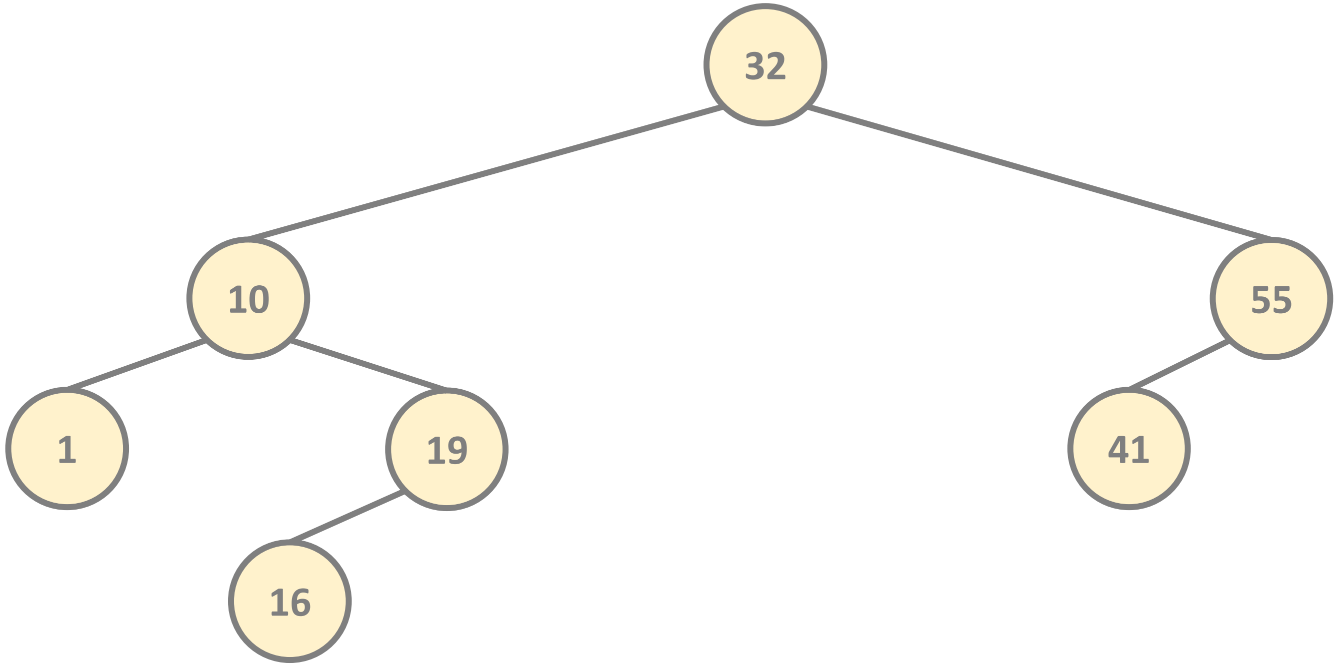
Priority Queues

(Algorithms and Data Structures)

Priority Queues

- it is an **abstract data type** such as queue
- every item has an additional property – the so-called **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 **heap data structures** but it can be implemented with self balancing trees as well
- it is very similar to queues with some modification: **the highest priority** element is retrieved first

Priority Queues



Priority Queues

Sometimes we do not specify the **priority** for example when implementing heap data structures

→ the value of an integer (or float) can be interpreted as a priority

→ so we can omit the priority when inserting new integers or floats

For example: 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

Priority Queues

The concept of priority queues naturally suggest a **sorting algorithm** where we have to insert all the elements to be sorted into a **priority queue**

→ remove the items one by one from the priority queue and it yields the sorted order

→ if we take out a given item then it will be the one with the highest priority value

→ this is exactly how **heapsort** works

Heap Data Structure

(Algorithms and Data Structures)

Heaps

- heaps are basically **binary trees**
- two main binary heap types: **min heap** and **max heap**
- it was first constructed back in **1964** by **J. W. J. Williams**

1.) 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 (max value) is in the root node.

2.) MIN HEAP

In a **min heap** the keys of parent nodes are less than or equal to those of the children and the lowest key (min item) is in the root node

Heaps

- heaps are basically **binary trees**
- two main binary heap types: **min heap** and **max heap**
- it is **complete** so it cannot be imbalanced
- we insert every new item to the next available place
- **APPLICATIONS**: Dijkstra's algorithm, Prim's algorithm

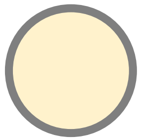
*„Bad programmers worry about the code. Good programmers worry about **data structures** and their relationships"*

Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete

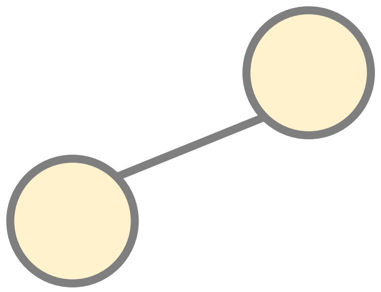
Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



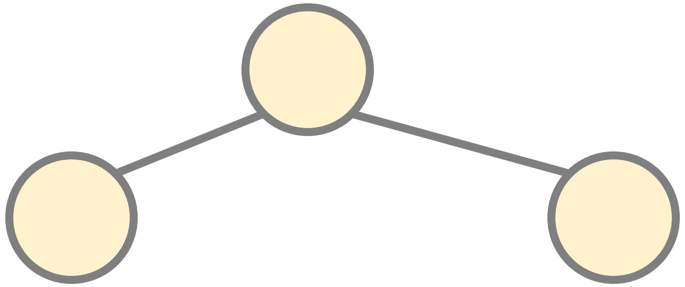
Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



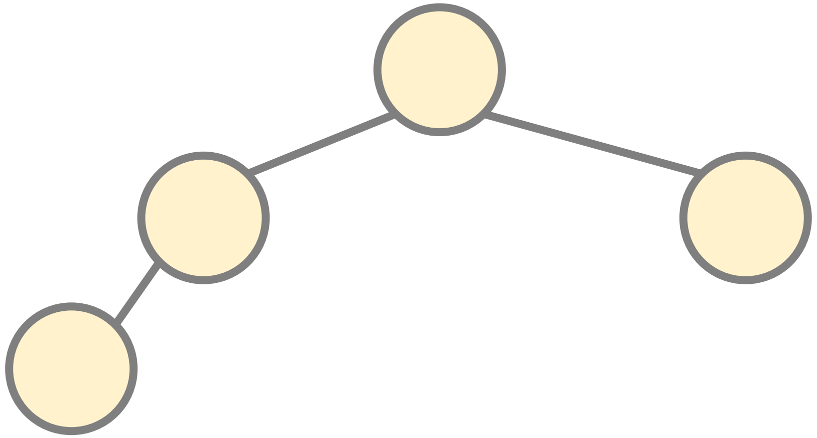
Heap Properties

1.) **COMPLETENESS:** we construct the **heap** from left to right across each row – of course the last row may not be fully complete



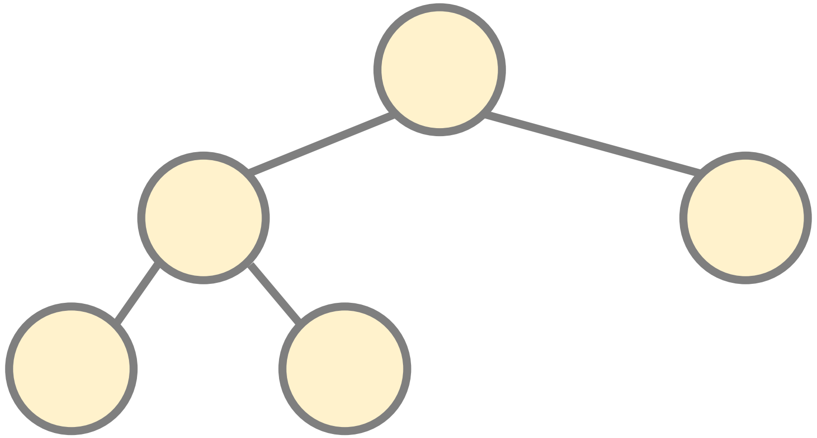
Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



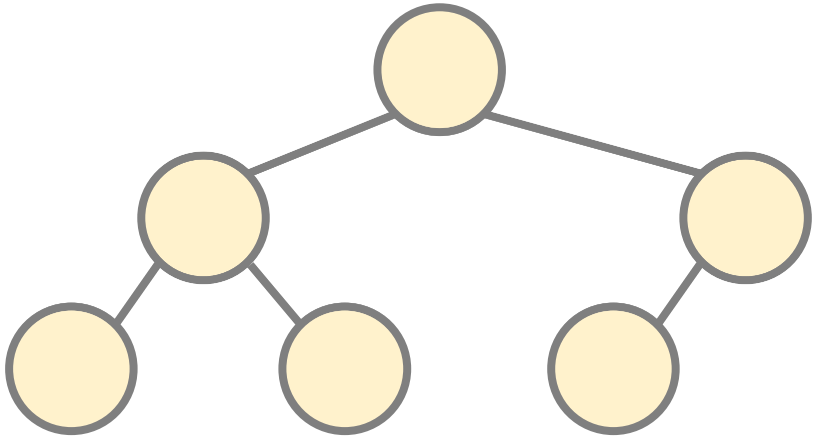
Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



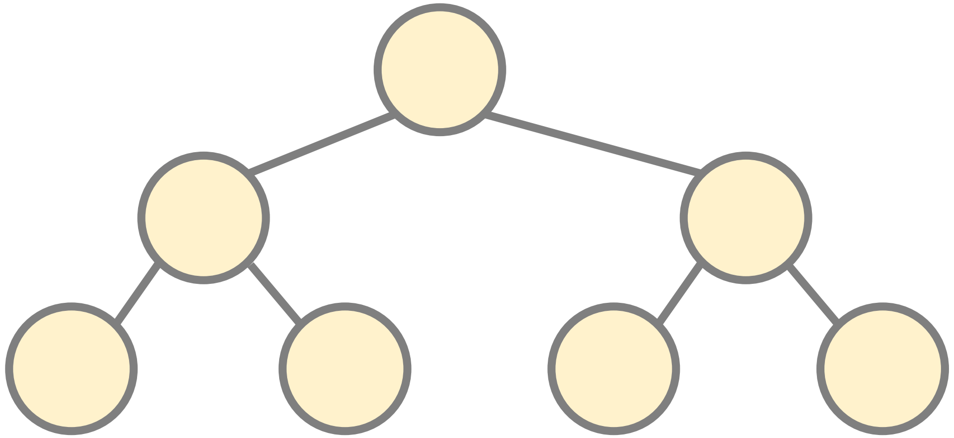
Heap Properties

1.) **COMPLETENESS:** we construct the **heap** from left to right across each row – of course the last row may not be fully complete



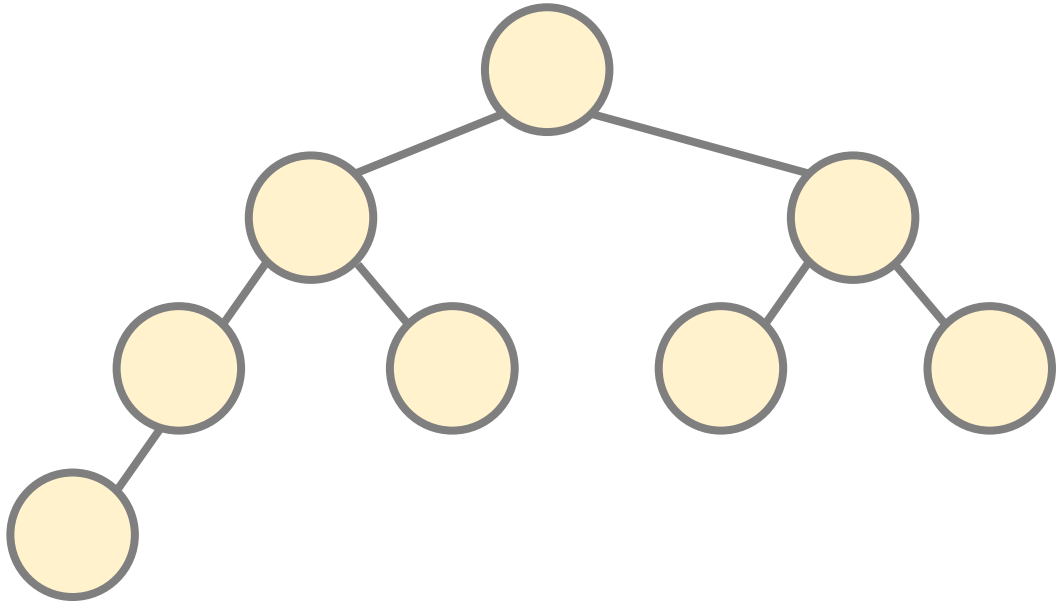
Heap Properties

1.) **COMPLETENESS:** we construct the **heap** from left to right across each row – of course the last row may not be fully complete



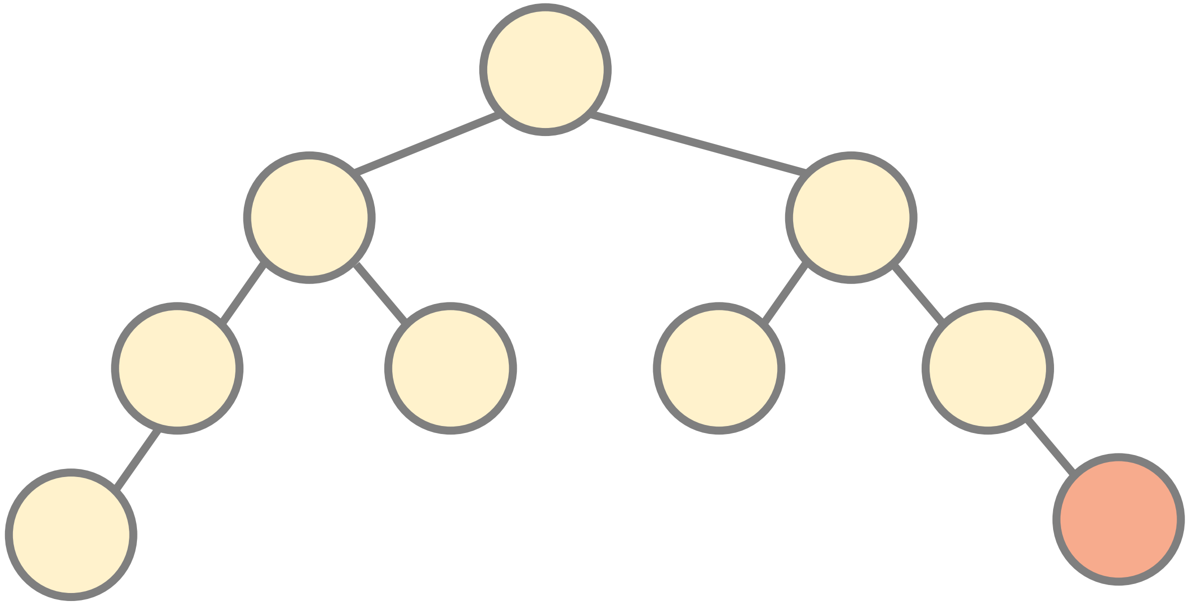
Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



Heap Properties

1.) COMPLETENESS: we construct the **heap** from left to right across each row – of course the last row may not be fully complete



*it is not a valid heap because he **completeness property** is violated*

Heap Properties

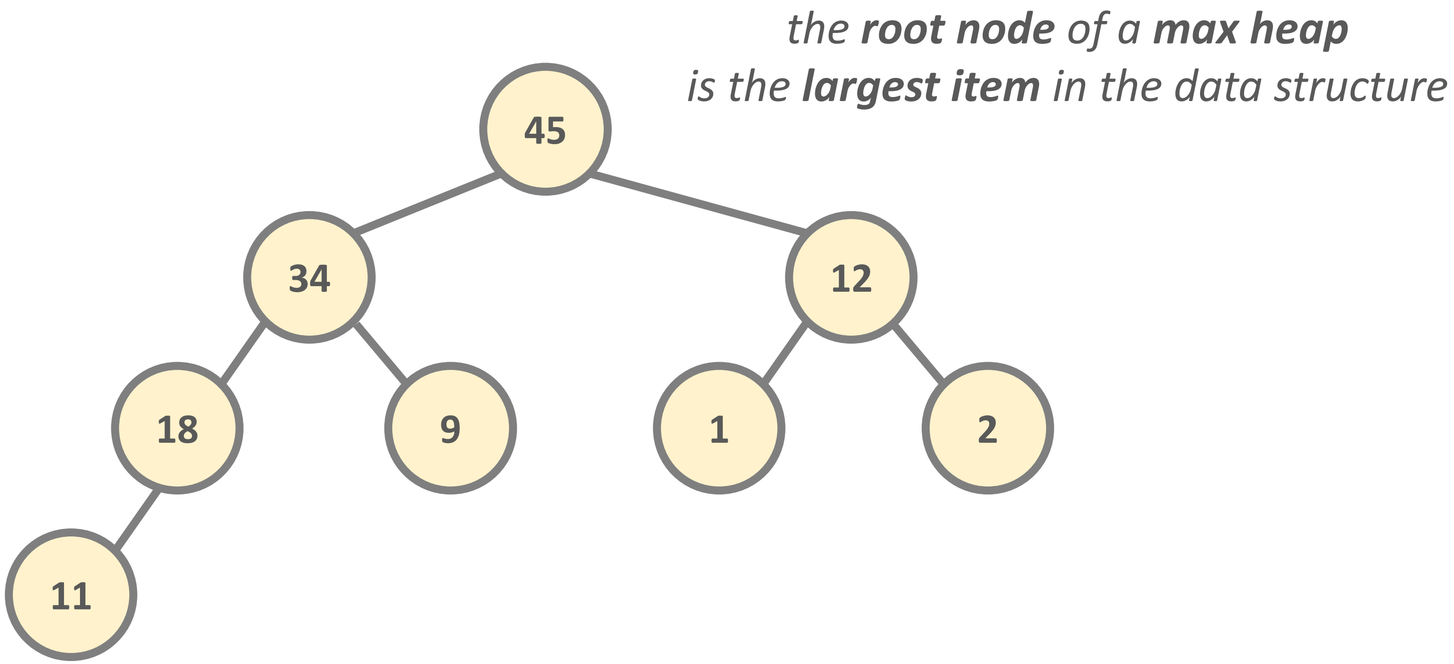
2.) HEAP PROPERTY: every node can have **2** children so heaps are almost-complete binary trees.

→ **min heap**: the parent node is always **smaller** than the child nodes (left and right nodes)

→ **max heap**: the parent node is always **greater** than the child nodes (left and right nodes)

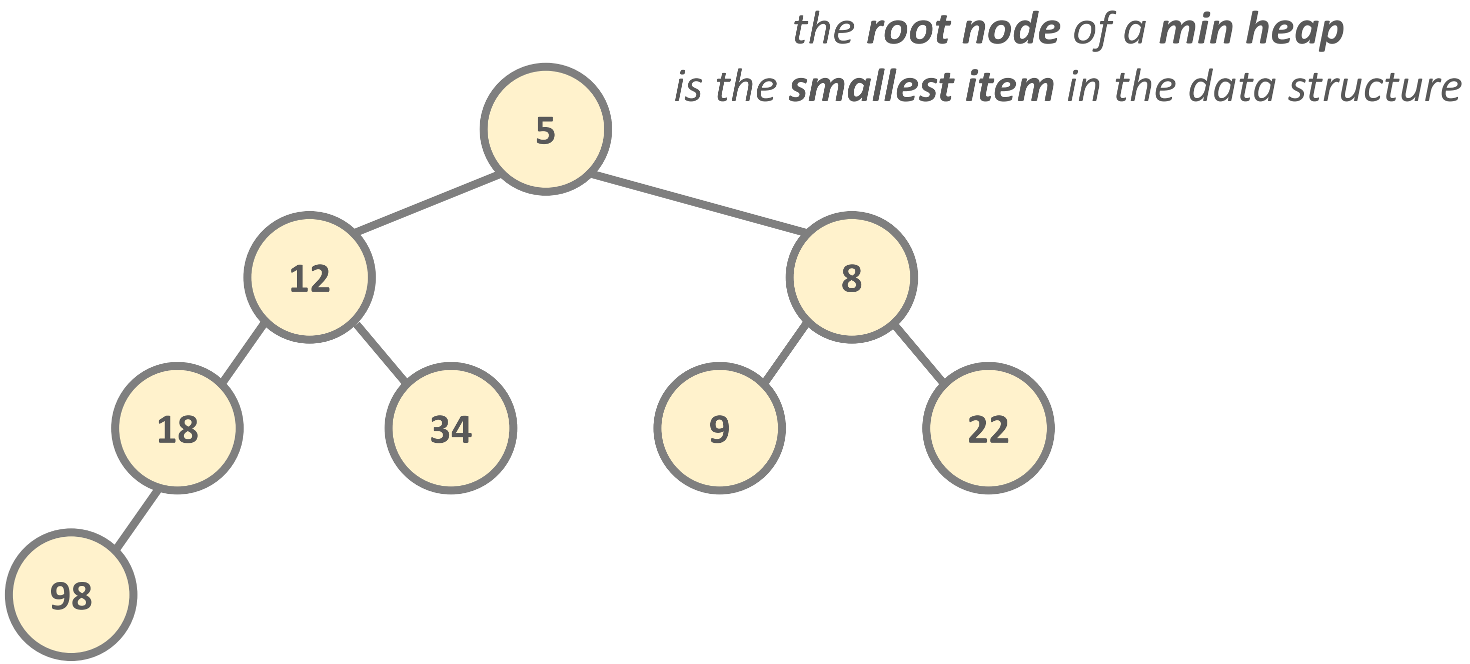
Heap Properties

2.) HEAP PROPERTY: every node can have **2** children so heaps are almost-complete binary trees.



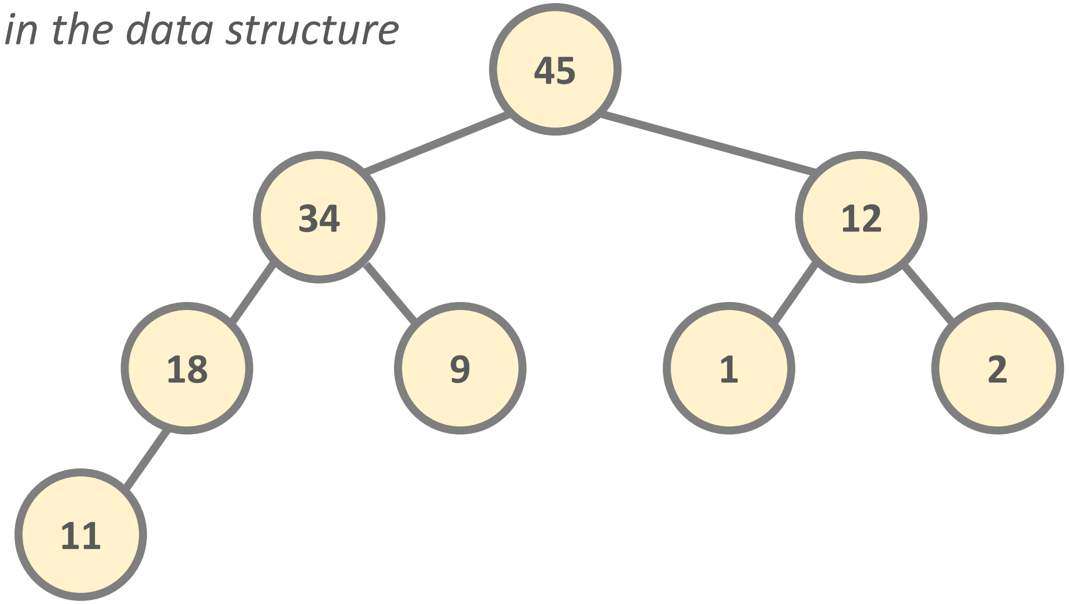
Heap Properties

2.) HEAP PROPERTY: every node can have **2** children so heaps are almost-complete binary trees.



