Introduction to Priority Queues and Heap

Priority Queue

Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Applications of Priority Queues

- Any event/job management that assign priority to events/jobs
- In Operating Systems
  - Scheduling jobs
- In Simulators
  - Scheduling the next event (smallest event time)

Priority Queue Implementation

- Implemented as Linked lists
    - O(N) worst-case time on either insert() or deleteMin()
- Binary Search Trees
    - O(log(N)) on insert() and delete()
    - Overkill: all elements are sorted
        - However, we only need the minimum element
- Heaps
    - This is what we will study and use to implement Priority Queues
    - O(logN) worst case for both insertion and delete operations

Priority Queue Implementation

A typical priority queue supports following operations.

- **insert(item, priority):** Inserts an item with given priority.
- **getHighestPriority():** Returns the highest priority item.
- **deleteHighestPriority():** Removes the highest priority item.

Priority Queue Implementation

- **getHighestPriority**() operation can be implemented by linearly searching the highest priority item in array. This operation takes **O(n)** time.
- **deleteHighestPriority**() operation can be implemented by first linearly searching an item, then removing the item **by moving all subsequent items one position back.**
- We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don’t have to move items.

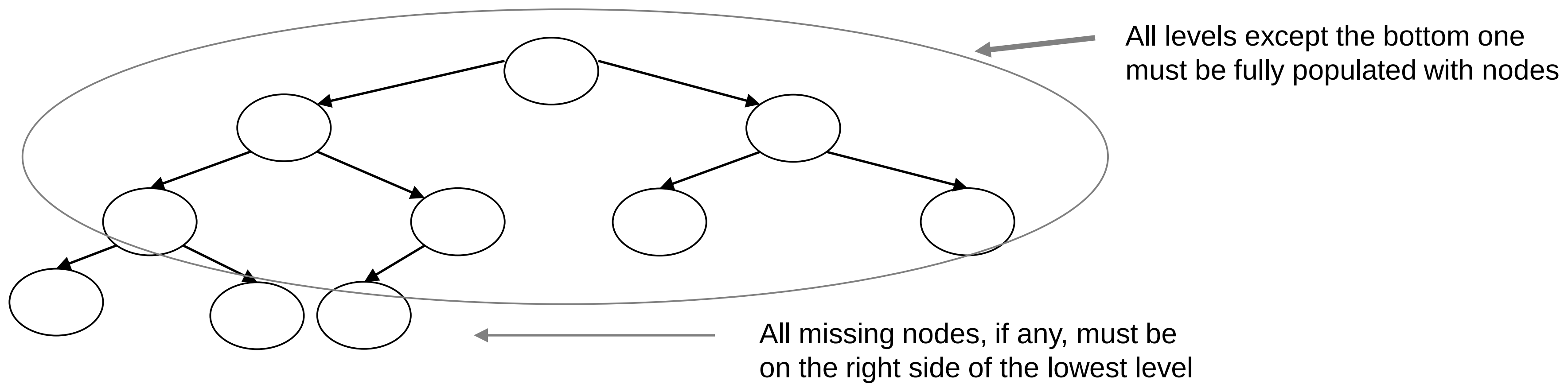
Priority Queue Implementation

Using Heaps:

- Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list.
- In a Binary Heap, **getHighestPriority**() can be implemented in O(1) time, **insert**() can be implemented in O(logn) time and **deleteHighestPriority**() can also be implemented in O(logn) time.

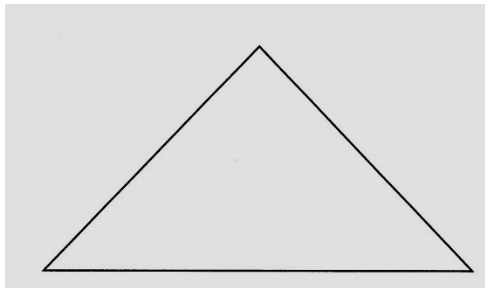
What is a Binary Heap?

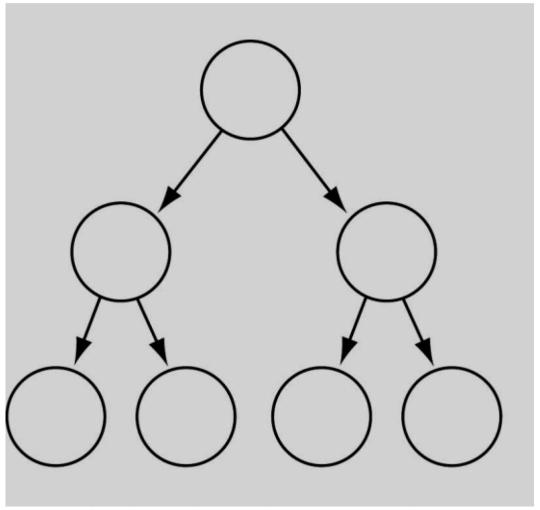
- A **binary heap** is a *complete binary tree* with one (or both) of the following heap order properties:
  - **MinHeap property:** Each node must have a key that is less or equal to the key of each of its children.
  - **MaxHeap property:** Each node must have a key that is greater or equal to the key of each of its children.
- A binary heap satisfying the MinHeap property is called a MinHeap.
- A binary heap satisfying the MaxHeap property is called a MaxHeap.
- A binary heap with all keys equal is both a MinHeap and a MaxHeap.
- Recall: A complete binary tree may have missing nodes only on the right side of the lowest level.


