Luiss
Libera Università Internazionale degli Studi Sociali Guido Carli

Algorithms A.Y. 2022/2023

Lab – Binary Heaps and Applications

Irene Finocchi, Flavio Giorgi, Bardh Prenkaj
finocchi@luiss.it, fgiorgi@luiss.it, bprenkaj@luiss.it©

17 February 2023

courtesy of: *Andrea Coletta*

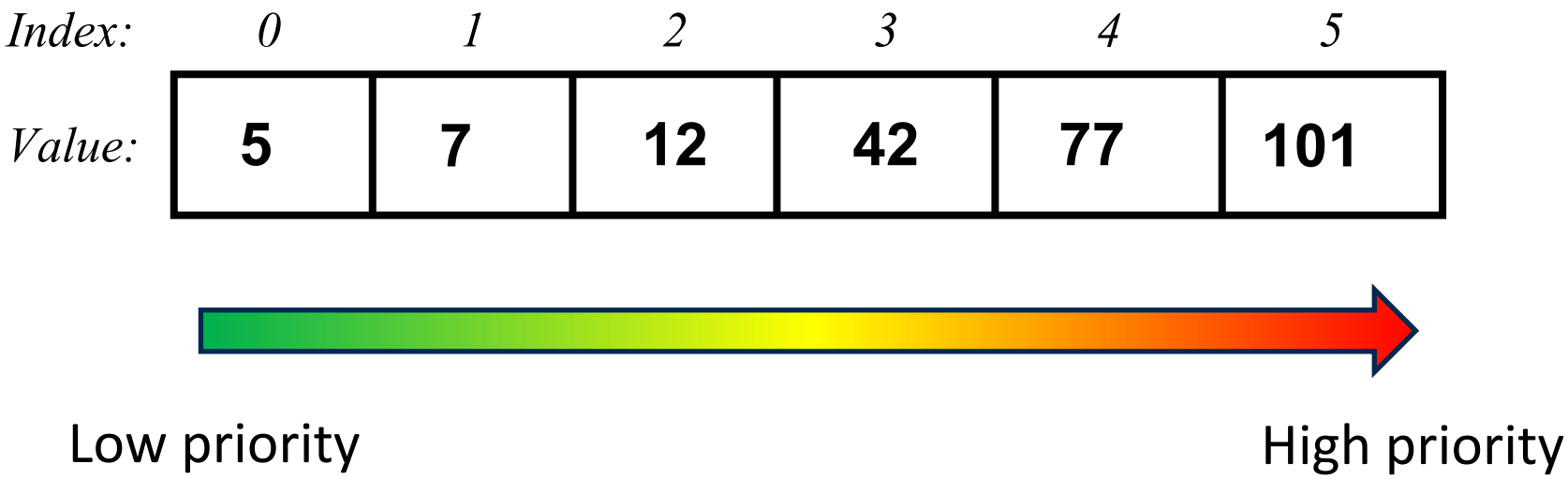
LUISS

Dipartimento di Impresa e Management

Priority Queue

What is a priority queue?

A **priority queue** is a type of queue that arranges elements based on their priority values.



Priority Queue

What is a priority queue?

A **priority queue** is a type of queue that arranges elements based on their priority values.

Elements with **higher priority values** are (usually) retrieved **before** elements with **lower priority values**.

Priority Queue

When a new element is added to the queue it is inserted in a position based on its priority value.
Low priority values go to the back of the queue, high priority values instead go to the front.

Priority Queue

What they are used for?

There is a wide variety of application for example:

Priority queuing is used to **manage limited resources** like bandwidth in a network.

Priority queues allow us to **prioritize traffic** (such as real-time traffic for streaming services).

For instance in modern protocols for local area networks (also known as LAN) include **priority** queues at the **media access control** (MAC) sub-layer to ensure that high-priority applications experience lower latency than others.

One example is IEEE 802.11e standard also known as Wi-Fi

Priority Queue

What they are used for?
Operating systems also use priority queues to decide **which process will run on the CPU**.
There are many mechanisms for example Shortest Job First, Longest Job First and other similar policy.

Priority Queue

We might imagine that since a priority queue is a queue with priorities, we should be able to implement it using a simple list.
Is it possible to do that?

Priority Queue

We might imagine that since a priority queue is a queue with priorities, we should be able to implement it using a simple list.
Is it possible to do that? YES!

Priority Queue

We might imagine that since a priority queue is a queue with priorities, we should be able to implement it using a simple list.
Is it possible to do that? YES!
In that case the maximum value (i.e., the highest-priority item) will be the first item of the list, and so is readily available in constant (i.e. $O(1)$) time. Same thing for the minimum value.

Priority Queue

What if we have to add a value to the queue?

Priority Queue

What if we have to add a value to the queue?
It would be pretty expensive, in fact the worst case takes linear time (i.e. $O(n)$)

Priority Queue

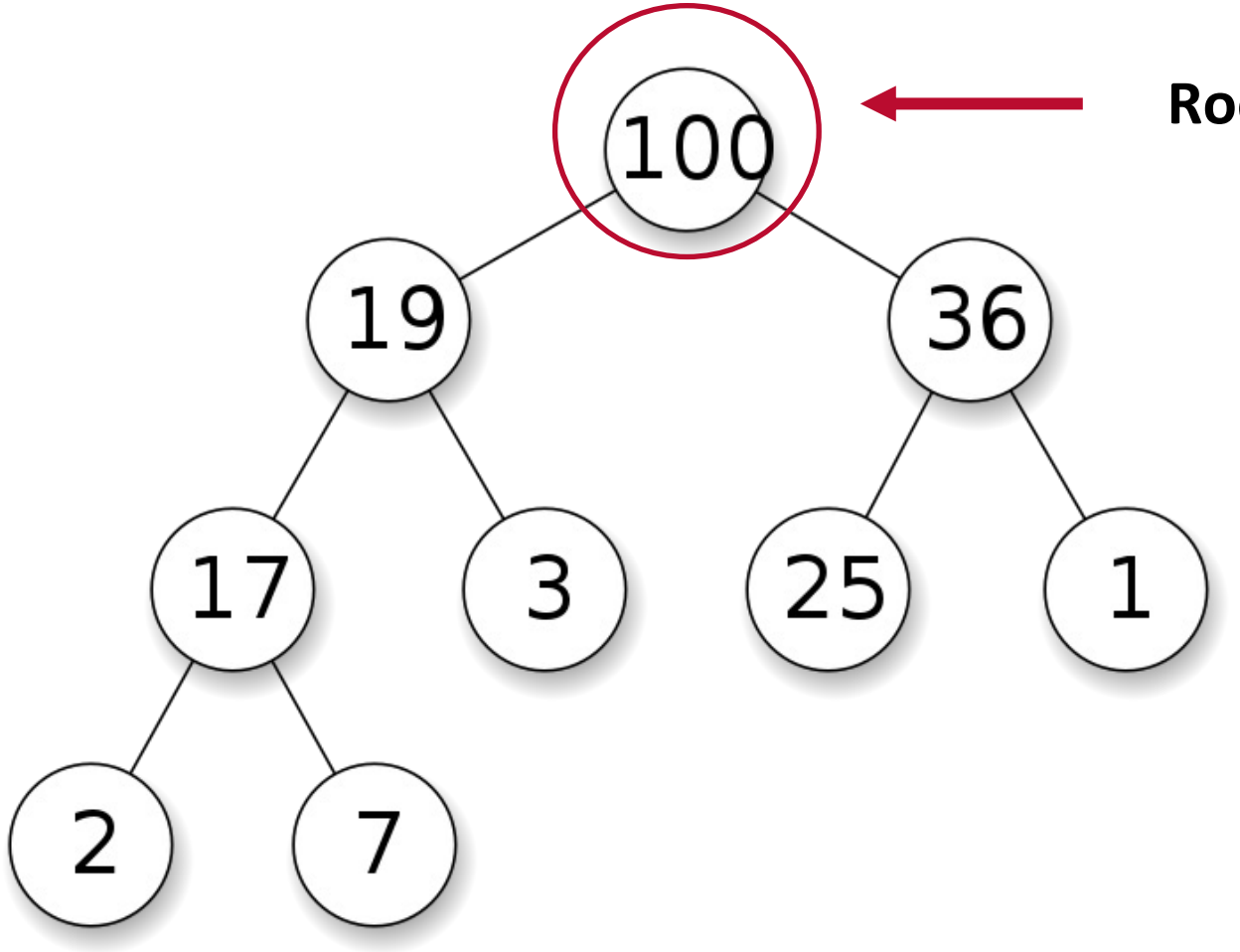
What if we have to add a value to the queue?
It would be pretty expensive, in fact the worst case takes linear time (i.e. $O(n)$)
It does not seem to be the most efficient solution!

Heaps

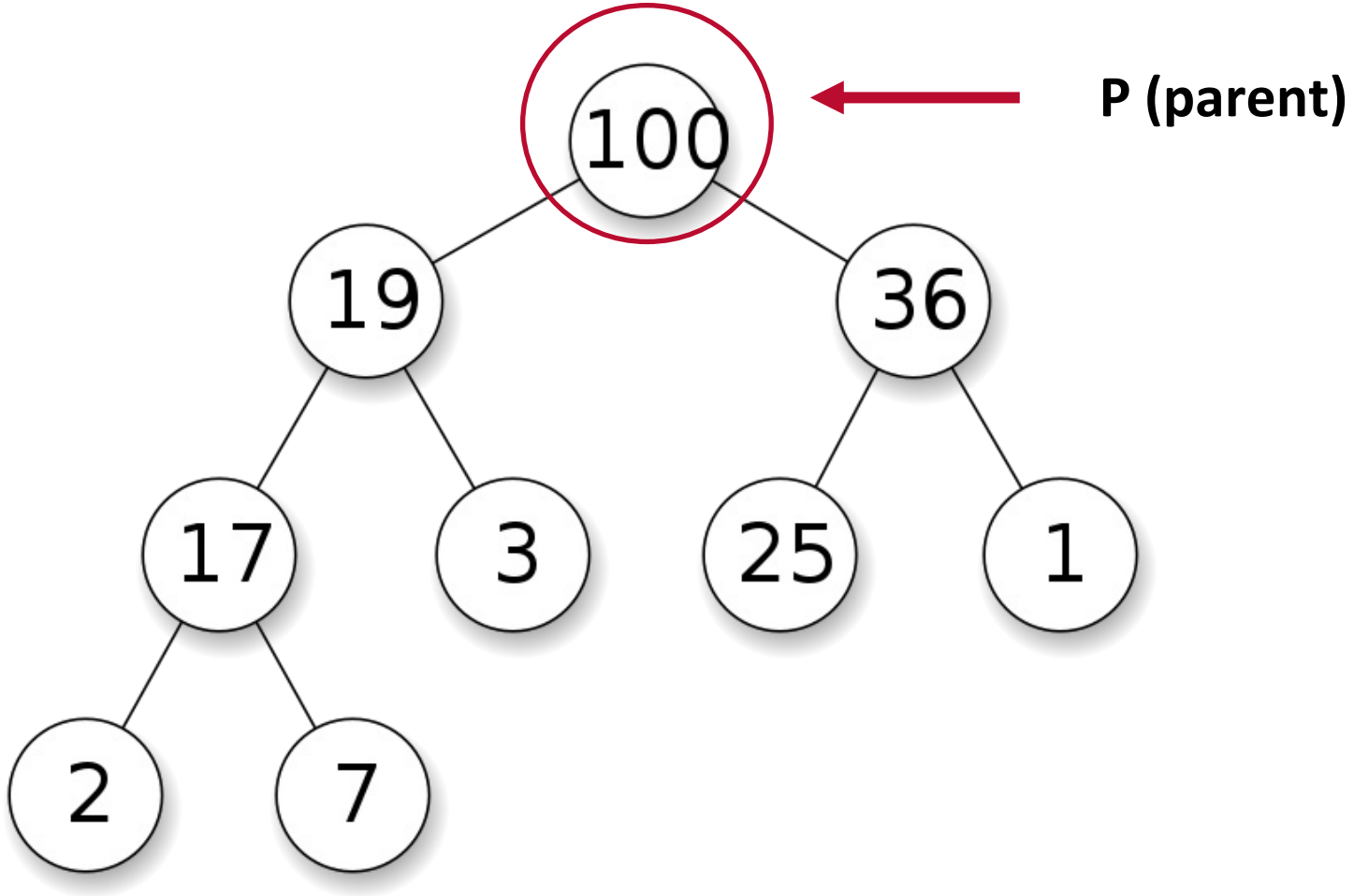
To be more efficient we can use a data structure called **Heap.** An **heap** is a **tree-based** data structure that satisfies the **heap property**: in a ***max heap***, for any given node **C**, if **P** is a parent node of **C**, then the key (the value) of **P** is greater than or equal to the key of **C**. In a ***min heap***, the key of **P** is less than or equal to the key of **C**

The node at the "top" of the heap (with no parents) is called the root node.