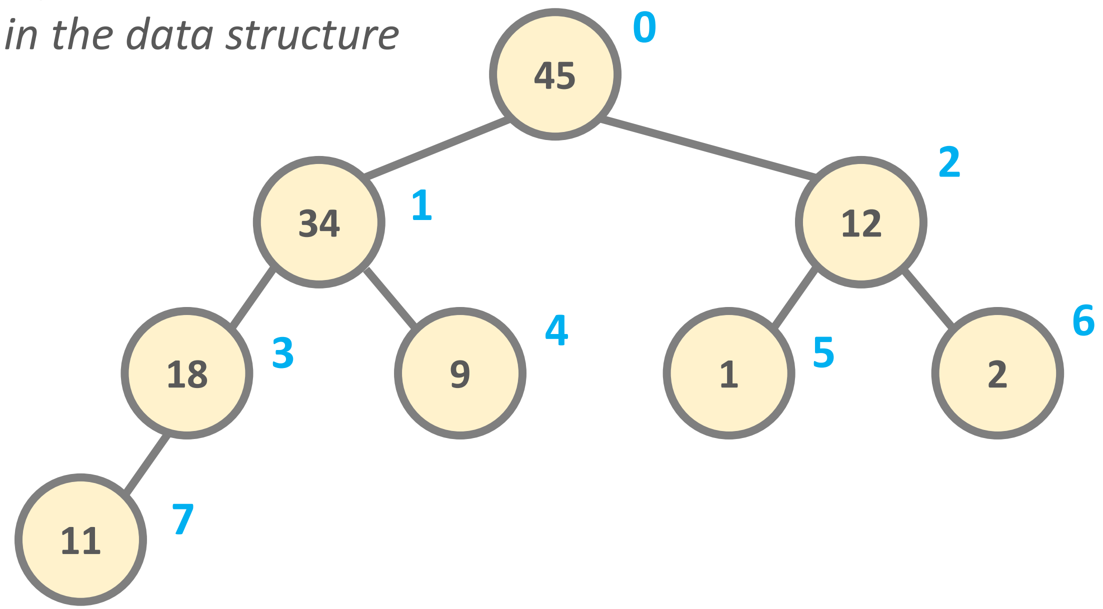
Representing Heaps

*the **root node** of a **max heap***
*is the **largest item** in the data structure*



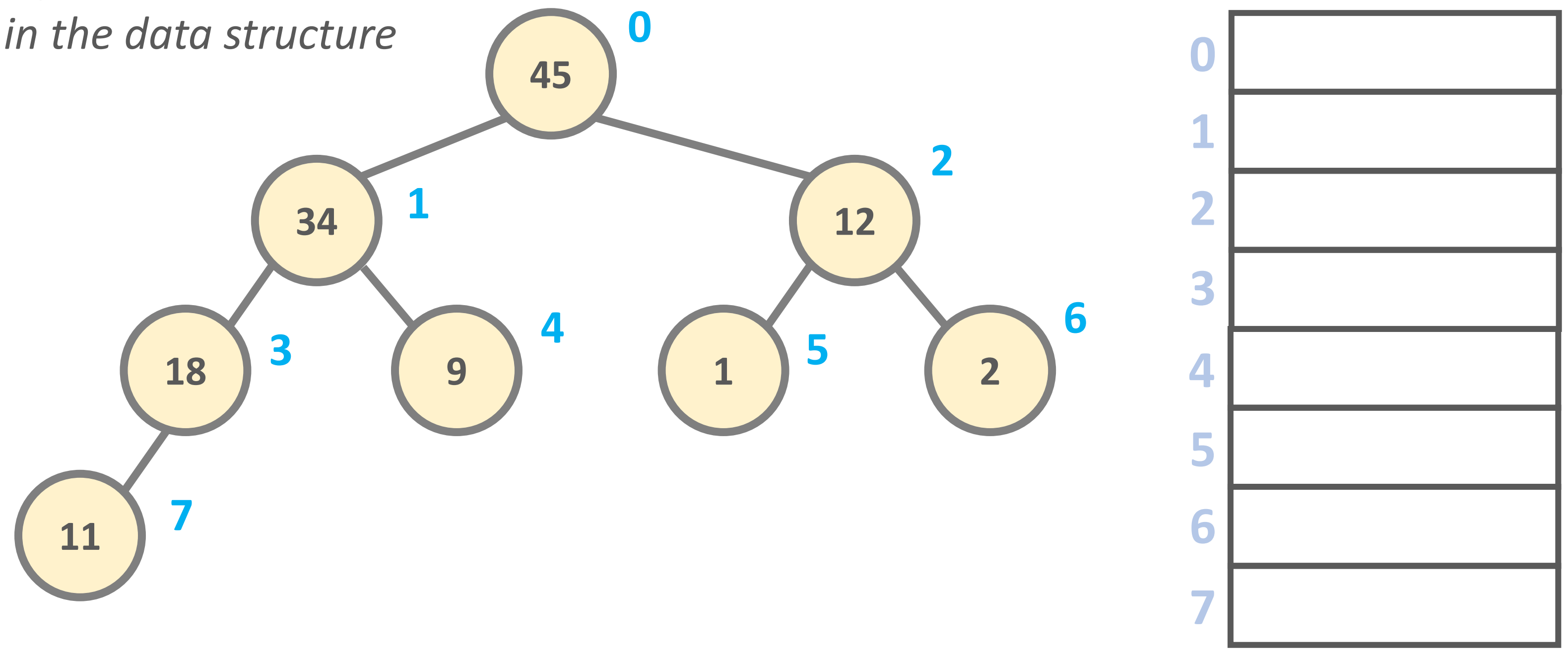
Representing Heaps

*the **root node** of a **max heap***
*is the **largest item** in the data structure*



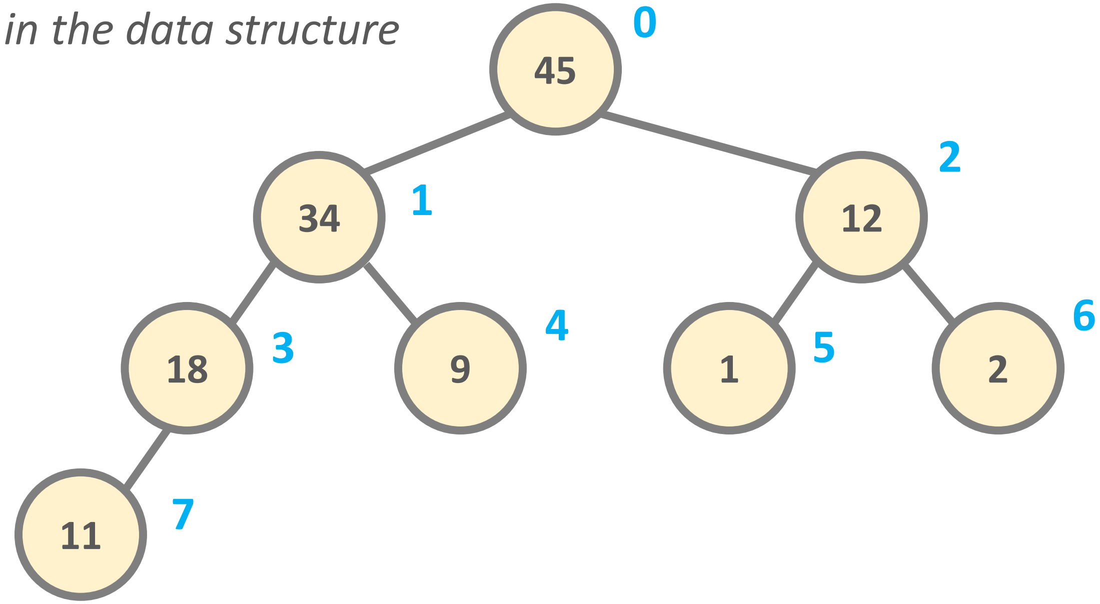
Representing Heaps

*the **root node** of a **max heap***
*is the **largest item** in the data structure*



Representing Heaps

*the **root node** of a **max heap***
*is the **largest item** in the data structure*

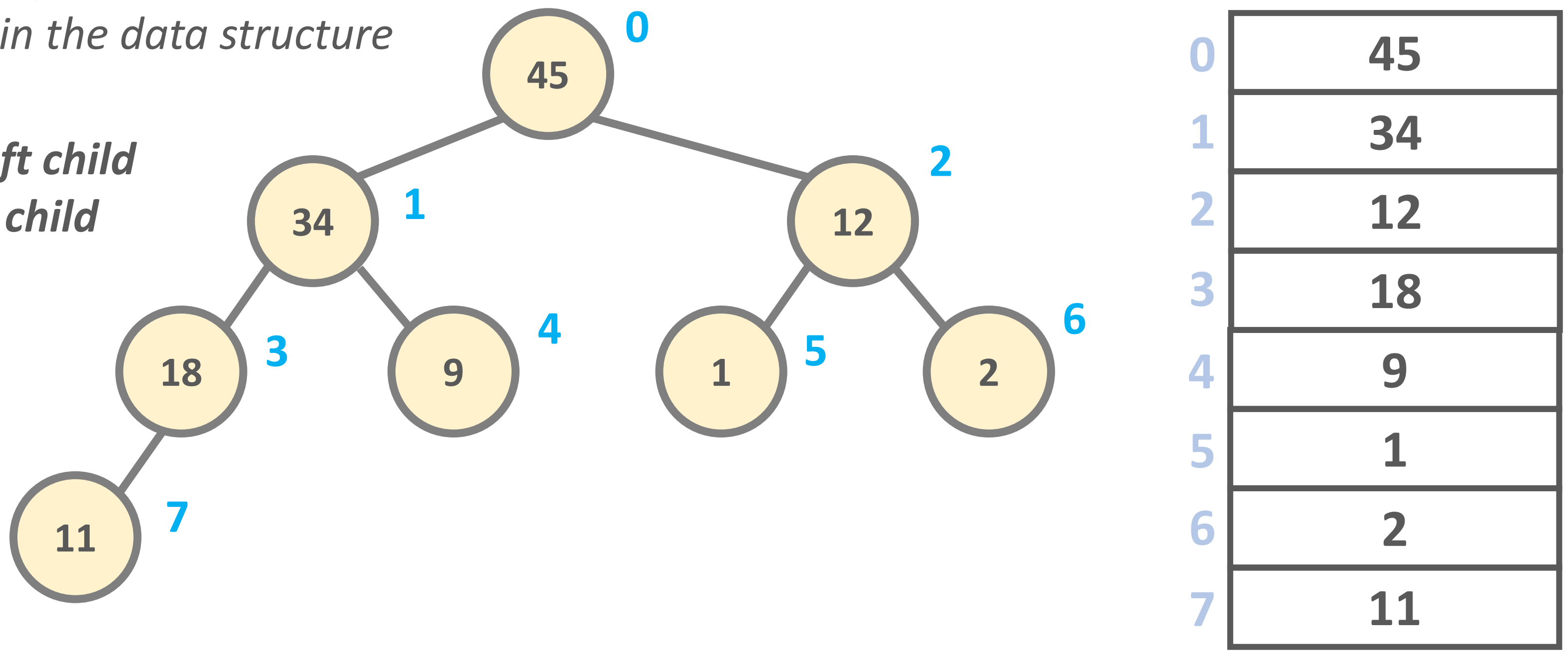


0	45
1	34
2	12
3	18
4	9
5	1
6	2
7	11

Representing Heaps

*the **root node** of a **max heap***
*is the **largest item** in the data structure*

*the node with index **i** has **left child** with index **2i+1** and **right child** with index **2i+2***



Building a Max Heap

INSERT(23)

0	
1	
2	
3	
4	
5	
6	
7	

Building a Max Heap

INSERT(23)

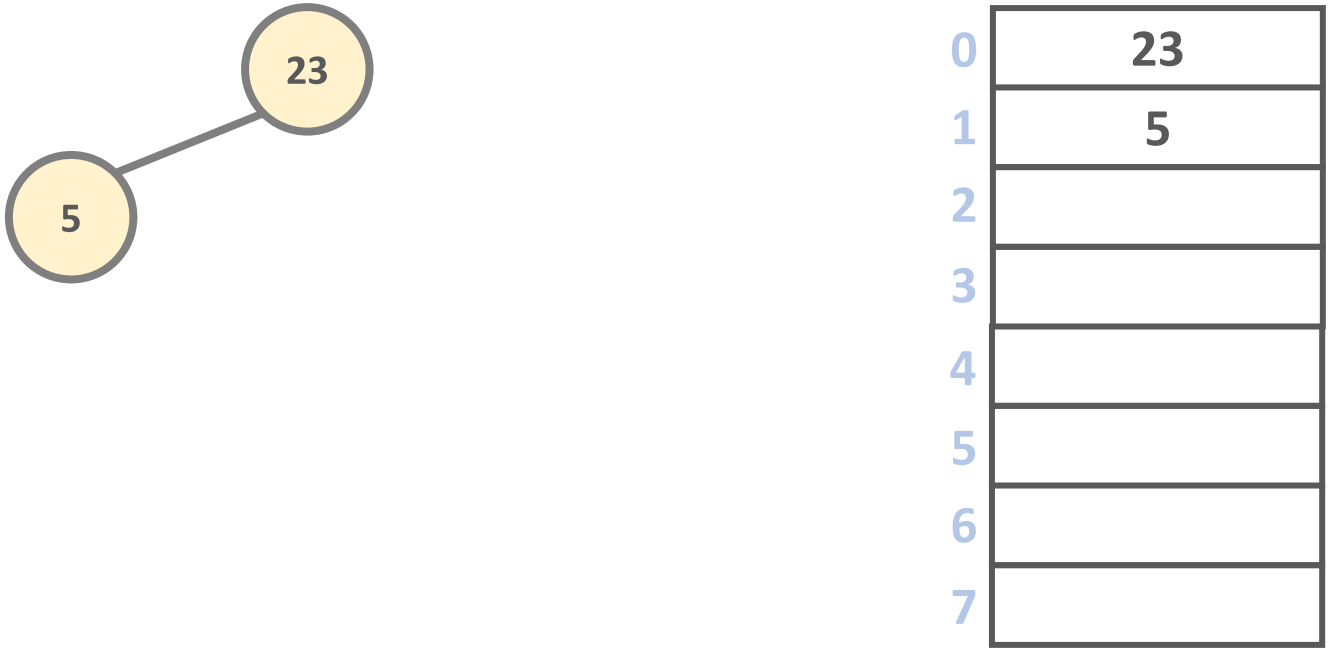
0	23
1	
2	
3	
4	
5	
6	
7	

Building a Max Heap

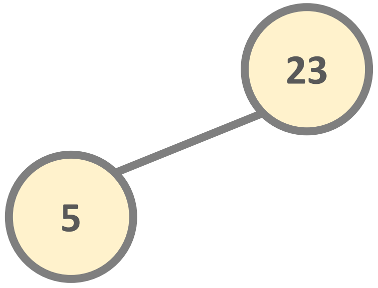
0	23
1	
2	
3	
4	
5	
6	
7	

Building a Max Heap

INSERT(5)



Building a Max Heap



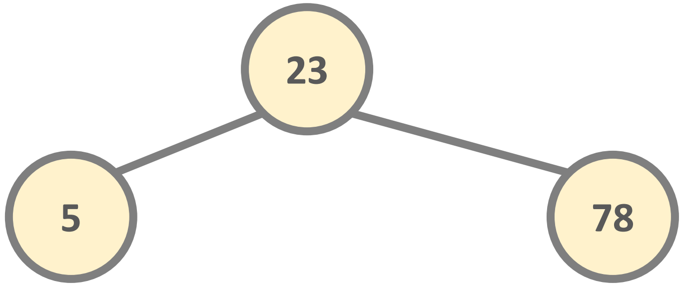
0	23
1	5
2	
3	
4	
5	
6	
7	

Building a Max Heap

INSERT(78)

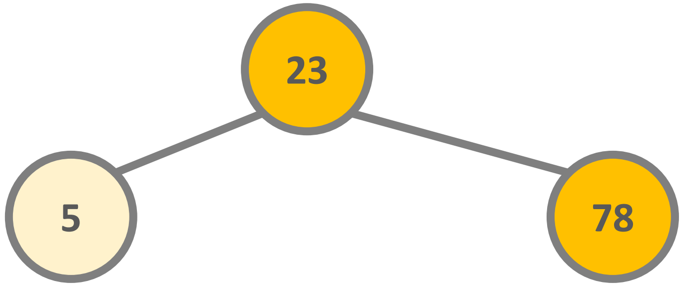
Building a Max Heap

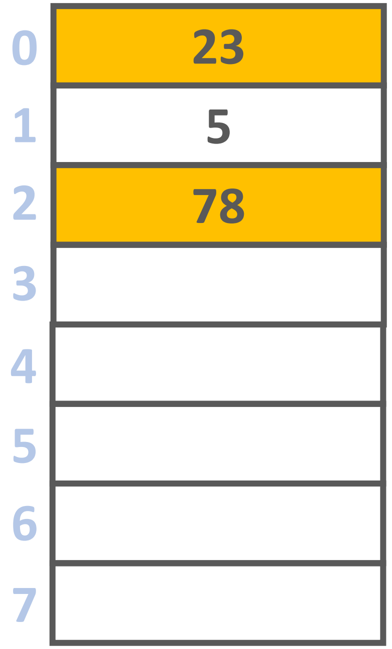
INSERT(78)



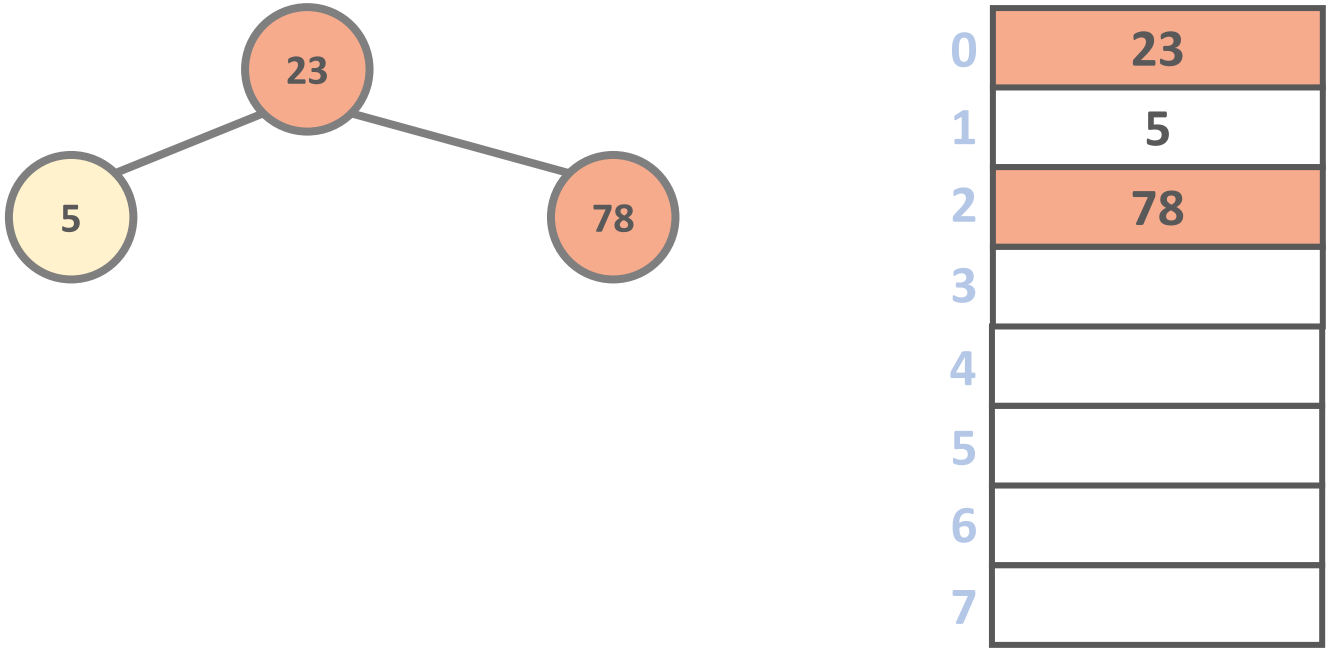
0	23
1	5
2	78
3	
4	
5	
6	
7	

Building a Max Heap

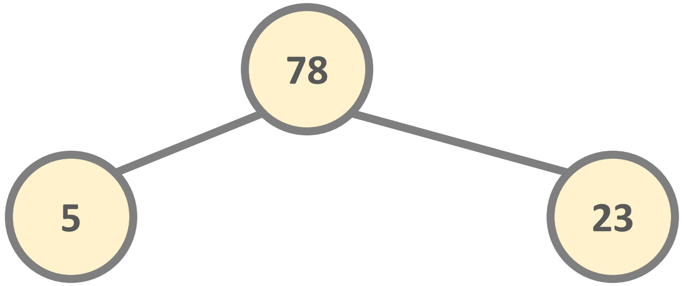




Building a Max Heap



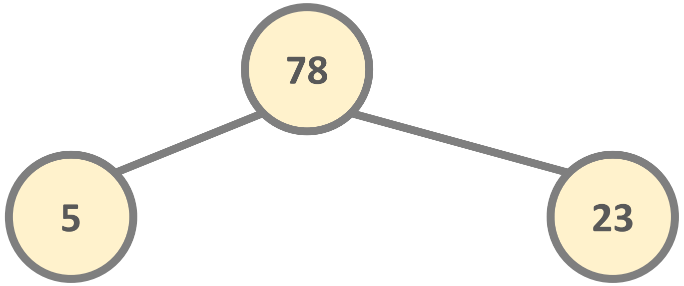
Building a Max Heap

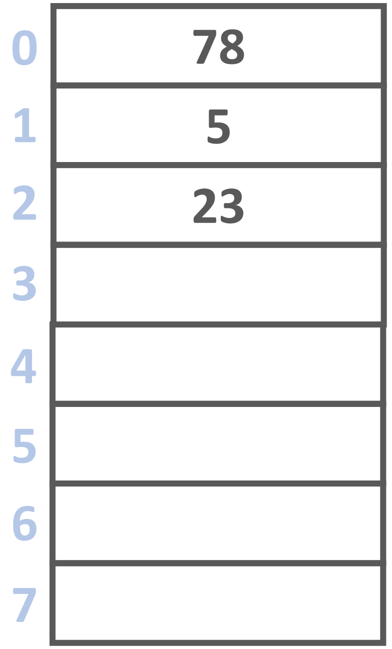


0	78
1	5
2	23
3	
4	
5	
6	
7	

Building a Max Heap

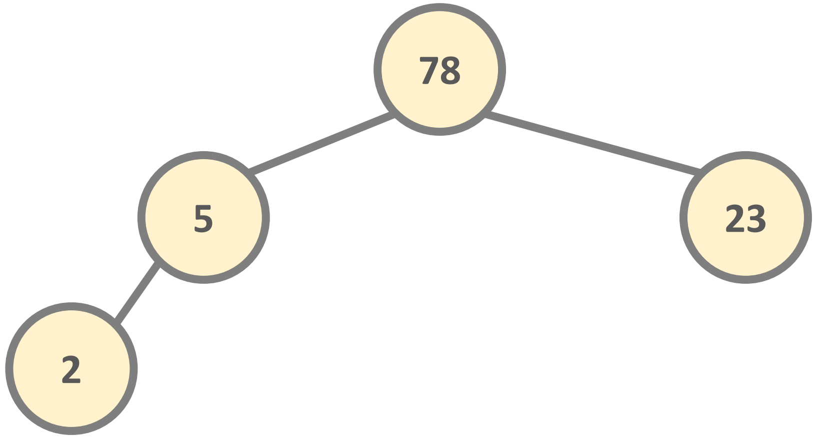
INSERT(2)





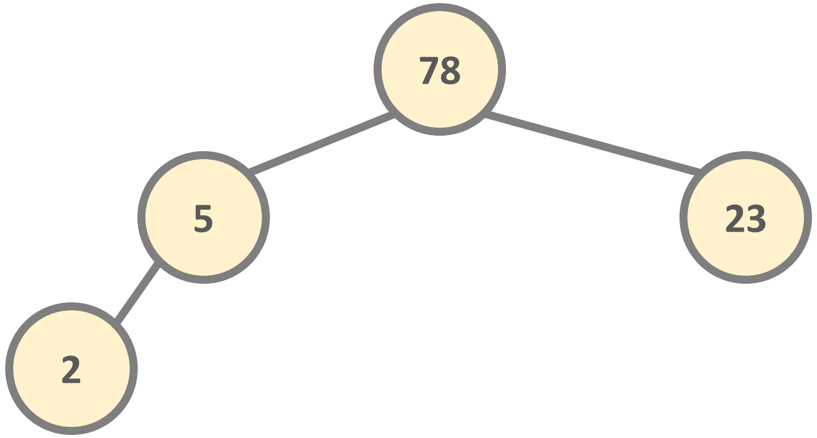
Building a Max Heap

INSERT(2)



0	78
1	5
2	23
3	2
4	
5	
6	
7	

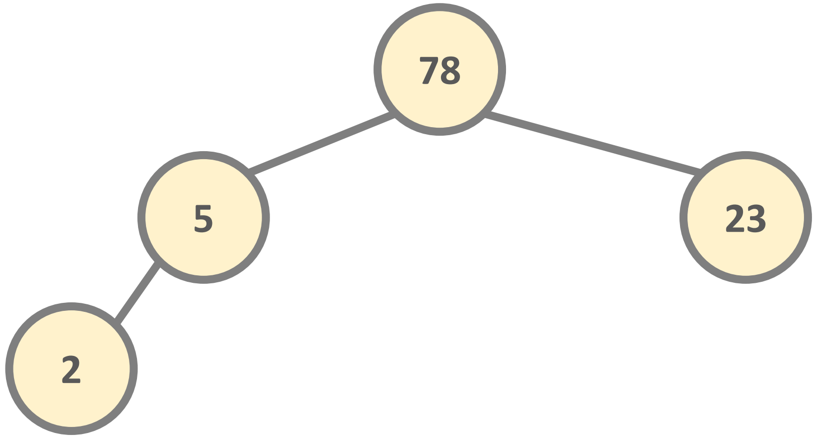
Building a Max Heap



0	78
1	5
2	23
3	2
4	
5	
6	
7	

Building a Max Heap

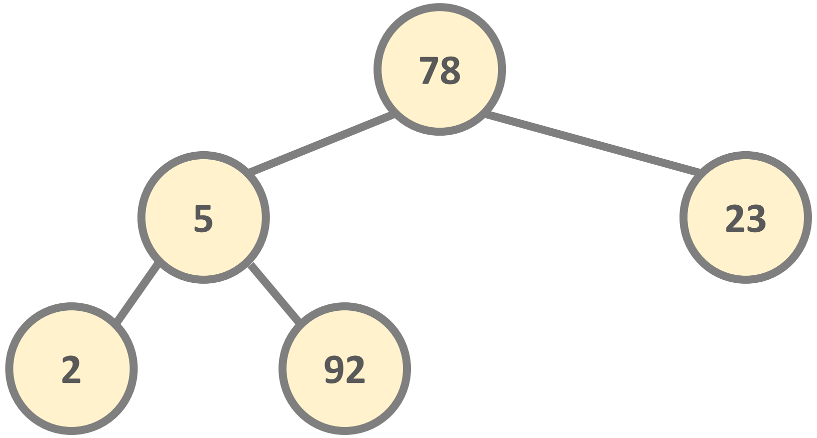
INSERT(92)



0	78
1	5
2	23
3	2
4	
5	
6	
7	

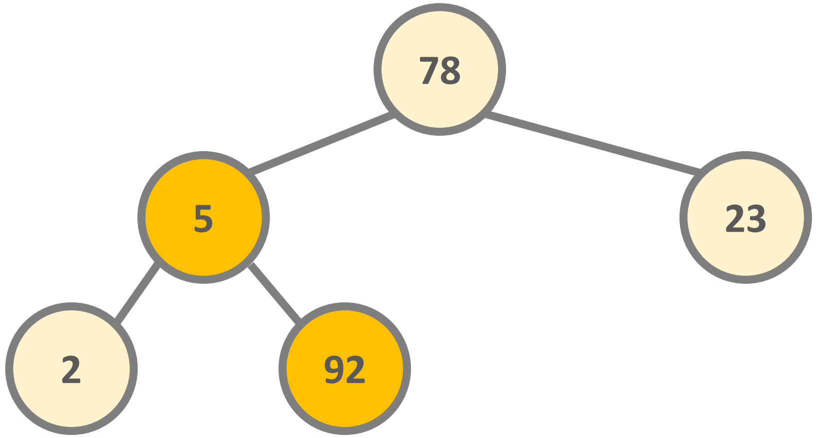
Building a Max Heap

INSERT(92)



