Heaps

CS223: Data Structures

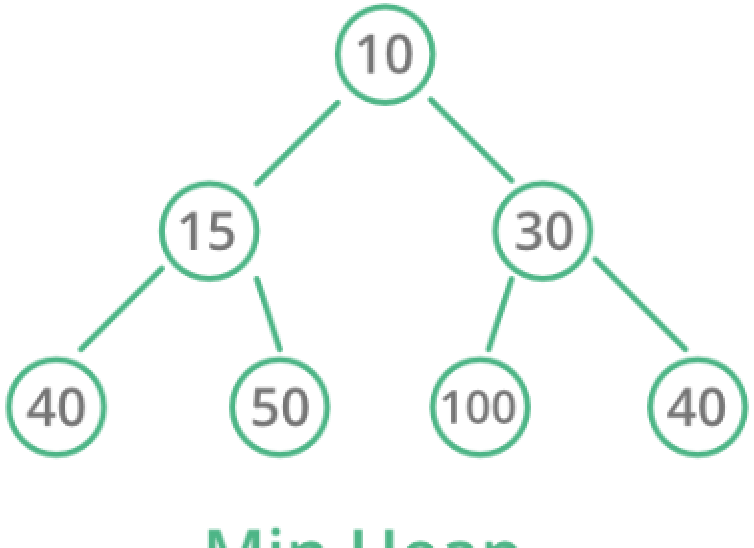
Binary Heaps

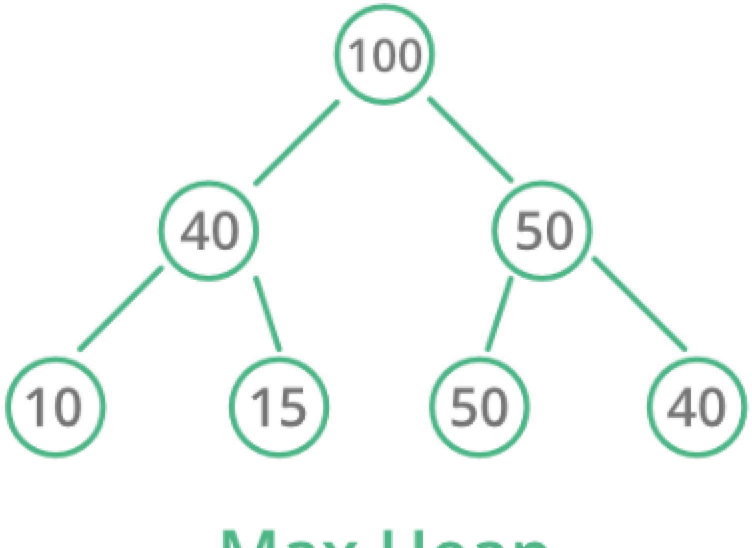
A binary heap is a special tree-based data structure in which the tree is a complete binary tree.

- Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right

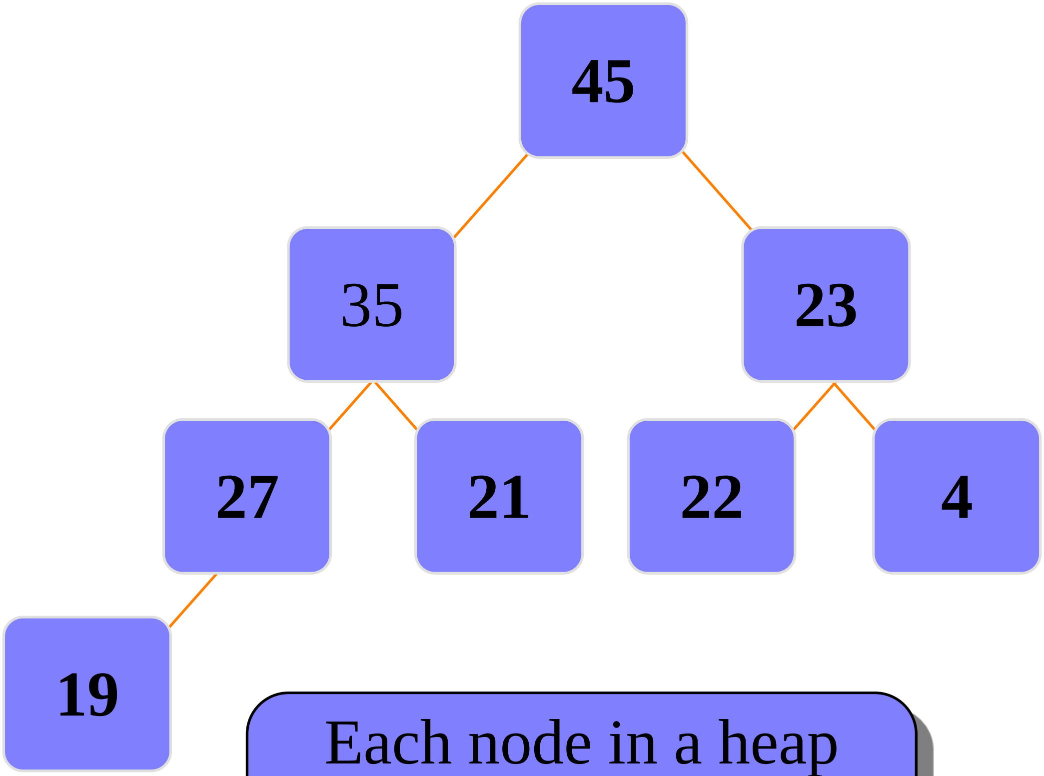
Generally, Heaps can be of two types:

- **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



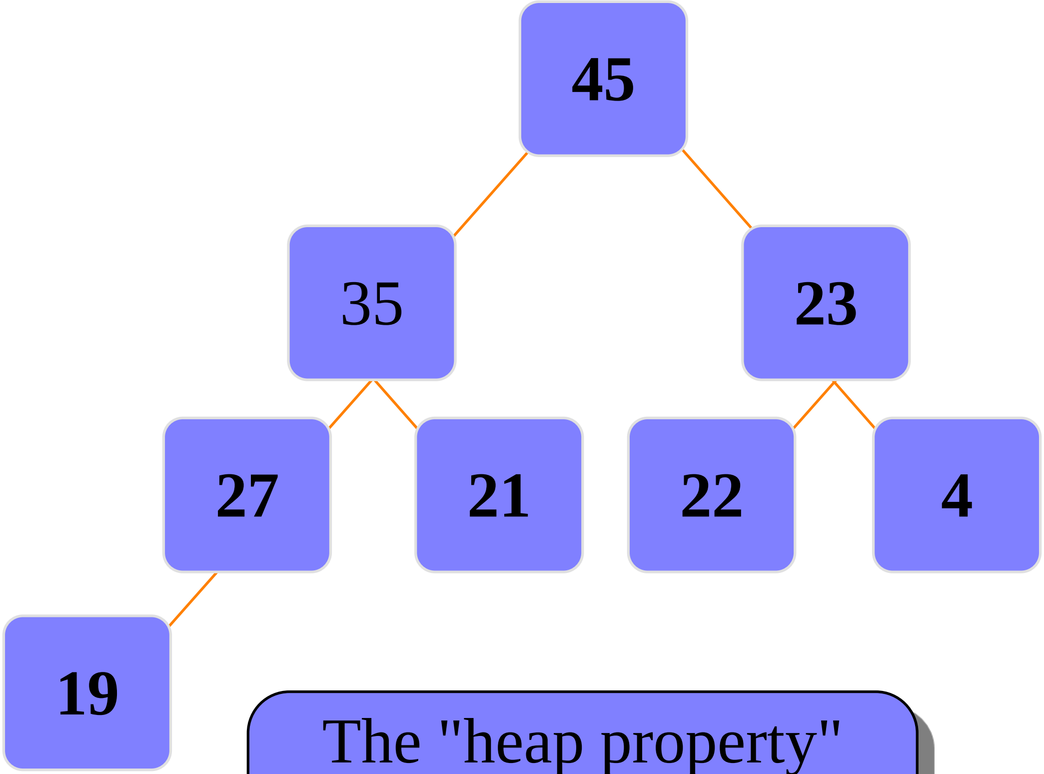


Heaps



Each node in a heap contains a key that can be compared to other nodes' keys.