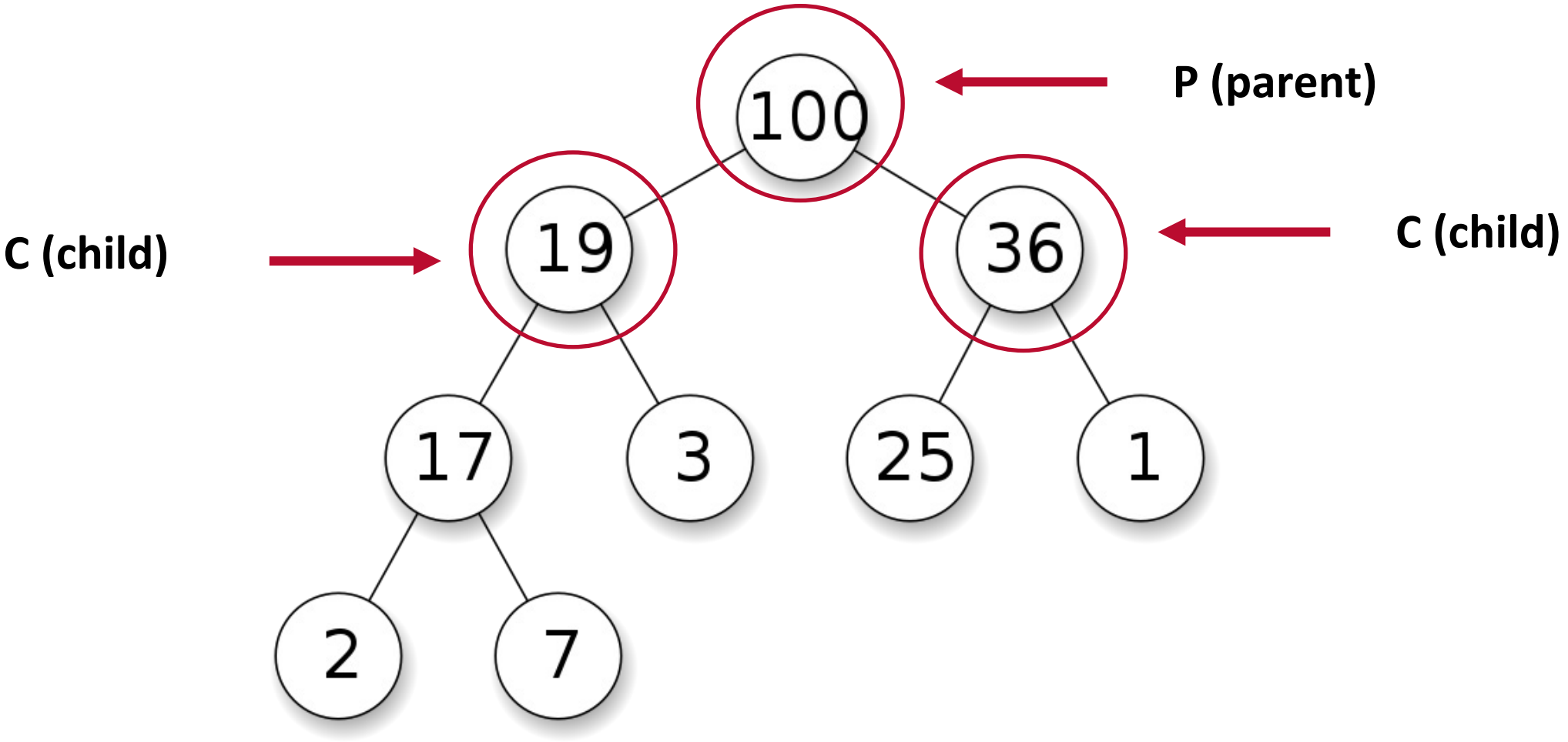
Heaps



Heaps

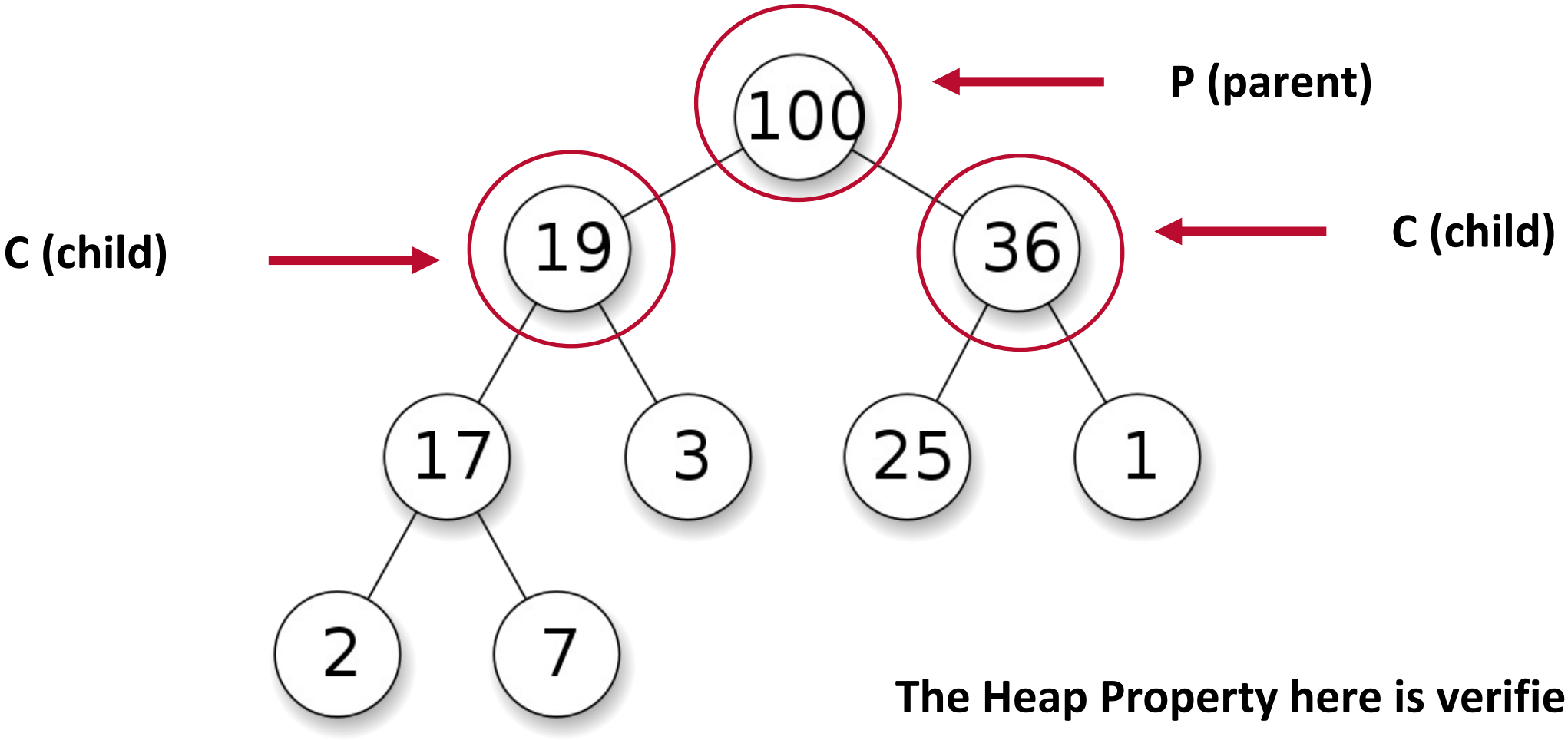


Heaps



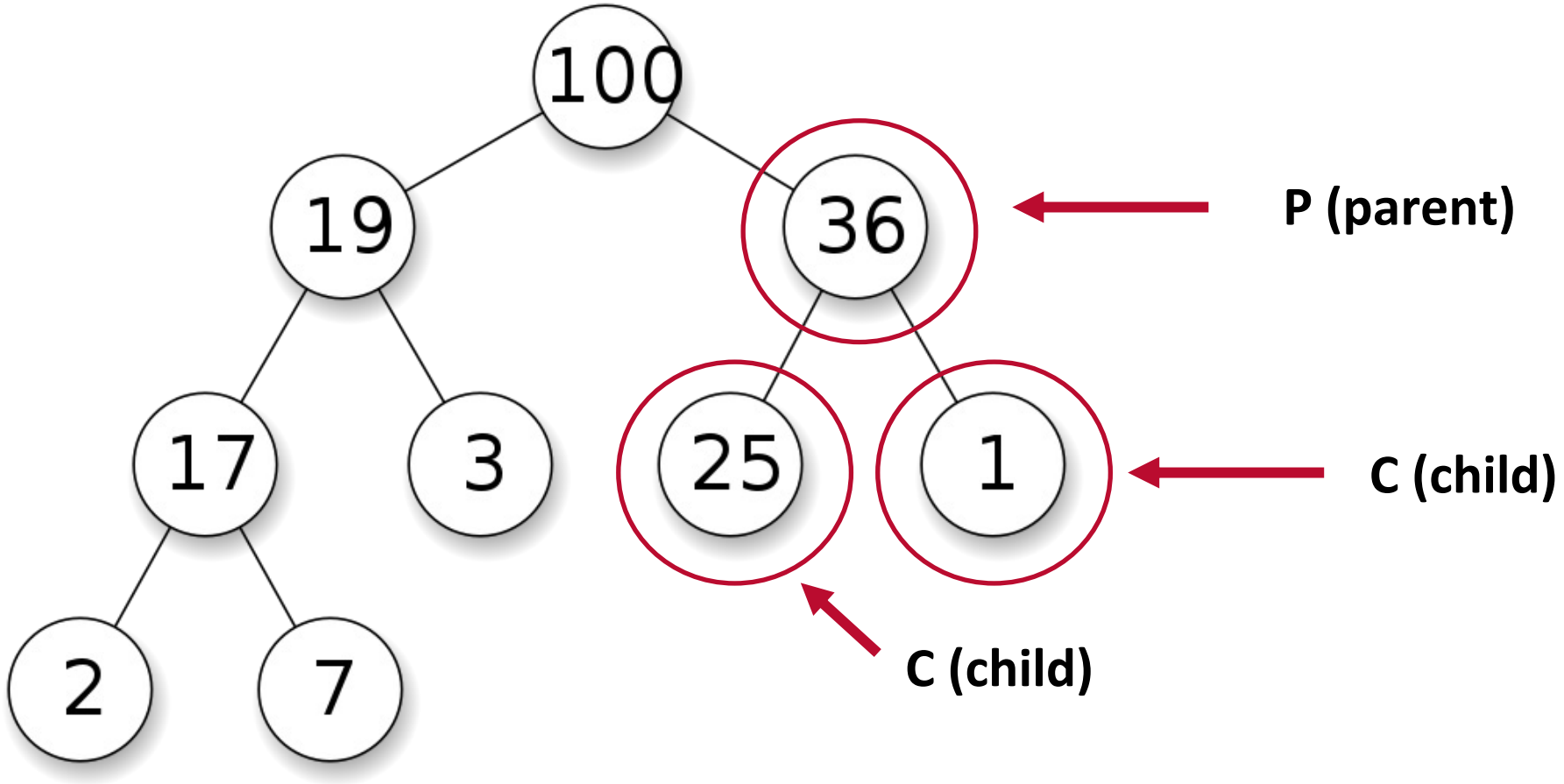
LUISS

Heaps

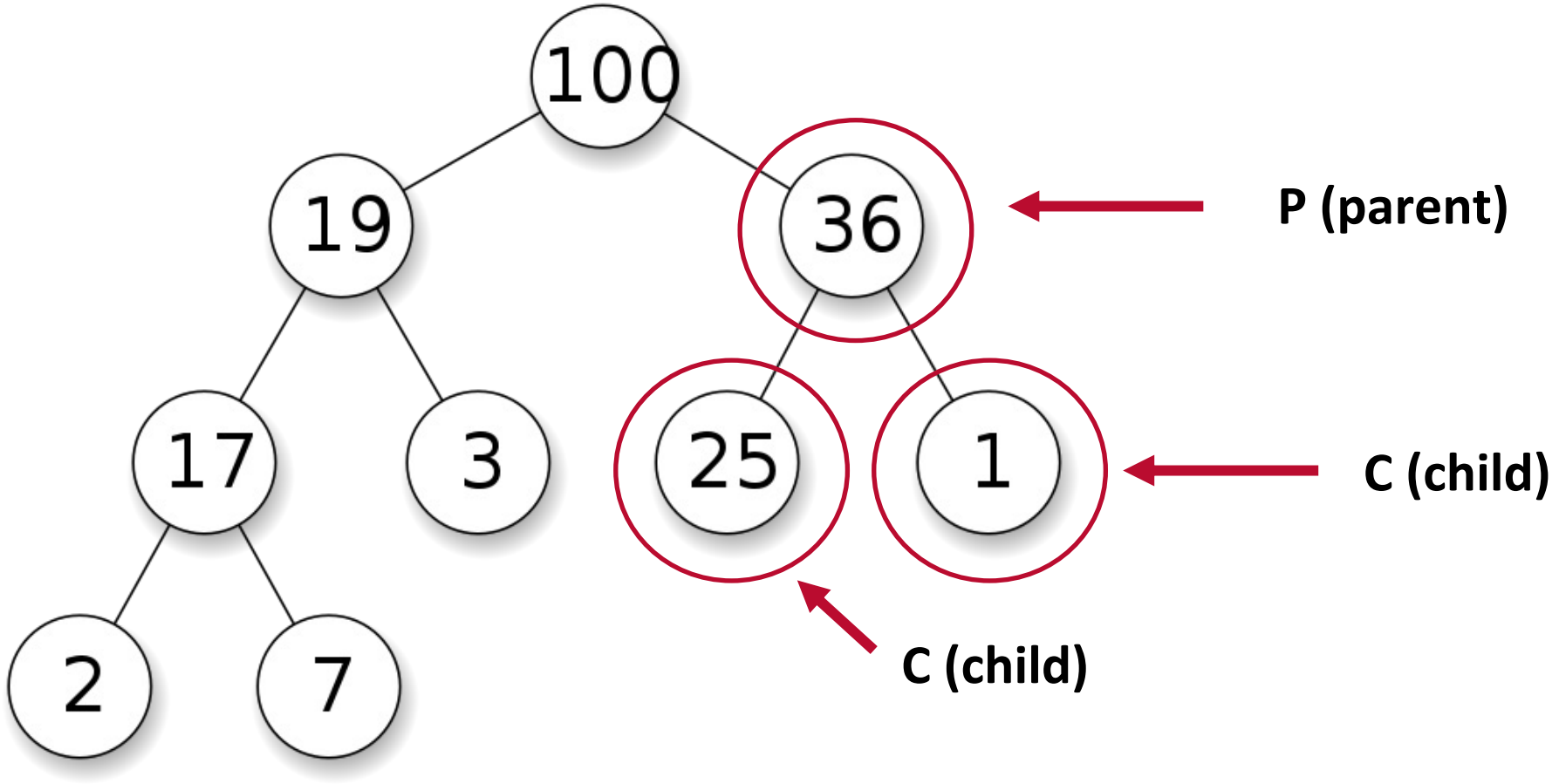


The Heap Property here is verified!

Heaps



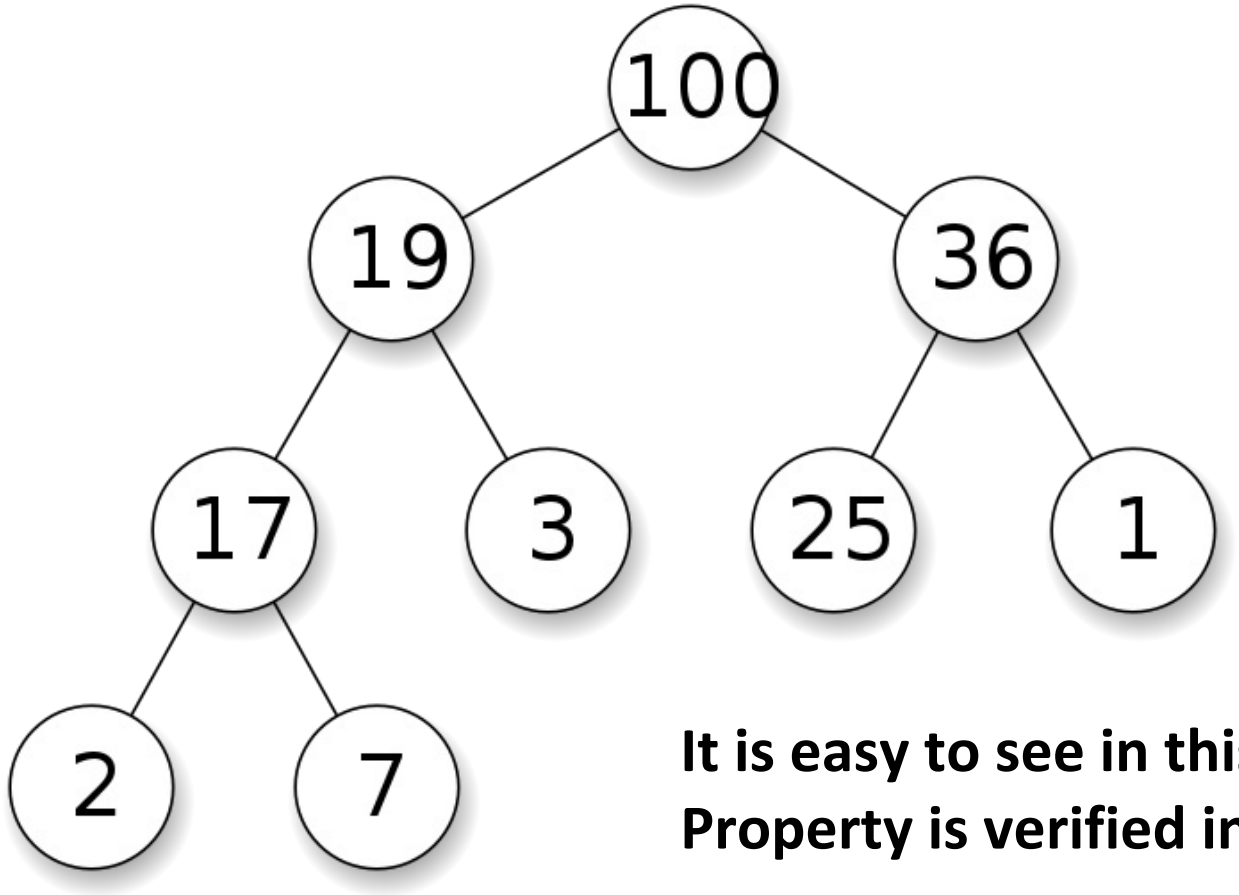
Heaps



The Heap Property here is verified!

LUISS

Heaps



It is easy to see in this case that the Heap Property is verified in the entire Heap

Heaps – Common Operations

create-heap: create a heap out of given array of elements
insert: adding a new key to the heap
delete: delete an arbitrary node

Heaps – Heapify

Heapify is a recursive function that create a heap data structure starting from a binary tree represented using an list.

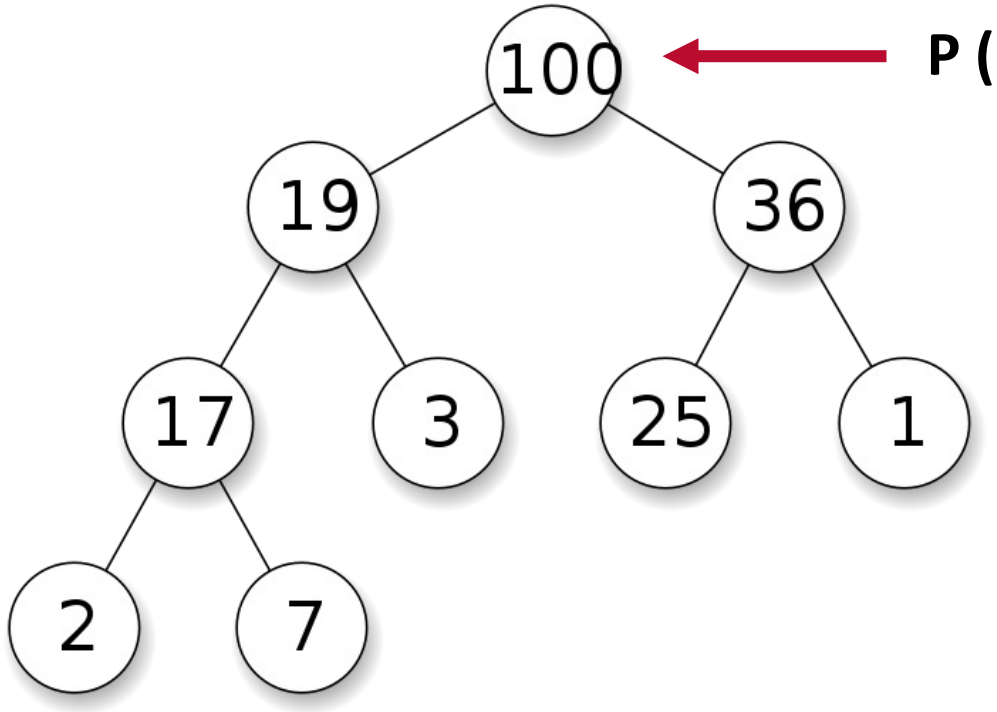
Heaps – Heapify

Heapify is a recursive function that enforce the max(min)-heap property starting from a binary tree represented using an list.

How can we represent a binary tree using a list?

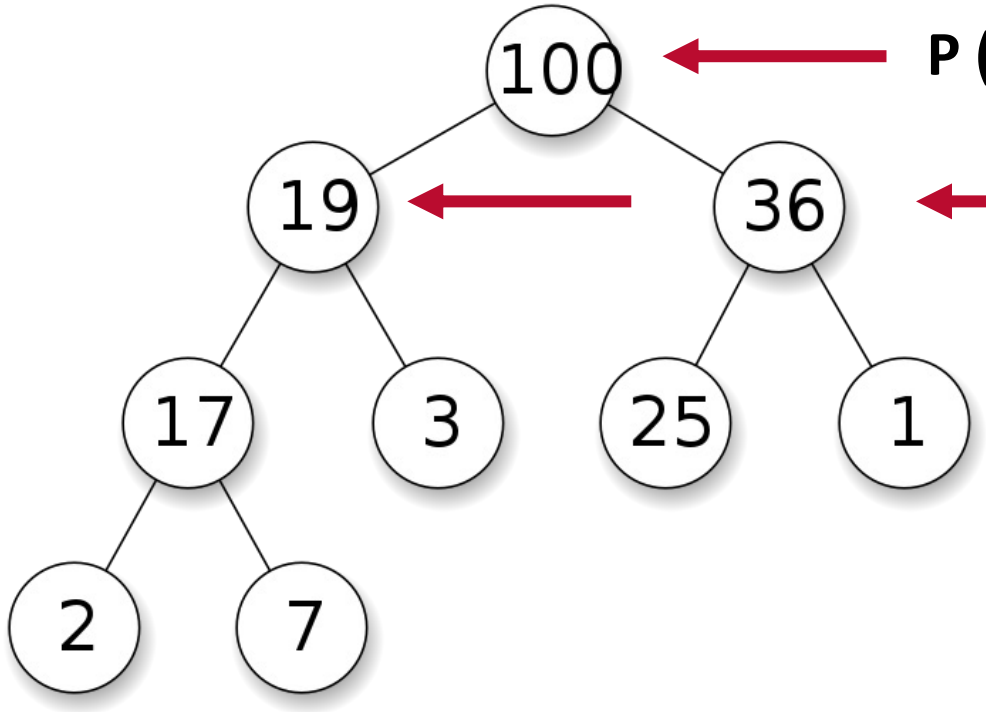
Given an element with index P (the parent) then the left child will be stored at index 2P + 1 and the right child will be stored at index 2P + 2.