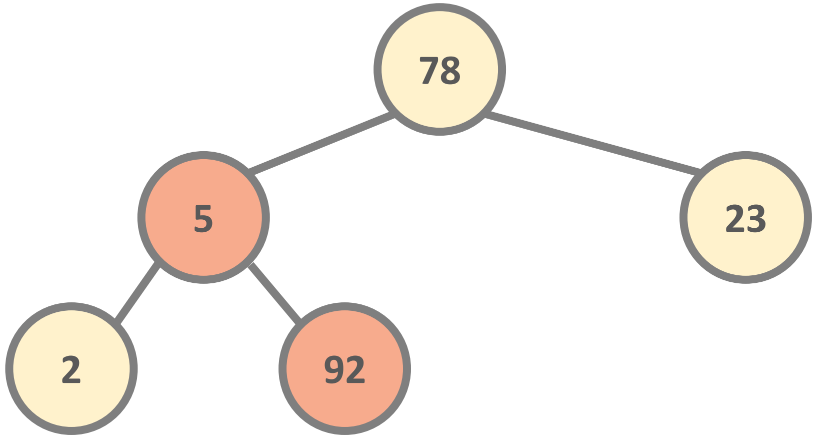
0	78
1	5
2	23
3	2
4	92
5	
6	
7	

Building a Max Heap



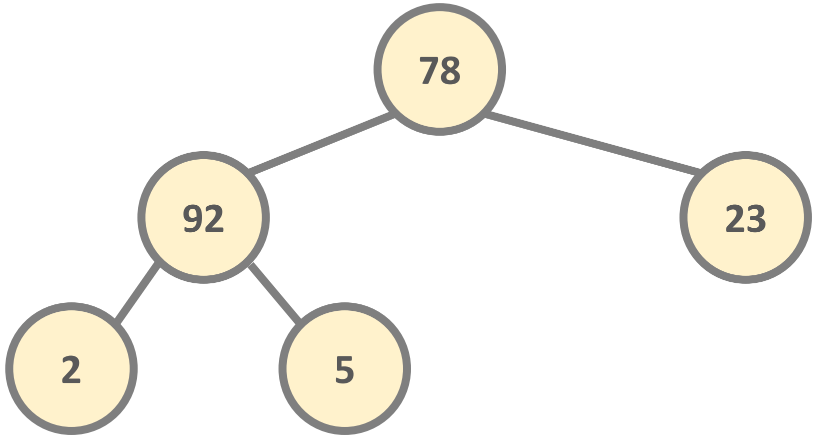
0	78
1	5
2	23
3	2
4	92
5	
6	
7	

Building a Max Heap



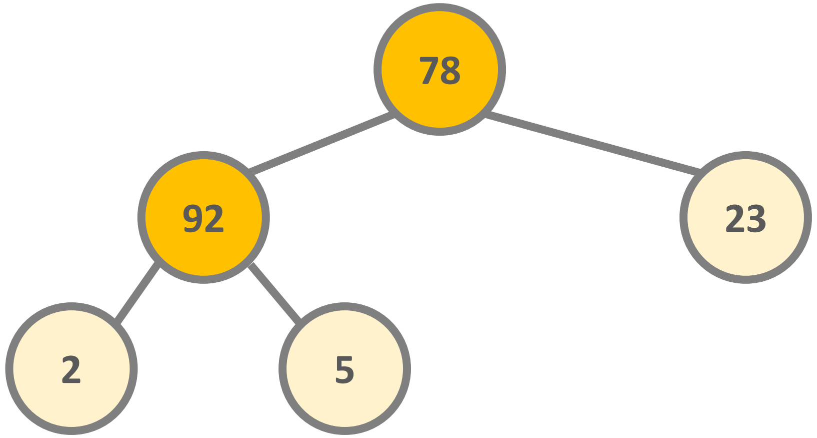
0	78
1	5
2	23
3	2
4	92
5	
6	
7	

Building a Max Heap



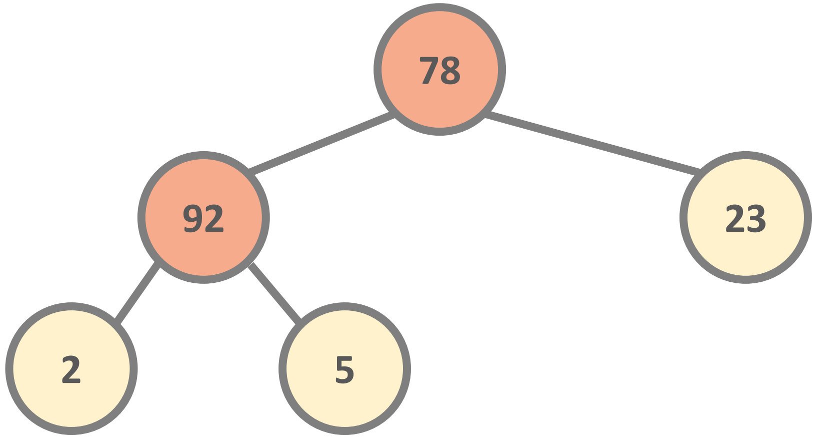
0	78
1	92
2	23
3	2
4	5
5	
6	
7	

Building a Max Heap



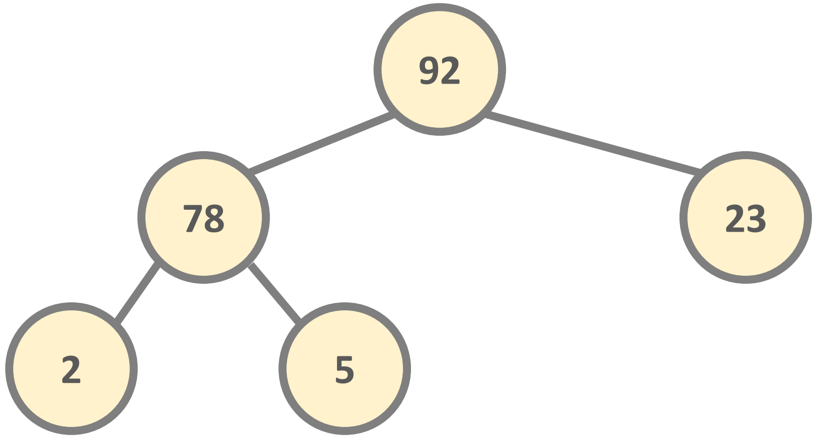
0	78
1	92
2	23
3	2
4	5
5	
6	
7	

Building a Max Heap



0	78
1	92
2	23
3	2
4	5
5	
6	
7	

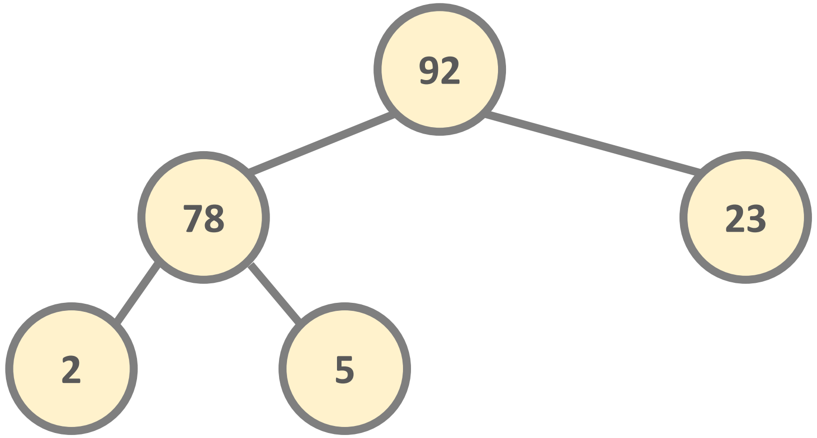
Building a Max Heap



0	92
1	78
2	23
3	2
4	5
5	
6	
7	

Building a Max Heap

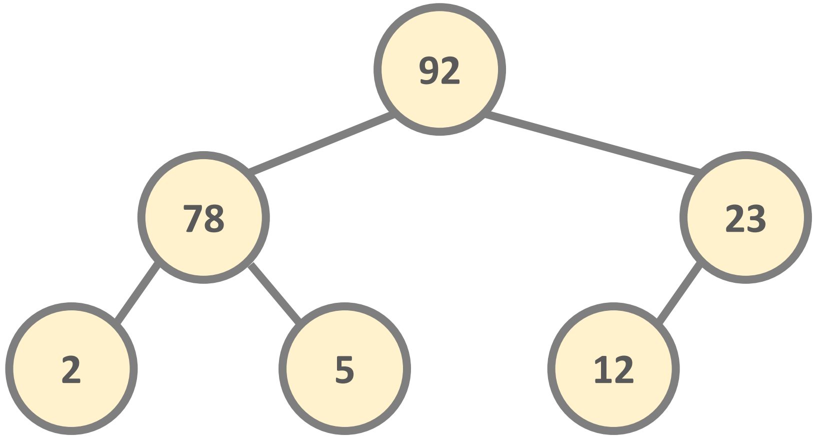
INSERT(12)



0	92
1	78
2	23
3	2
4	5
5	
6	
7	

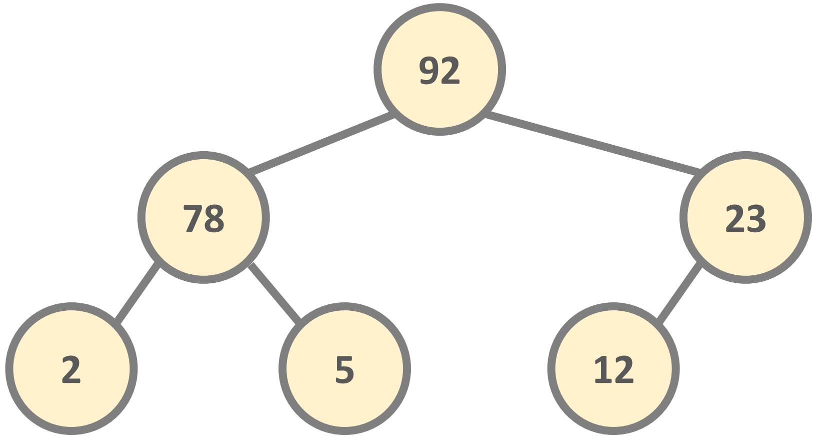
Building a Max Heap

INSERT(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	
7	

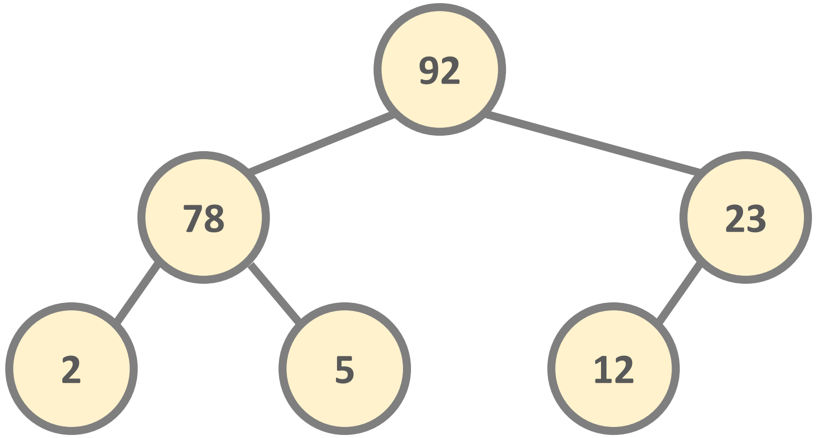
Building a Max Heap



0	92
1	78
2	23
3	2
4	5
5	12
6	
7	

Building a Max Heap

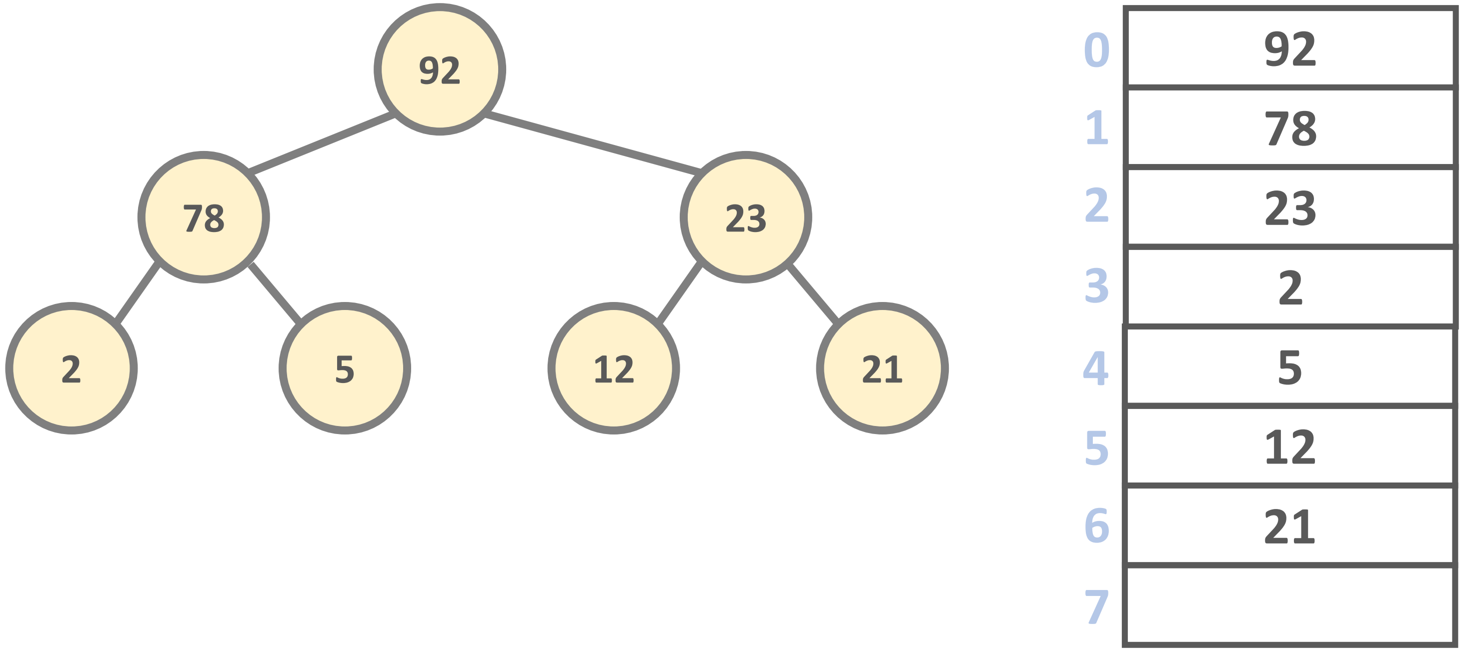
INSERT(21)



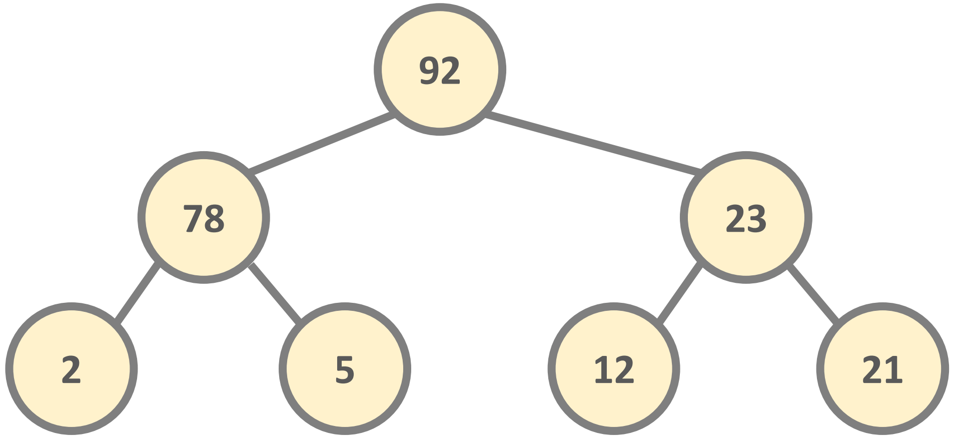
0	92
1	78
2	23
3	2
4	5
5	12
6	
7	

Building a Max Heap

INSERT(21)



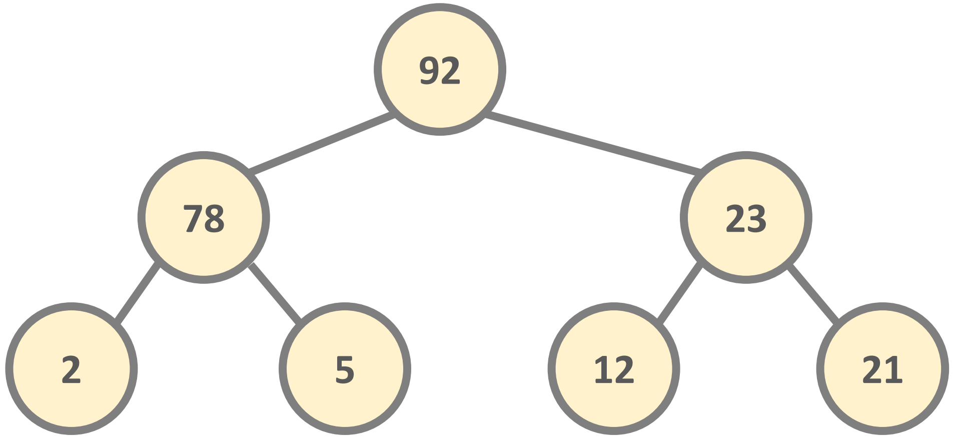
Building a Max Heap



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Building a Max Heap

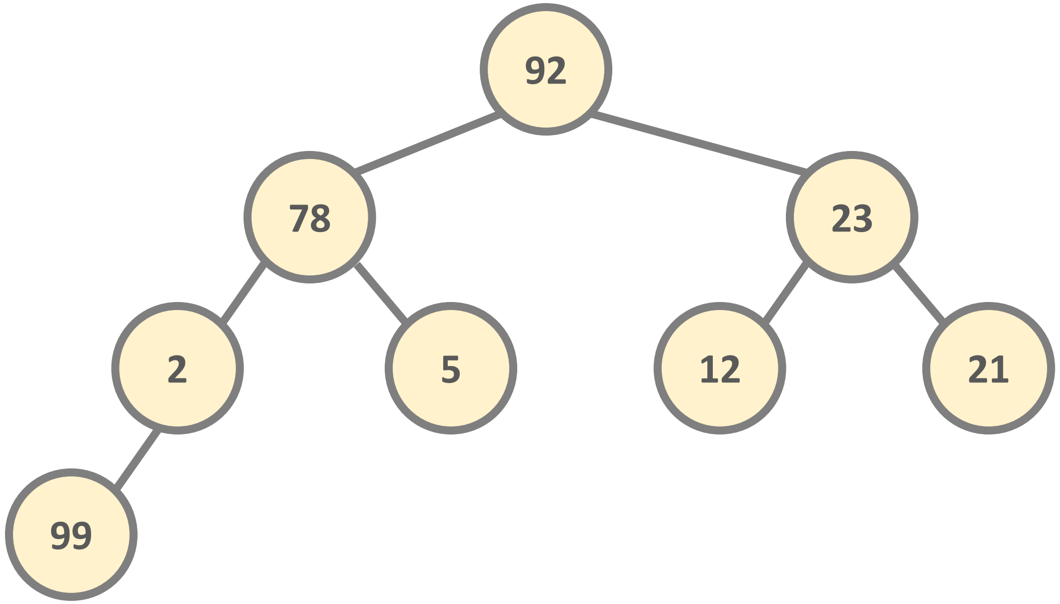
INSERT(99)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

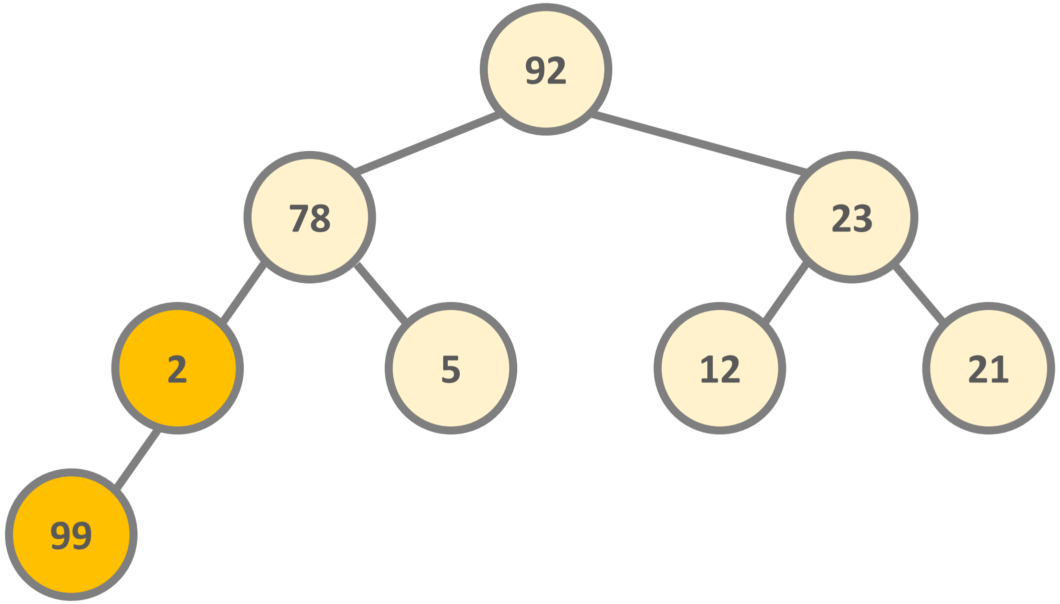
Building a Max Heap

INSERT(99)



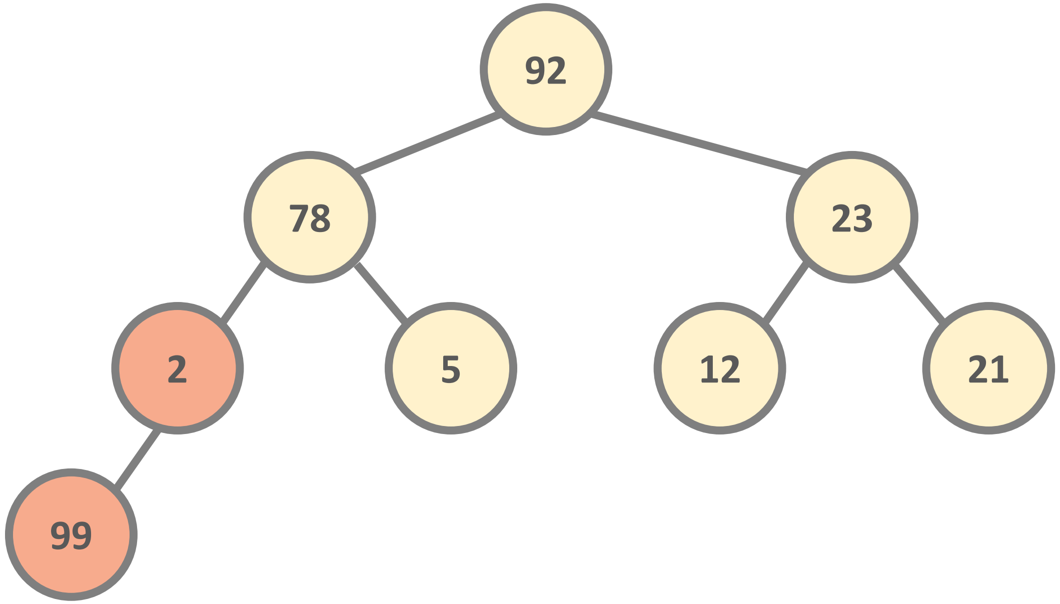
0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	99

Building a Max Heap



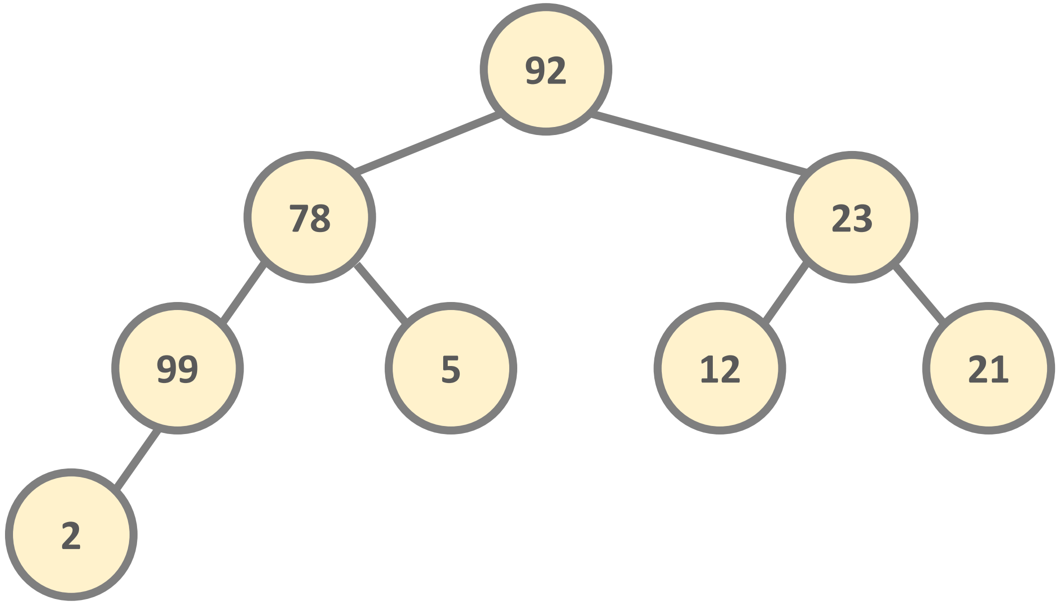
0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	99

