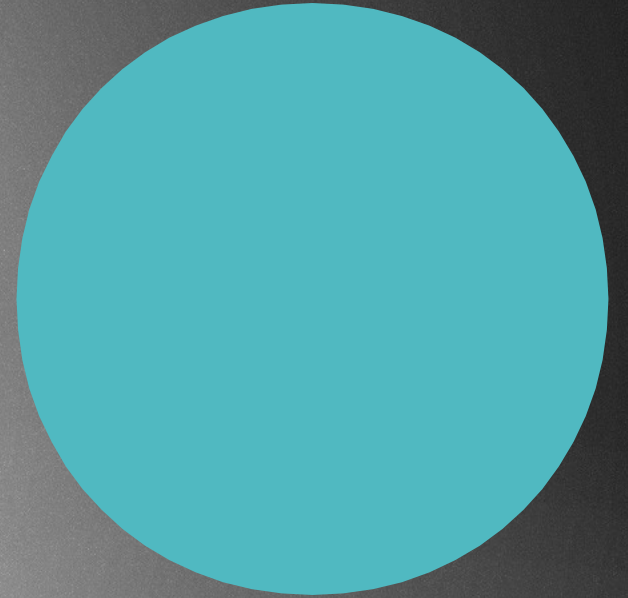



PRIORITY QUEUES



- 
- ▶ It is an abstract data type such as stack or queue
 - ▶ BUT every item has an additional property: a priority value
 - ▶ In a priority queue, an element with high priority is served before an element with lower priority
 - ▶ Priority queues are usually implemented with heaps, but it can be implemented with self balancing trees as well
 - ▶ Very similar to queues with some modification: when we would like to get the next item → the highest priority element is retrieved first !!!
 - ▶ No FIFO structure here !!!

Integer values as priorities

Sometimes: we do not specify the priority // for example when implementing heap

- the value of an integer or double can be interpreted as a priority
- so we can omit the priority when inserting new integers or doubles
- the priority of 10 will be greater than that of 5 because $10 > 5$ so there is no need to store the priority in another variable !!!

Operations

`insertWithPriority(data, priority)` // sometimes we do not specify the priority

This method will insert new item into the priority queue. We have to specify the data we want to insert and the priority associated with the given data

`getHighestPriorityElement()`

Returns the element with highest priority: we have to reconstruct the heap
Max heap: returns maximum element
Min heap: returns minimum element

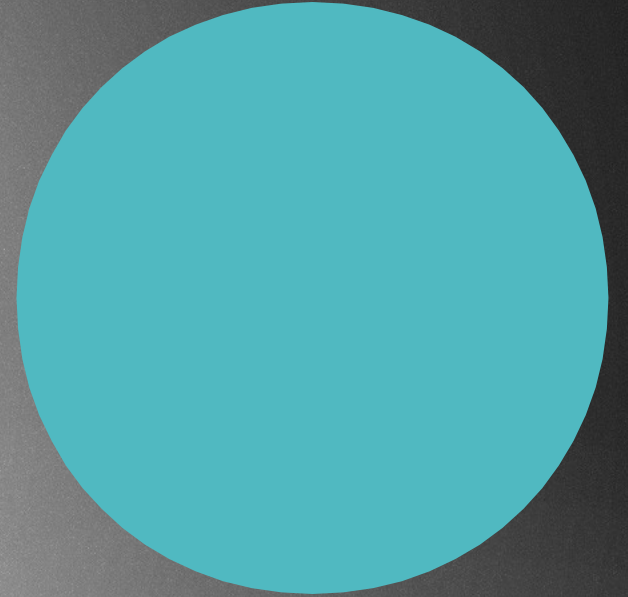
`peek()`

Returns the element with highest priority: the structure of the heap does not change !!!

Sorting

- ▶ The concept of priority queues naturally suggest a sorting algorithm
- ▶ Insert all the elements to be sorted into a priority queue
- ▶ Sequentially remove them: it will be the sorted order !!!
- ▶ Why is it working?
 - ▶ We have been discussing that priority queues rely heavily on priorities
 - ▶ We take out items → the one with highest priority will be returned
 - ▶ Result: sequency of decreasing priorities
 - ▶ This is the sorted order
 - ▶ For example: tree sort, heapsort

HEAPS



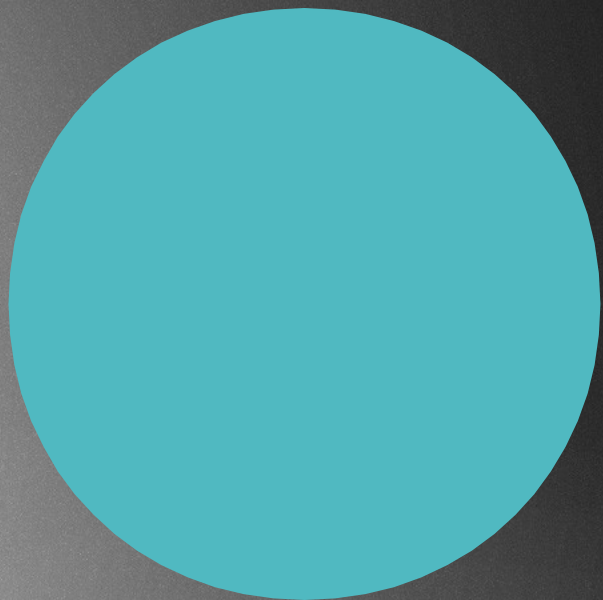
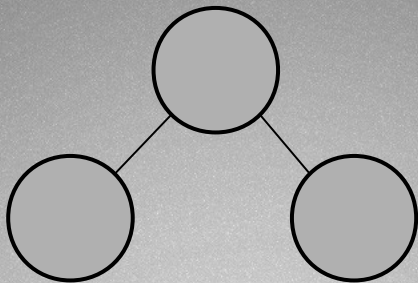
Heap

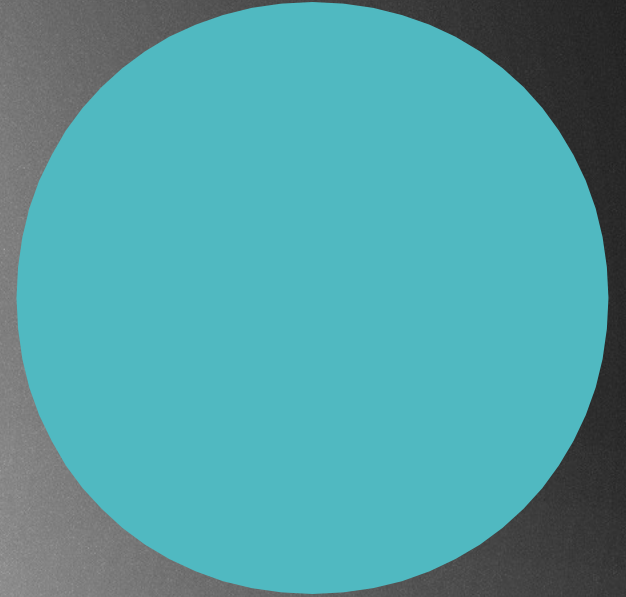
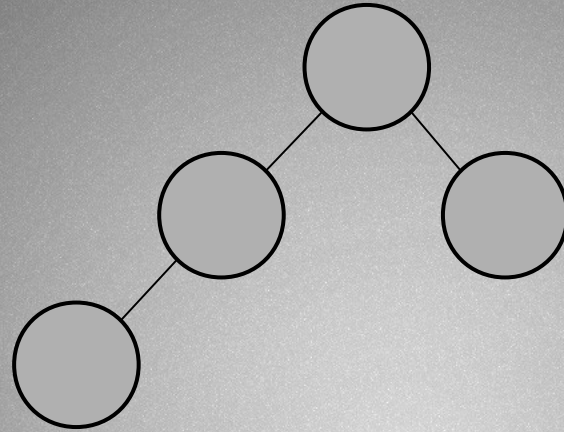
- ▶ It is basically a binary tree
- ▶ Two main binary heap types: min and max heap
- ▶ In a max heap, the keys of parent nodes are always greater than or equal to those of the children → the highest key is in the root node.
- ▶ In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node
- ▶ It is complete: it cannot be unbalanced !!! We insert every new item to the next available place
- ▶ Applications: Dijkstra algorithm, Prim's algorithm
- ▶ The heap is one maximally efficient implementation of a priority queue ADT
- ▶ It has nothing to do with the pool of memory from which dynamically allocated memory is allocated

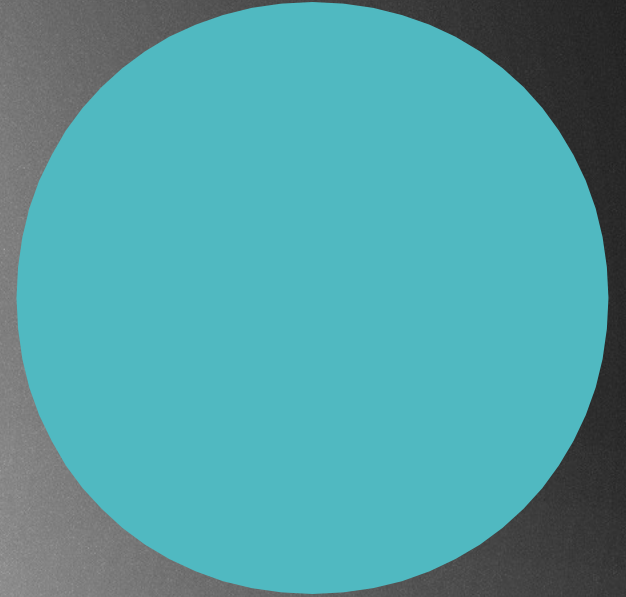
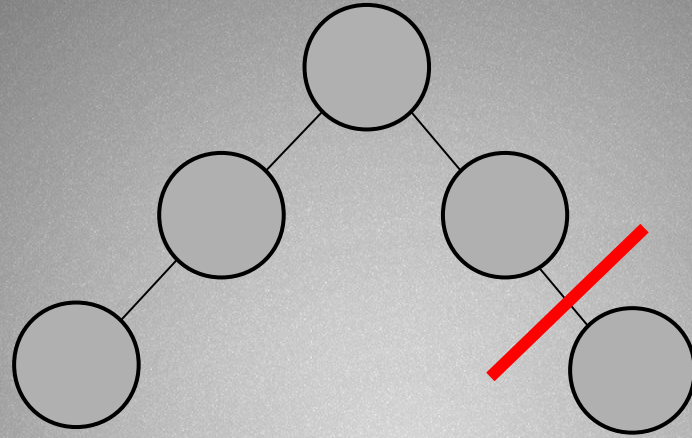
Heap properties

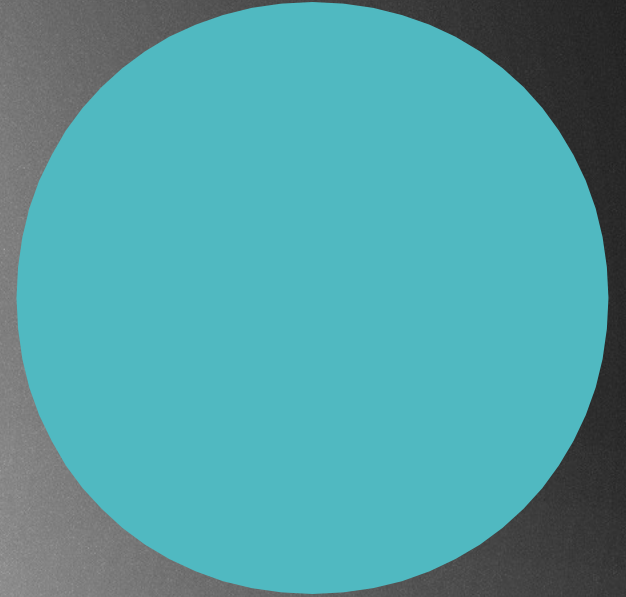
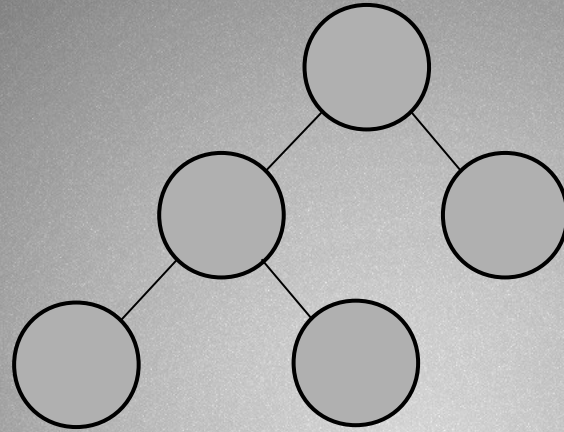
- ▶ 1.) **Complete** -> we construct the heap from left to right across each row // of course the last row may not be completely full