Heaps – Heapify



Index:	0	1	2	3	4	5	6	…
Value:	100	19	36	17	3	25	1	…

Heaps – Heapify



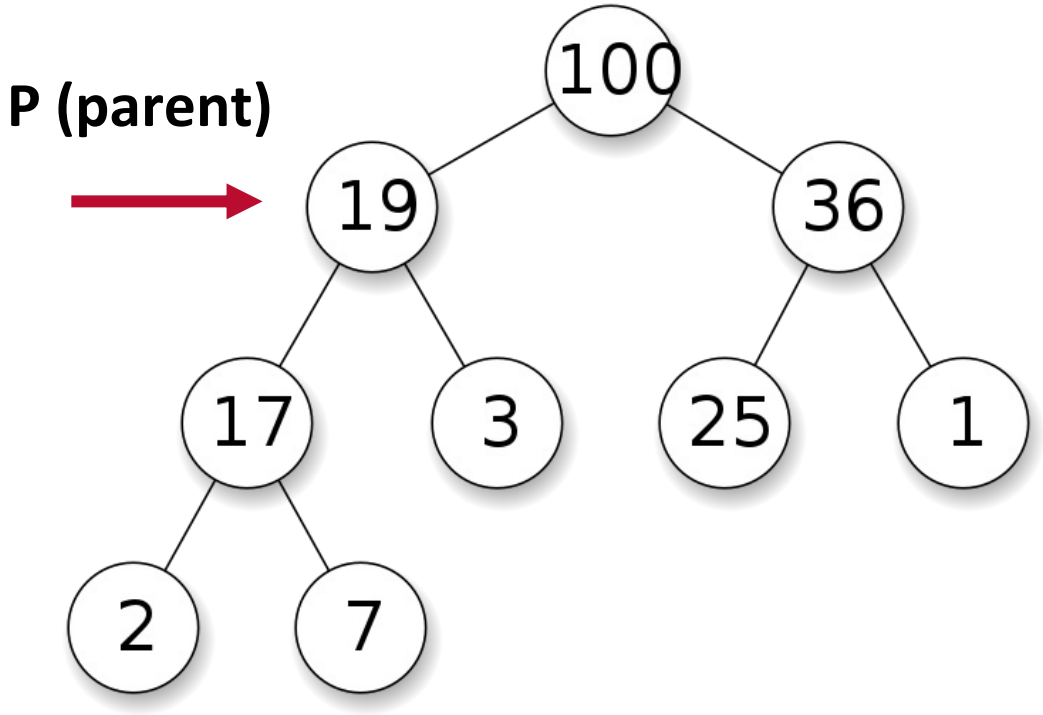
P (parent)

C (children)

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	…
Value:	100	19	36	17	3	25	1	…

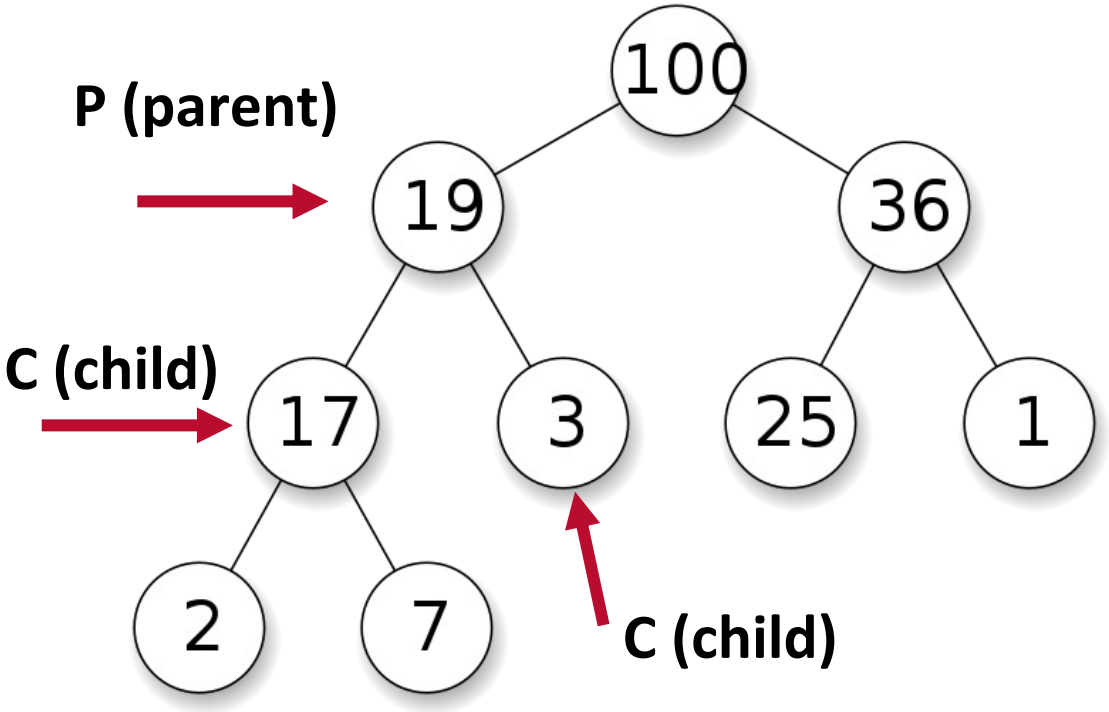
LUISS

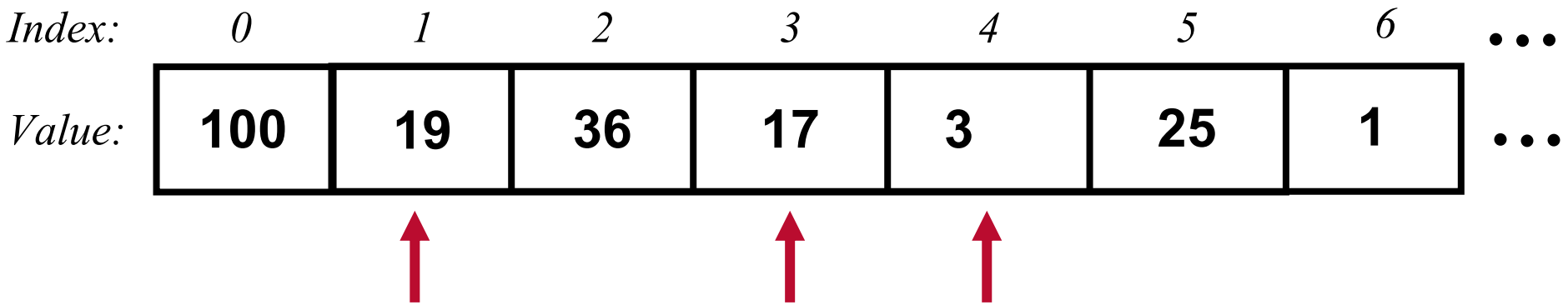
Heaps – Heapify



Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
Value:	**100**	**19**	**36**	**17**	**3**	**25**	**1**	...

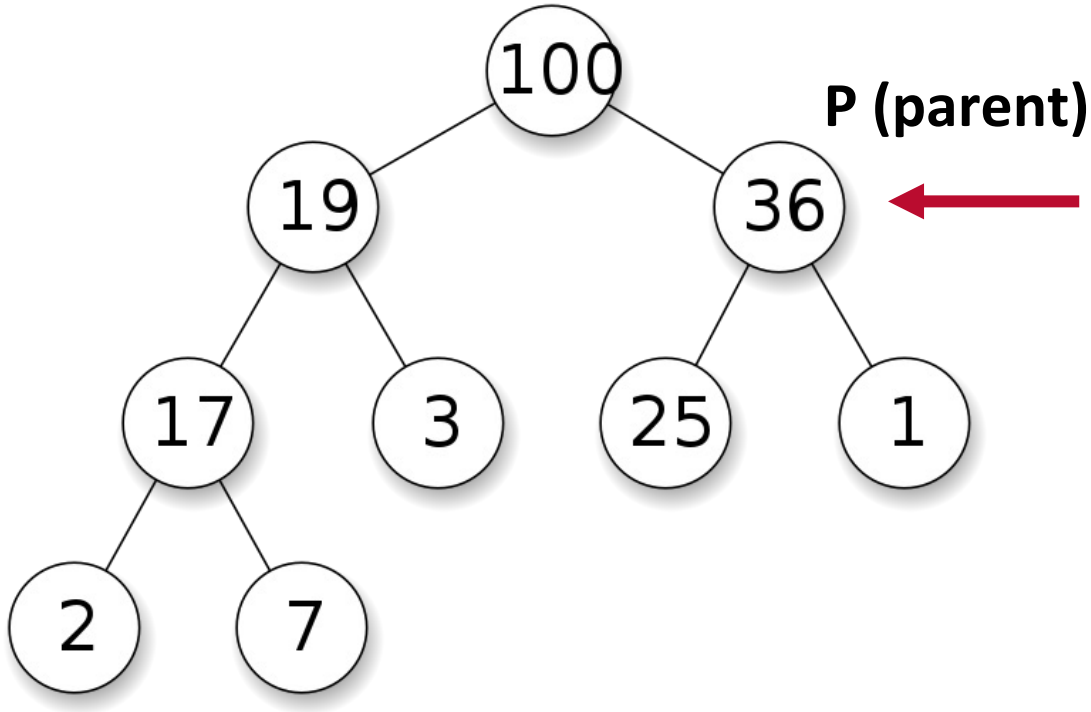
Heaps – Heapify





LUISS

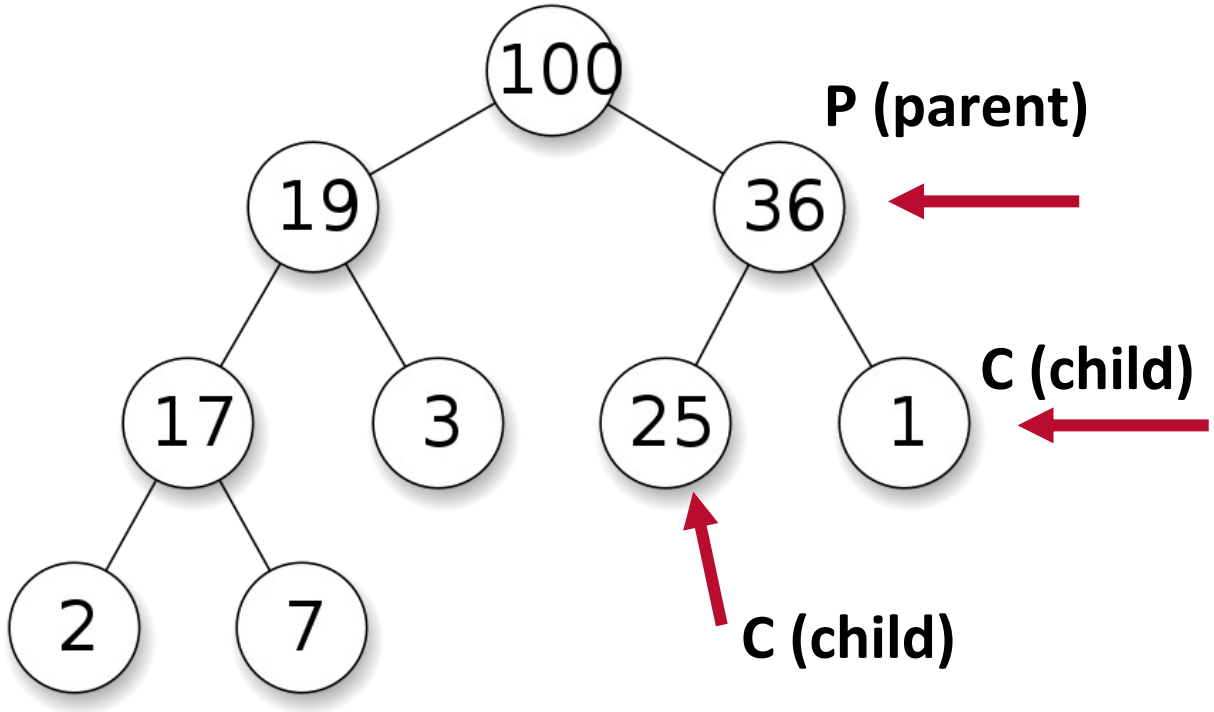
Heaps – Heapify

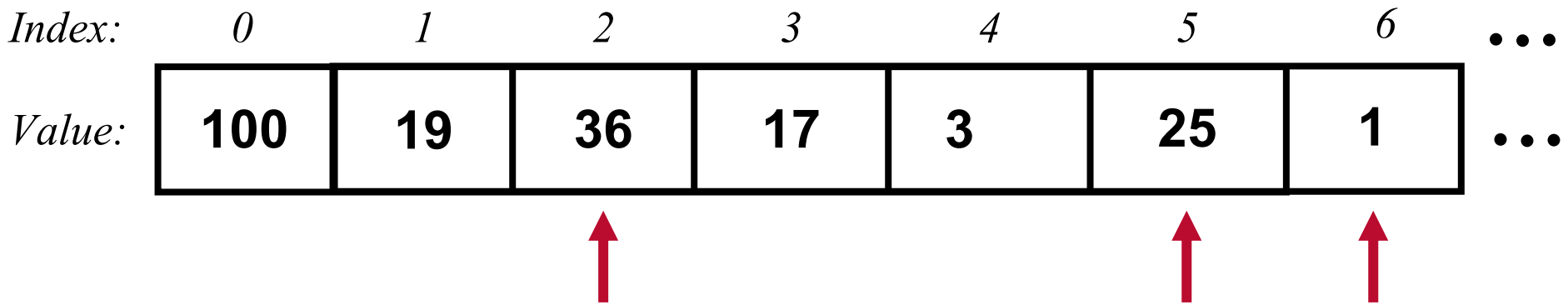


Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	…
Value:	100	19	36	17	3	25	1	…

LUISS

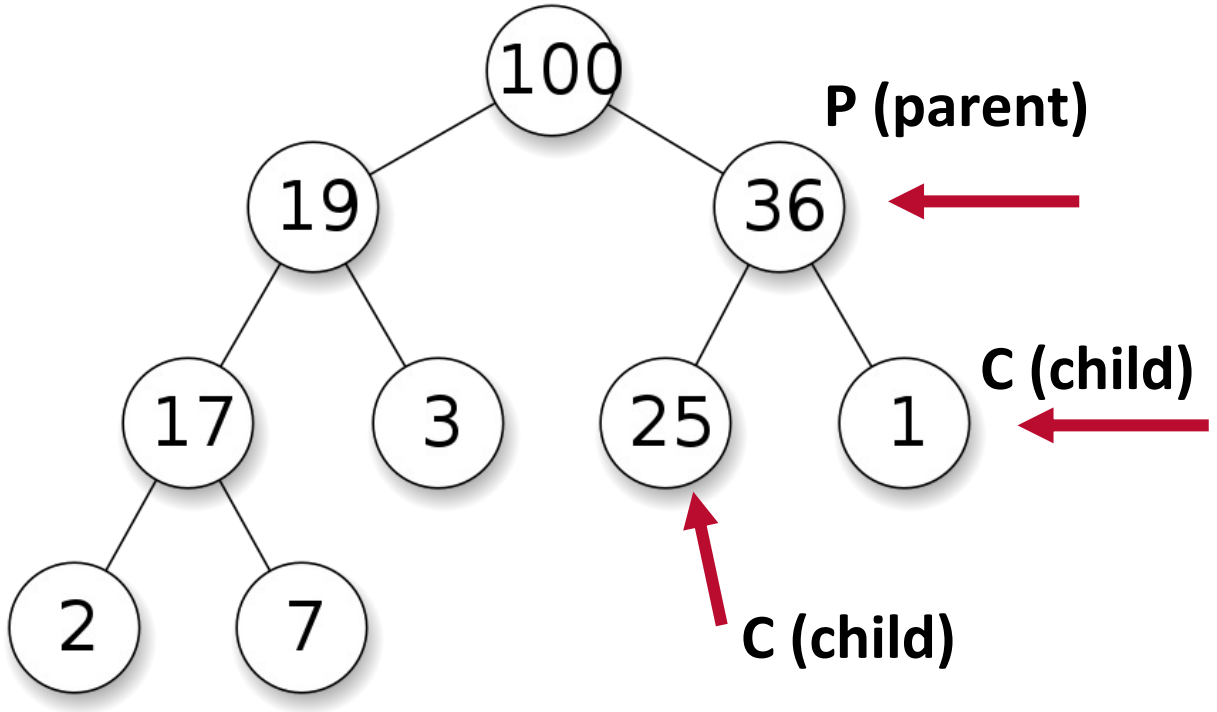
Heaps – Heapify



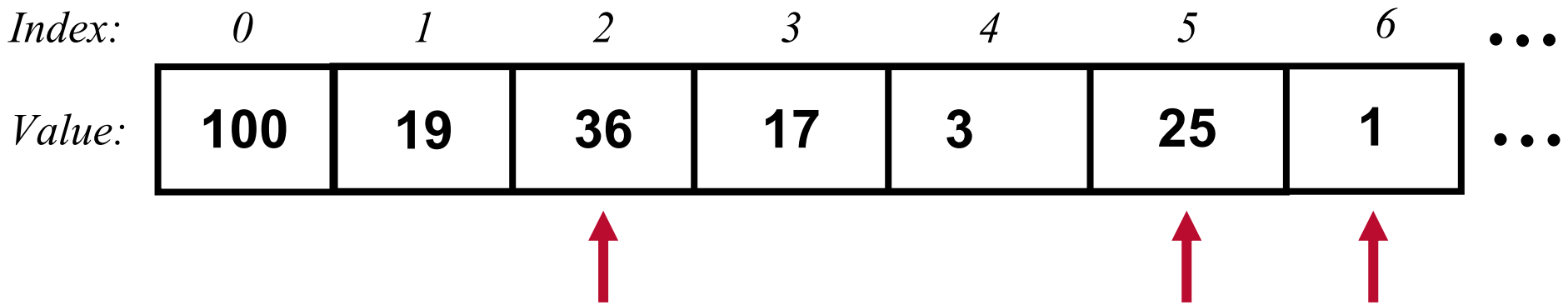


LUISS

Heaps – Heapify



And so on…



LUISS

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Heaps – Heapify

The input list

The index of the current element

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

The index of the left element given the current index (2*index +1)

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

The index of the right element given the current index (2*index +2)

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Check if the le is within the bound of the heap

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

If the left child of the element is bigger than the parent element

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Then the largest element up to now is the left child

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Then the largest element up to now is the left child

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Check again if the right side is within the heap boundaries

