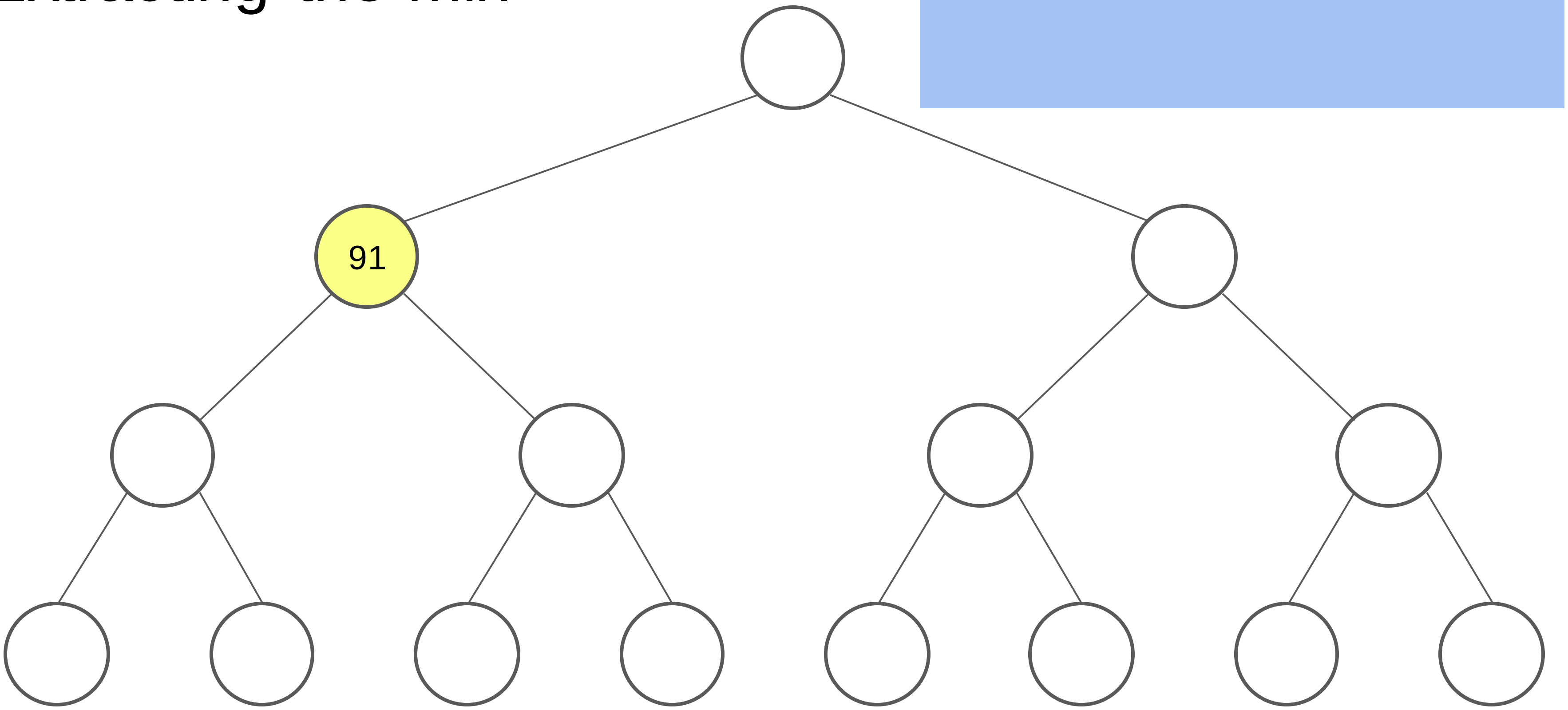
Extracting the min

- 87 is returned next



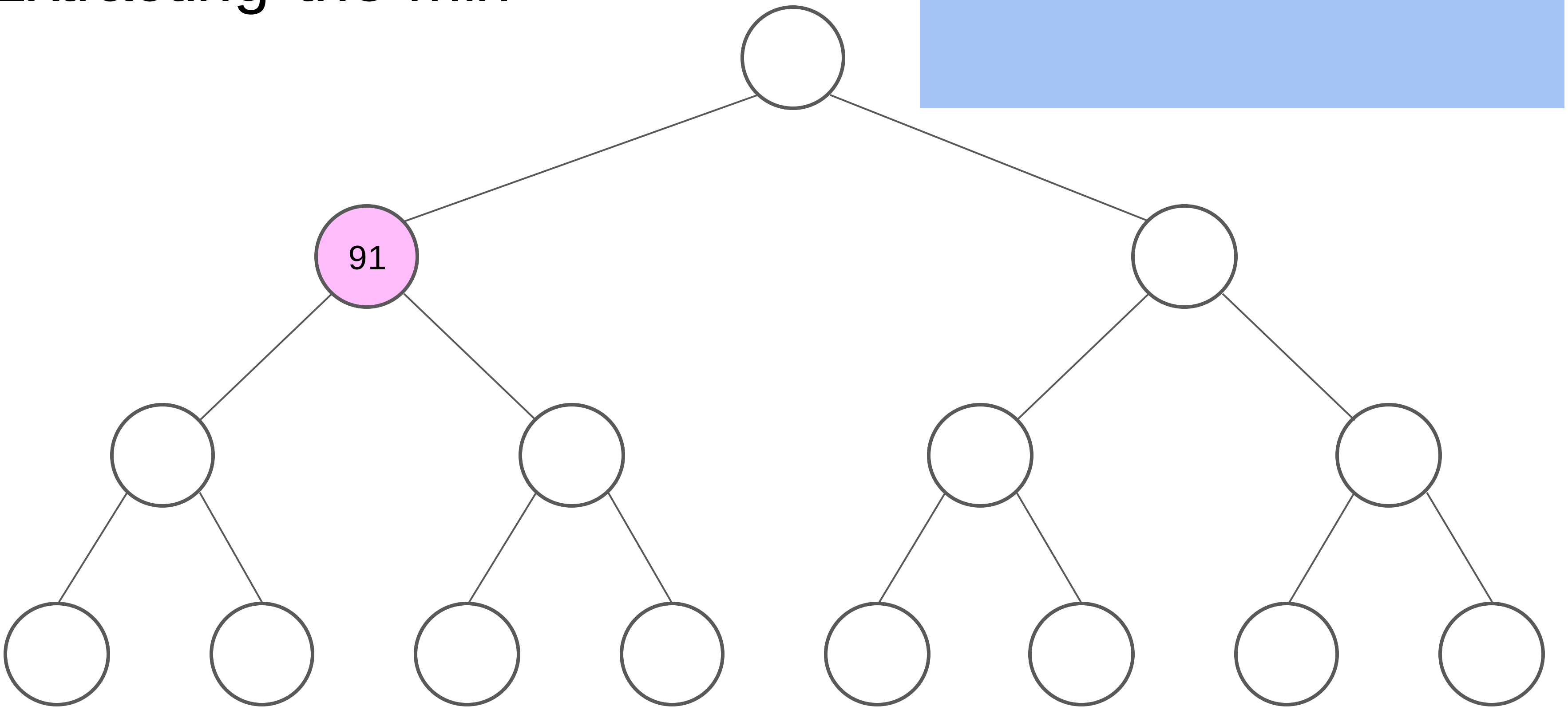
Extracting the min

- You know….



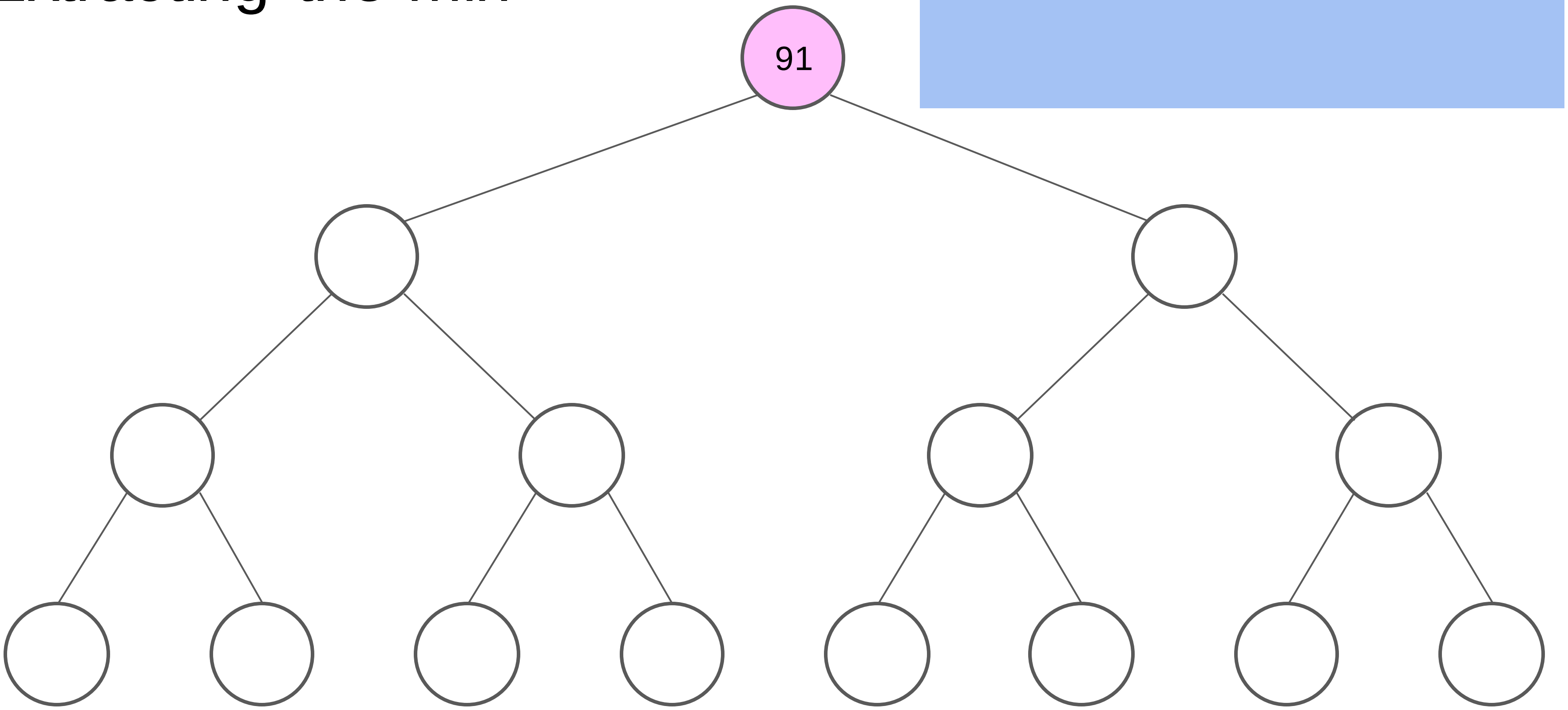
Extracting the min

- You know....



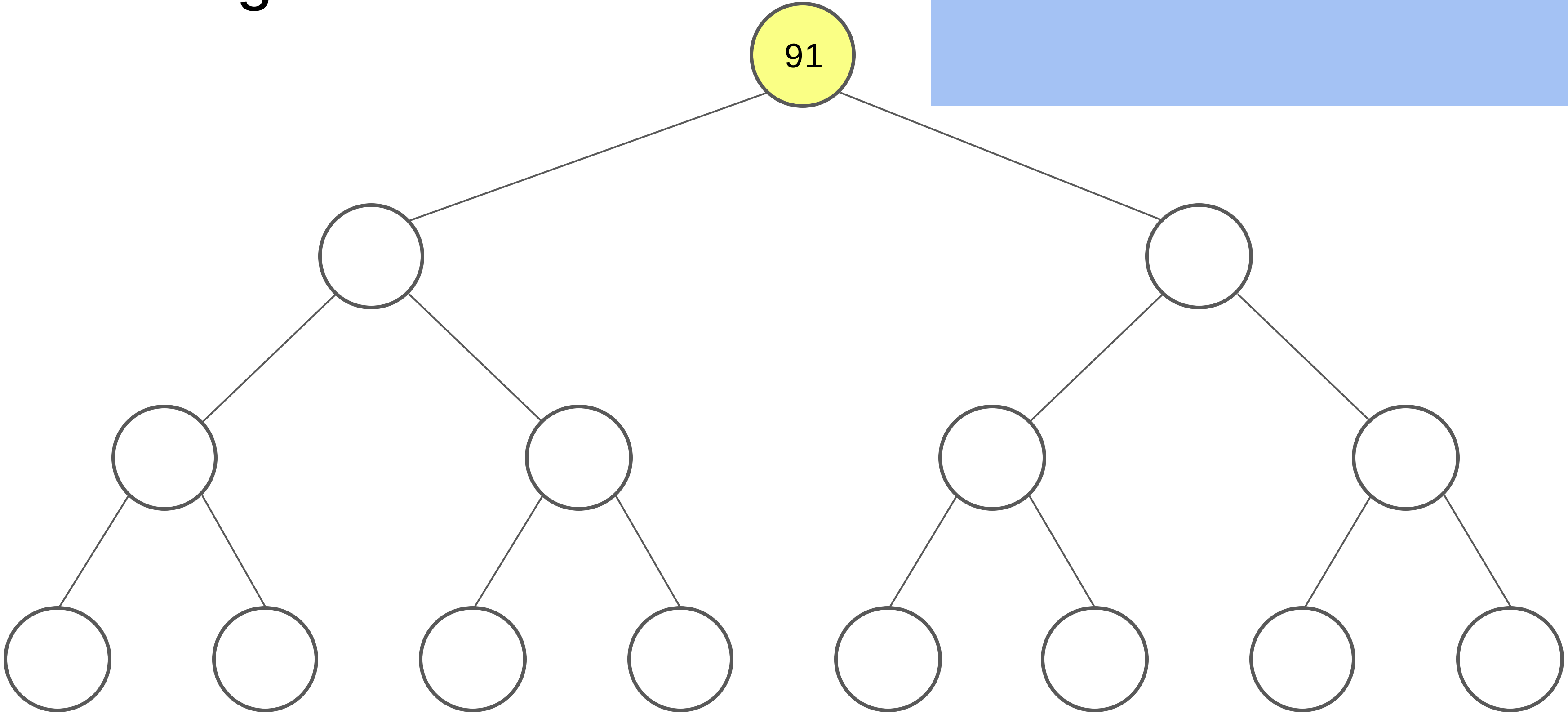
Extracting the min

- You know….



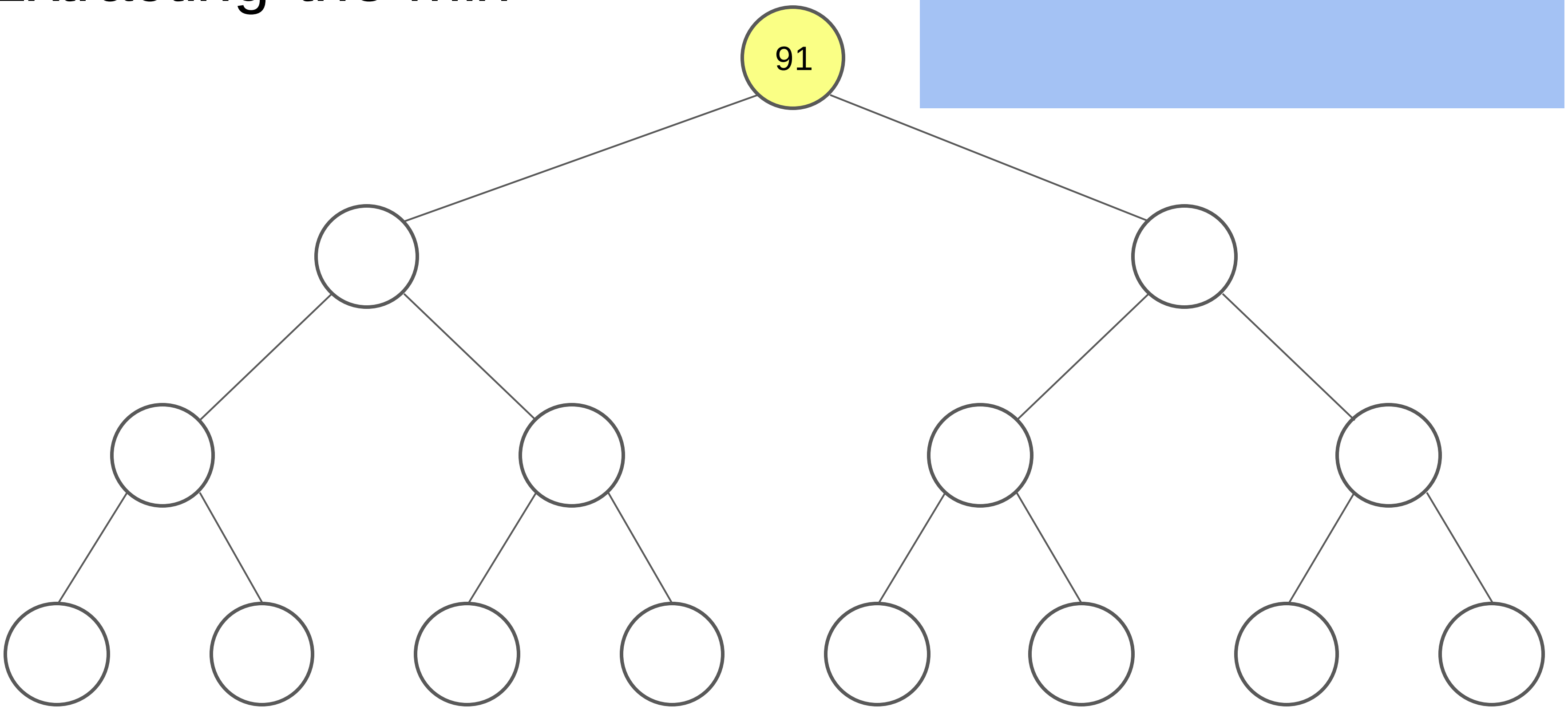
Extracting the min

- You know....



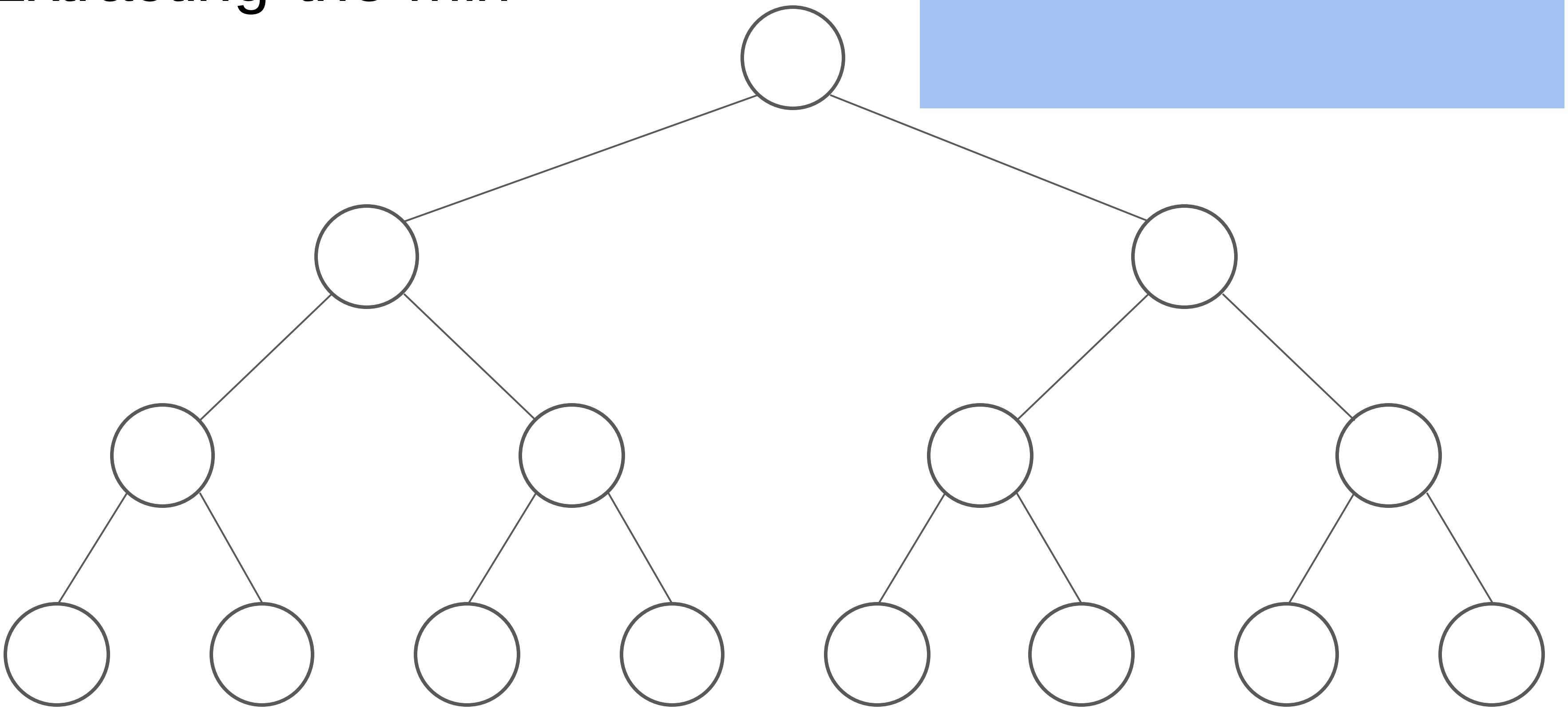
Extracting the min

- Finally 91 is returned



Extracting the min

- Finally 91 is returned
- And the heap is empty



Time For Performance Analysis

- We now have **correct** procedures for the binary heap operations

Time For Performance Analysis

- We now have **correct?** procedures for the binary heap operations
- (Should really write proofs that heap property is restored… later)
- **Right now, we ask: just how fast are these operations?**

Intuition

- We want to give the cost of operations on a heap of size n.

Intuition

- We want to give the cost of operations on a heap of size *n*.
- An insert or decrease might move a value from the bottom of the tree up to the root.
- An extractMin might move a value (the new root) from the root of the tree down to the bottom.

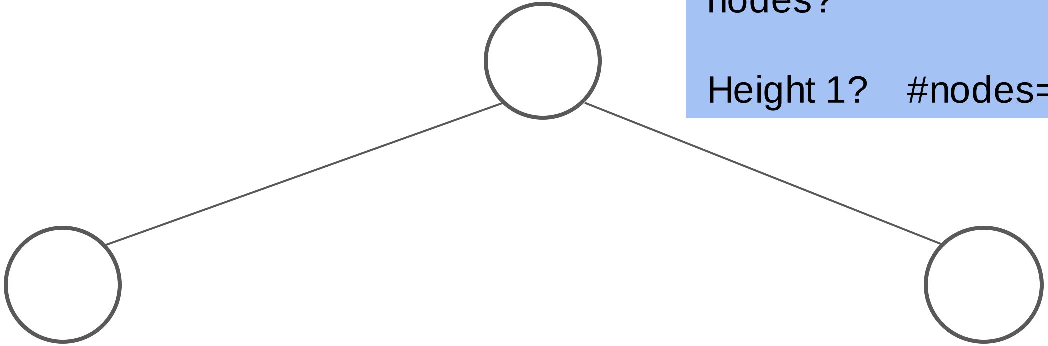
Intuition

- We want to give the cost of operations on a heap of size *n*.
- An insert or decrease might move a value from the bottom of the tree up to the root.
- An extractMin might move a value (the new root) from the root of the tree down to the bottom.
- So we need to reason about how tall a heap with n elements is.

Height vs # Nodes

For a **complete** binary tree, how does the height of tree affect the number of nodes?

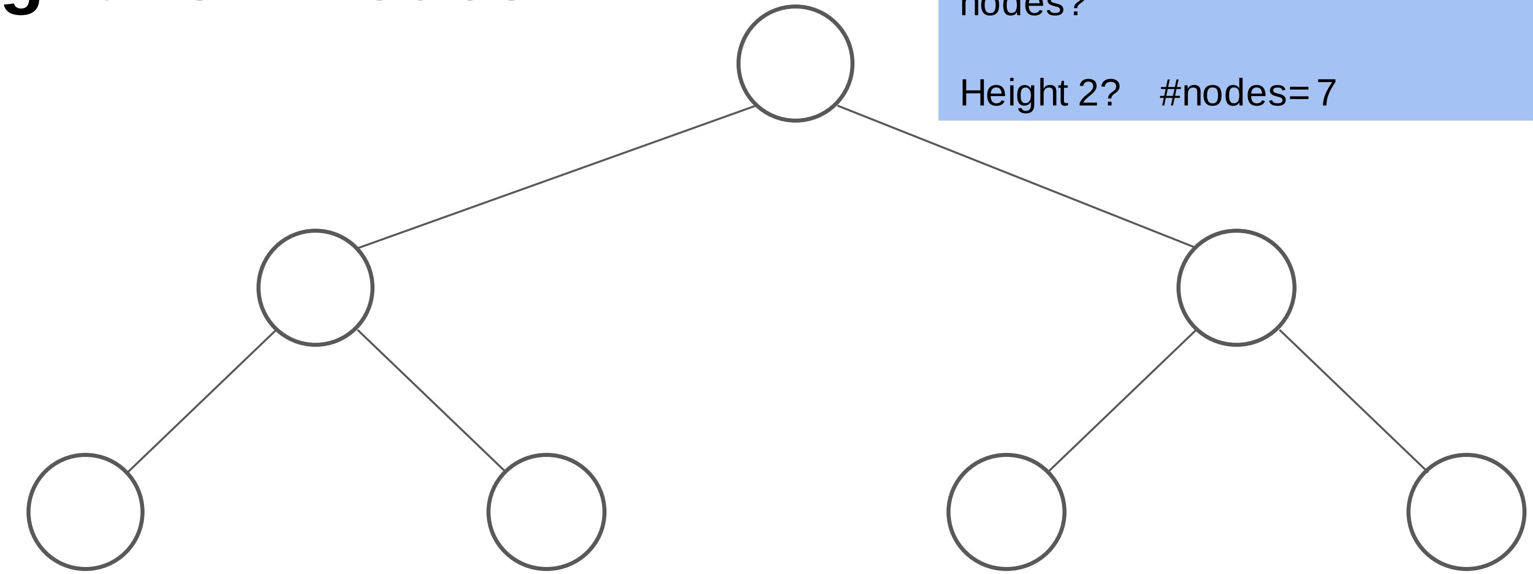
Height 1?   #nodes= 3



Height vs # Nodes

For a complete binary tree, how does the height of tree affect the number of nodes?

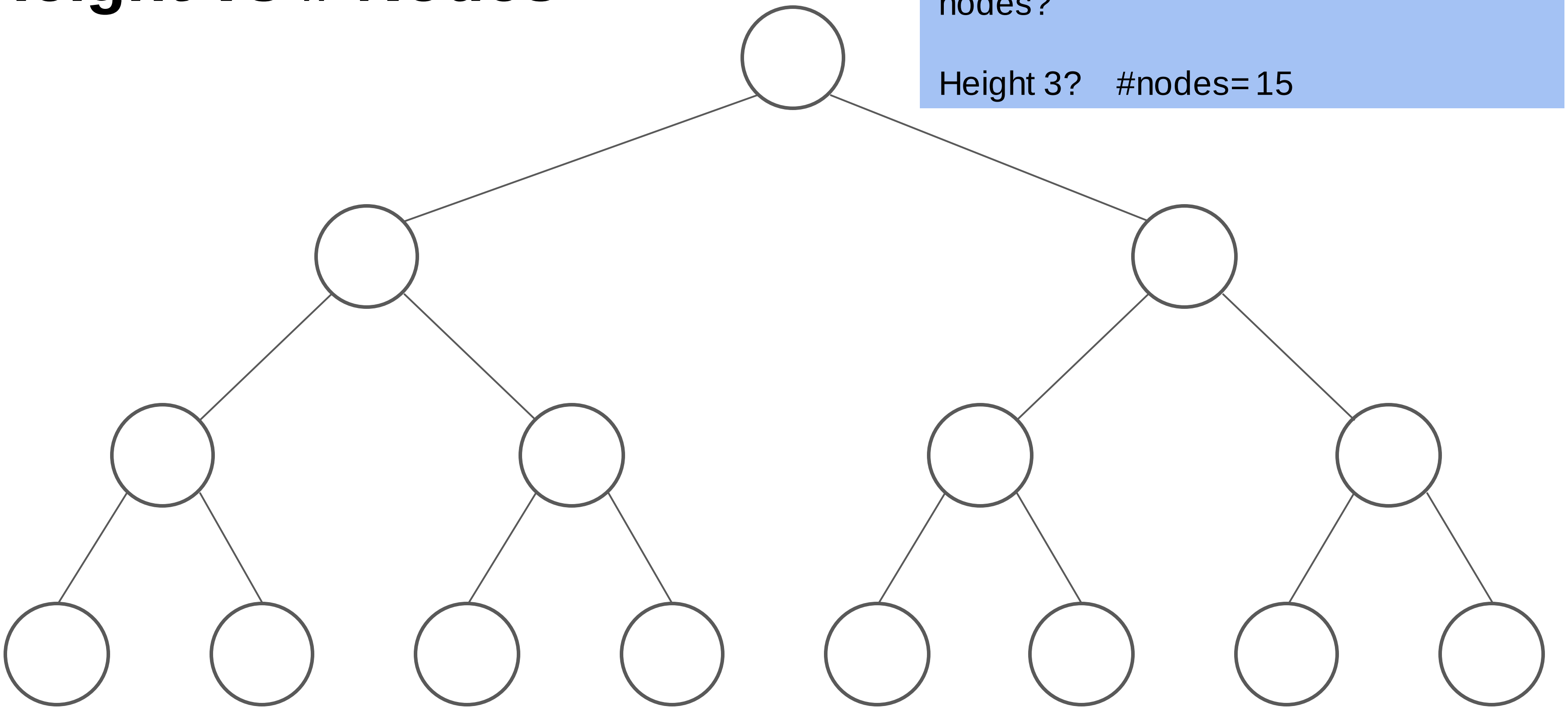
Height 2?   #nodes= 7



Height vs # Nodes

For a complete binary tree, how does the height of tree affect the number of nodes?

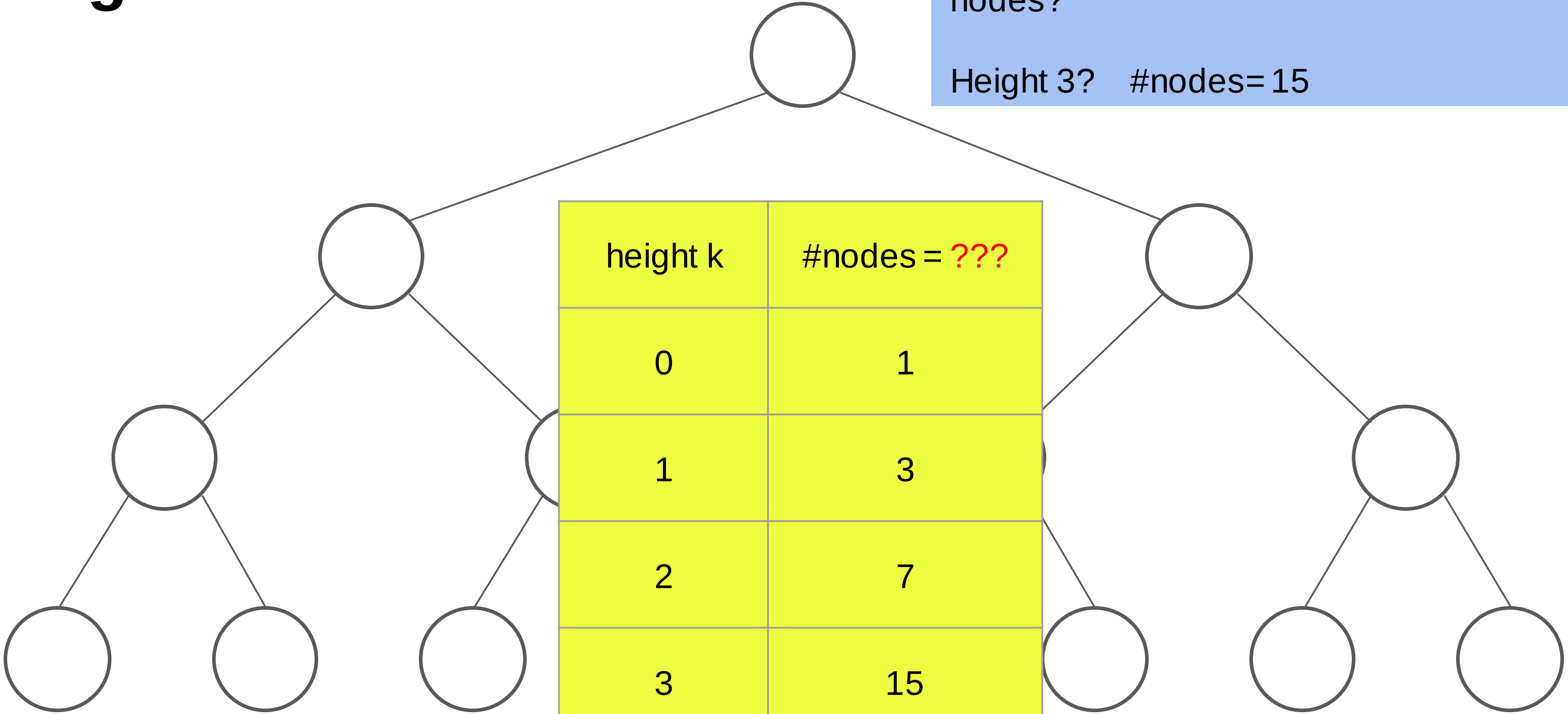
Height 3? #nodes= 15



Height vs # Nodes

For a complete binary tree, how does the height of tree affect the number of nodes?