Building a Max Heap



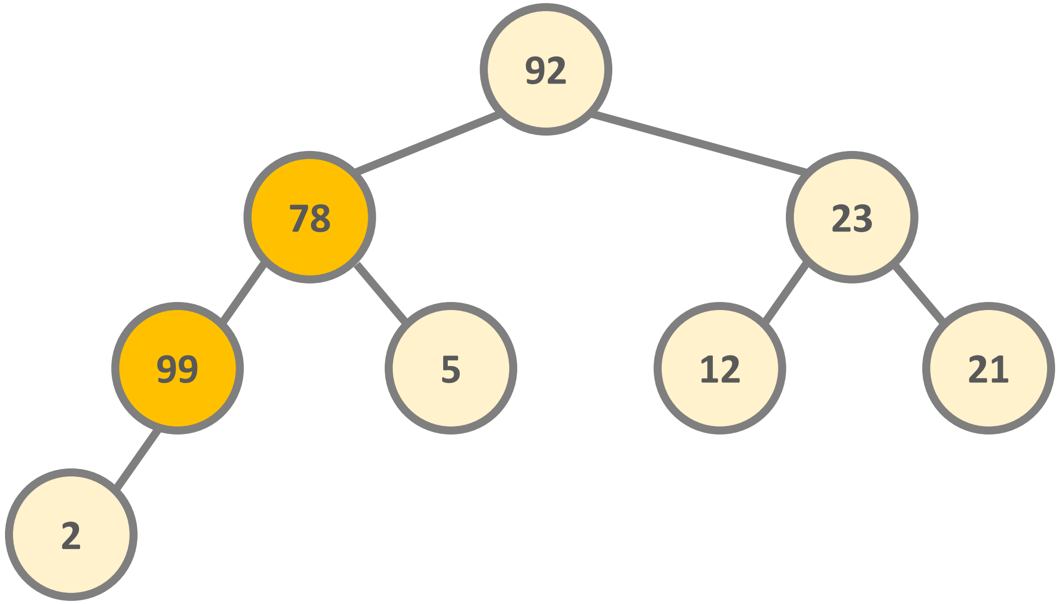
0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	99

Building a Max Heap



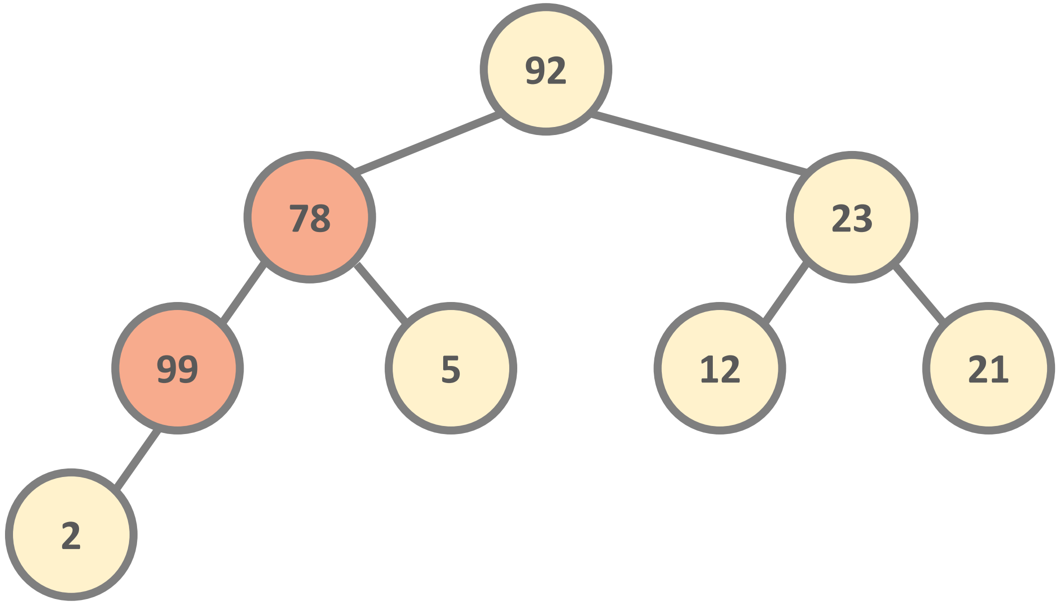
0	92
1	78
2	23
3	99
4	5
5	12
6	21
7	2

Building a Max Heap



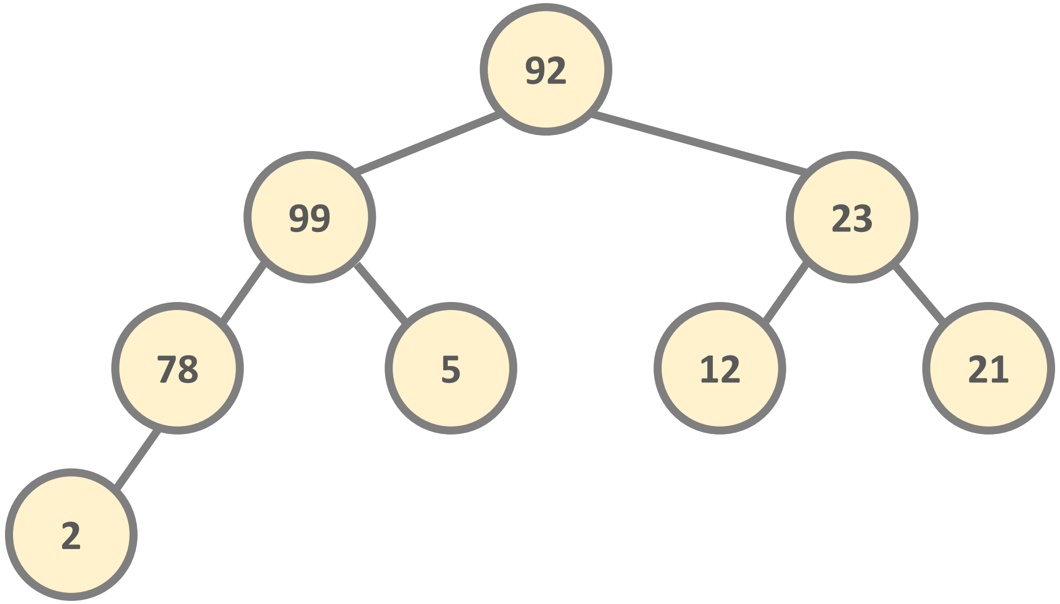
0	92
1	78
2	23
3	99
4	5
5	12
6	21
7	2

Building a Max Heap



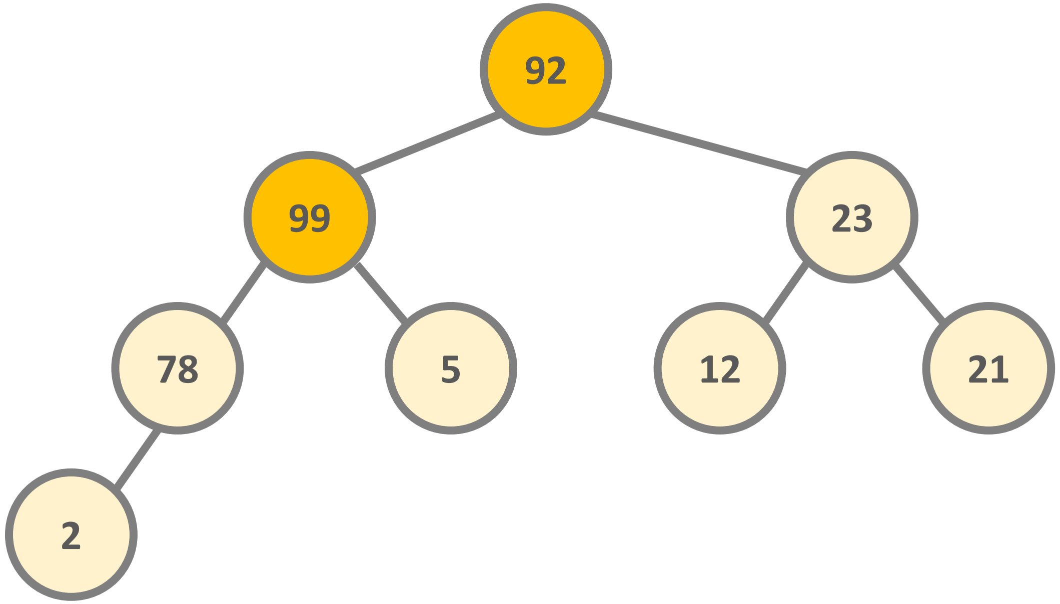
0	92
1	78
2	23
3	99
4	5
5	12
6	21
7	2

Building a Max Heap



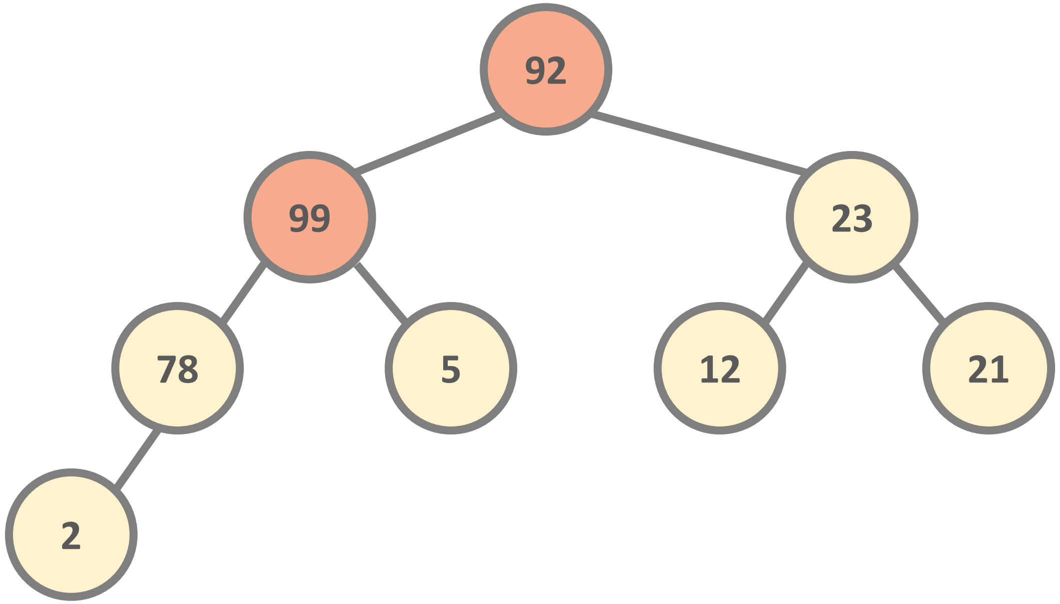
0	92
1	99
2	23
3	78
4	5
5	12
6	21
7	2

Building a Max Heap



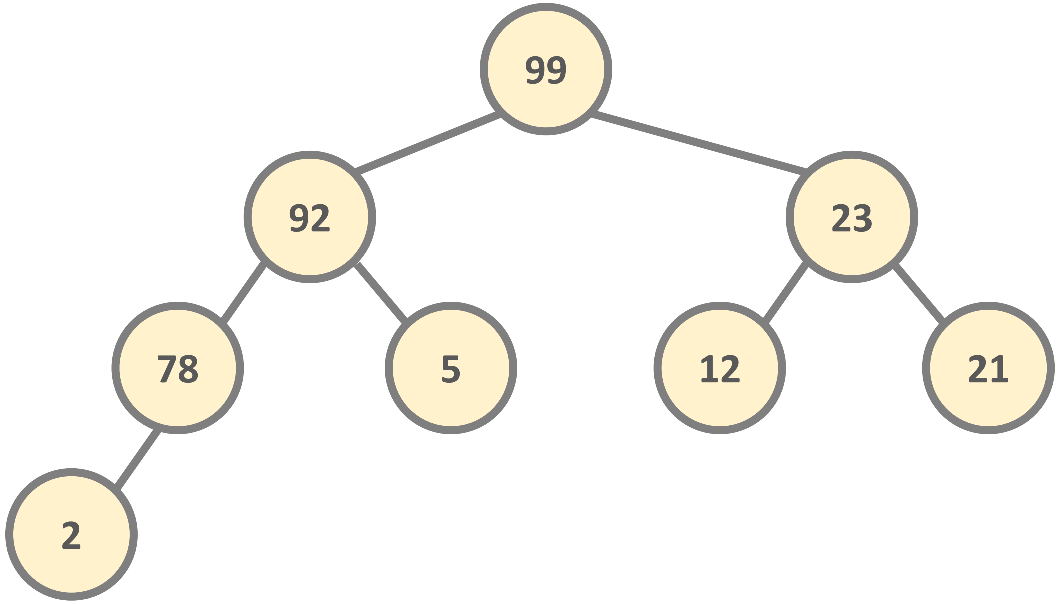
0	92
1	99
2	23
3	78
4	5
5	12
6	21
7	2

Building a Max Heap



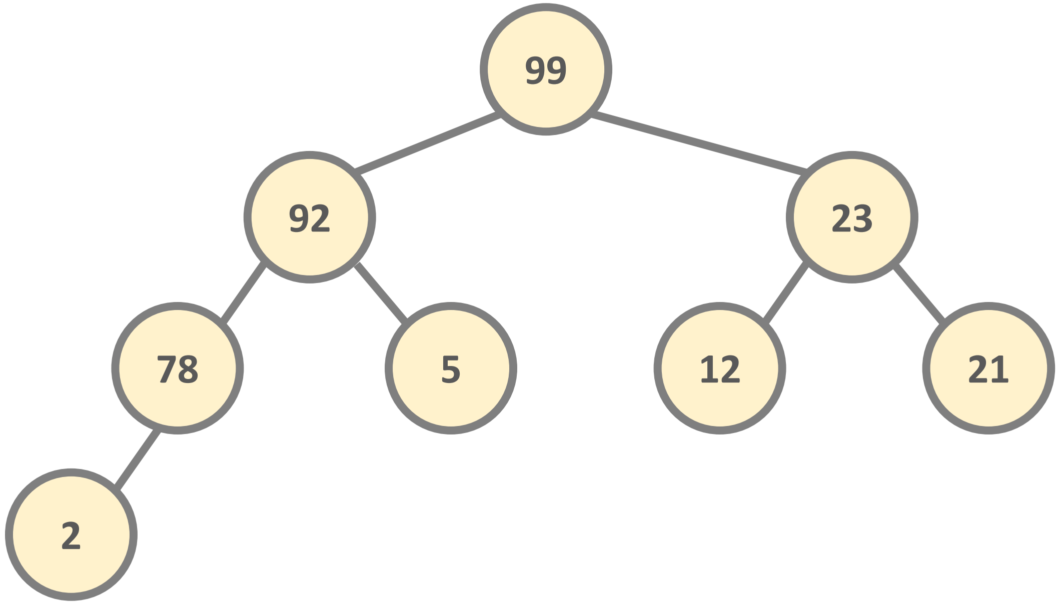
0	92
1	99
2	23
3	78
4	5
5	12
6	21
7	2

Building a Max Heap



0	99
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Building a Max Heap



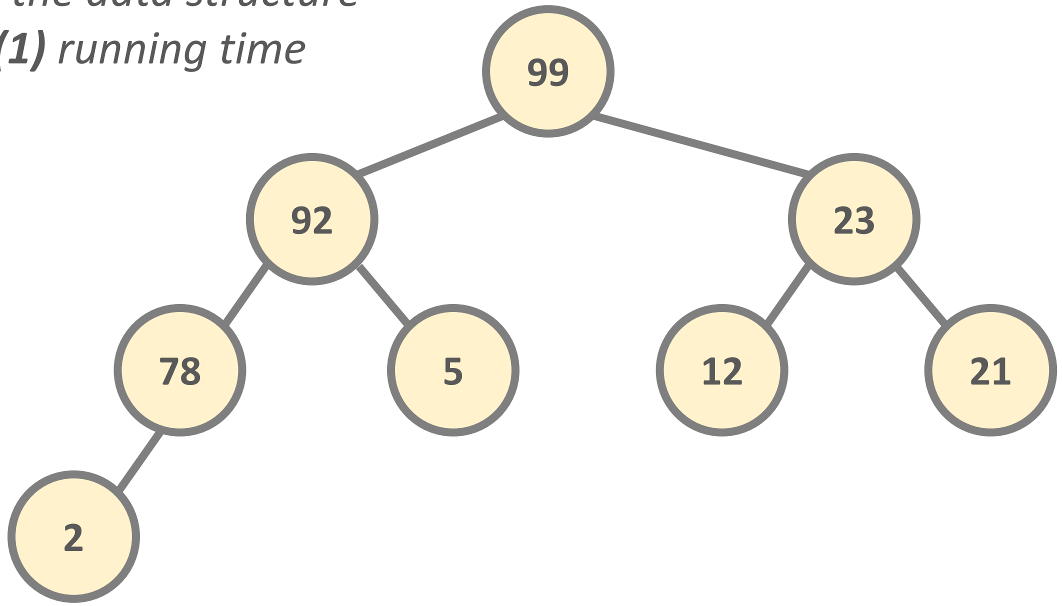
0	99
1	92
2	23
3	78
4	5
5	12
6	21
7	2

WE CAN GET THE MAX (MIN) ITEM IN O(1) RUNNING TIME

- of course after that we have to rearrange the tree -

Removing Max (Min) Item

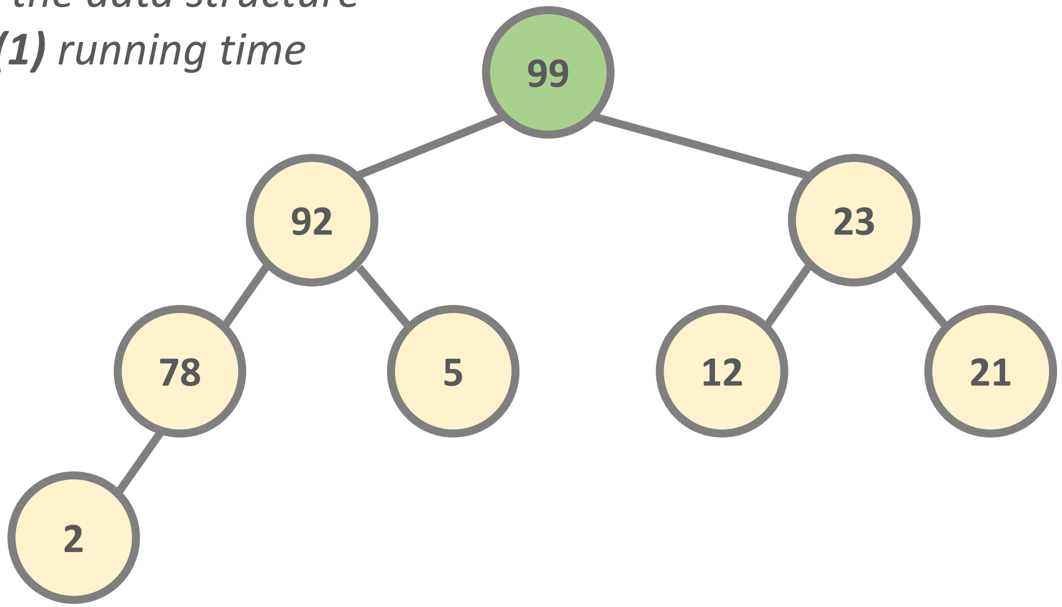
*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



0	99
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Removing Max (Min) Item

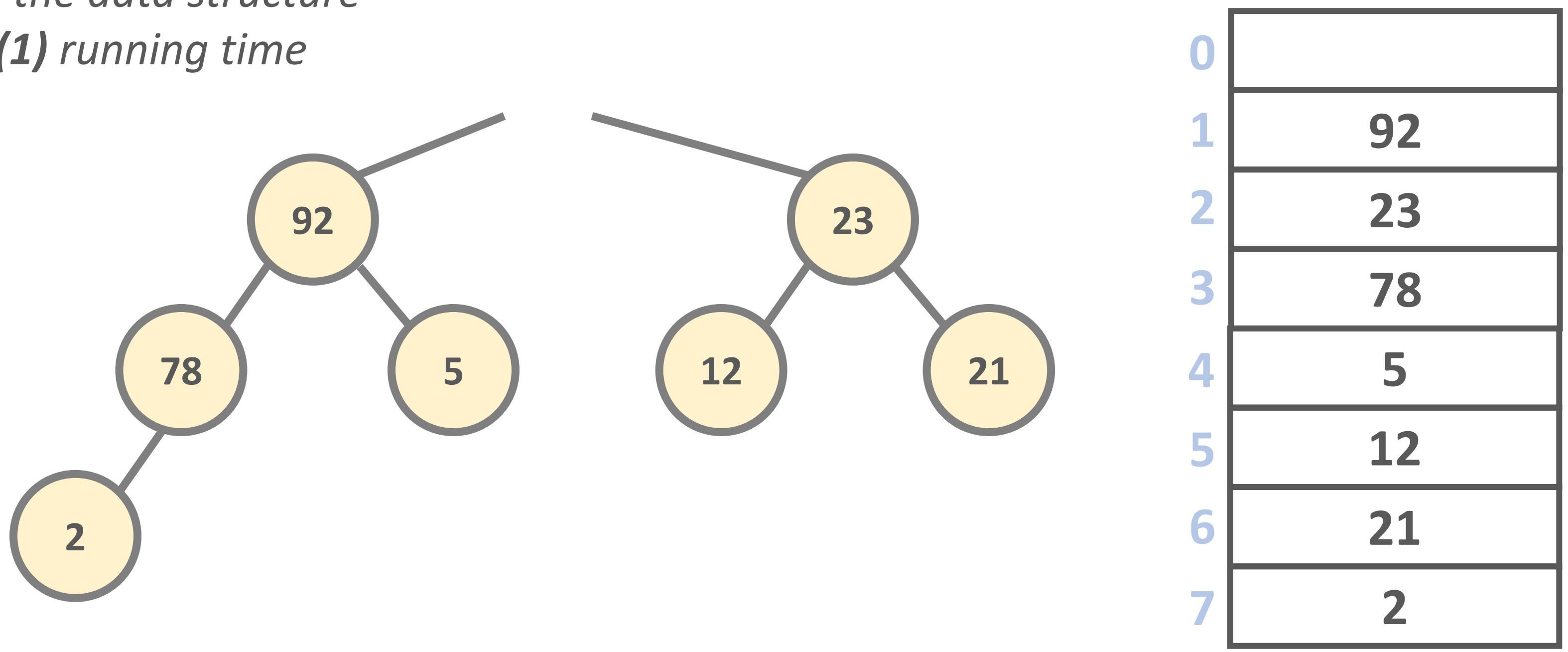
*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



0	99
1	92
2	23
3	78
4	5
5	12
6	21
7	2

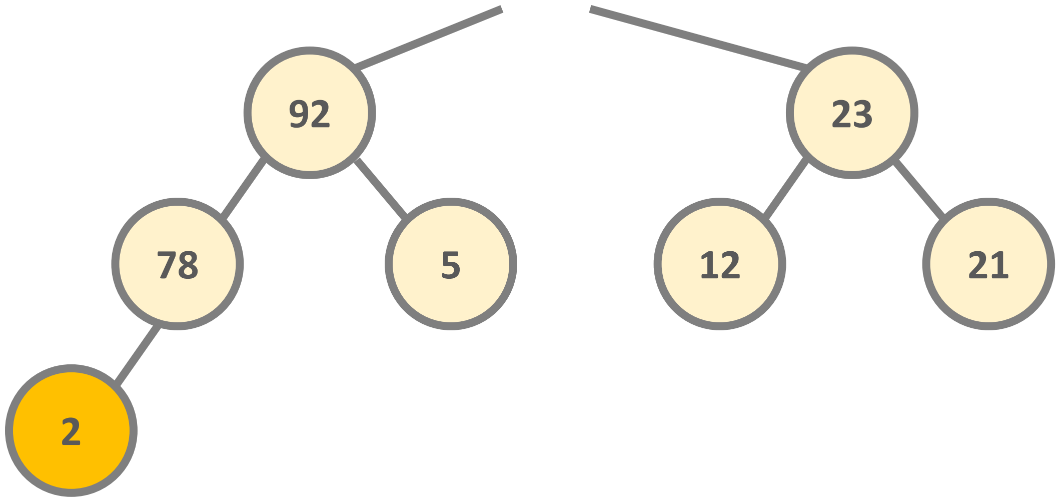
Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



Removing Max (Min) Item

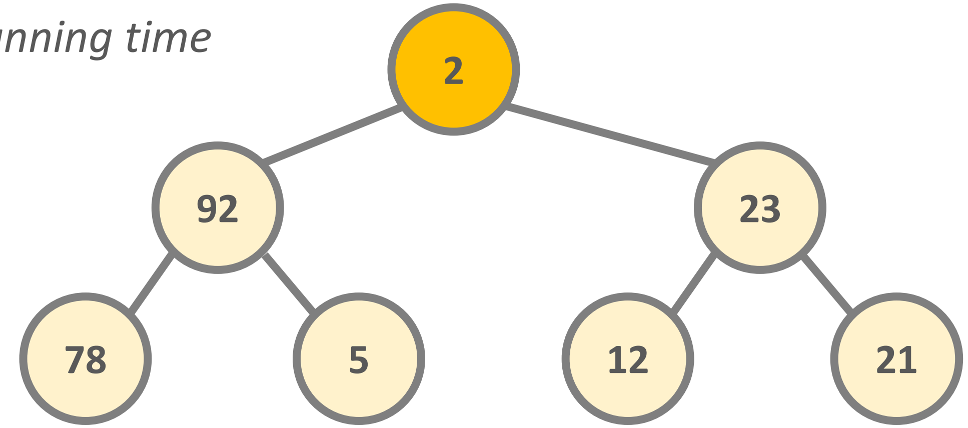
*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



0	
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Removing Max (Min) Item

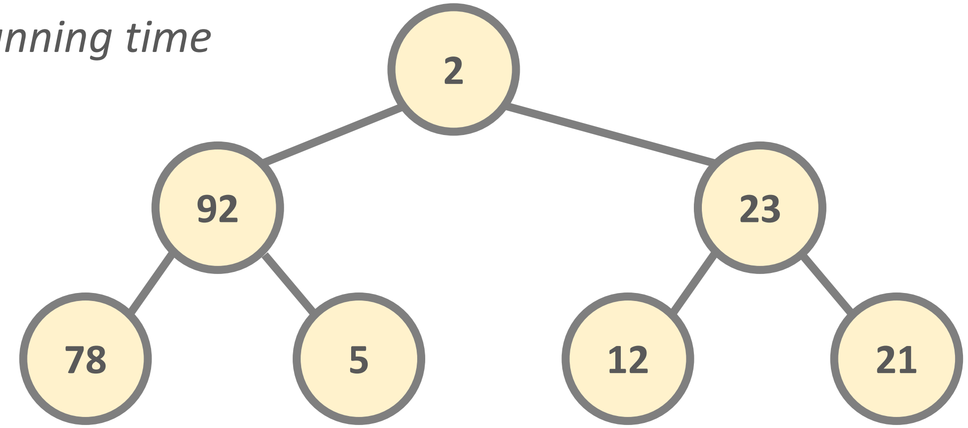
*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