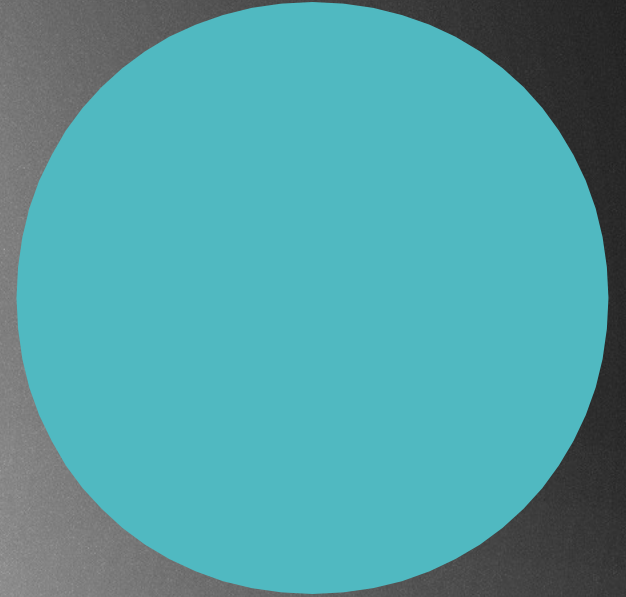
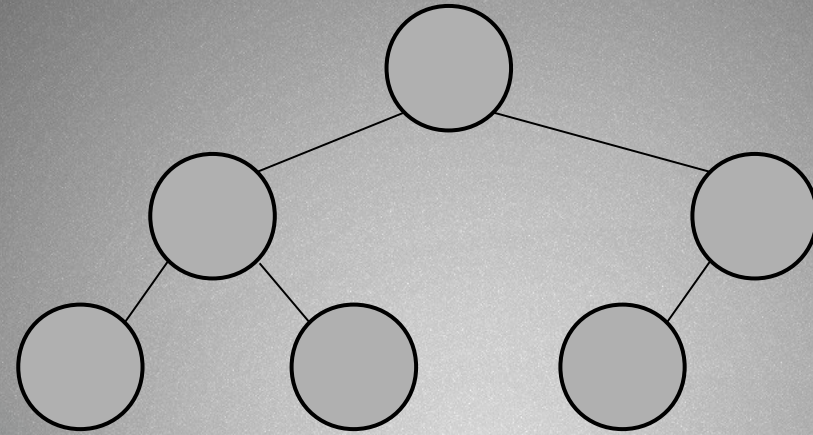
There is no missing node from left to right in a layer

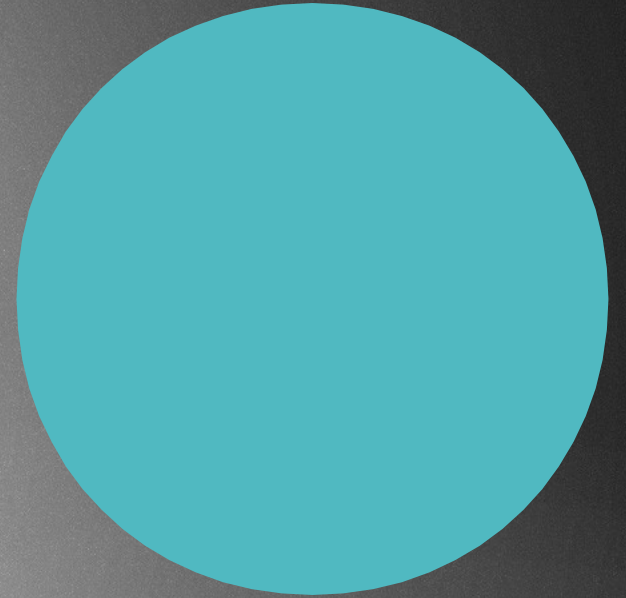
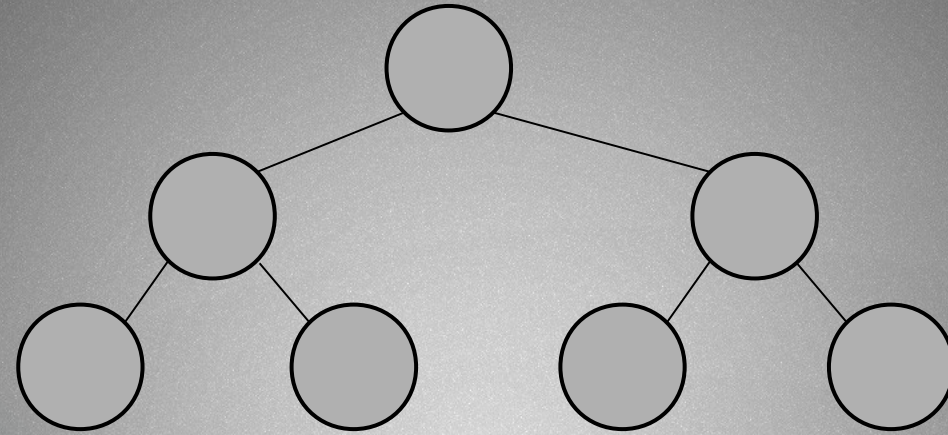


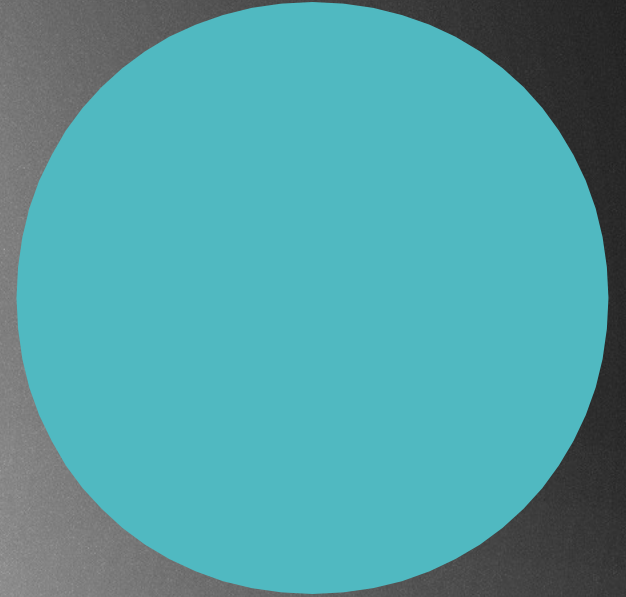
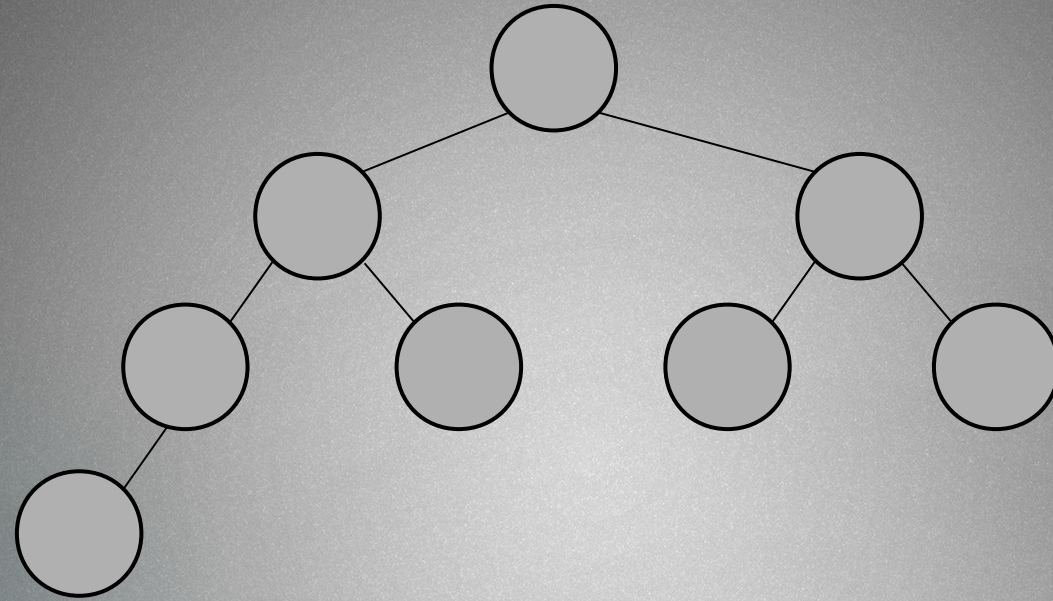




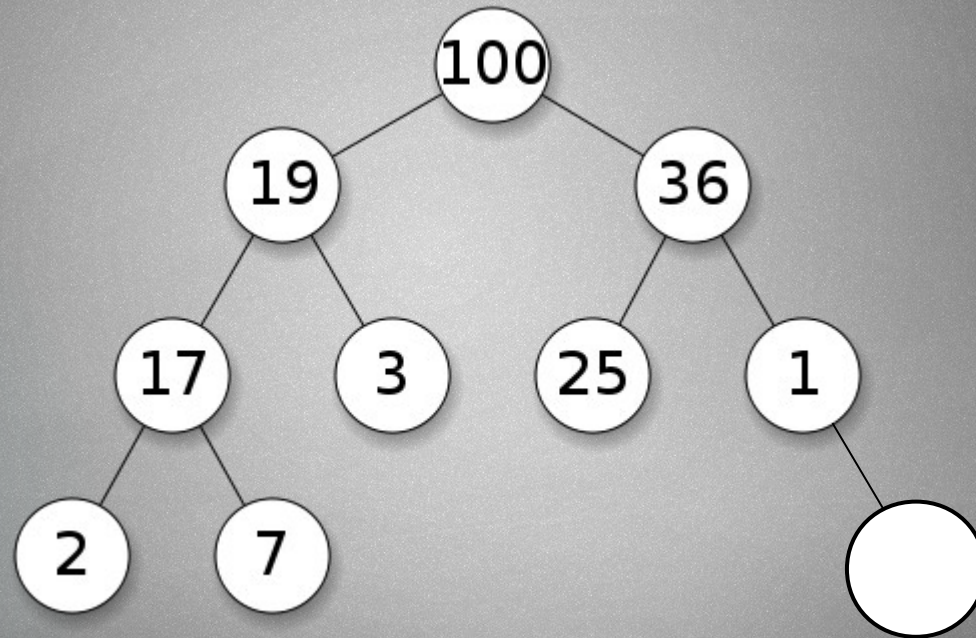








Not a binary heap



This is not a heap: not complete because of this node

Heap properties

- ▶ 1.) **Complete** -> we construct the heap from left to right across each row // of course the last row may not be completely full

There is no missing node from left to right in a layer

- ▶ 2.) In a binary heap every node can have 2 children, left child and right child

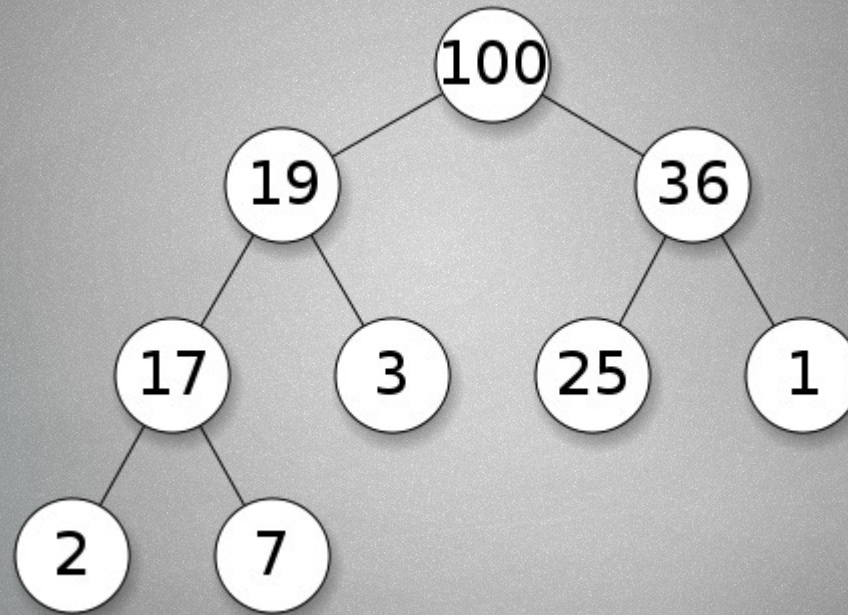
- ▶ 3.) Min heap → the parent is always smaller than the values of the children

Max heap → the parent is always greater

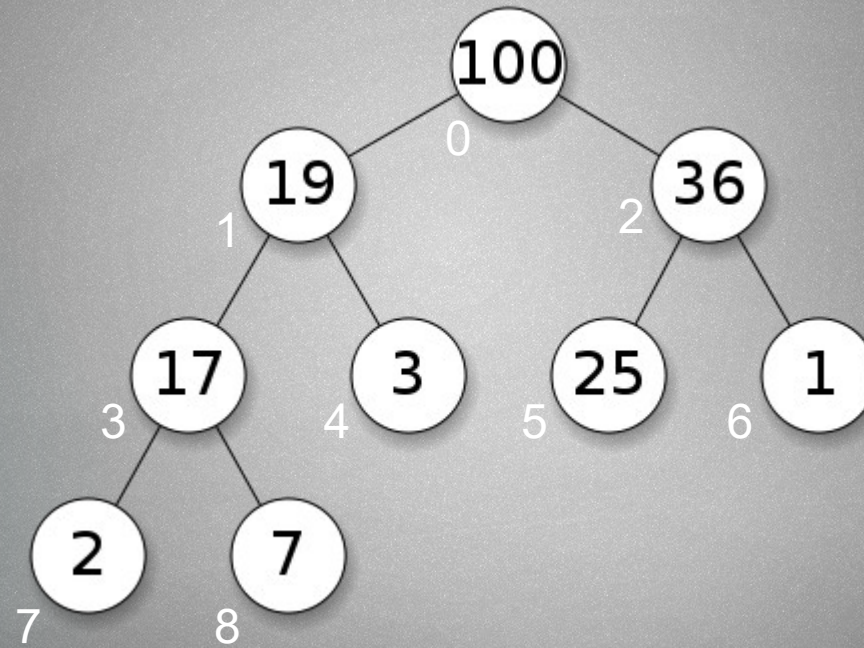
So: the root node will be the smallest/ greatest value in the heap

// $O(1)$ access !!!