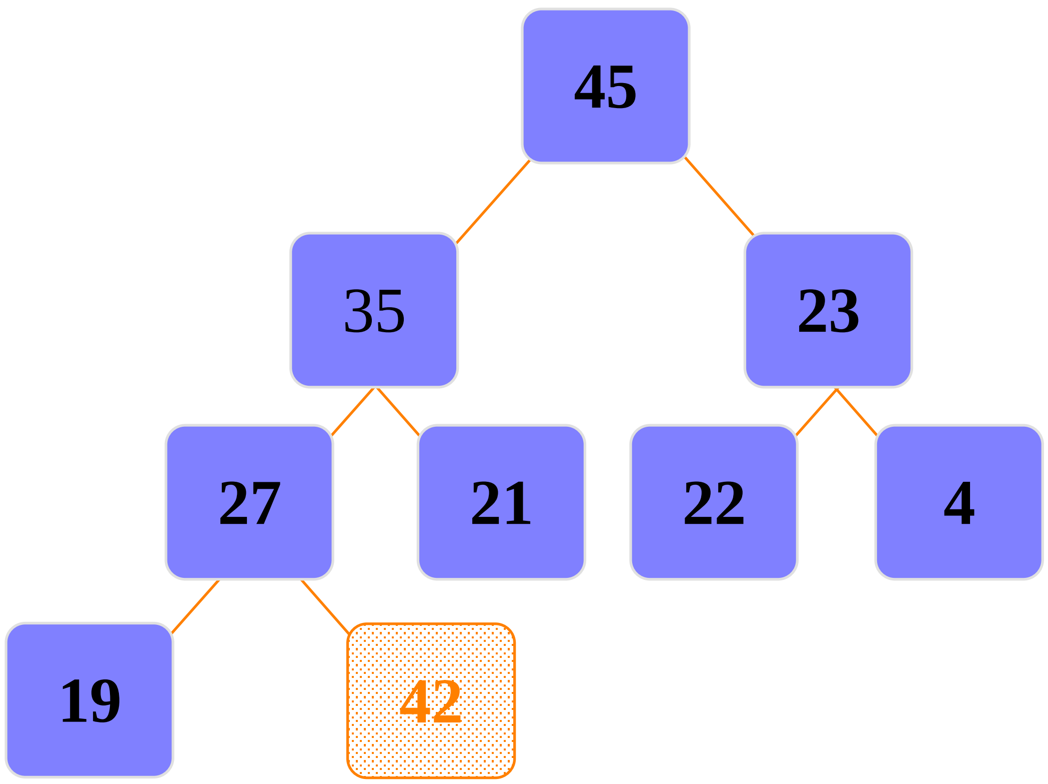
Heaps



The "heap property" requires that each node's key is >= the keys of its children

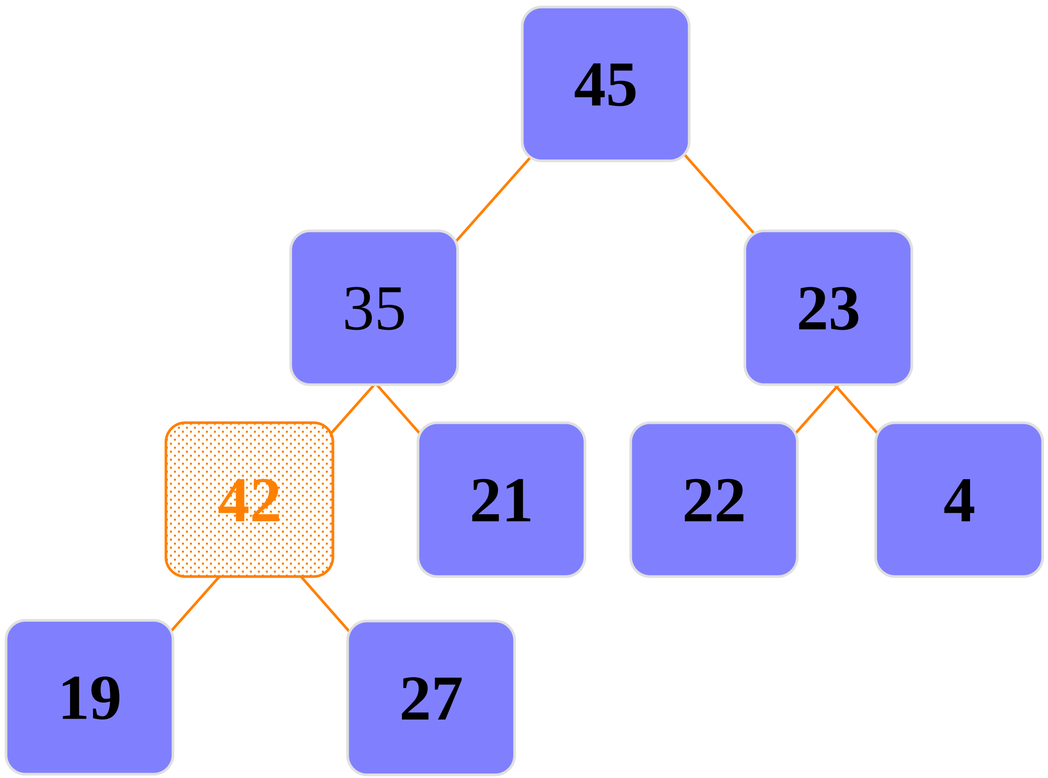
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



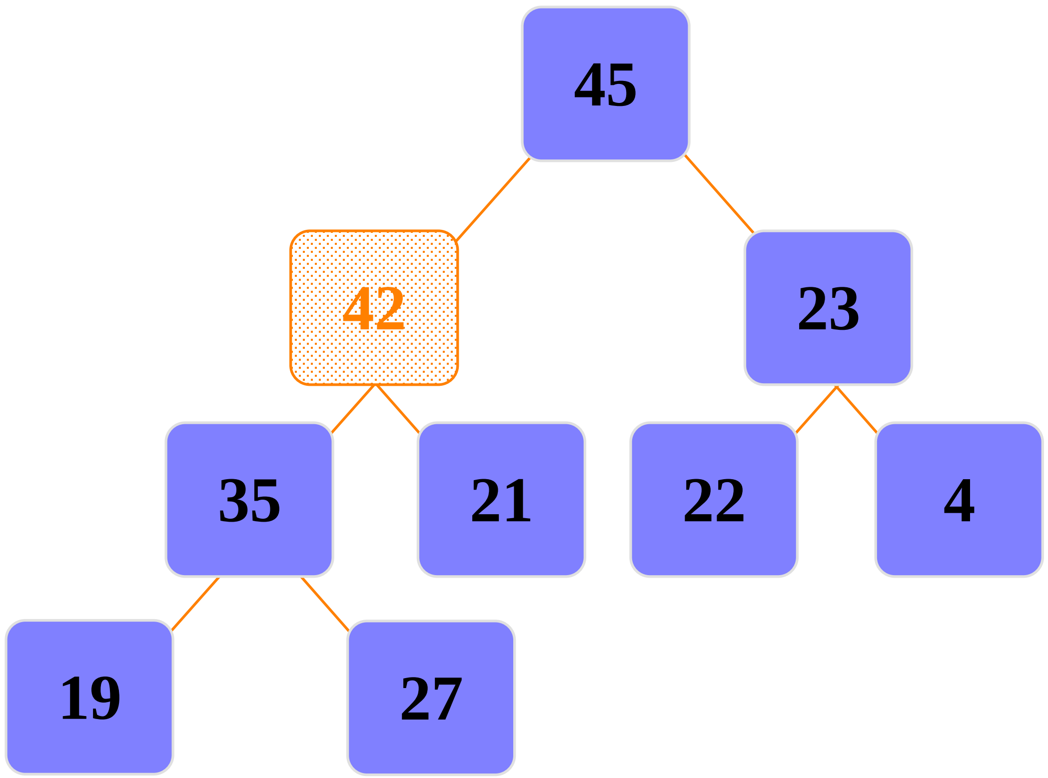
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



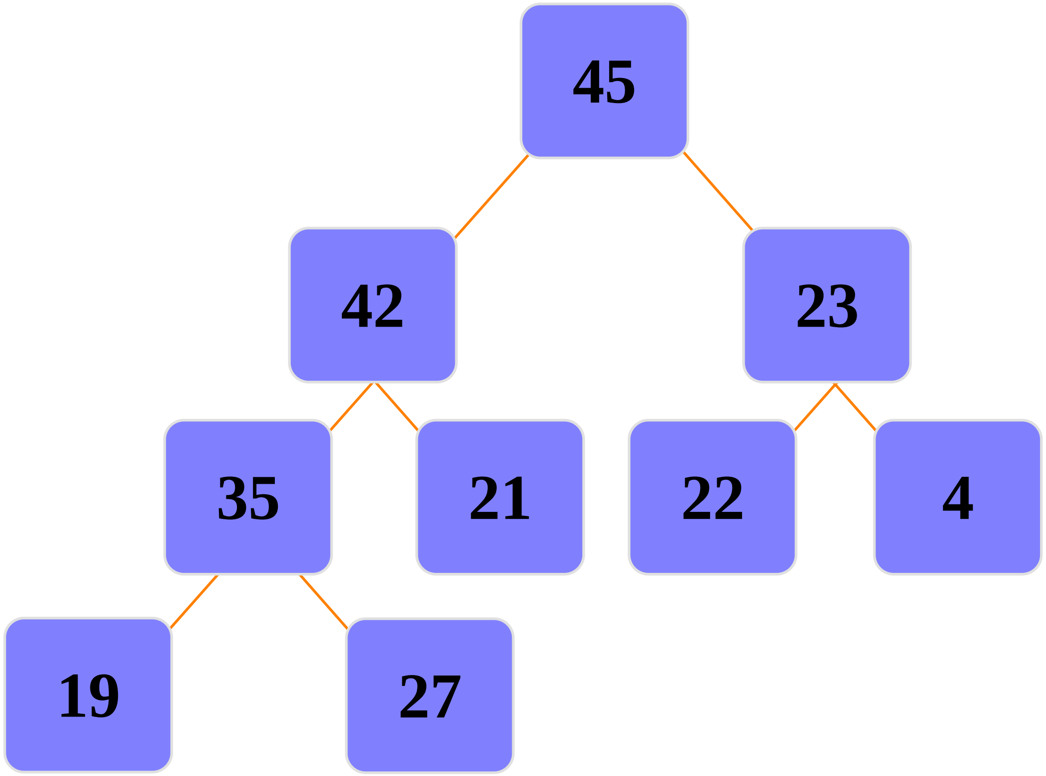
Adding a Node to a Heap

- ❑ Put the new node in the next available spot.
- ❑ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