0	2
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



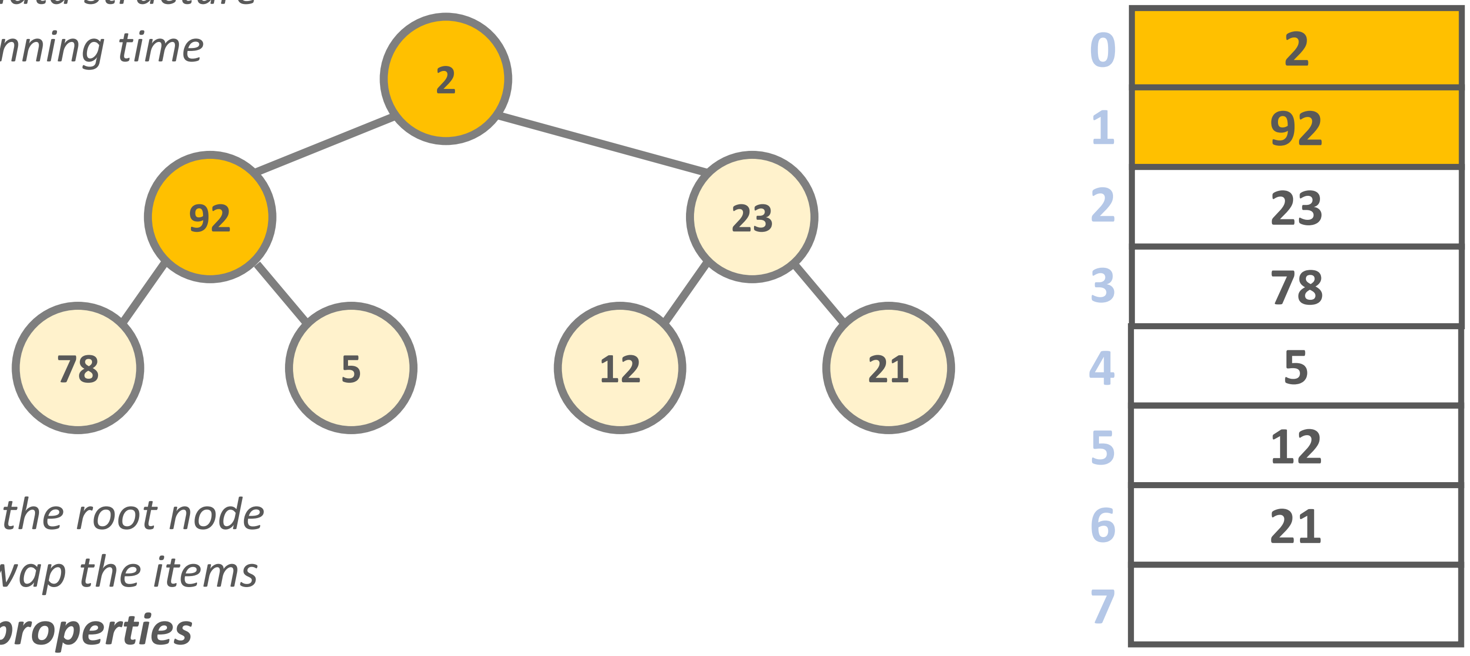
0	2
1	92
2	23
3	78
4	5
5	12
6	21
7	

we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*

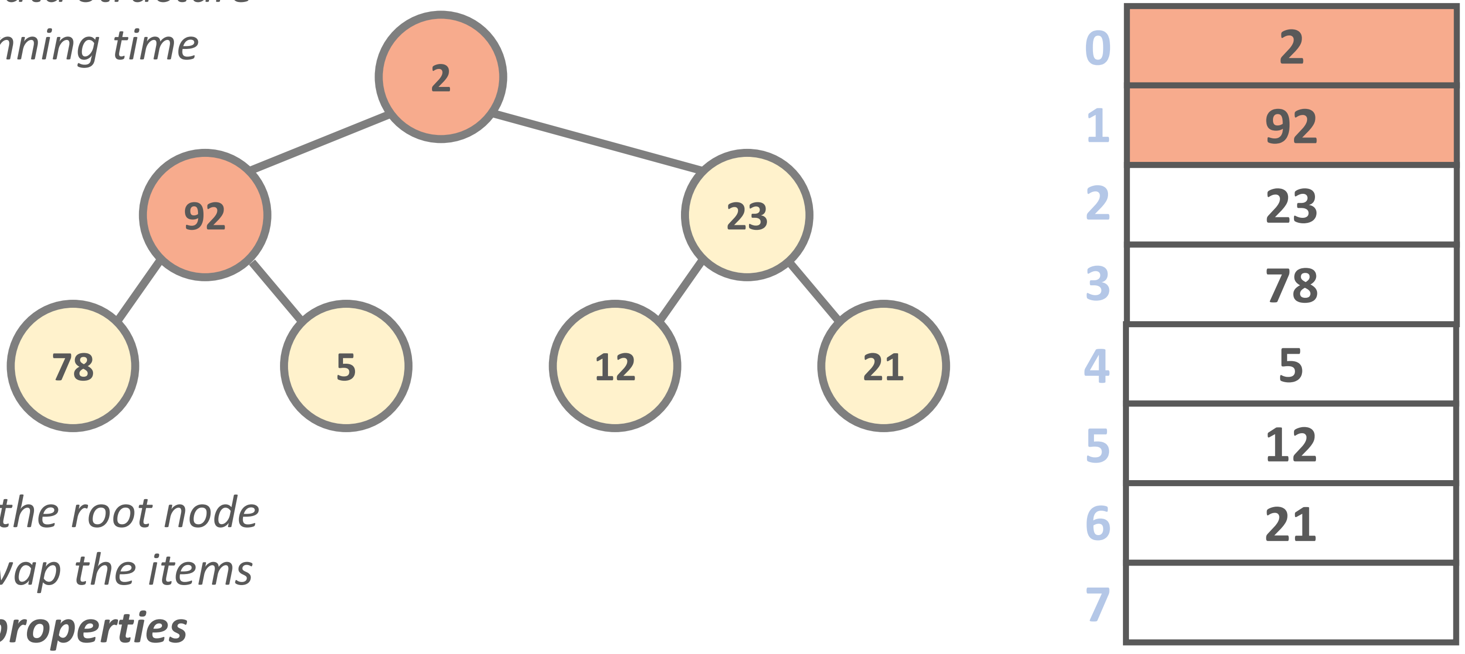


we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*

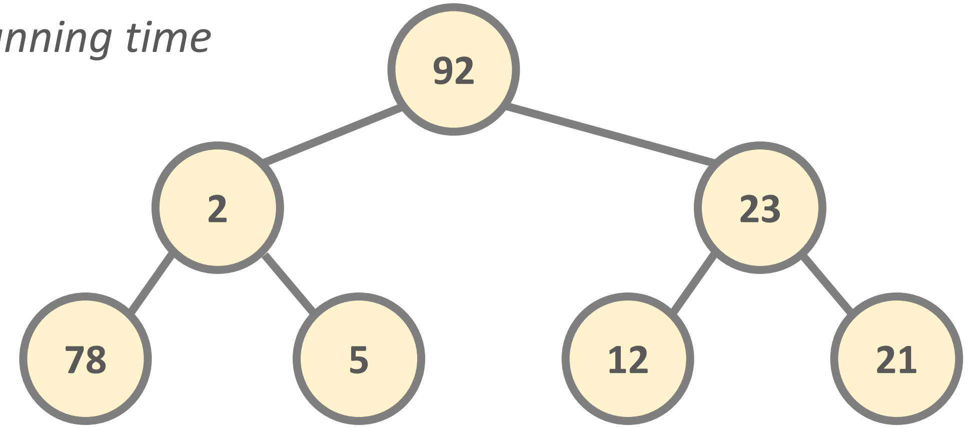


we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap** is the **largest item** in the data structure we can get it in **O(1)** running time*



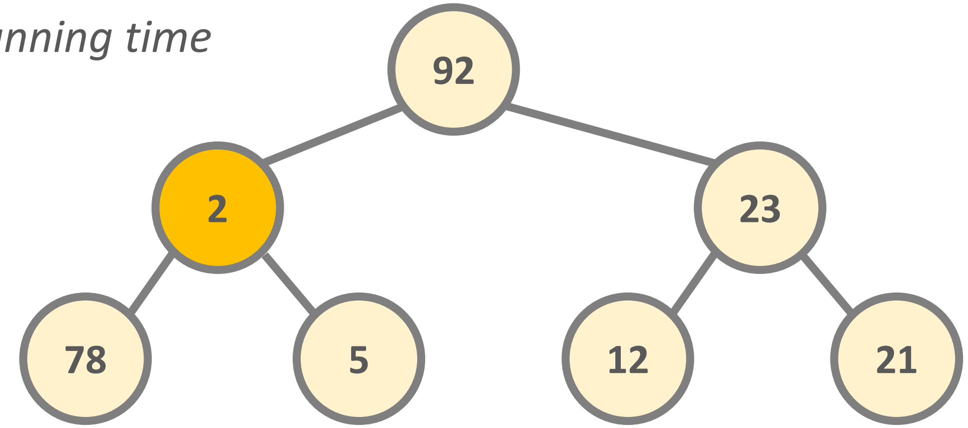
0	92
1	2
2	23
3	78
4	5
5	12
6	21
7	

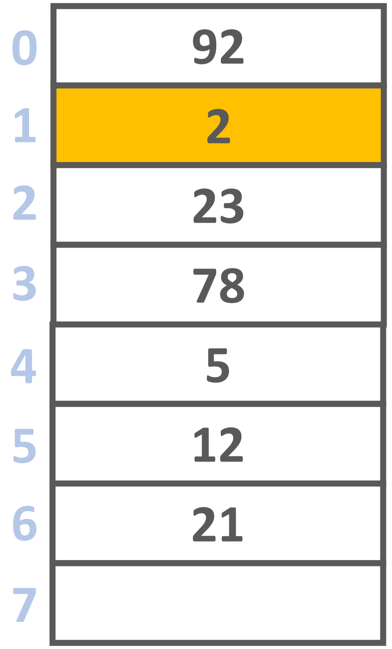
*we have to check starting with the root node to the leaf nodes whether to swap the items in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



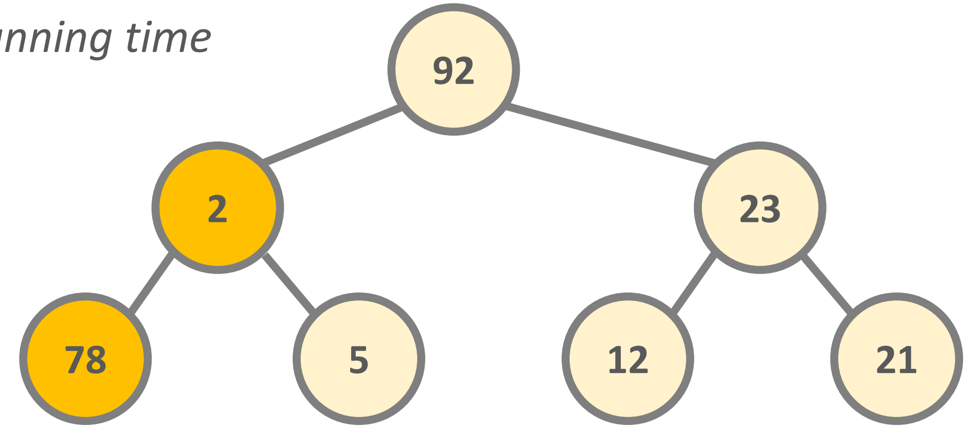


we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap** is the **largest item** in the data structure we can get it in **O(1)** running time*



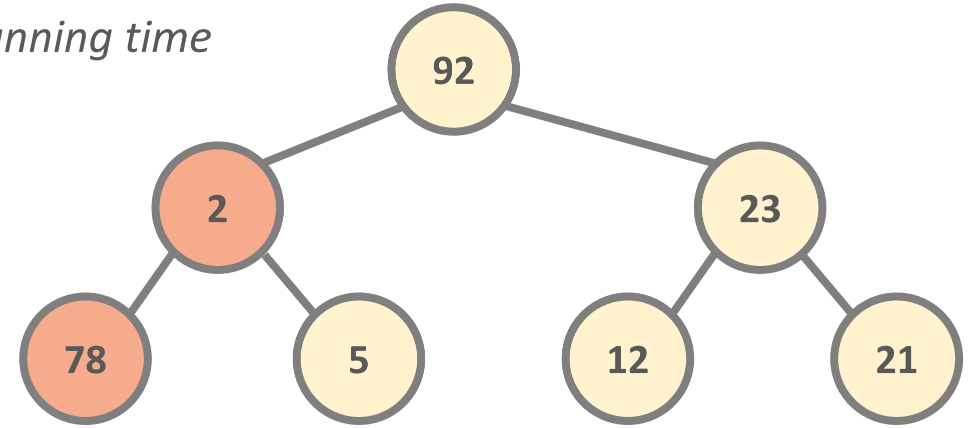
0	92
1	2
2	23
3	78
4	5
5	12
6	21
7	

*we have to check starting with the root node to the leaf nodes whether to swap the items in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



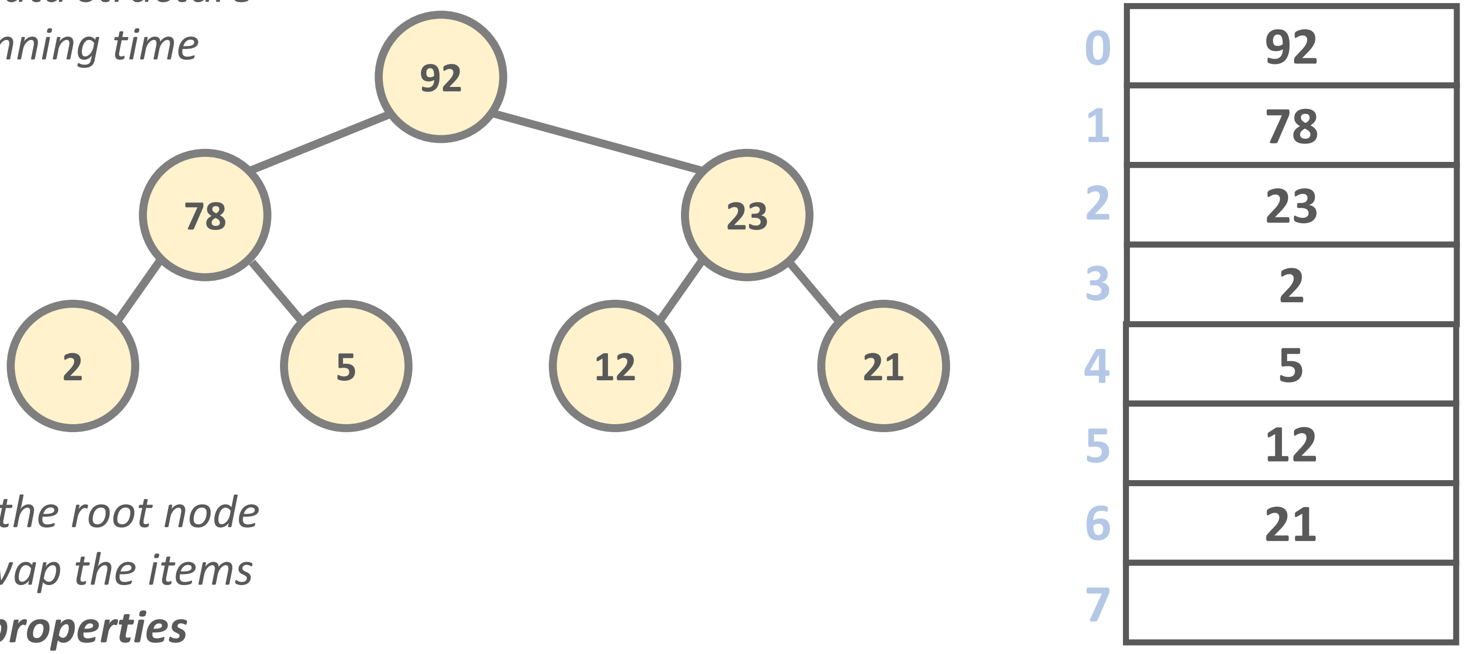
0	92
1	2
2	23
3	78
4	5
5	12
6	21
7	

we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION

Removing Max (Min) Item

*the **root node** of a **max heap***
*is the **largest item** in the data structure*
*we can get it in **O(1)** running time*



we have to check starting with the root node
to the leaf nodes whether to swap the items
*in order to verify the **heap properties***

HEAPIFY OPERATION in O(logN)

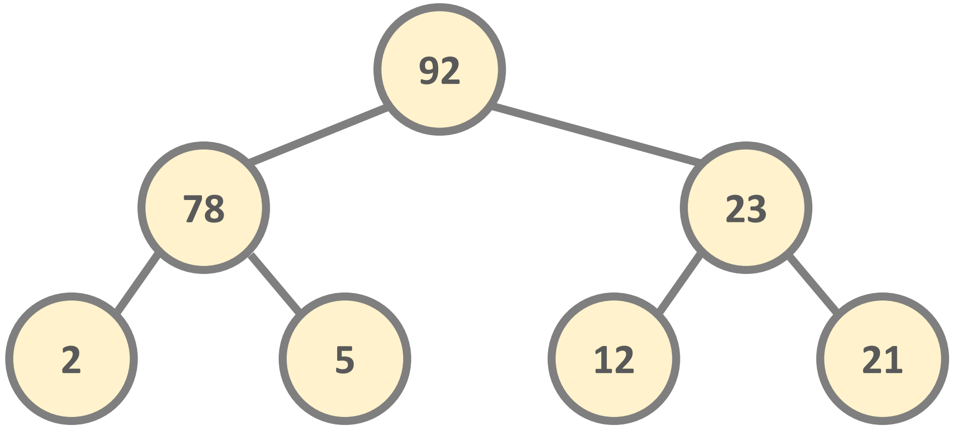
Heap Data Structure

(Algorithms and Data Structures)

Heaps

- removing the **root node** (and usually this is the case) can be done in **O(logN)** running time
- what if we want to remove an arbitrary item?
- first we have to find it in the array with **O(N)** linear search and then we can remove it in **O(logN)**
- **REMOVING AN ARBITRARY ITEM TAKES O(N) TIME**
- this is the same if we want to **find an item** in a heap
- heaps came to be to find and manipulate the **root node** (max or min item) in an efficient manner

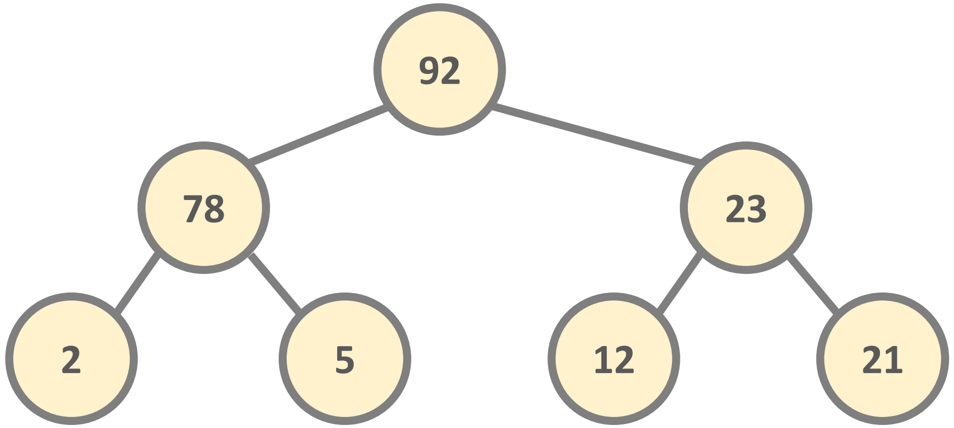
Heaps



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

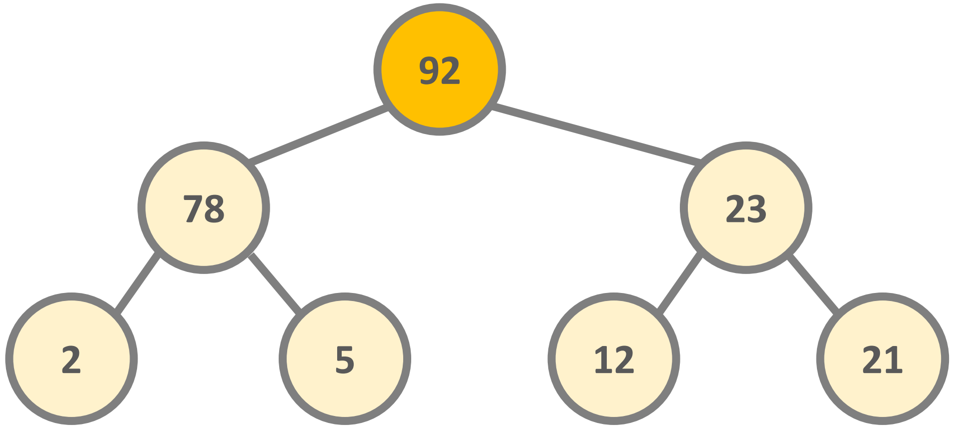
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

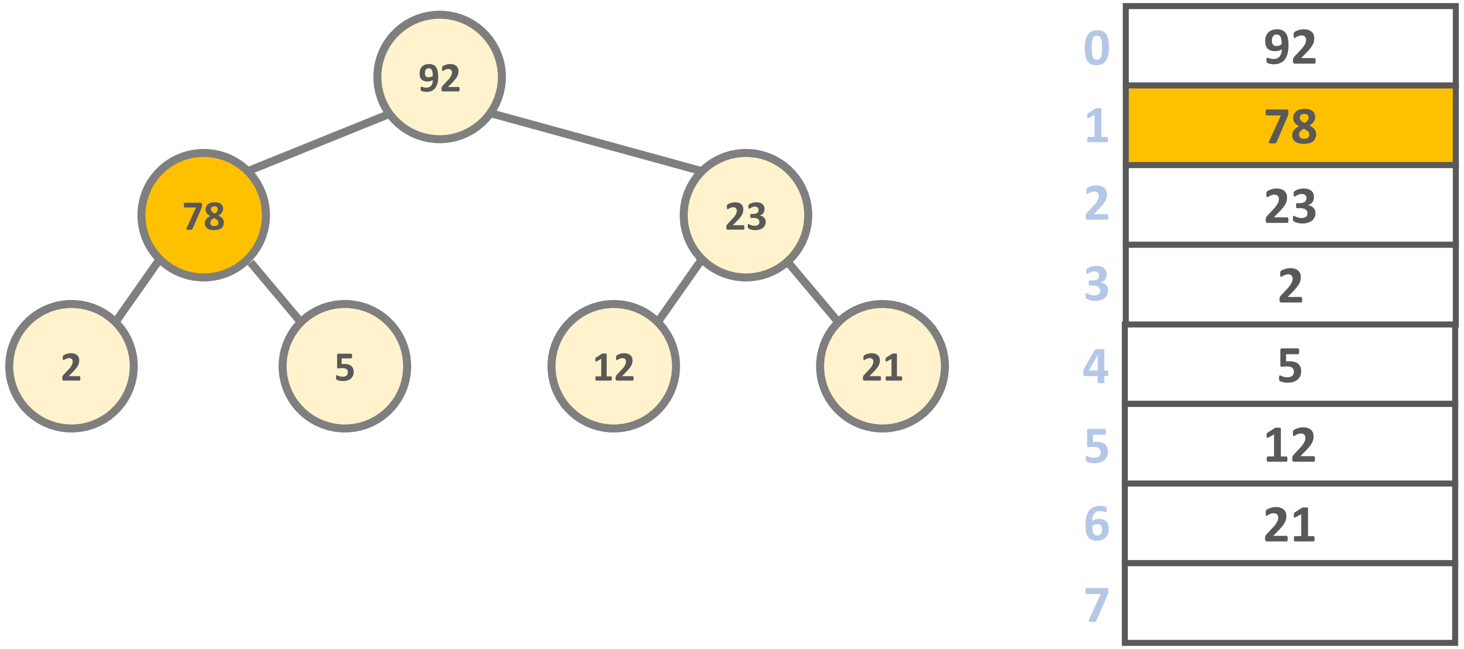
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

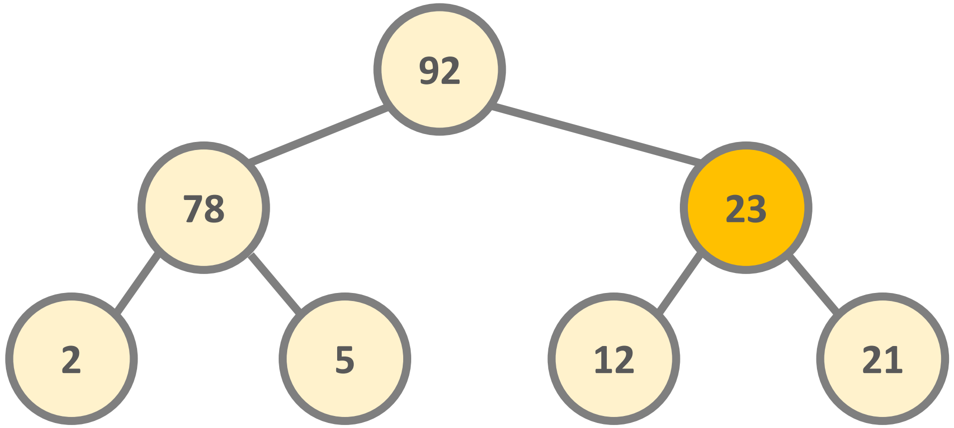
Heaps

REMOVE(12)