Height 3?   #nodes= 15

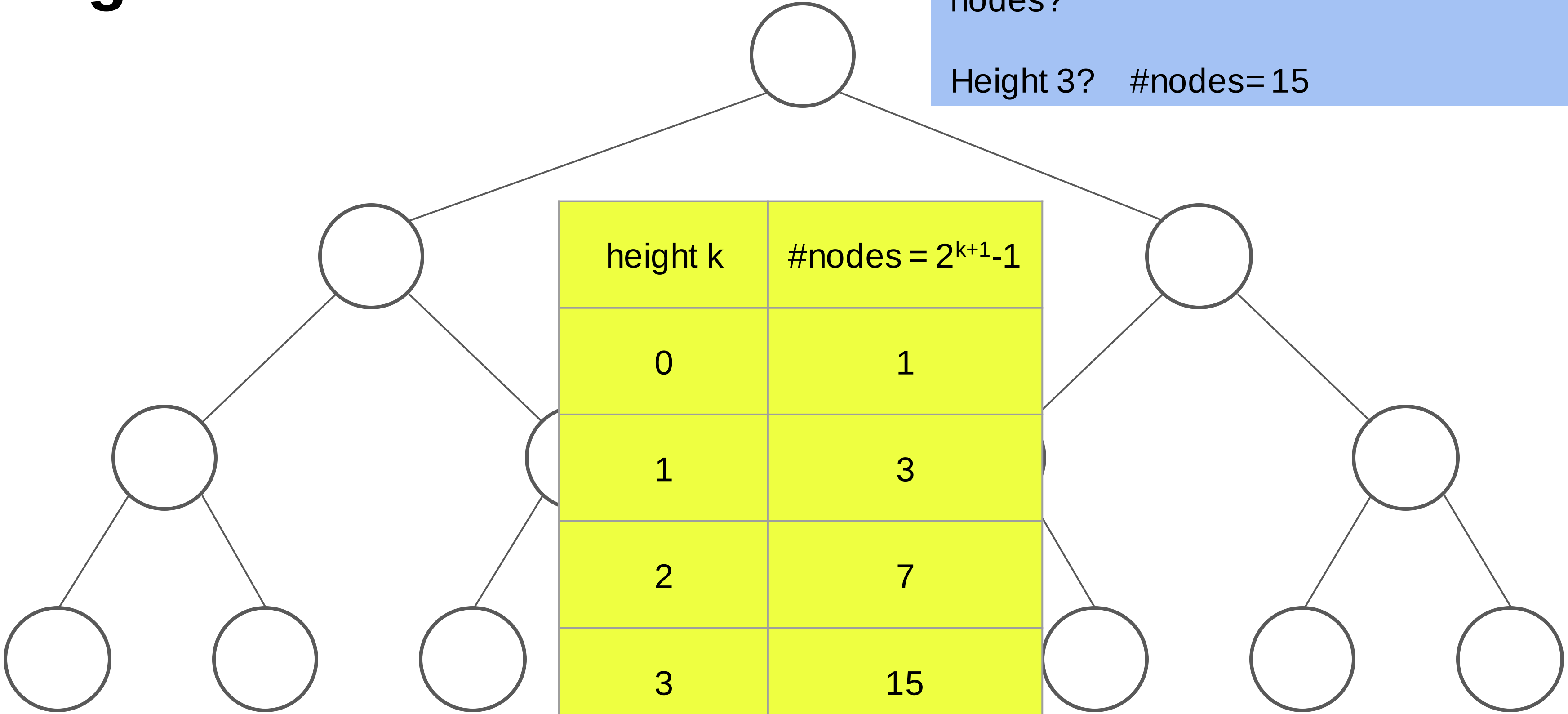


height k	#nodes = ???
0	1
1	3
2	7
3	15

Height vs # Nodes

For a complete binary tree, how does the height of tree affect the number of nodes?

Height 3?   #nodes= 15



height k	#nodes = 2^{k+1}-1
0	1
1	3
2	7
3	15

Theorem

- A complete binary tree (all non-leaves have two children) of height k has $2^{k+1} - 1$ nodes.

But First…

- **Lemma: a complete binary tree of height k has 2^k leaves.**

But First…

- Lemma: a complete binary tree of height k has 2^k leaves.
- Pf: By induction on k.

But First…

- Lemma: a complete binary tree of height k has 2^k leaves.
- Pf: By induction on k.
- Base: $k = 0 \rightarrow$ tree is a single node $\rightarrow 2^0 = 1$ leaf.

But First…

- Lemma: a complete binary tree of height k has 2^k leaves.
- Pf: By induction on k.
- Base: $k = 0 \rightarrow$ tree is a single node $\rightarrow 2^0 = 1$ leaf.
- Ind: Suppose true for tree T of height k.

But First…

- Lemma: a complete binary tree of height k has 2^k leaves.
- Pf: By induction on k.
- Base: $k = 0 \rightarrow$ tree is a single node $\rightarrow 2^0 = 1$ leaf.
- Ind: Suppose true for tree T of height k.
- We extend T by one level, adding two leaves below each node at the bottom of T. By IH, T has 2^k leaves, so extension has 2^{k+1}.

QED

Back To Theorem…

- **Thm: a complete binary tree of height k has $2^{k+1} - 1$ nodes.**

- **Pf: By induction on k.**

Back To Theorem…

- Thm: a complete binary tree of height k has $2^{k+1} – 1$ nodes.

- Pf: By induction on k.

- Bas: $k = 0 \rightarrow$ single node $\rightarrow 2^{0+1} – 1 = 1$ nodes.

Back To Theorem…

- **Thm: a complete binary tree of height k has $2^{k+1} – 1$ nodes.**
- **Pf: By induction on k.**
- **Bas: $k = 0$ → single node → $2^{0+1} – 1 = 1$ nodes.**
- **Ind: Suppose true for tree T of height k.**

Back To Theorem…

- Thm: a complete binary tree of height k has $2^{k+1} - 1$ nodes.
- Pf: By induction on k.
- Bas: $k = 0 \rightarrow$ single node $\rightarrow 2^{0+1} - 1 = 1$ nodes.
- Ind: Suppose true for tree T of height k.
- By IH, T has $2^{k+1} - 1$ nodes. Adding k+1st level adds 2^{k+1} leaves.

Back To Theorem…

- Thm: a complete binary tree of height k has $2^{k+1} – 1$ nodes.
- Pf: By induction on k.
- Bas: $k = 0 \rightarrow$ single node $\rightarrow 2^{0+1} – 1 = 1$ nodes.
- Ind: Suppose true for tree T of height k.

by Lemma

- By IH, T has $2^{k+1} – 1$ nodes. Adding k+1st level adds 2^{k+1} leaves.

Back To Theorem…

- Thm: a complete binary tree of height k has $2^{k+1} – 1$ nodes.
- Pf: By induction on k.
- Bas: $k = 0 \rightarrow$ single node $\rightarrow 2^{0+1} – 1 = 1$ nodes.
- Ind: Suppose true for tree T of height k.

by Lemma

- By IH, T has $2^{k+1} – 1$ nodes. Adding k+1st level adds 2^{k+1} leaves.
- Extended tree has $2(2^{k+1}) – 1 = 2^{k+2} – 1$ nodes. QED

So What?

- Complete binary tree of height k has **$\Theta(2^k)$** nodes.
- Hence, complete binary tree with n nodes has height **$\Theta(\log n)$.**

So What?

- Complete binary tree of height k has $\Theta(2^k)$ nodes.
- Hence, complete binary tree with n nodes has height $\Theta(\log n)$.
- **What about compact but not complete trees?**

So What?

- Complete binary tree of height k has $\Theta(2^k)$ nodes.
- Hence, complete binary tree with n nodes has height $\Theta(\log n)$.
- **What about compact but not complete trees?**
- All levels except the bottom are full → can show tree of height k has *at least* 2^k *nodes*.
- Conclude that a **compact** tree with n nodes still has height $\Theta(\log n)$.

Conclusions About Running Time

- decrease/insert/heapify may move an element from the bottom to top or top to bottom of a compact tree – $\Theta(\log n)$ levels.

- Time to move is $O(1)$ per level of tree.

- Conclude that these operations take *worst-case* time $\Theta(\log n)$.

Another Way to Analyze Complexity

- Heapify is often written as a **recursive procedure**.

Another Way to Analyze Complexity

- Heapify is often written as a **recursive procedure**.

Heapify(*tree rooted at v*)
    if (v is bigger than its smallest child c)
        swap values of nodes v and c
        Heapify(*tree rooted at c*)

Another Way to Analyze Complexity

- Heapify is often written as a **recursive procedure**.

> **Heapify**(*tree rooted at v*)
>   if (v is bigger than its smallest child c)
>   swap values of nodes v and c
>   **Heapify**(*tree rooted at c*)

- **How can we analyze complexity of code like this?**

Basic Approach

- Suppose a recursive procedure runs in time $T(n)$ on inputs of size n.
- Procedure does work $f(n)$, plus a recursive call on input of size $g(n) < n$.
- Then we can write $\mathbf{T(n) = T(g(n)) + f(n)}$

Basic Approach

- Suppose a recursive procedure runs in tim T(n) n inputs of size n.

recurrence
for T(n)

- Procedure does work f(n), plus t of size $g(n) < n$.

- Then we can write **$T(n) = T(g(n)) + f(n)$**

Basic Approach

- Suppose a recursive procedure runs in time T(n) on inputs of size n.

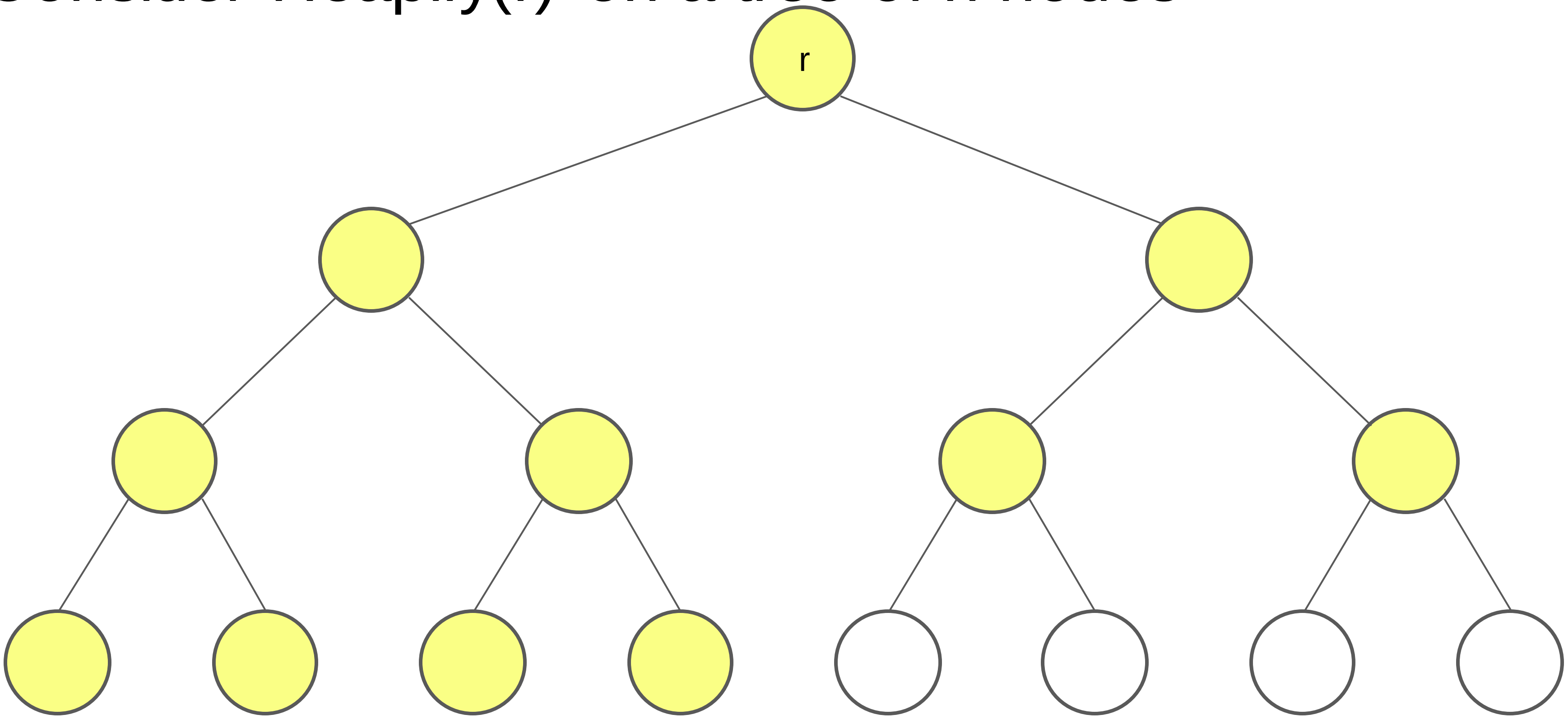
- Procedure does work f(n), plus ... of size $g(n) < n$.

recurrence for T(n)

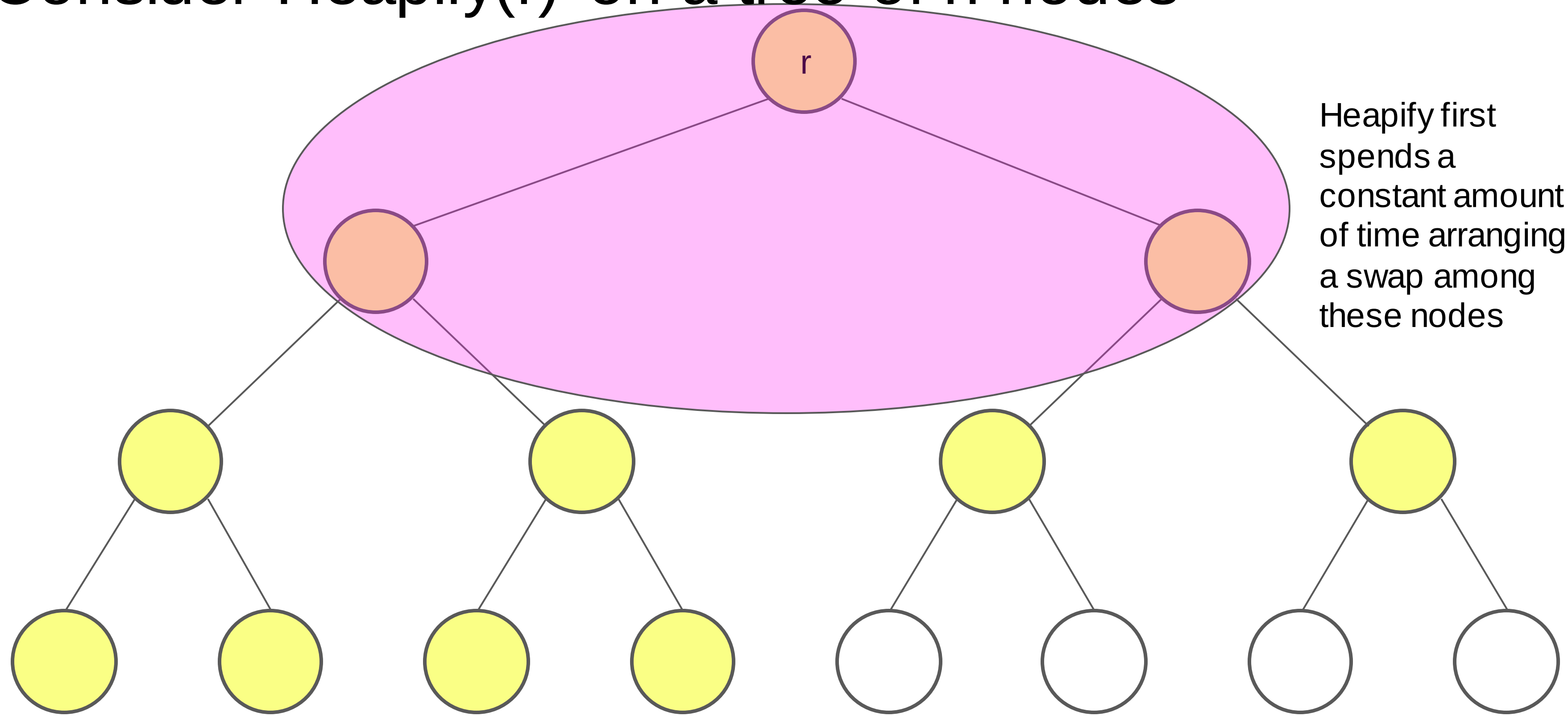
- Then we can write **$T(n) = T(g(n)) + f(n)$**

Let's apply this approach to the analysis of heapify

Consider Heapify(r) on a tree of n nodes



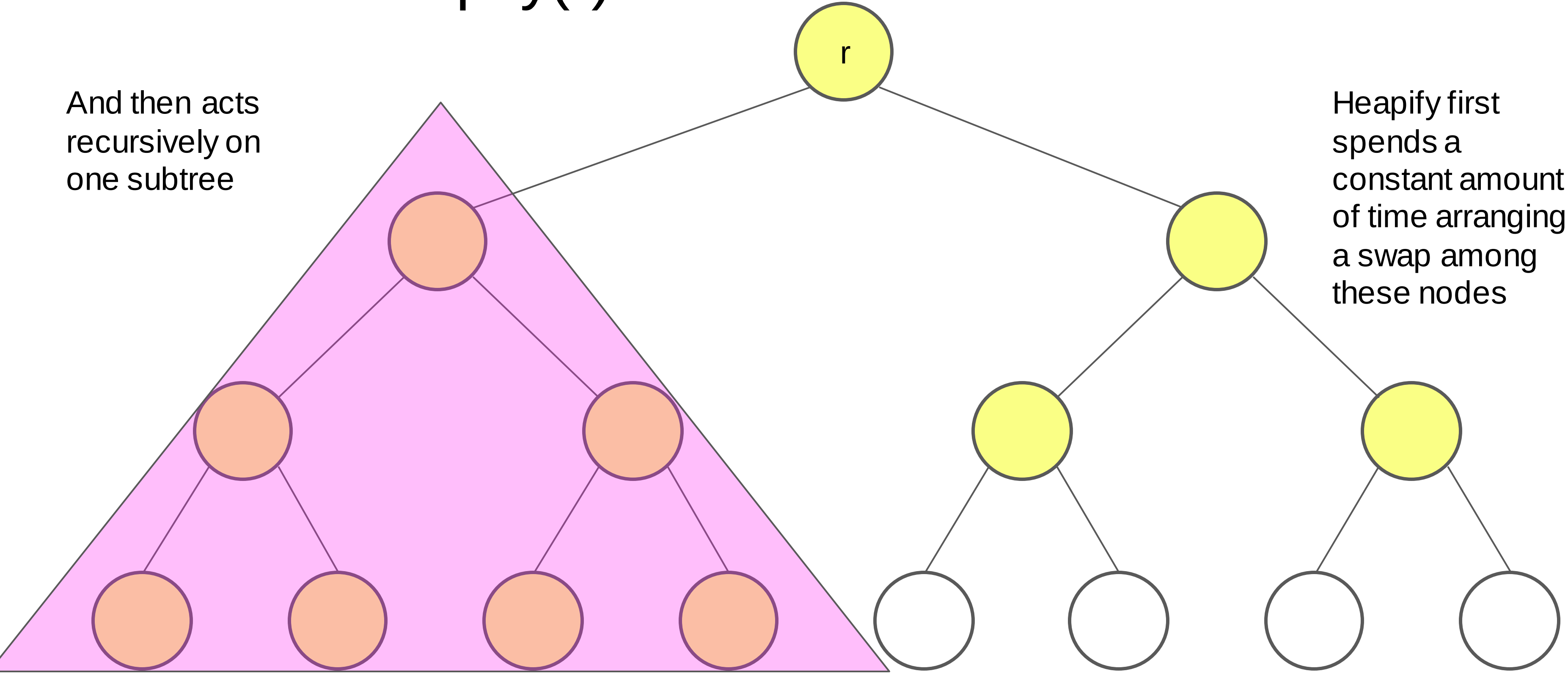
Consider Heapify(r) on a tree of n nodes



Heapify first spends a constant amount of time arranging a swap among these nodes

Consider Heapify(r) on a tree of n nodes

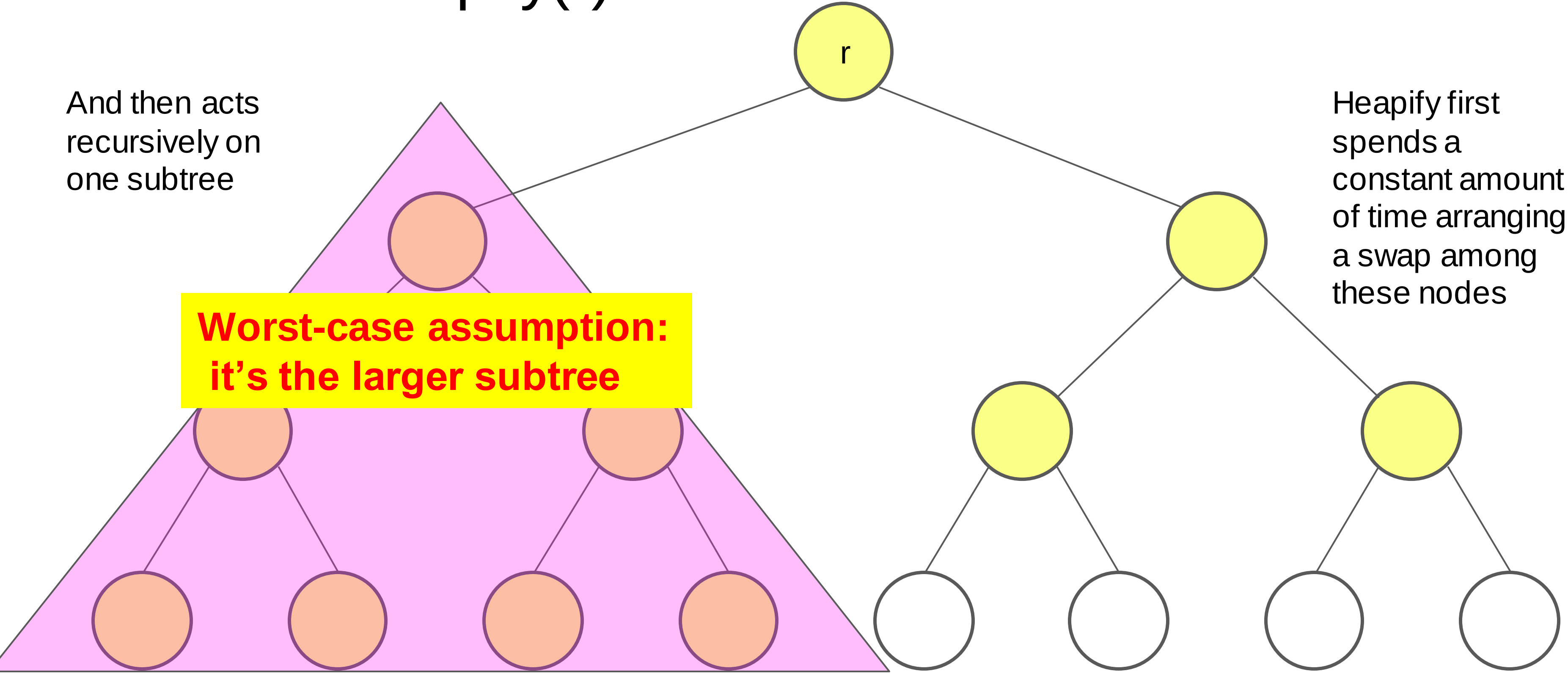
And then acts recursively on one subtree



Heapify first spends a constant amount of time arranging a swap among these nodes

Consider Heapify(r) on a tree of n nodes

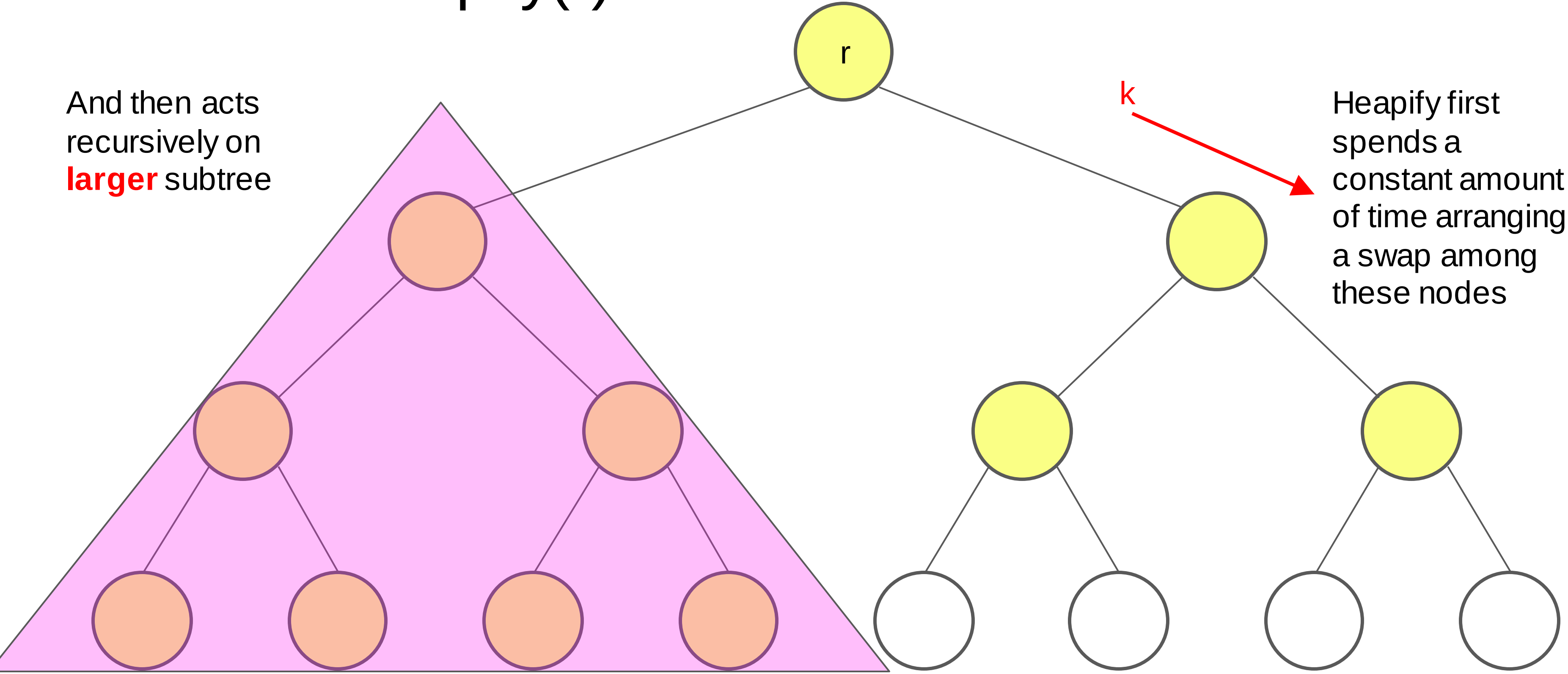
And then acts recursively on one subtree



Heapify first spends a constant amount of time arranging a swap among these nodes

Consider Heapify(r) on a tree of n nodes

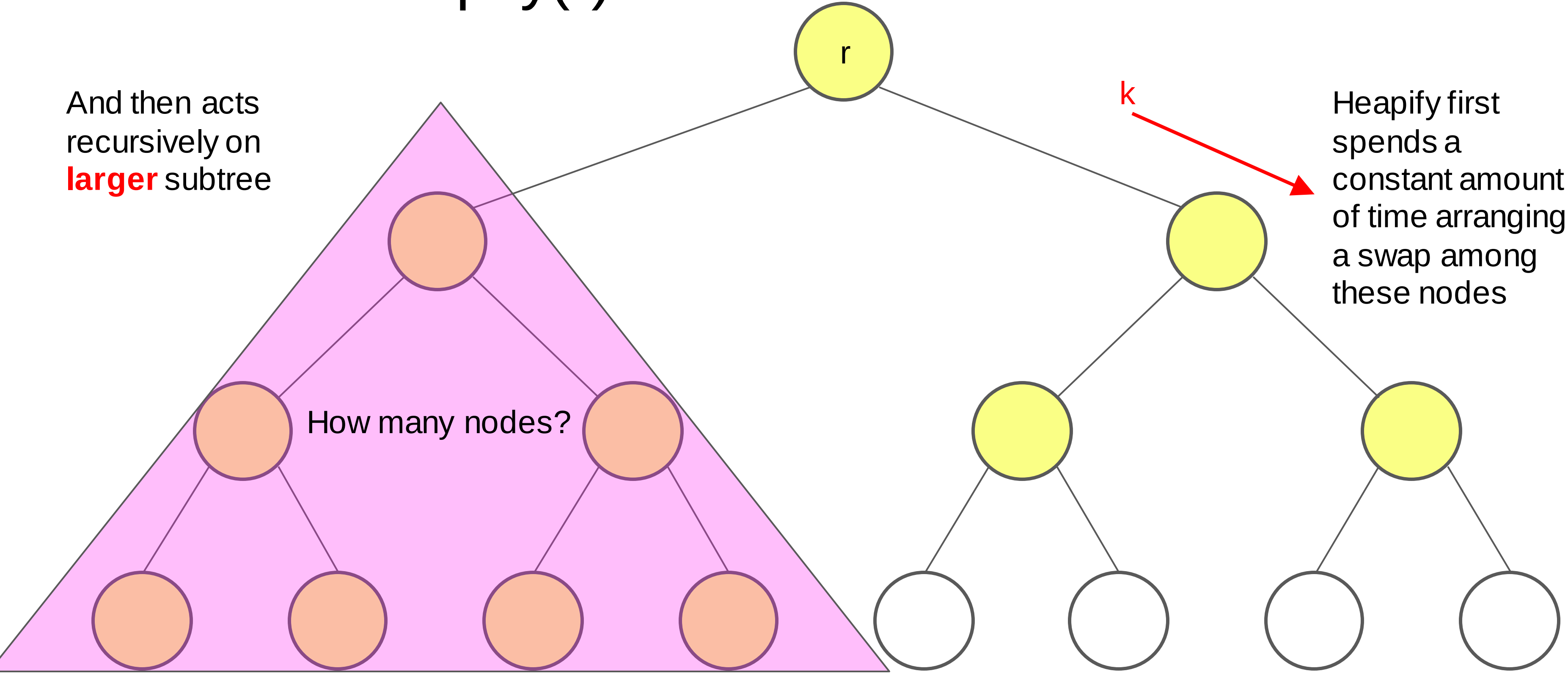
And then acts recursively on **larger** subtree



Heapify first spends a constant amount of time arranging a swap among these nodes

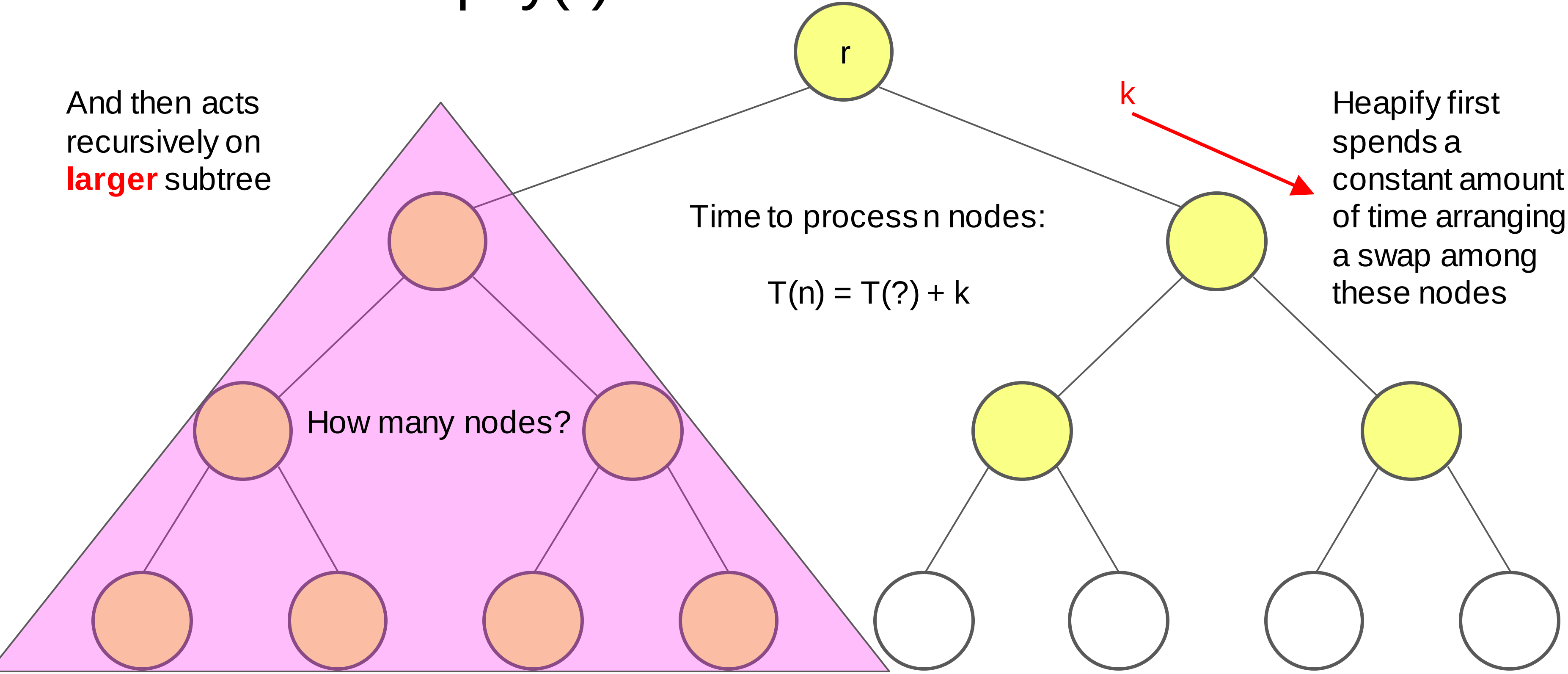
Consider Heapify(r) on a tree of n nodes