All levels except the bottom one must be fully populated with nodes
All missing nodes, if any, must be on the right side of the lowest level

Full Binary Tree

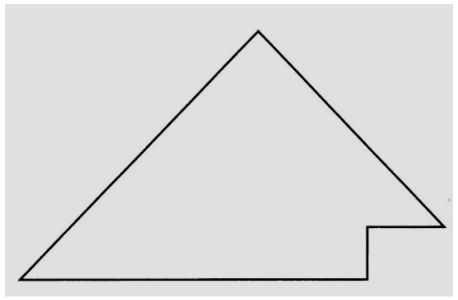
- Every non-leaf node has two children
- All the leaves are on the same level

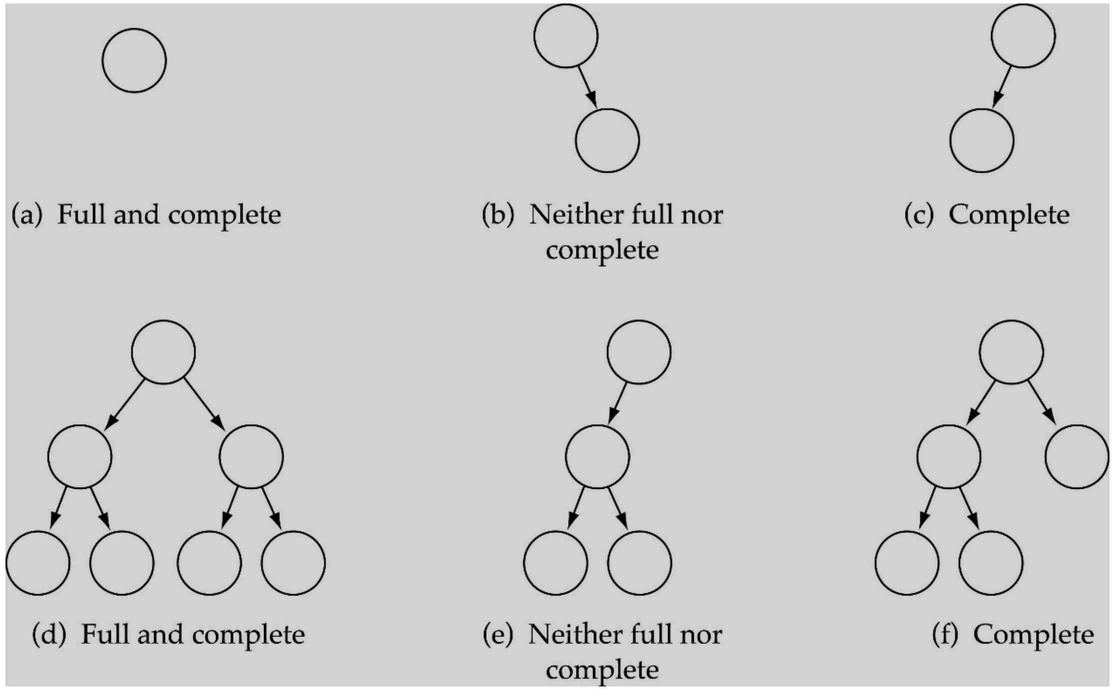




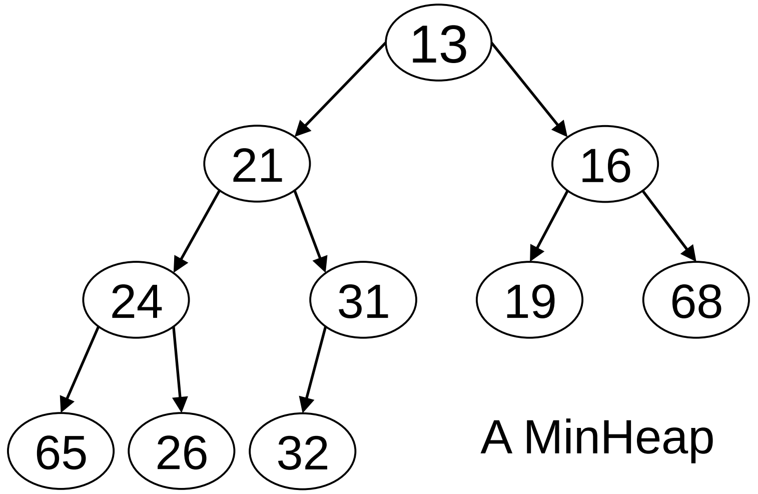
Complete Binary Tree

- A binary tree that is either full or full through the next-to-last level
- The last level is full **from left to right** (i.e., leaves are as far to the left as possible)