Heaps

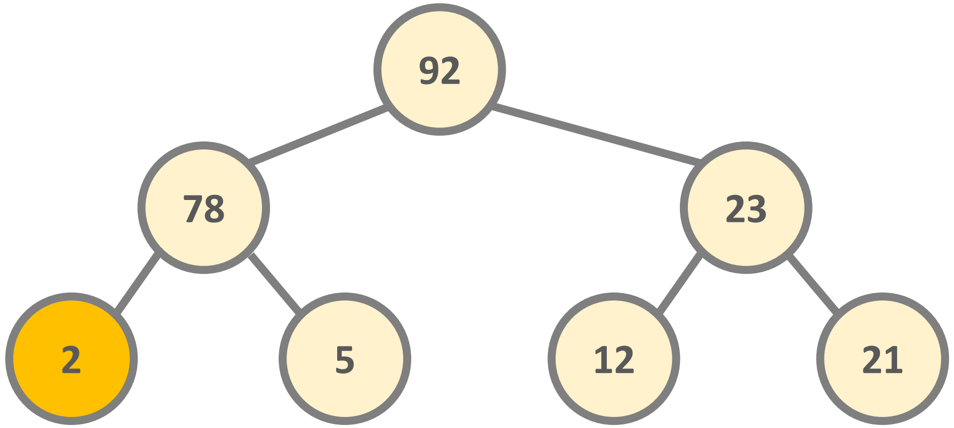
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

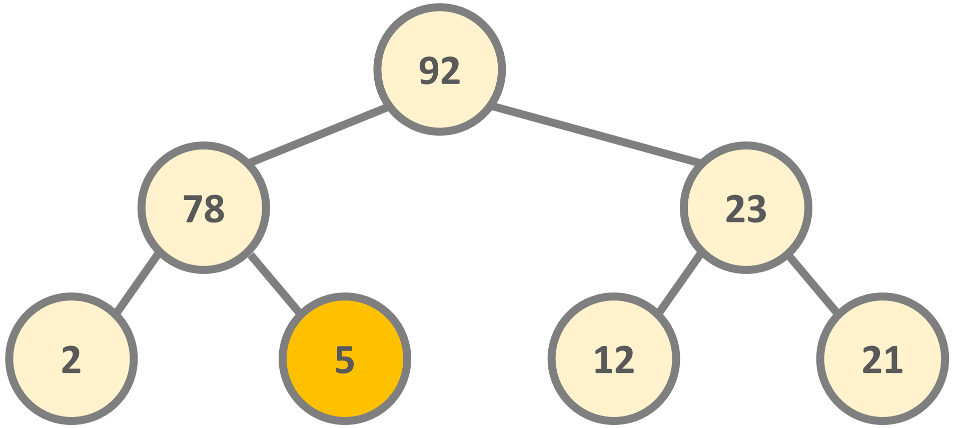
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

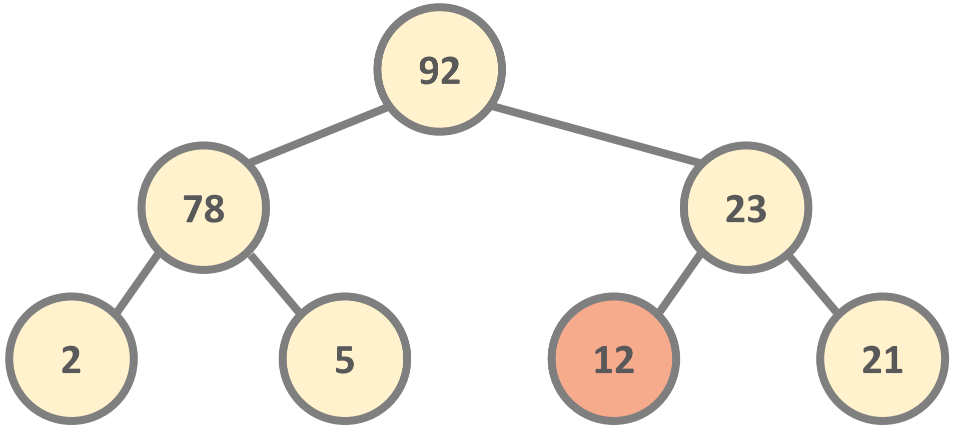
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

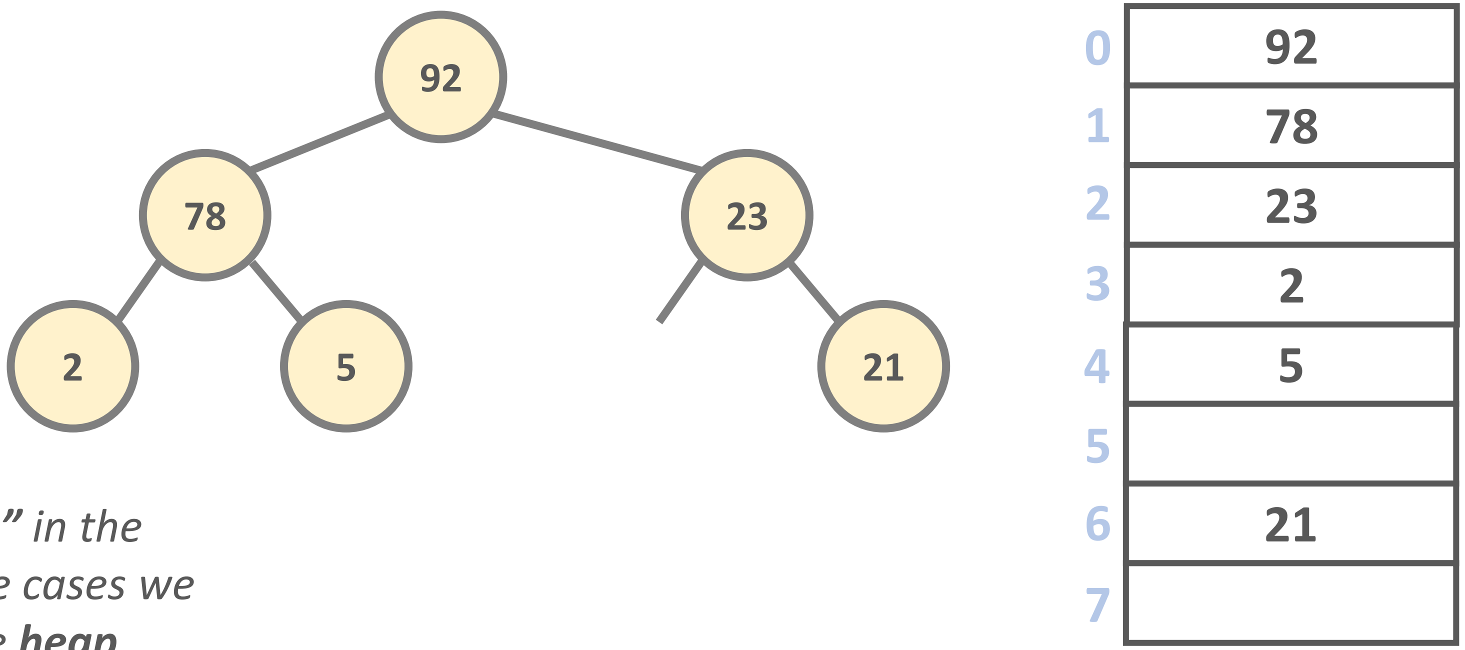
REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Heaps

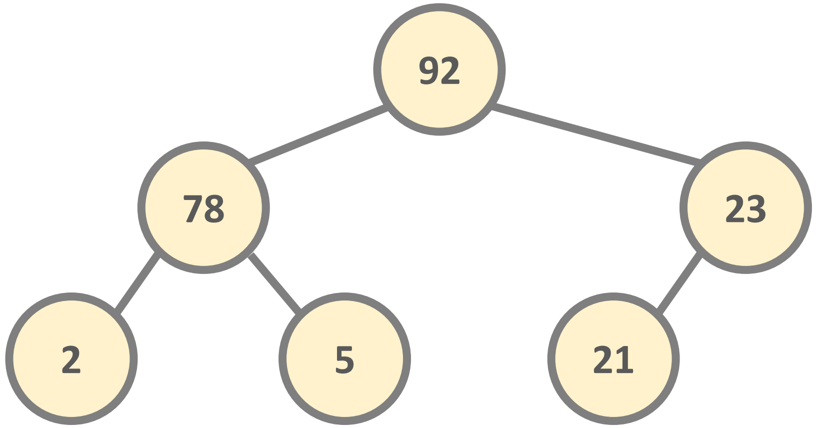
REMOVE(12)



*there can not be a „**hole**" in the data structure and in these cases we use the last item in the **heap***

Heaps

REMOVE(12)

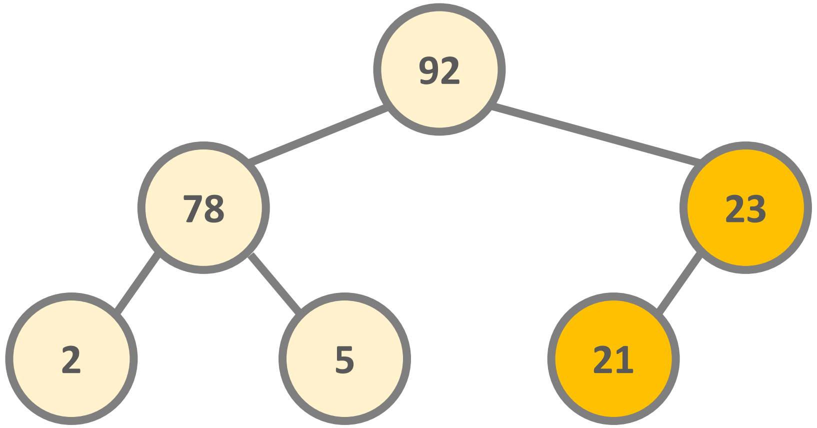


*we have to check **recursively** up to the root node whether the **heap peoperties** are violated or not*

0	92
1	78
2	23
3	2
4	5
5	21
6	
7	

Heaps

REMOVE(12)

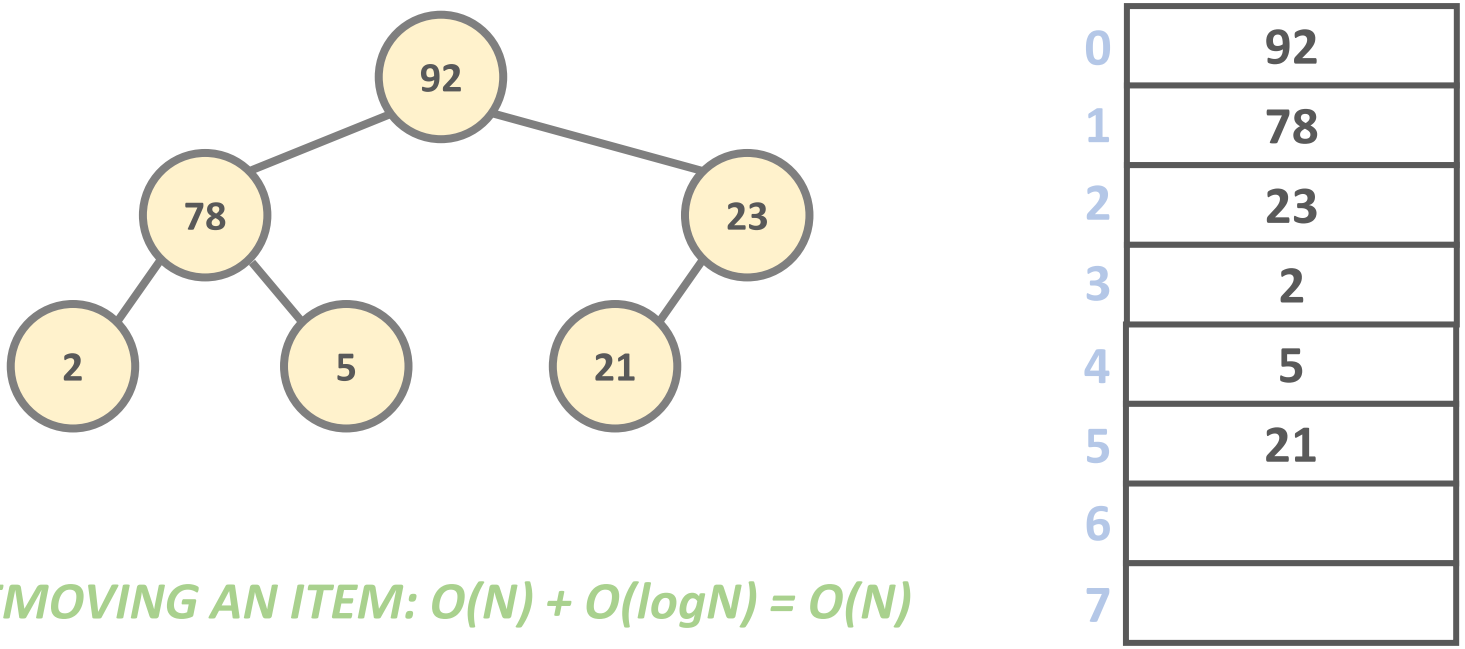


0	92
1	78
2	23
3	2
4	5
5	21
6	
7	

*we have to check **recursively** up to the root node whether the **heap peoperties** are violated or not*

Heaps

REMOVE(12)



REMOVING AN ITEM: O(N) + O(logN) = O(N)

Heapsort

(Algorithms and Data Structures)

Heapsort

- it was constructed back in **1964** by **J. W. J. Williams**
- **heapsort** is a comparison-based sorting algorithm
- uses **heap** data structure rather than a linear-time search to find the maximum
- it is a bit **slower in practice** on most machines than a well-implemented quicksort
- but it has the advantage of a more favorable **O(NlogN)** worst-case running time complexity

Heapsort

- **heapsort** is an in-place algorithm
- **DOES NOT NEED ADDITIONAL MEMORY** – of course we have to store the **N** items
- but it is **not a stable sort** – which means it does not keep the relative order of items with same values
- first we have to construct the heap data structure from the numbers we want to sort
- we have to consider the items one by one in **O(N)** and we have to insert them into the heap in **O(logN)** so the total runing time will be **O(NlogN)**

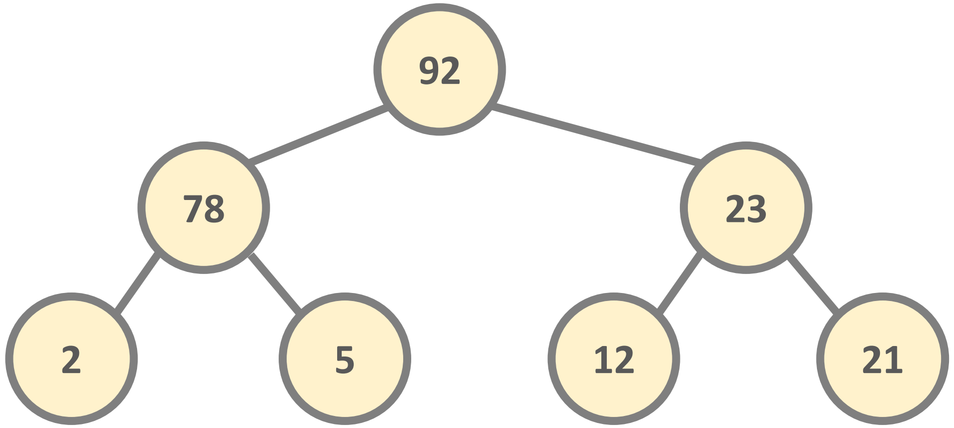
Heapsort

1.) we take the **root node** (include it in the solution set) and swap it with the last item

2.) do **heapify** starting with the root node because the heap properties may be violated

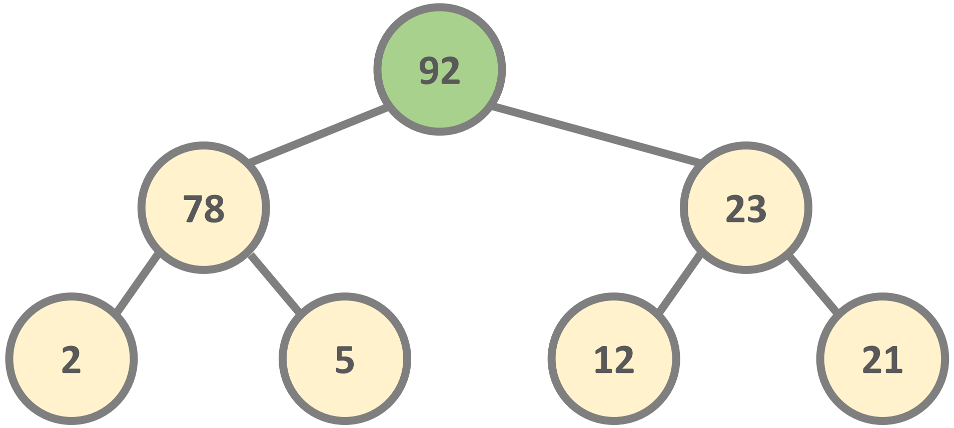
3.) we do it **N** times (for all the items in the data structue)

Heapsort



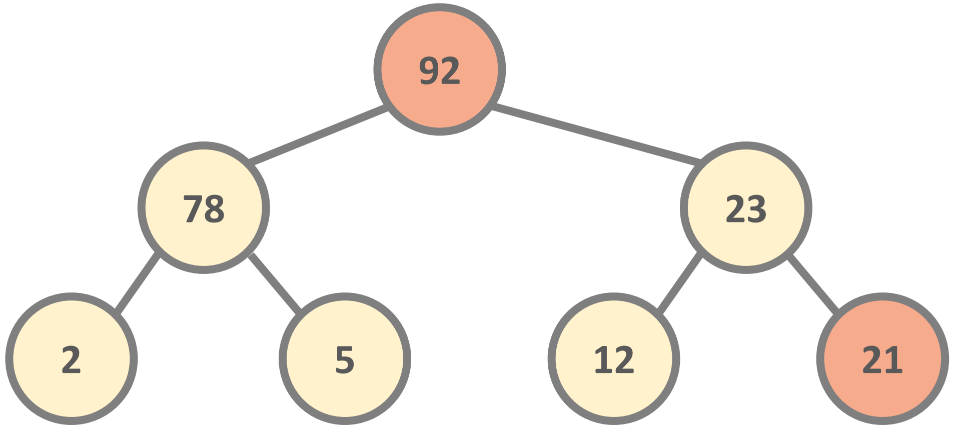
[]

Heapsort



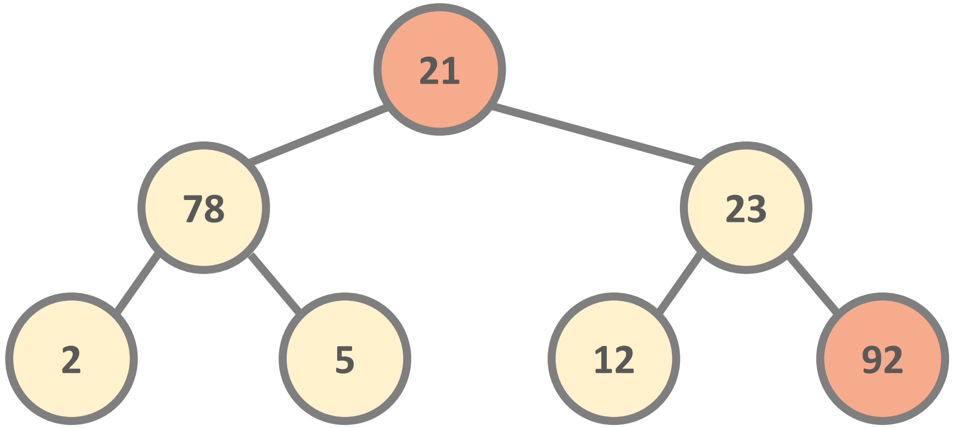
[92]

Heapsort



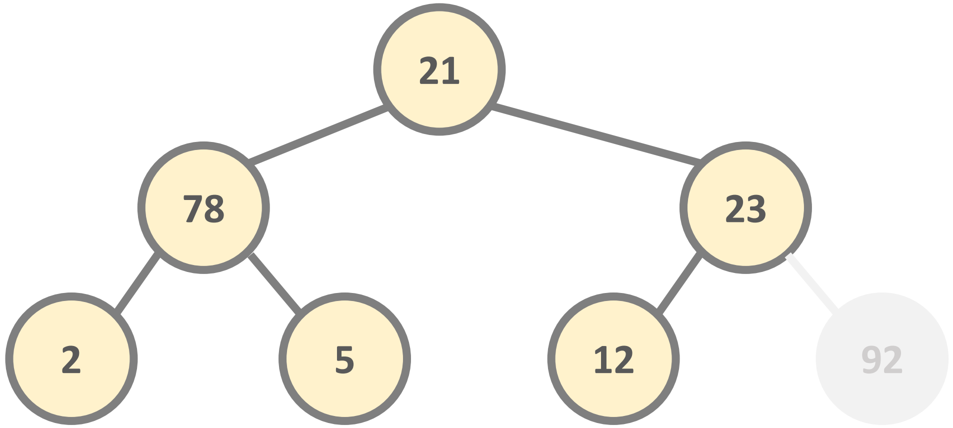
[92]

Heapsort



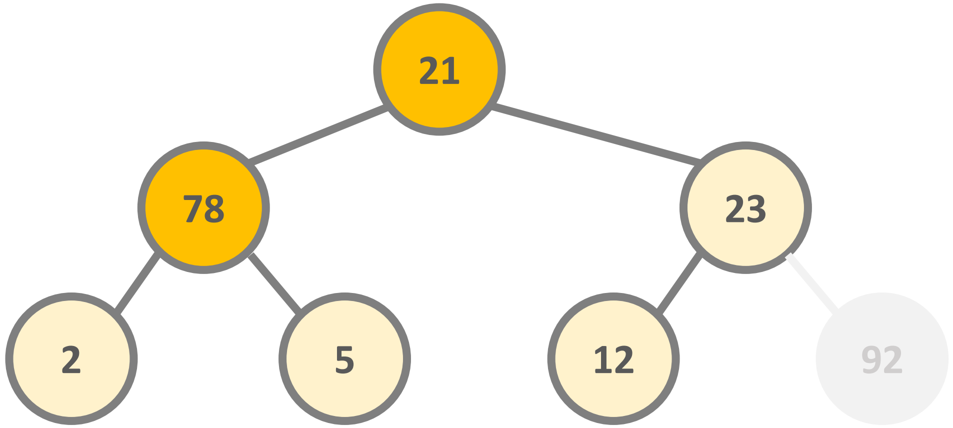
[92]

Heapsort



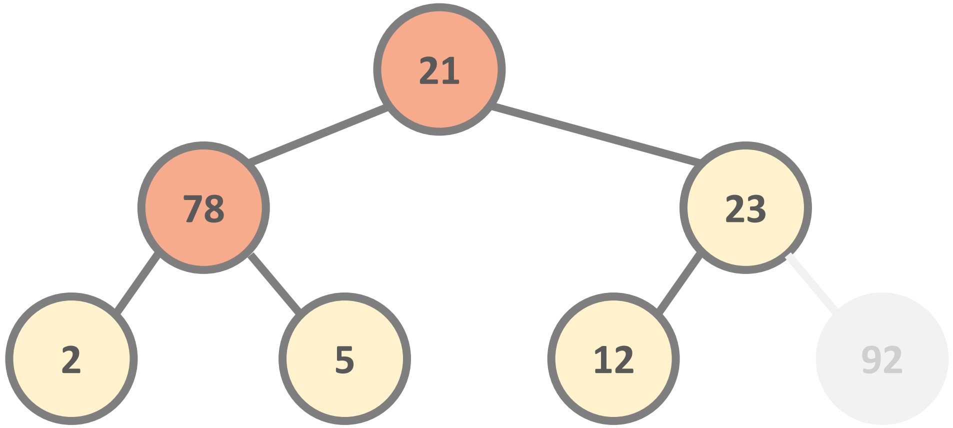
[92]

Heapsort



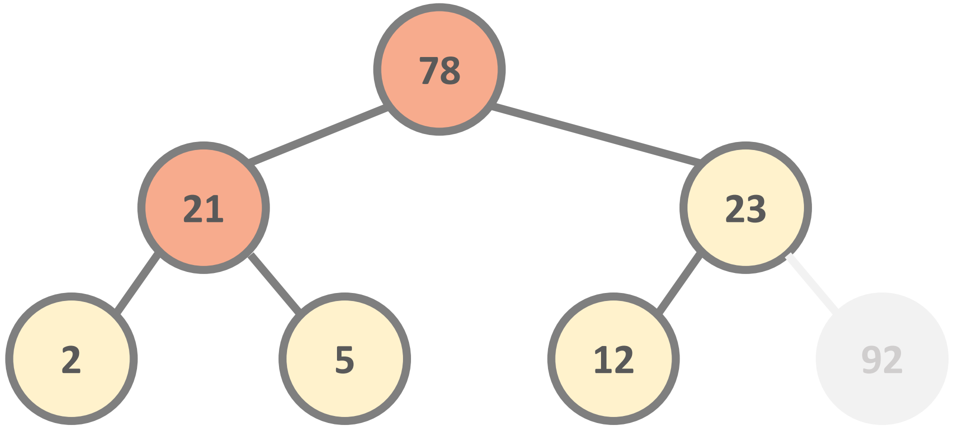
[92]

Heapsort



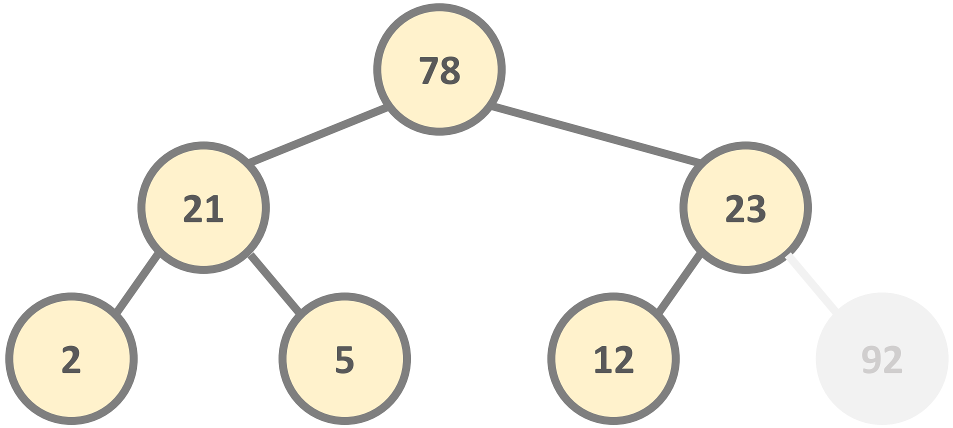
[92]