Binary heap: maximum heap



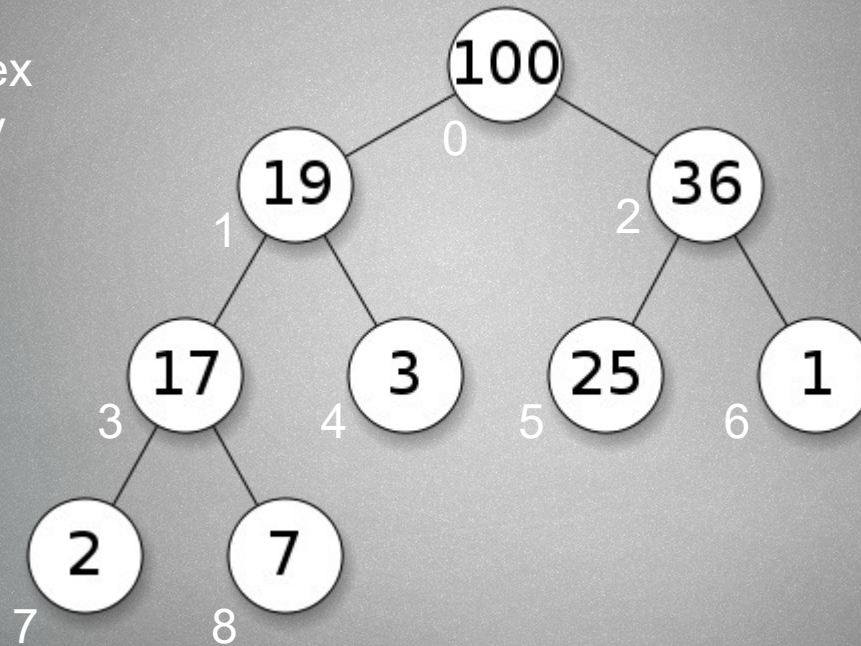
Binary heap: maximum heap



Represent heap as array

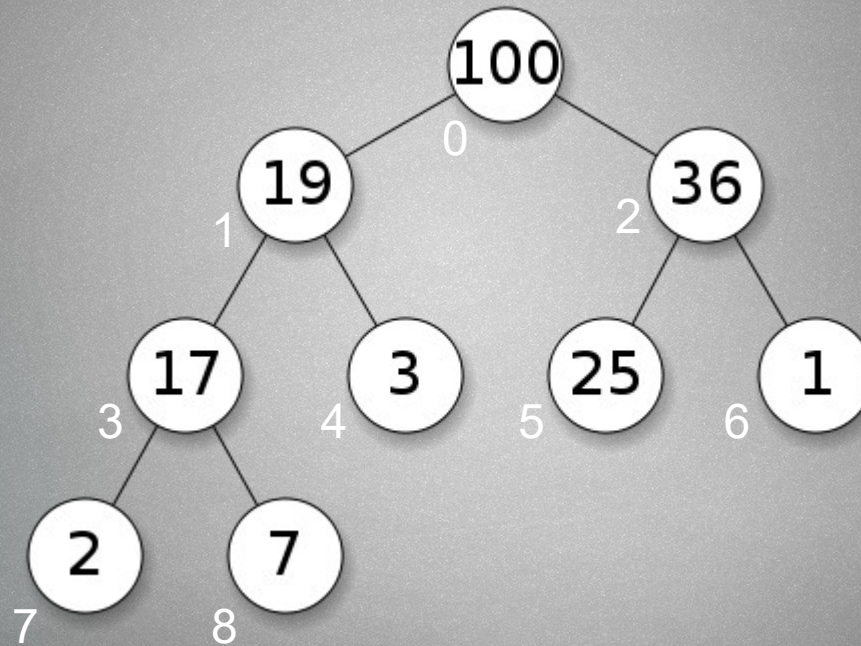
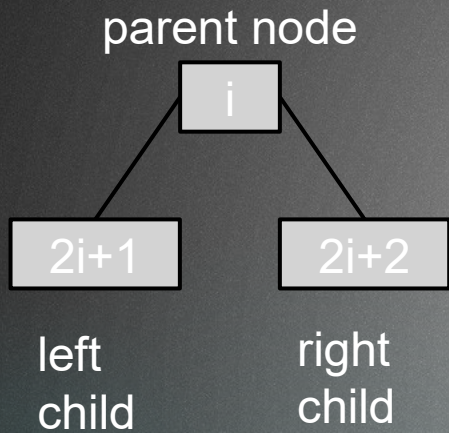
We assign indexes to every node in the heap !!!

~ the index will be the index in a one dimensional array



100	0
19	1
36	2
17	3
3	4
25	5
1	6
2	7
7	8

Represent heap as array

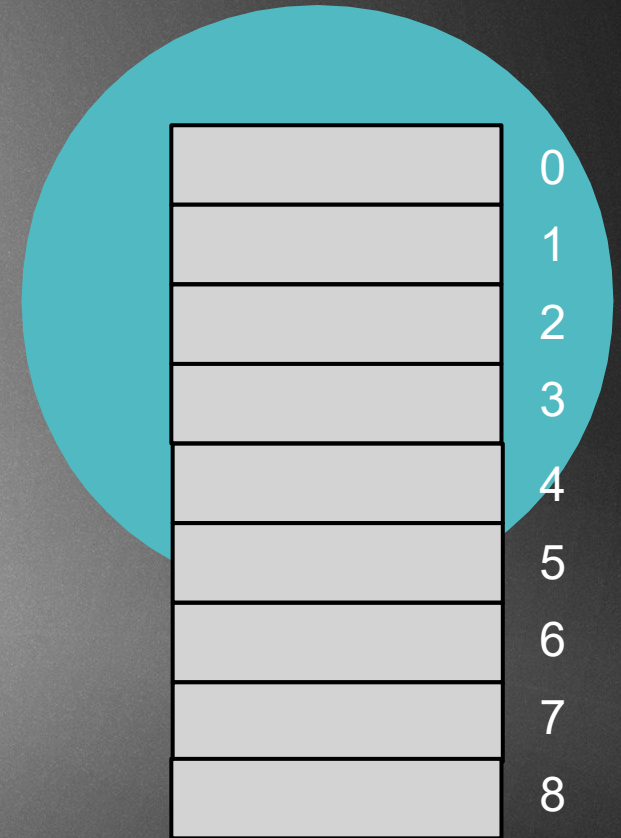


100	0
19	1
36	2
17	3
3	4
25	5
1	6
2	7
7	8

Accessing the root node: `array[0]` !!!

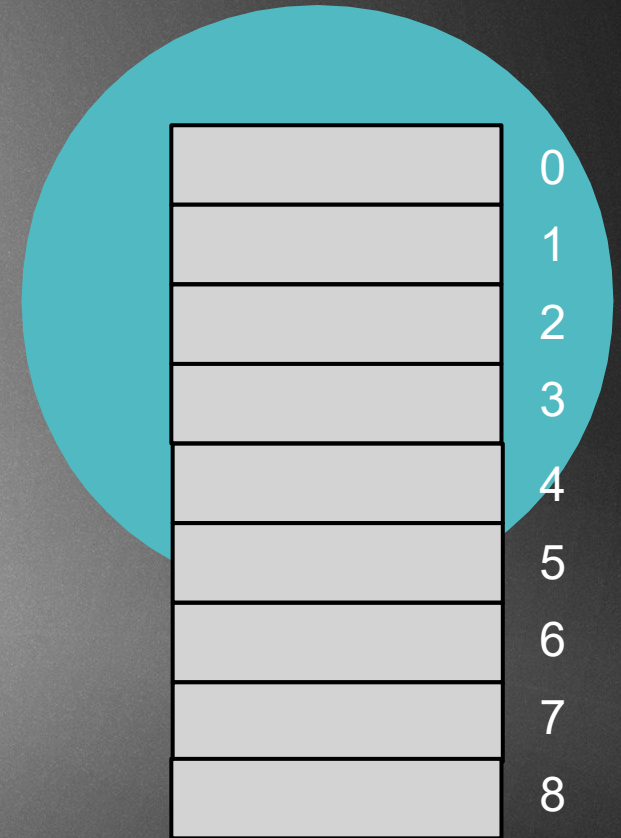
Building a heap: first we insert the data to the heap and we check whether the heap properties are met

~ if the heap properties are violated: we reconstruct the heap in order to make it a valid heap !!!
„heapify process”

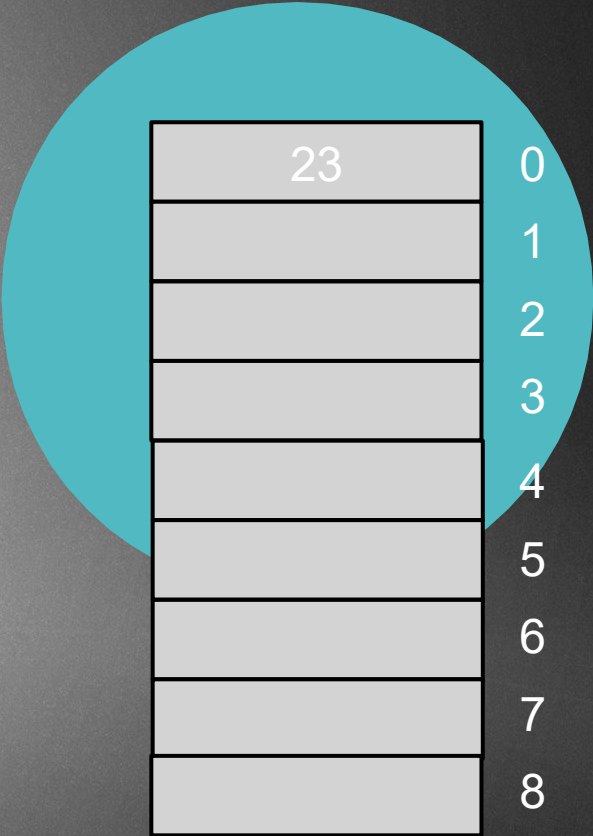


Building a heap:

Insert: 23

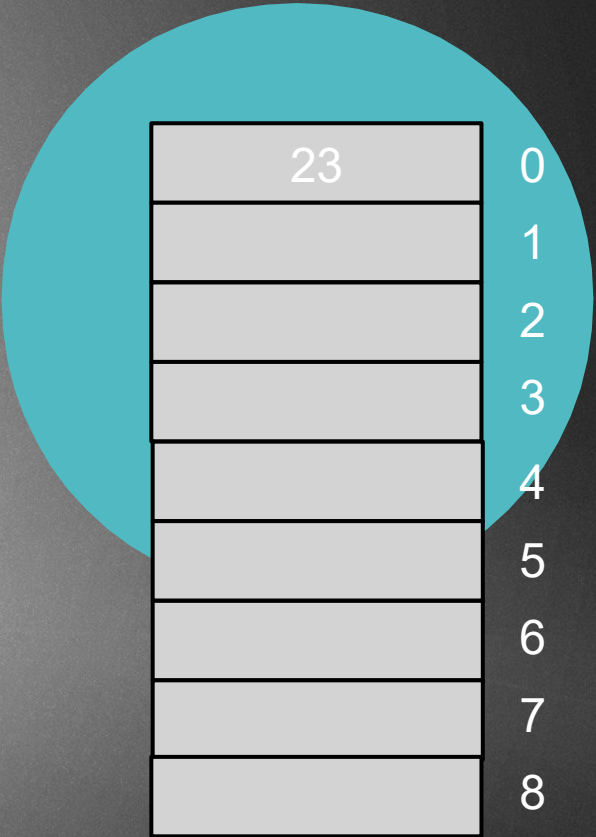


Building a heap

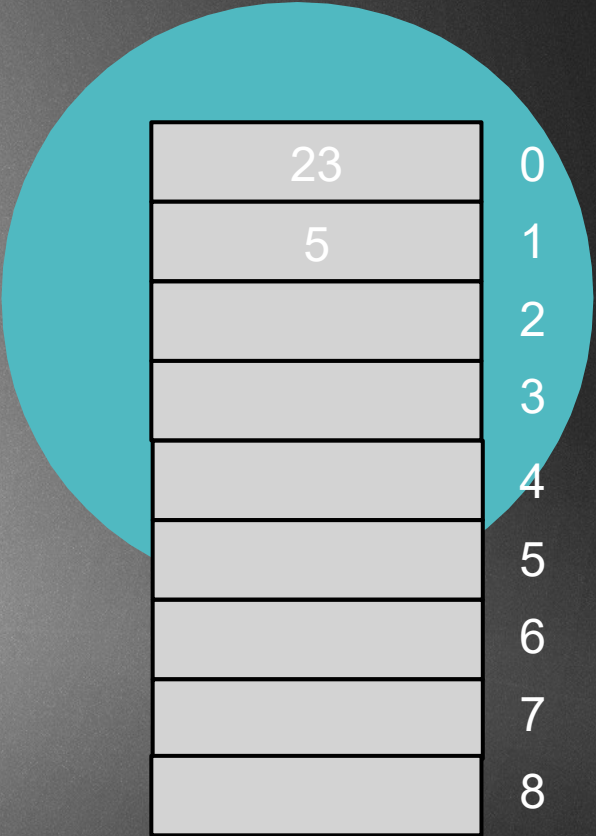
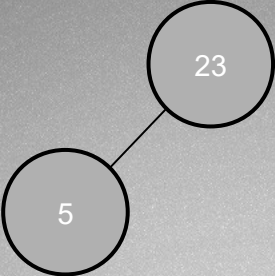