Heaps – Heapify

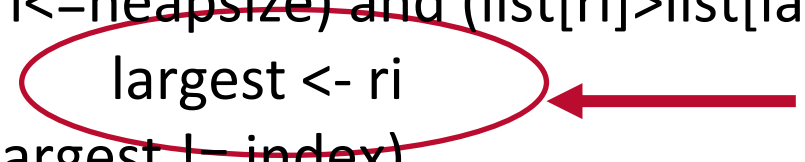
```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Check if the right child is larger than the Largest value found up to noe

LUISS

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```



If that is true then use as largest value the right child

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

If the largest value is not the current root then

Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Swap the root and the largest element found

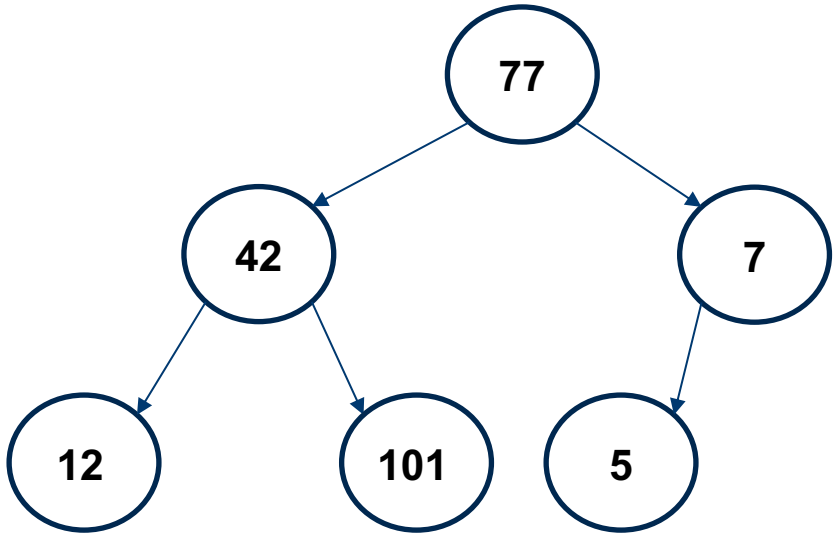
Heaps – Heapify