Heapsort



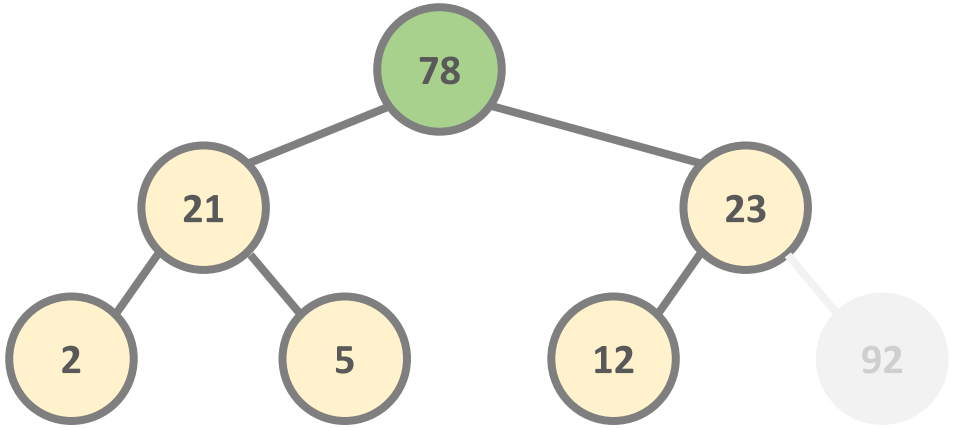
[92]

Heapsort



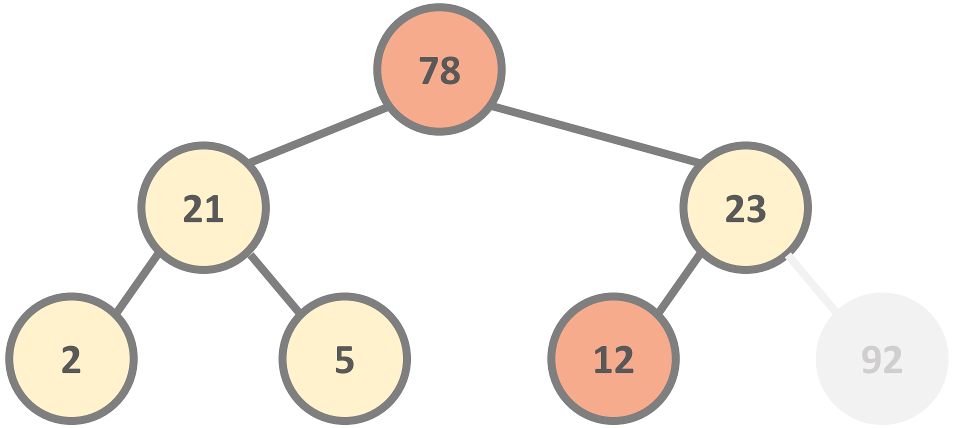
[92]

Heapsort



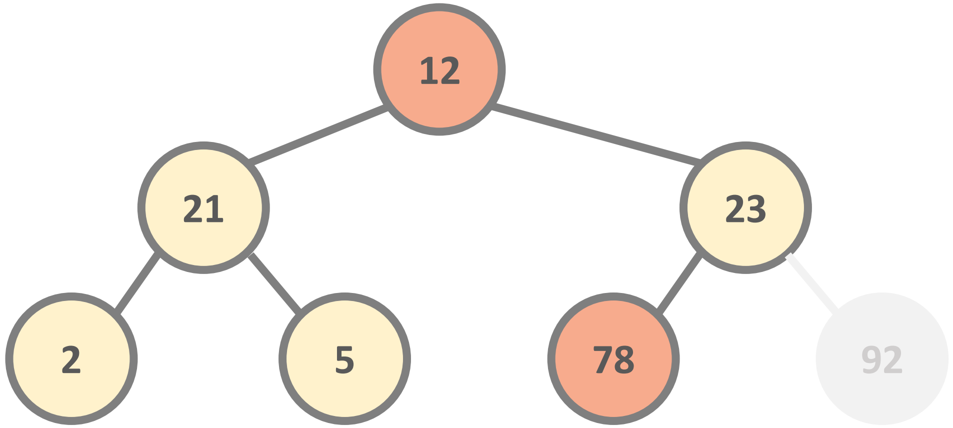
[92, 78]

Heapsort



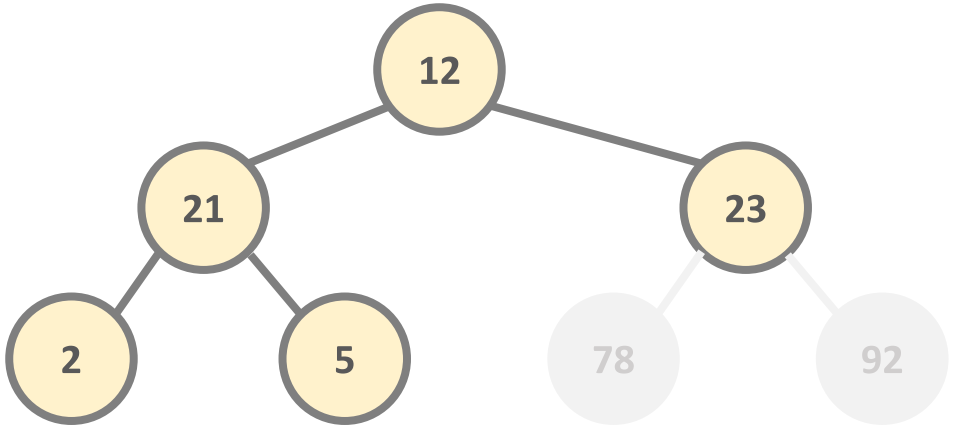
[92, 78]

Heapsort



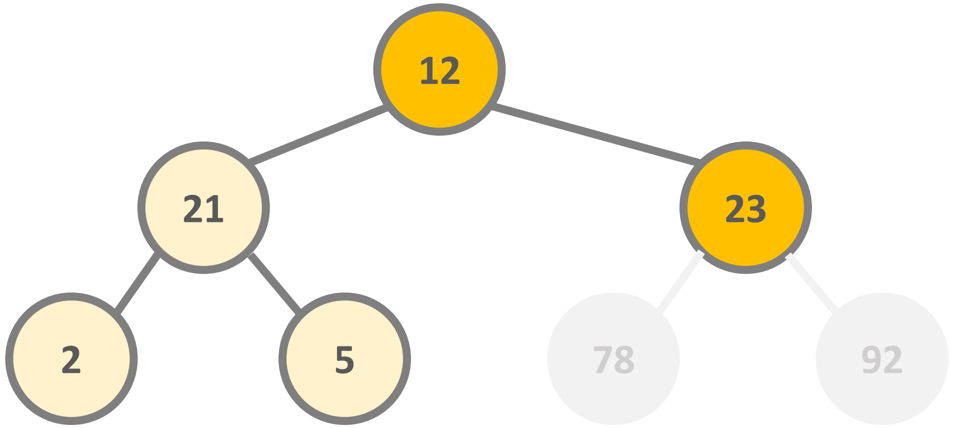
[92, 78]

Heapsort



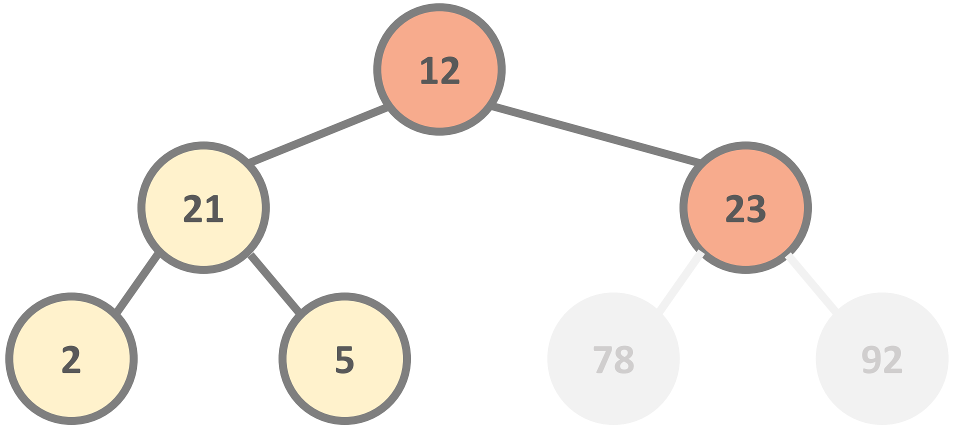
[92, 78]

Heapsort



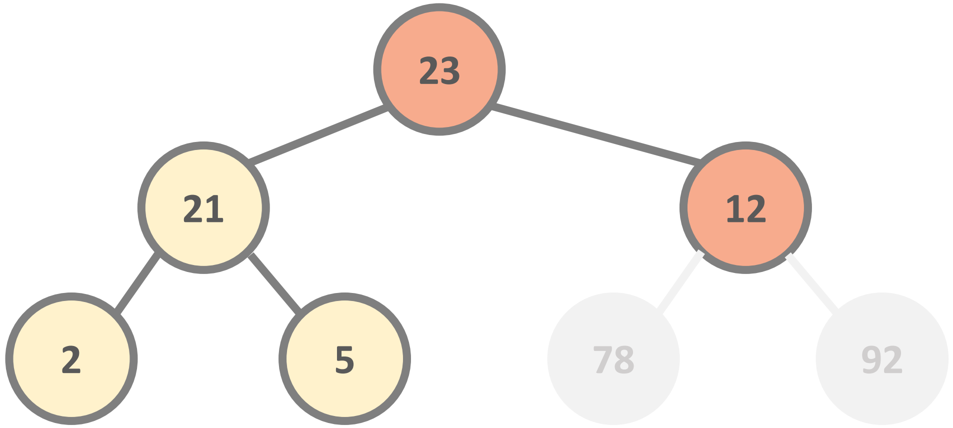
[92, 78]

Heapsort



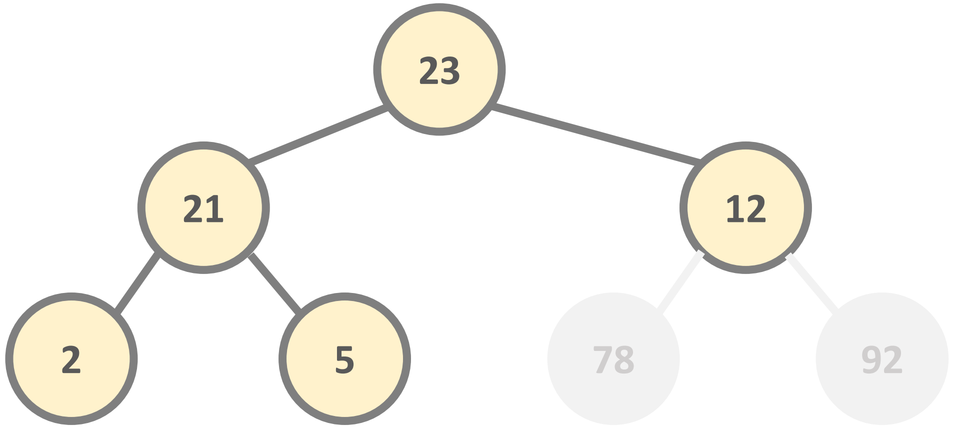
[92, 78]

Heapsort



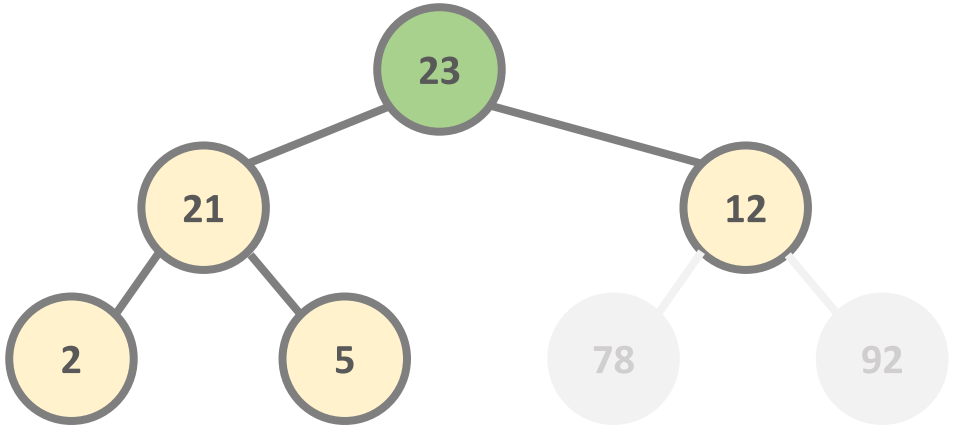
[92, 78]

Heapsort



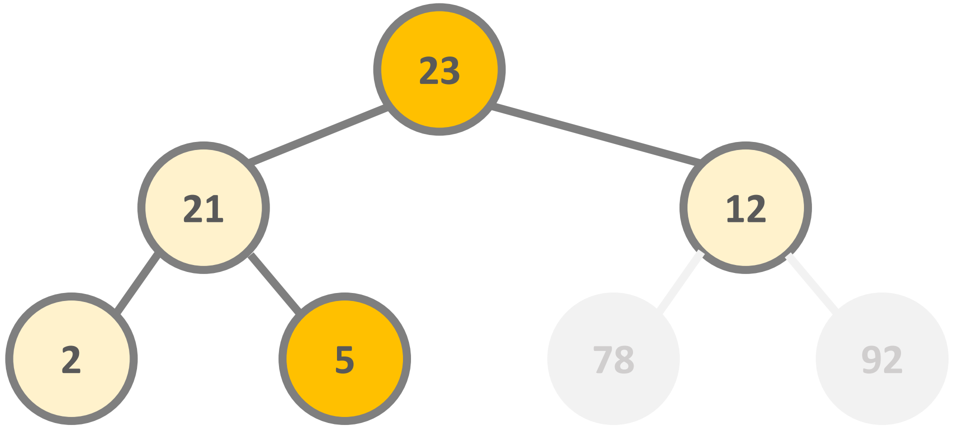
[92, 78]

Heapsort



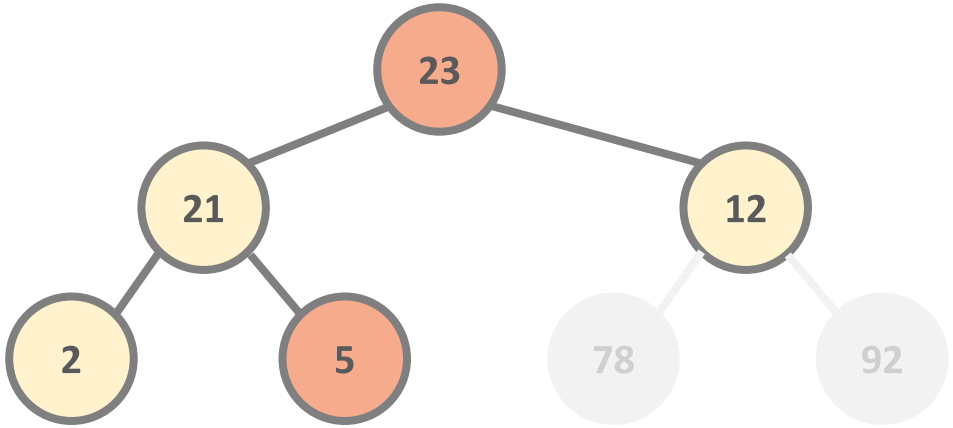
[92, 78, 23]

Heapsort



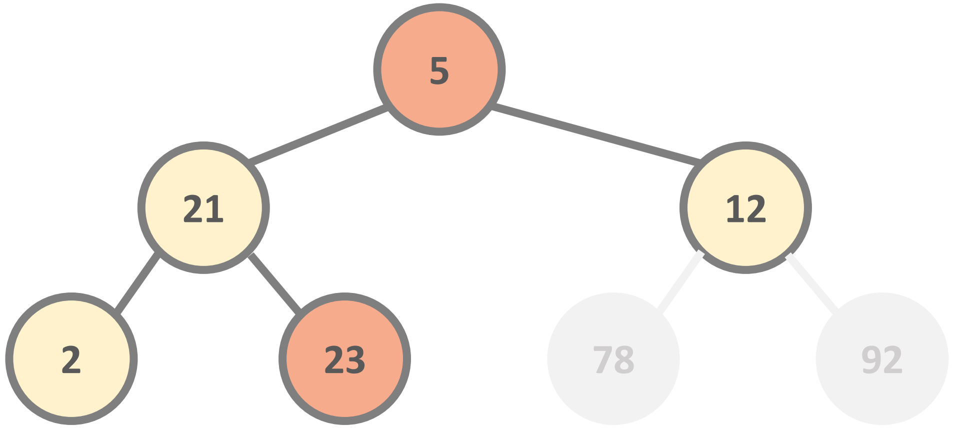
[92, 78, 23]

Heapsort



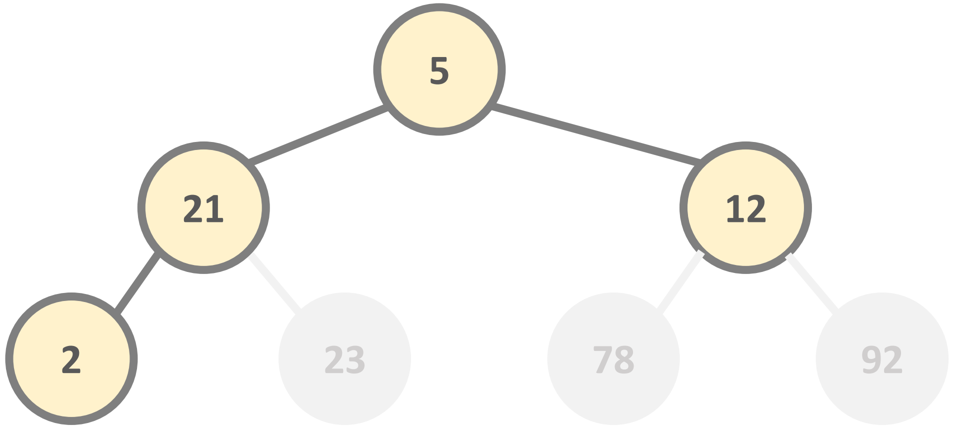
[92, 78, 23]

Heapsort



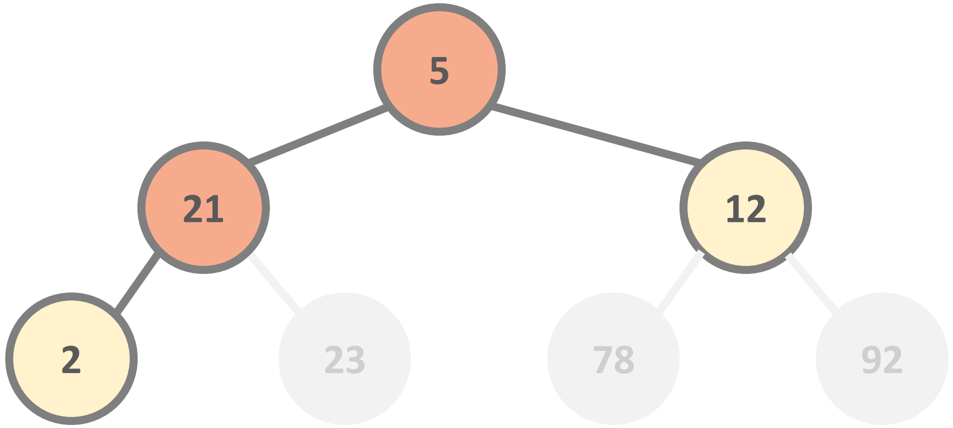
[92, 78, 23]

Heapsort



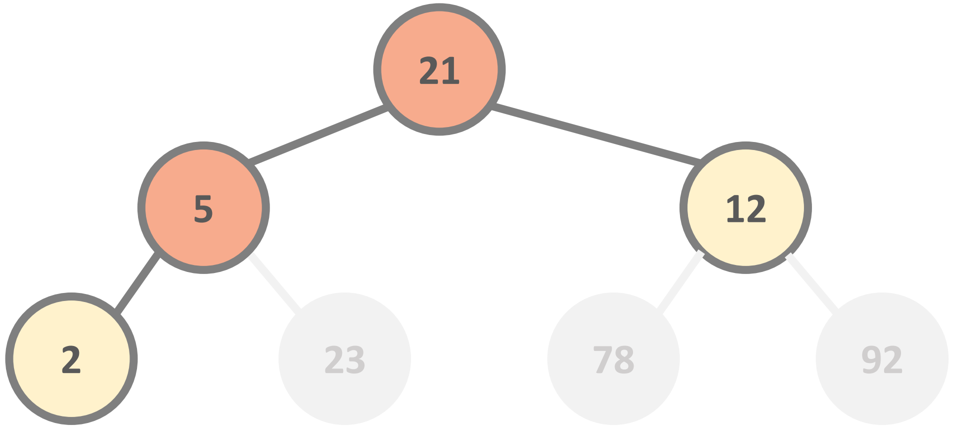
[92, 78, 23]

Heapsort



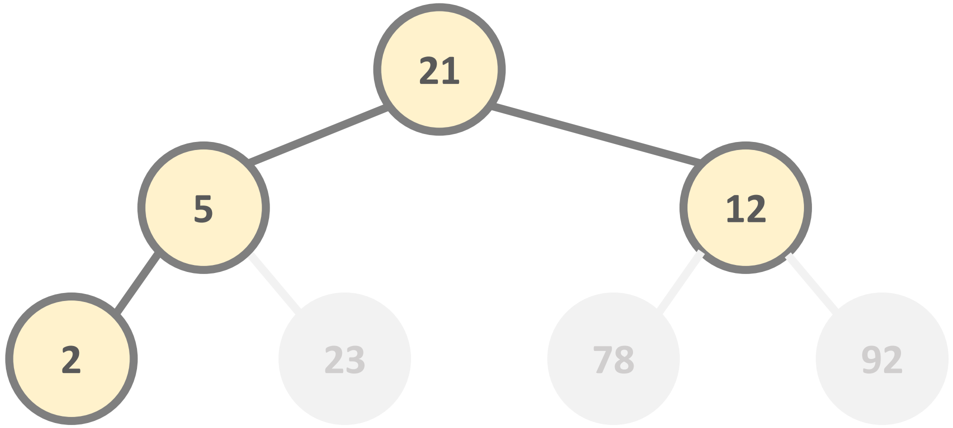
[92, 78, 23]

Heapsort



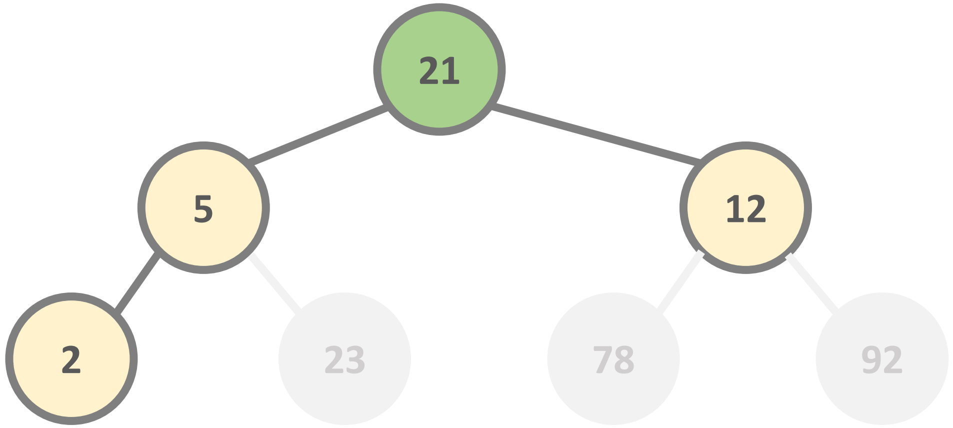
[92, 78, 23]

Heapsort



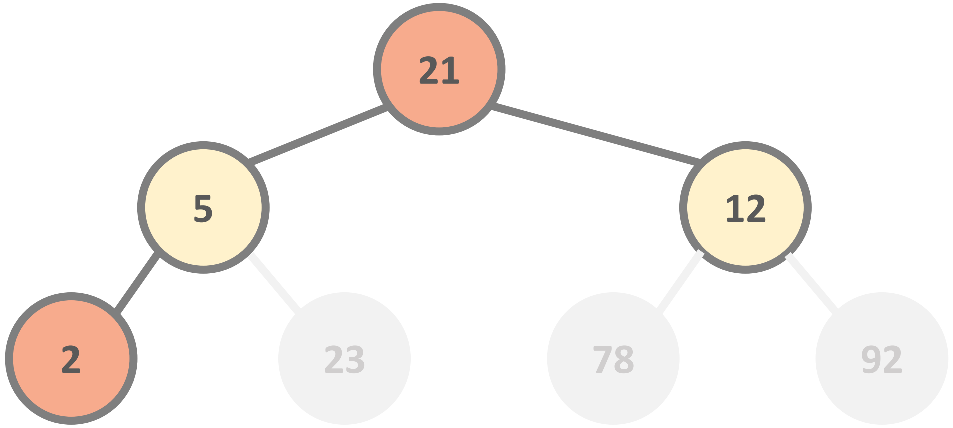
[92, 78, 23]

Heapsort



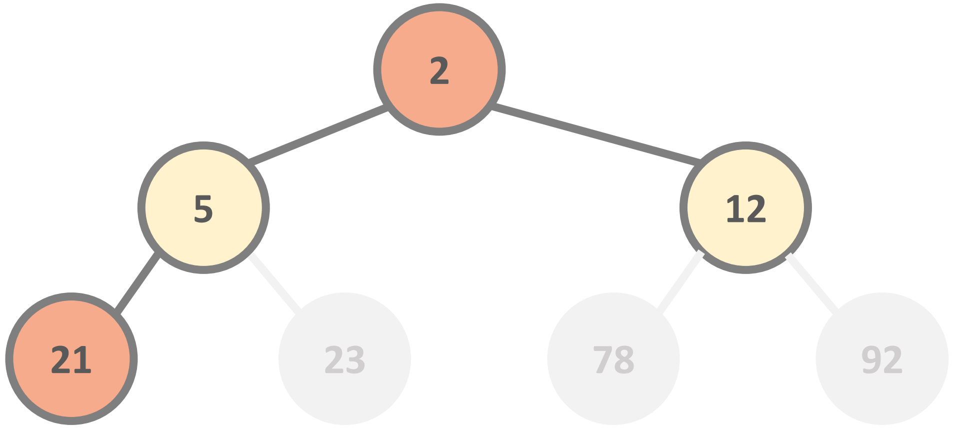
[92, 78, 23, 21]