Building a heap

Insert: 5

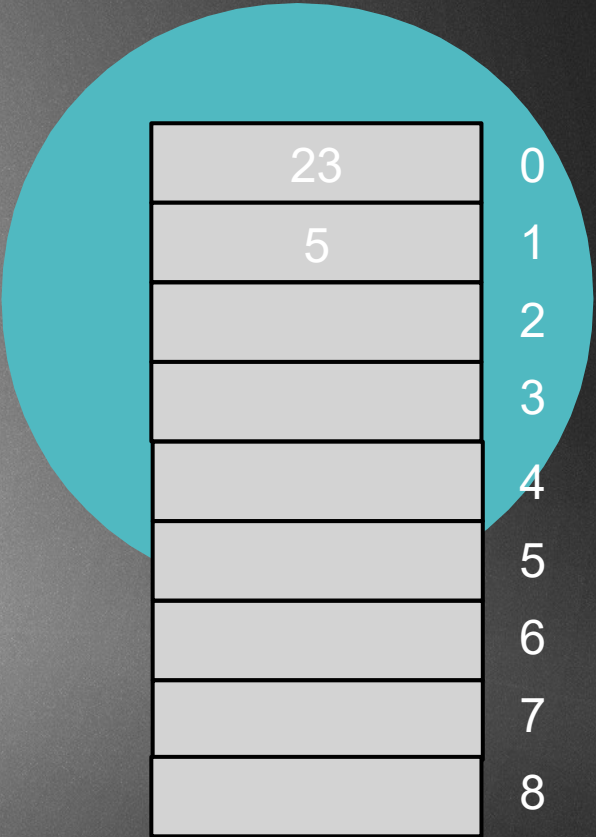
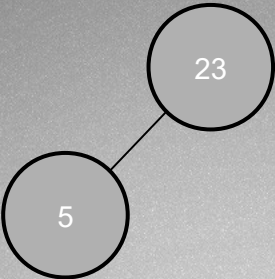


Building a heap

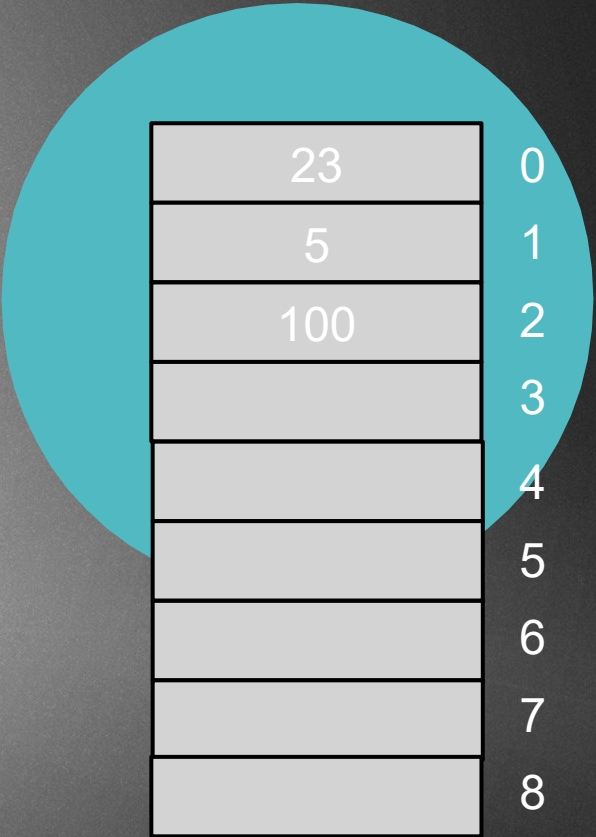
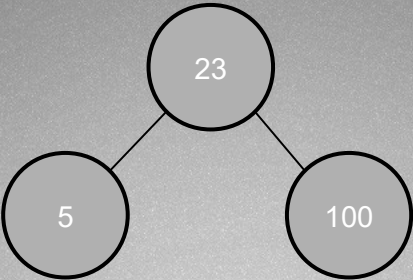


Building a heap

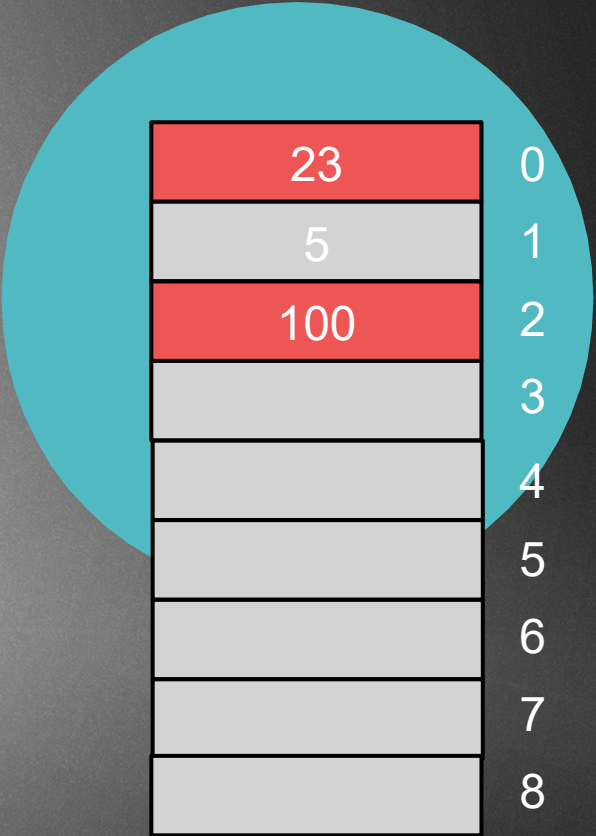
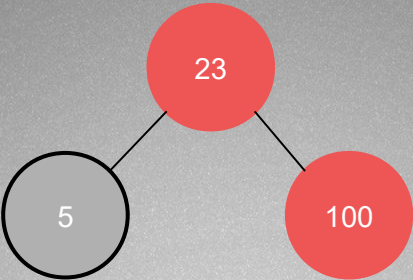
Insert: 100



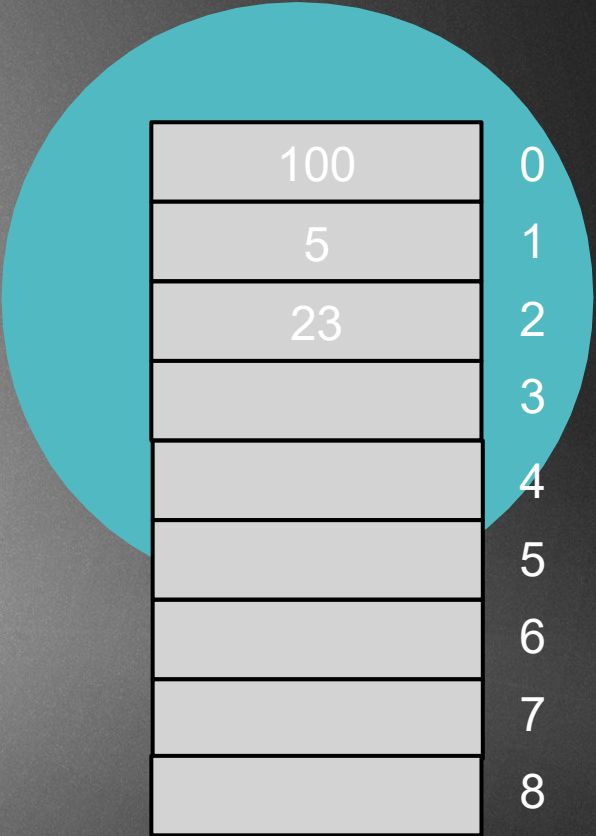
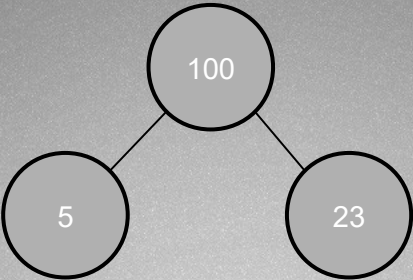
Building a heap



Building a heap

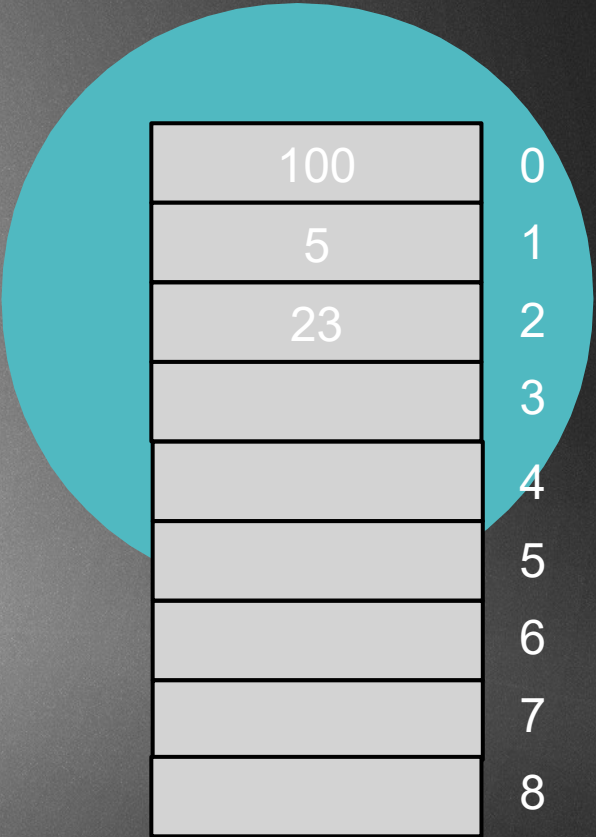
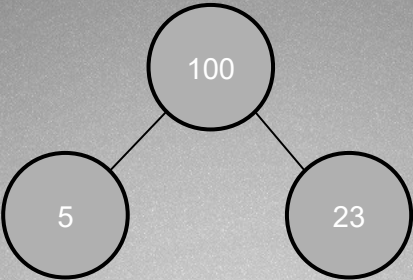


Building a heap

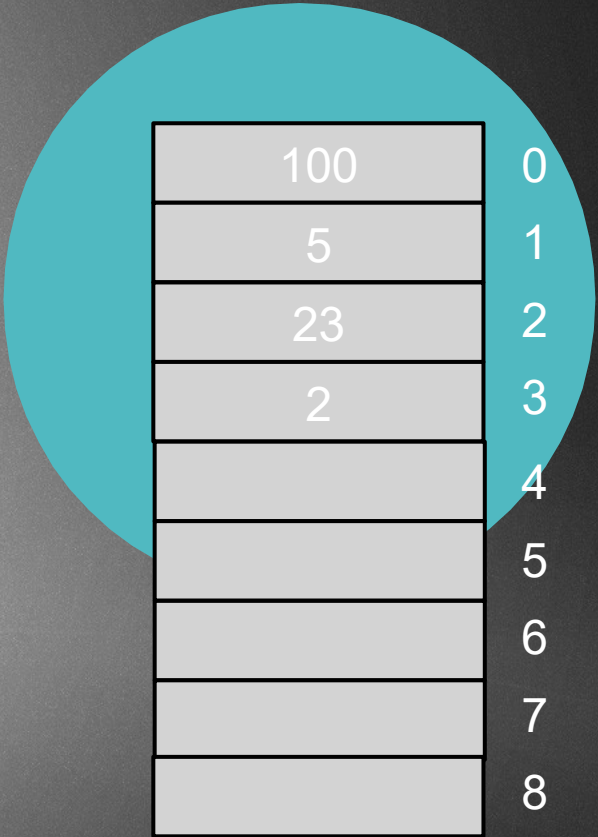
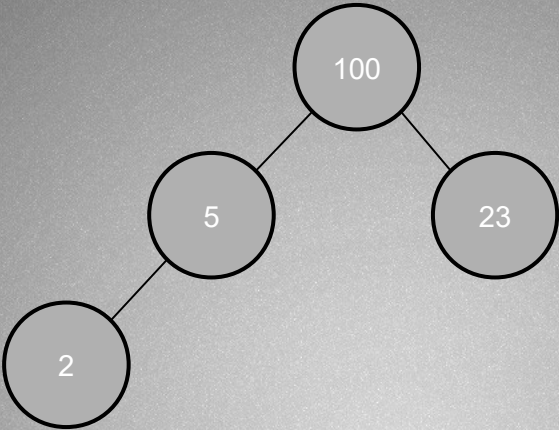


Building a heap

Insert: 2

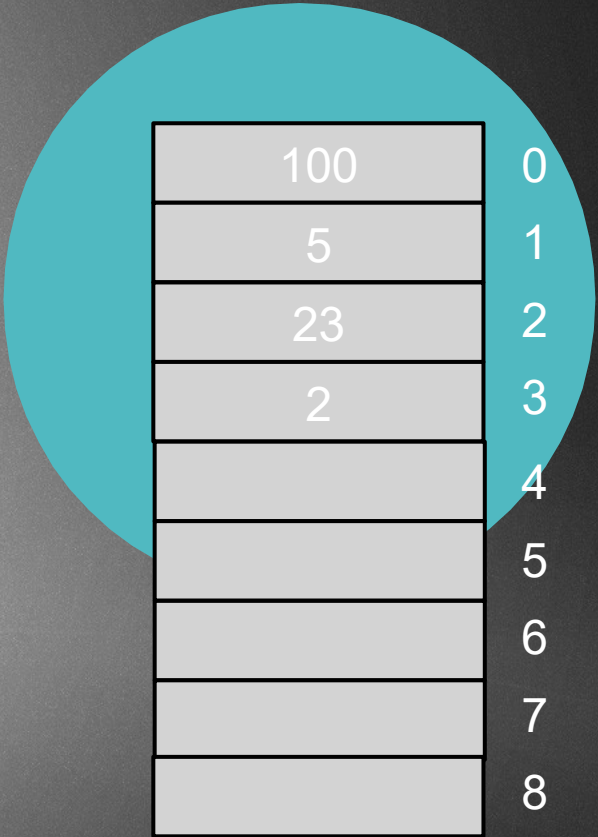
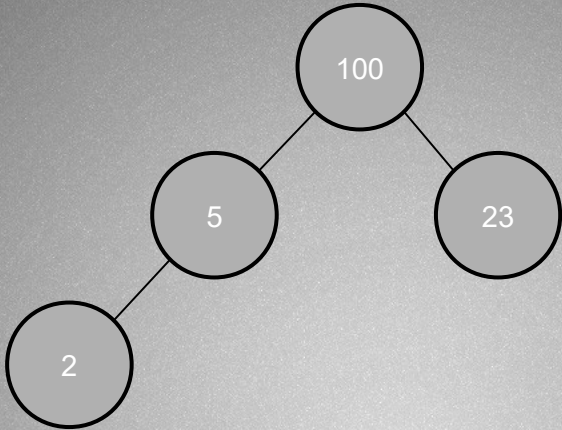