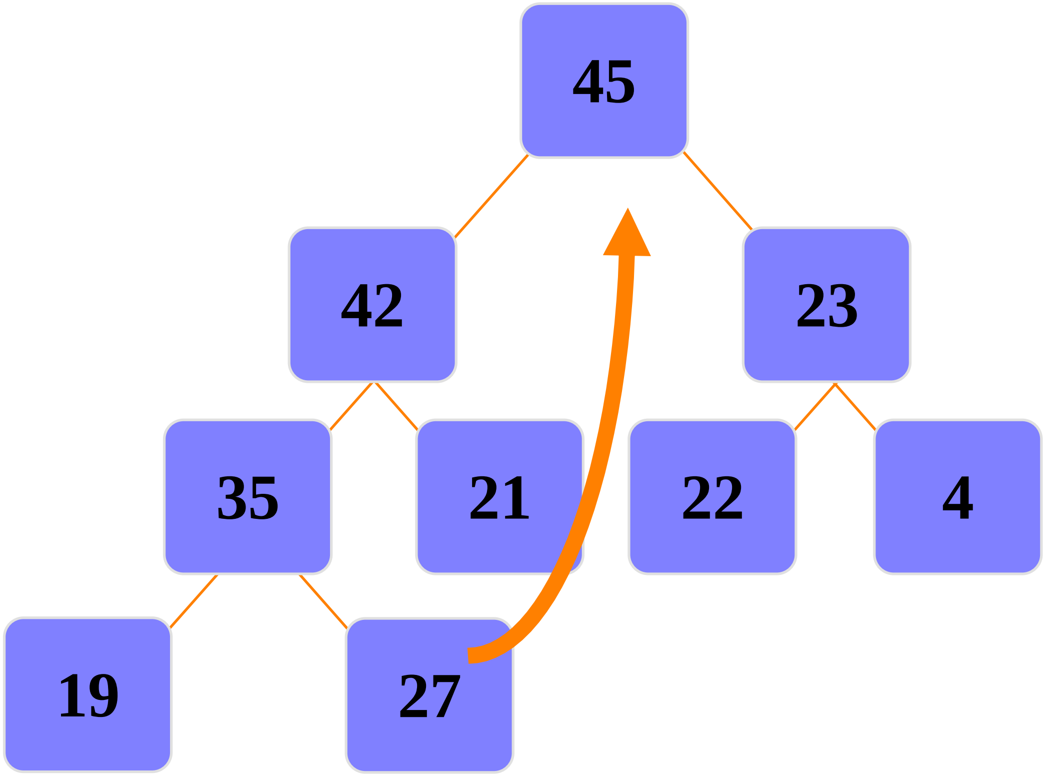
Adding a Node to a Heap

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



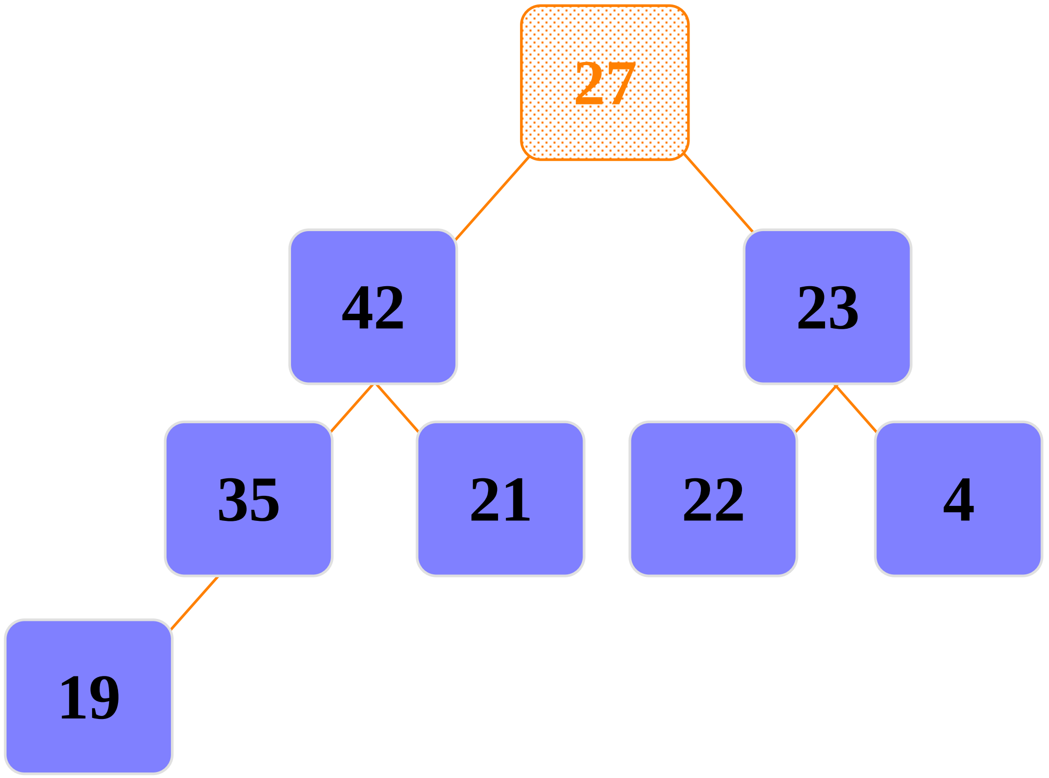
Removing the Top of a Heap

- ❑extractMin() or extractMax()
- ❑Move the last node onto the root.



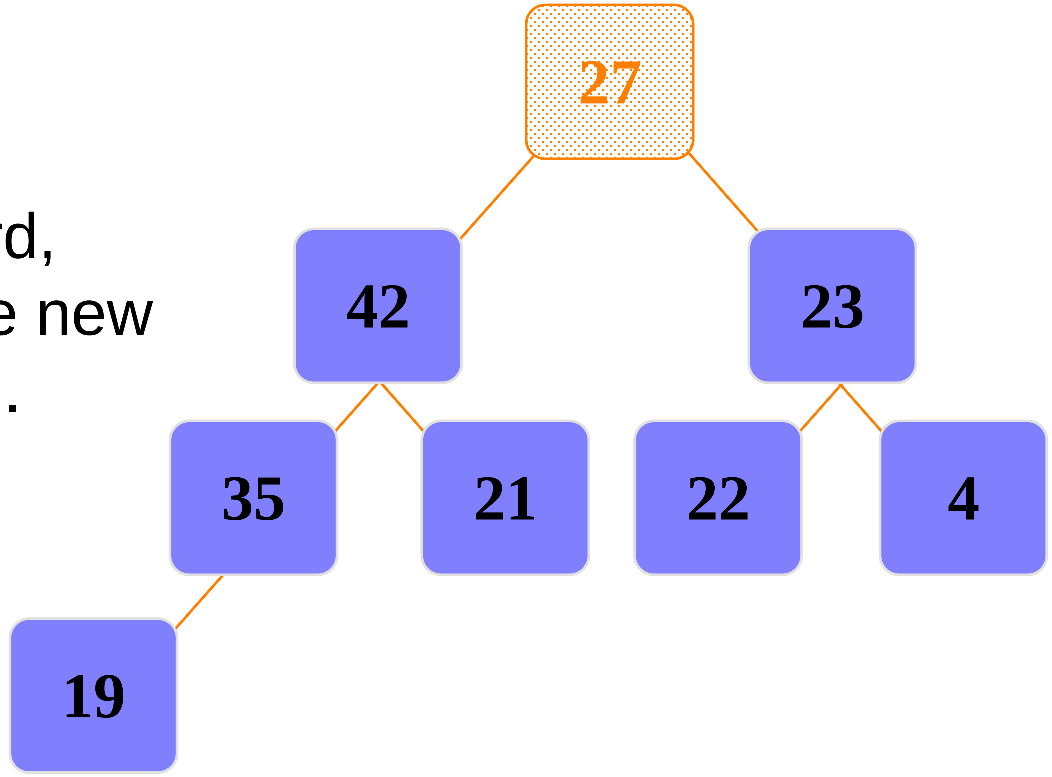
Removing the Top of a Heap

- Move the last node onto the root.



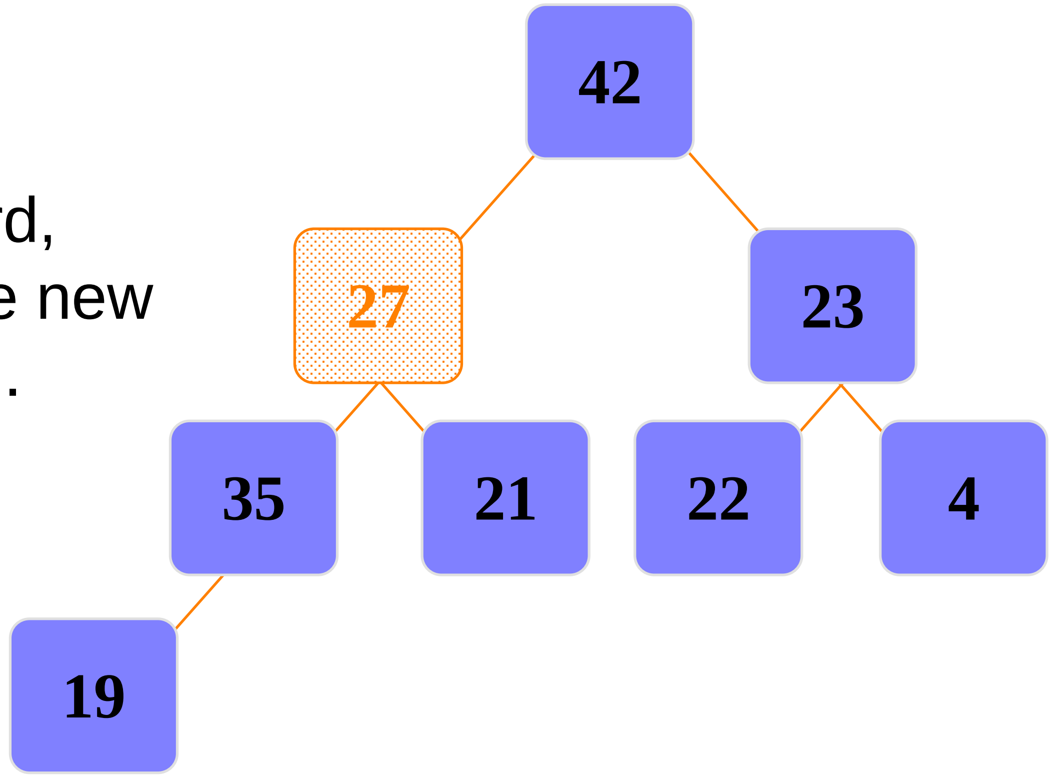
Removing the Top of a Heap

- ❑ Move the last node onto the root.
- ❑ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