Heapsort



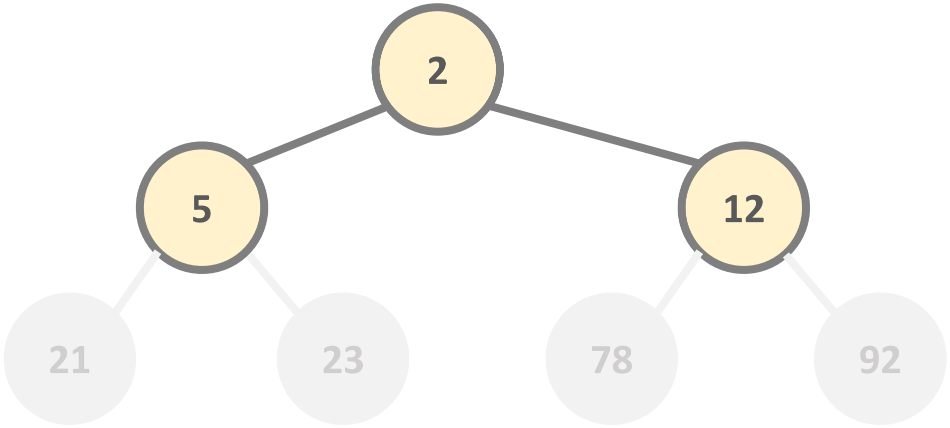
[92, 78, 23, 21]

Heapsort



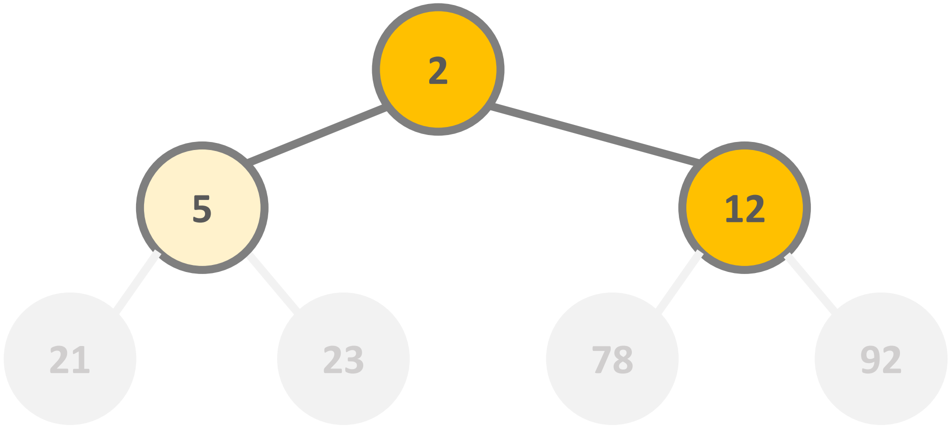
[92, 78, 23, 21]

Heapsort



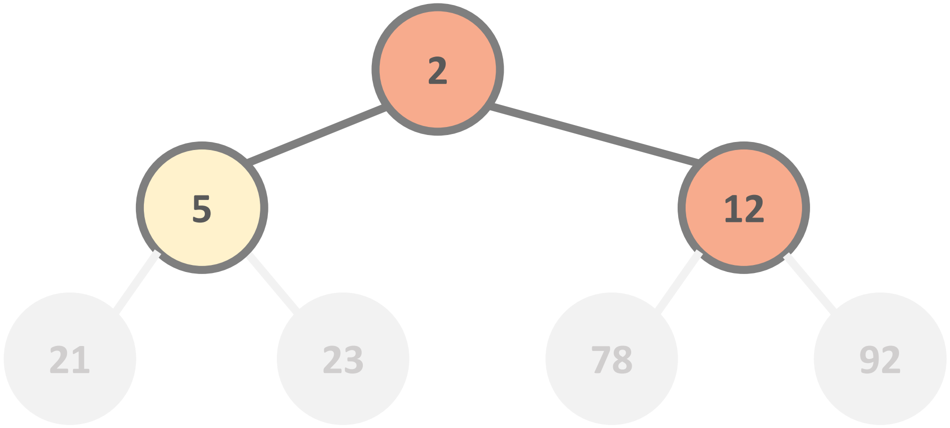
[92, 78, 23, 21]

Heapsort



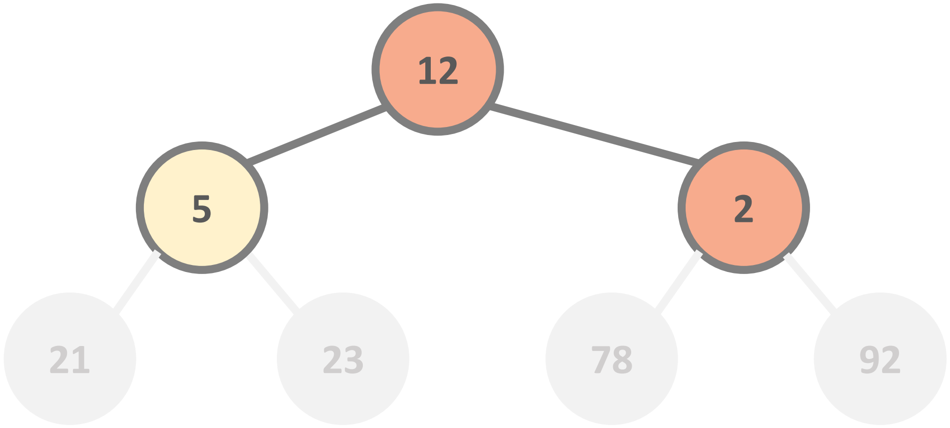
[92, 78, 23, 21]

Heapsort



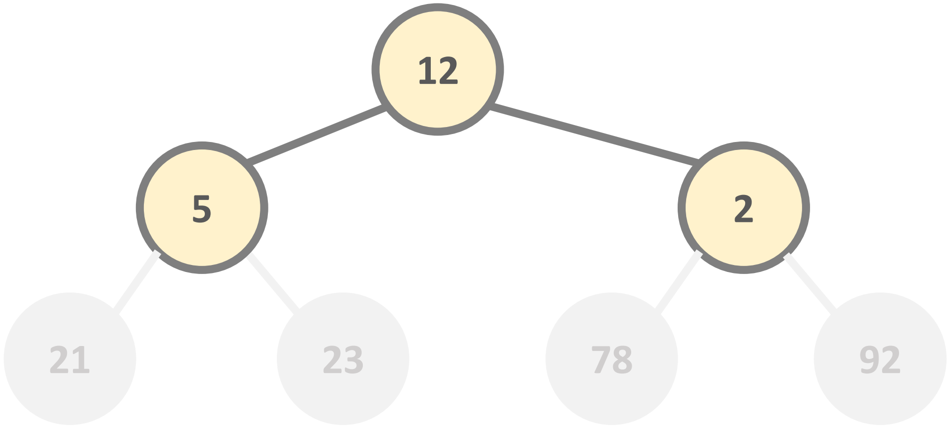
[92, 78, 23, 21]

Heapsort



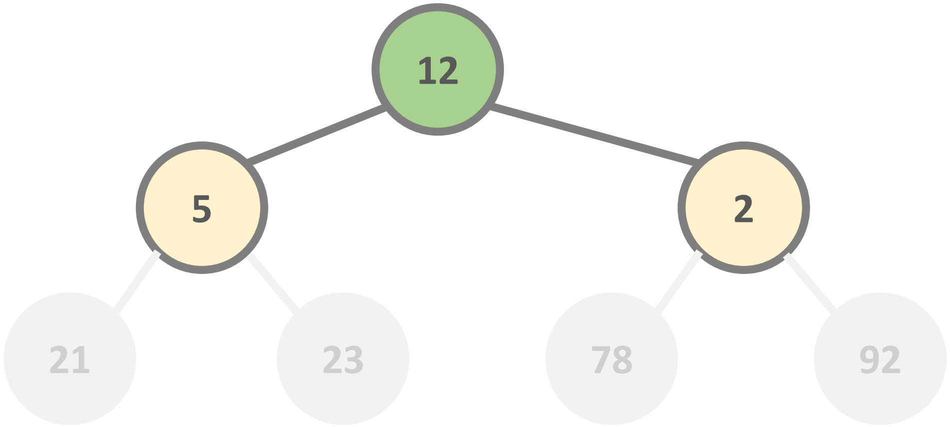
[92, 78, 23, 21]

Heapsort



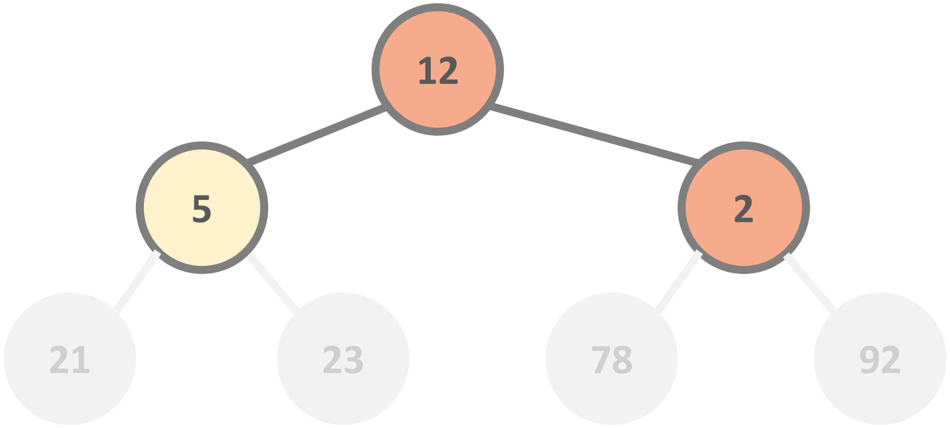
[92, 78, 23, 21]

Heapsort



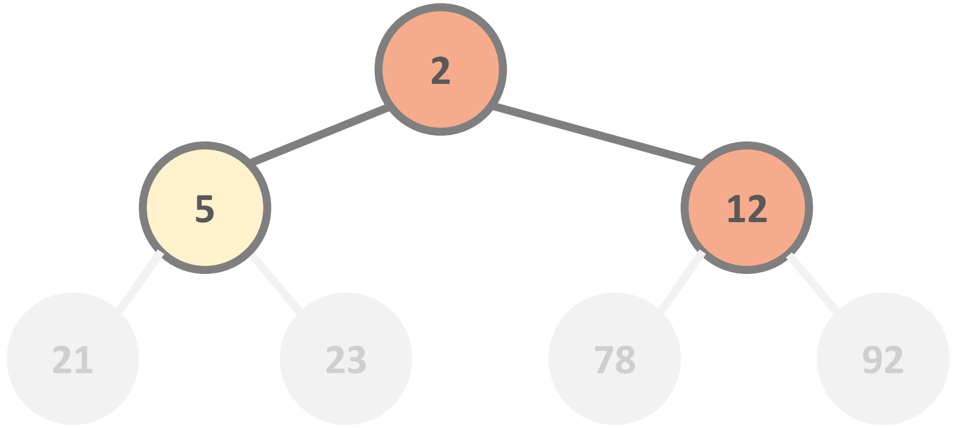
[92, 78, 23, 21, 12]

Heapsort



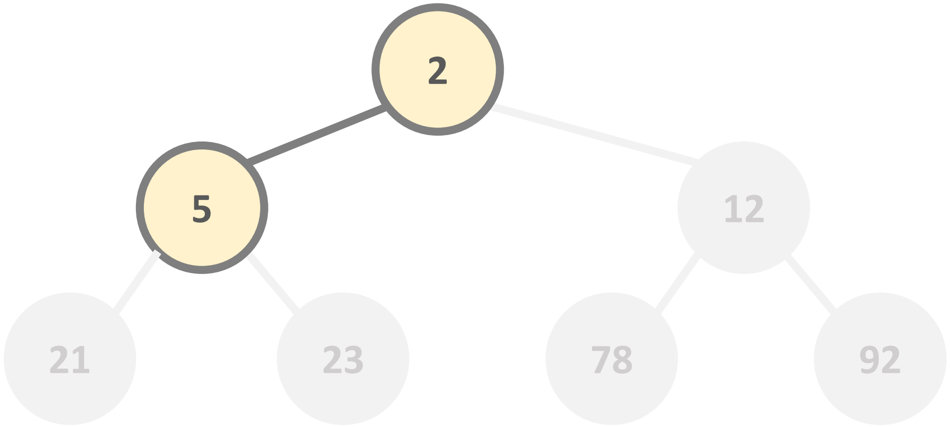
[92, 78, 23, 21, 12]

Heapsort



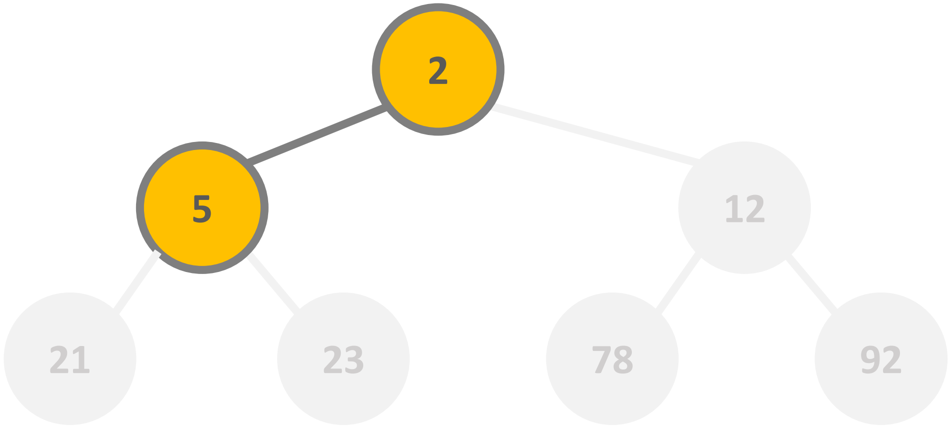
[92, 78, 23, 21, 12]

Heapsort



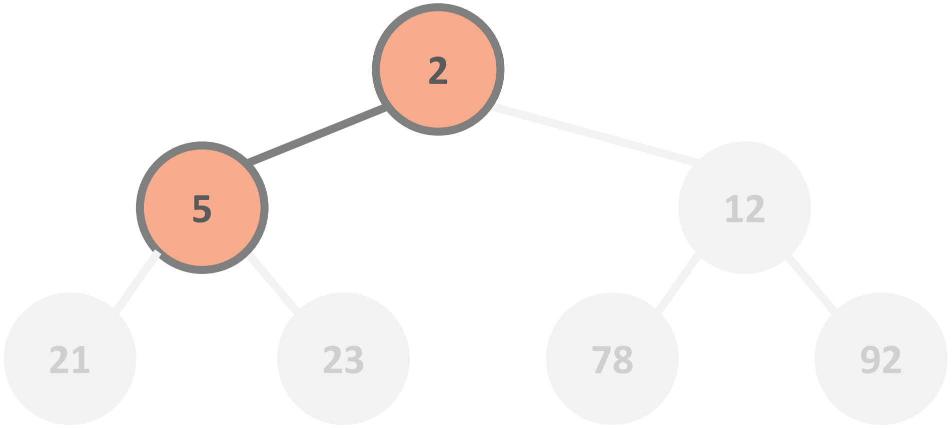
[92, 78, 23, 21, 12]

Heapsort



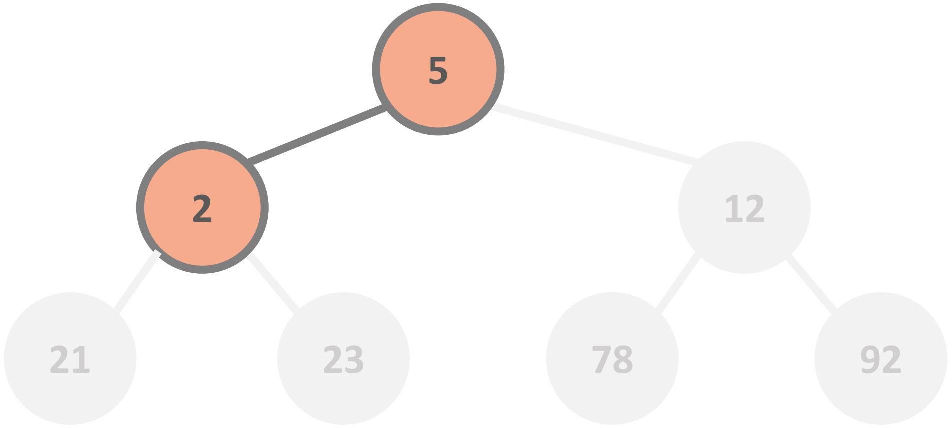
[92, 78, 23, 21, 12]

Heapsort



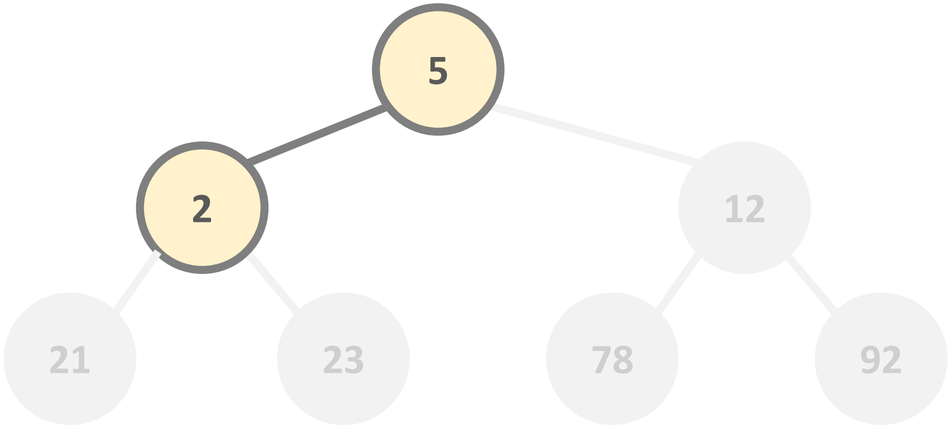
[92, 78, 23, 21, 12]

Heapsort



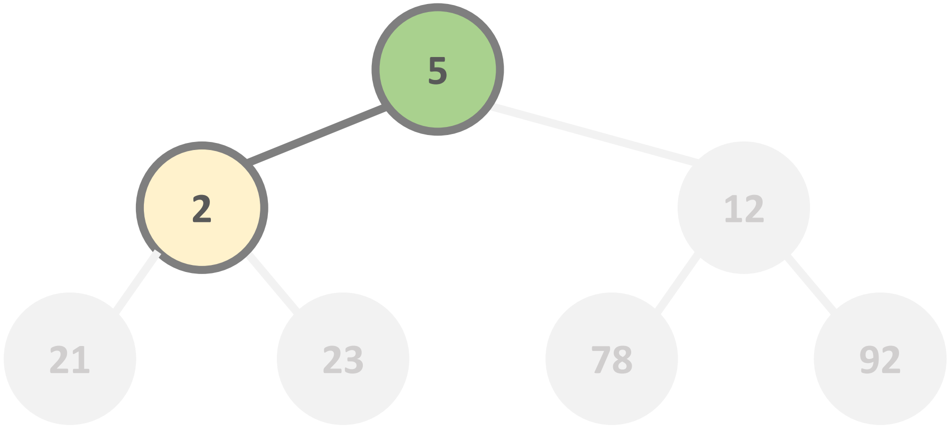
[92, 78, 23, 21, 12]

Heapsort



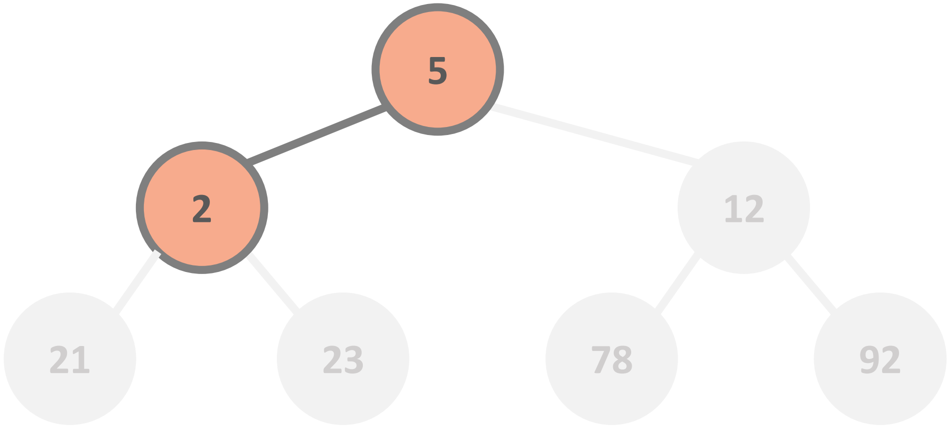
[92, 78, 23, 21, 12]

Heapsort



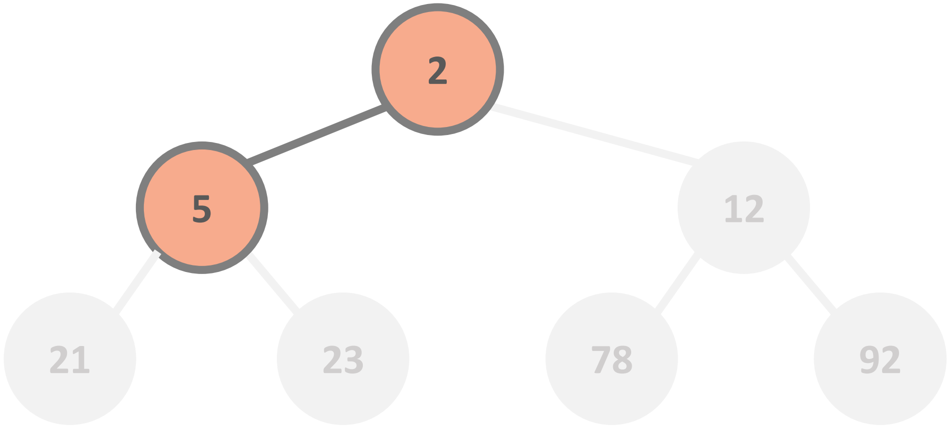
[92, 78, 23, 21, 12, 5]

Heapsort



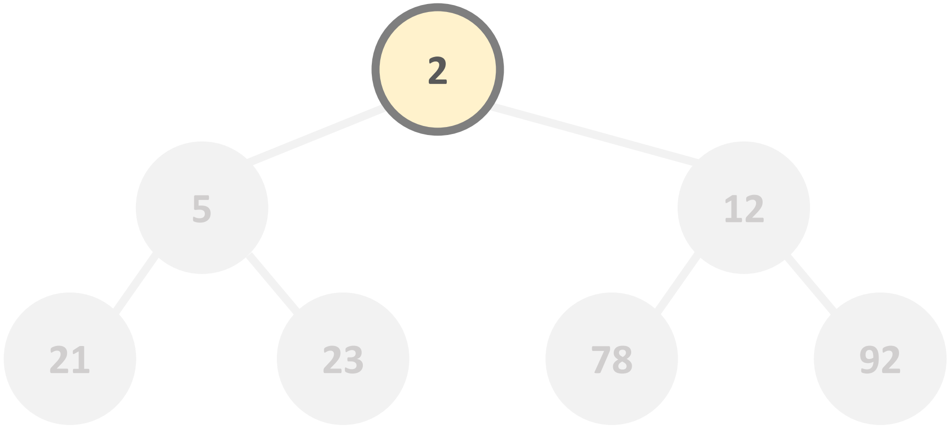
[92, 78, 23, 21, 12, 5]

Heapsort



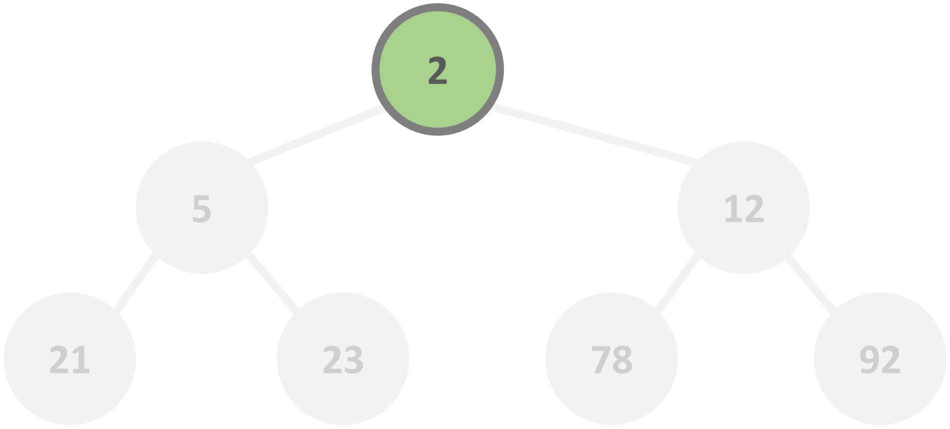
[92, 78, 23, 21, 12, 5]

Heapsort



[92, 78, 23, 21, 12, 5]

Heapsort



[92, 78, 23, 21, 12, 5, 2]

Heapsort

[92, 78, 23, 21, 12, 5, 2]

Advanced Heaps
(Algorithms and Data Structures)

Binomial Heaps

- 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 trees**
- the **O(logN)** logarithmic insertion time complexity can be reduced to **O(1)** constant time complexity with the help of binomial heaps

Fibonacci Heaps

- **Fibonacci heaps** are 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** – the number of children are usually kept low
- we can achieve **O(1)** running time for insertion operation instead of **O(logN)** logarithmic running time
- every node has degree at most **O(logN)** and the size of a subtree rooted in a node of degree **k** is at least $\mathbf{F_{k+2}}$ where $\mathbf{F_k}$ is the **k-**th Fibonacci number

Heaps Running Time

	BINARY	BINOMIAL	FIBONACCI
find min	O(1)	O(1)	O(1)
delete min	O(logN)	O(logN)	O(logN)
insertion	O(logN)	O(1)	**O(1)**
decrease key	O(logN)	O(logN)	**O(1)**
merge	-	O(logN)	**O(1)**

Fibonacci-heaps *are hard to implement*
but they are ***extremely powerful***