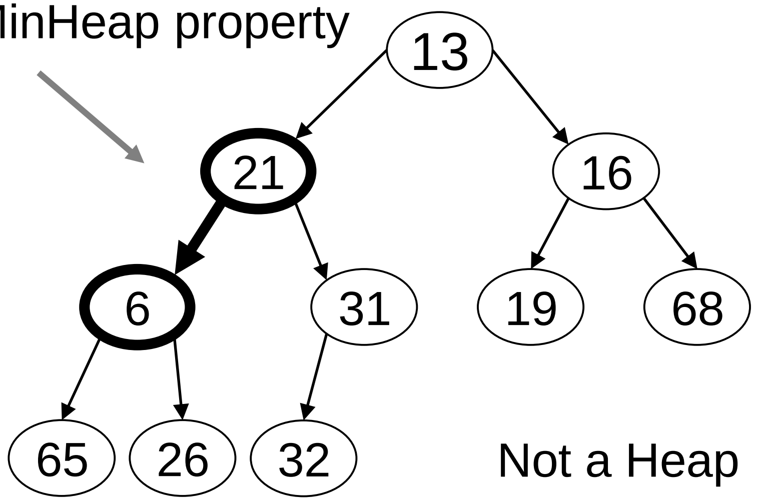


MinHeap and non-MinHeap examples

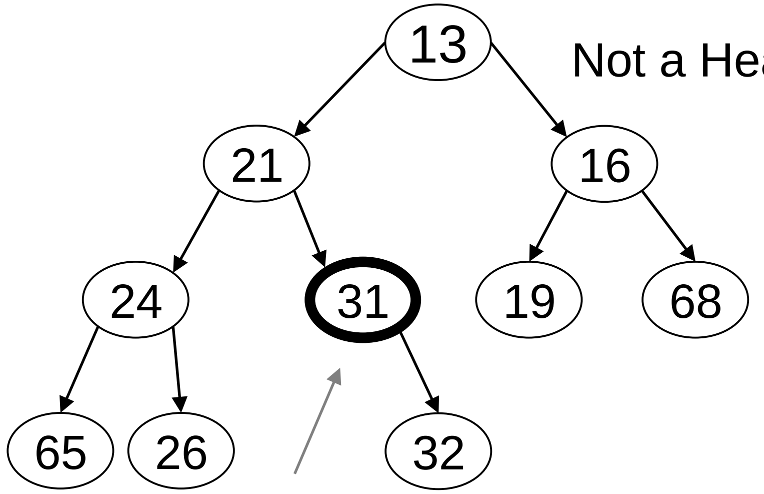


A MinHeap

Violates MinHeap property 21>6

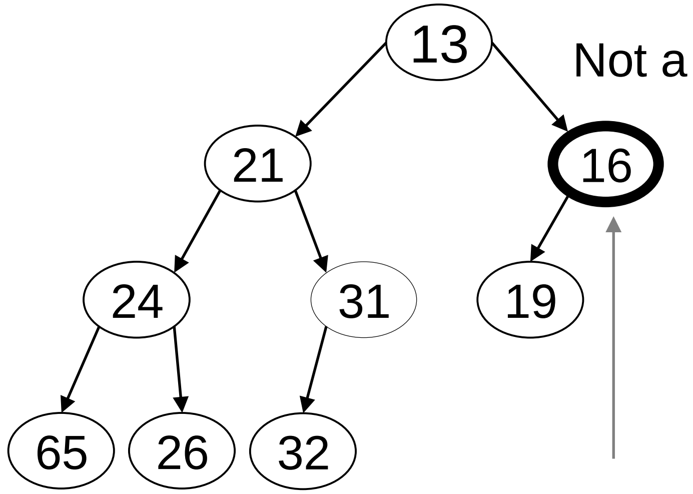


Not a Heap



Not a Heap

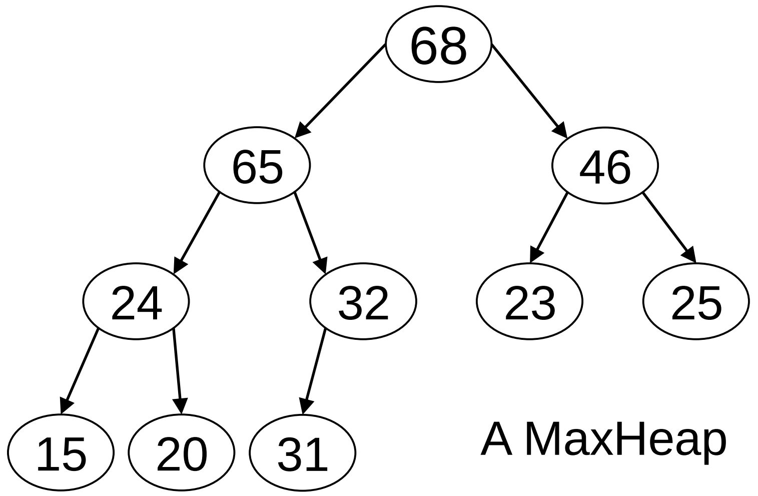
Violates heap structural property



Not a Heap

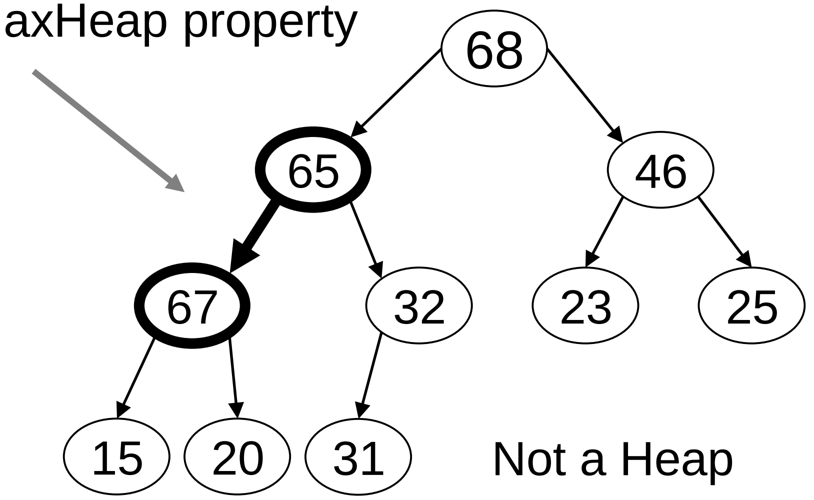
Violates heap structural property

MaxHeap and non-MaxHeap examples

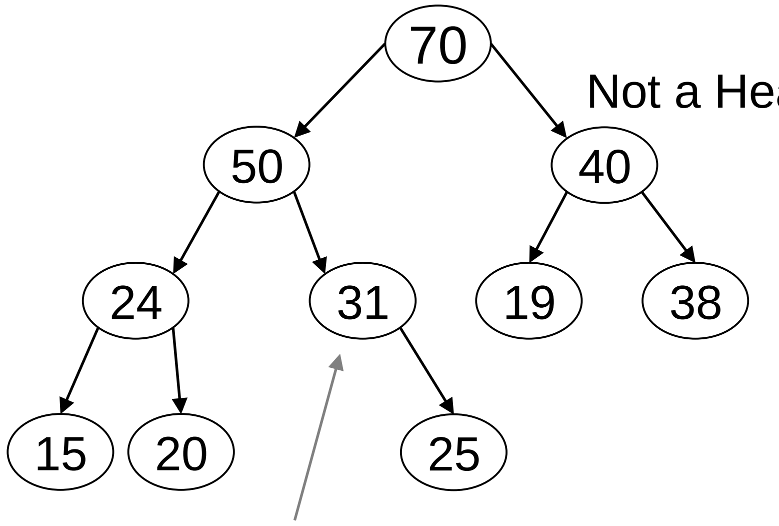


A MaxHeap

Violates MaxHeap property
65 < 67

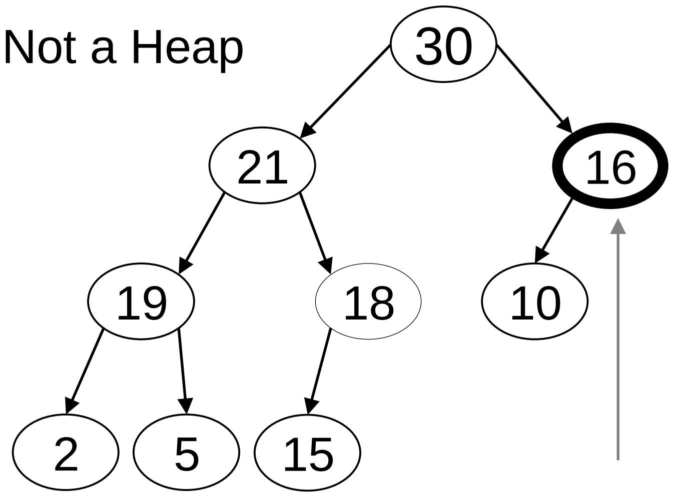


Not a Heap



Not a Heap

Not a Heap



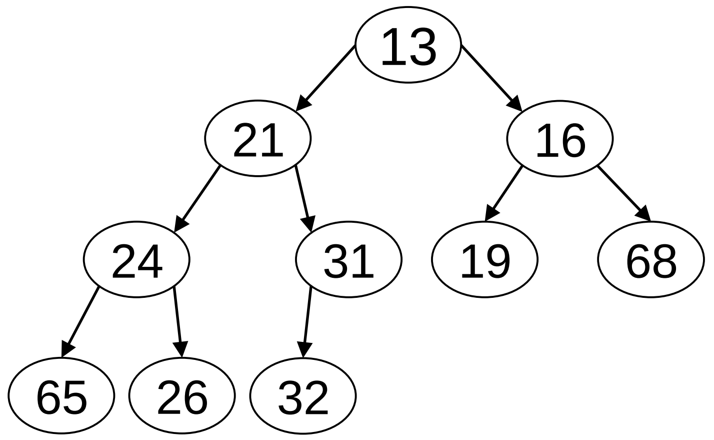
Violates heap structural property

Violates heap structural property

Array Representation of a Binary Heap

- A heap is a dynamic data structure that is represented and manipulated more efficiently using an array.
- Since a heap is a complete binary tree, its node values can be stored in an array, without any gaps, in a breadth-first order, where:

$$\text{Value}(\text{node}_{i+1}) \longrightarrow \text{array}[\,i\,],\ \text{for } i \geq 0$$



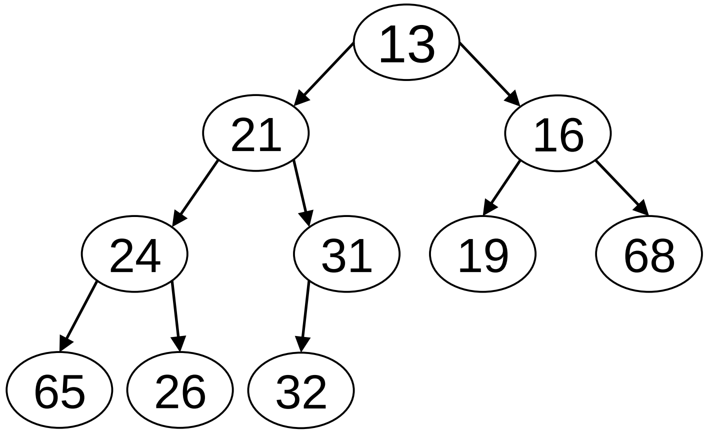
13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9

- The root is array[0]
- The parent of array[i] is array[(i – 1)/2], where i > 0
- The left child, if any, of array[i] is array[2i+1].
- The right child, if any, of array[i] is array[2i+2].

Array Representation of a Binary Heap (contd.)

- We shall use an implementation in which the heap elements are stored in an array starting at index 1.

$$\text{Value}(\text{node}_i) \longrightarrow \text{array}[i]\ ,\ \text{for } i \geq 1$$



	13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9	10

- The root is array[1].
- The parent of array[i] is array[i/2], where i > 1
- The left child, if any, of array[i] is array[2i].
- The right child, if any, of array[i] is array[2i+1].