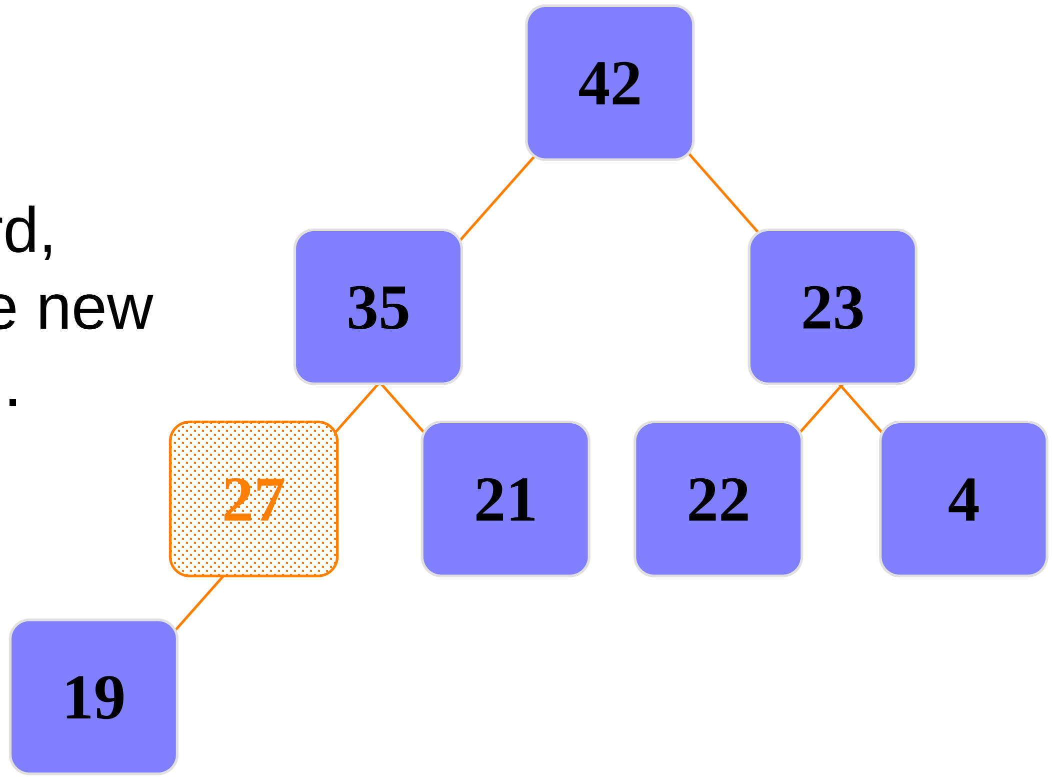
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



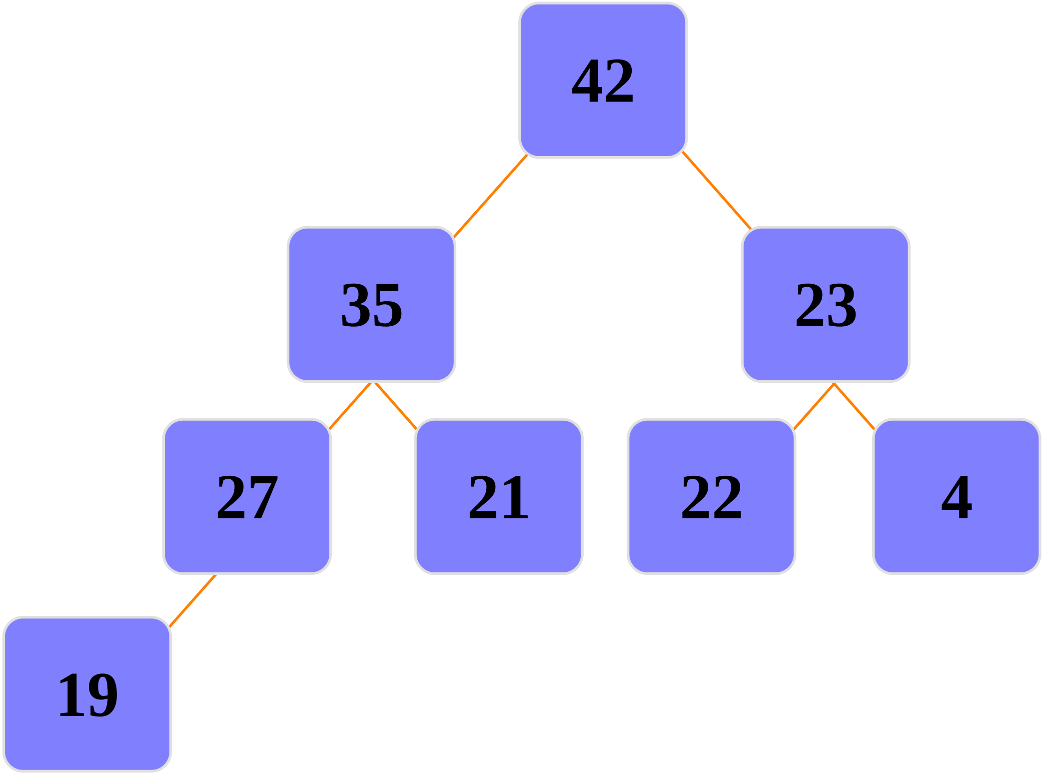
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



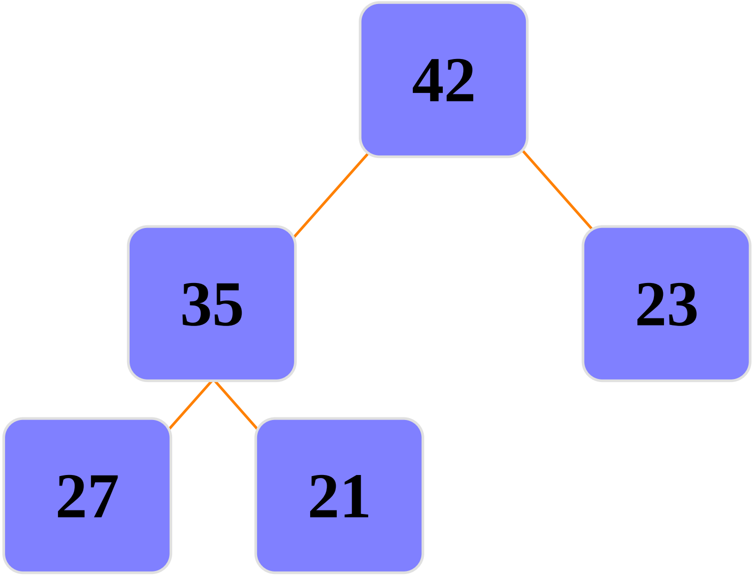
Removing the Top of a Heap

- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.

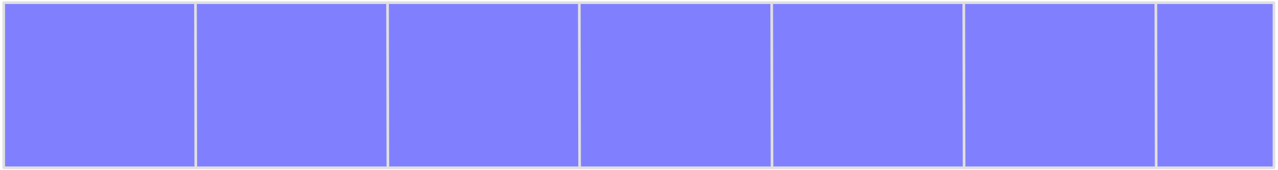


Implementing a Heap

- We will store the data from the nodes in a partially-filled array.