And then acts recursively on **larger** subtree



Heapify first spends a constant amount of time arranging a swap among these nodes

Consider Heapify(r) on a tree of n nodes



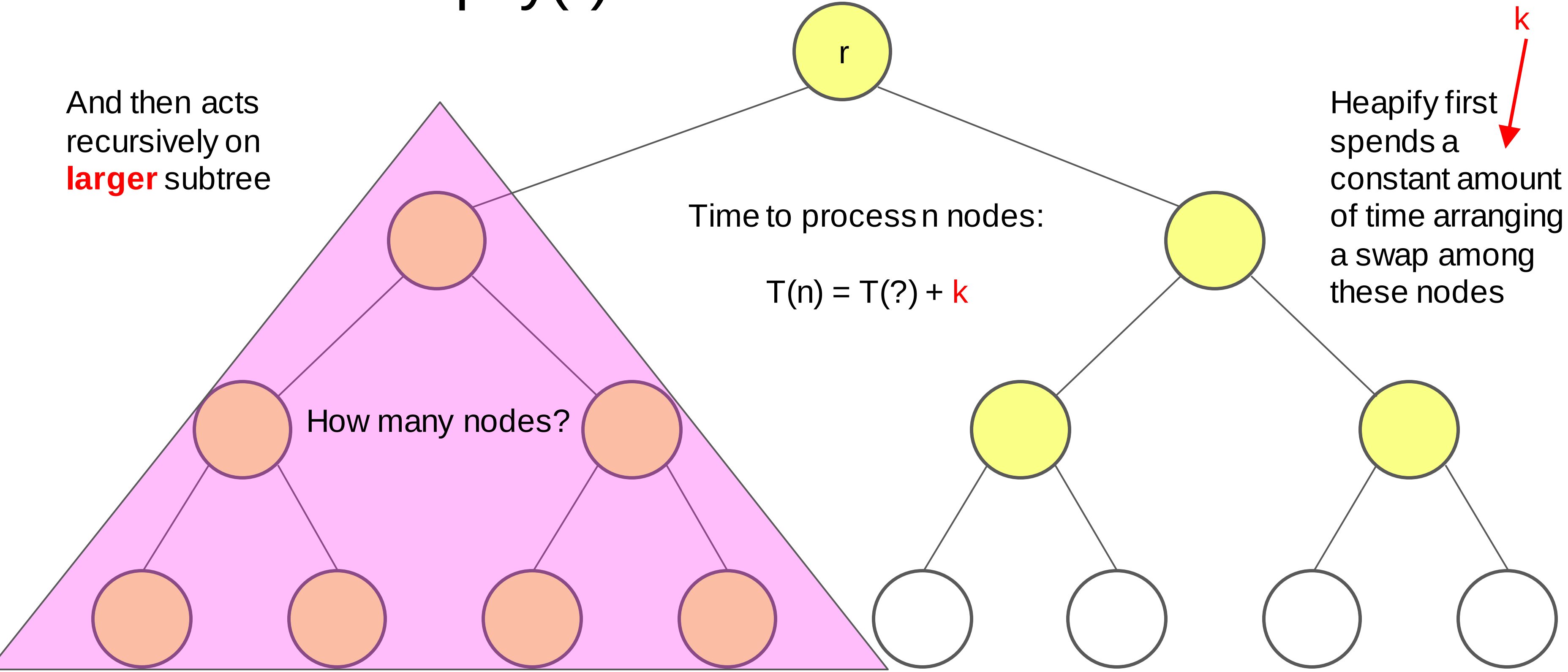
And then acts recursively on **larger** subtree

Time to process n nodes:

T(n) = T(?) + k

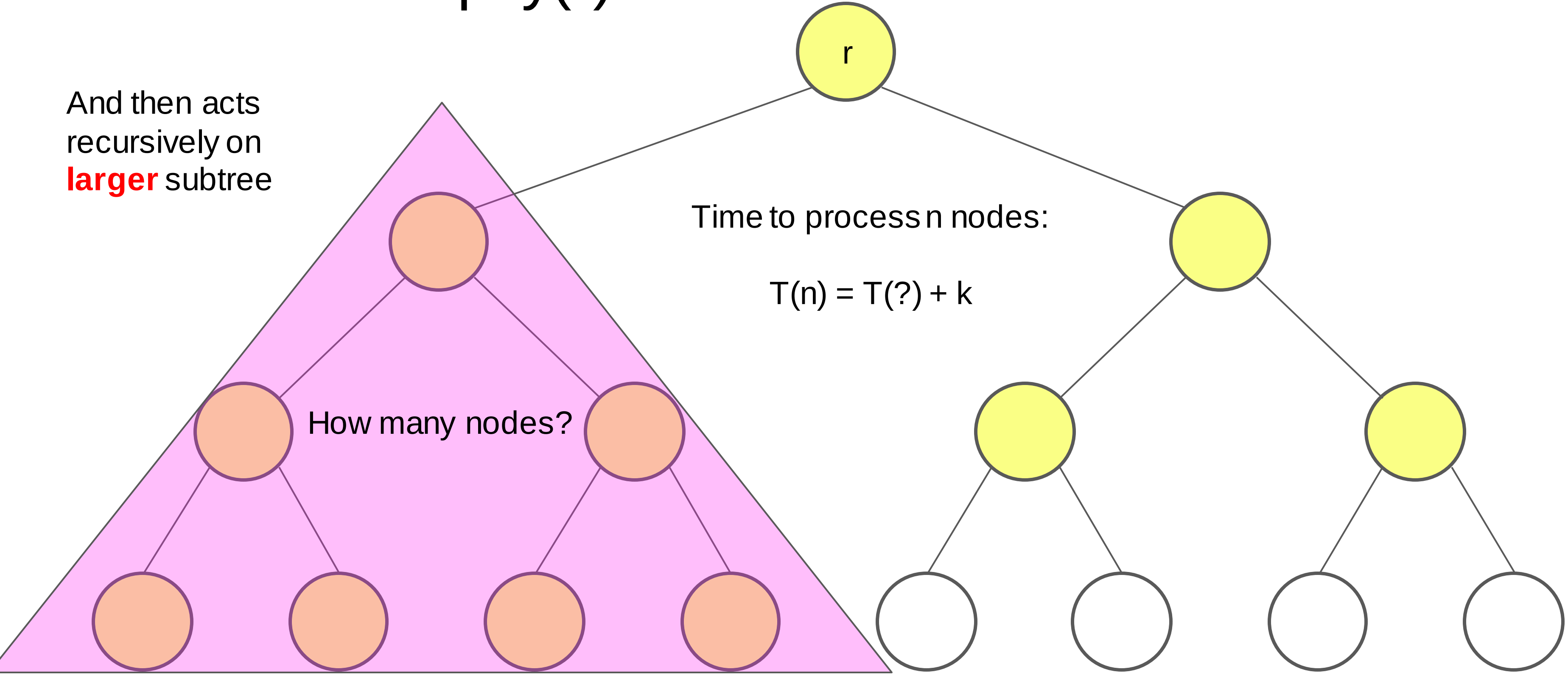
Heapify first spends a constant amount of time arranging a swap among these nodes

Consider Heapify(r) on a tree of n nodes



Consider Heapify(r) on a tree of n nodes

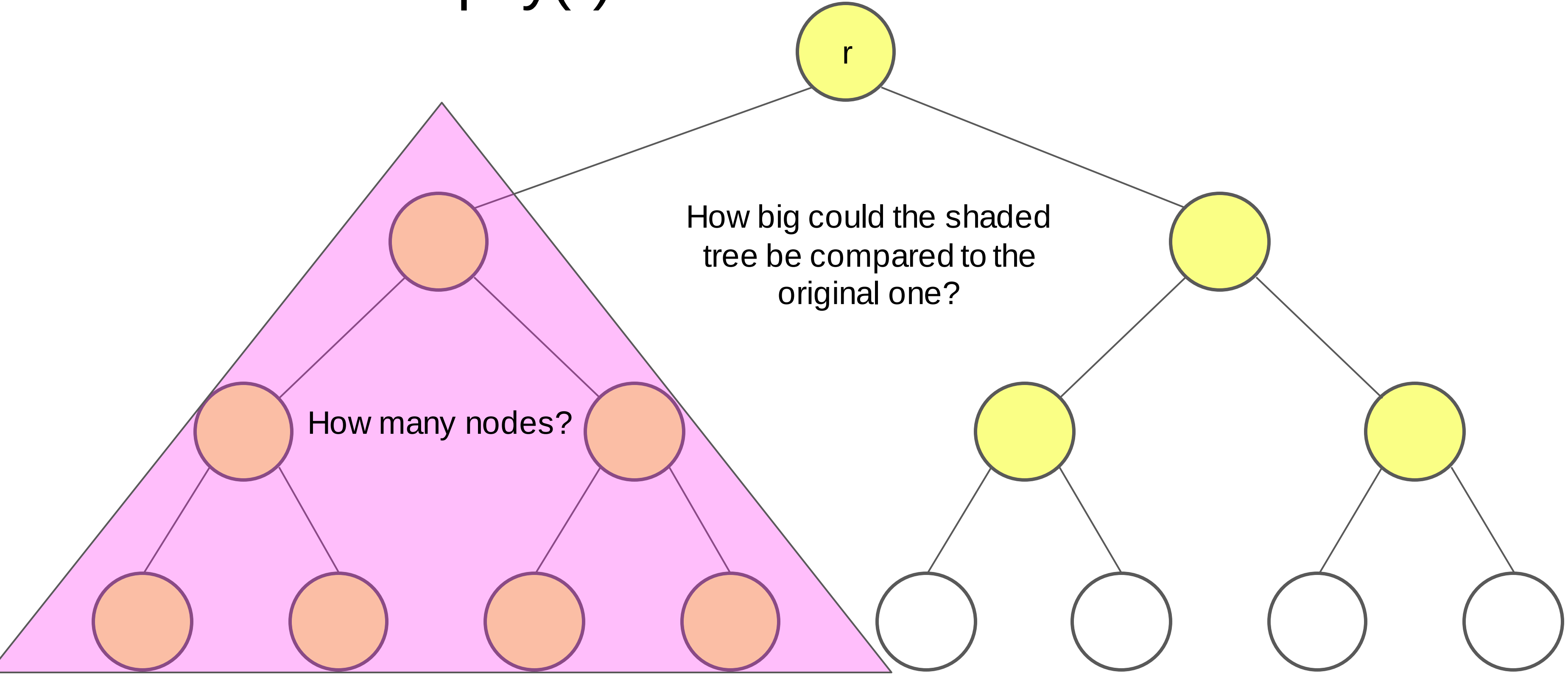
And then acts recursively on **larger** subtree



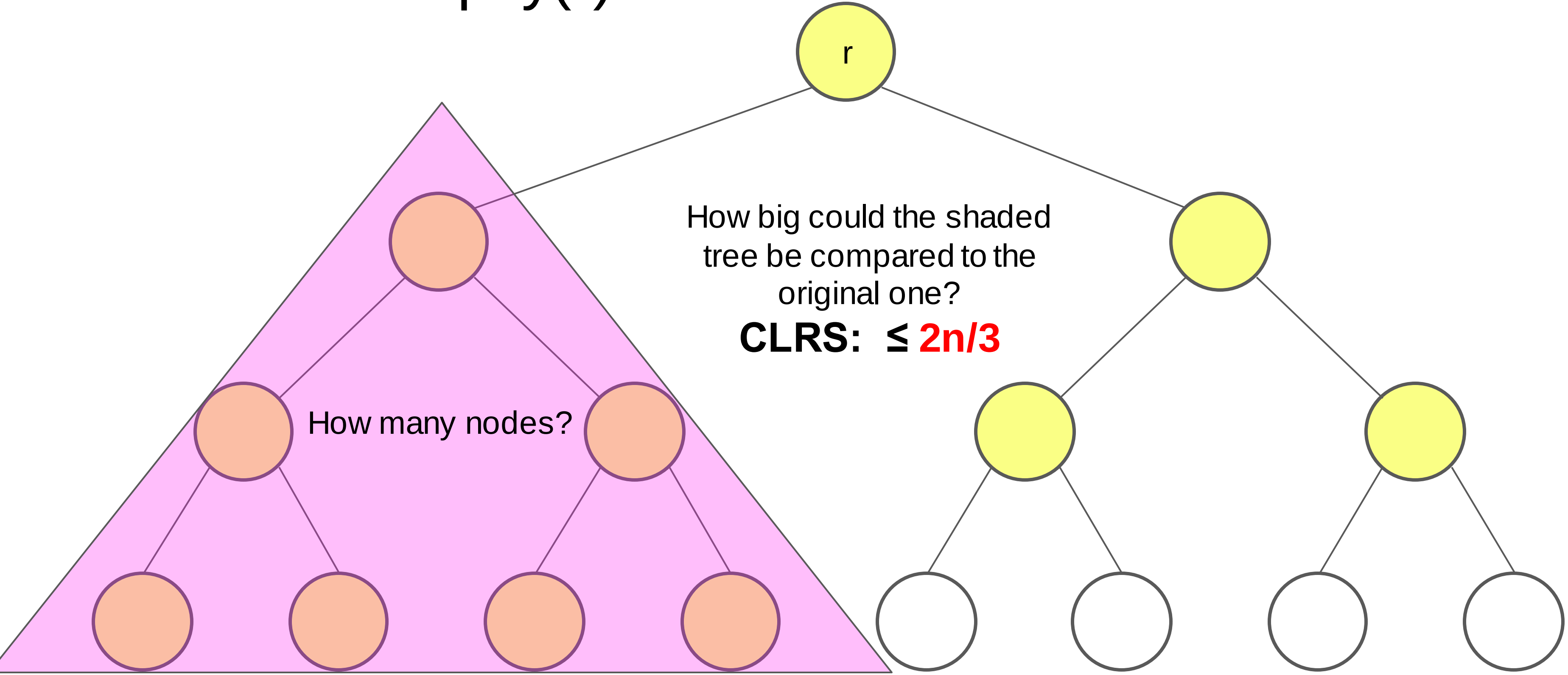
Time to process n nodes:

T(n) = T(?) + k

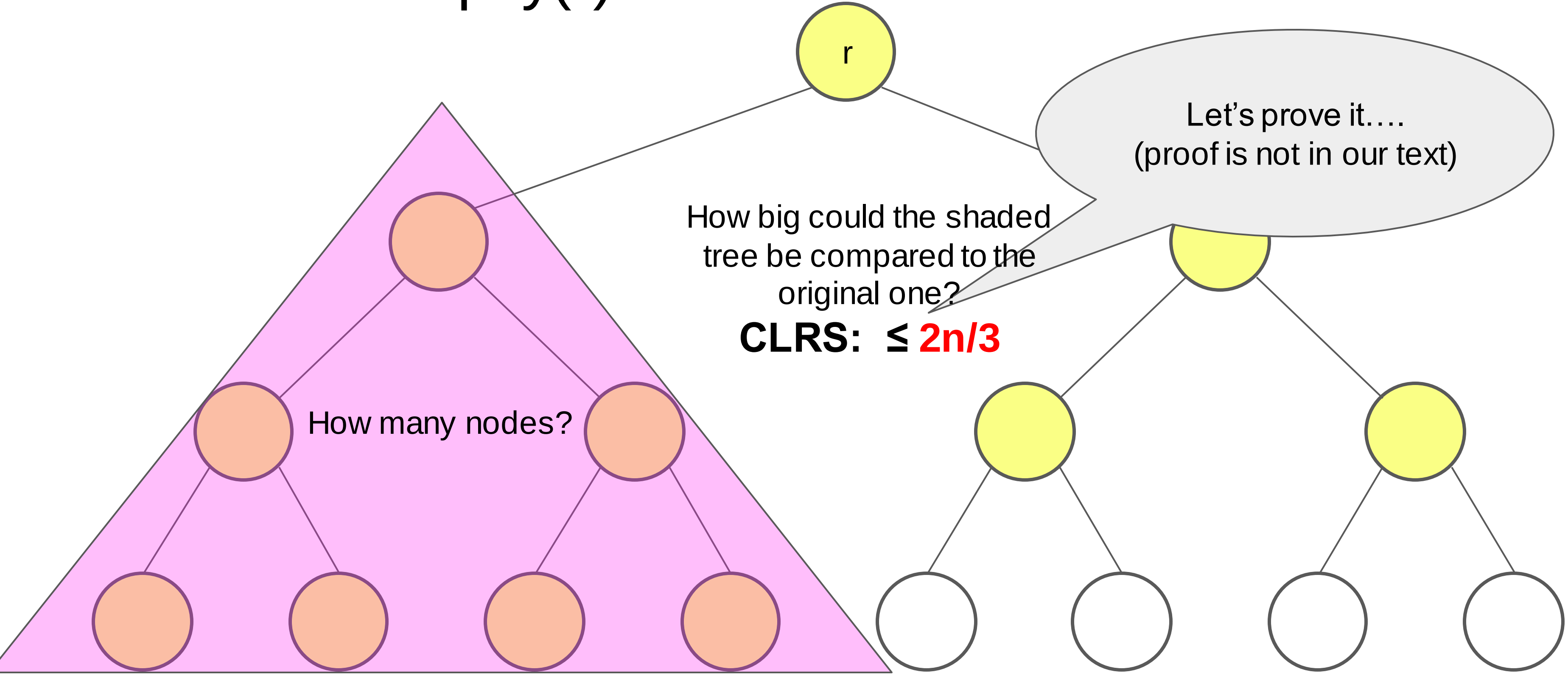
Consider Heapify(r) on a tree of n nodes



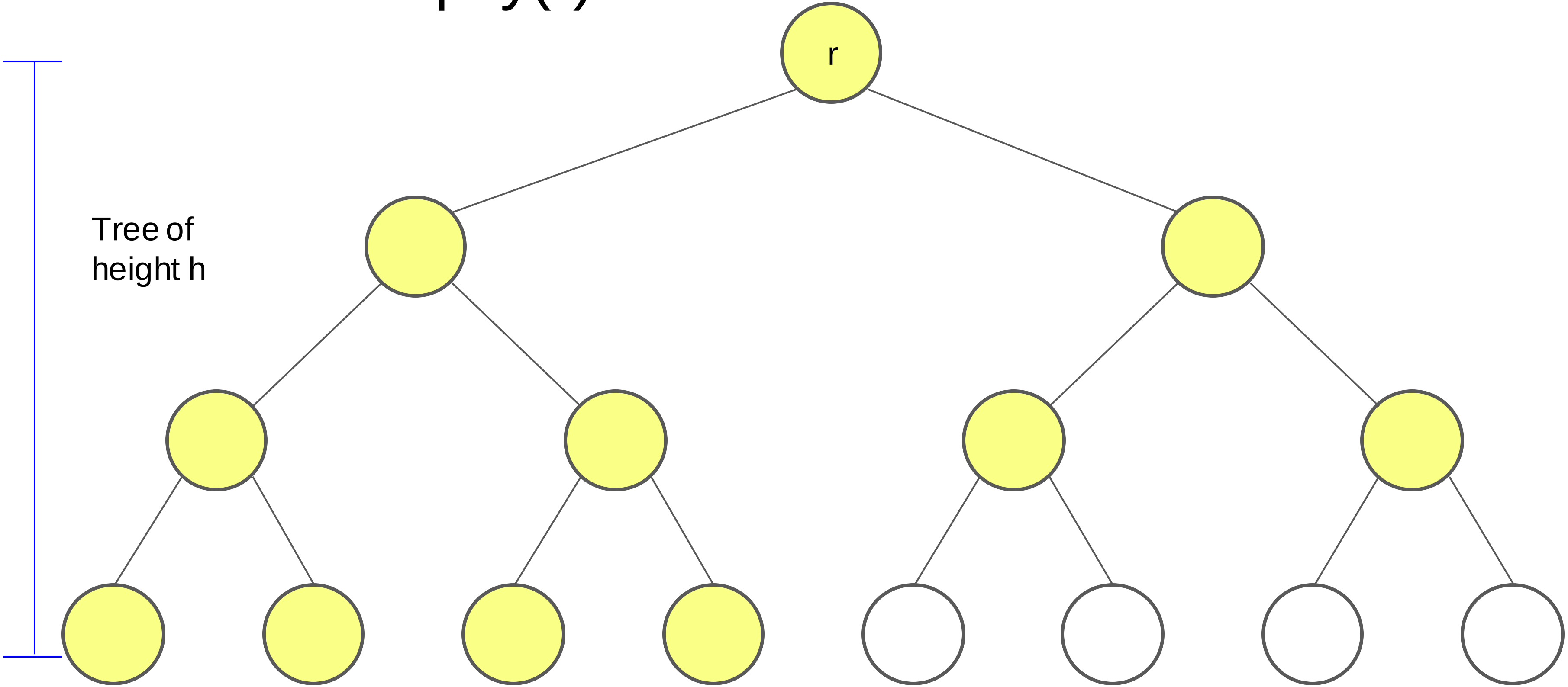
Consider Heapify(r) on a tree of n nodes



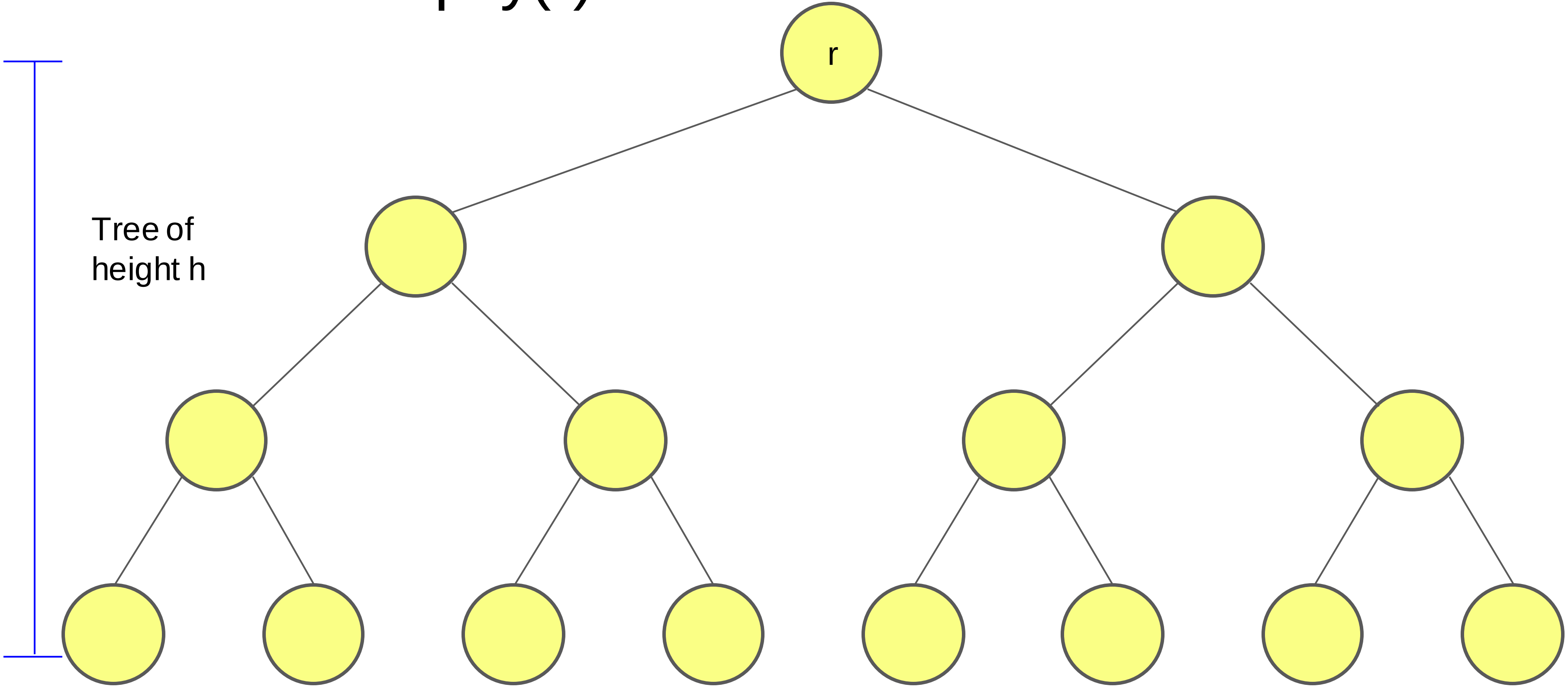
Consider Heapify(r) on a tree of n nodes



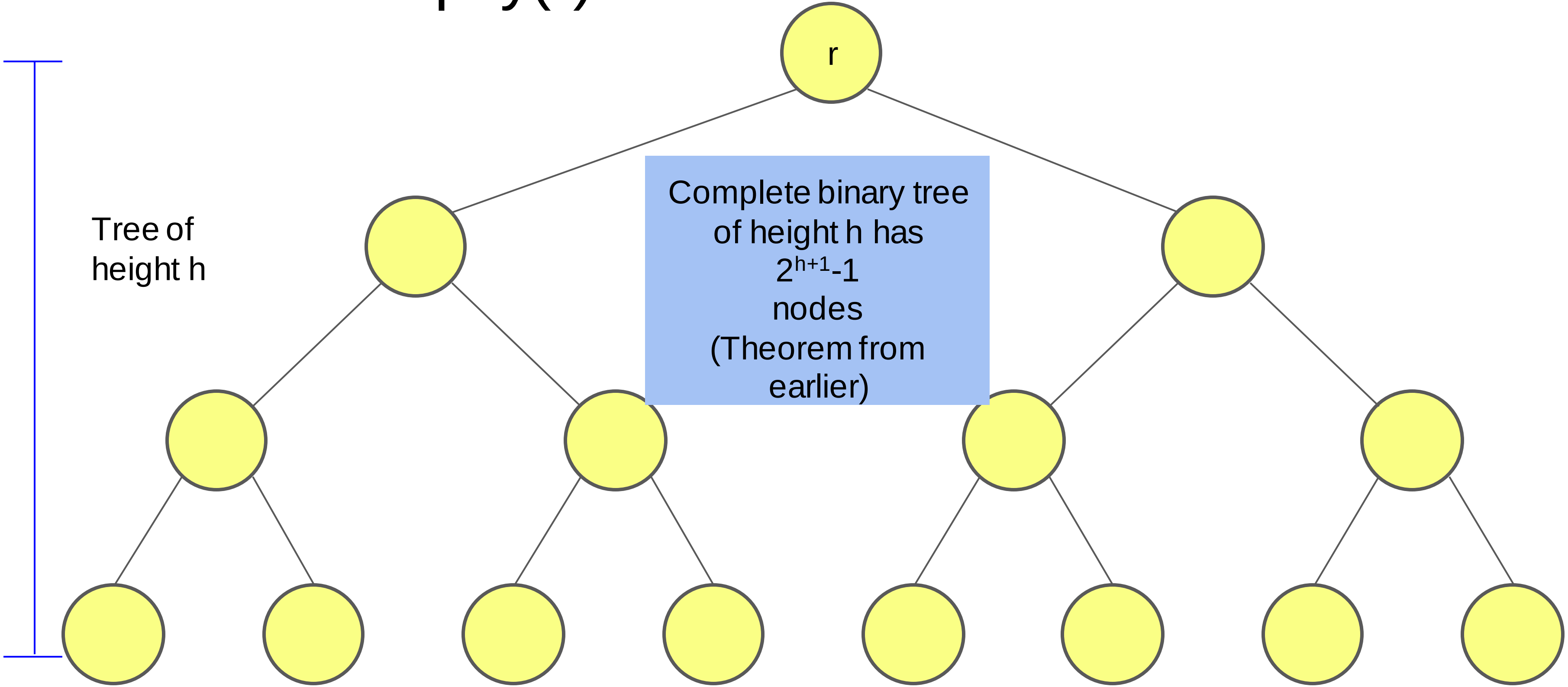
Consider Heapify(r) on a tree of n nodes



Consider Heapify(r) on a tree of n nodes

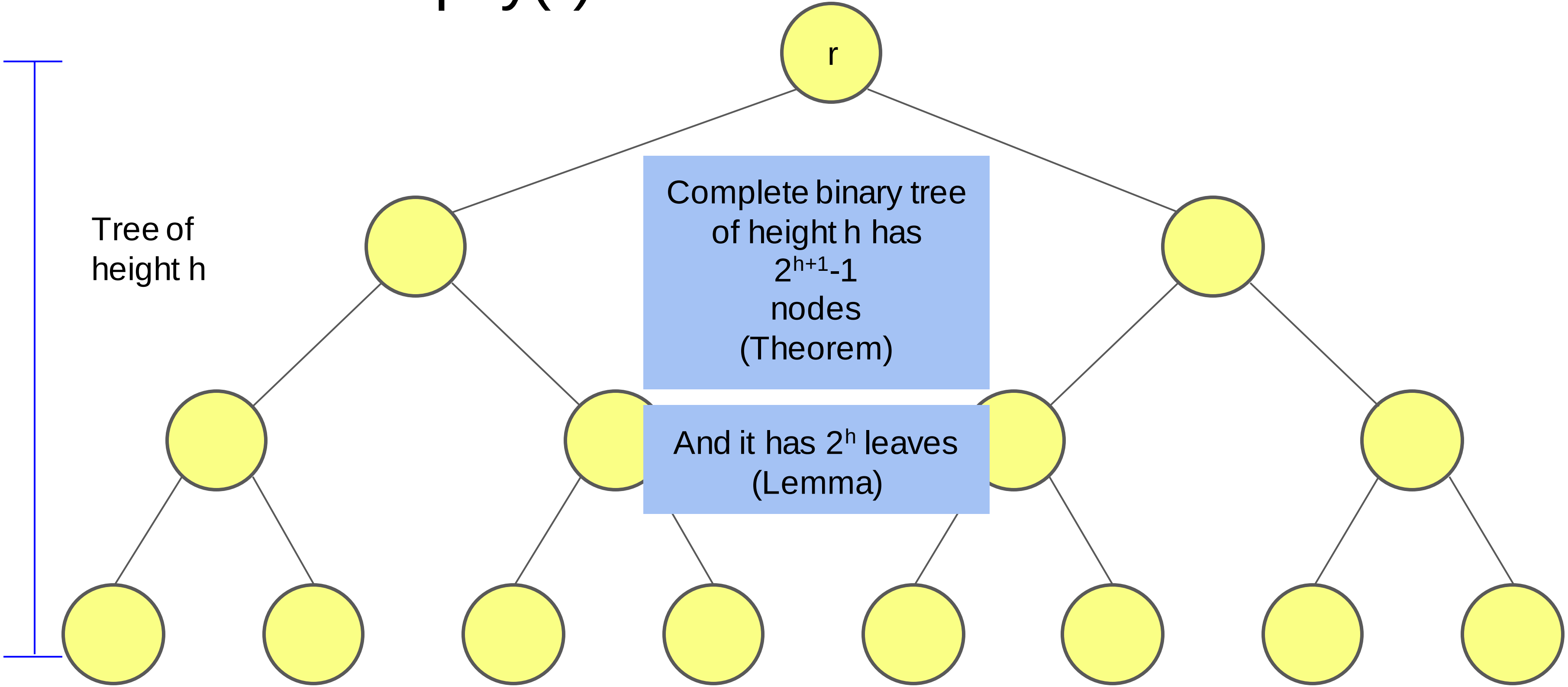


Consider Heapify(r) on a tree of n nodes

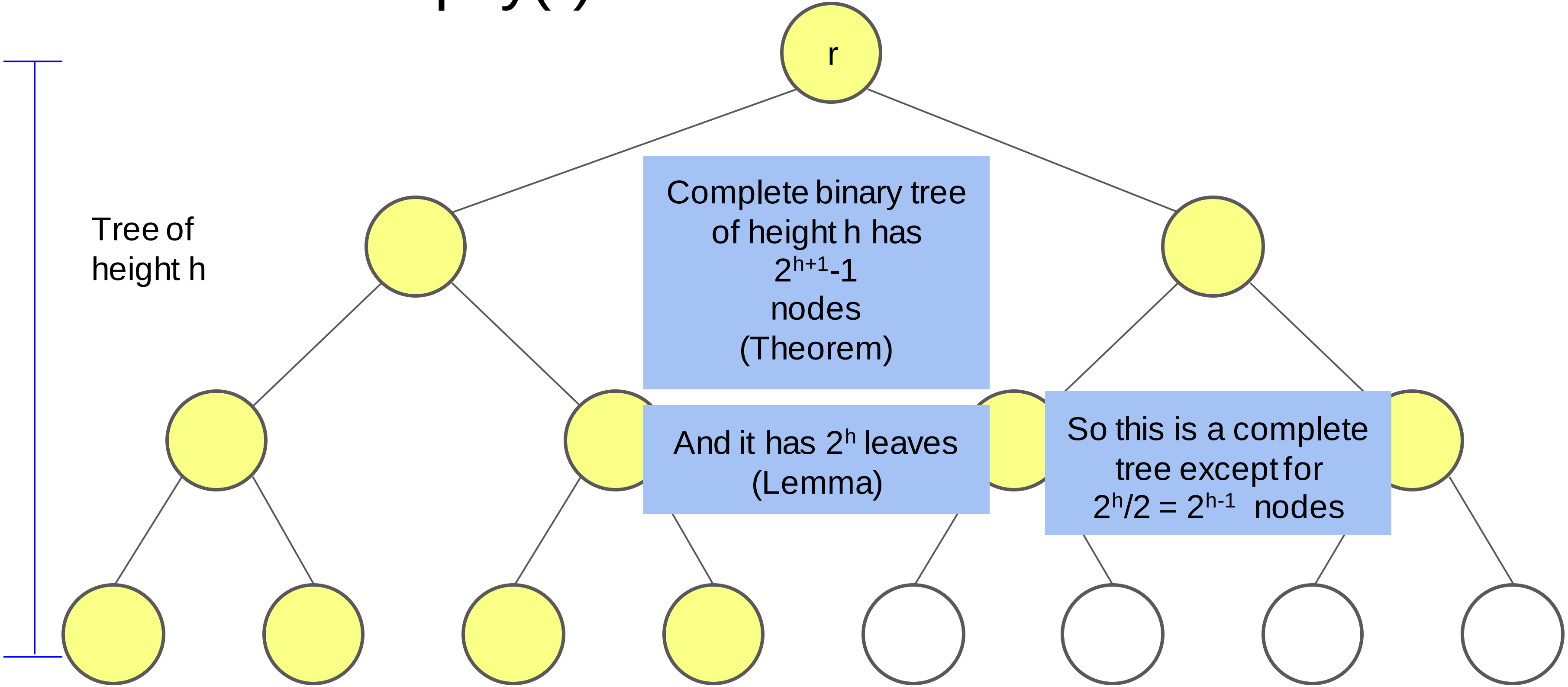


Complete binary tree of height h has 2^{h+1}-1 nodes (Theorem from earlier)