MinHeap enqueue

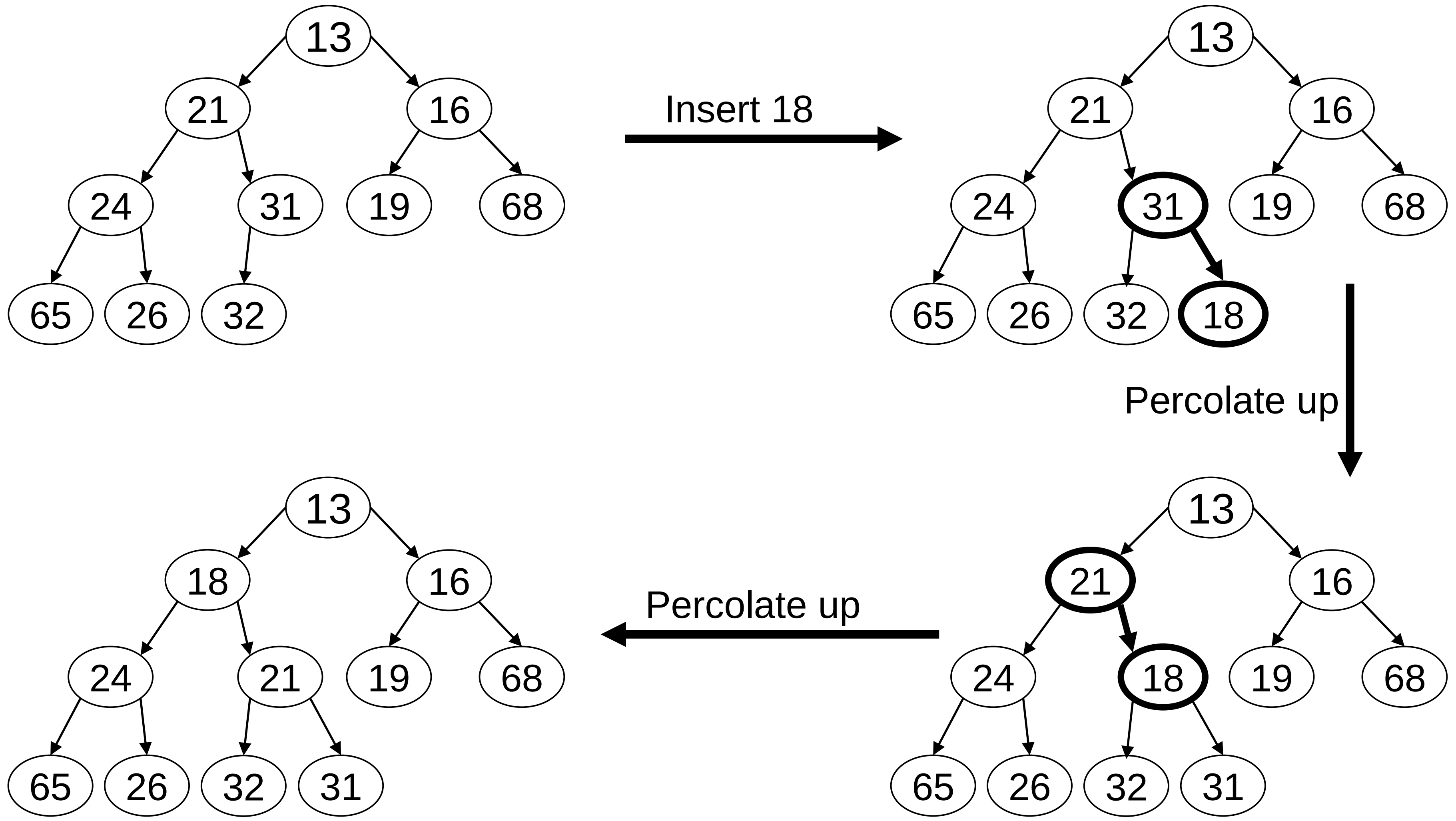
- The pseudo code algorithm for enqueing a key in a MinHeap is:

```
enqueue(e1)
{
    if(the heap is full) throw an exception ;
    insert e1 at the end of the heap ;
    while(e1 is not in the root node and e1 < parent(e1))
        swap(e1 , parent(e1)) ;
}
```

- The process of swapping an element with its parent, in order to restore the heap order property is called percolate up, sift up, or reheapification upward.

- Thus, the steps for enqueue are:
  1. Enqueue the key at the end of the heap.
  2. As long as the heap order property is violated, percolate up.