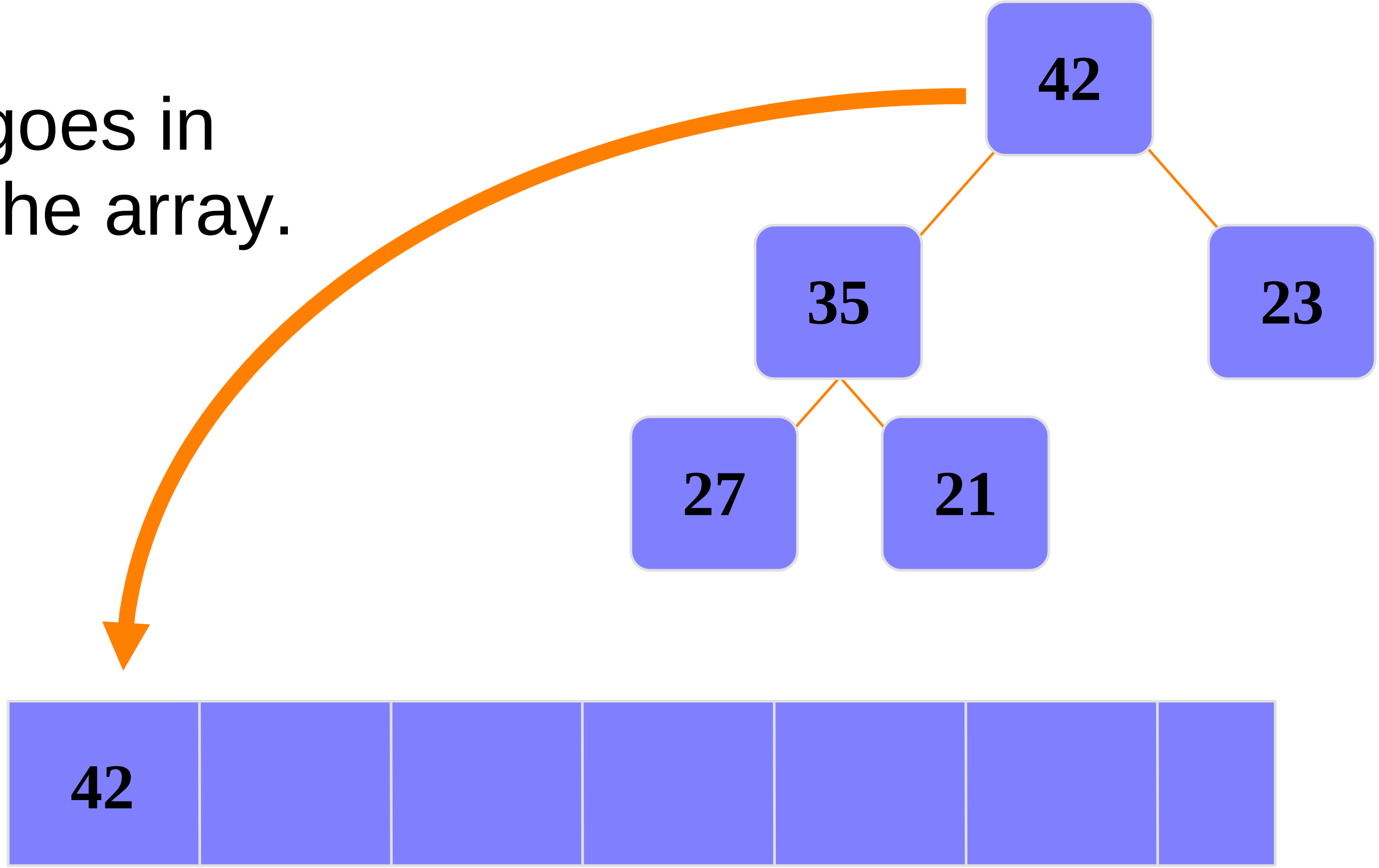


An array of data



Implementing a Heap

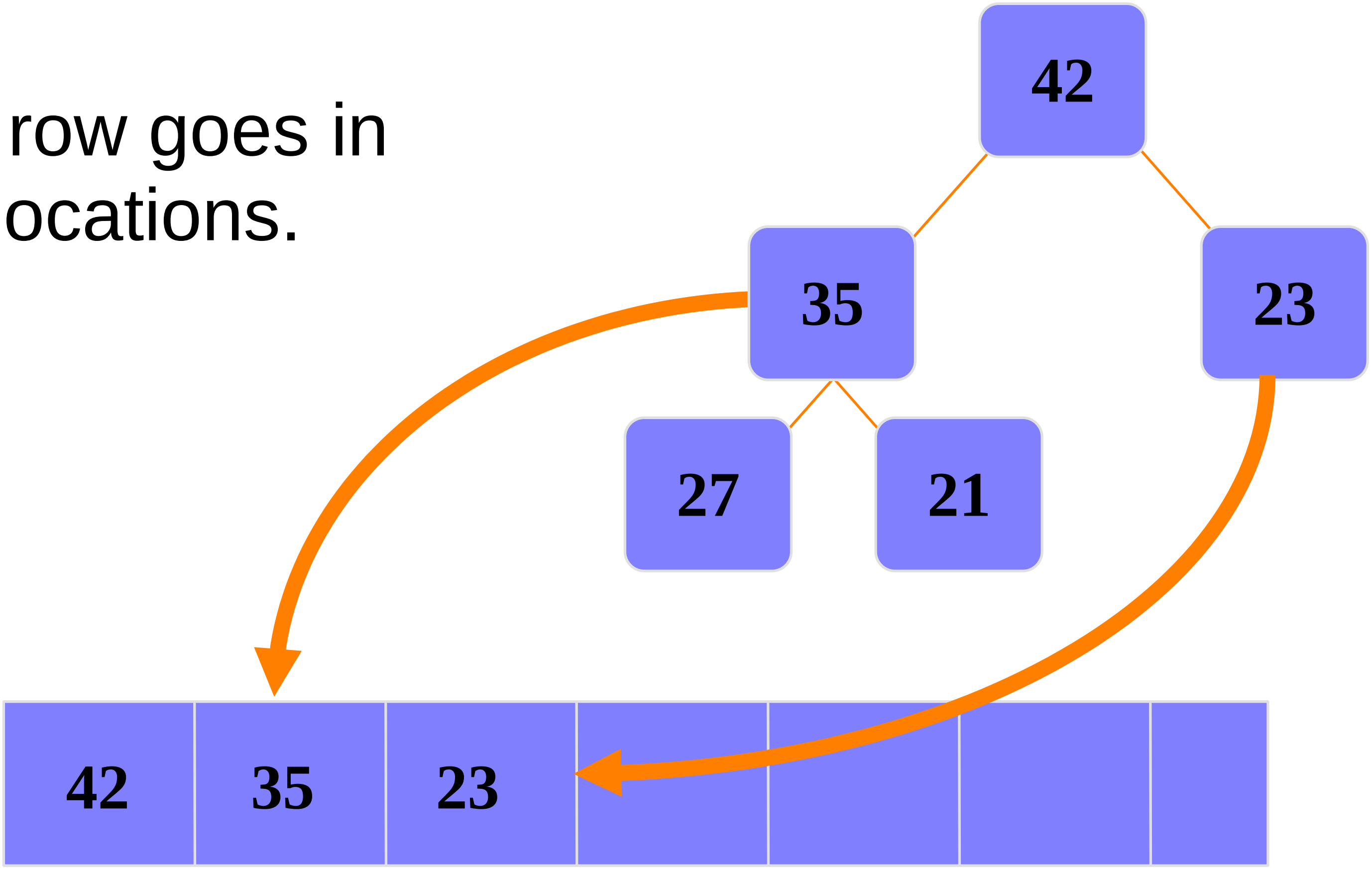
Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

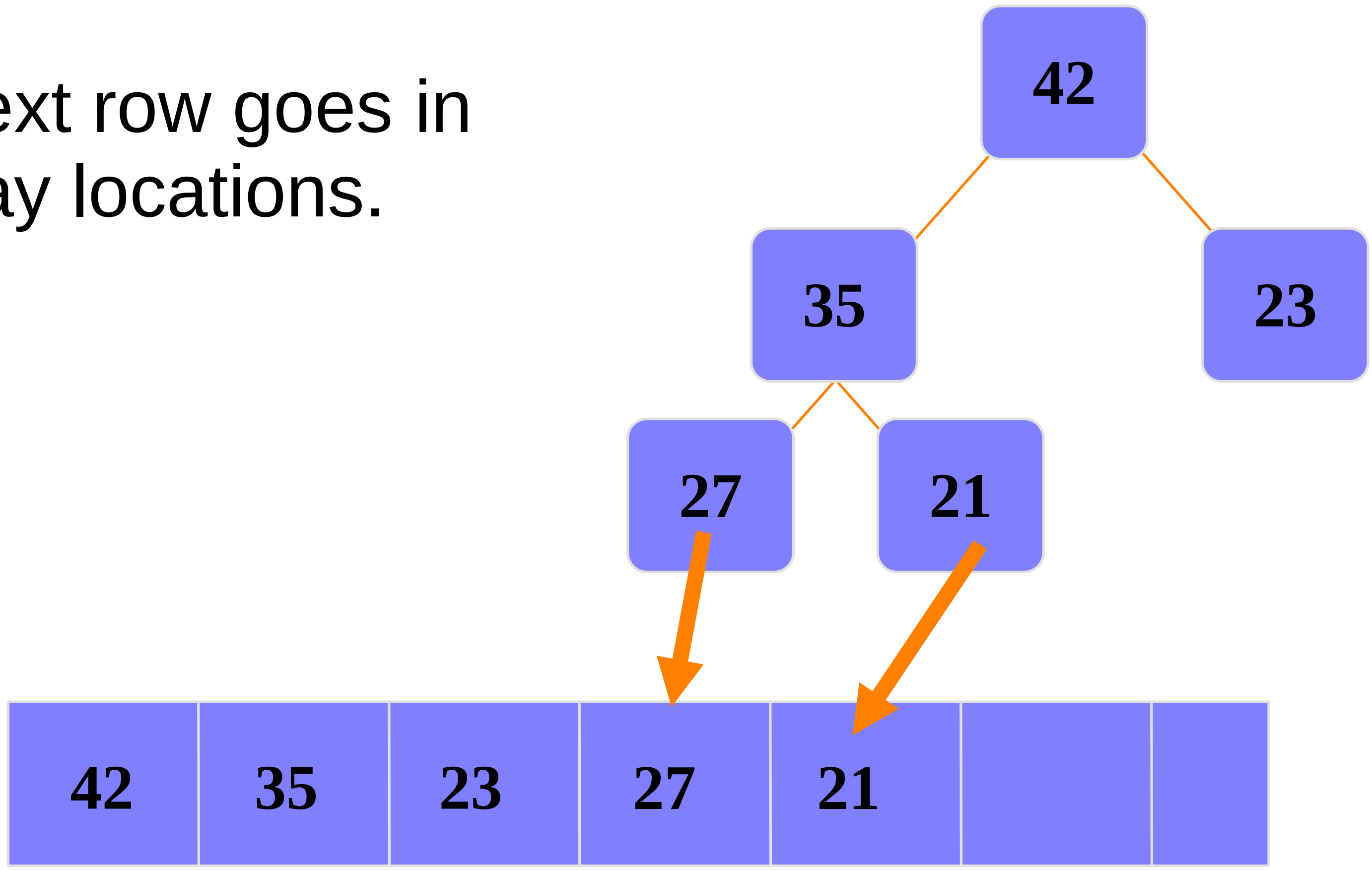
Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

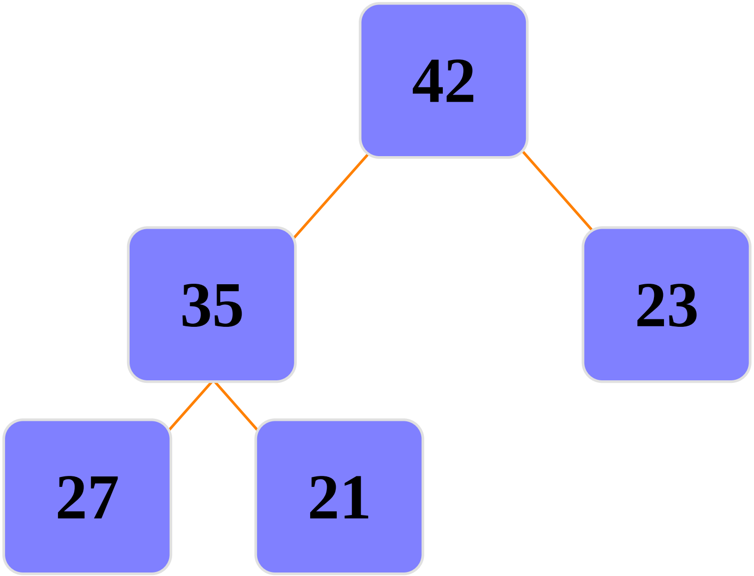
Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

Data from the next row goes in the next two array locations.