```
def Heapify(list, index):
        le <- left(index)
        ri <- right(index)
        if (le<=heapsize) and (list[le]>list[index])
                largest <- le
        else
                largest <- index
        if (ri<=heapsize) and (list[ri]>list[largest])
                largest <- ri
        if (largest != index)
                swap list[index] with list[largest]
                Heapify(list, largest)
```

Recursively call the function

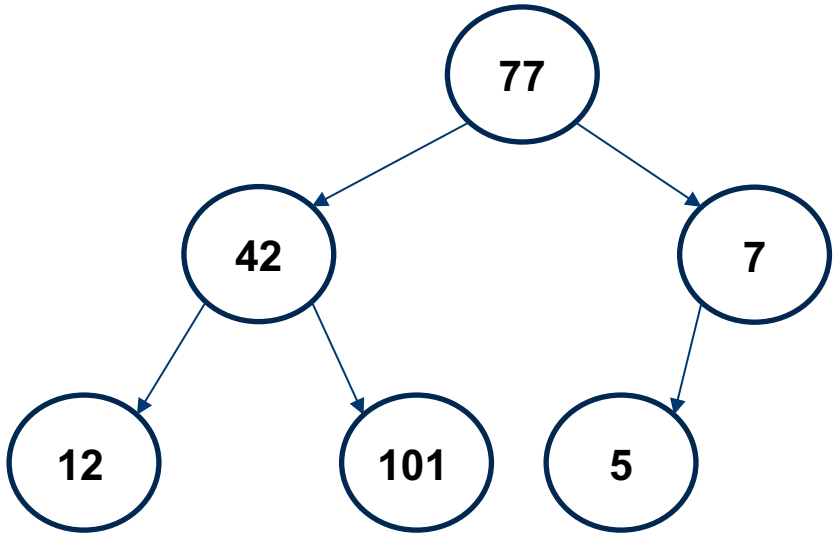
Heaps – How to build an Heap

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**7**	**12**	**101**	**5**



Heaps – How to build an Heap

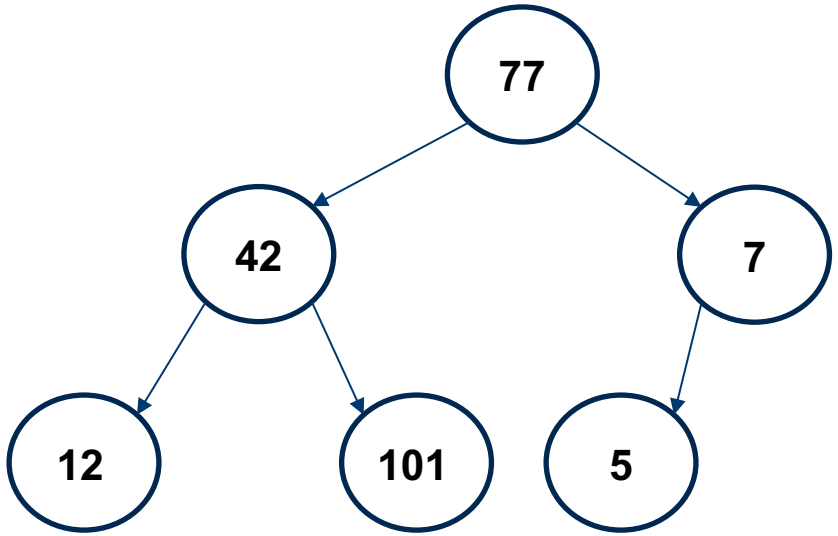
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	77	42	7	12	101	5



Is it a Max-Heap?

Heaps – How to build an Heap

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	77	42	7	12	101	5

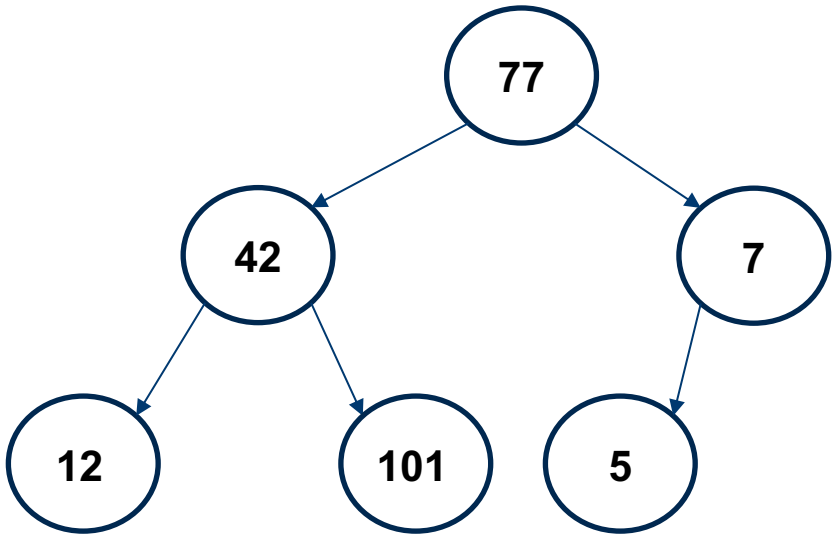


Is it a Max-Heap?

Clearly not!

Heaps – How to build an Heap

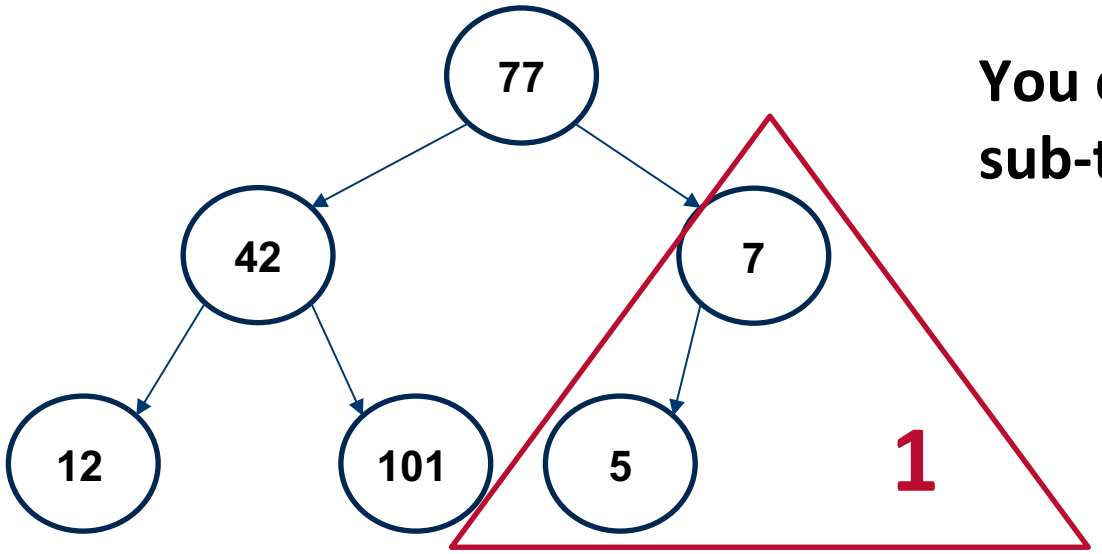
Index:	0	1	2	3	4	5
Value:	77	42	7	12	101	5



We can use heapify to enforce the property starting from the bottom and going up

Heaps – How to build an Heap

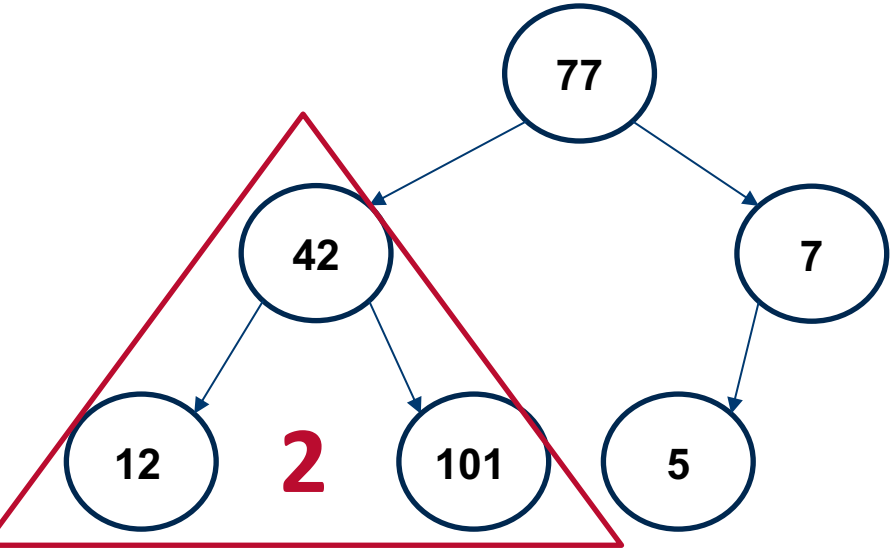
Index:	0	1	2	3	4	5
Value:	77	42	7	12	101	5



You can think to the procedure as we check all the sub-trees starting from the bottom tree

Heaps – How to build an Heap

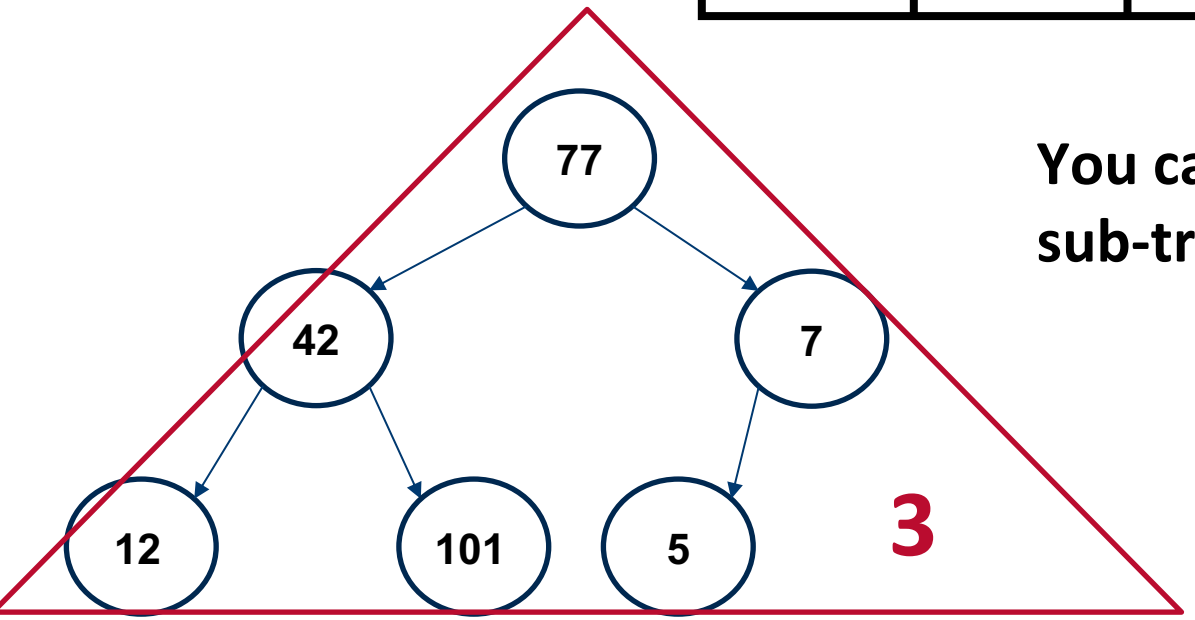
Index:	0	1	2	3	4	5
Value:	77	42	7	12	101	5



You can think to the procedure as we check all the sub-trees starting from the bottom tree

Heaps – How to build an Heap

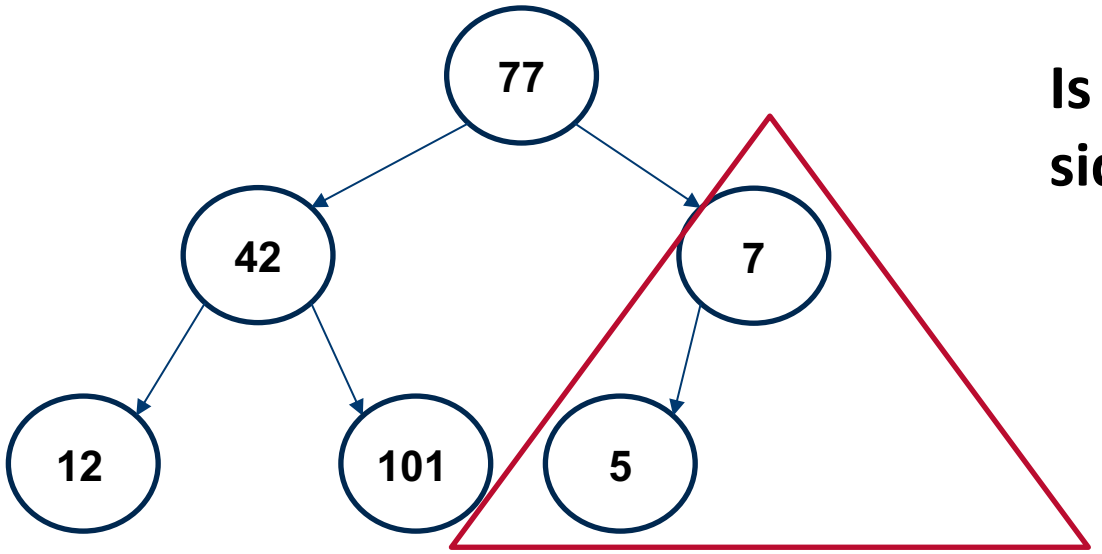
Index:	0	1	2	3	4	5
Value:	77	42	7	12	101	5



You can think to the procedure as we check all the sub-trees starting from the bottom tree

Heaps – How to build an Heap

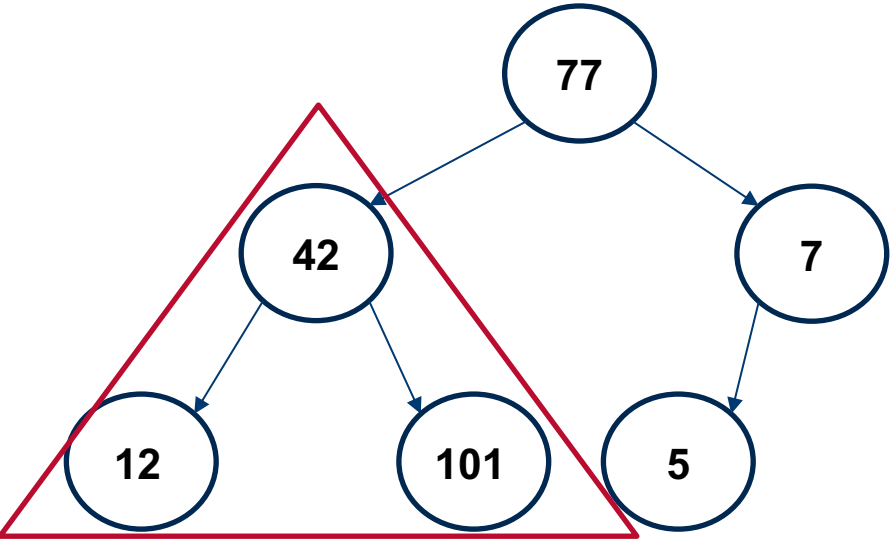
Index:	0	1	2	3	4	5
Value:	77	42	7	12	101	5



Is the root of the tree (index = 2) grater than the left side element? Yes! We can pass at the next sub tree

Heaps – How to build an Heap

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	77	42	7	12	101	5

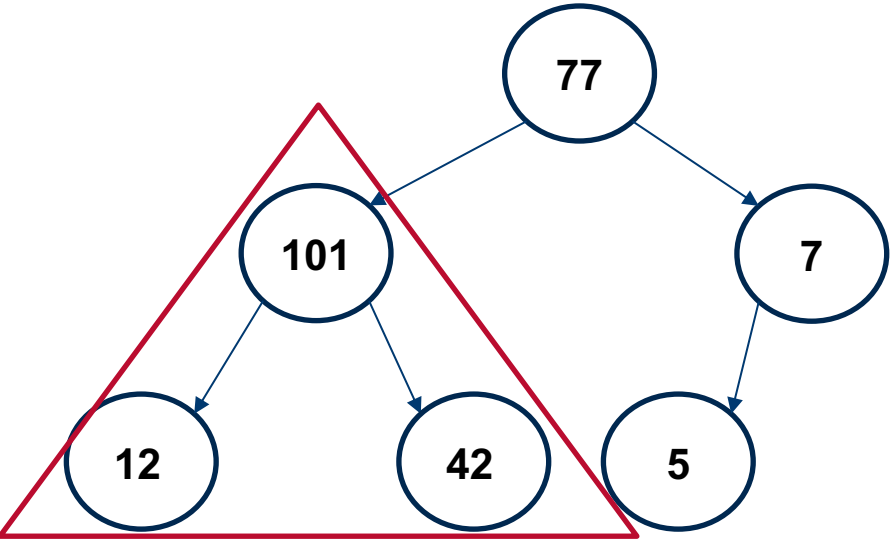


We can use heapify to enforce the property starting from the bottom and going up

Is the root of the tree (index = 1) grater than the left side element? Yes!

