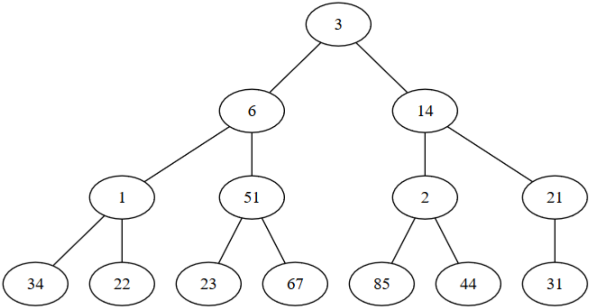
Recap: Binary Heap

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31



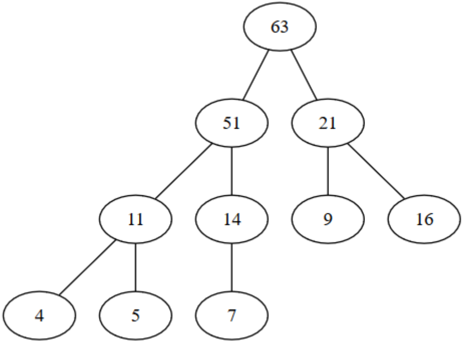
- A heap can be used as representation for a Priority Queue and it has two specific operations:
  - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
  - remove (we always remove the root of the heap - no other element can be removed).

Heap:
  cap: Integer
  len: Integer
  elems: TElem[]

- Depending on the problem, you might need to have a *relation* as well, as part of the heap.
- For the implementation we will assume that we have a MAX-HEAP.

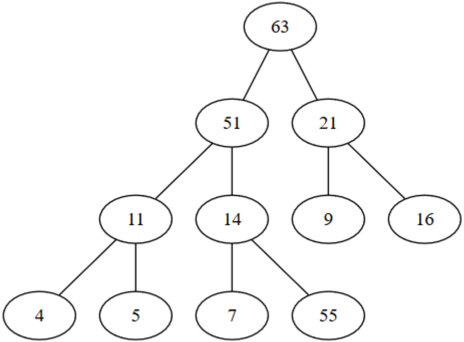
Binary Heap - Add - example

- Consider the following (MAX) heap:



- Let's add the number 55 to the heap.

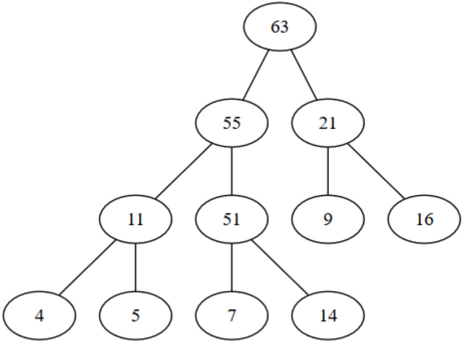
- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).



- *Heap property* is not kept: 14 has as child node 55 (since it is a MAX-heap, each node has to be greater than or equal to its descendants).
- In order to restore the heap property, we will start a *bubble-up* process: we will keep swapping the value of the new node with the value of its parent node, until it gets to its final place. No other node from the heap is changed.

Binary Heap - Add - example

- When *bubble-up* ends:



Binary Heap - add

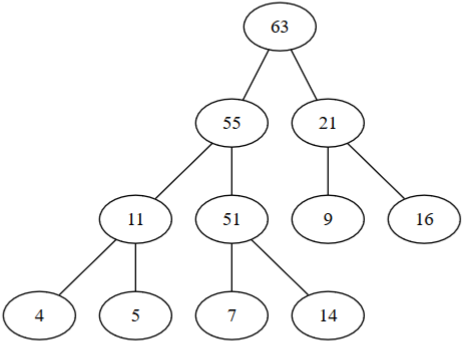
```
subalgorithm add(heap, e) is:
//heap - a heap
//e - the element to be added
   if heap.len = heap.cap then
      @ resize
   end-if
   heap.elems[heap.len+1] ← e
   heap.len ← heap.len + 1
   bubble-up(heap, heap.len)
end-subalgorithm
```