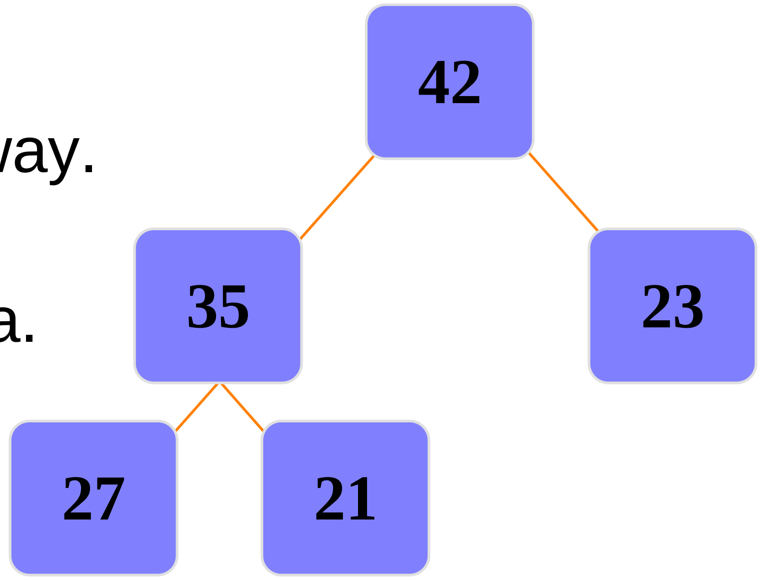
42	35	23	27	21		

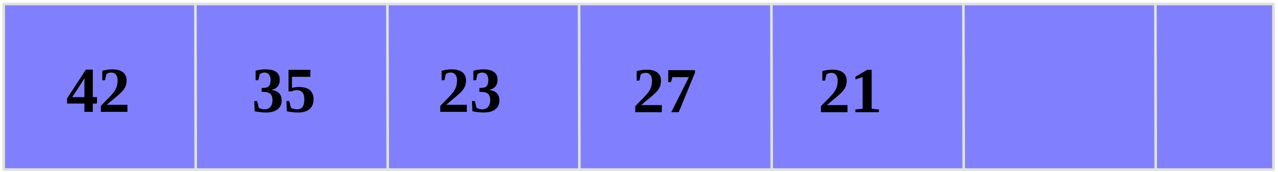
An array of data

We don't care what's in this part of the array.

Important Points about the Implementation

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

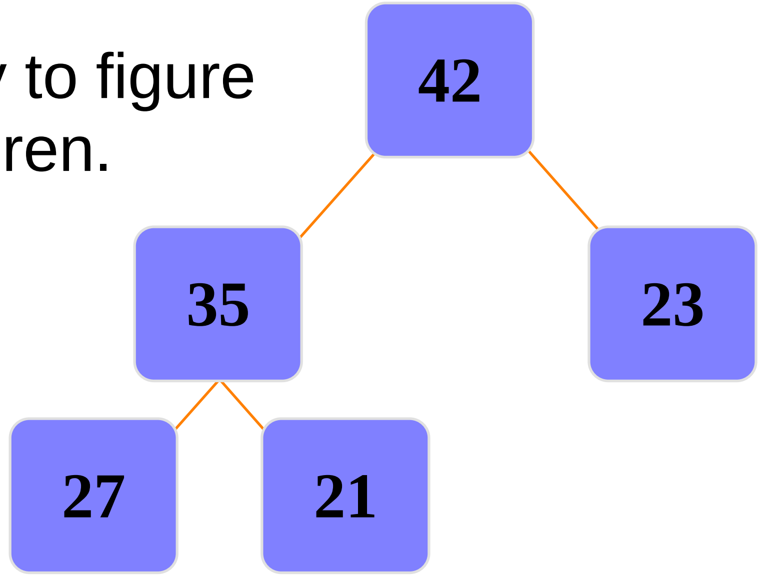


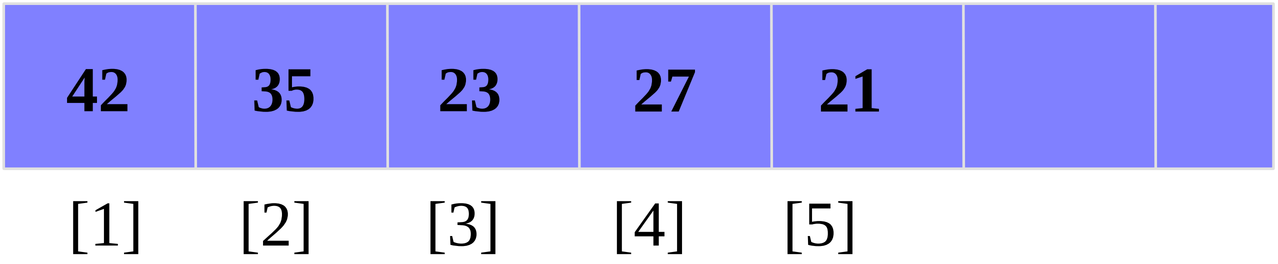


An array of data

Important Points about the Implementation

If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.





Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

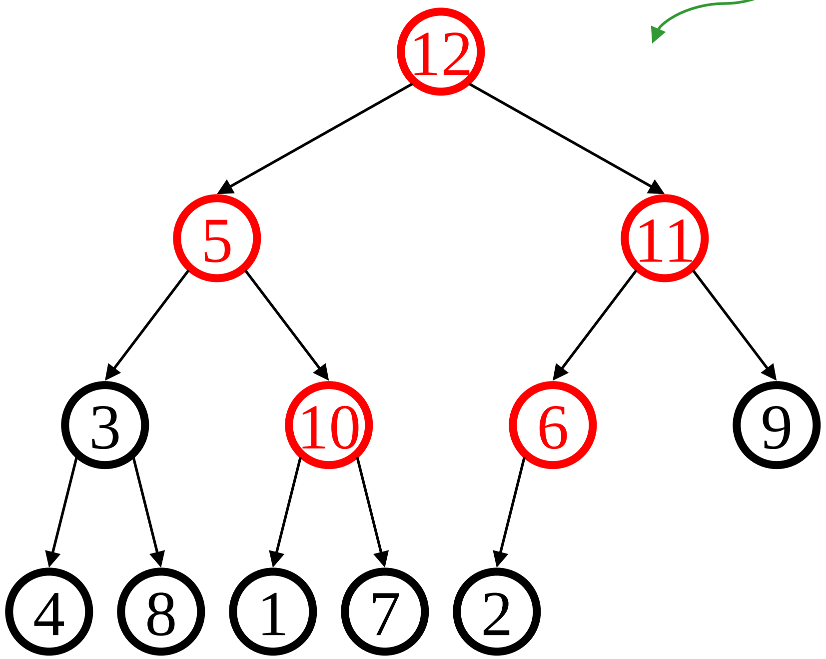
```
PARENT(i)
return ⌊i/2⌋

LEFT(i)
return 2i

RIGHT(i)
return 2i + 1
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

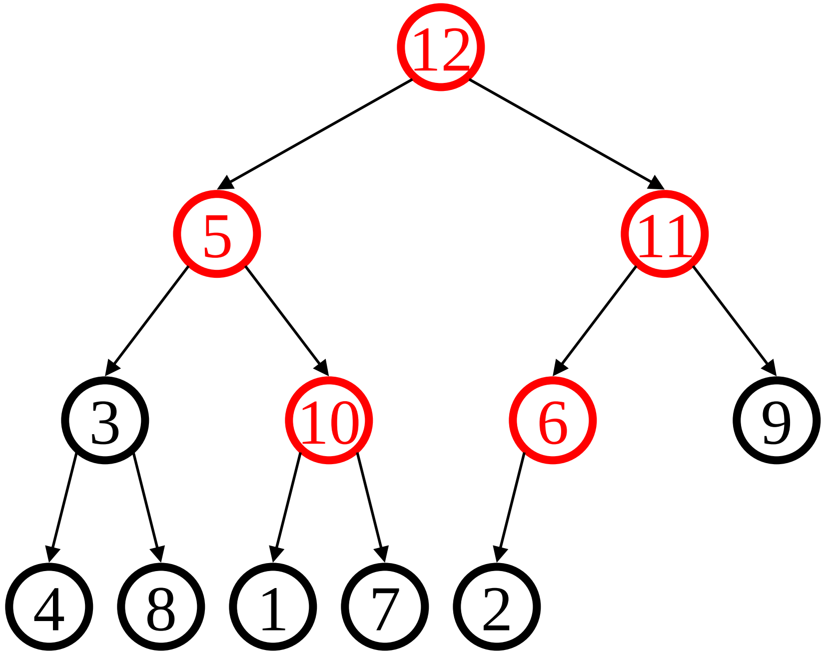
BuildHeap

12	5	11	3	10	6	9	4	8	1	7	2

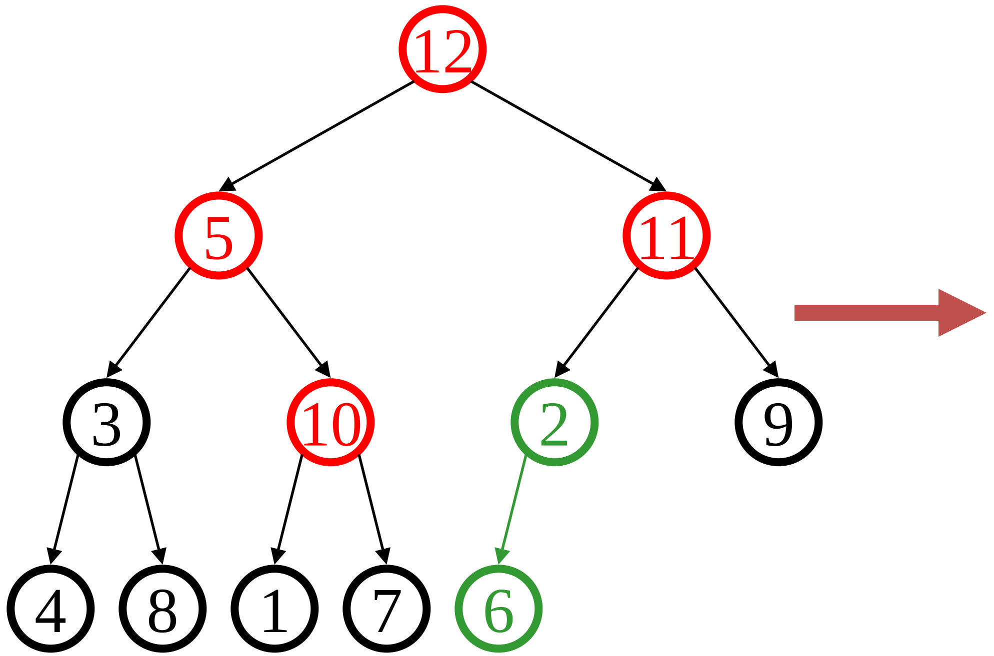
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



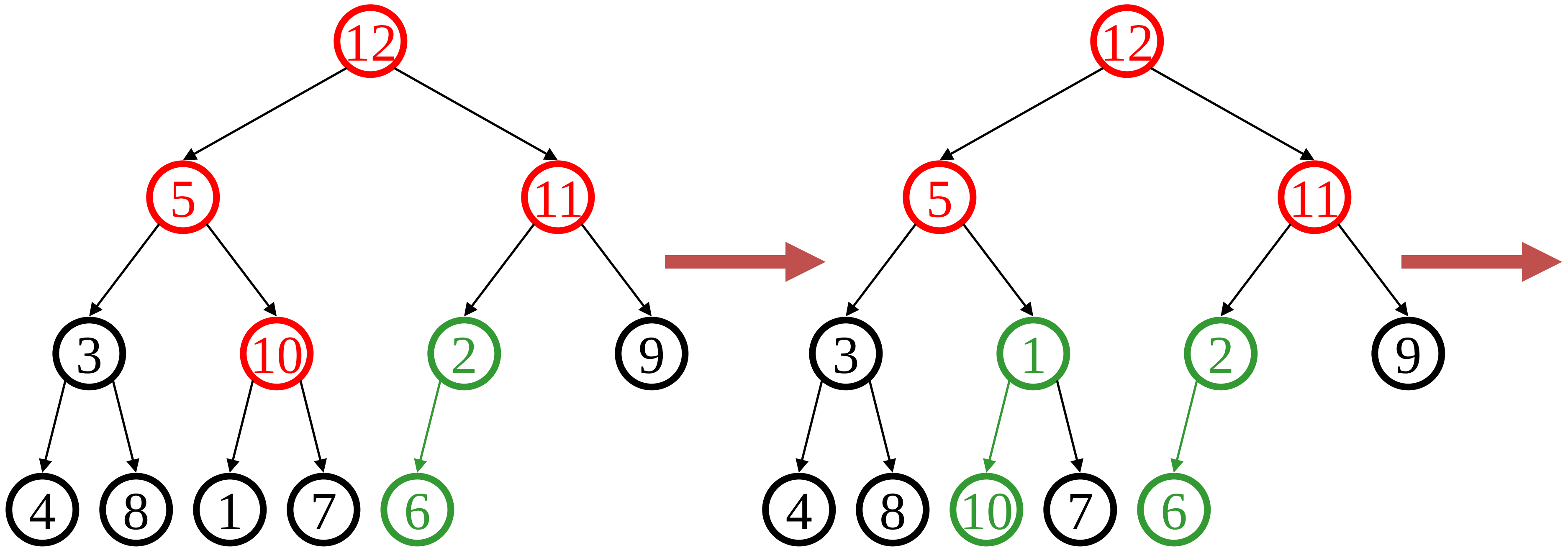
BuildHeap



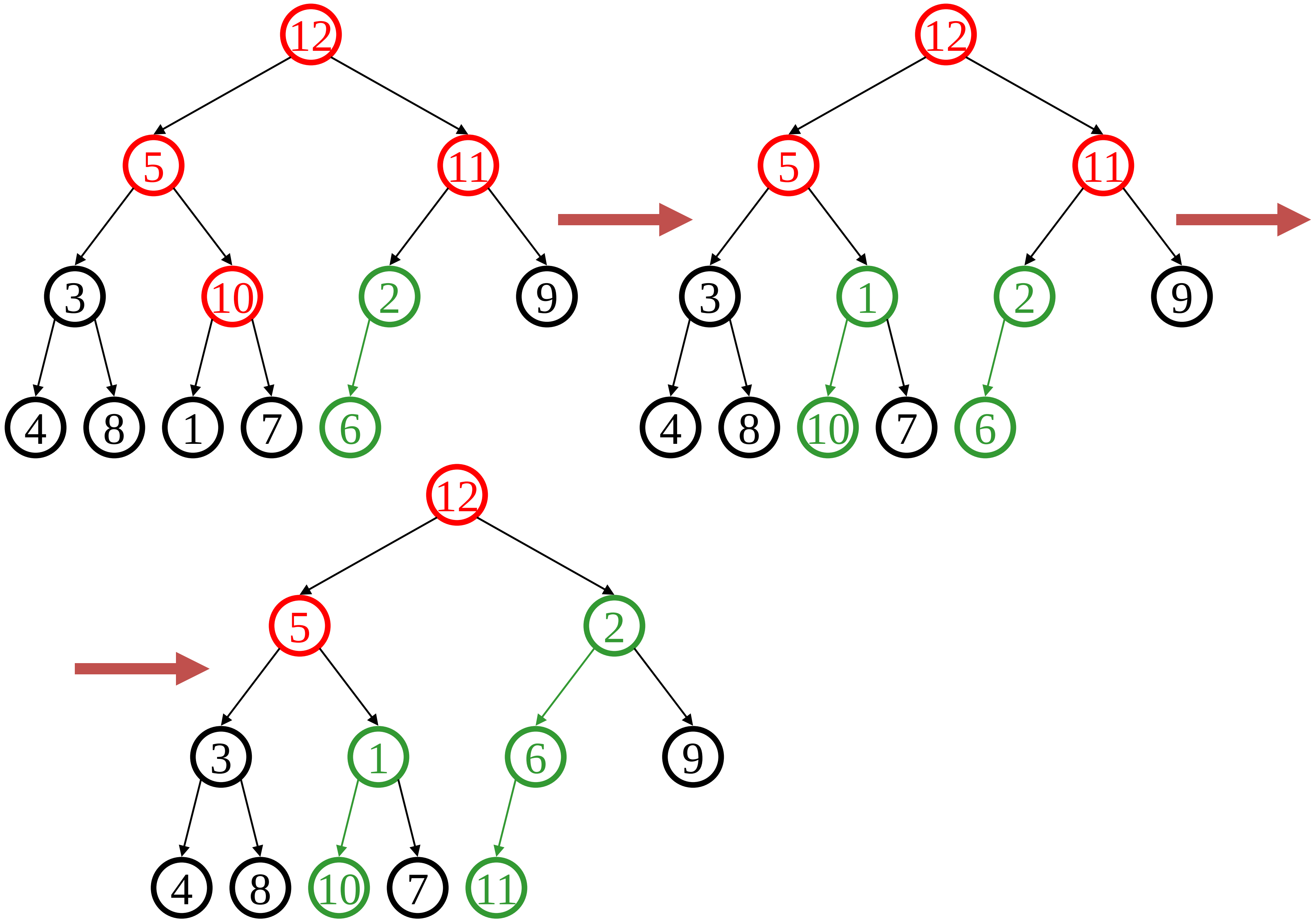
BuildHeap



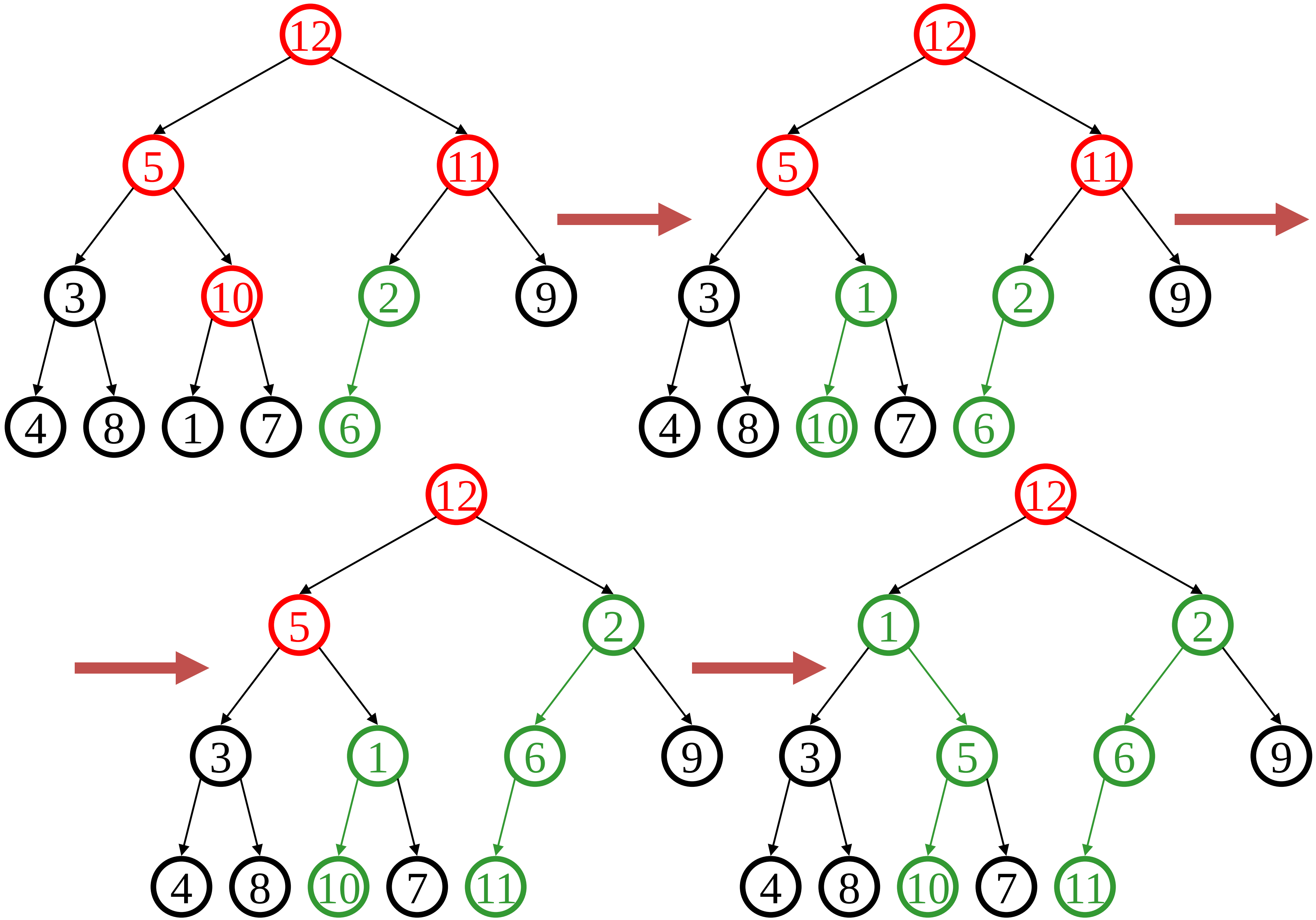
BuildHeap



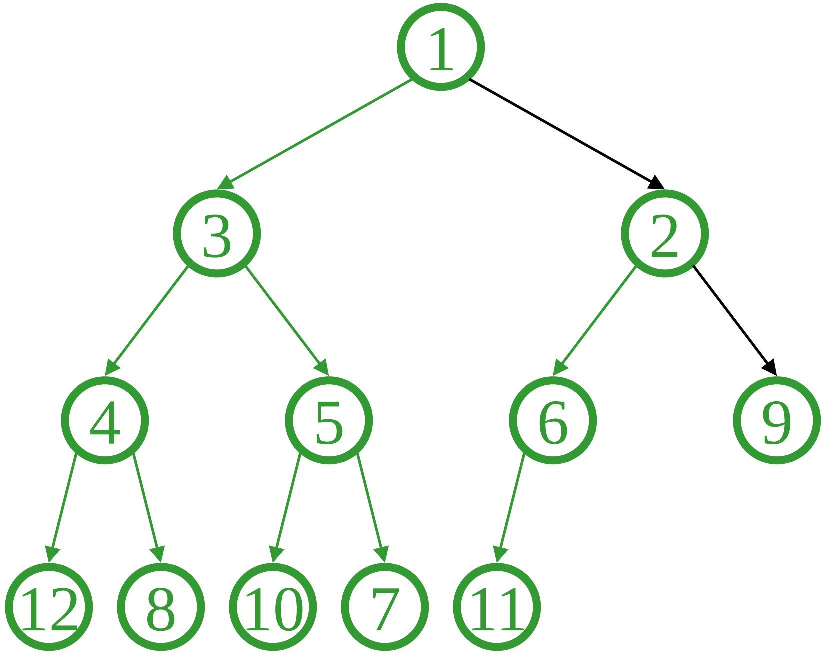
BuildHeap



BuildHeap