Building a heap

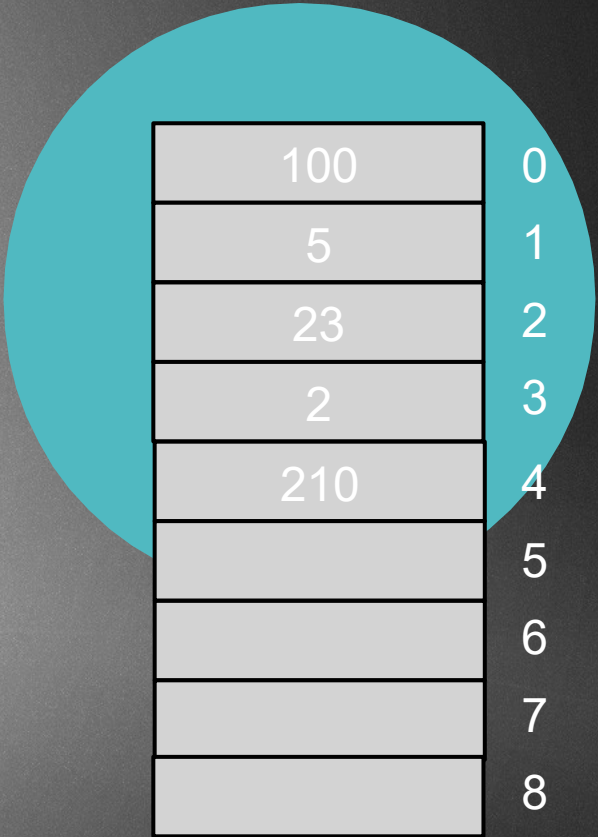
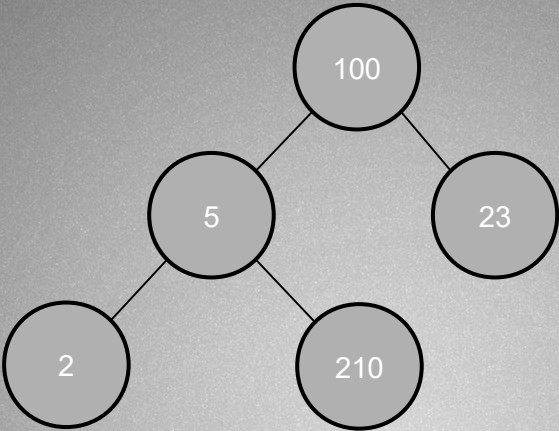


Building a heap

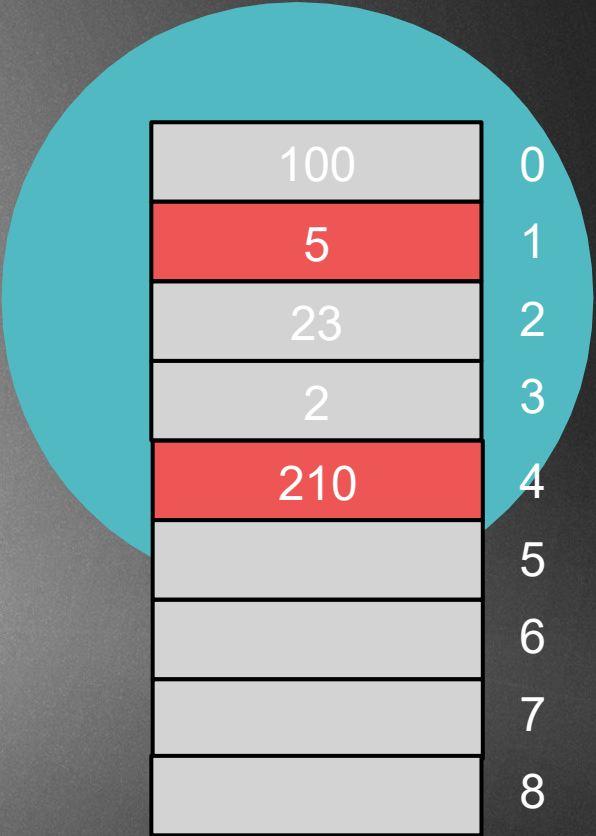
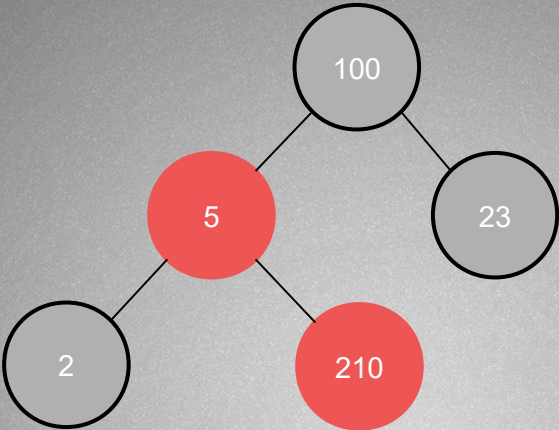
Insert: 210



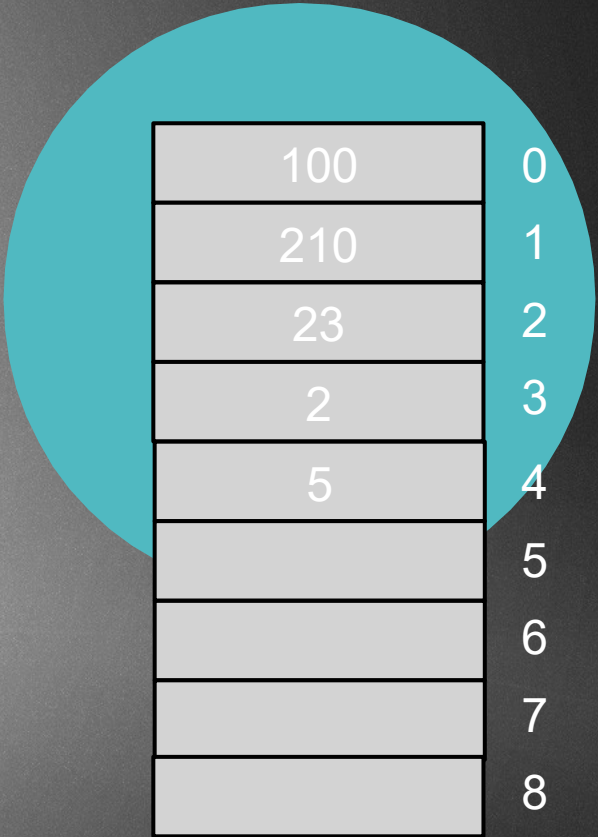
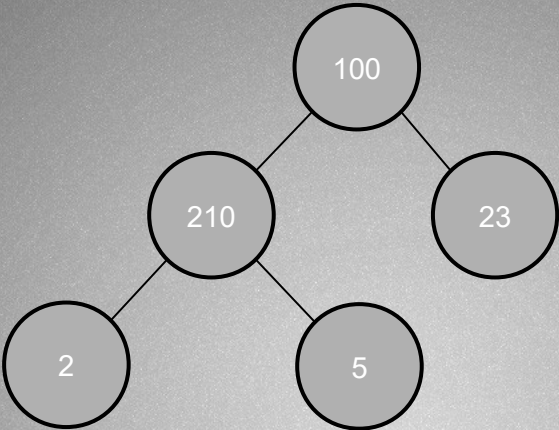
Building a heap



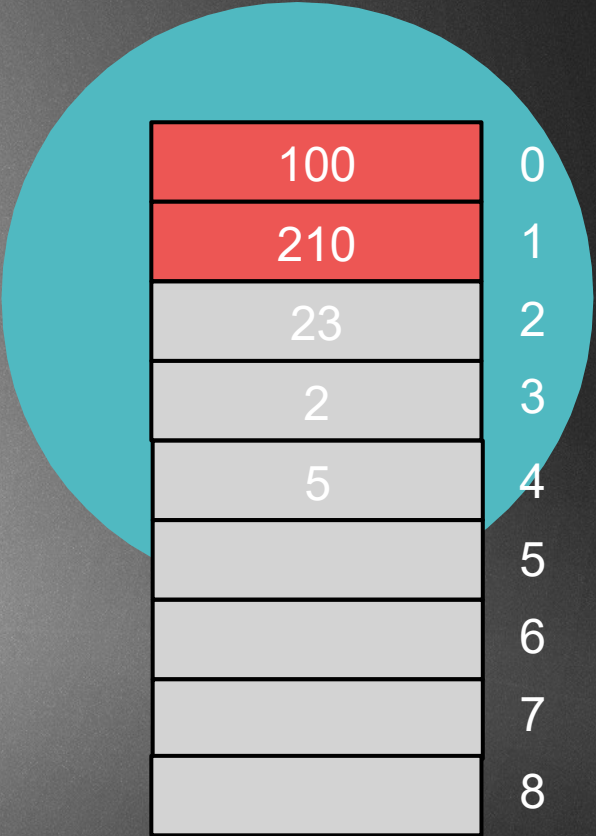
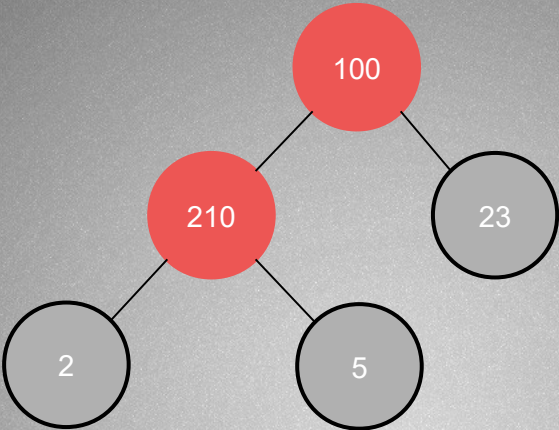
Building a heap



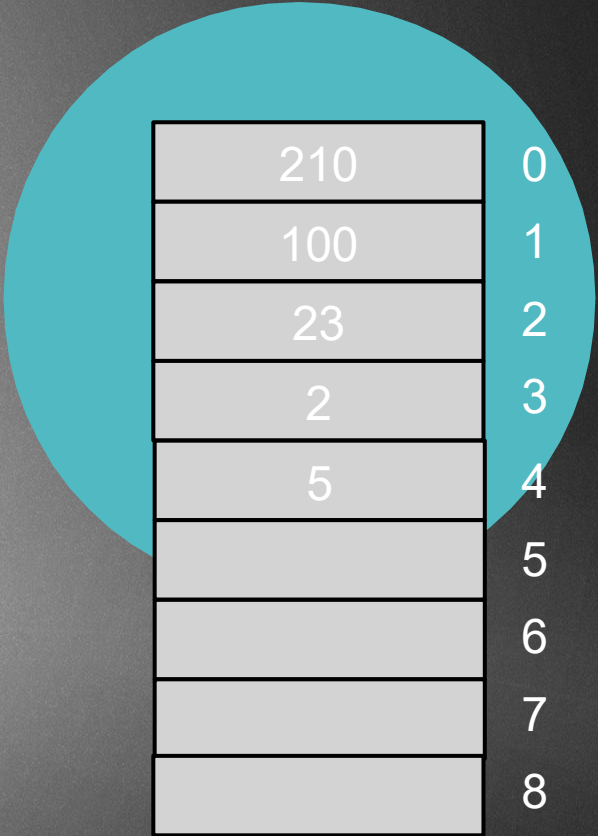
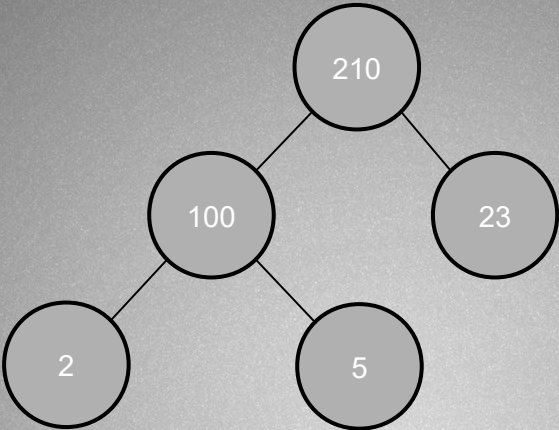
Building a heap