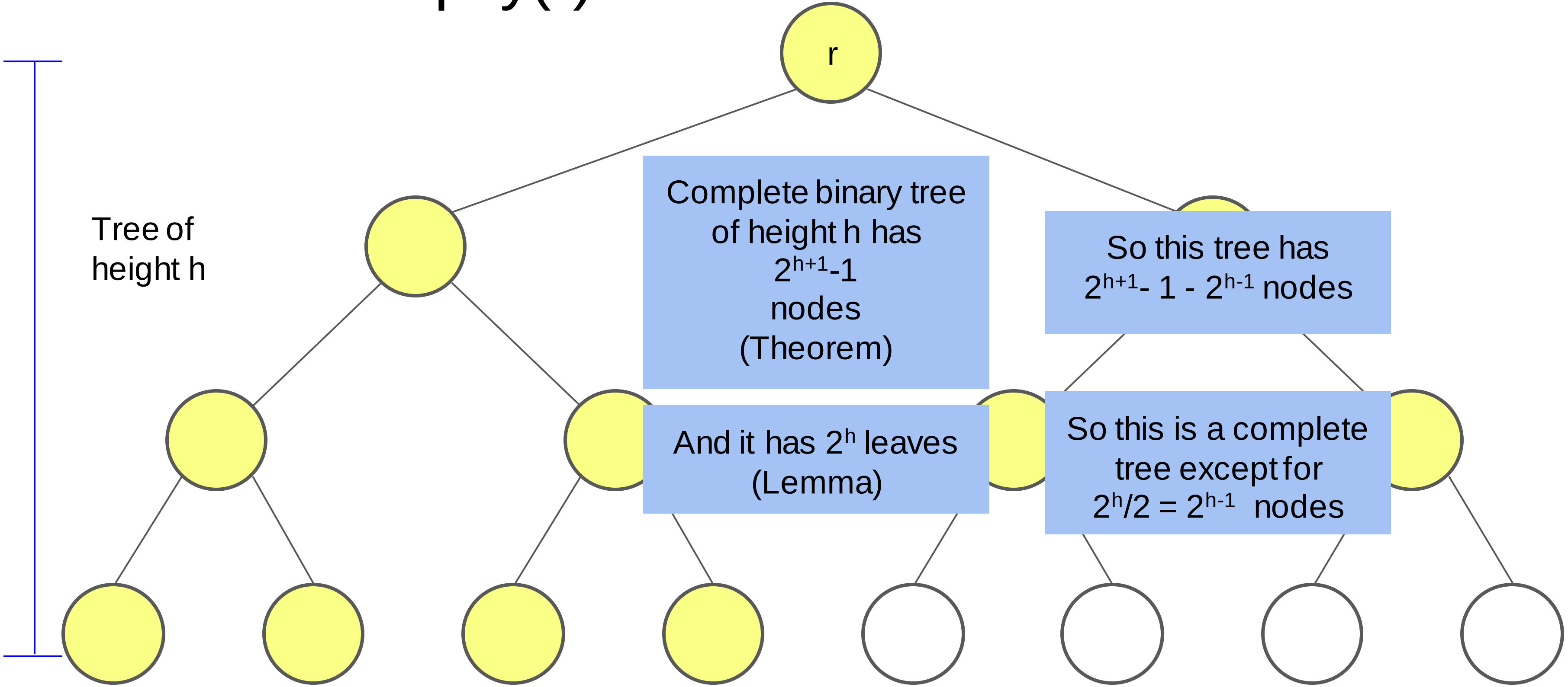
Consider Heapify(r) on a tree of n nodes



Consider Heapify(r) on a tree of n nodes

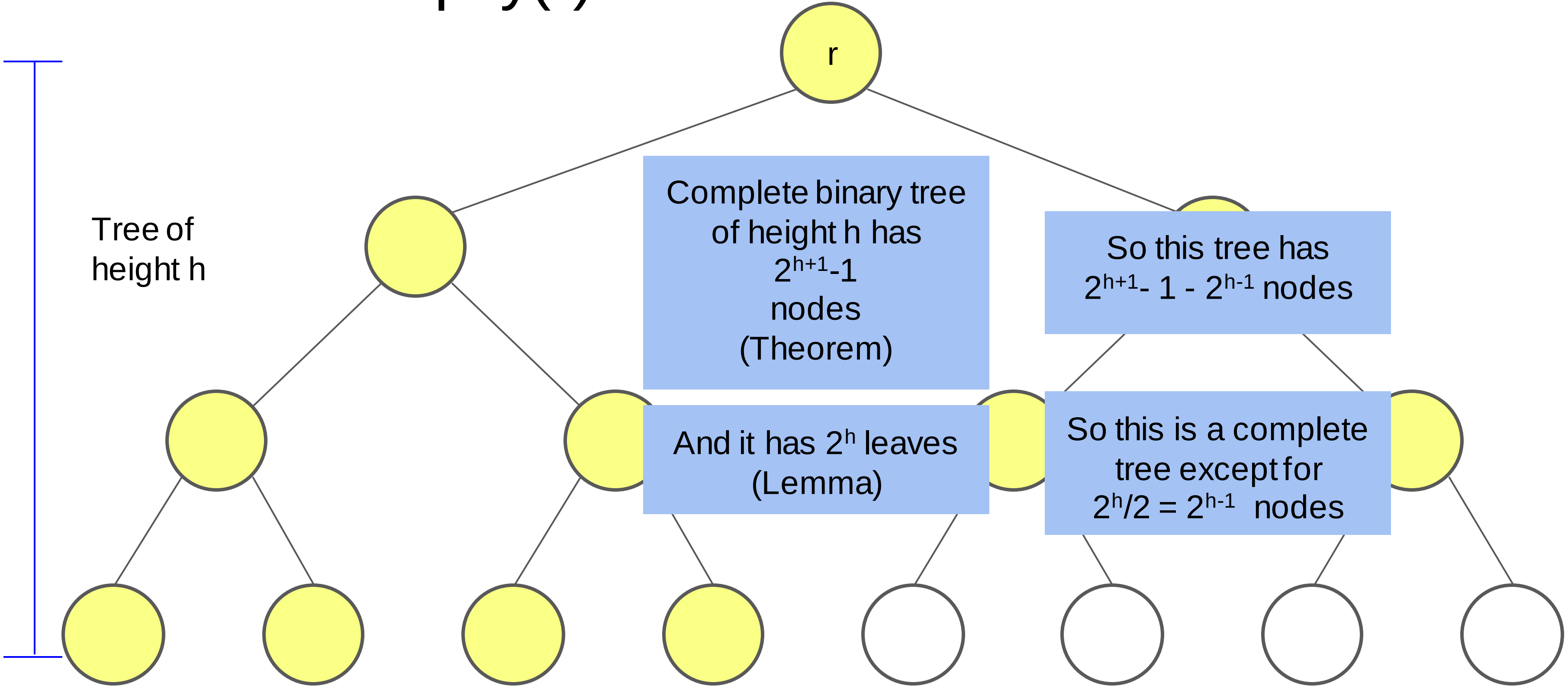


Consider Heapify(r) on a tree of n nodes



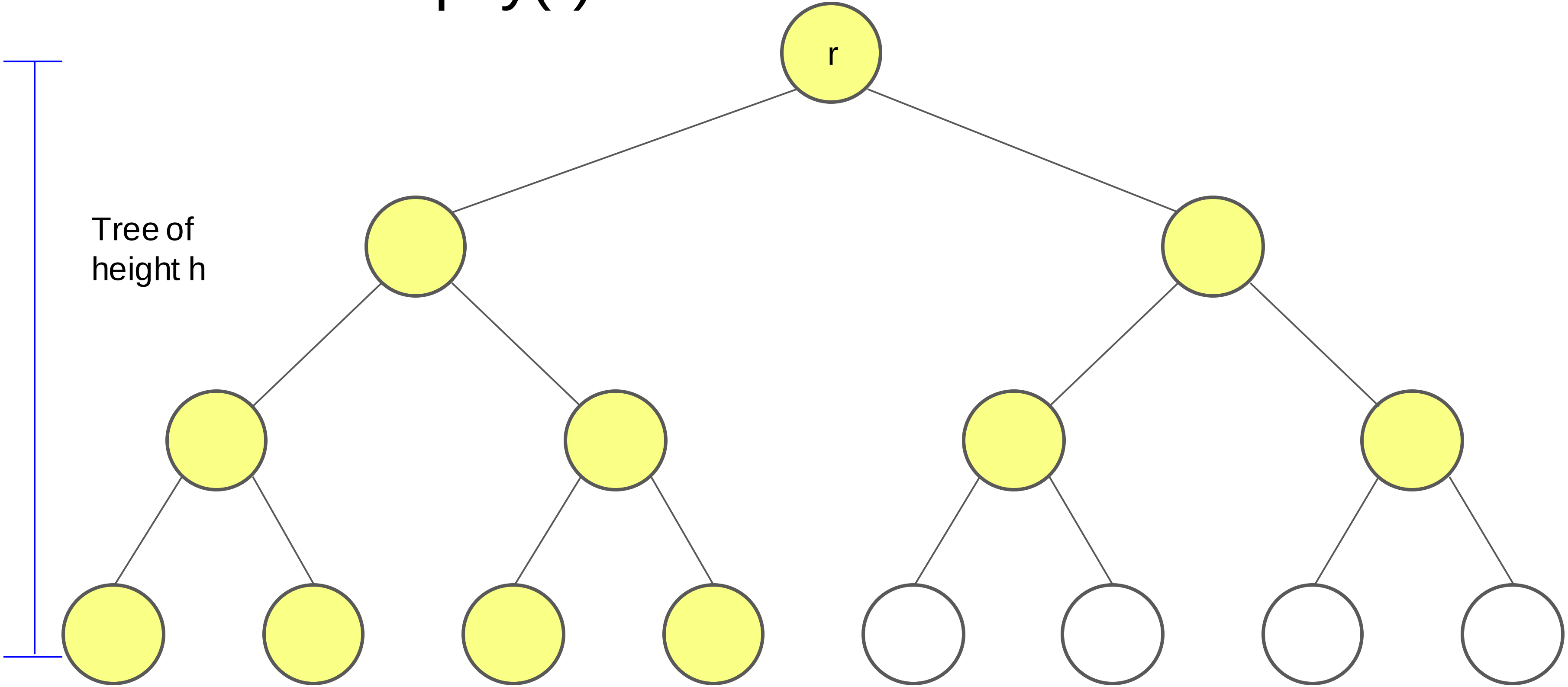
#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

Consider Heapify(r) on a tree of n nodes



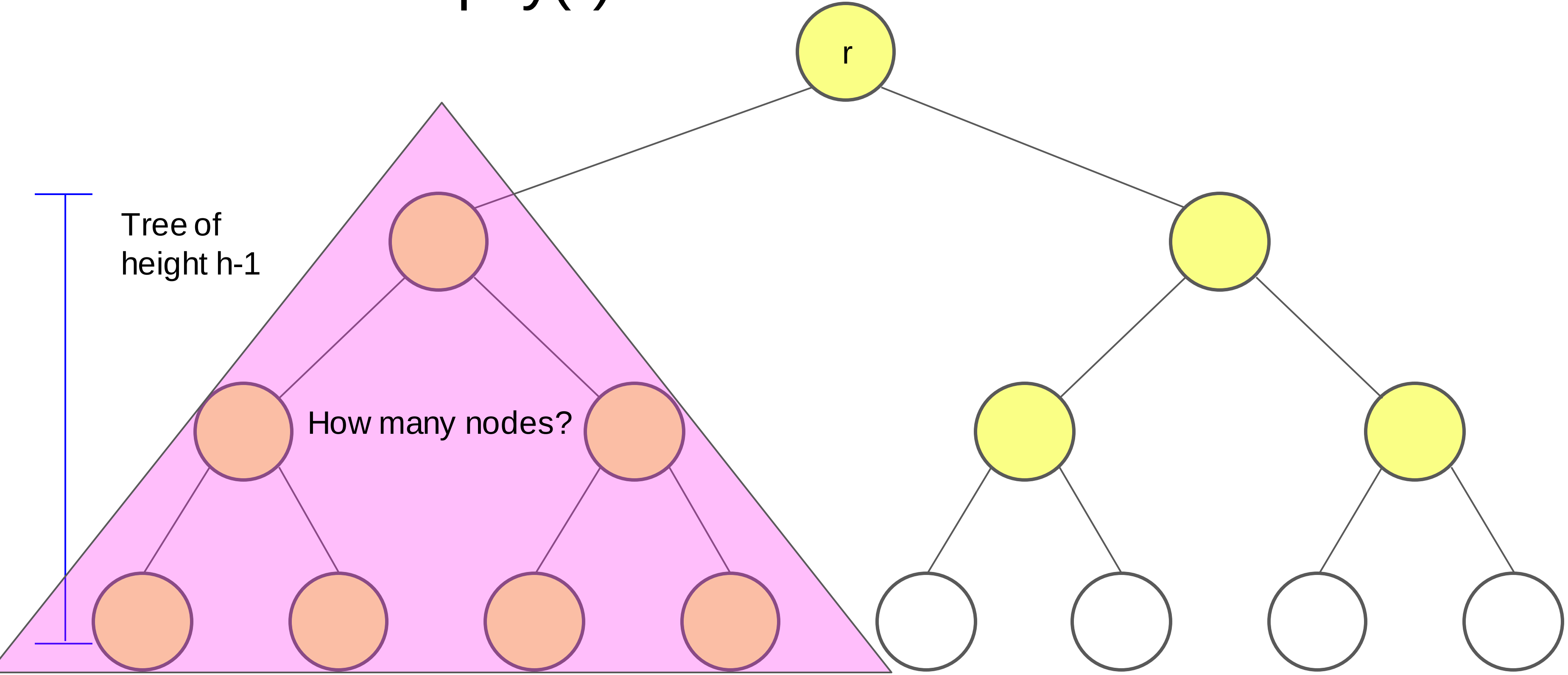
#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

Consider Heapify(r) on a tree of n nodes



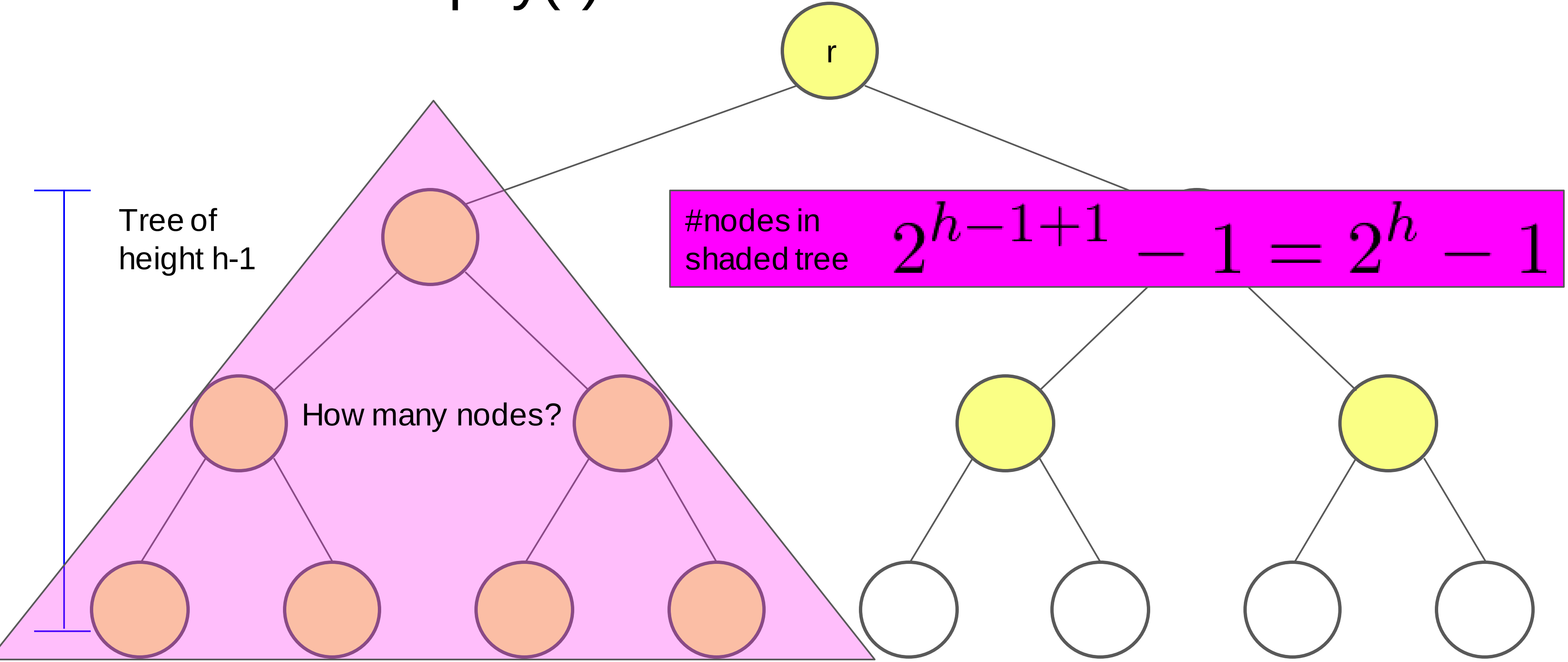
#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

Consider Heapify(r) on a tree of n nodes



#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

Consider Heapify(r) on a tree of n nodes



#nodes in shaded tree $2^{h-1+1} - 1 = 2^h - 1$

Let's compute the ratio

#nodes in shaded tree $2^{h-1+1} - 1 = 2^h - 1$

#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

Let’s compute the ratio

#nodes in shaded tree $2^{h-1+1} - 1 = 2^h - 1$

#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

$$= \frac{2^h - 1}{2^{h+1} - 2^{h-1} - 1}$$

Let’s compute the ratio

#nodes in shaded tree $2^{h-1+1} - 1 = 2^h - 1$

#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

$$= \frac{2^h - 1}{2^{h+1} - 2^{h-1} - 1}$$

$$= \frac{2 \times 2^{h-1} - 1}{4 \times 2^{h-1} - 2^{h-1} - 1}$$

Let's compute the ratio

#nodes in shaded tree $2^{h-1+1} - 1 = 2^h - 1$

#nodes in this tree: $2^{h+1} - 1 - 2^{h-1}$

$$= \frac{2^h - 1}{2^{h+1} - 2^{h-1} - 1}$$

$$= \frac{2 \times 2^{h-1} - 1}{4 \times 2^{h-1} - 2^{h-1} - 1} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

Let's compute the ratio

$$\frac{\boxed{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}}{\boxed{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

$$= \frac{2^h - 1}{2^{h+1} - 2^{h-1} - 1}$$

$$= \frac{2 \times 2^{h-1} - 1}{4 \times 2^{h-1} - 2^{h-1} - 1} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

Let's compute the ratio

$$\frac{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

Let’s compute the ratio

$$\frac{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

$$\lim_{h \to \infty} \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

Let's compute the ratio

$$\frac{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

$$\lim_{h \to \infty} \frac{\boxed{2 \times 2^{h-1}} - 1}{\boxed{3 \times 2^{h-1}} - 1}$$

Let's compute the ratio

$$\frac{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

$$\lim_{h\to\infty} \frac{\boxed{2 \times 2^{h-1}} - 1}{\boxed{3 \times 2^{h-1}} - 1} = \frac{2}{3}$$

Let's compute the ratio

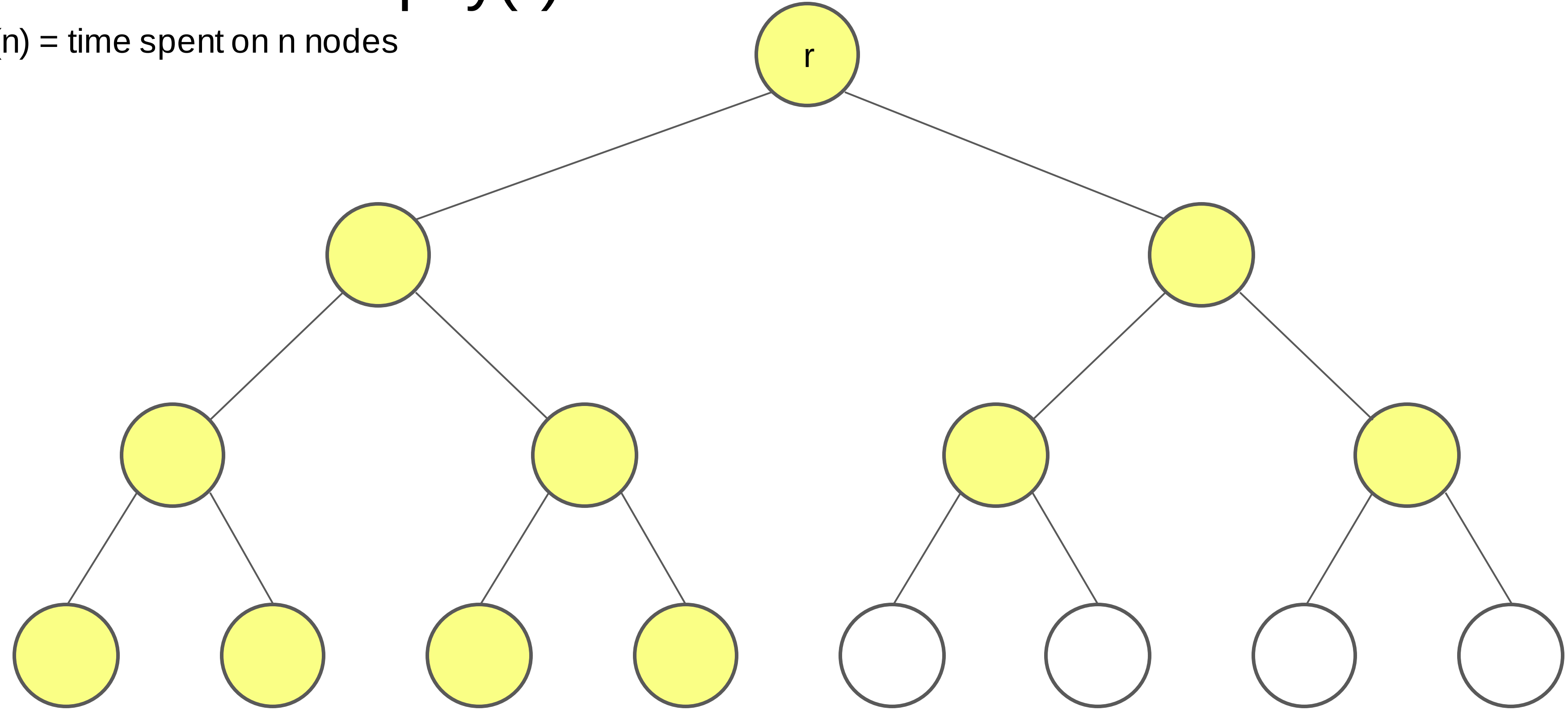
$$\frac{\text{\#nodes in shaded tree} \quad 2^{h-1+1} - 1 = 2^h - 1}{\text{\#nodes in this tree:} \quad 2^{h+1} - 1 - 2^{h-1}} = \frac{2 \times 2^{h-1} - 1}{3 \times 2^{h-1} - 1}$$

$$\lim_{h \to \infty} \frac{\boxed{2 \times 2^{h-1}} - 1}{\boxed{3 \times 2^{h-1}} - 1} = \frac{2}{3}$$

#nodes in shaded tree $= \dfrac{2}{3}$ #nodes in the larger tree

Consider Heapify(r) on a tree of n nodes

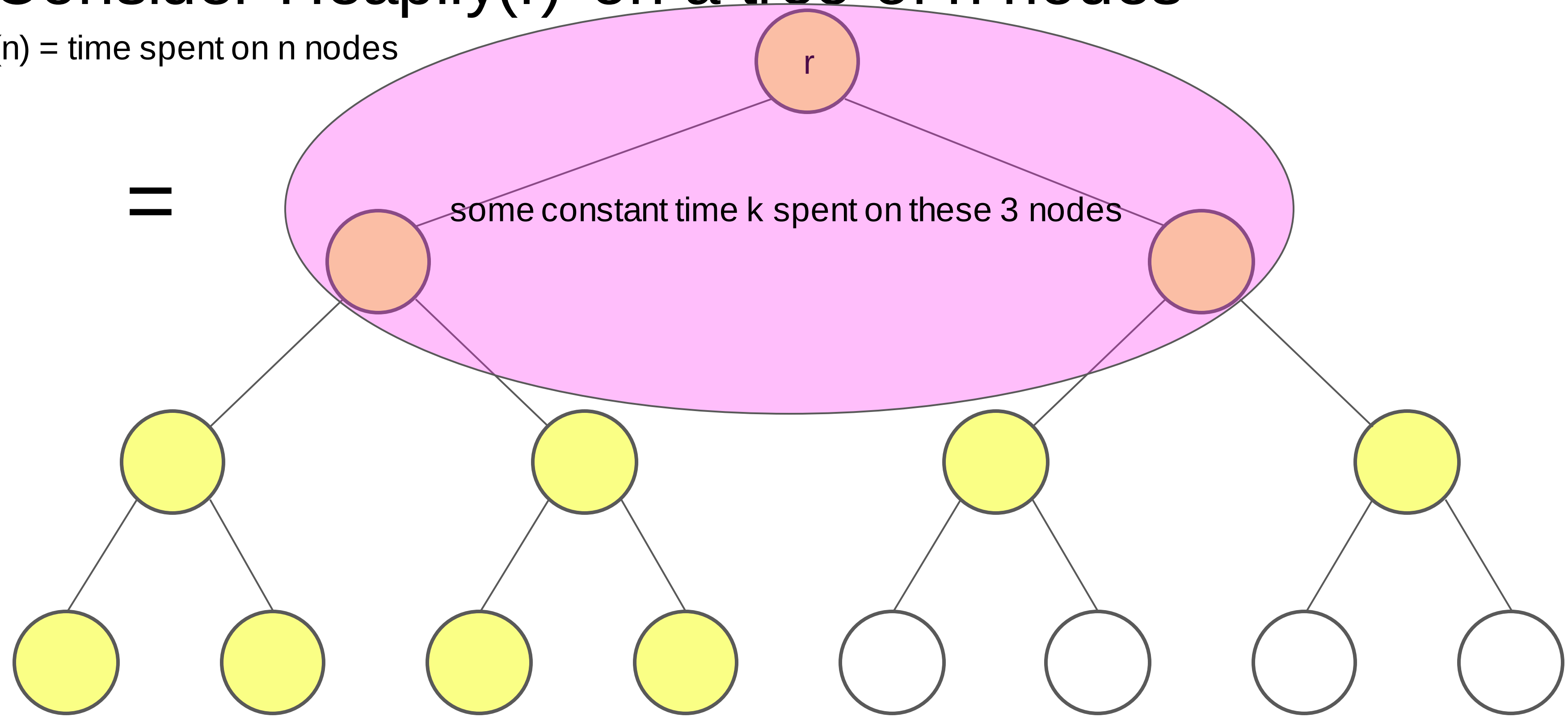
$T(n)$ = time spent on n nodes



Consider Heapify(r) on a tree of n nodes

T(n) = time spent on n nodes

=

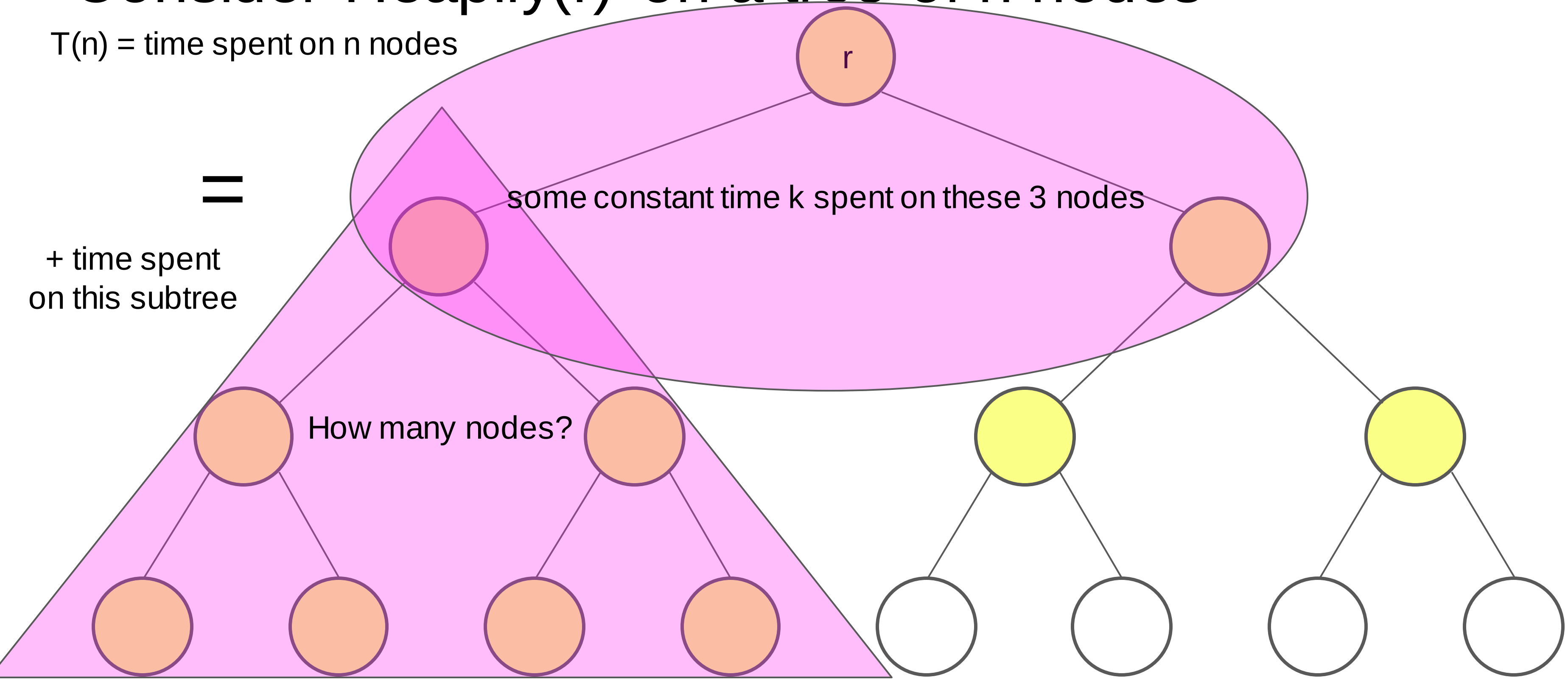


Consider Heapify(r) on a tree of n nodes

T(n) = time spent on n nodes

=

+ time spent
on this subtree

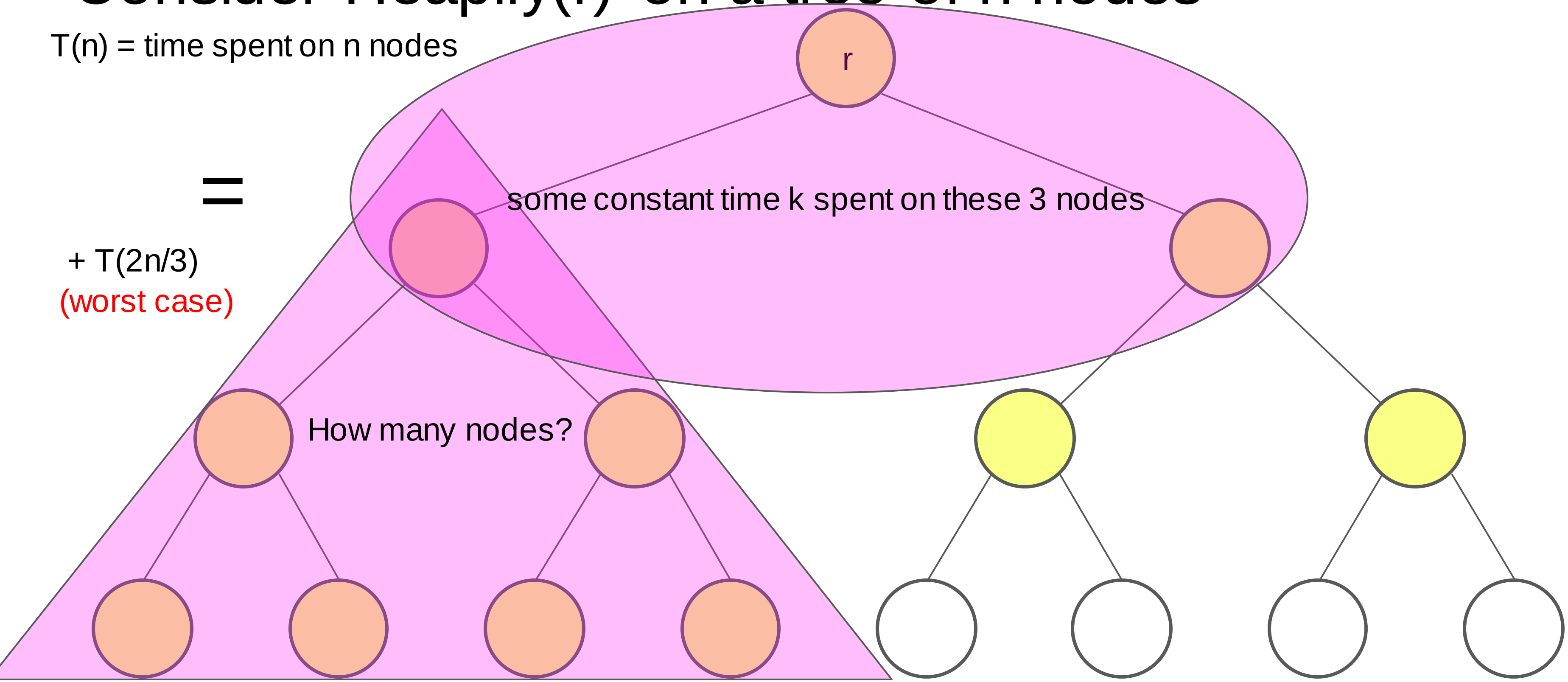


Consider Heapify(r) on a tree of n nodes

T(n) = time spent on n nodes

=

+ T(2n/3)
(worst case)



Consider Heapify(r) on a tree of n nodes

- $T(n) =$
  - k constant time spent on the top 3 nodes
  - $+ T(2n/3)$
- $T(n) = T(2n/3) + k$

Consider Heapify(r) on a tree of n nodes

- $T(n) =$
  - k constant time spent on the top 3 nodes
  - + T(2n/3)
- $T(n) = T(2n/3) + k$

$$
\begin{aligned}
T(100) &= T(66) + k \\
&= T(44) + k + k \\
&= T(29) + k + k + k \\
&= T(19) + k + k + k + k \\
&= T(12) + k + k + k + k + k \\
&= T(8) \quad + k + k + k + k + k + k \\
&= T(5) \quad + k + k + k + k + k + k + k \\
&= T(3) \quad + k + k + k + k + k + k + k + k \\
&= T(2) \quad + k + k + k + k + k + k + k + k + k \\
&= T(1) \quad + k + k + k + k + k + k + k + k + k + k \\
&= 0 \qquad + k + k + k + k + k + k + k + k + k + k \quad = 10\,k
\end{aligned}
$$

Consider Heapify(r) on a tree of n nodes

- $T(n) =$
  - k constant time spent on the top 3 nodes
  - $+ T(2n/3)$
- $T(n) = T(2n/3) + k$

$$
\begin{aligned}
T(100) &= T(66) + k \\
&= T(44) + k + k \\
&= T(29) + k + k + k \\
&= T(19) + k + k + k + k \\
&= T(12) + k + k + k + k + k \\
&= T(8) + k + k + k + k + k + k \\
&= T(5) + k + k + k + k + k + k + k \\
&= T(3) + k + k + k + k + k + k + k + k \\
&= T(2) + k + k + k + k + k + k + k + k + k \\
&= T(1) + k + k + k + k + k + k + k + k + k + k \\
&= 0 + k + k + k + k + k + k + k + k + k + k = 10\,k
\end{aligned}
$$

If the size of our problem is multiplied by 1.5, to get T(150), it takes just one more step, so T(150) = 11 k

Consider Heapify(r) on a tree of n nodes

We will be able to show soon that this $T(n) = \Theta(\log n)$

- $T(n) =$
  - k constant time spent on the top 3 nodes
  - $+ T(2n/3)$
- $T(n) = T(2n/3) + k$

If the size of our problem is multiplied by 1.5, to get T(150), it takes just one more step, so T(150) = 11 k

$$
\begin{aligned}
T(100) &= T(66) + k \\
&= T(44) + k + k \\
&= T(29) + k + k + k \\
&= T(19) + k + k + k + k \\
&= T(12) + k + k + k + k + k \\
&= T(8) + k + k + k + k + k + k \\
&= T(5) + k + k + k + k + k + k + k \\
&= T(3) + k + k + k + k + k + k + k + k \\
&= T(2) + k + k + k + k + k + k + k + k + k \\
&= T(1) + k + k + k + k + k + k + k + k + k + k \\
&= 0 + k + k + k + k + k + k + k + k + k + k = 10\,k
\end{aligned}
$$

Consider Heapify(r) on a tree of n nodes

- $T(n) =$
  - k constant time spent on the top 3 nodes
  - $+ T(2n/3)$
- $T(n) = T(2n/3) + k$
- [magic we have not yet studied but will do so next week]
- $T(n) = \Theta(\log n)$

- **Same asymptotic result as we got the other way**

- **Approach applies to many recursive procedures, as we'll see**

Summary of Binary Heap Performance

- Decrease: worst-case $\Theta(\log n)$
- Insert: worst-case $\Theta(\log n)$ [reduction from decrease]
- ExtractMin: worst-case $\Theta(\log n)$

Summary of Binary Heap Performance

- Decrease: worst-case Θ(log n)
- Insert: worst-case Θ(log n) [reduction from decrease]
- ExtractMin: worst-case Θ(log n)

Moral: we can dynamically maintain the minimum of a binary heap in time Θ(log n) per operation.