12
5
11
3
10
2
9
4
8
1
7
6
12
5
11
3
1
2
9
4
8
10
7
6
12
5
2
3
1
6
9
4
8
10
7
11
12
1
2
3
5
6
9
4
8
10
7
11

BuildHeap

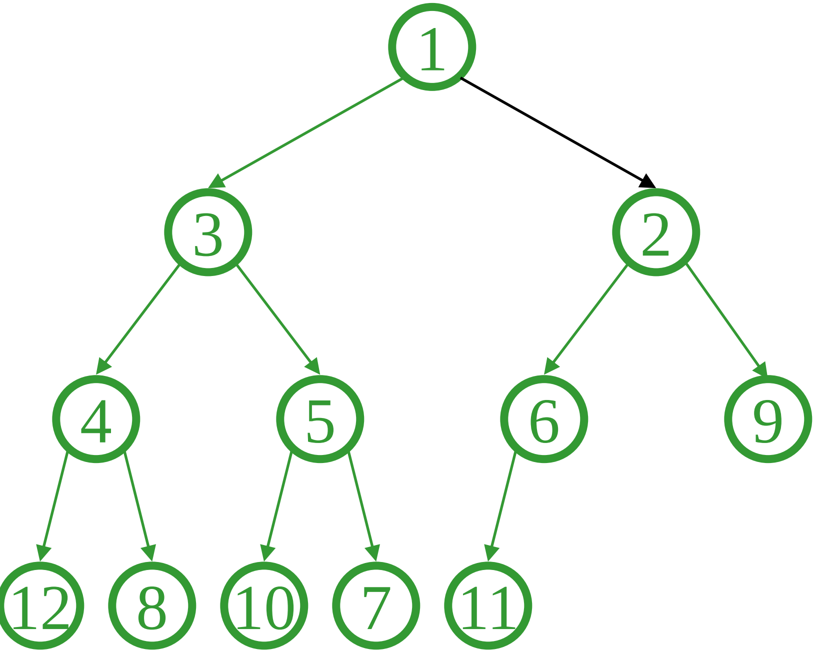


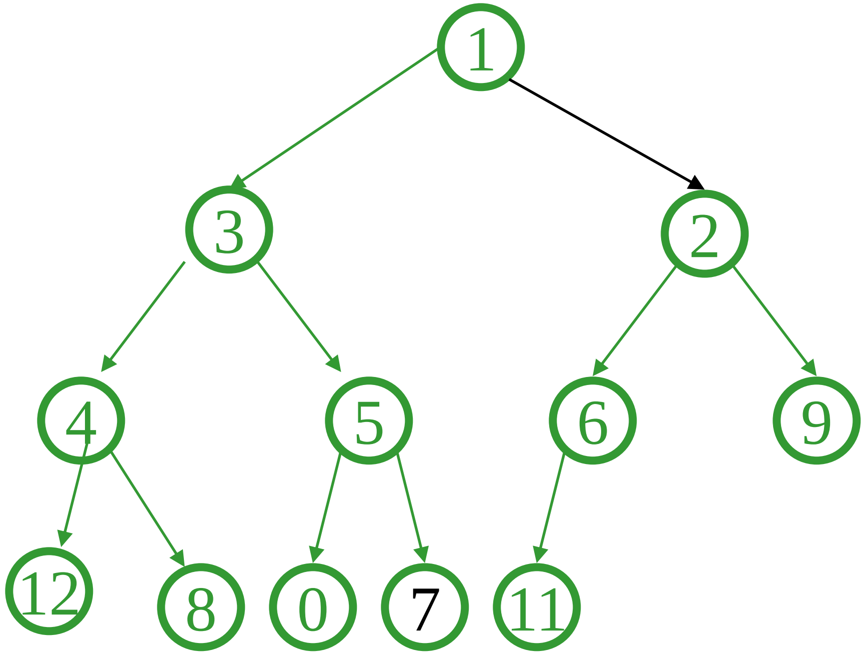
Decrease key

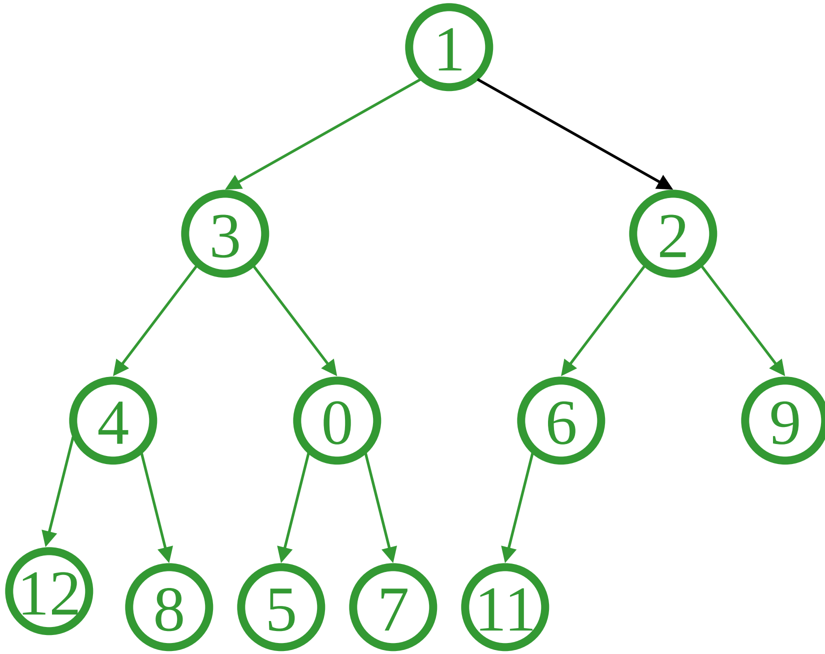
- Decrease key is used for some application to increase the priority of the key.
    - First decrease the key value
    - Perform reheapification upward
  - If it is max heap, to increase the priority, it will increase key but the steps are the same
    - First increase the key value
    - Perform reheapification upward

Example

Decrease Key 10 to 0







Example