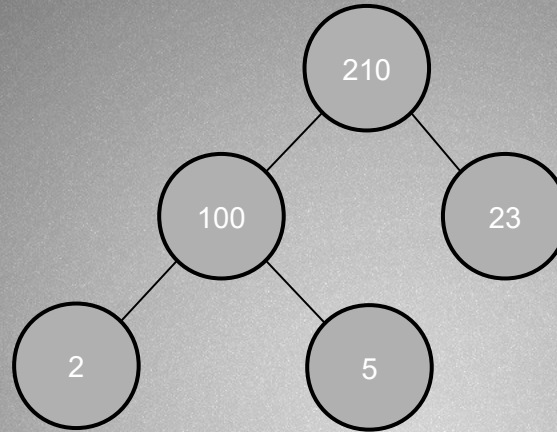
Building a heap



Building a heap



Building a heap



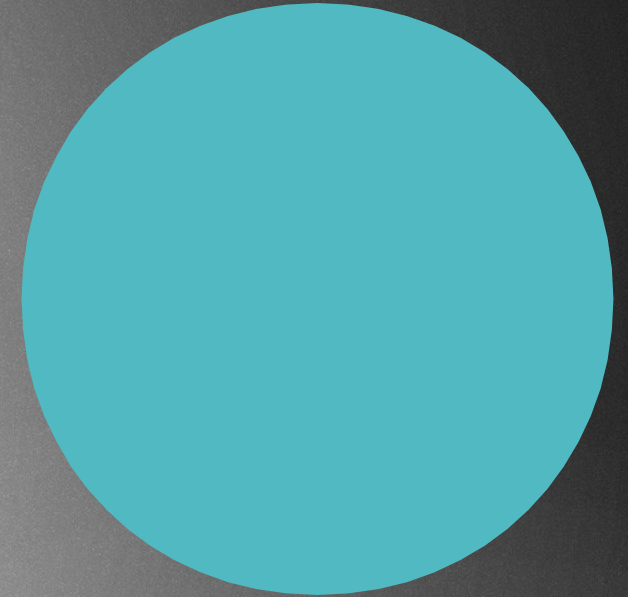
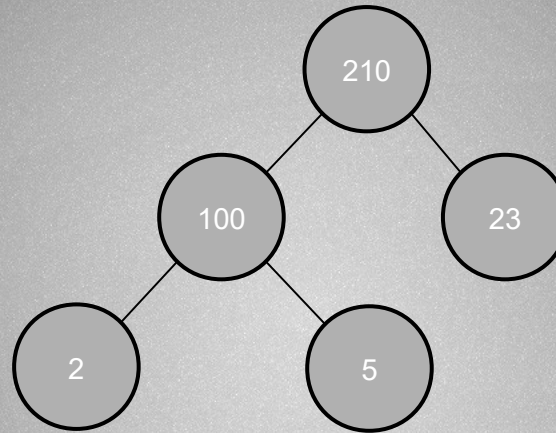
210	0
100	1
23	2
2	3
5	4
	5
	6
	7
	8

- it is an $O(N)$ process to construct a heap
- OK we have to reconstruct it if the heap properties are violated
but it takes $O(\log N)$ time
 $O(N) + O(\log N) = O(N)$
- inserting an item to the heap is just adding the data to the array with incremented index !!!

Remove operation

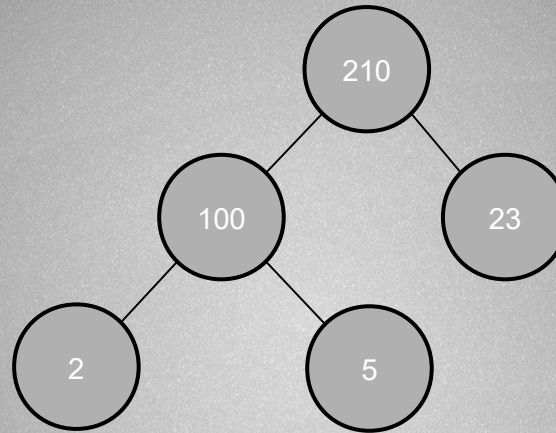


Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!

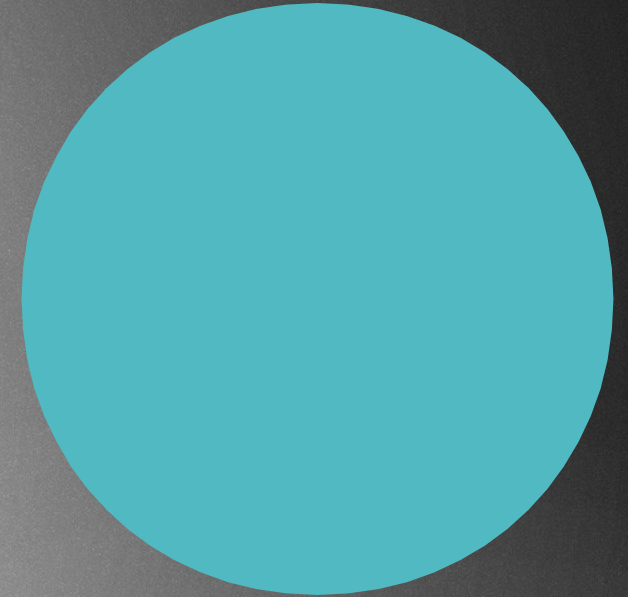
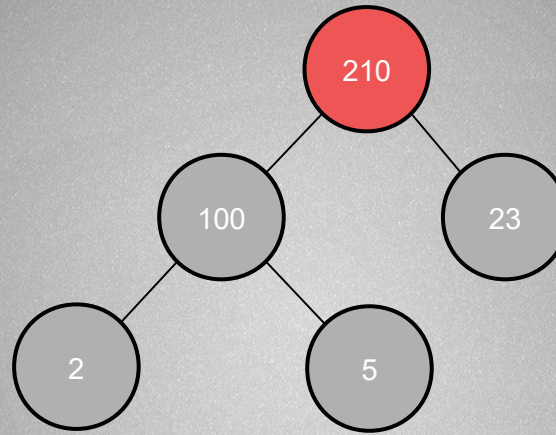


Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!

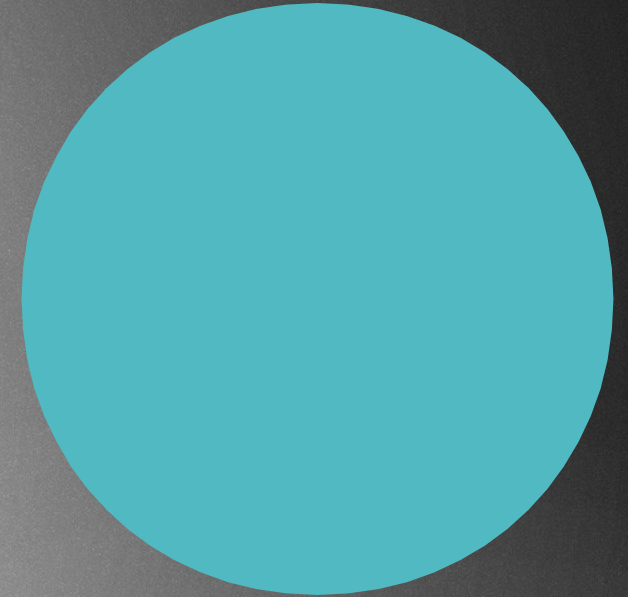
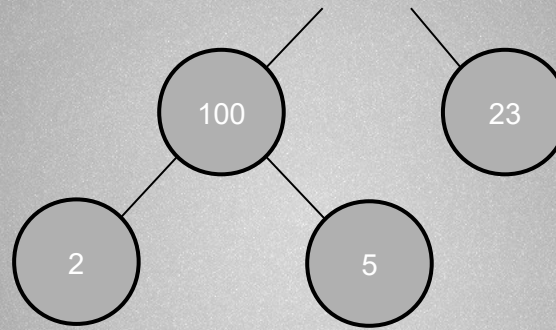
`deleteNode(210);`



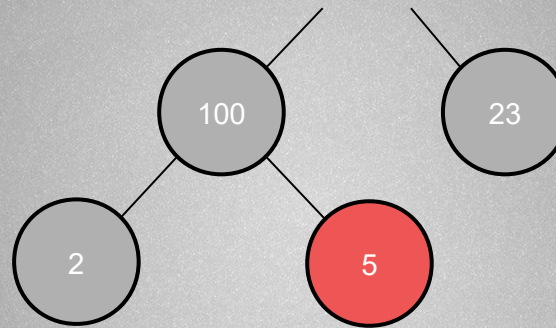
Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



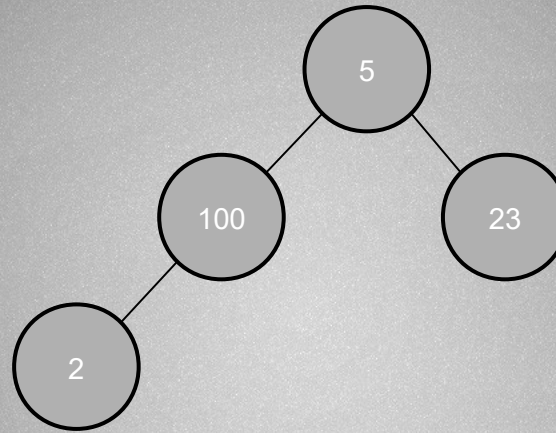
Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



OK, we have to find the last item in the heap:

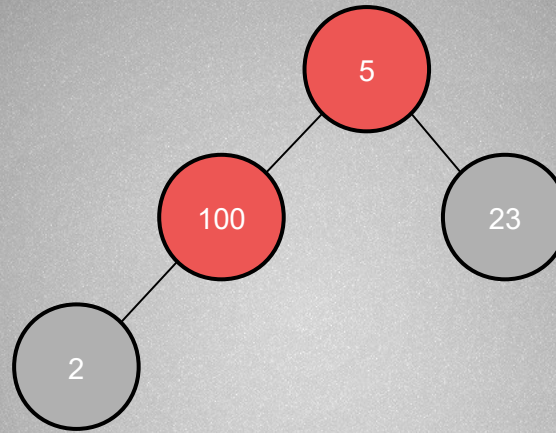
`heapArray[lastIndex]` → very easy to find it !!!

Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



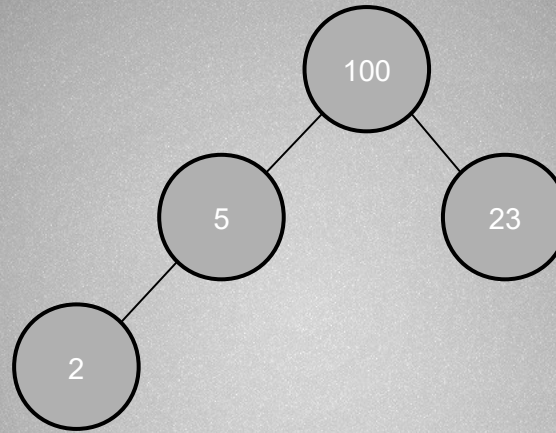
OK, we end up with a complete heap again, but we have to swap some items to make it valid !!!

Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



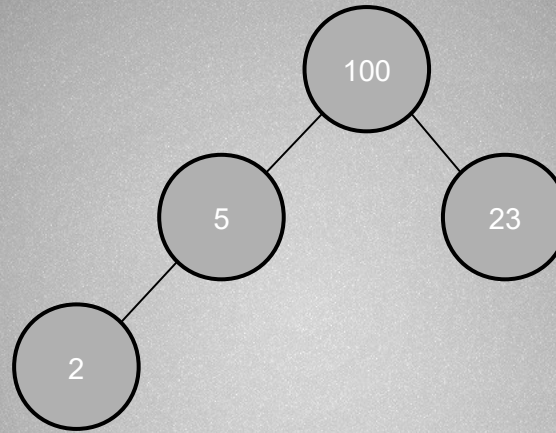
OK, we end up with a complete heap again, but we have to swap some items to make it valid !!!

Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



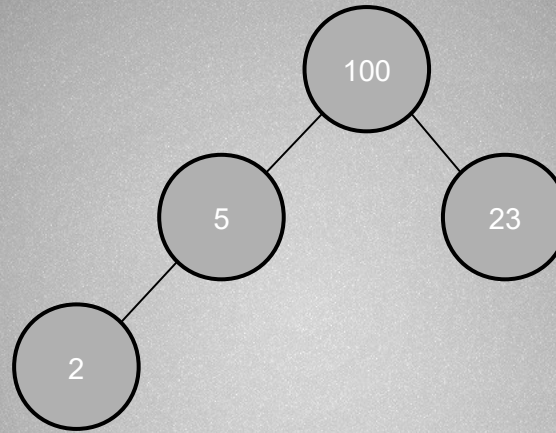
OK, we end up with a complete heap again, but we have to swap some items to make it valid !!!

Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



So: we have managed to get rid of the root node and to make some reconstructions in order to end up with a valid heap again !!!

Deleting an item: we just get rid of the item we want to delete. OK, but there will be a „hole” in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions !!!



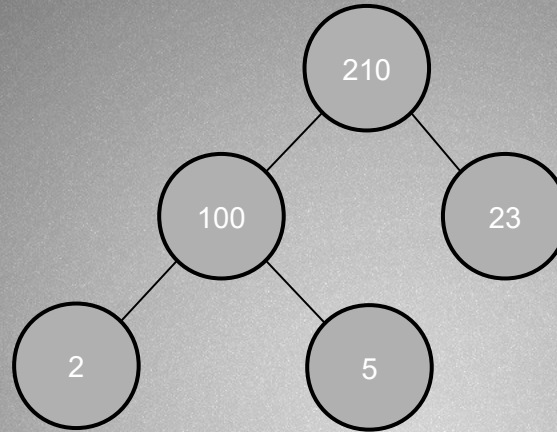
So: we have managed to get rid of the root node and to make some reconstructions in order to end up with a valid heap again !!!