Heaps – How to build an Heap

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	77	101	7	12	42	5



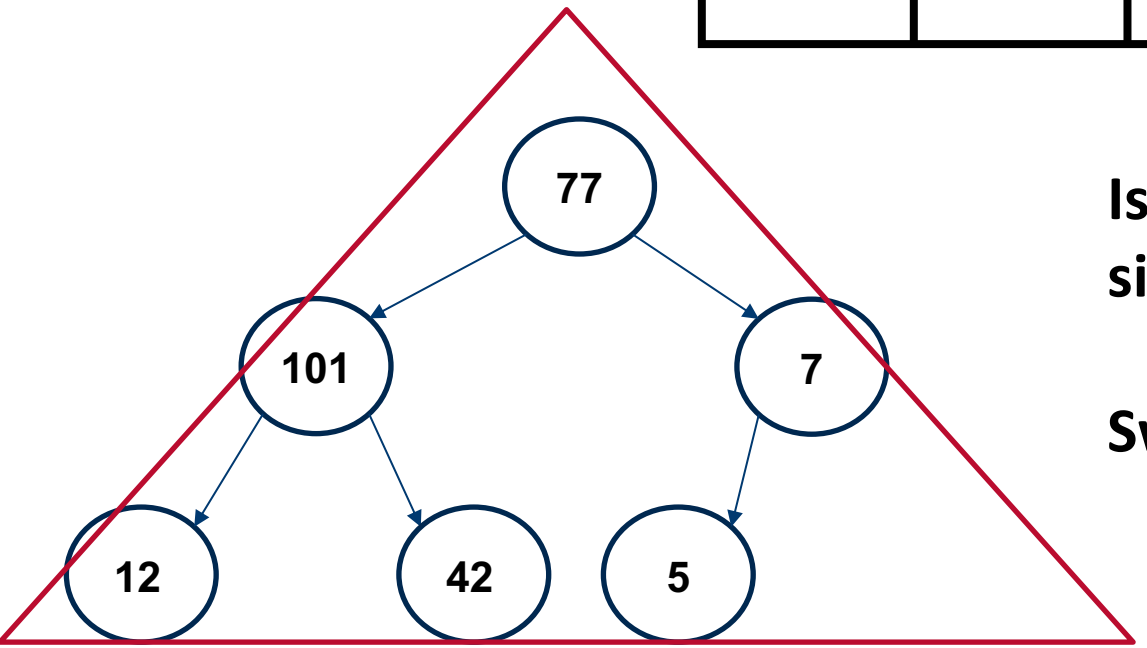
We can use heapify to enforce the property starting from the bottom and going up

Is the root of the tree (index = 1) grater than the right side element? No!

Swap the elements!

Heaps – How to build an Heap

Index:	0	1	2	3	4	5
Value:	77	101	7	12	42	5

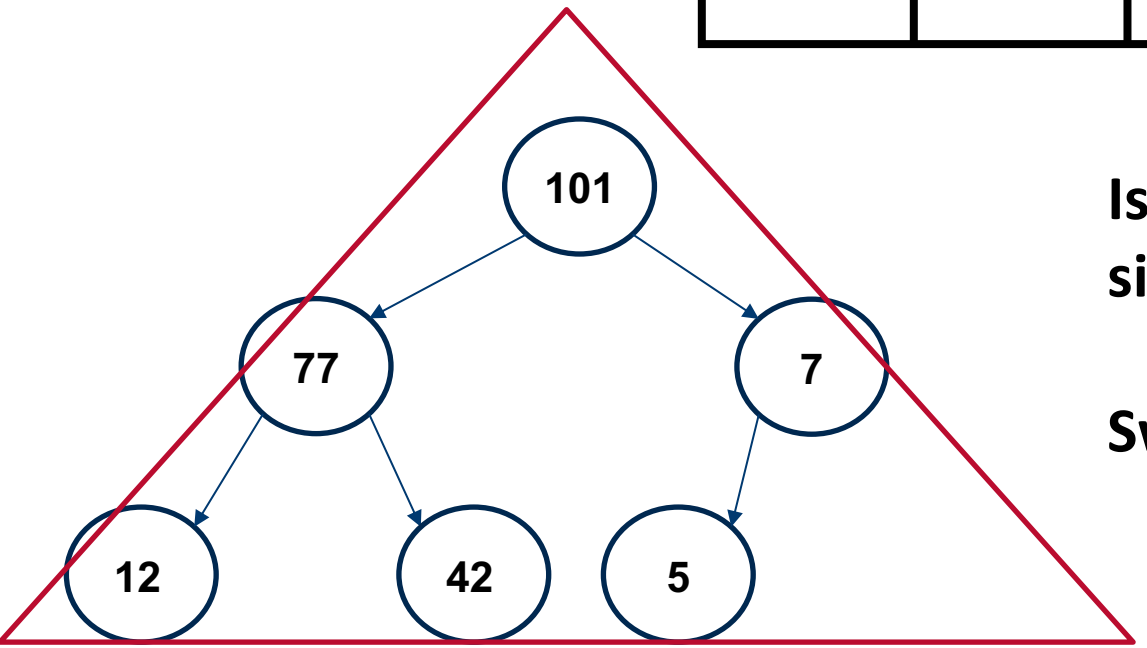


Is the root of the tree (index = 0) grater than the right side element? No! the left side is grater? No!

Swap the elements!

Heaps – How to build an Heap

Index:	0	1	2	3	4	5
Value:	101	77	7	12	42	5

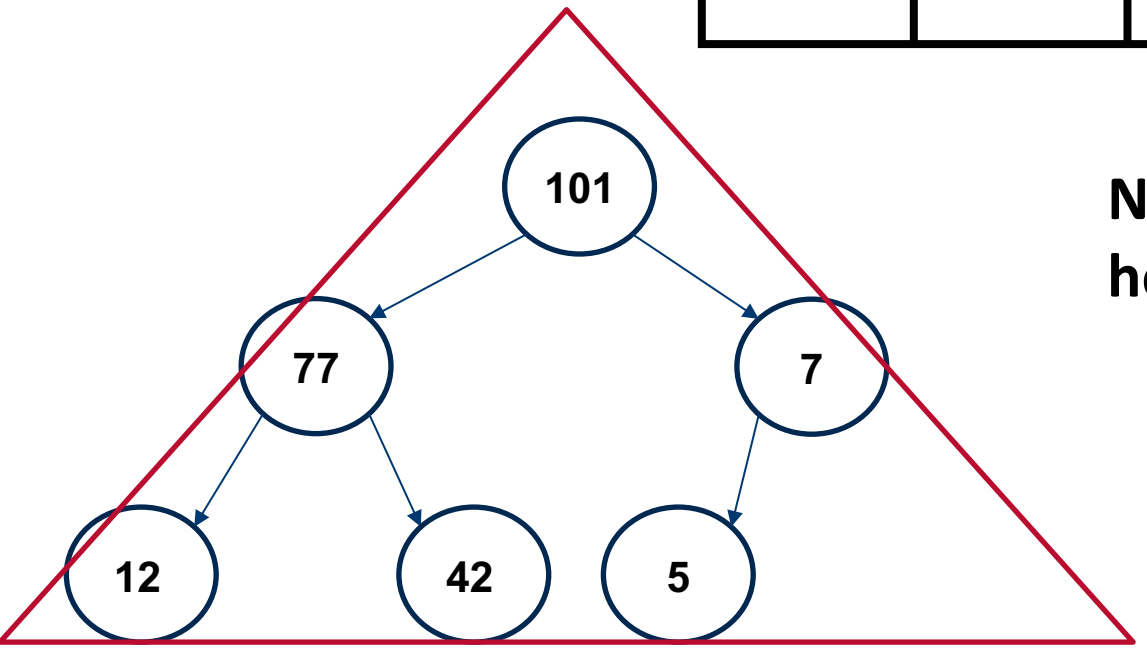


Is the root of the tree (index = 0) grater than the right side element? No! the left side is grater? No!

Swap the elements!

Heaps – How to build an Heap

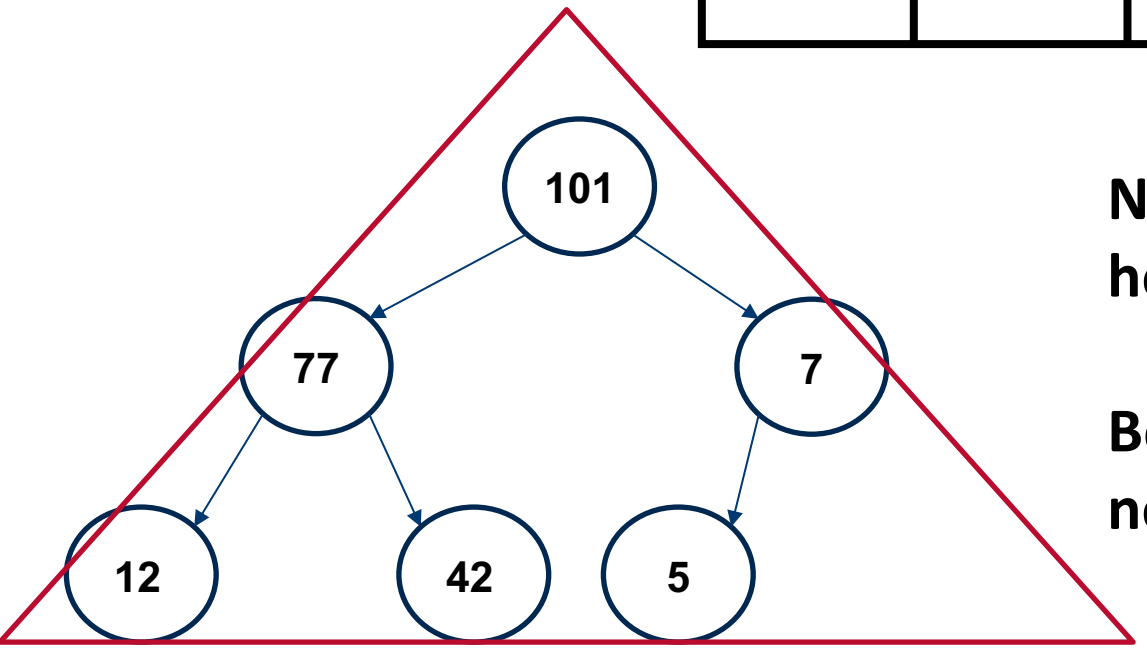
Index:	0	1	2	3	4	5
Value:	101	77	7	12	42	5



Now since the order changed we have to call again heapify on index = 1 why?

Heaps – How to build an Heap

Index:	0	1	2	3	4	5
Value:	101	77	7	12	42	5

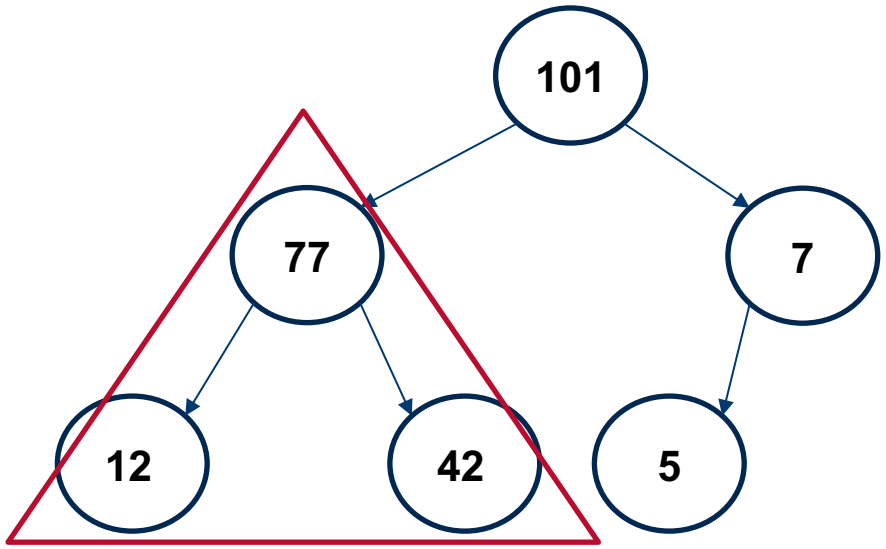


Now since the order changed we have to call again heapify on index = 1 why?

Because after the swap the max-heap property could not hold anymore even if we checked it before!

Heaps – How to build an Heap

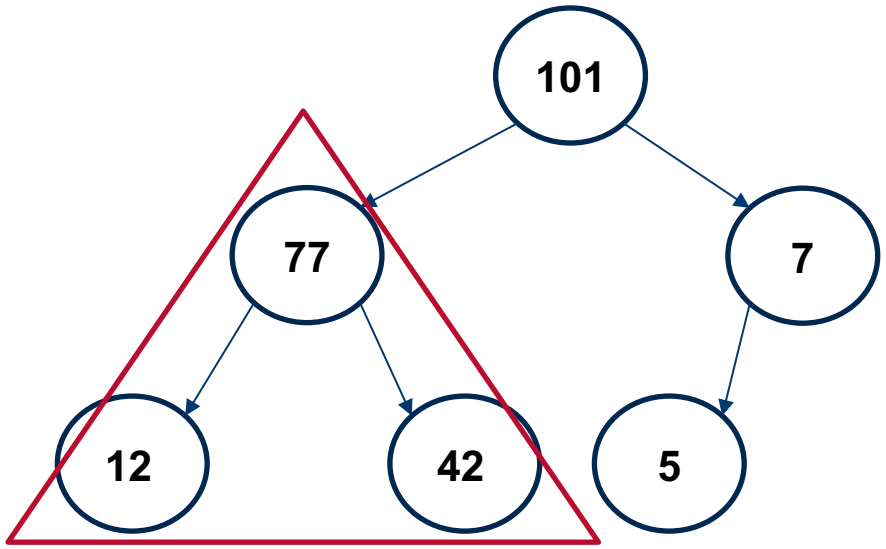
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	101	77	7	12	42	5



So again we have to check if the sub-tree (red triangle) is a max-heap

Heaps – How to build an Heap

Index:	0	1	2	3	4	5
Value:	101	77	7	12	42	5

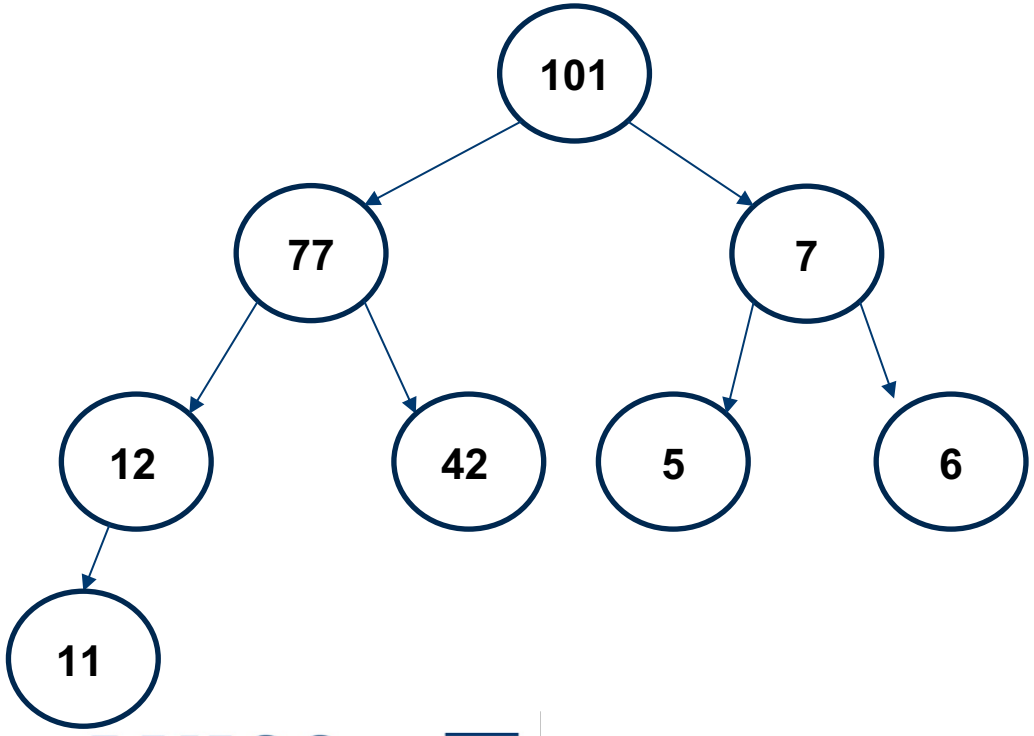


So again we have to check if the sub-tree (red triangle) is a max-heap

Even if we swapped the elements, the property for the sub-tree still holds so the procedure ends.

Heaps – Insert an element

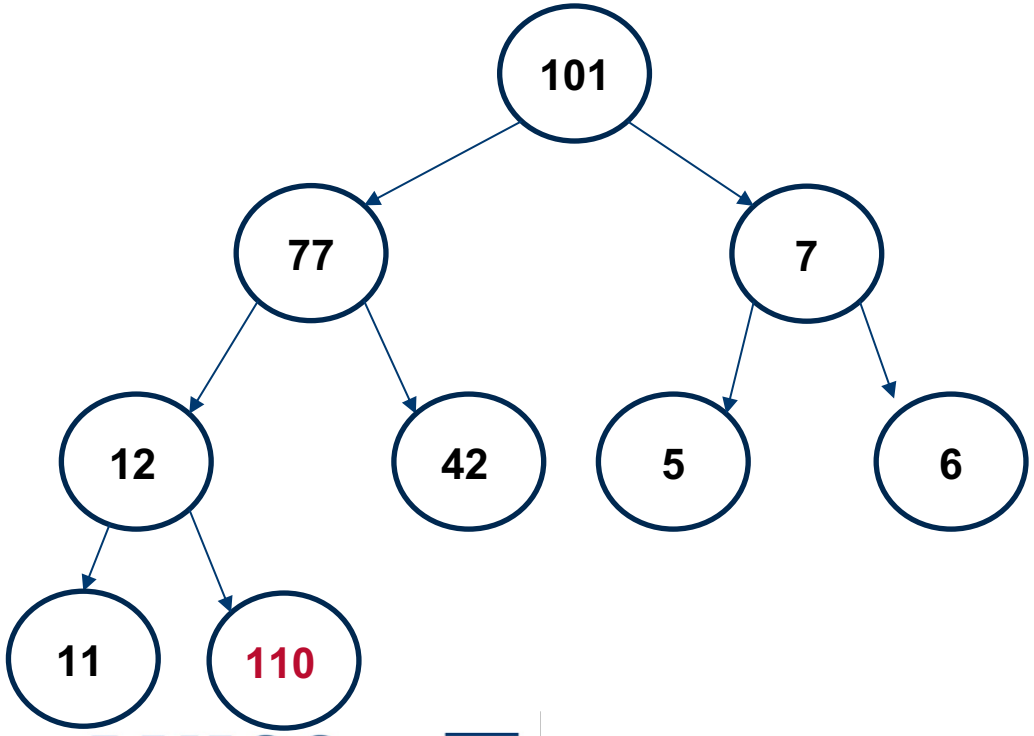
Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Value:	101	77	7	12	42	5	6	11	110



Suppose we want to add an element to the heap

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	101	77	7	12	42	5	6	11	110

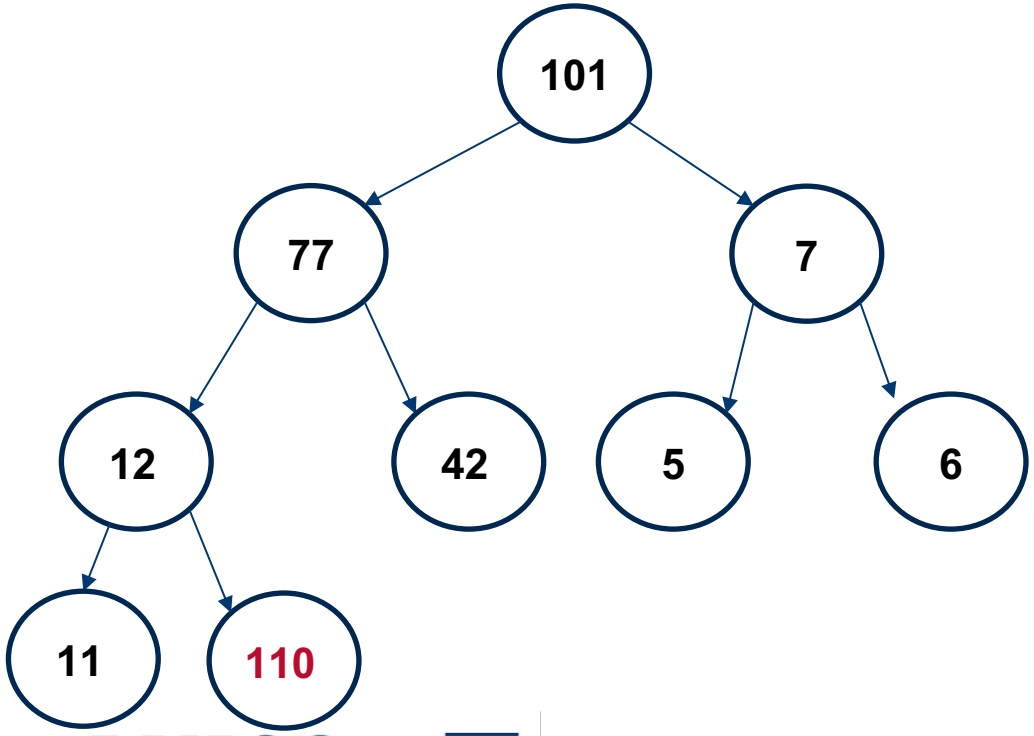


We put the element at the end of the list, so It will be the last leaf of the tree.

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	101	77	7	12	42	5	6	11	110