MinHeap Insertion Example



MinHeap dequeue

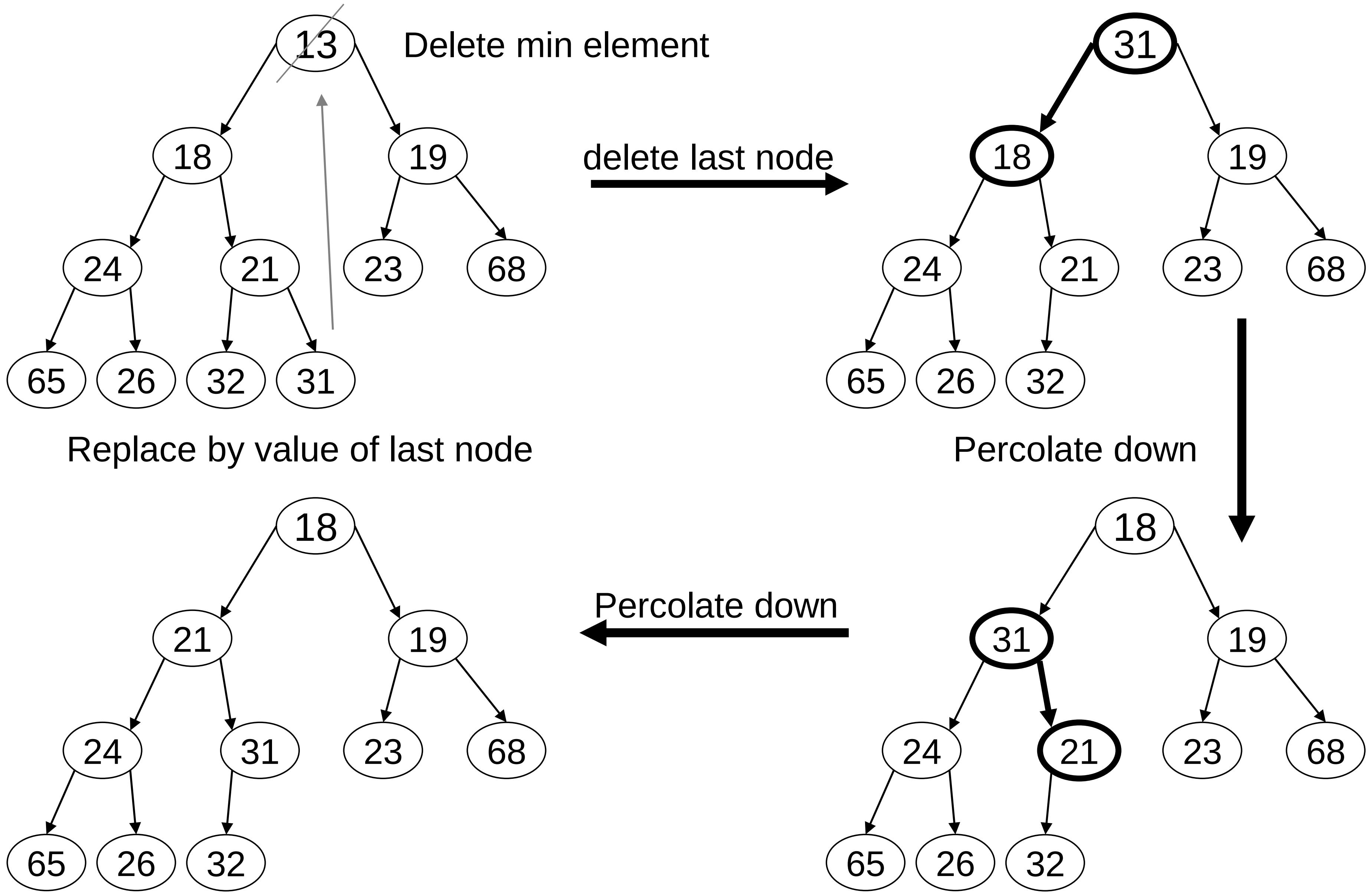
- The pseudo code algorithm for deleting the root key in a MinHeap is:

```
dequeueMin(){
    if(Heap is empty) throw an exception ;
    extract the element from the root ;
    if(root is a leaf node){ delete root ; return; }
    copy the element from the last leaf to the root ;
    delete last leaf ;
    p = root ;
    while(p is not a leaf node and p > any of its children)
        swap p with the smaller child ;
    return ;
}
```

- The process of swapping an element with its child, in order to restore the heap order property is called percolate down, sift down, or reheapification downward.

- Thus, the steps for deletion are:
  1. Replace the key at the root by the key of the last leaf node.
  2. Delete the last leaf node.
  3. As long as the heap order property is violated, percolate down.