Follow-up: Time To Build Heap

- What does it cost to do n successive insertions into an empty heap?

Follow-up: Time To Build Heap

- What does it cost to do n successive insertions into an empty heap?

- k log(1) + k log(2) + …. + k log(n)

Follow-up: Time To Build Heap

- What does it cost to do n successive insertions into an empty heap?
- $k \log(1) + k \log(2) + \ldots. + k \log(n)$
- $\leq k \log(n) + k \log(n) + \ldots. + k \log(n) = kn \log(n)$

Follow-up: Time To Build Heap

- What does it cost to do n successive insertions into an empty heap?
- $k \log(1) + k \log(2) + \ldots. + k \log(n)$
- $\leq k \log(n) + k \log(n) + \ldots. + k \log(n) = kn \log(n)$
- So building a heap takes worst-case time $O(n \log n)$

Follow-up: Time To Build Heap

- What does it cost to do n successive insertions into an empty heap?
- k log(1) + k log(2) + …. + k log(n)
- ≤ k log(n) + k log(n) + …. + k log(n) = kn log(n)
- So building a heap takes worst-case time O(n log n)
- (But in fact, this O is not a Θ – see text for better bound!)

Practical advice: do not actually store your binary heap as a tree!

Efficient representation of binary heap

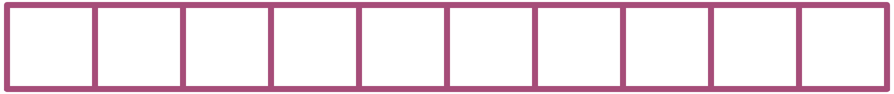
- Binary heaps have so far been depicted as trees
  - And we could implement them that way
  - But there is a more efficient treatment
  - Motivated by
    - Max size is known *a priori*
    - Elements are always added to the end for `insert(T thing)`
    - In response to `extractMin()`, `heapify()` removes the last element
- So an array is actually a good way to store a tree
  - But how do we keep track of
    - parents
    - children
- Easy solution to that for a binary tree

Tree implemented as an array

- An important implementation note
    - Java arrays
        - Start at 0

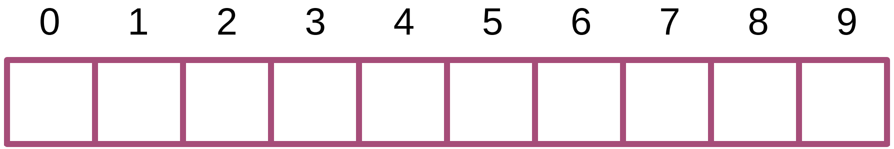
Tree implemented as an array

- An important implementation note
  - Java arrays
    - Start at 0
    - So `new int[10]`



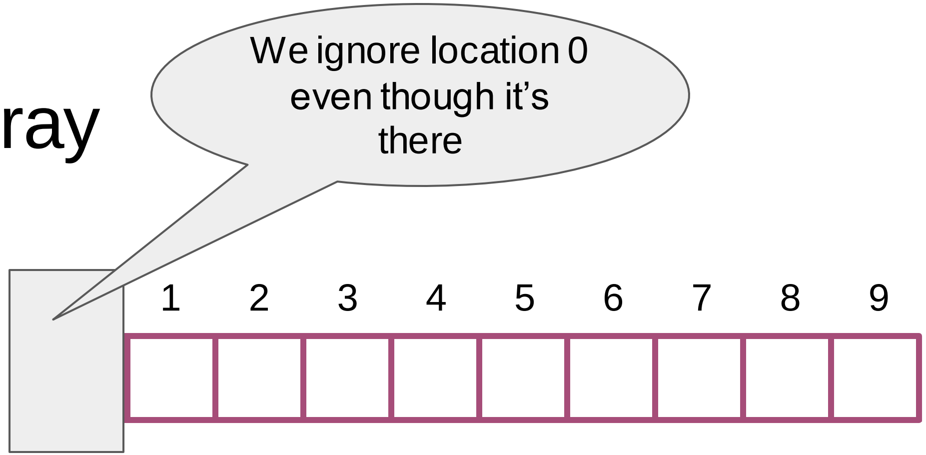
Tree implemented as an array

- An important implementation note
  - Java arrays
    - Start at 0
    - So `new int[10]`
      - Provides for 10 integer locations
      - Numbered 0….9



- We could start filling in the array at 0
- But for the purposes of the binary heap we will start at 1
  - The text does it this way, so we'll be consistent with it.
  - The math that follows is *very slightly* easier starting with 1
  - Older programming languages started arrays at 1 (some, like Matlab, still do)

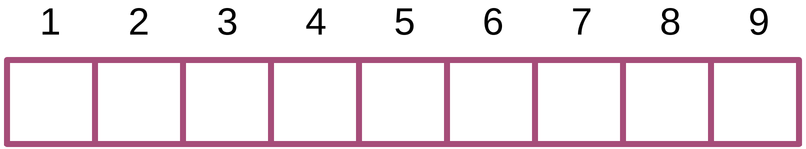
Tree implemented as an array



- An important implementation note
  - Java arrays
    - Start at 0
    - So `new int[10]`
      - Provides for 10 integer locations
      - Numbered 0….9
- We could start filling in the array at 0
- But for the purposes of the binary heap we will start at 1
  - The text does it this way, so we'll be consistent with it.
  - The math that follows is *very slightly* easier starting with 1
  - Older programming languages started arrays at 1 (some, like Matlab, still do)

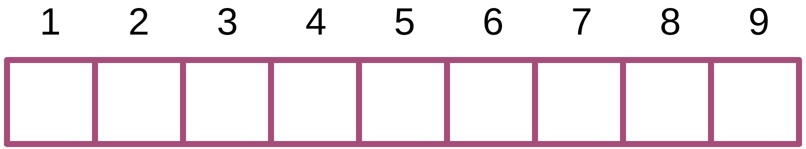
Tree implemented as an array

- So we will store the tree in an array
- It’s a binary tree
  - So each node has at most 2 children
- It’s compact
  - So it’s predictable where childless parents will appear
    - Near the end



Tree implemented as an array

- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent



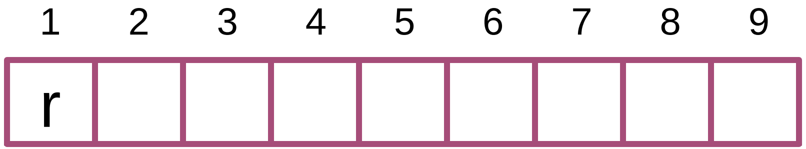
Tree implemented as an array

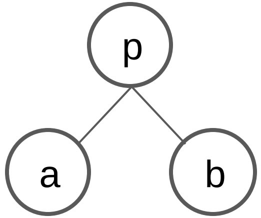
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1

1	2	3	4	5	6	7	8	9
r								

Tree implemented as an array

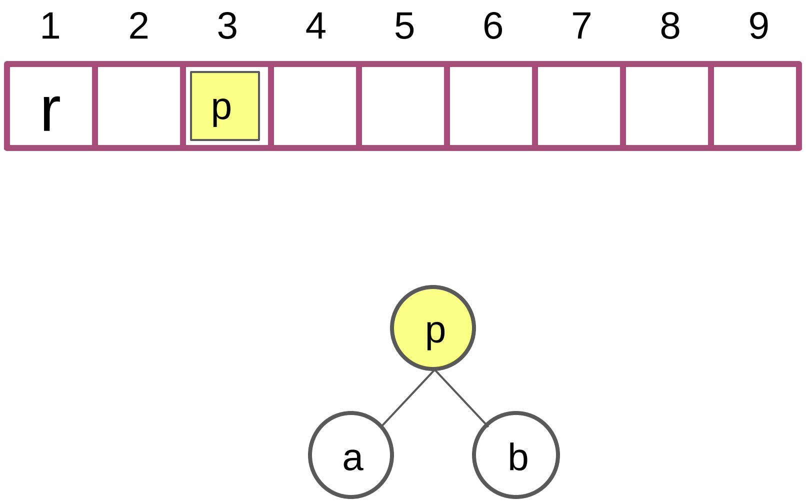
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b





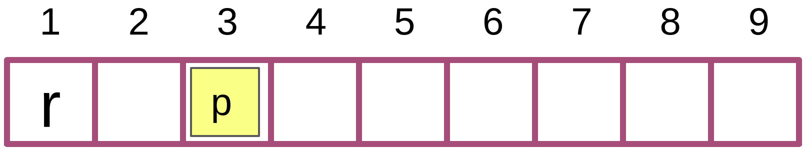
Tree implemented as an array

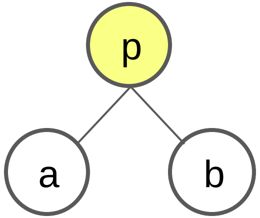
- So we will store the tree in an array
- How do we infer the relationship
    - Between a parent and its children
    - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
    - Left child a
    - Right child b
- If p is stored at index i



Tree implemented as an array

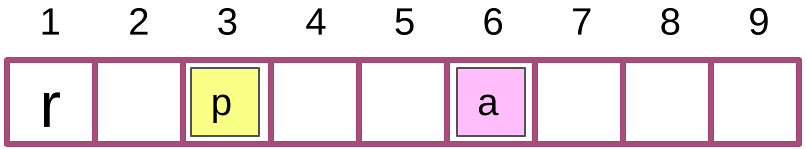
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b
- If p is stored at index i
  - The left child is stored at 2 x i

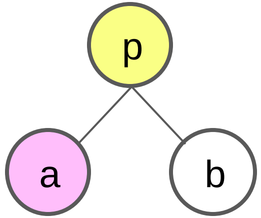




Tree implemented as an array

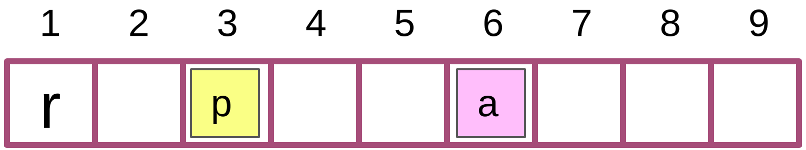
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b
- If p is stored at index i
  - The left child is stored at 2 x i

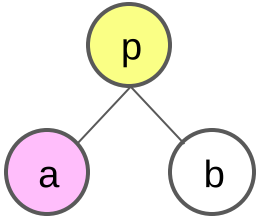




Tree implemented as an array

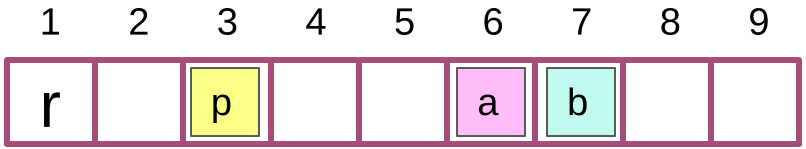
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b
- If p is stored at index i
  - The left child is stored at 2 x i
  - The right child is stored at 2 x i + 1

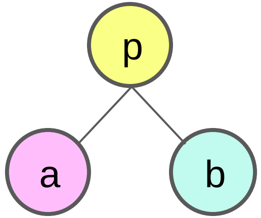




Tree implemented as an array

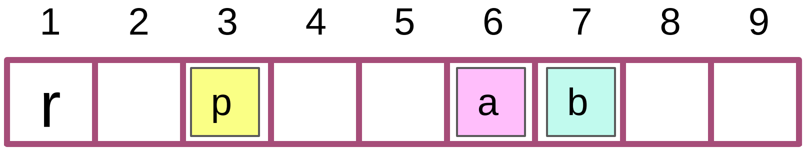
- So we will store the tree in an array
- How do we infer the relationship
    - Between a parent and its children
    - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
    - Left child a
    - Right child b
- If p is stored at index i
    - The left child is stored at 2 x i
    - The right child is stored at 2 x i + 1

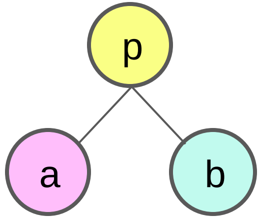




Tree implemented as an array

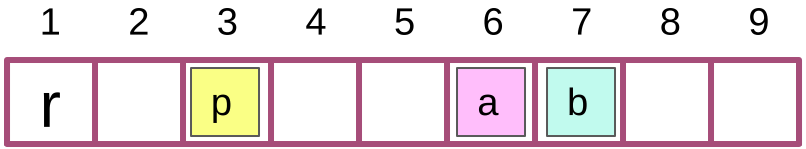
- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b
- If p is stored at index i
  - The left child is stored at 2 x i
  - The right child is stored at 2 x i + 1
- For every node n except the root
  - The parent of n is at $\left\lfloor \frac{n}{2} \right\rfloor$



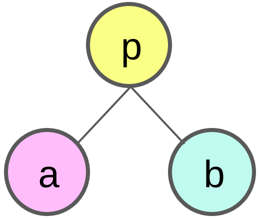


Tree implemented as an array

- So we will store the tree in an array
- How do we infer the relationship
  - Between a parent and its children
  - Between a child and its parent
- The root will always be stored at 1
- Given a parent node p
  - Left child a
  - Right child b
- If p is stored at index i
  - The left child is stored at 2 x i
  - The right child is stored at 2 x i + 1
- For every node n except the root
  - The parent of n is at $\left\lfloor \frac{n}{2} \right\rfloor$



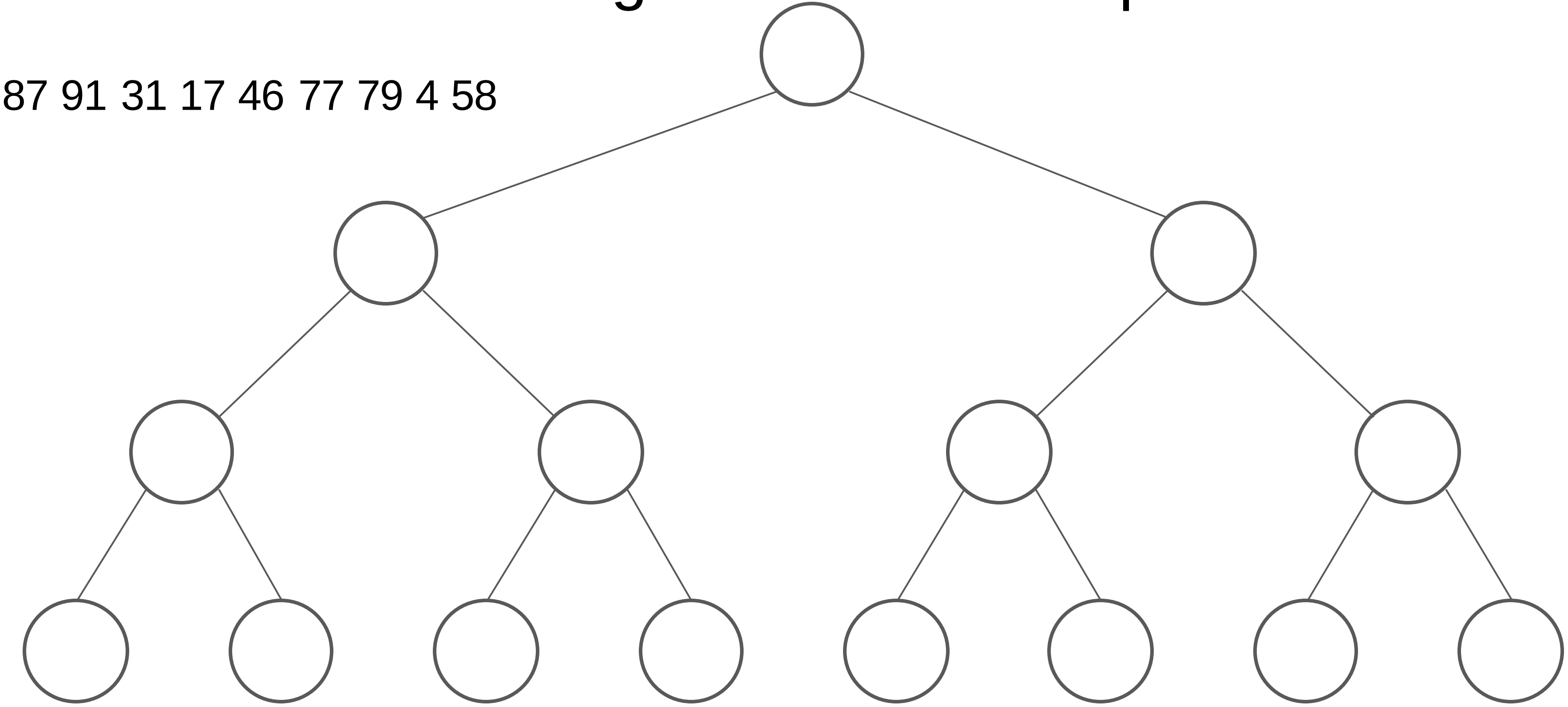
This is the result you get from normal int division:
6/2 = 3
7/2 = 3