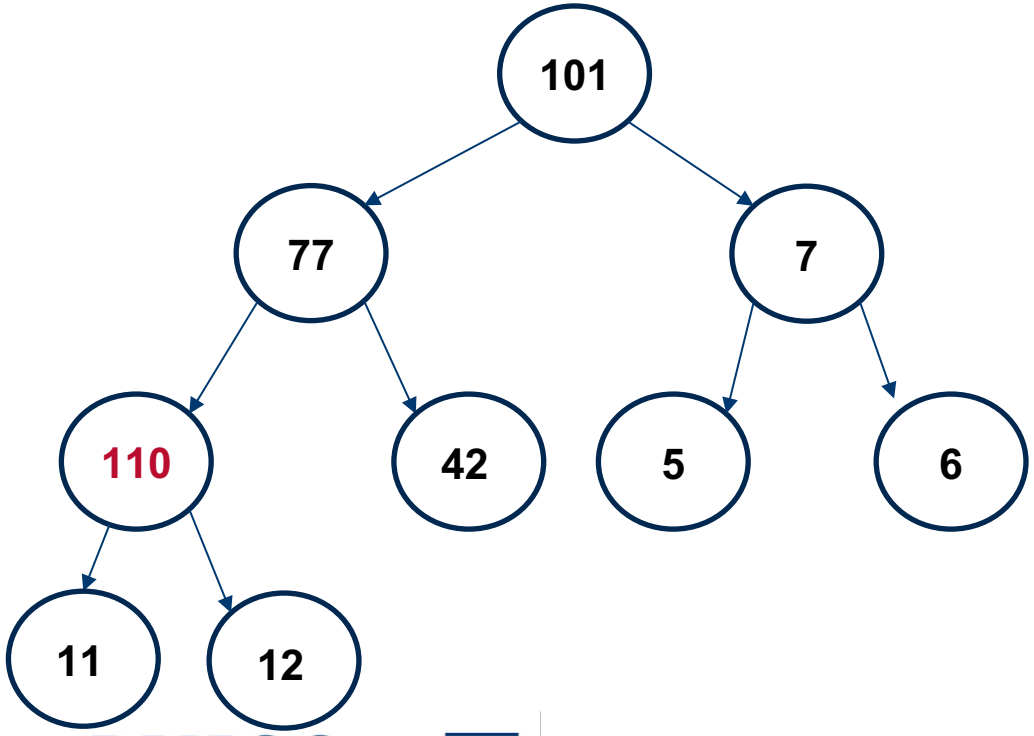
The we start comparing the element with its parent!

If the parent is smaller than we swap the elements!

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	101	77	7	110	42	5	6	11	12



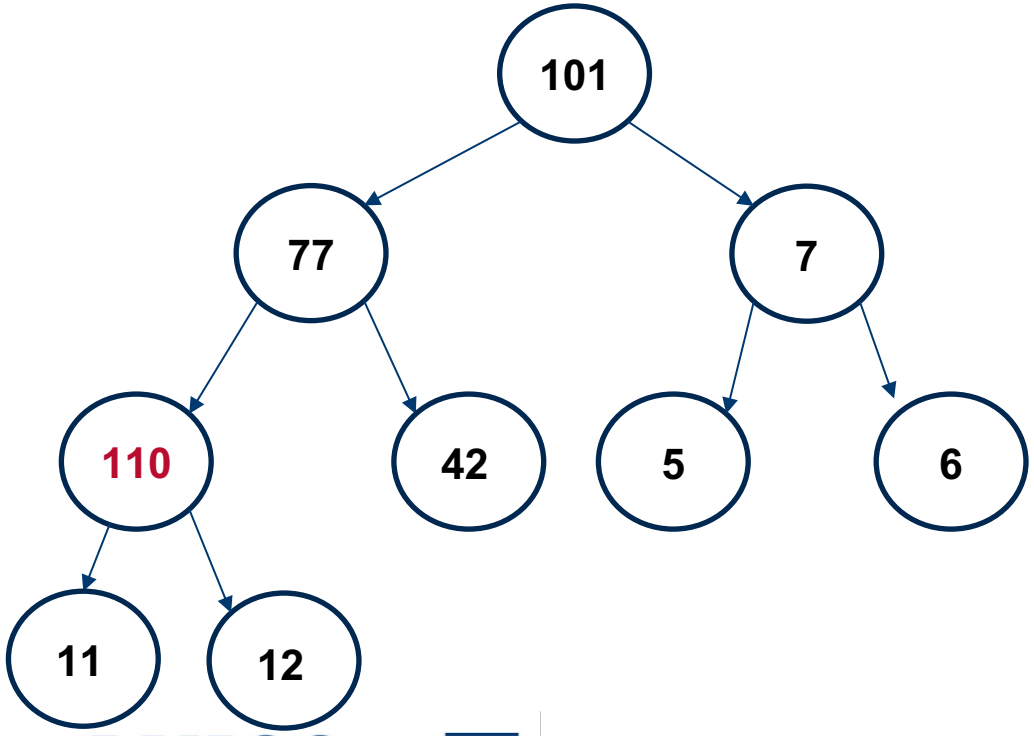
The we start comparing the element with its parent!

If the parent is smaller than we swap the elements!

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	101	77	7	110	42	5	6	11	12

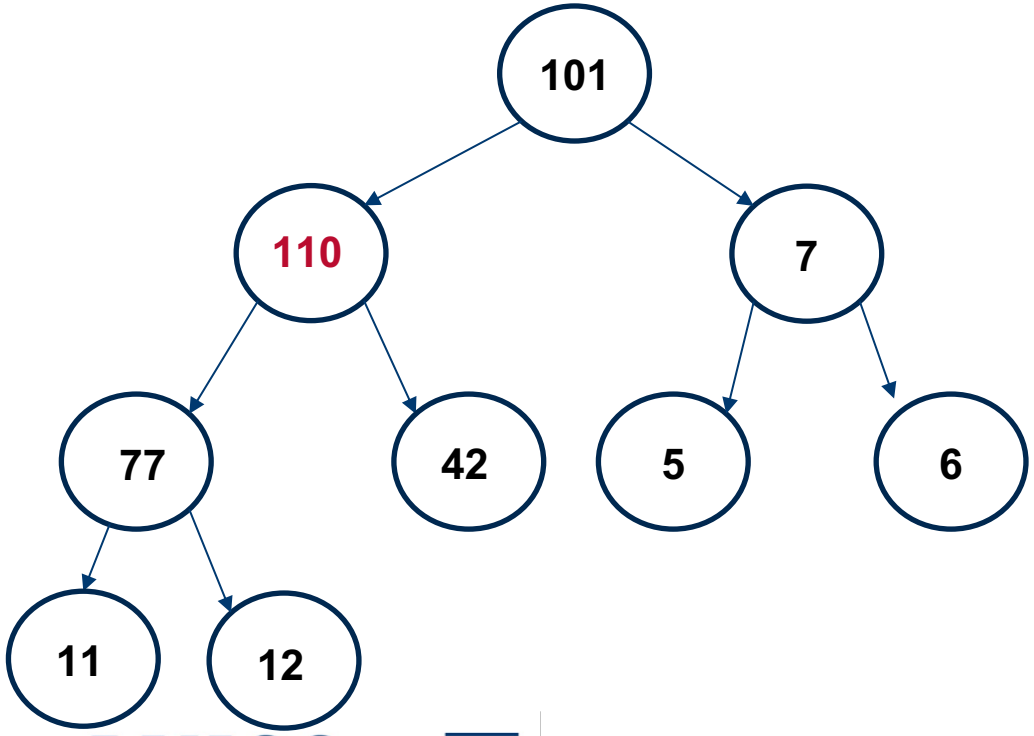


We repeat the process over and over until we reach The final position

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	101	110	7	77	42	5	6	11	12

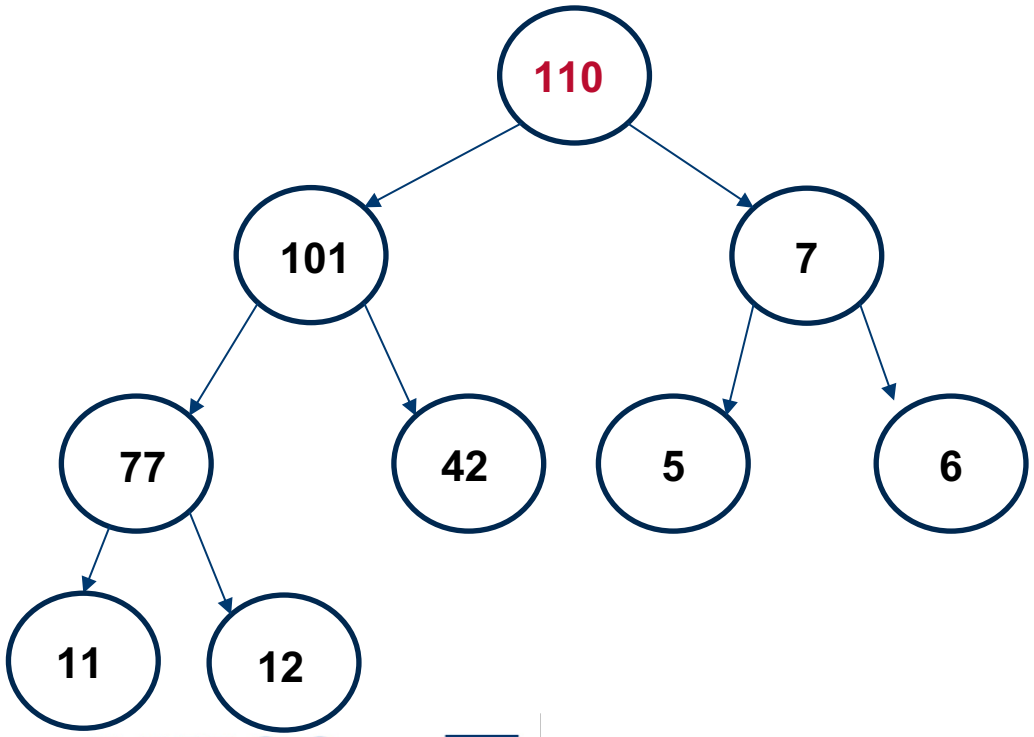


We repeat the process over and over until we reach The final position

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12

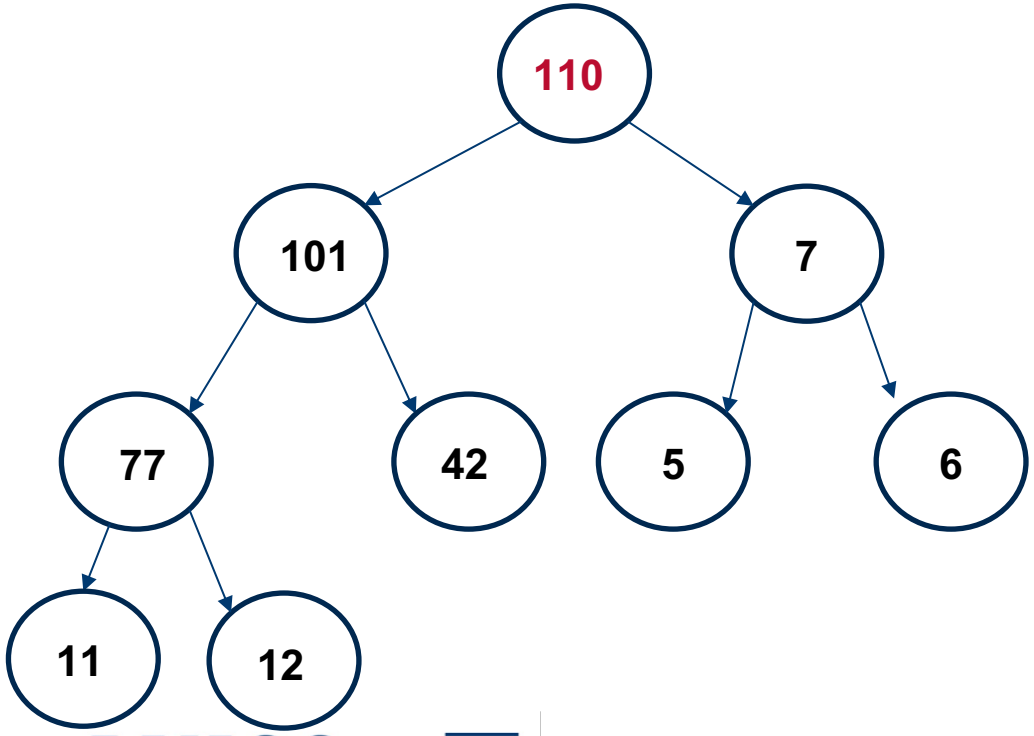


We repeat the process over and over until we reach The final position

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



We repeat the process over and over until we reach The final position

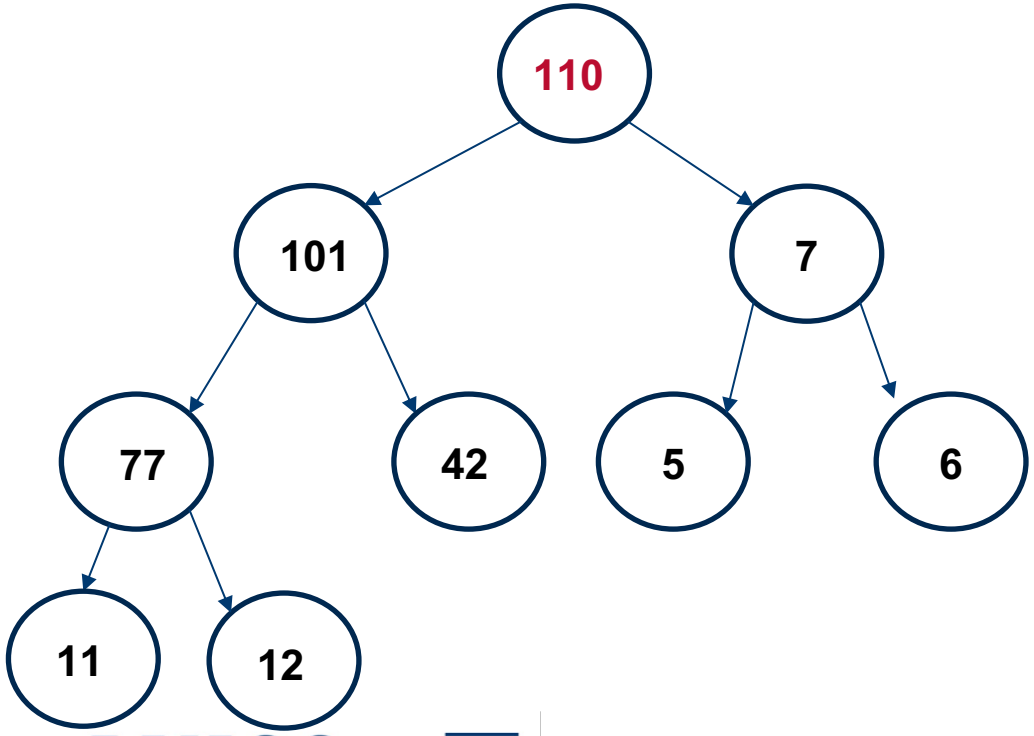
DONE!

How many operation we did to place the new element In the correct position?

LUISS

Heaps – Insert an element

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



We repeat the process over and over until we reach The final position

DONE!

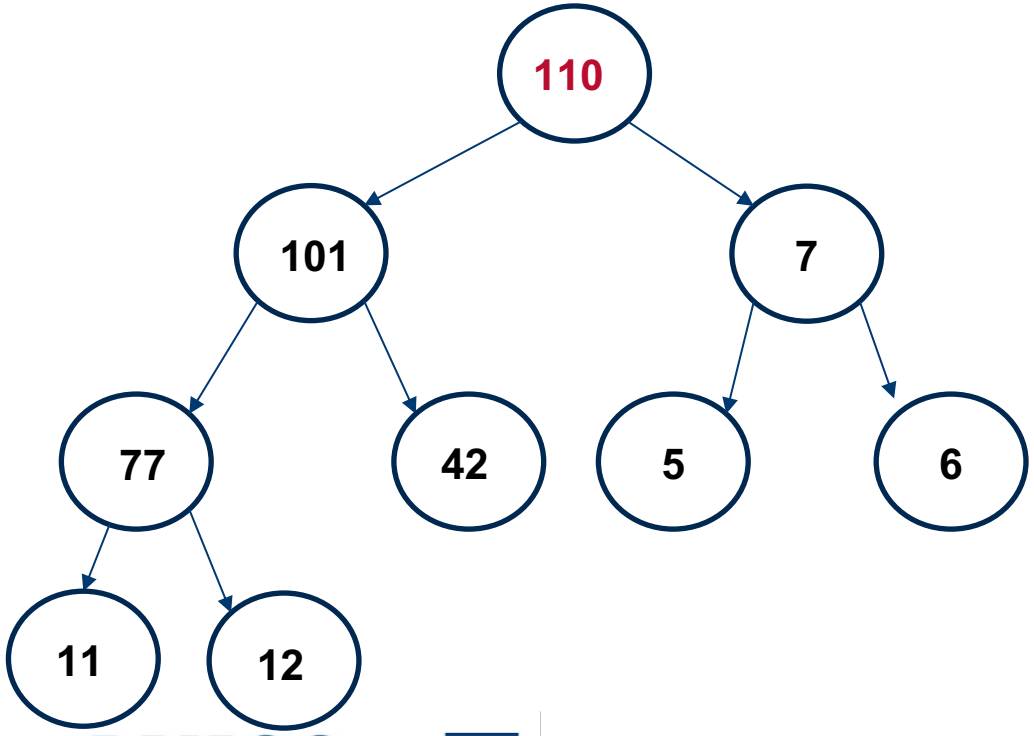
How many operation we did to place the new element In the correct position?

O(log n)

LUISS

Heaps – Max

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Value:	**110**	**101**	**7**	**77**	**42**	**5**	**6**	**11**	**12**

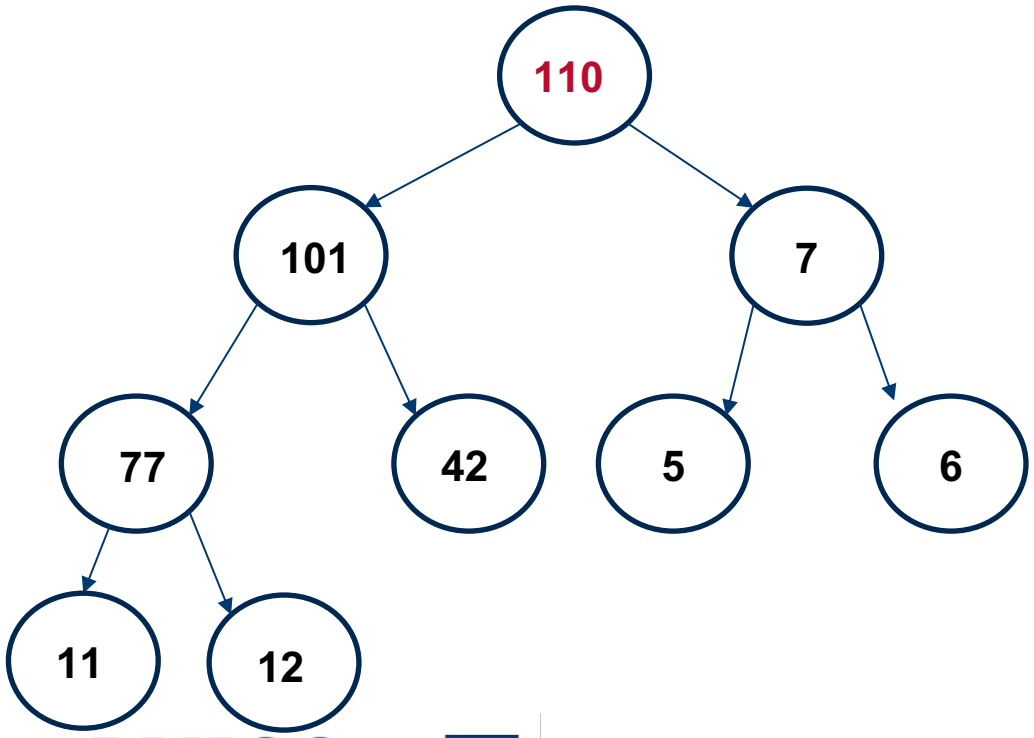


Where is the maximum element?

LUISS

Heaps – Max

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



Where is the maximum element?

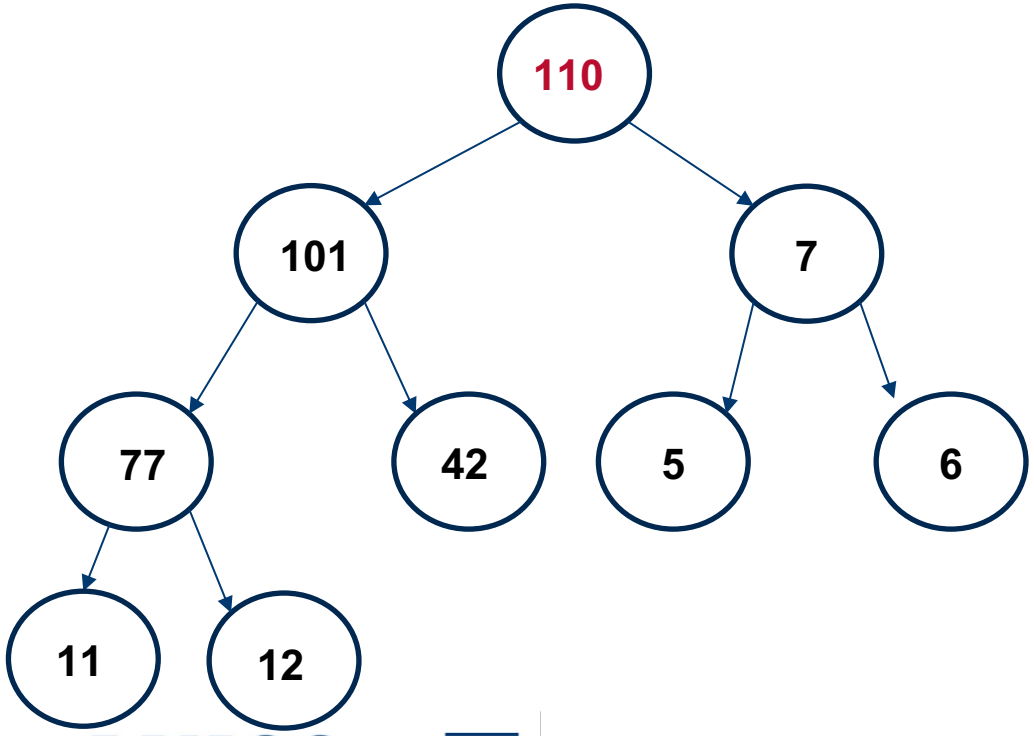
It is the root so it is in position 0

Cost?

LUISS

Heaps – Max

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Value:	110	101	7	77	42	5	6	11	12



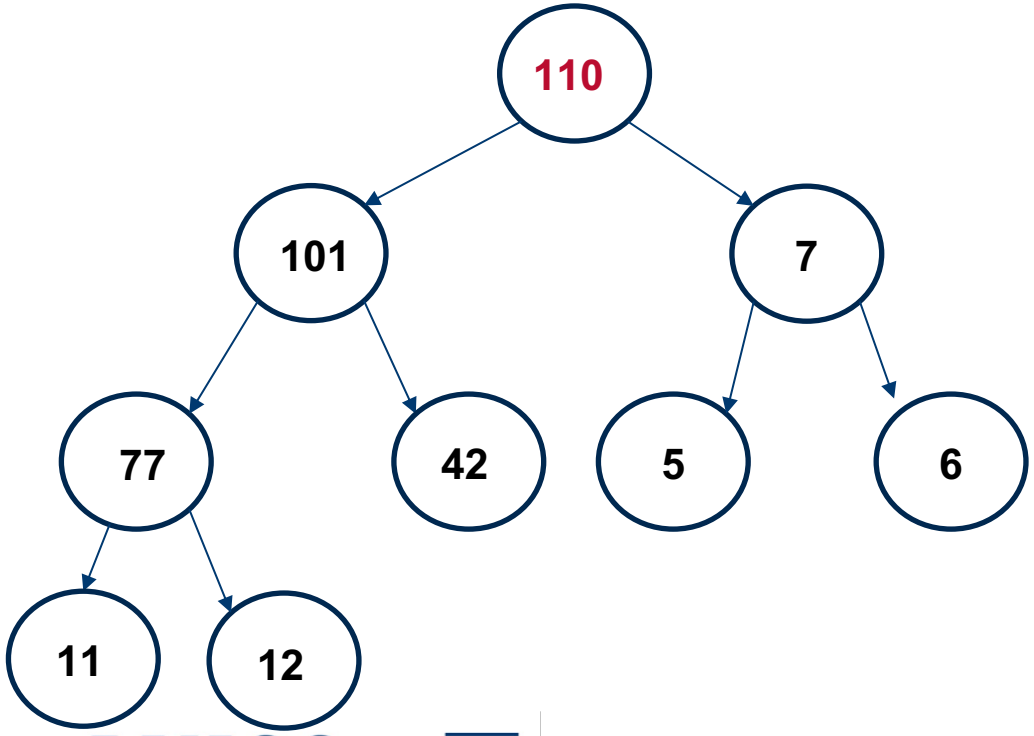
Where is the maximum element?

It is the root so it is in position 0

Cost? O(1)

Heaps – Max

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Value:	110	101	7	77	42	5	6	11	12