Binary Heap - add

```
subalgorithm bubble-up (heap, p) is:
//heap - a heap
//p - position from which we bubble the new node up
   poz ← p
   elem ← heap.elems[p]
   parent ← p / 2
   while poz > 1 and elem > heap.elems[parent] execute
      //move parent down
      heap.elems[poz] ← heap.elems[parent]
      poz ← parent
      parent ← poz / 2
   end-while
   heap.elems[poz] ← elem
end-subalgorithm
```

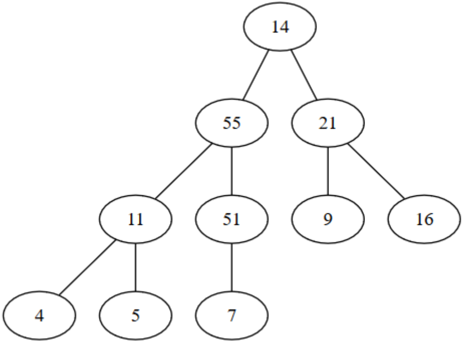
- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

- From a heap we can only remove the root element.



Binary Heap - Remove - example

- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.

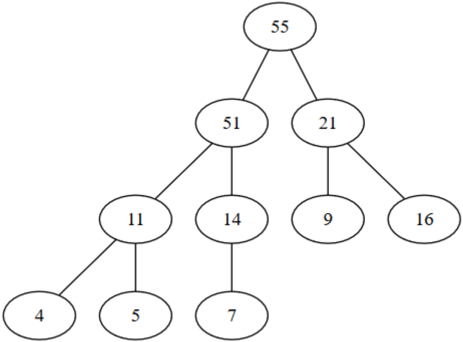


Binary Heap - Remove - example

- *Heap property* is not kept: the root is no longer the maximum element.
- In order to restore the heap property, we will start a *bubble-down* process, where the new node will be swapped with its maximum child, until it becomes a leaf, or until it will be greater than both children.

Binary Heap - Remove - example

- When the bubble-down process ends:



Binary Heap - remove

function remove(heap) **is:**
//heap - is a heap
    if heap.len = 0 **then**
        @ error - empty heap
    end-if
    deletedElem ← heap.elems[1]
    heap.elems[1] ← heap.elems[heap.len]
    heap.len ← heap.len - 1
    bubble-down(heap, 1)
    remove ← deletedElem
end-function

```
subalgorithm bubble-down(heap, p) is:
//heap - is a heap
//p - position from which we move down the element
   poz ← p
   elem ← heap.elems[p]
   while poz < heap.len execute
      maxChild ← -1
      if poz * 2 ≤ heap.len then
      //it has a left child, assume it is the maximum
         maxChild ← poz*2
      end-if
      if poz*2+1 ≤ heap.len and heap.elems[2*poz+1]>heap.elems[2*poz] th
      //it has two children and the right is greater
         maxChild ← poz*2 + 1
      end-if
//continued on the next slide...
```

```
    if maxChild ≠ -1 and heap.elems[maxChild] > elem then
        tmp ← heap.elems[poz]
        heap.elems[poz] ← heap.elems[maxChild]
        heap.elems[maxChild] ← tmp
        poz ← maxChild
    else
        poz ← heap.len + 1
        //to stop the while loop
    end-if
  end-while
end-subalgorithm
```

- Complexity: $O(log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $log_2 n$ (best case scenario)?

Questions

- In a max-heap where can we find the:
  - maximum element of the array?
  - minimum element of the array?
- Assume you have a MAX-HEAP and you need to add an operation that returns the minimum element of the heap. How would you implements this operation, using constant time and space? (Note: we only want to return the minimum, we do not want to be able to remove it).

- Consider an initially empty Binary MAX-HEAP and insert the elements 8, 27, 13, 15*, 32, 20, 12, 50*, 29, 11* in it. Draw the heap in the tree form after the insertion of the elements marked with a * (3 drawings). Remove 3 elements from the heap and draw the tree form after every removal (3 drawings).

- Insert the following elements, in this order, into an initially empty MIN-HEAP: 15, 17, 9, 11, 5, 19, 7. Remove all the elements, one by one, in order from the resulting MIN HEAP. Draw the heap after every second operation (after adding 17, 11, 19, etc.)