Operation: deleting the root node $O(1)$ + reconstruction $O(\log N) = O(\log N)$!!!

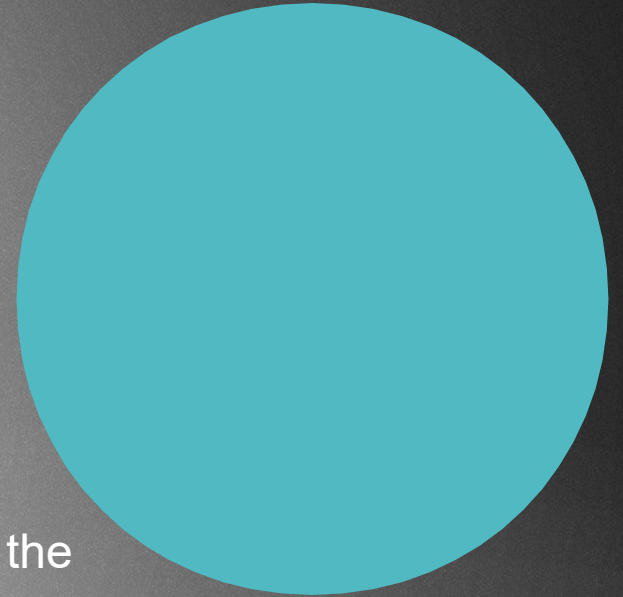
Heapsort

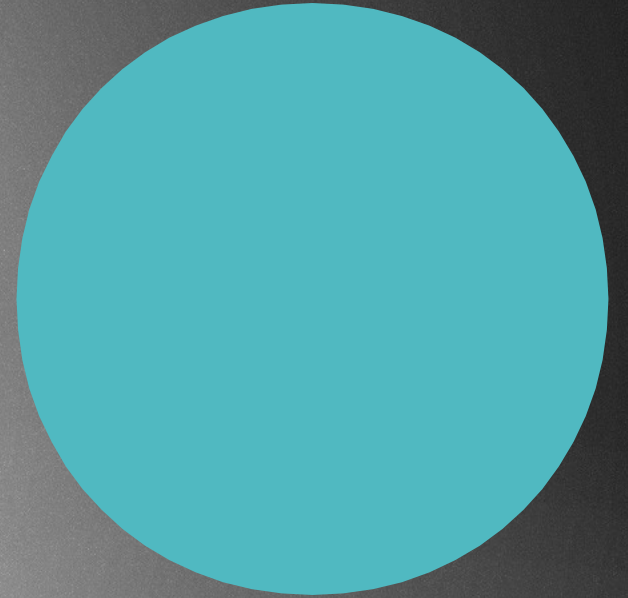
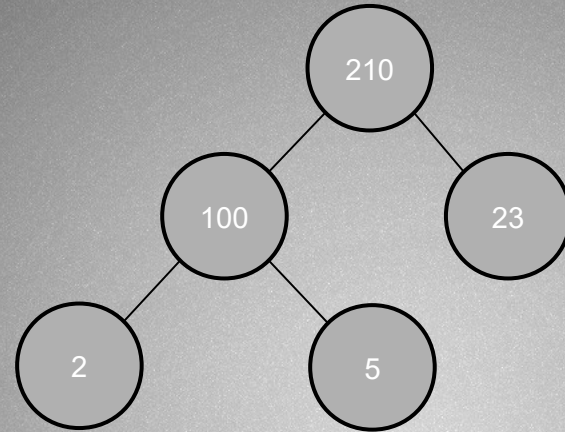
- ▶ Comparison-based sorting algorithm
- ▶ Use heap data structure rather than a linear-time search to find the maximum
- ▶ A bit slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case $O(n \log n)$ runtime
- ▶ It is an in-place algorithm, but it is not a stable sort
- ▶ DOES NOT NEED ADDITIONAL MEMORY
- ▶ Problem: first we have to construct the heap itself from the numbers we want to sort $\rightarrow O(N)$ time complexity !!!

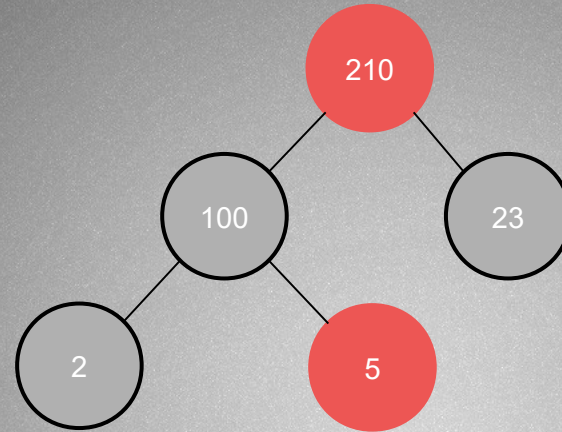
Keep swapping the root (because it is a heap, we know for certain that the root is the item with highest priority)
with the last element + maintain heap properties !!!



In this case, we are dealing with a max-heap, the root is the item with greatest value in the whole heap !!!

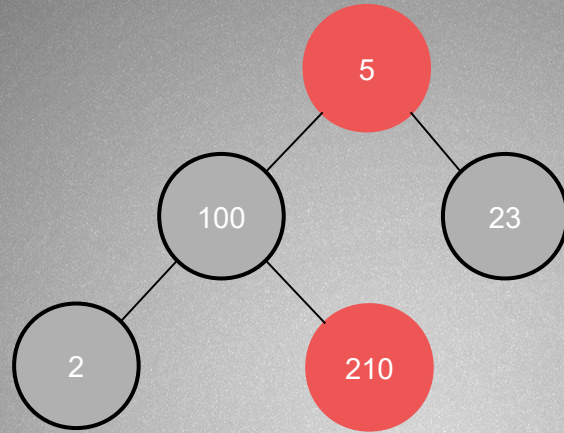




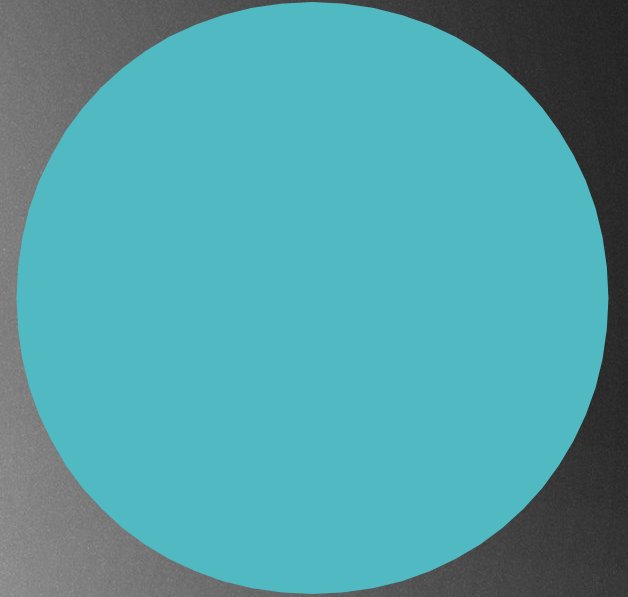


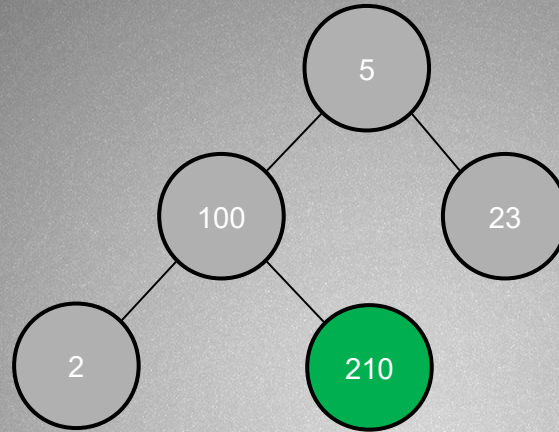
Sorted order:





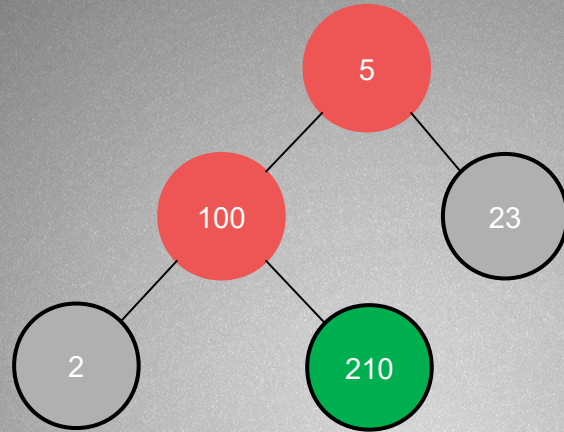
Sorted order:





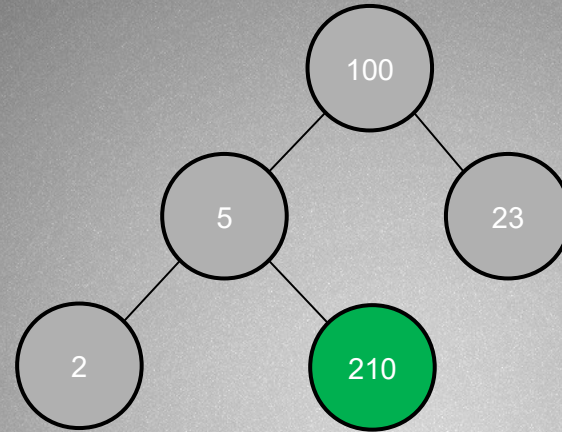
- After swapping with the root:
- we consider the last item to be sorted: no longer part of the tree !!!
 - check whether it is a valid heap or not