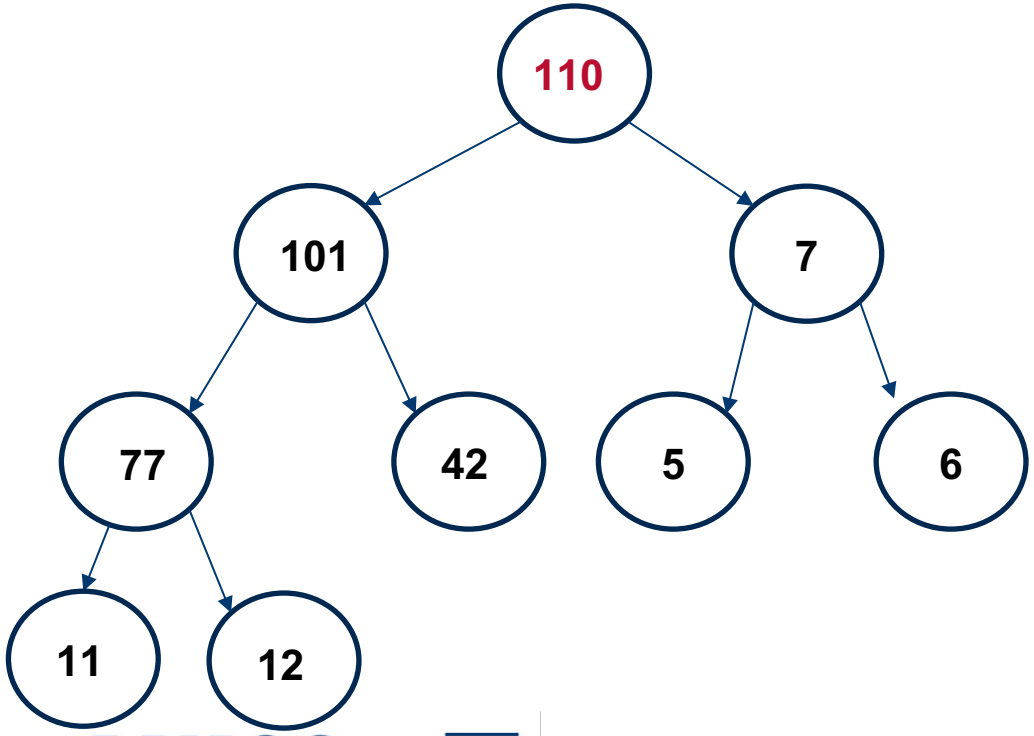


What is the cost to extract the maximum value?

LUISS

Heaps – Max

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



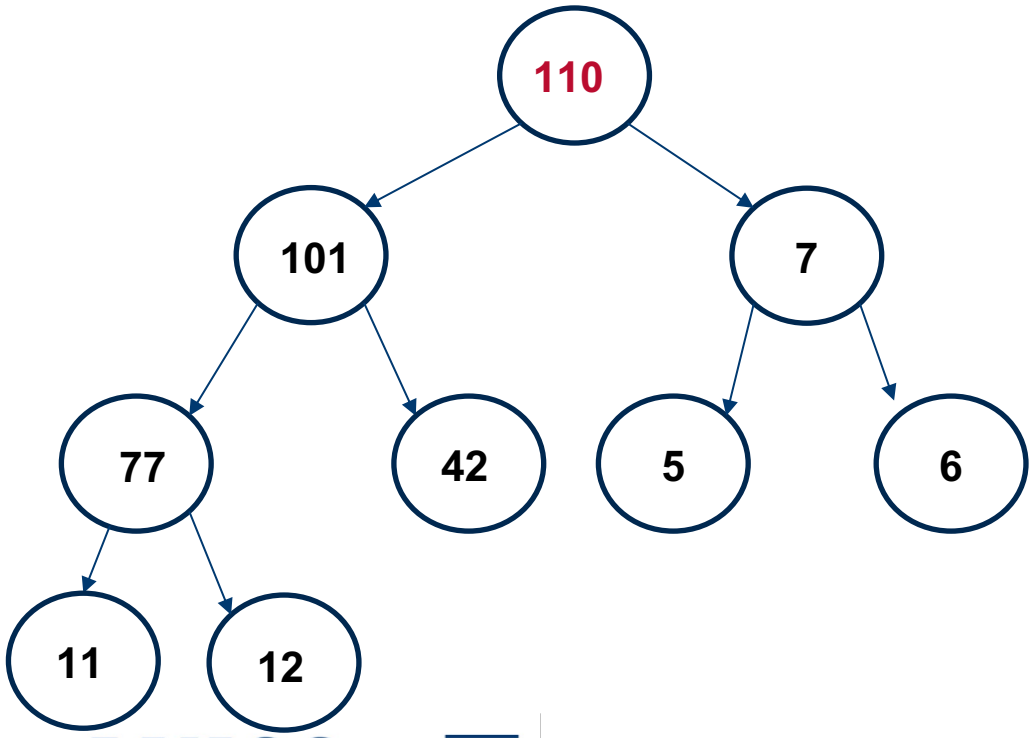
What is the cost to extract the maximum value?

O(log n)

LUISS

Heaps – Max

Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



What is the cost to extract the maximum value?

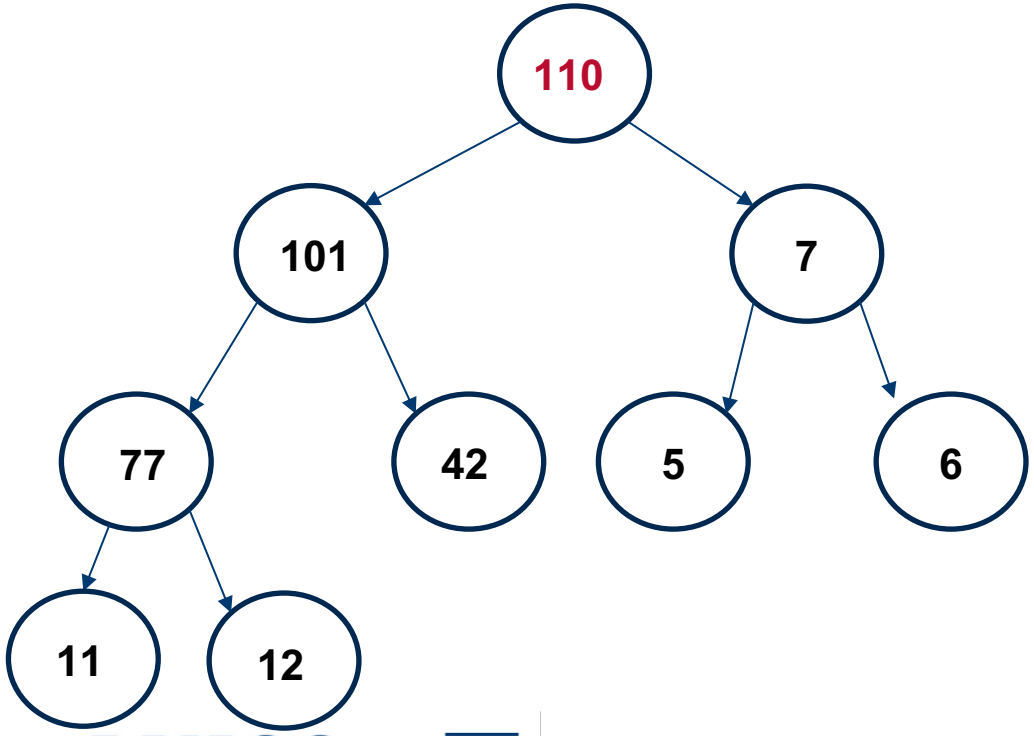
O(log n)

Why?

LUISS

Heaps – Max

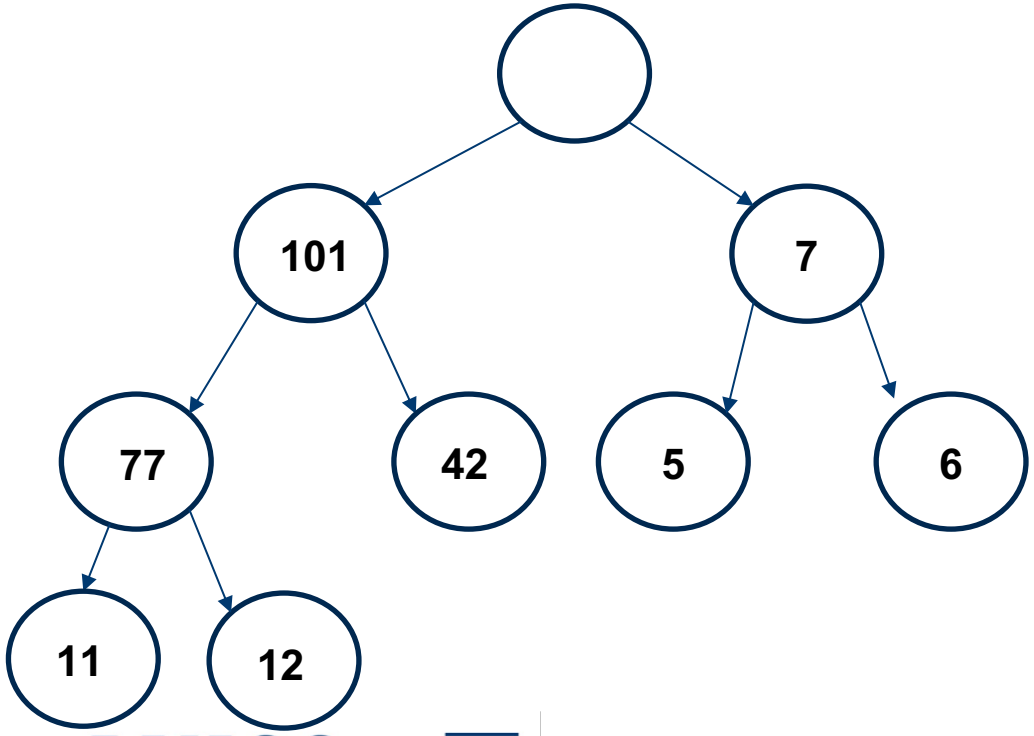
Index:	0	1	2	3	4	5	6	7	8
Value:	110	101	7	77	42	5	6	11	12



First of all we have to remove the element in position 0, namely, the grater element

Heaps – Max

Index:	0	1	2	3	4	5	6	7	8
Value:		101	7	77	42	5	6	11	12



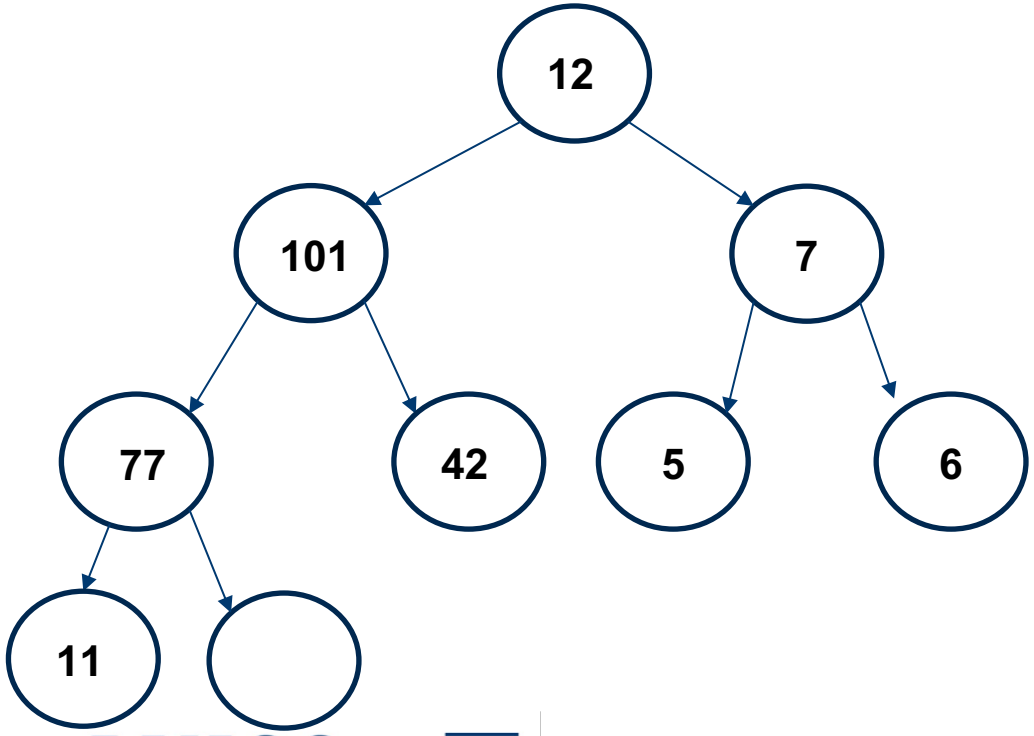
First of all we have to remove the element in position 0, namely, the grater element

Then we have to place the last element of the list on top of the list

LUISS

Heaps – Max

Index:	0	1	2	3	4	5	6	7	8
Value:	12	101	7	77	42	5	6	11	



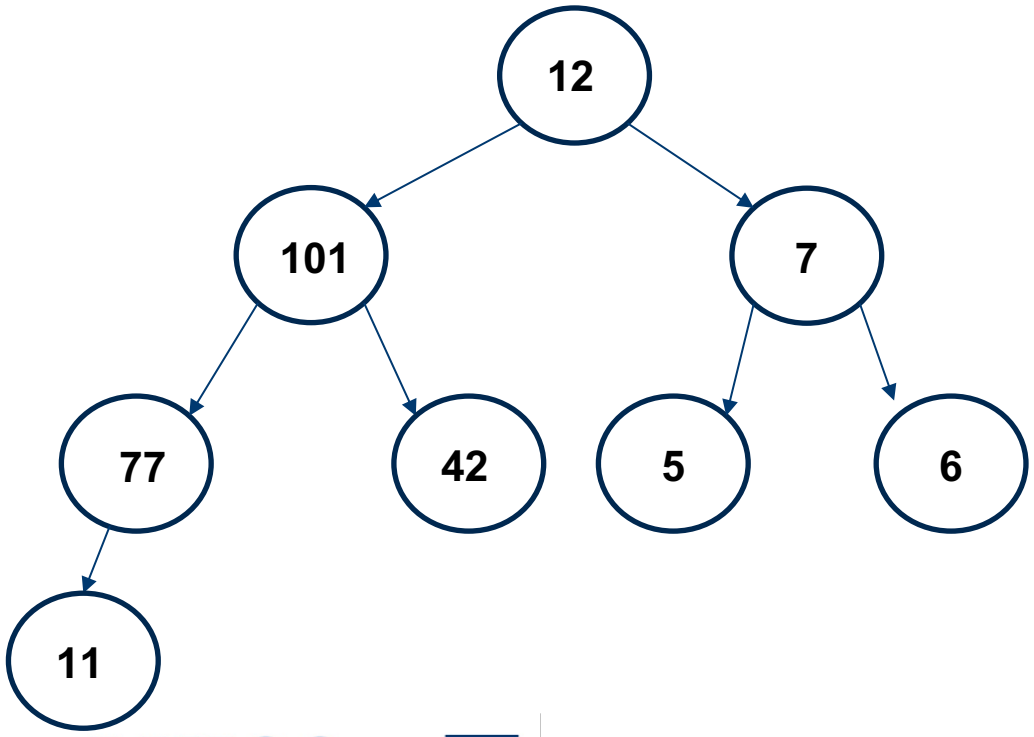
First of all we have to remove the element in position 0, namely, the grater element

Then we have to place the last element of the list on top of the list

LUISS

Heaps – Max

Index:	0	1	2	3	4	5	6	7
Value:	12	101	7	77	42	5	6	11



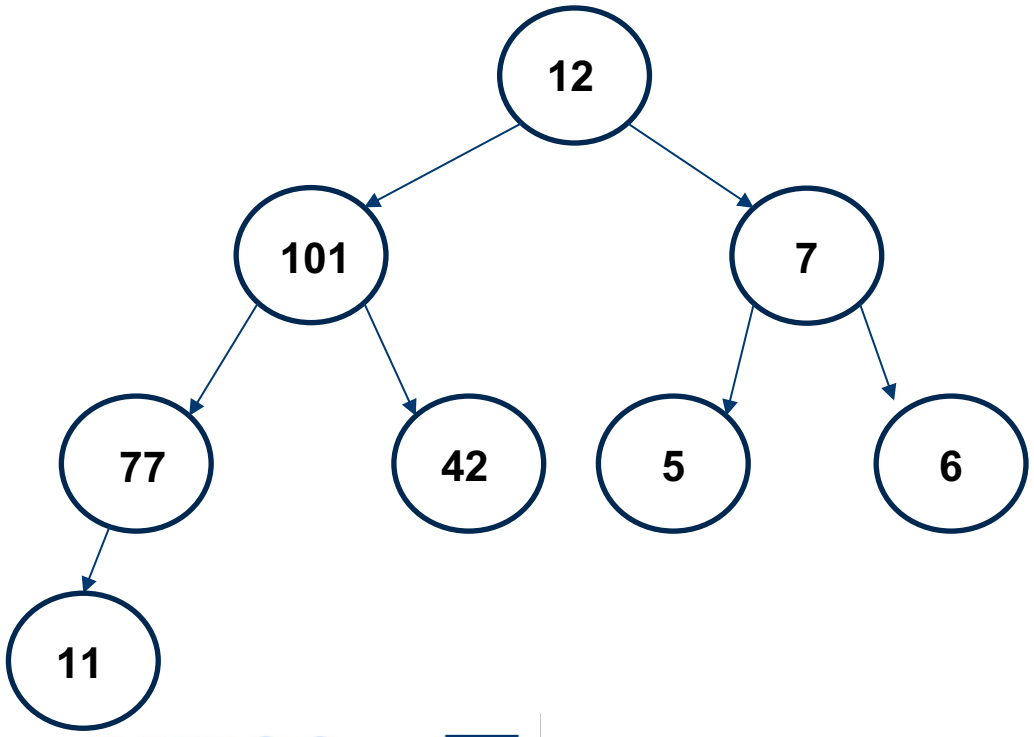
First of all we have to remove the element in position 0, namely, the grater element

Then we have to place the last element of the list on top of the list

And we can shrink the list

Heaps – Max

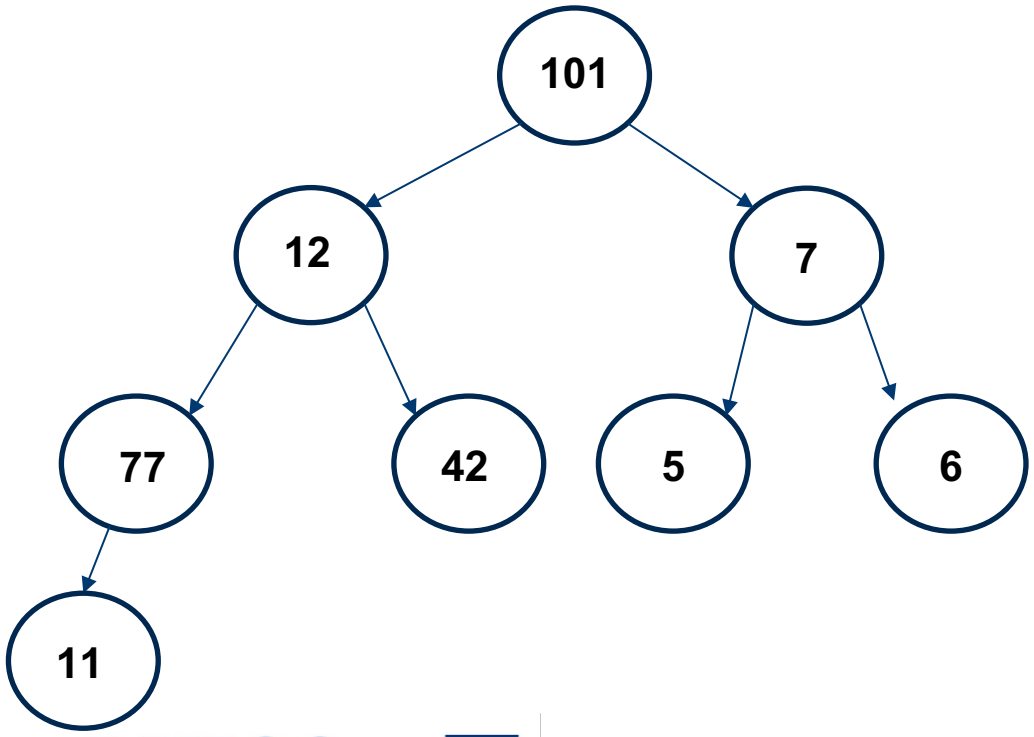
Index:	0	1	2	3	4	5	6	7
Value:	12	101	7	77	42	5	6	11



Finally we have to call max-heapify on the list starting From the first element

Heaps – Max

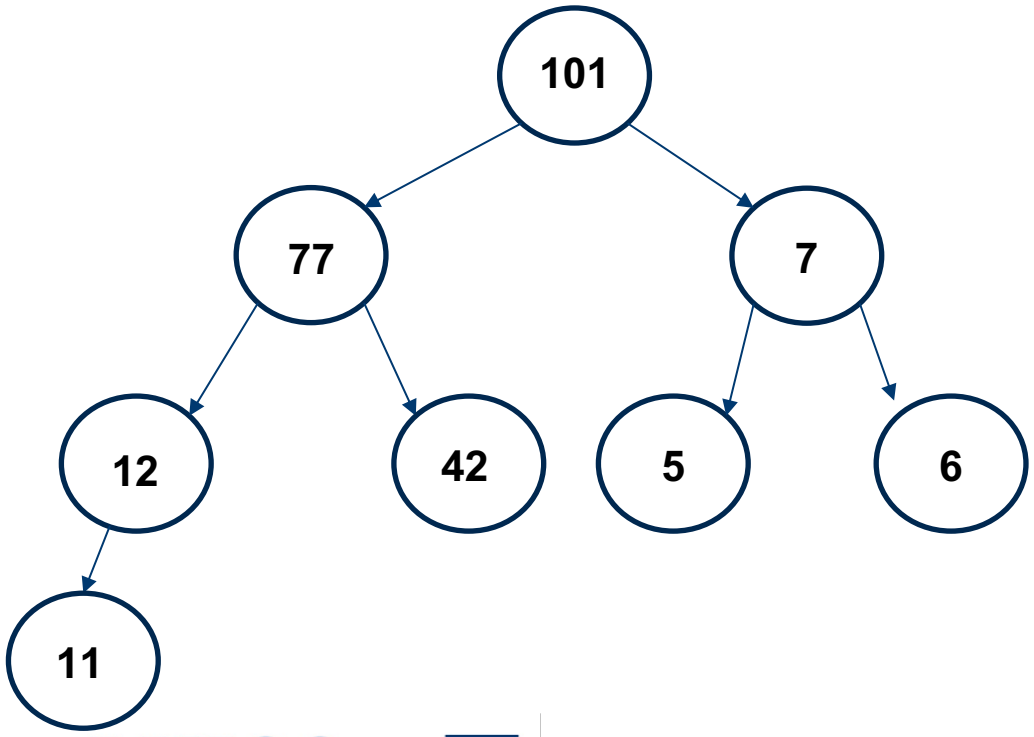
Index:	0	1	2	3	4	5	6	7
Value:	101	12	7	77	42	5	6	11



Finally we have to call max-heapify on the list starting From the first element

Heaps – Max

Index:	0	1	2	3	4	5	6	7
Value:	101	77	7	12	42	5	6	11



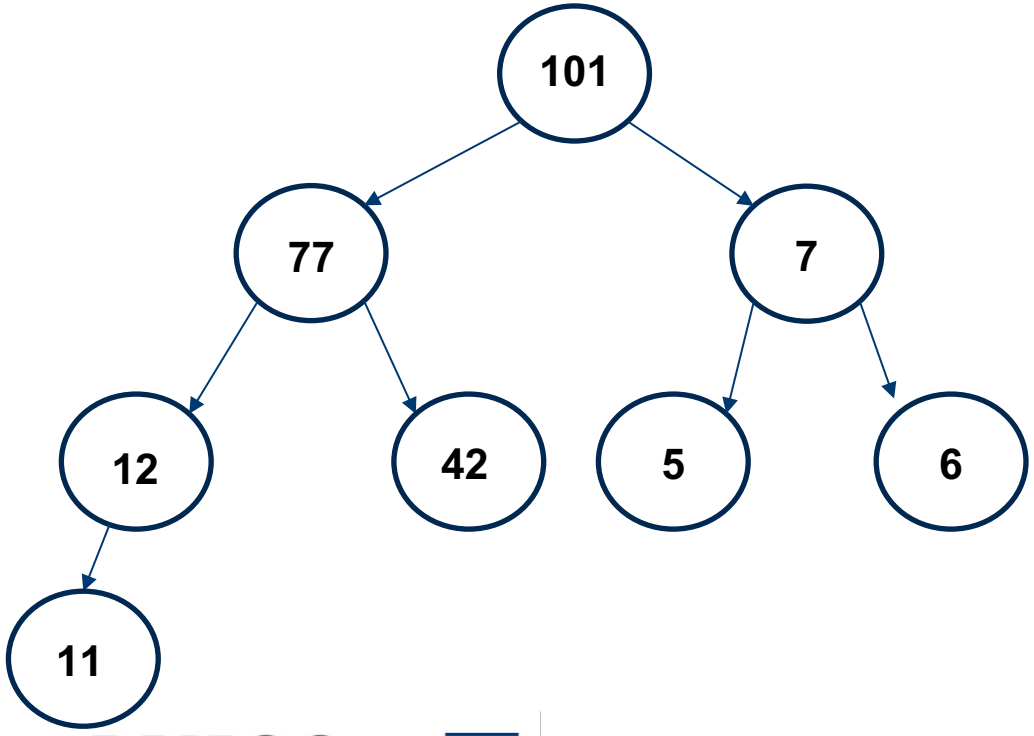
Finally we have to call max-heapify on the list starting From the first element

Until the final position is reached!

LUISS

Heaps – Max

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Value:	**101**	**77**	**7**	**12**	**42**	**5**	**6**	**11**



This is basically a single iteration of the heap-sort!