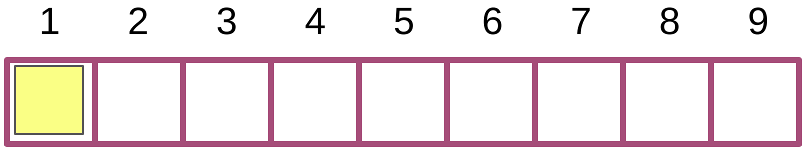
Back to our example, but using an array

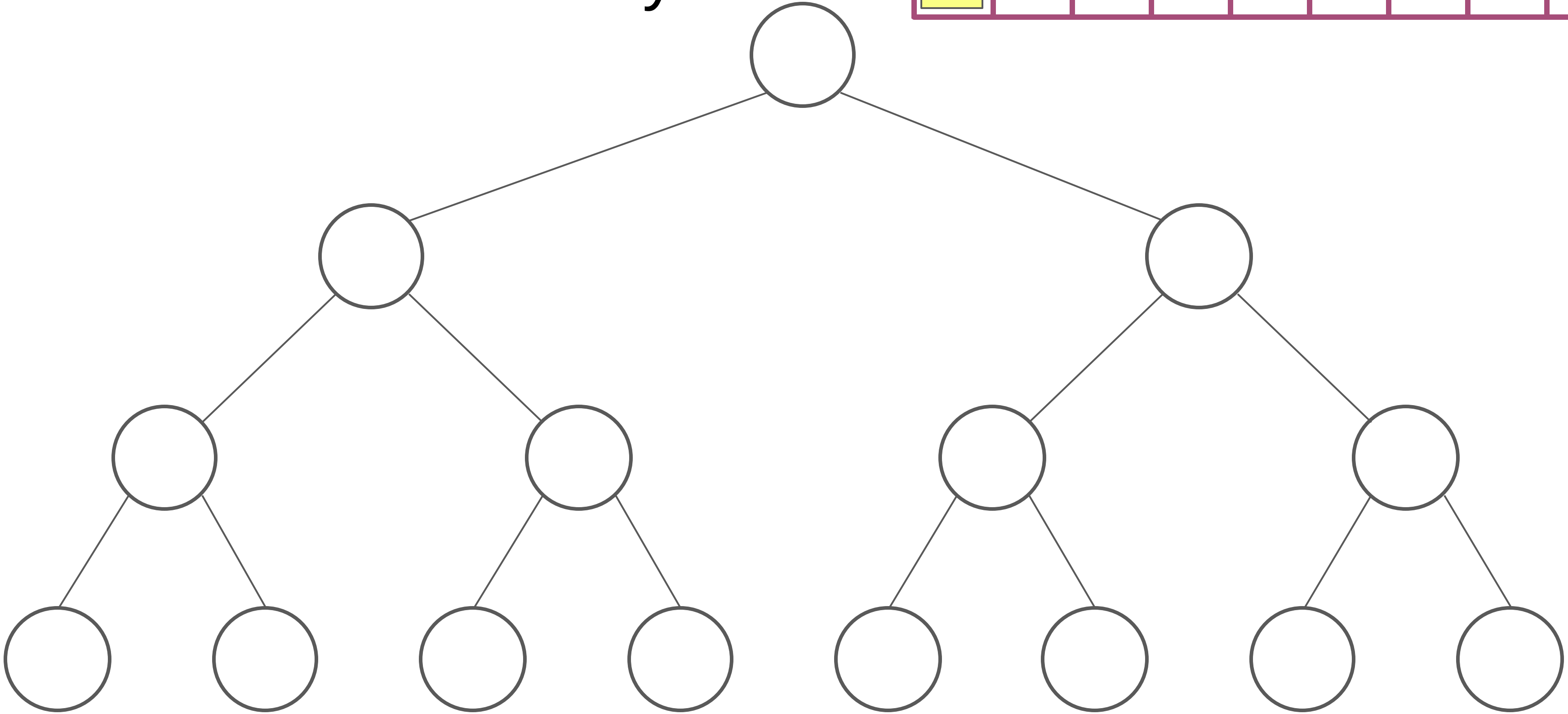
Consider the following values in a heap

87 91 31 17 46 77 79 4 58

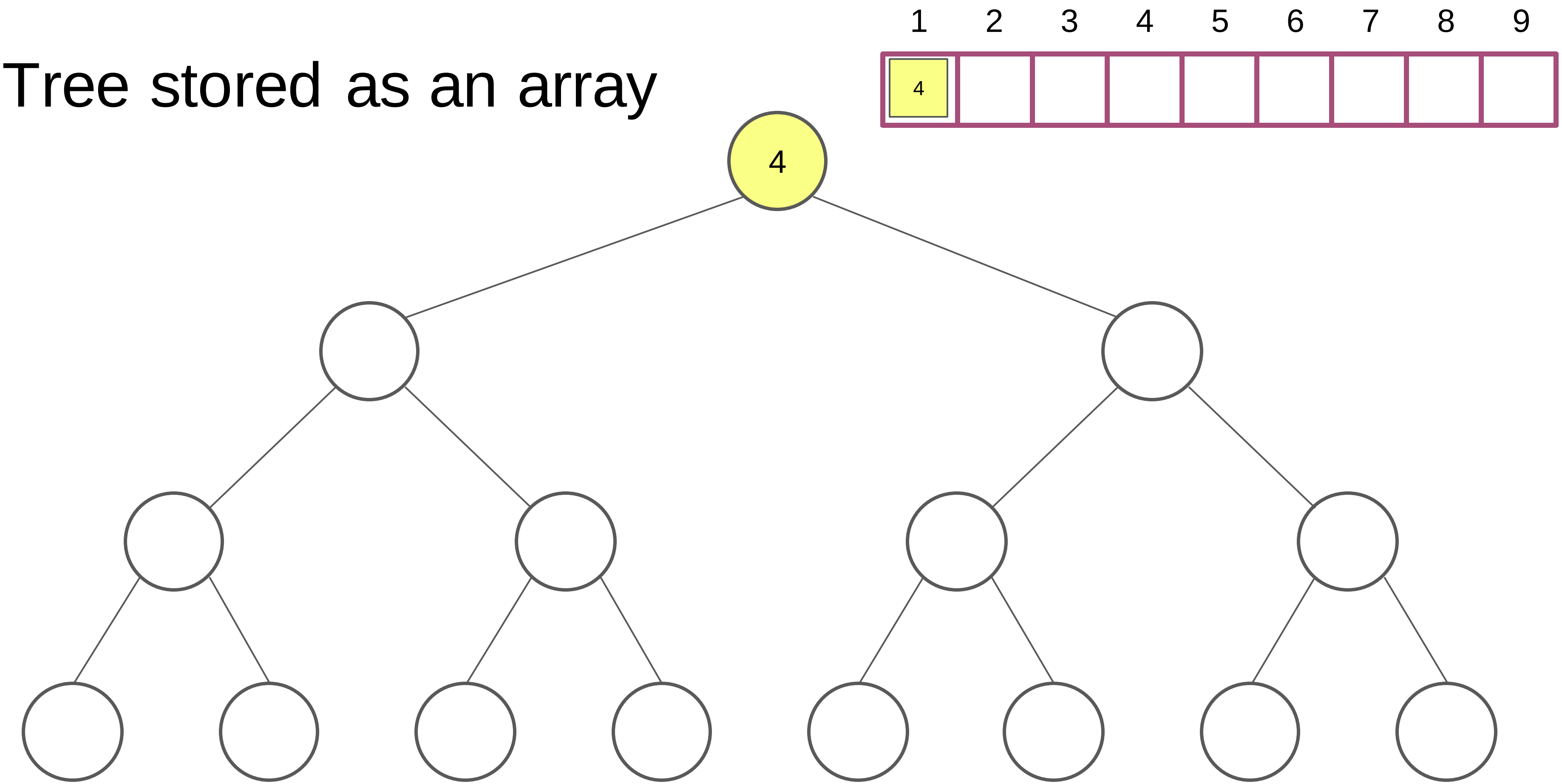


Tree stored as an array

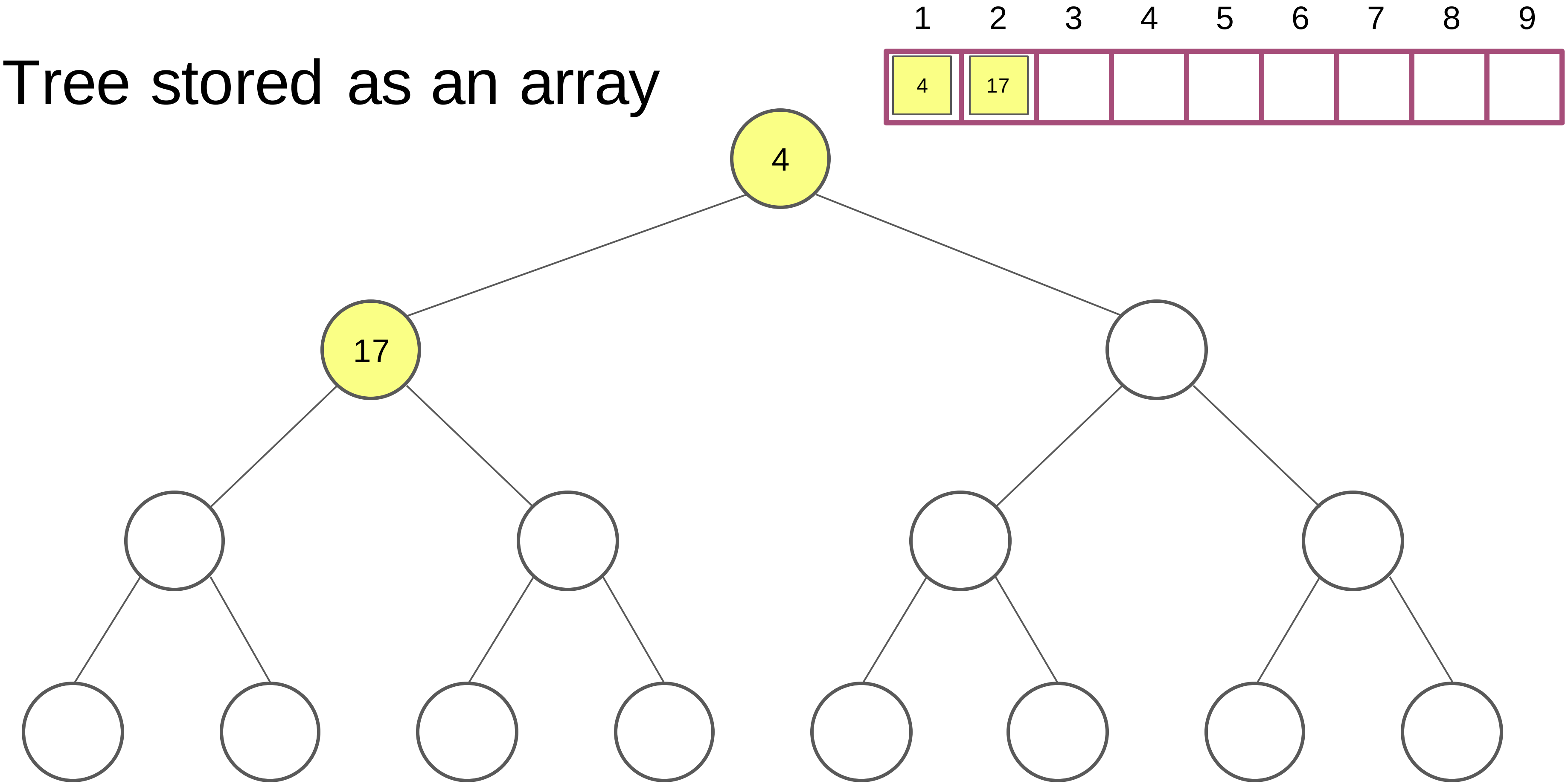




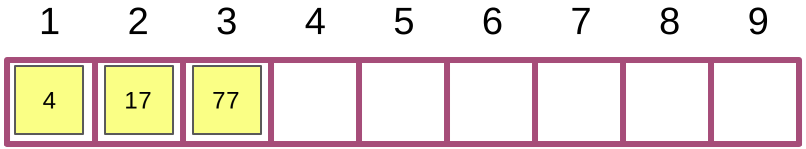
Tree stored as an array

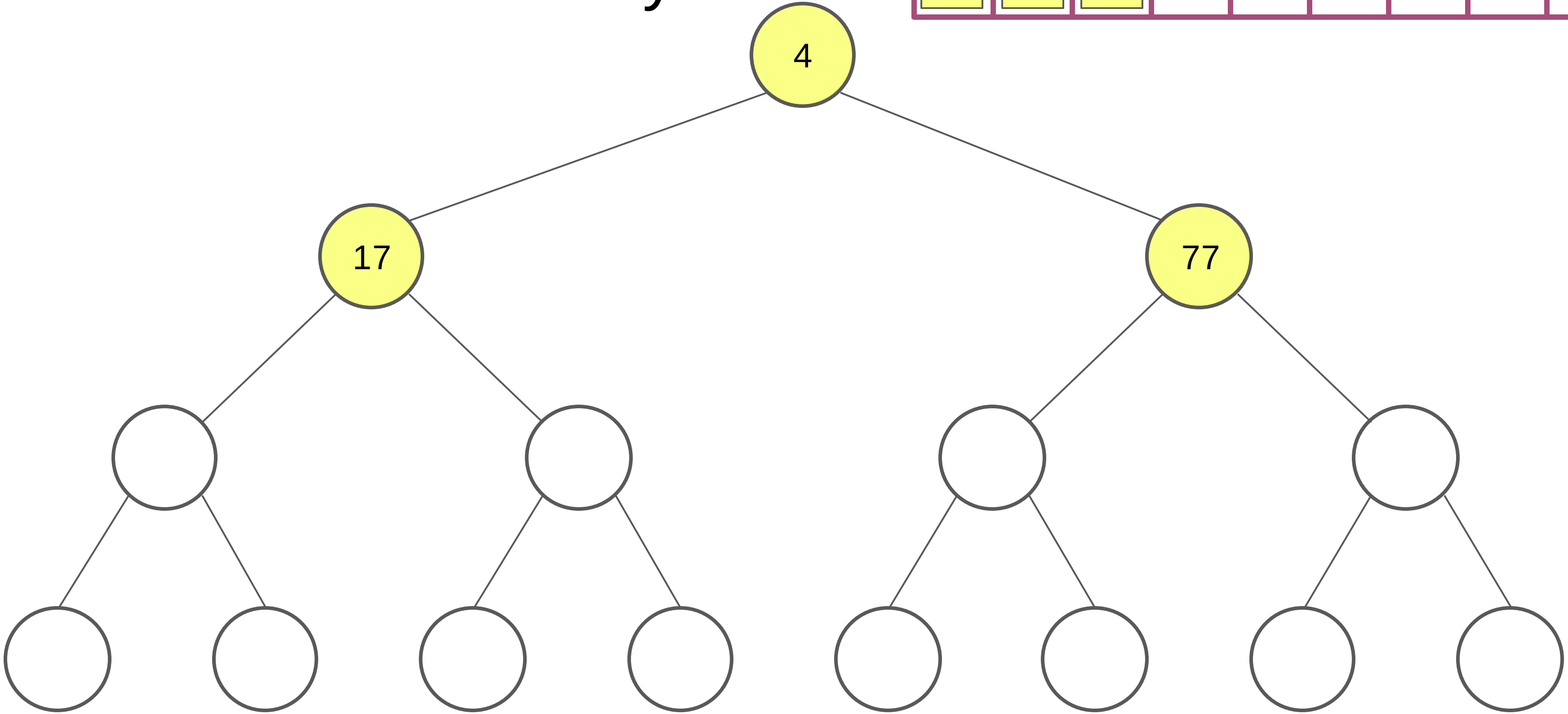


Tree stored as an array

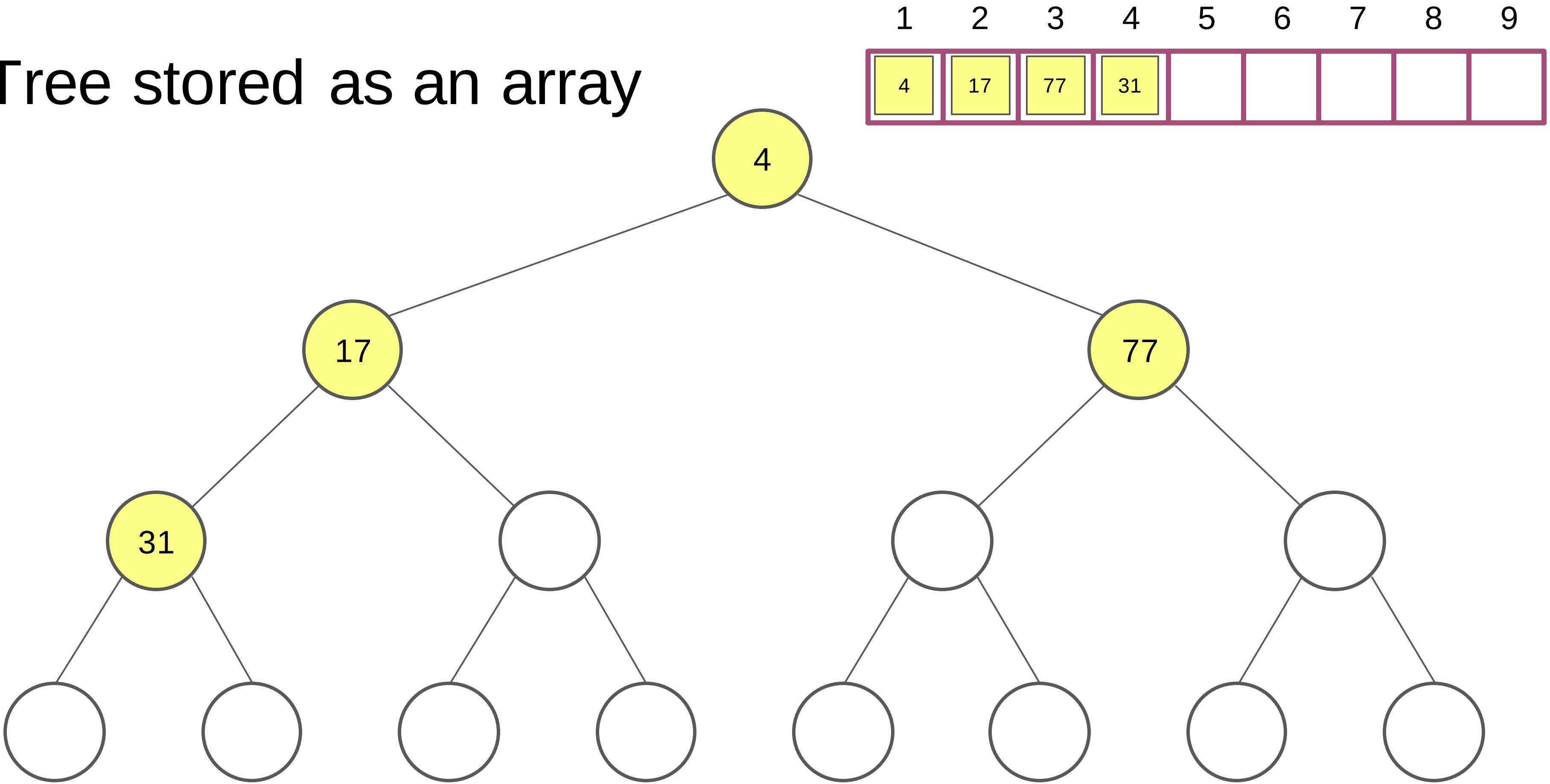


Tree stored as an array

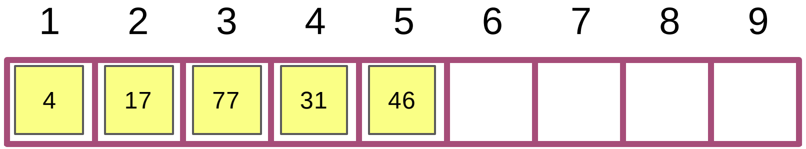


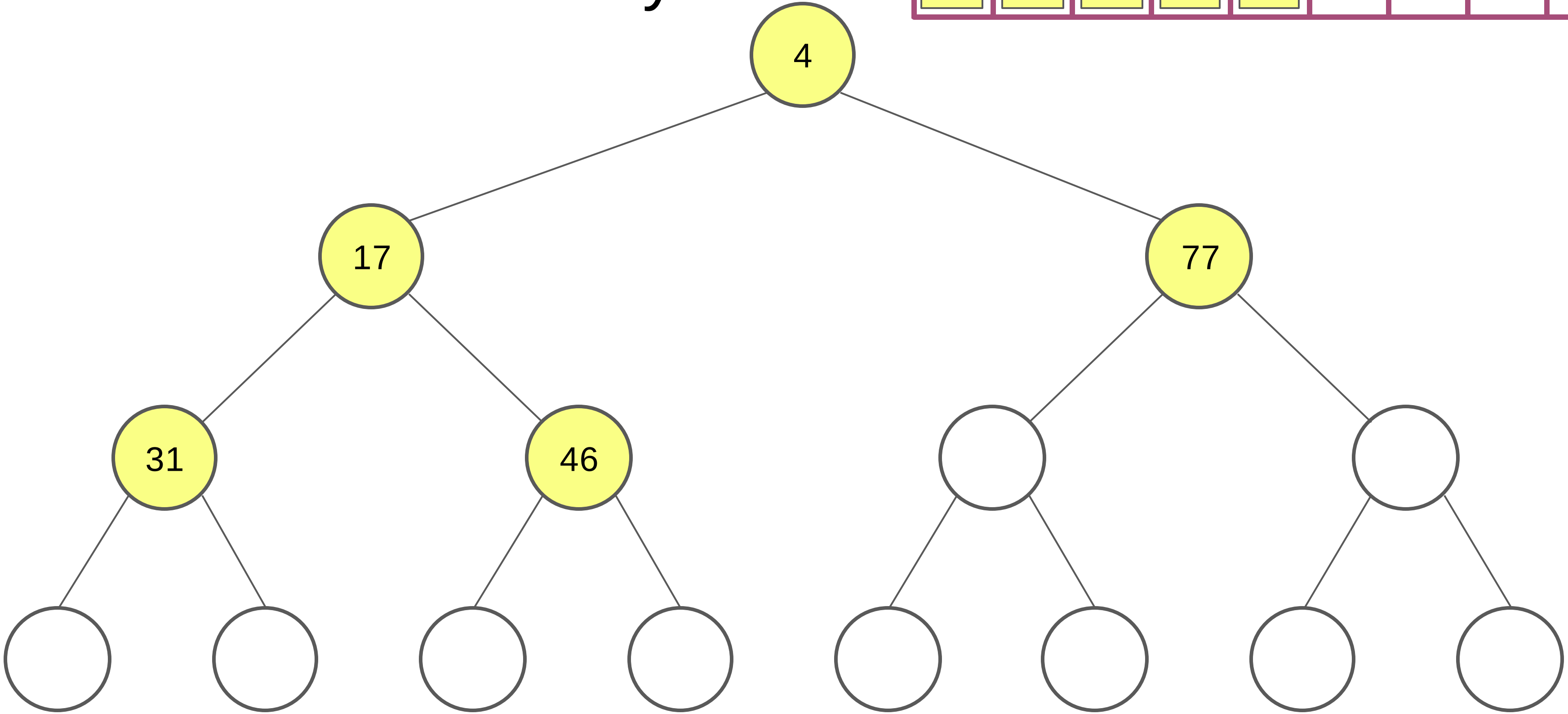


Tree stored as an array

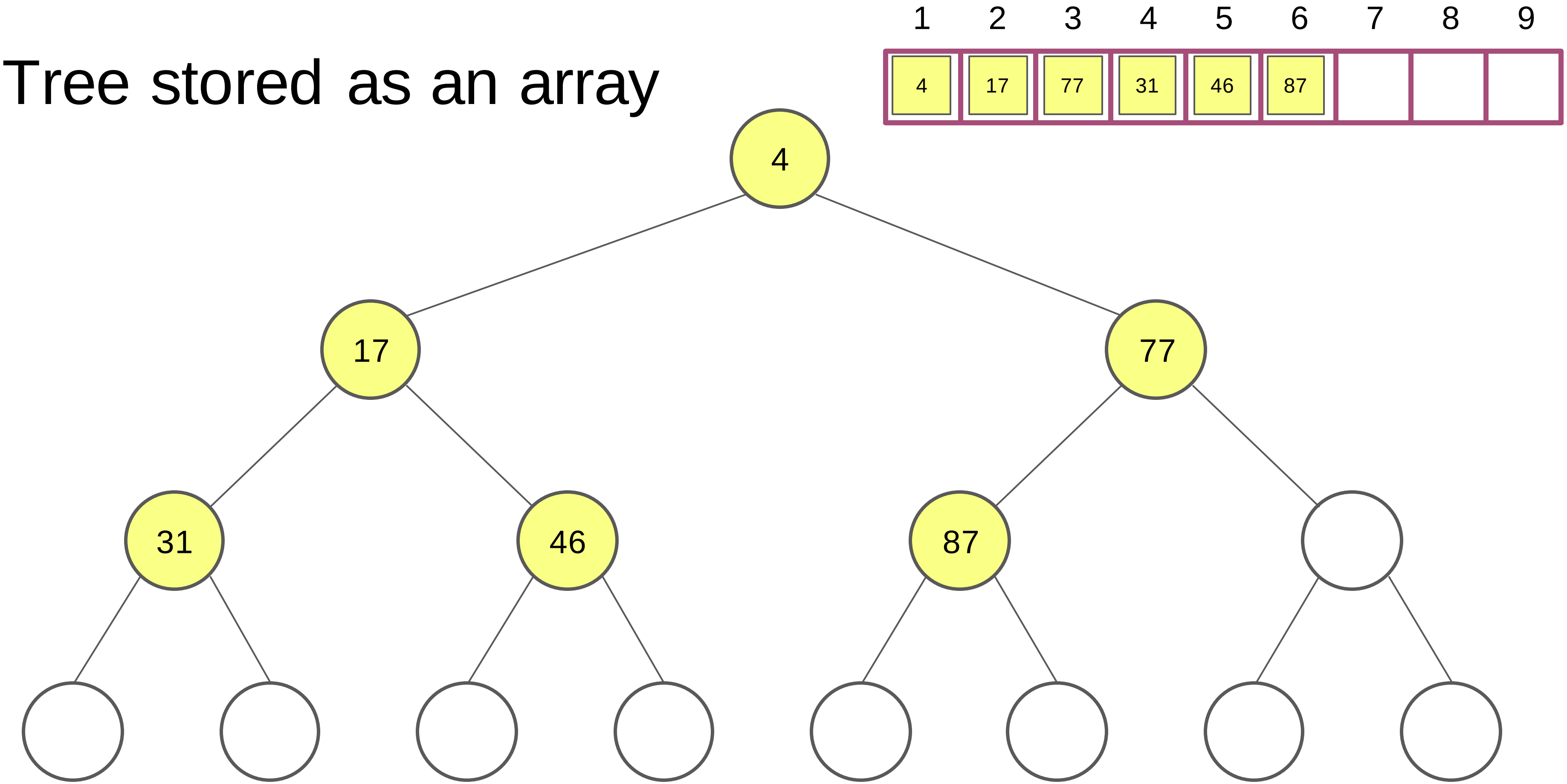


Tree stored as an array

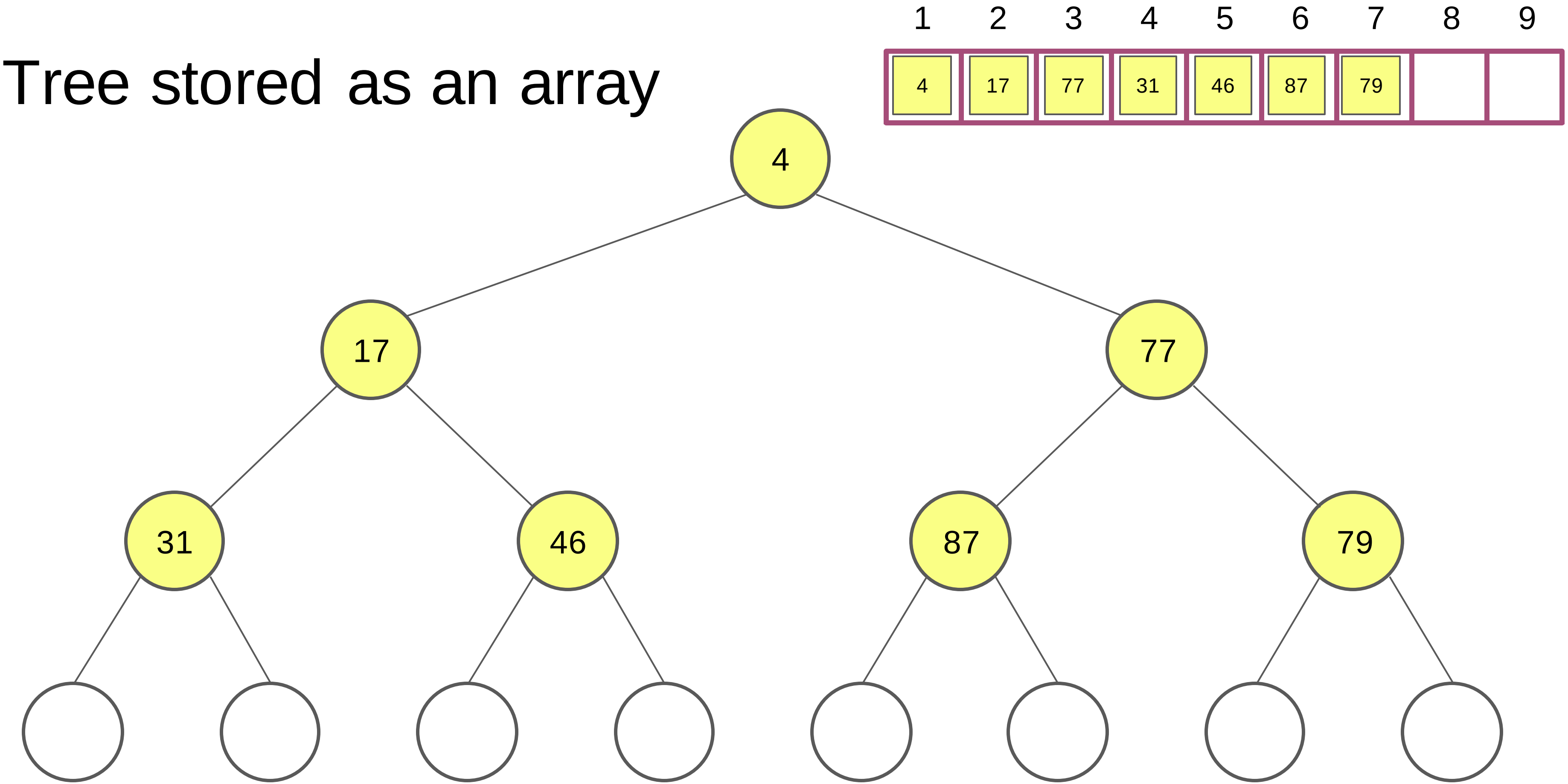




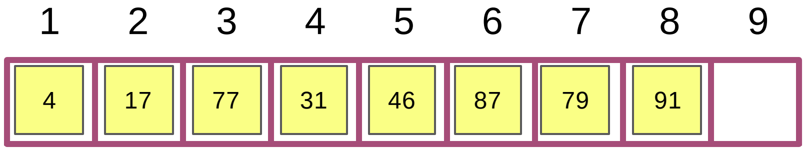
Tree stored as an array

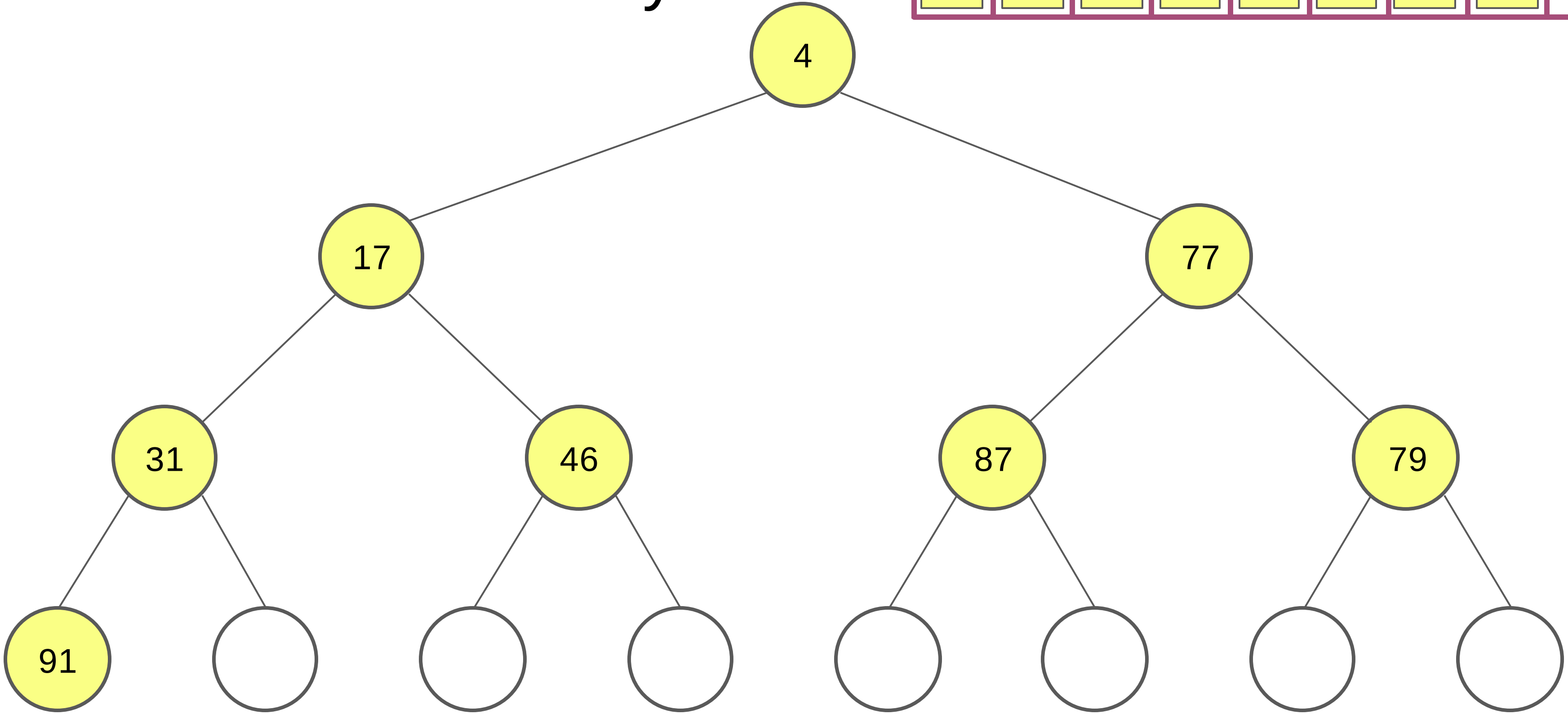


Tree stored as an array

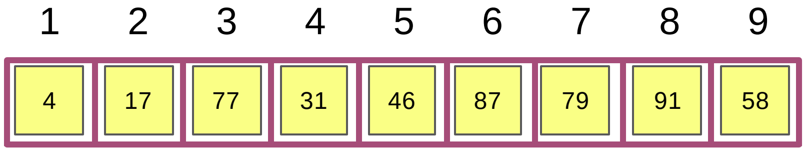


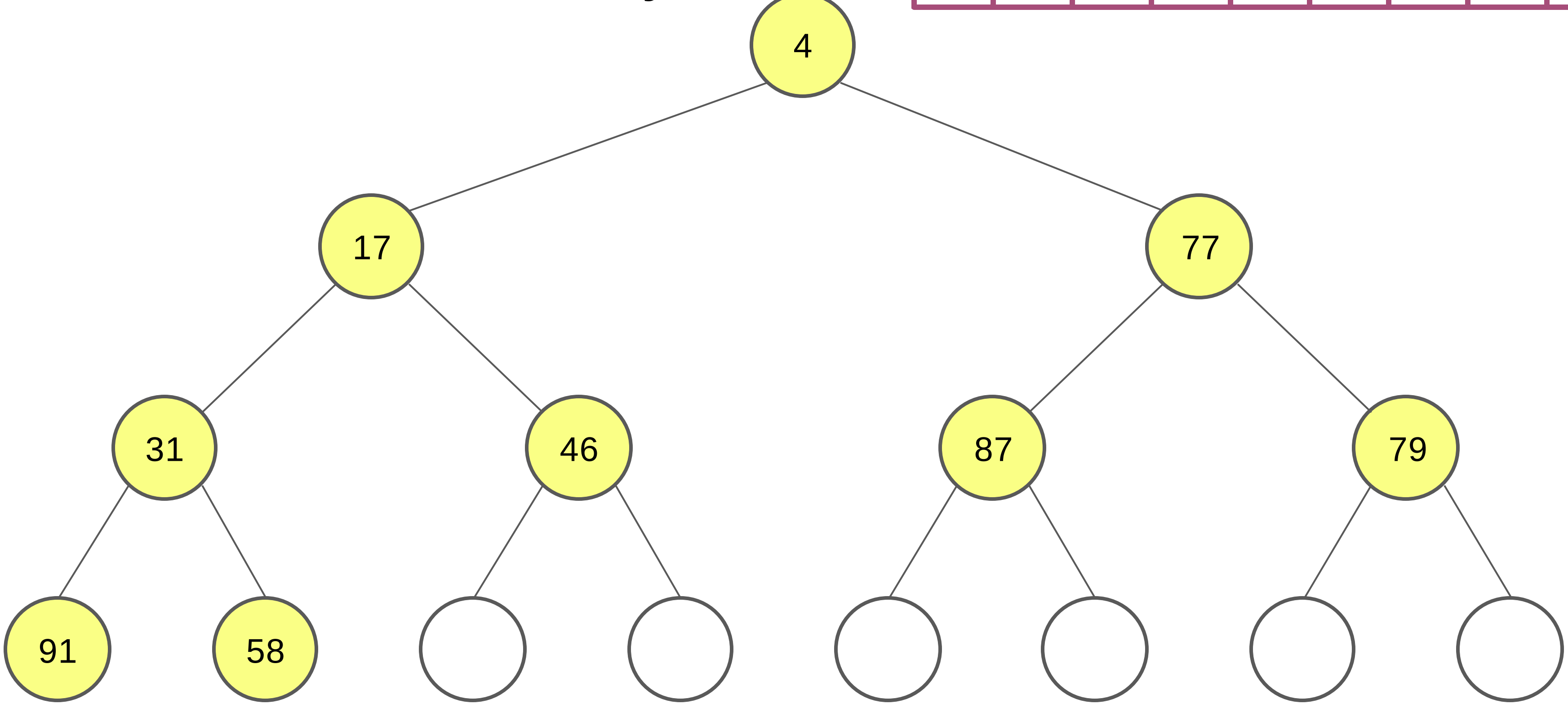
Tree stored as an array





Tree stored as an array





Lab 3: Implement Heap

For studio

Some possible implementations

- Let’s think through some implementation possibilities
  - Using data structures we already know
  - Reasoning about their complexity
- The “n” here
  - Means the current size of our priority queue
- Given a priority queue of n items
  - How expensive is each of the methods we have described so far
  - For a particular implementation
    - Binary heap
    - List
      - Links vs. array
      - Ordered vs. not ordered

Running example

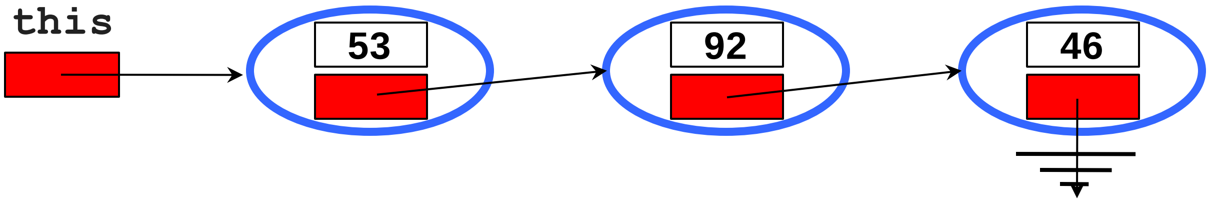
Implementation	insert	extractMin
Unordered list		
Ordered list		
Unordered array		
Ordered array		

- Table will track complexity
- We are interested in *worst-case* times
  - We'll come back to this shortly

Running example

PQ contains 53, 92, 46

Implementation	insert	extractMin
Unordered list		
Ordered list		
Unordered array		
Ordered array		

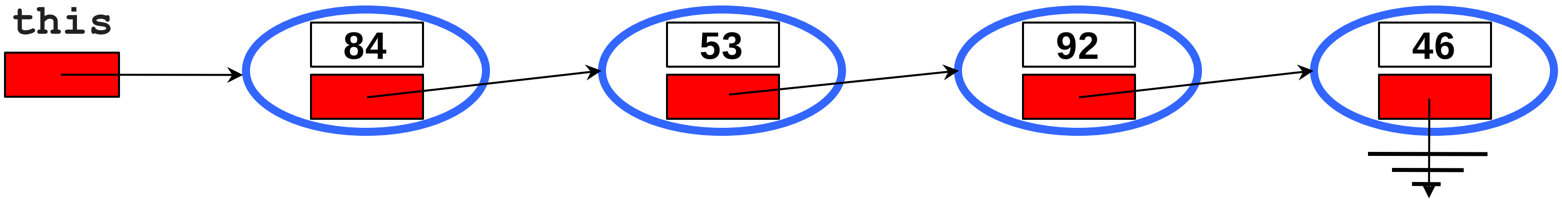


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list		
Ordered list		
Unordered array		
Ordered array		

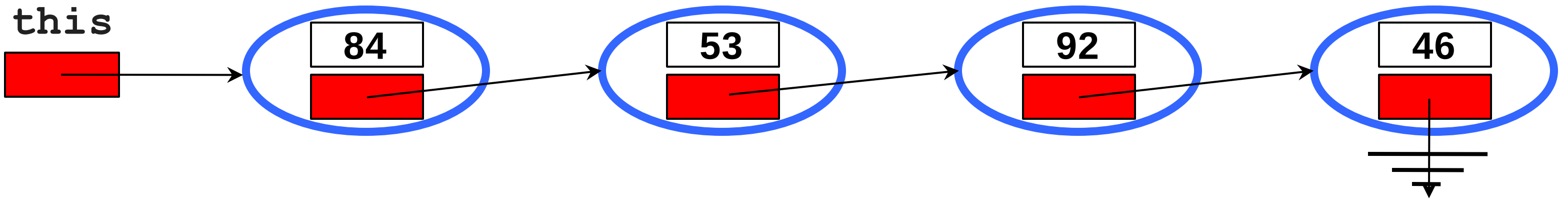


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	
Ordered list		
Unordered array		
Ordered array		

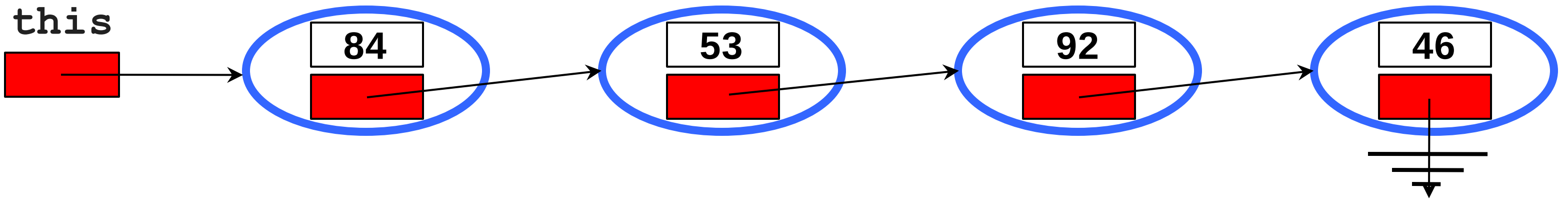


Running example

PQ contains 53, 92, 46

extractMin()   84

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	
Ordered list		
Unordered array		
Ordered array		

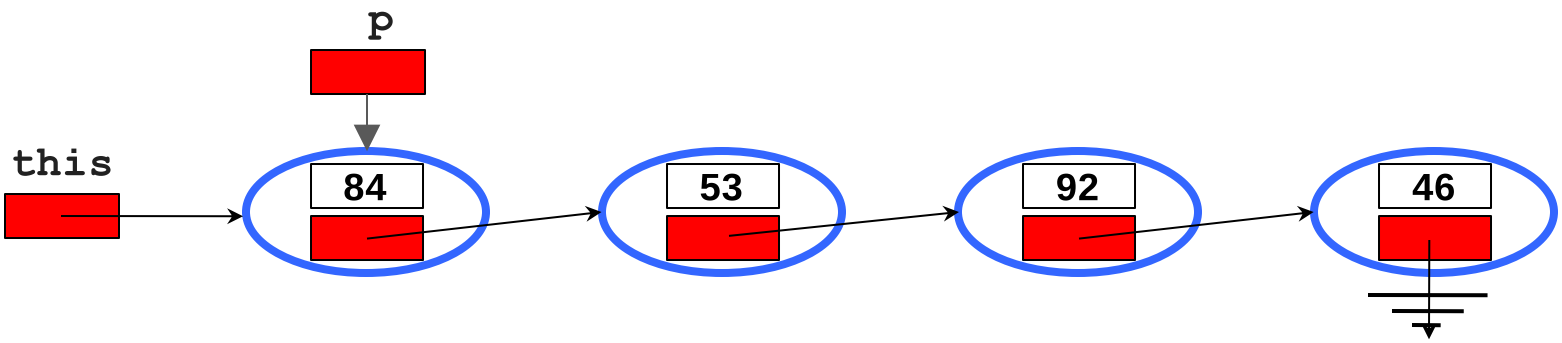


Running example

PQ contains 53, 92, 46

extractMin()   84

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	
Ordered list		
Unordered array		
Ordered array		