Sorted order: 210



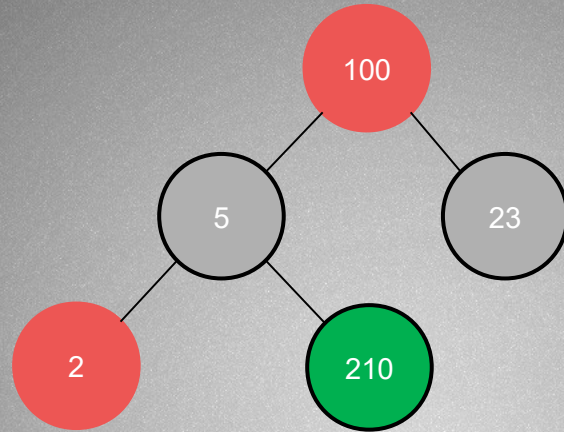
Sorted order: 210





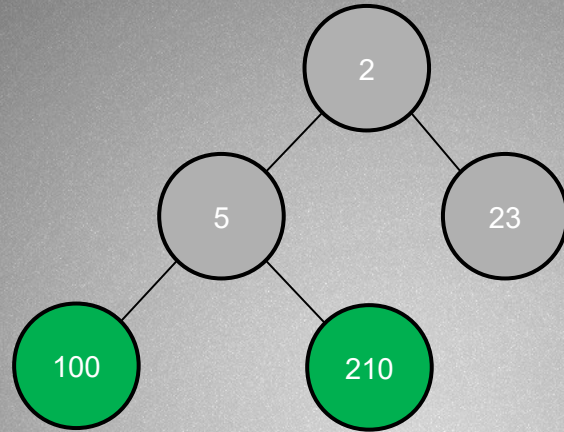
Sorted order: 210





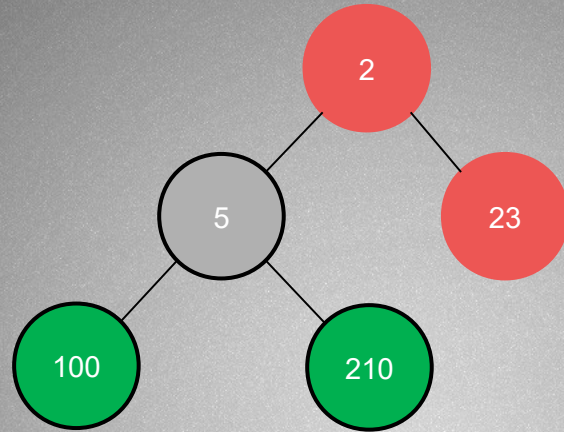
Sorted order: 210





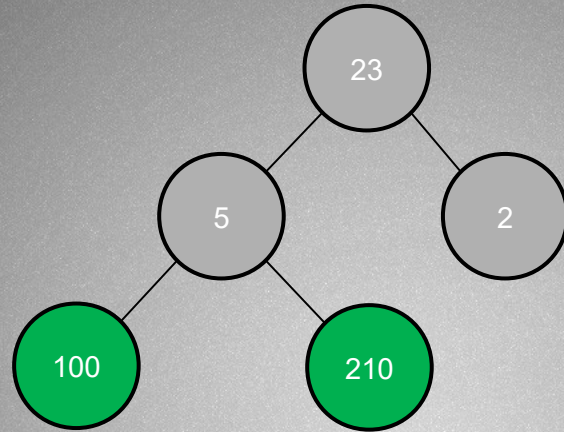
Sorted order: 210 , 100





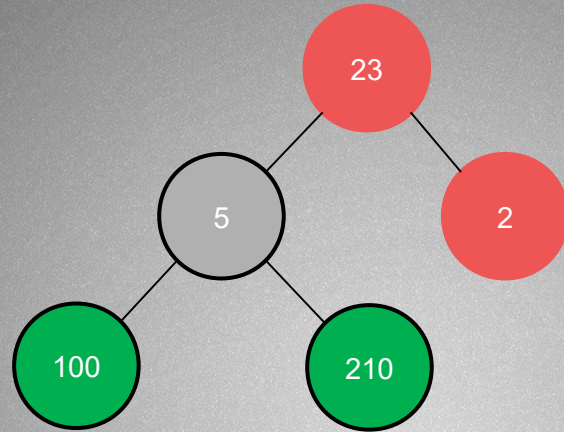
Sorted order: 210 , 100





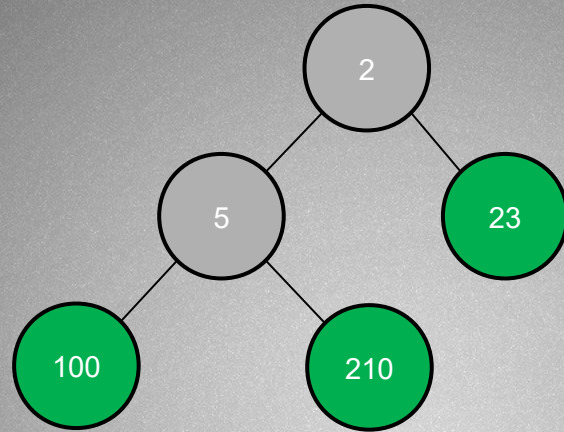
Sorted order: 210 , 100





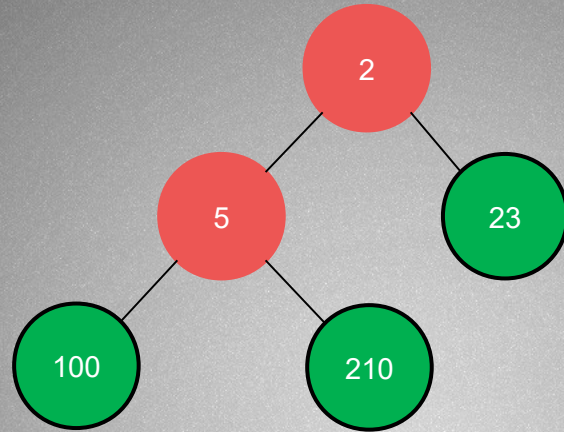
Sorted order: 210 , 100





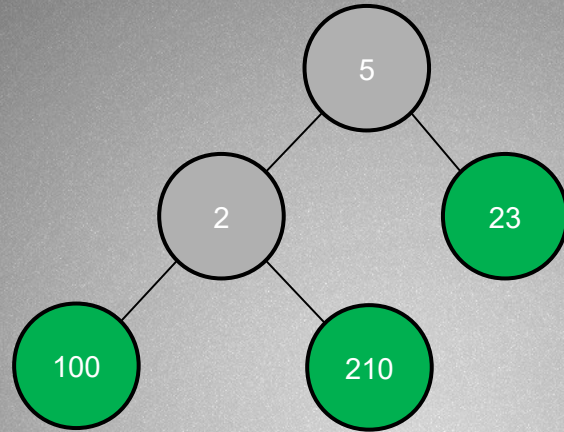
Sorted order: 210 , 100 , 23





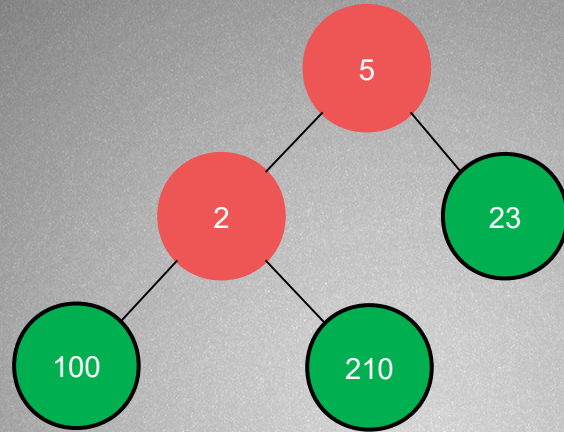
Sorted order: 210 , 100 , 23





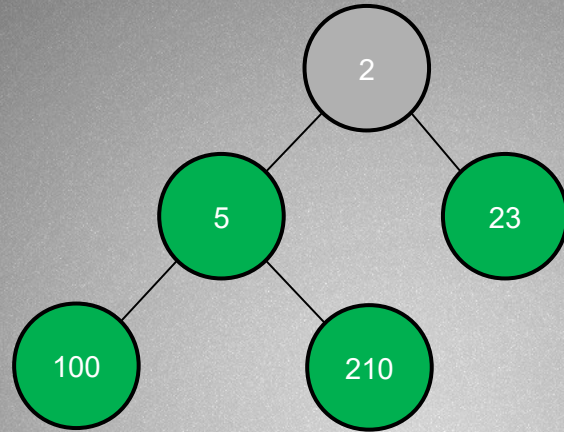
Sorted order: 210 , 100 , 23





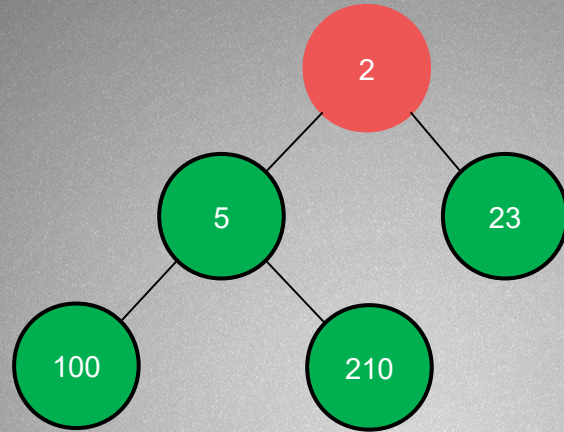
Sorted order: 210 , 100 , 23





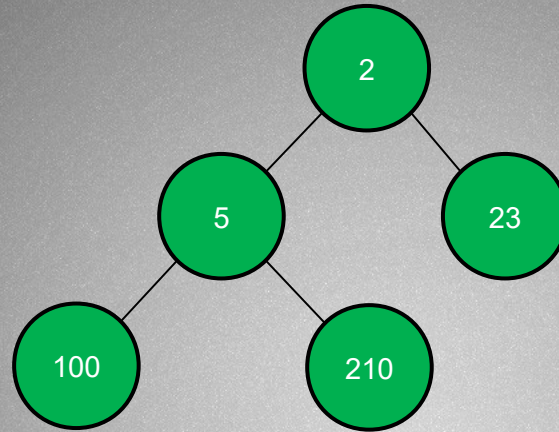
Sorted order: 210 , 100 , 23 , 5





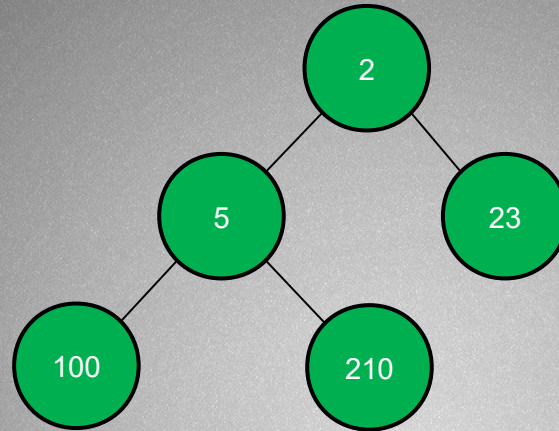
Sorted order: 210 , 100 , 23 , 5





Sorted order: 210 , 100 , 23 , 5 , 2

We have managed to sort the elements !!!



Sorted order: 210 , 100 , 23 , 5 , 2

We have managed to sort the elements !!!

Running time: we have to consider N items + have to make some swappings if necessary

$O(N \cdot \log N)$

Running time

Memory complexity: we have N items we want to store in the heap
We have to allocate memory for an array with size N
 $O(N)$ memory complexity



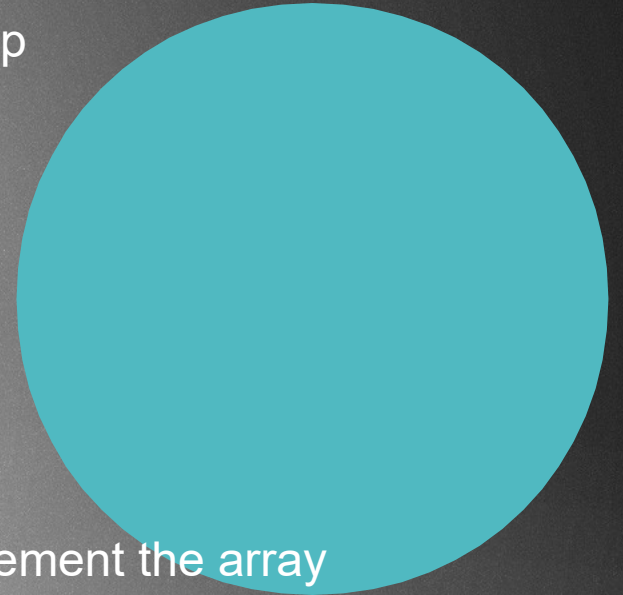
Running time

Memory complexity: we have N items we want to store in the heap
We have to allocate memory for an array with size N
 $O(N)$ memory complexity

Find the minimum / maximum: $O(1)$ very fast
Because in a heap the highest priority item is at the root node, it is easy
`heapArray[0]` will be the item we are looking for

Insert new item: we can insert at the next available place, so increment the array index and insert it $\rightarrow O(1)$ fast
BUT we have to make sure the heap properties are met ...
it may take $O(\log N)$ time

$O(\log_2 N)$ Why? Because a node has at most $\log_2 N$ parents so at most $\log_2 N$ swaps are needed



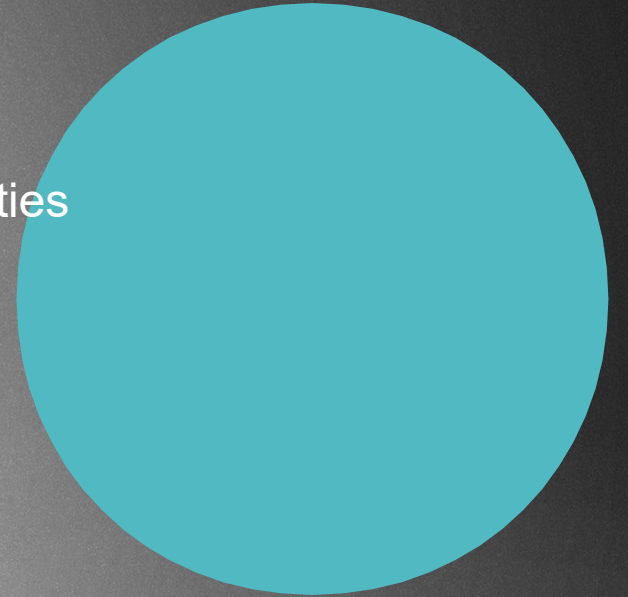
Running time

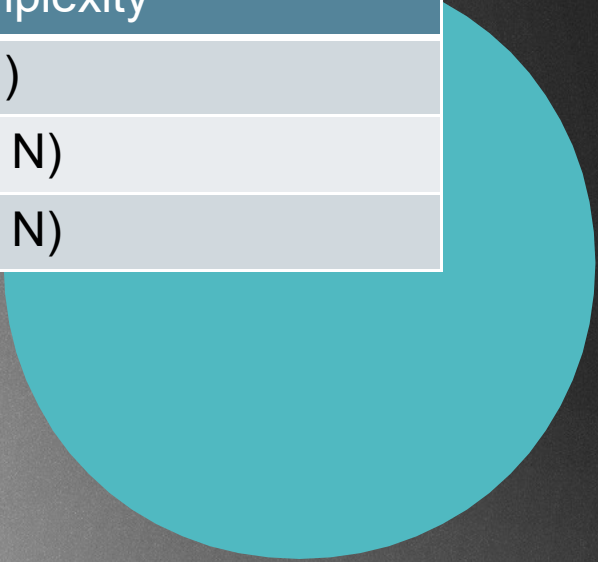

Remove item: we usually remove the root node

Removing it is quite fast: just delete it in $O(1)$ time

BUT we have to make sure we met the heap properties

$O(\log N)$ time to reconstruct the heap !!!





Operation	Time Complexity
Find minimum/maximum	$O(1)$
Remove min / max	$O(\log N)$
Insert	$O(\log N)$

Binomial heap

- ▶ Similar to a binary heap but also supports quick merging of two heaps
- ▶ It is important as an implementation of the mergeable heap abstract data type (meldable heap)
- ▶ Which is a priority queue basically + supporting merge operation
- ▶ A binomial heap is implemented as a collection of tree
- ▶ Insertion $O(\log n)$ time complexity can be reduced to $O(1)$ constant time complexity with the help of binomial heaps

Fibonacci heap

- ▶ Faster than the classic binary heap
- ▶ Dijkstra's shortest path algorithm and Prim's spanning tree algorithm run faster if they rely on Fibonacci heap instead of binary heaps
- ▶ BUT very hard to implement efficiently so usually does not worth the effort
- ▶ Unlike binary heaps, it can have several children: number of children are usually kept low
- ▶ We can achieve $O(1)$ insert operation instead of $O(\log n)$!!!
- ▶ Every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2} , where F_k is the k -th Fibonacci number

Time complexities

	Binary	Binomial	Fibonacci
Find min	$O(1)$	$O(1)$	$O(1)$
Delete min	$O(\log n)$	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(1)$	$O(1)$
Decrease key	$O(\log n)$	$O(\log n)$	$O(1)$
merge	-	$O(\log n)$	$O(1)$