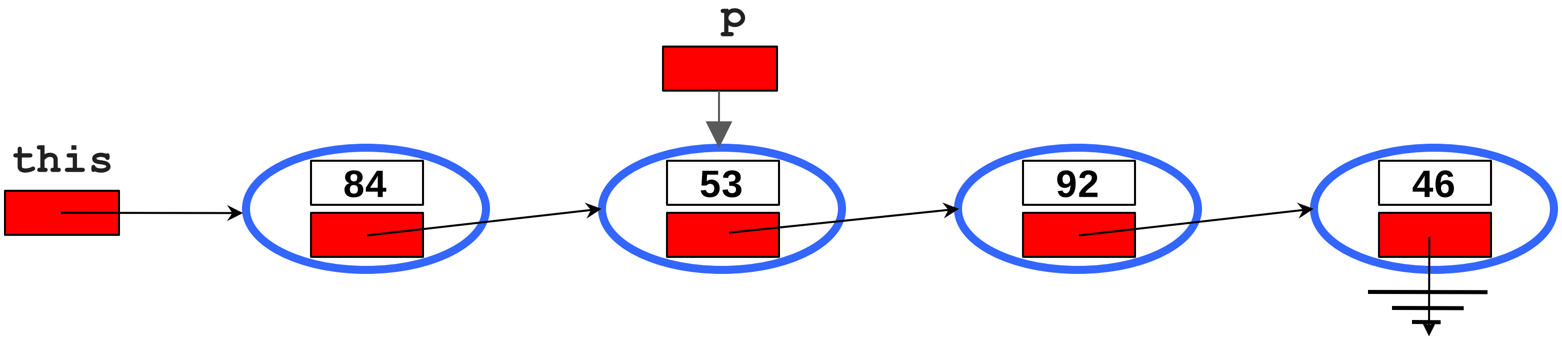


Running example

PQ contains 53, 92, 46

extractMin() 53

Implementation	insert	extractMin
Unordered list	Θ(1)	
Ordered list		
Unordered array		
Ordered array		

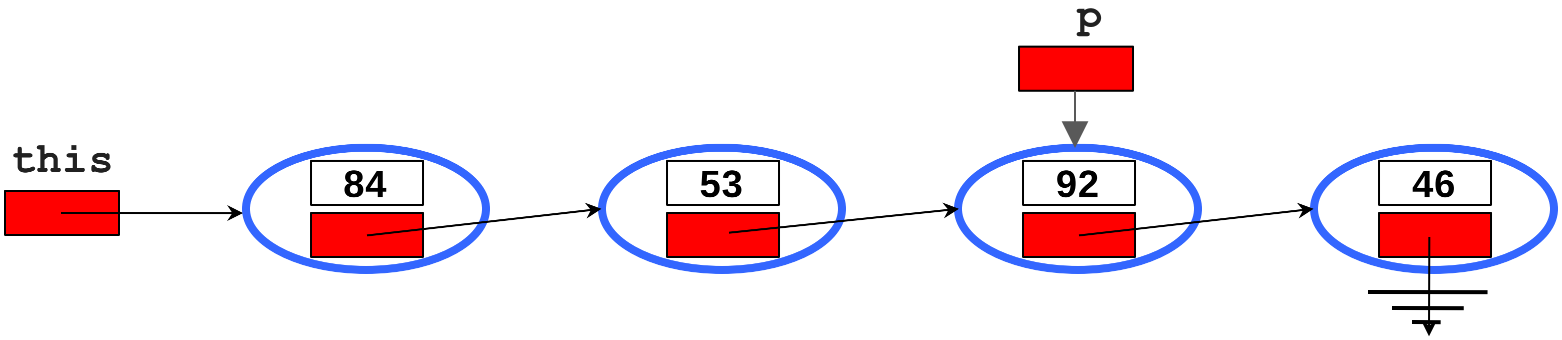


Running example

PQ contains 53, 92, 46

extractMin()   53

Implementation	insert	extractMin
Unordered list	Θ(1)	
Ordered list		
Unordered array		
Ordered array		

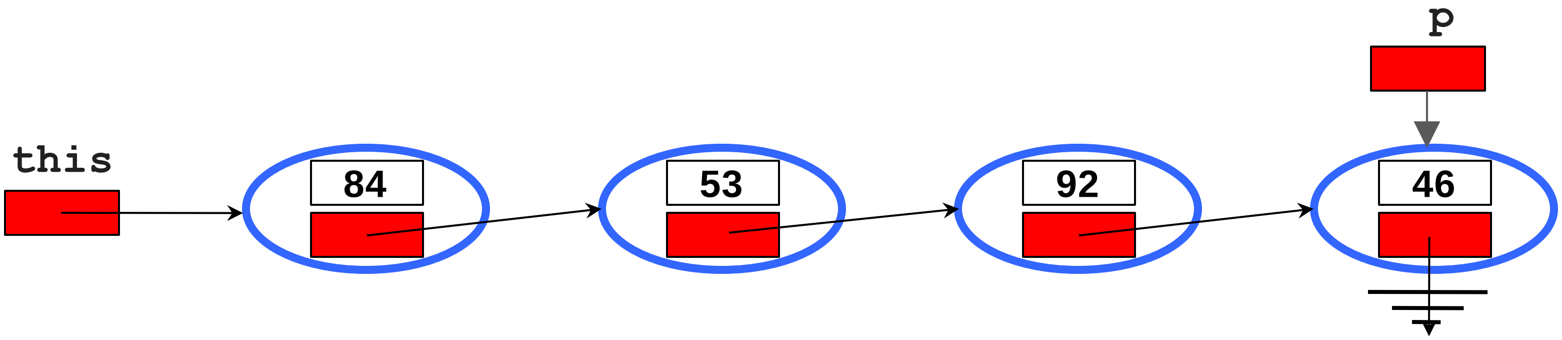


Running example

PQ contains 53, 92, 46

extractMin()   46

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	
Ordered list		
Unordered array		
Ordered array		

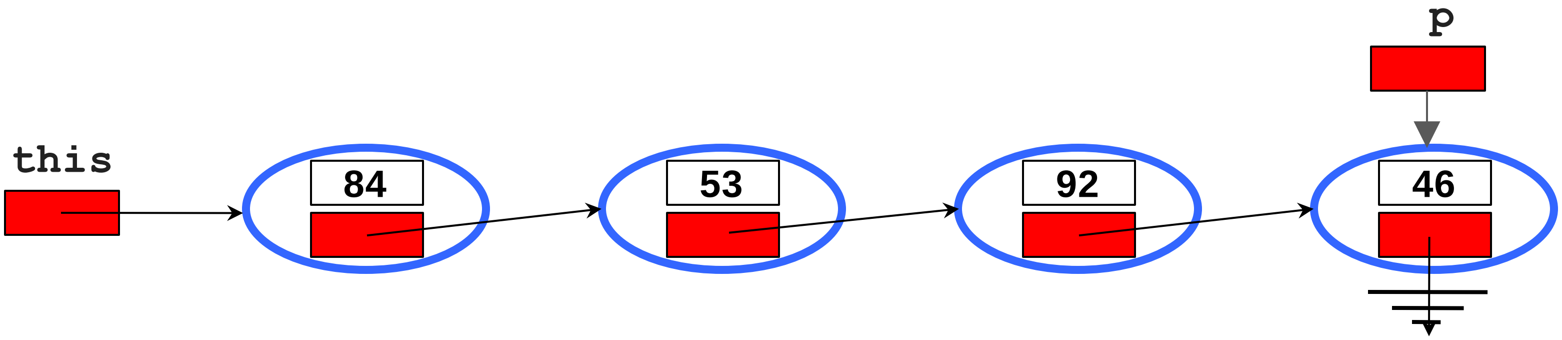


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	
Ordered list		
Unordered array		
Ordered array		

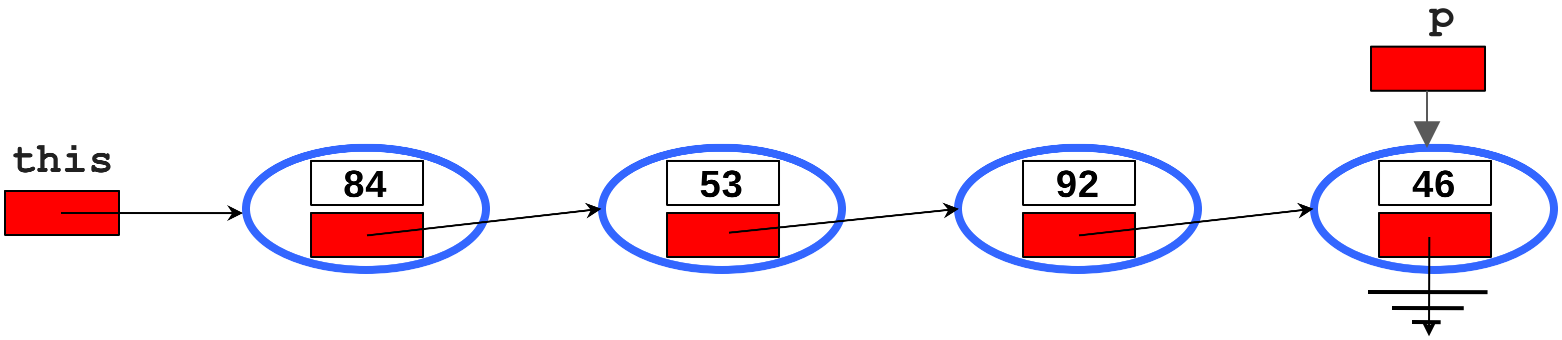


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list		
Unordered array		
Ordered array		

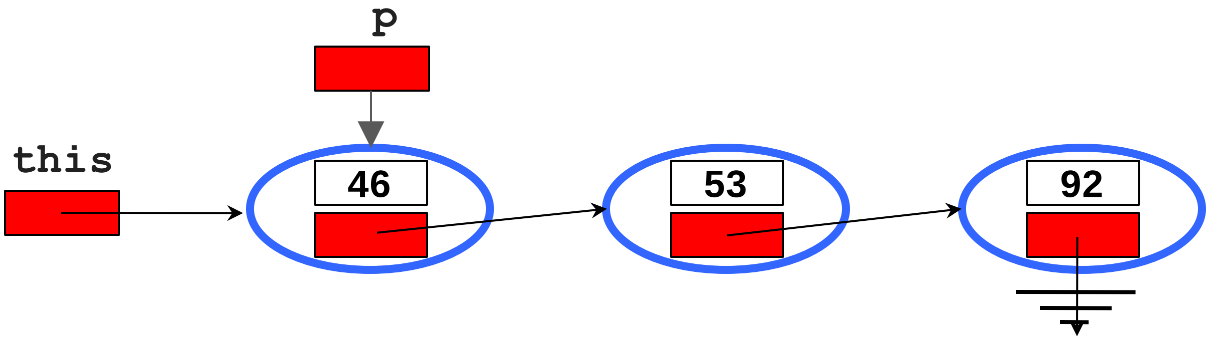


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list		
Unordered array		
Ordered array		

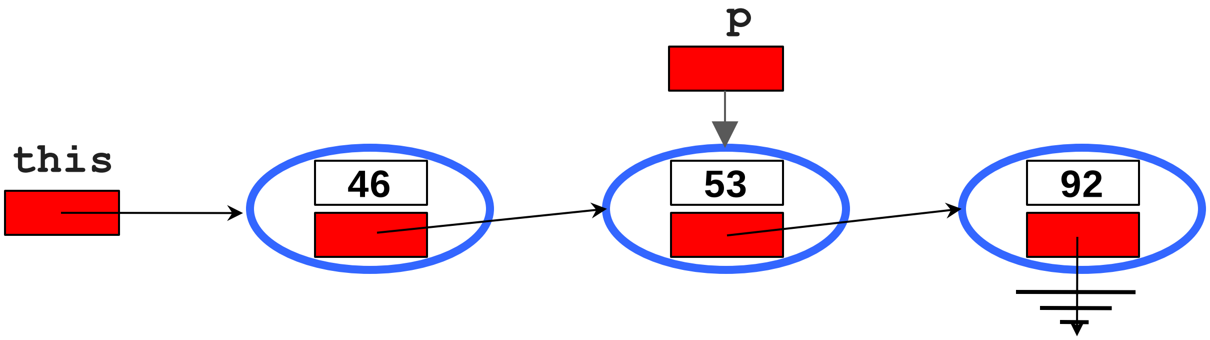


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list		
Unordered array		
Ordered array		

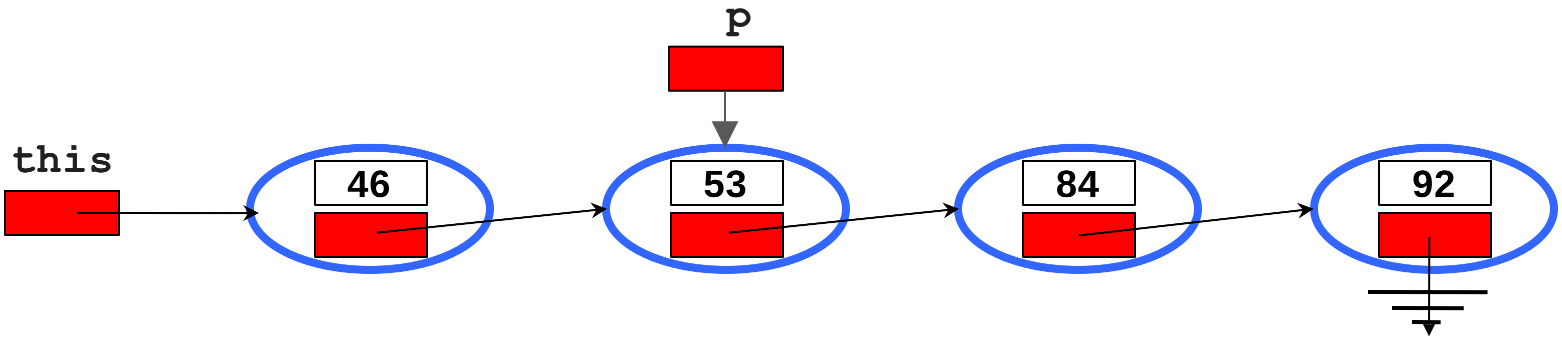


Running example

PQ contains 53, 92, 46

insert(84)

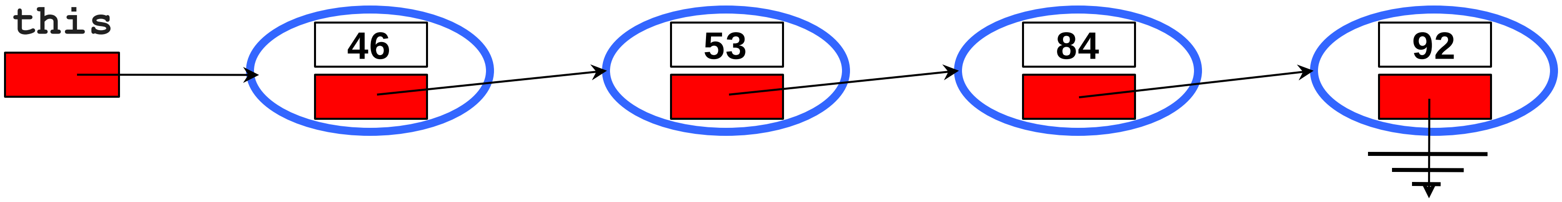
Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		



Wait! Is this right? Look at what happens next if we wanted to insert “1” into the ordered list

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

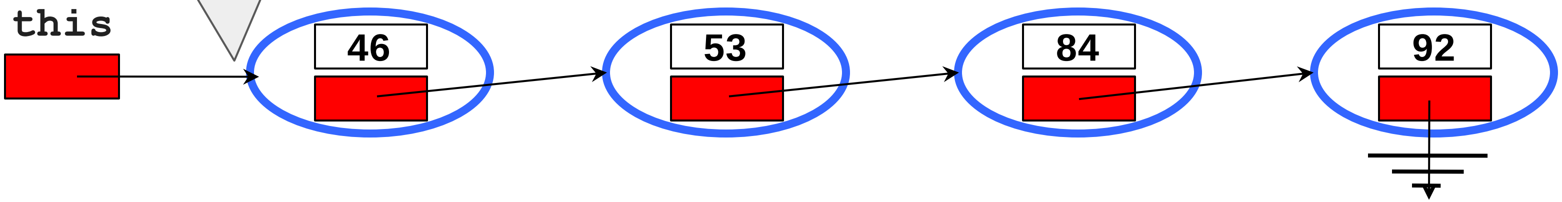


insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

Wait! Is this right? Look at what happens next if we wanted to insert "1" into the ordered list...

It would go here, seemingly taking constant time!



P

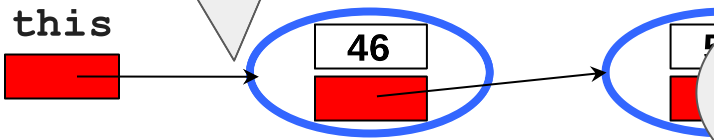
P

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

Wait! Is this right? Look at what happens next if we wanted to insert "1" into the ordered list...

It would go here, seemingly taking constant time!



Remember we set out to analyze the complexity of the *worst-case*. That complexity is as shown here, bound above and below by n

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$
- To avoid this
  - First think about the function f(n) that characterizes the property of interest
    - Worst-case
    - Best-case
    - Average-case

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$
- To avoid this
  - First think about the function f(n) that characterizes the property of interest
    - Worst-case
    - Best-case
    - Average-case
  - Then think about whether that f(n) is bounded

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$
- To avoid this
  - First think about the function f(n) that characterizes the property of interest
    - Worst-case
    - Best-case
    - Average-case
  - Then think about whether that f(n) is bounded
    - From above $O(\cdots)$

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$
- To avoid this
  - First think about the function f(n) that characterizes the property of interest
    - Worst-case
    - Best-case
    - Average-case
  - Then think about whether that f(n) is bounded
    - From above $O(\cdots)$
    - From below $\Omega(\cdots)$

A common source of confusion

- Many of us confuse
  - Best vs. worst case
  - $\Omega(\cdots)$ vs. $O(\cdots)$
- To avoid this
  - First think about the function f(n) that characterizes the property of interest
    - Worst-case
    - Best-case
    - Average-case
  - Then think about whether that f(n) is bounded
    - From above $O(\cdots)$
    - From below $\Omega(\cdots)$
    - Both $\Theta(\cdots)$

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

Wait! Is this right? Look at what happens next if we wanted to insert "1" into the ordered list…

insert(84)

It would go here, seemingly taking constant time!

Remember we set out to analyze the complexity of the *worst-case*. That complexity is as shown here, bound above and below by n

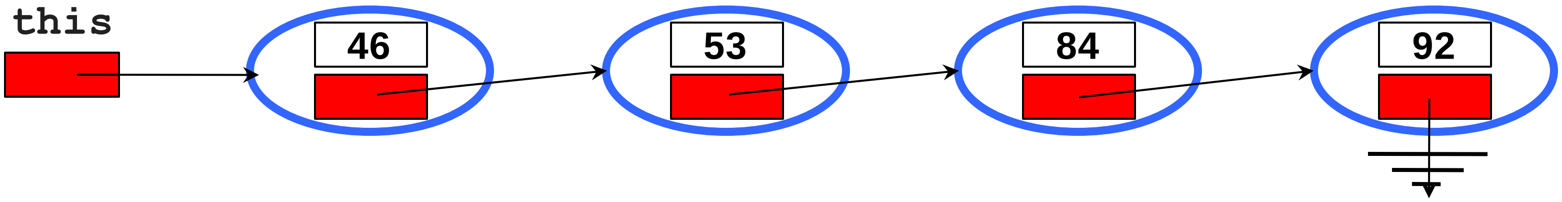
R

Just be sure, why is this right?

PQ

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		



R

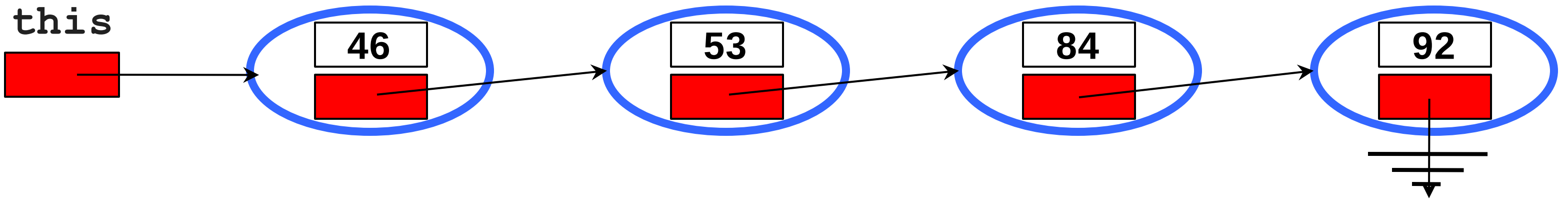
Just be sure, why is
this right?

And this?

PQ

insert(84)

Implementation	insert	extractMin
[illegible]	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

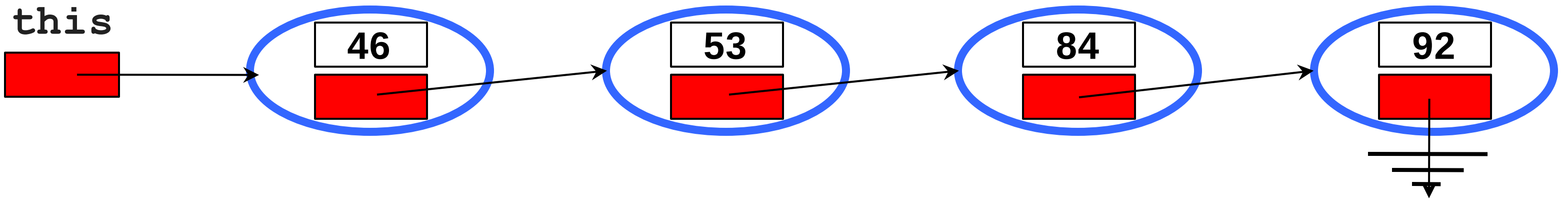


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

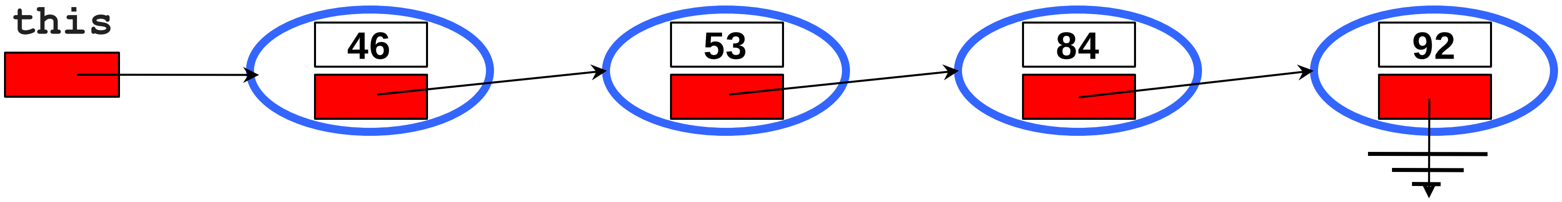


Running example

PQ contains 53, 92, 46

extractMin()

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

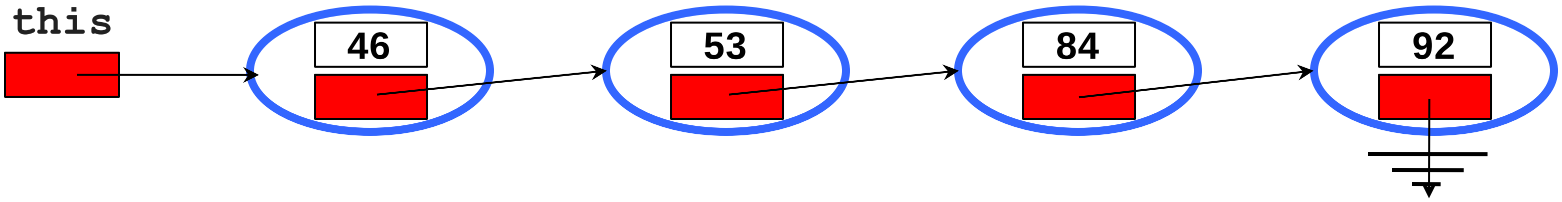


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

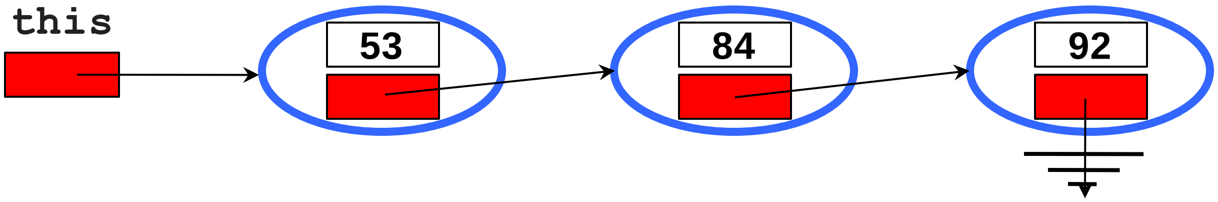


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	
Unordered array		
Ordered array		

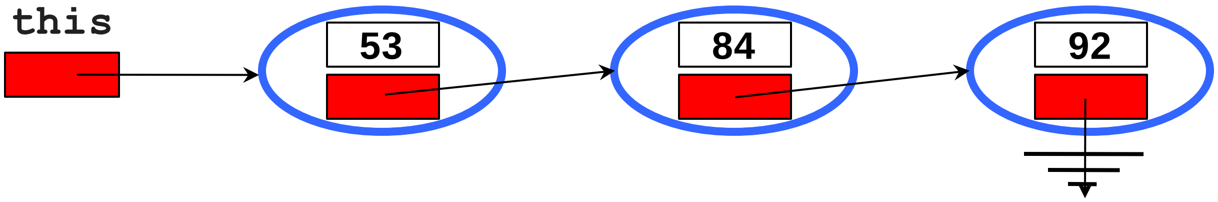


Running example

PQ contains 53, 92, 46

extractMin() 46

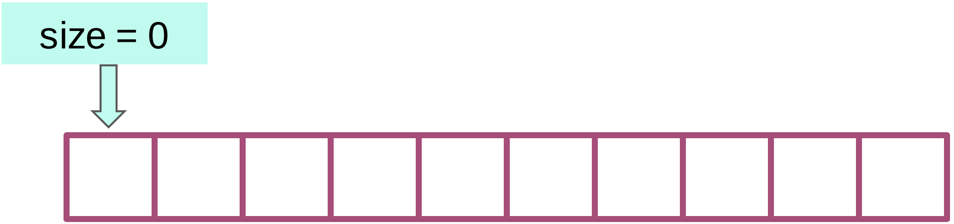
Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array		
Ordered array		



What about arrays?

Running example

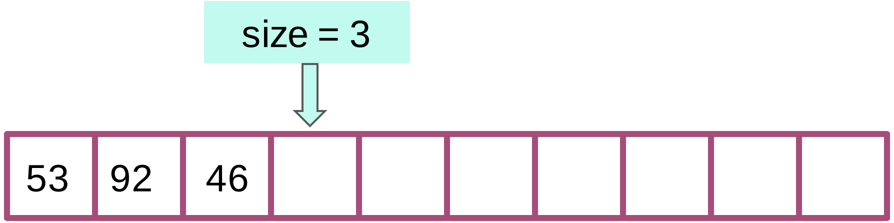
Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array		
Ordered array		



Running example

PQ contains 53, 92, 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array		
Ordered array		

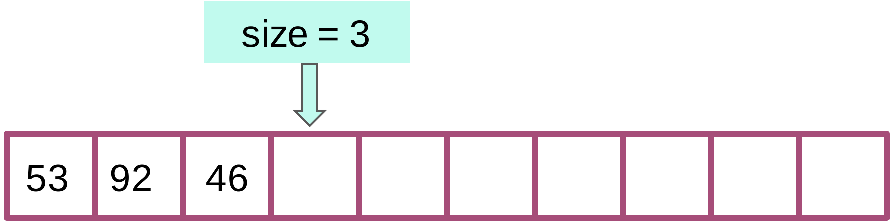


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array		
Ordered array		

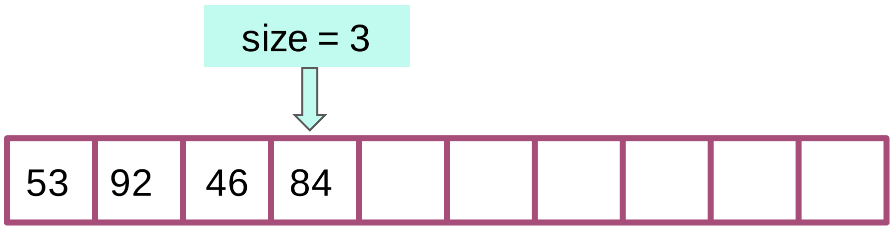


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array		
Ordered array		

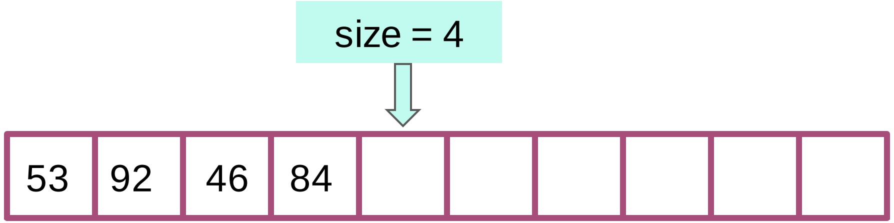


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	$\Theta(n)$
Ordered list	$\Theta(n)$	$\Theta(1)$
Unordered array		
Ordered array		

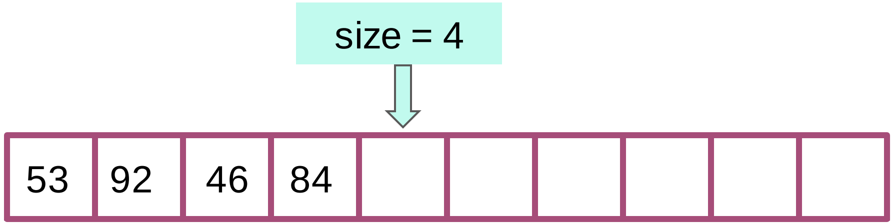


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		



Running example

PQ contains 53, 92, 46

extractMin()

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

size = 4

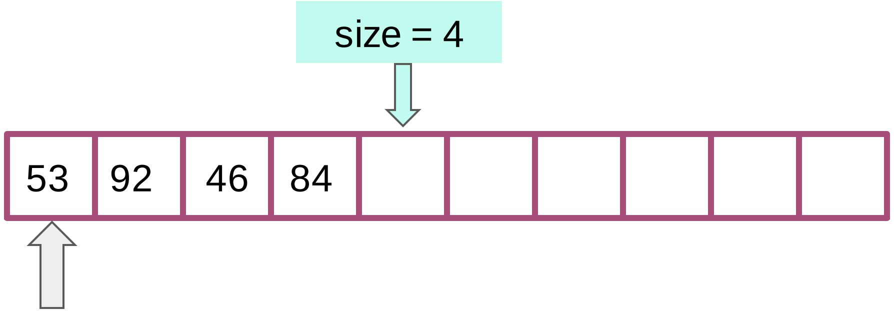
53	92	46	84						

Running example

PQ contains 53, 92, 46

extractMin()

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

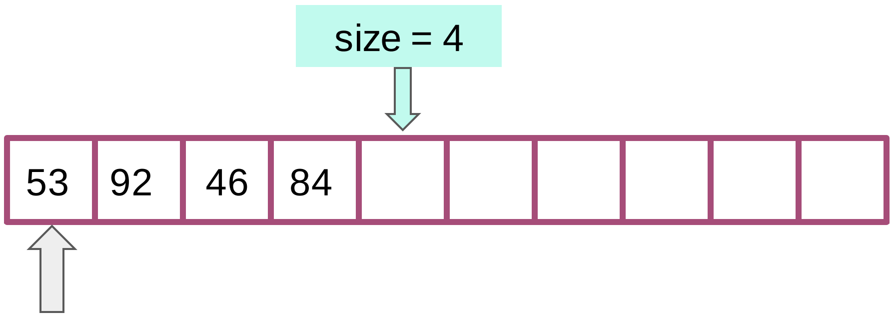


Running example

PQ contains 53, 92, 46

extractMin()   53

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

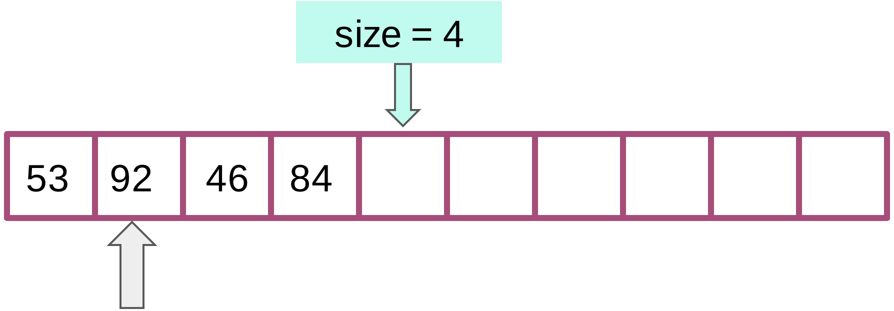


Running example

PQ contains 53, 92, 46

extractMin()   53

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

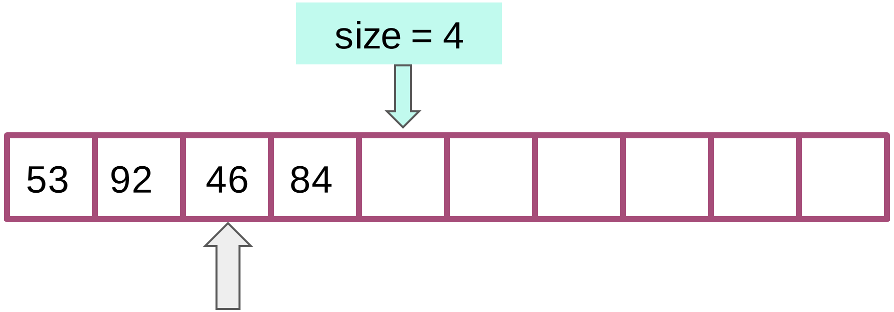


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

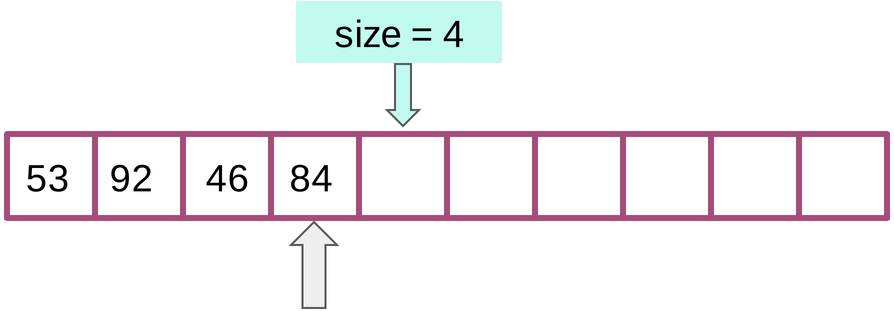


Running example

PQ contains 53, 92, 46

extractMin()   46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

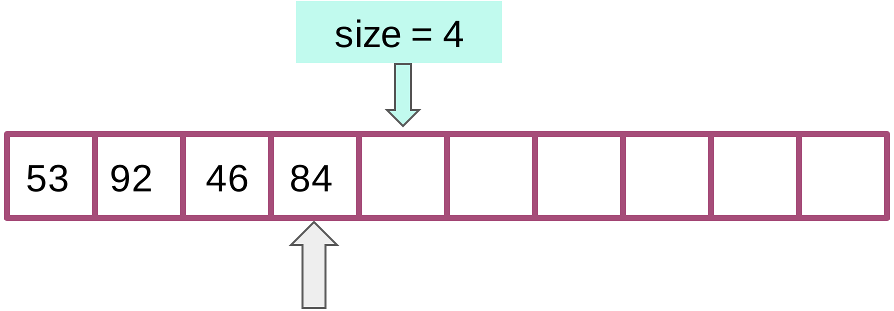


Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	
Ordered array		

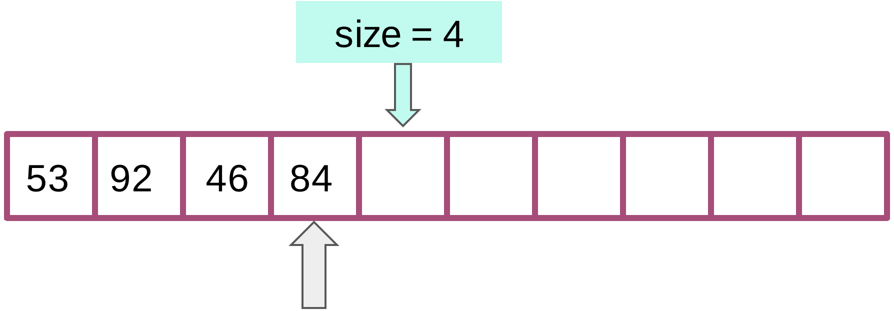


Running example

PQ contains 53, 92, 46

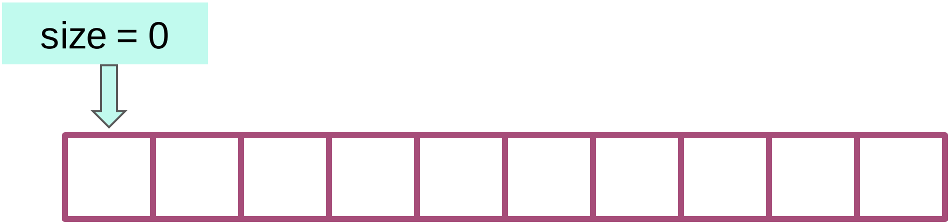
extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		



Running example

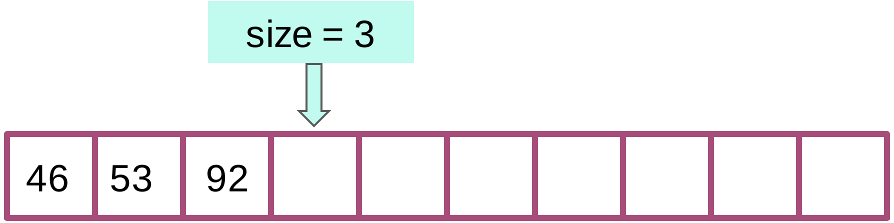
Implementation	insert	extractMin
Unordered list	$\Theta(1)$	$\Theta(n)$
Ordered list	$\Theta(n)$	$\Theta(1)$
Unordered array	$\Theta(1)$	$\Theta(n)$
Ordered array		



Running example

PQ contains 53, 92, 46

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	$\Theta(n)$
Ordered list	$\Theta(n)$	$\Theta(1)$
Unordered array	$\Theta(1)$	$\Theta(n)$
Ordered array		

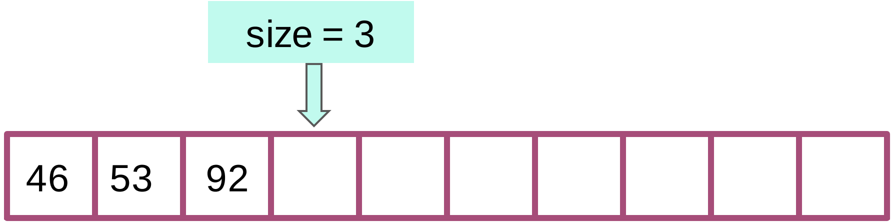


Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		



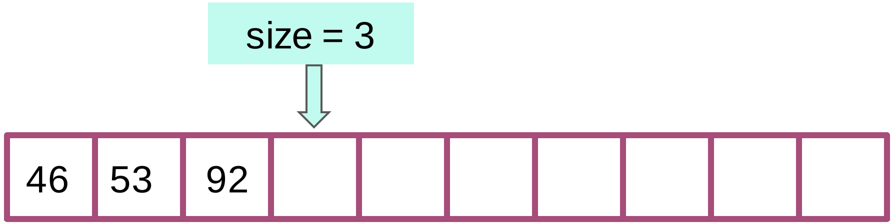
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go



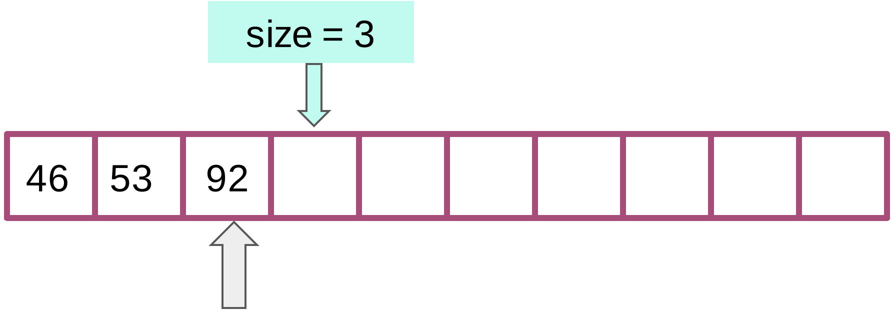
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go



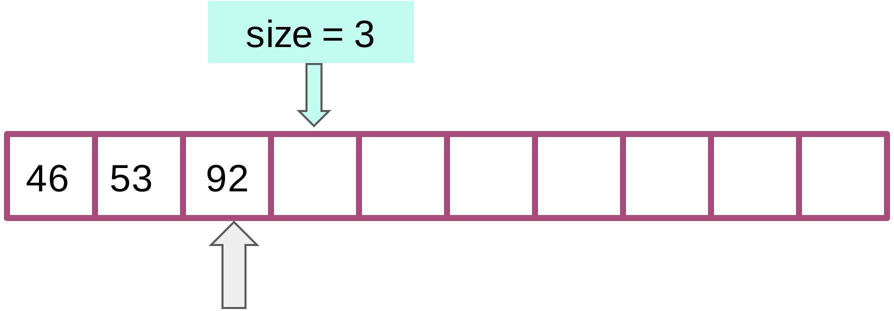
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84



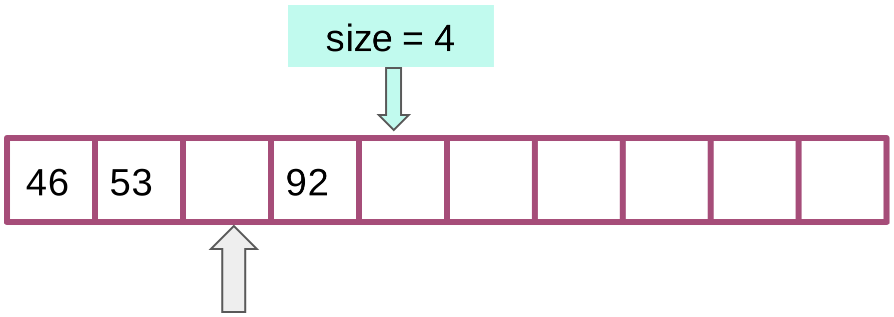
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84



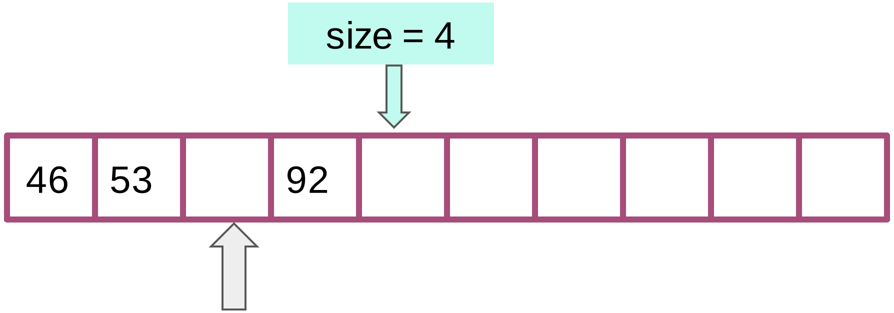
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84
3. Insert 84



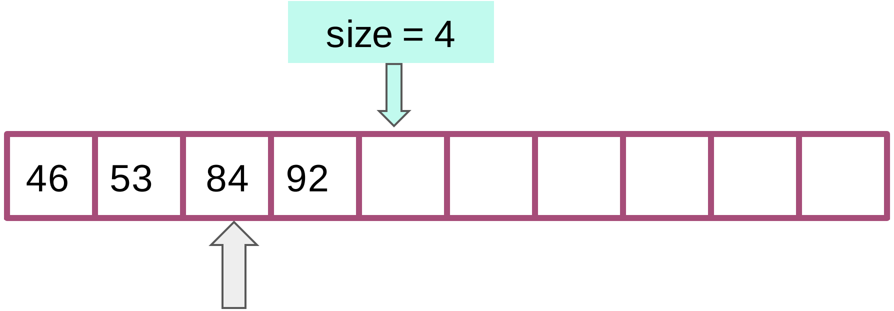
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84
3. Insert 84



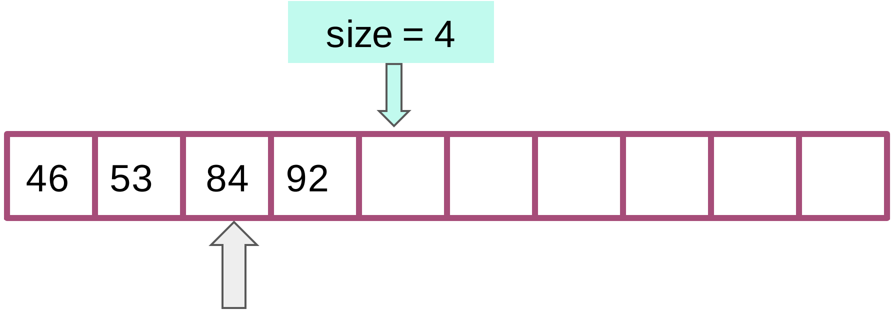
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	$\Theta(1)$	$\Theta(n)$
Ordered list	$\Theta(n)$	$\Theta(1)$
Unordered array	$\Theta(1)$	$\Theta(n)$
Ordered array	????	

1. Find where 84 should go
2. Move elements to make room for 84
3. Insert 84



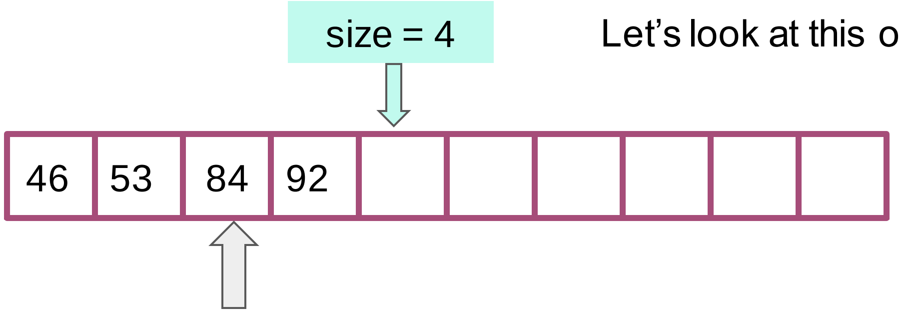
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	?????	

1. Find where 84 should go
2. Move elements to make room for 84
3. Insert 84



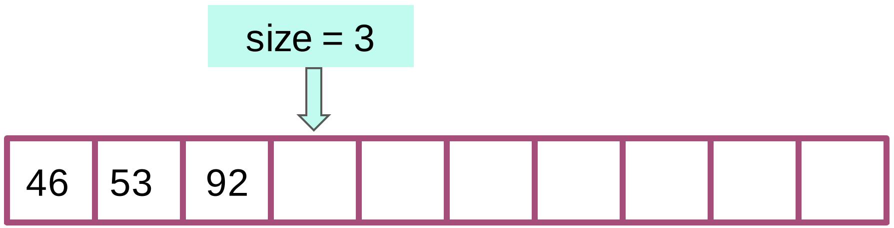
Let's look at this one step at a time

Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		



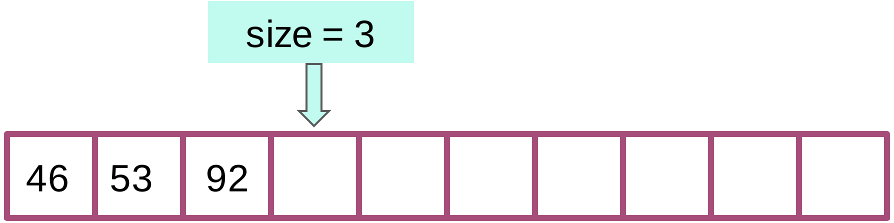
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go



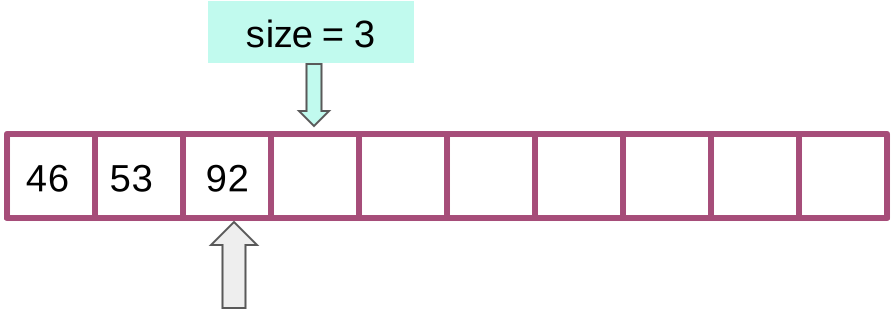
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go



Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
		Θ(n)
		Θ(1)
		Θ(n)
Ordered array		

What say you?

1. Find where 84 should go

size = 3

46	53	92							

Running example

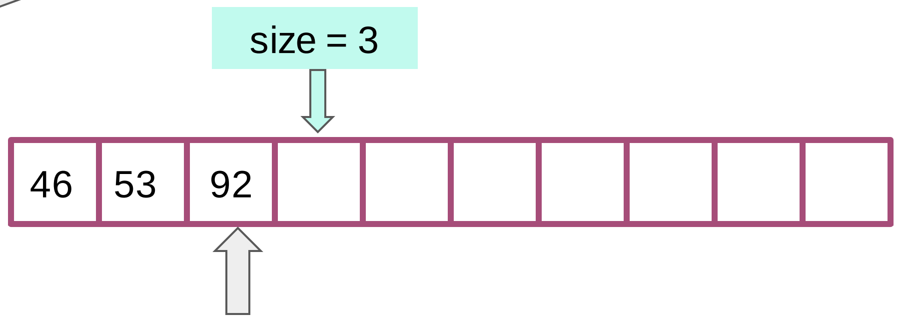
PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
		Θ(n)
		Θ(1)
	(1)	Θ(n)
Ordered array		

What say you? We could look through the entire array O(n)

1. Find where 84 should go



Running example

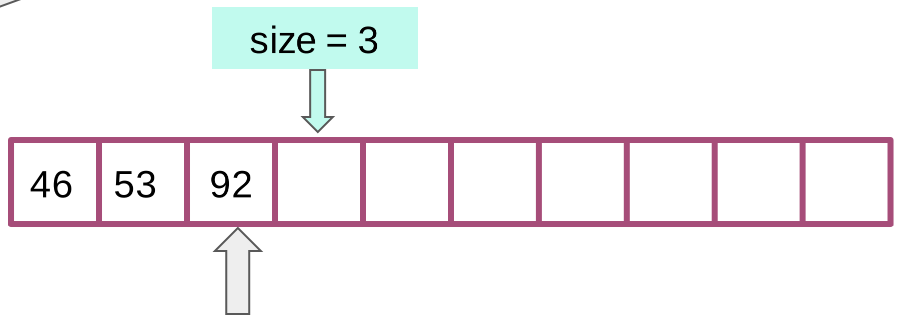
PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
[illegible]	[illegible]	Θ(n)
[illegible]	[illegible]	Θ(1)
[illegible]	(1)	Θ(n)
Ordered array		

What say you? We could look through the entire array O(n) but there is a faster way, do you see?

1. Find where 84 should go



Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
		Θ(n)
		Θ(1)
	(1)	Θ(n)
Ordered array		

What say you? We could look through the entire array O(n) but there is a faster way, do you see?

1. Find where 84 should go

size = 3

46	53	92							

Remember phone books? To find somebody, would you start at the first page and keep looking?

Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
	[illegible]	Θ(n)
		Θ(1)
	[illegible]	Θ(n)
Ordered array		

What say you? We could look through the entire array O(n) but there is a faster way, do you see?

1. Find where 84 should go

size = 3

46	53	92							

Remember phone books? To find somebody, would you start at the first page and keep looking? Or is there a faster way?

One method for searching a phone book efficiently

- Look in the middle
  - Is the target of your search later or earlier than what you find?
  - Throw away the half of the phone book that cannot contain your target
    - Really, throw it away, or shred it, or burn it
  - Repeat this procedure (recursively!) on the half you did not throw away
- We will soon study a general method to reason about this procedure's complexity
- But do you see what it is?
  - Think about the size of what remains to be searched
  - When it reaches 1 you are done
- $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ 3 steps $\log_2 8$

One method for searching a phone book efficiently

- Look in the middle
    - Is the target of your search later or earlier than what you find?
    - Throw away the half of the phone book that cannot contain your target
        - Really, throw it away, or shred it, or burn it
    - Repeat this procedure (recursively!) on the half you did not throw away
- We will soon study a general method to reason about this procedure's complexity
- But do you see what it is?
    - Think about the size of what remains to be searched
    - When it reaches 1 you are done
- $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ 3 steps $\log_2 8$
- Because these logarithms are so common, we abbreviate them using "lg"

One method for searching a phone book efficiently

- Look in the middle
  - Is the target of your search later or earlier than what you find?
  - Throw away the half of the phone book that cannot c
    - Really, throw it away, or shred it, or burn it
  - Repeat this procedure (recursively!) on the half yo
- We will soon study a general method to reason
  complexity
- But do you see what it is?
  - Think about the size of what remains to be searched
  - When it reaches 1 you are done
- $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ 3 steps $\log_2 8$
- Because these logarithms are so common, we abbreviate them using “lg”

However, asymptotically the base doesn’t matter, so we write O(log(n))

Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go

O(log(n))

size = 3

46	53	92							

Remember phone books? To find somebody, would you start at the first page and keep looking? Or is there a faster way?

Running example

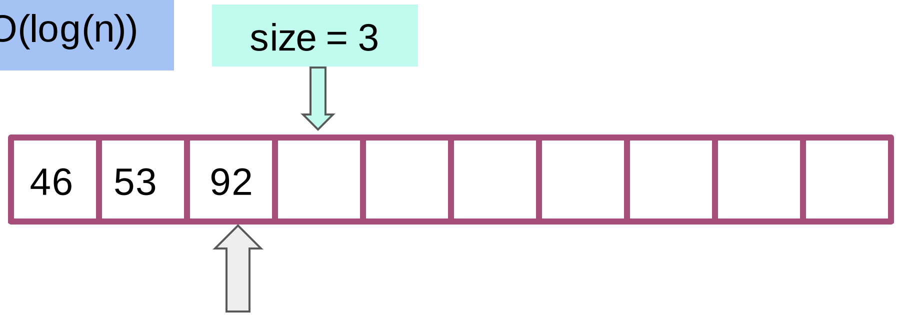
PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84

O(log(n))



Running example

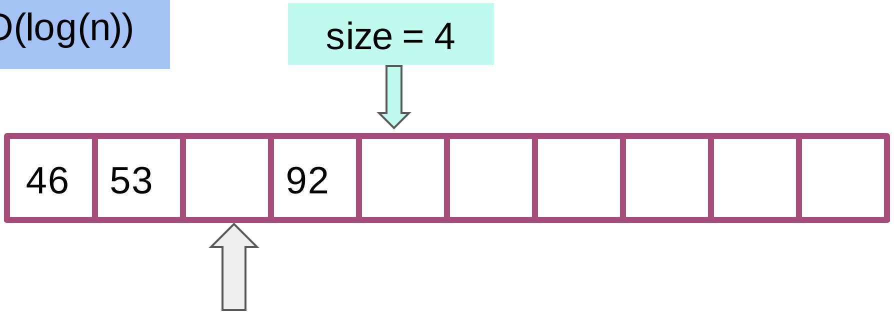
Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

PQ contains 53, 92, 46

insert(84)

1. Find where 84 should go
2. Move elements to make room for 84

O(log(n))



Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Un		Θ(n)
		1)
		Θ(n)
Order		

Here we only had to move one element, but worst case is that they all have to shift right by one cell.

1. Find where 84 should go
2. Move elements to make room for 84

O(log(n))

size

46	53		92						

Running example

PQ contains 53, 92, 46

insert(84)

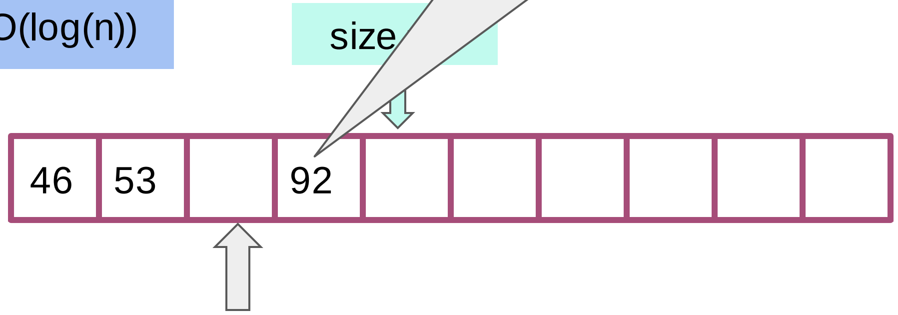
Implementation	insert	extractMin
Un		Θ(n)
		1)
		Θ(n)
Order		

Here we only had to move one element, but worst case is that they all have to shift right by one cell.

1. Find where 84 should go
2. Move elements to make room for 84

O(log(n))

O(n)



Running example

PQ contains 53, 92, 46

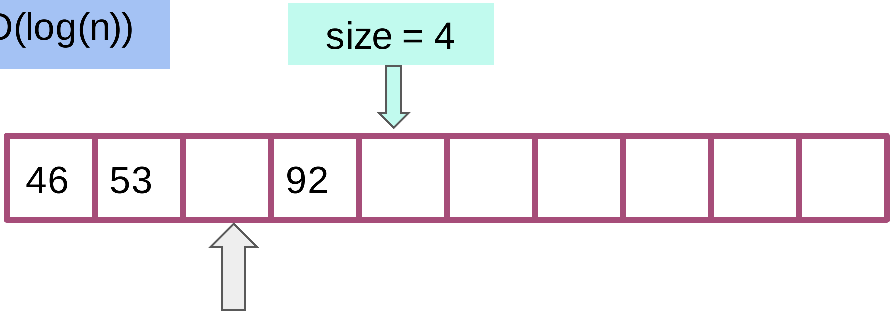
insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go
2. Move elements to make room for 84
3. Insert 84

O(log(n))

O(n)



Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

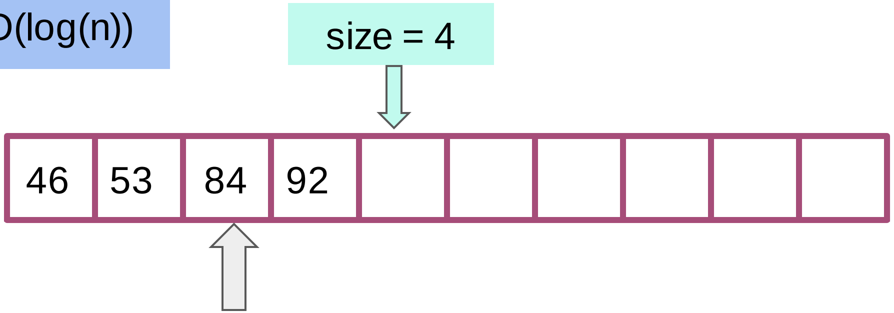
1. Find where 84 should go

O(log(n))

2. Move elements to make room for 84

O(n)

3. Insert 84



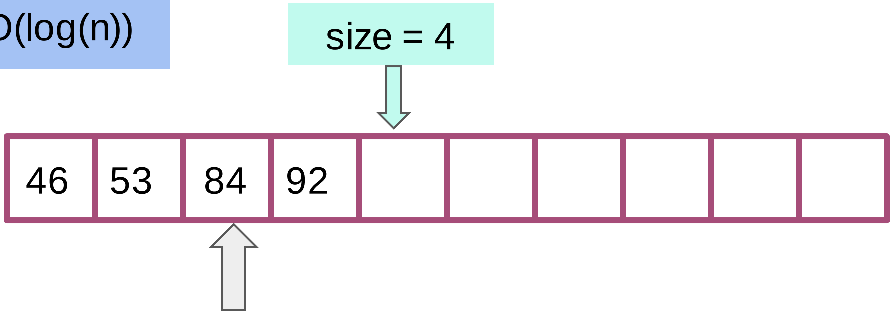
Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go — O(log(n))
2. Move elements to make room for 84 — O(n)
3. Insert 84 — O(1)



Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array		

1. Find where 84 should go — O(log(n))
2. Move elements to make room for 84 — O(n)
3. Insert 84 — O(1)

size = 4

46	53	84	92						

O(log(n)) + O(n) + O(1) = O(n)

Running example

PQ contains 53, 92, 46

insert(84)

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Find where 84 should go — O(log(n))
2. Move elements to make room for 84 — O(n)
3. Insert 84 — O(1)

size = 4

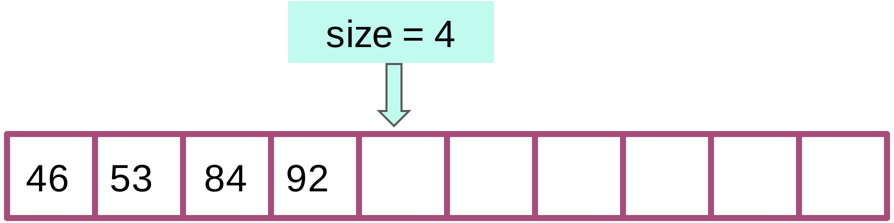
46	53	84	92						

O(log(n)) + O(n) + O(1) = O(n)

Running example

PQ contains 53, 92, 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

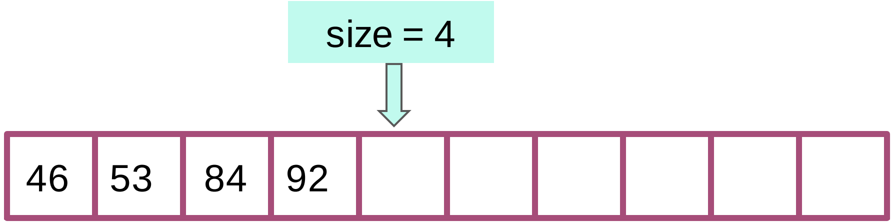


Running example

PQ contains 53, 92, 46

extractMin()

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	



Running example

PQ contains 53, 92, 46

extractMin()

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element

size = 4

46	53	84	92						

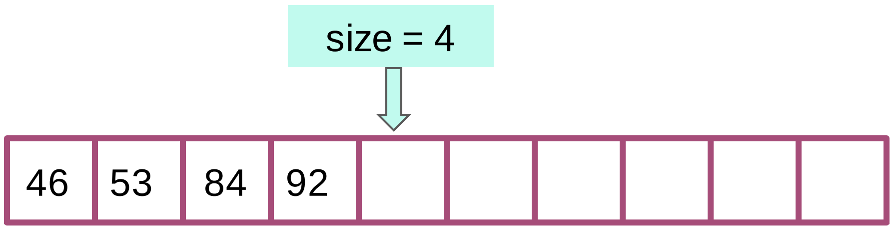
Running example

PQ contains 53, 92, 46

extractMin() 46

1. Capture the array's first element

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	



Running example

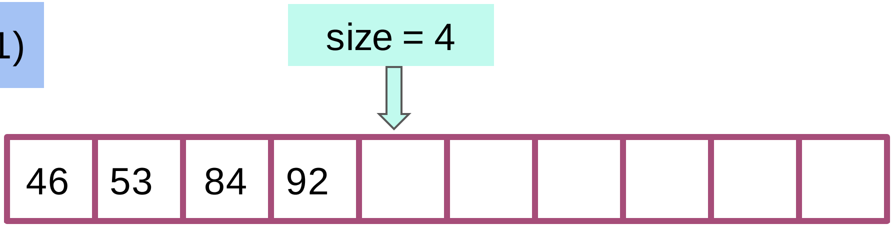
PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element

O(1)



Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element
2. Move all elements one cell to the left

O(1)

size = 4

46	53	84	92						

Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element
2. Move all elements one cell to the left

O(1)

size = 4

53	53	84	92						

Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element
2. Move all elements one cell to the left

O(1)

size = 4

53	84	84	92						

Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element
2. Move all elements one cell to the left

O(1)

size = 4

53	84	92	92						

Running example

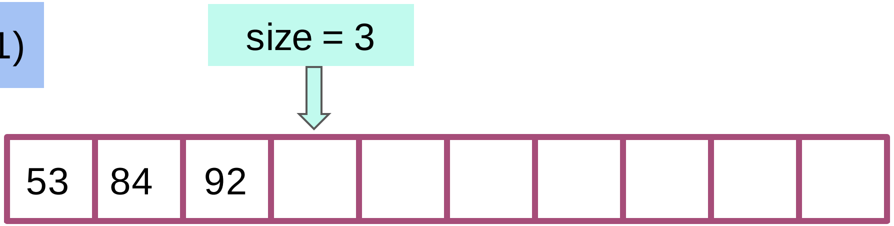
PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element
2. Move all elements one cell to the left

O(1)



Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element O(1)
2. Move all elements one cell to the left O(n)

size = 3

53	84	92							

Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	

1. Capture the array's first element — O(1)
2. Move all elements one cell to the left — O(n)

size = 3

53	84	92							

O(n) + O(1) = O(n)

Running example

PQ contains 53, 92, 46

extractMin() 46

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	Θ(n)

1. Capture the array's first element — O(1)
2. Move all elements one cell to the left — O(n)

size = 3

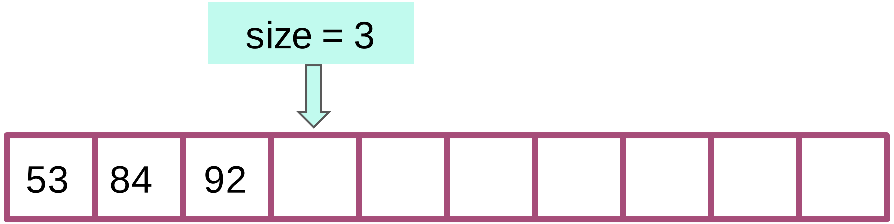
53	84	92							

O(n) + O(1) = O(n)

Running example

Do you see how to modify Ordered array so that extractMin() can be done in constant time?

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	Θ(n)



Running example

Implementation	insert	extractMin
Unordered list	Θ(1)	Θ(n)
Ordered list	Θ(n)	Θ(1)
Unordered array	Θ(1)	Θ(n)
Ordered array	Θ(n)	Θ(n)

- Not so great
  - Lists
  - Arrays
- Much better
  - Use a *heap*
  - A kind of a *tree*
  - Implemented using an array
  - Provides O(log(n)) time bound on all operations
    - O(1) peek at minimum element

Enrichment

https://xkcd.com/835/

- Don’t forget to read the mouseover text