MinHeap Dequeue Example

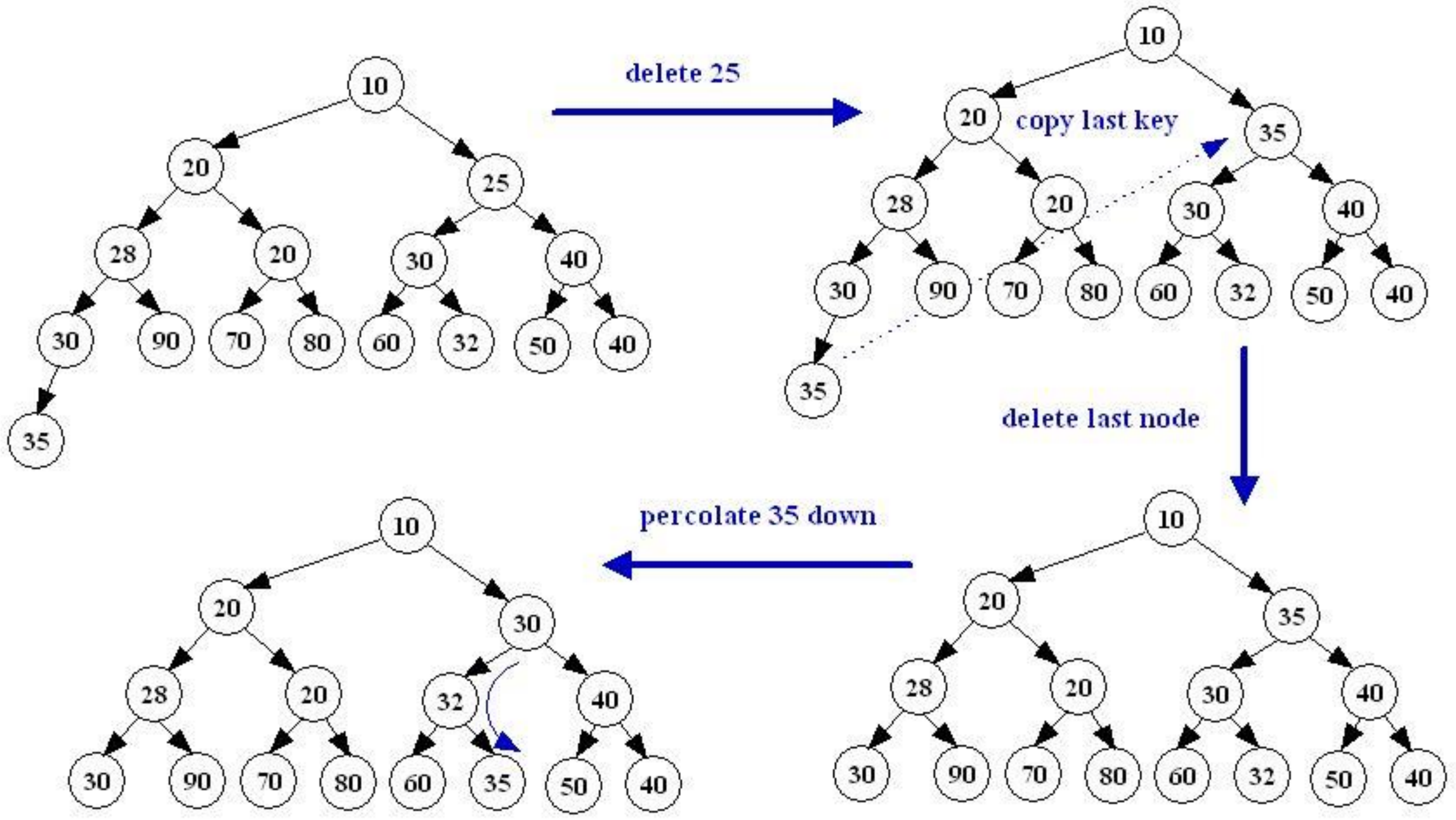


Deleting an arbitrary key

The algorithm of deleting an arbitrary key from a heap is:

- Copy the key **x** of the last node to the node containing the deleted key.
- Delete the last node.
- Percolate **x** down until the heap property is restored.

Example:

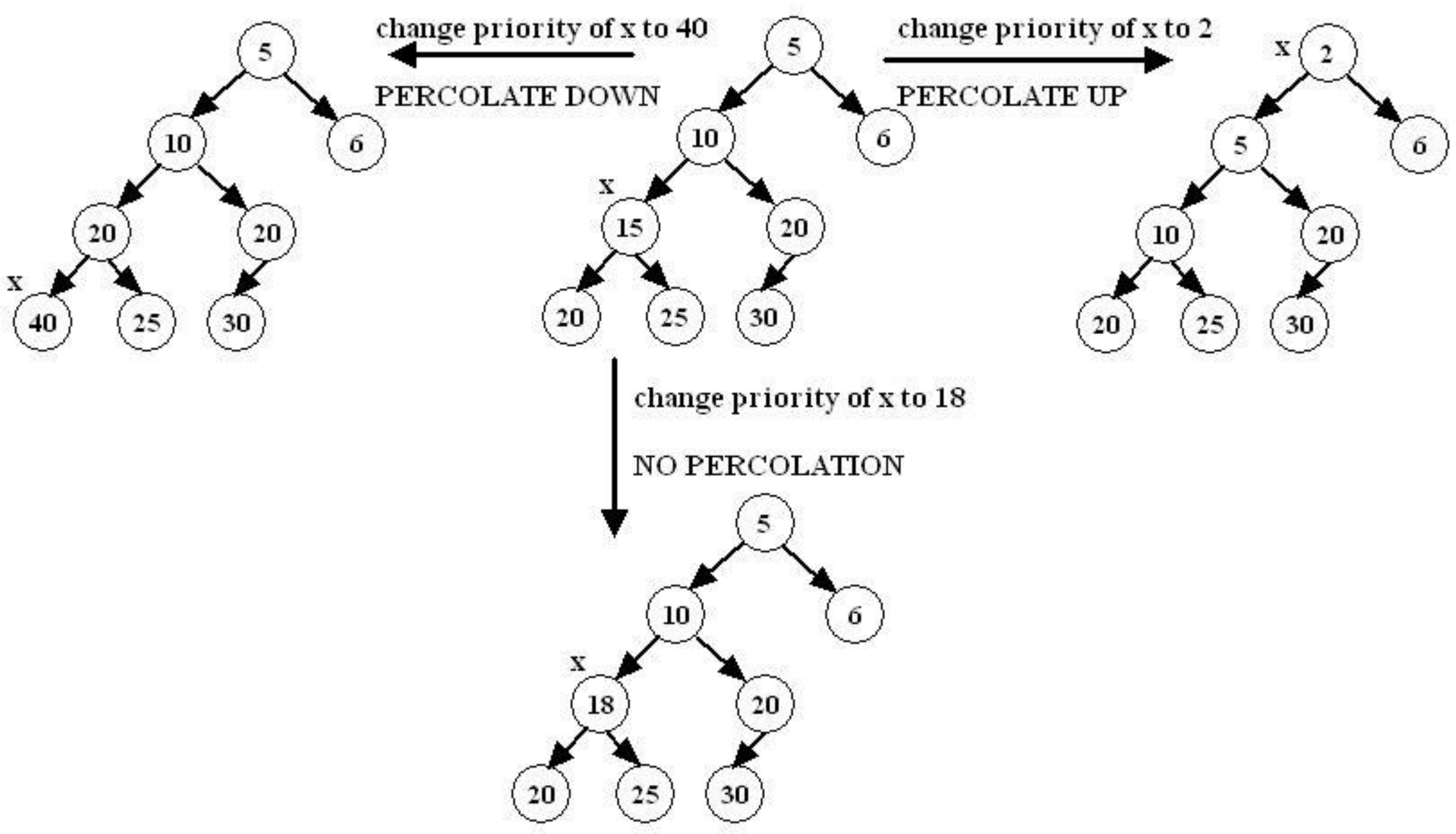


Changing the priority of a key

There are three possibilities when the priority of a key **x** is changed:

1. The heap property is not violated.
2. The heap property is violated and **x** has to be percolated up to restore the heap property.
3. The heap property is violated and **x** has to be percolated down to restore the heap property.

Example:


change priority of x to 40
PERCOLATE DOWN
change priority of x to 2
PERCOLATE UP
change priority of x to 18
NO PERCOLATION
5
10
6
x
15
20
20
25
30
40
18
2

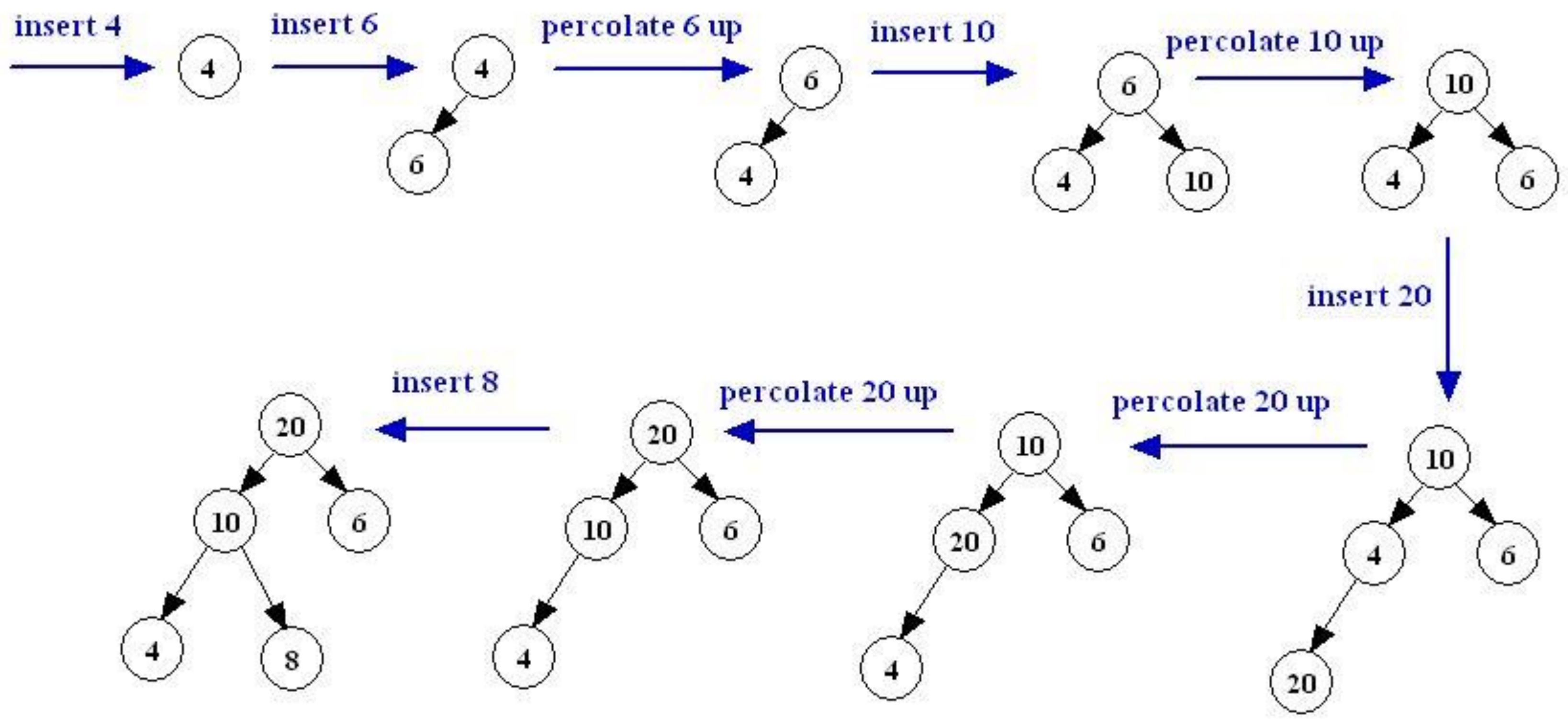
Building a heap (top down)

- A heap is built top-down by inserting one key at a time in an initially empty heap.
- After each key insertion, if the heap property is violated, it is restored by percolating the inserted key upward.

The algorithm is:

```
for(int i=1; i <= heapSize; i++){
   read  key;
   binaryHeap.enqueue(key);
}
```

Example: Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap



Converting an array into a Binary heap (Building a heap bottom-up)

- The algorithm to convert an array into a binary heap is:

1. Start at the level containing the last non-leaf node (i.e., array[n/2], where n is the array size).
2. Make the subtree rooted at the last non-leaf node into a heap by invoking percolateDown.
3. Move in the current level from right to left, making each subtree, rooted at each encountered node, into a heap by invoking percolateDown.
4. If the levels are not finished, move to a lower level then go to step 3.

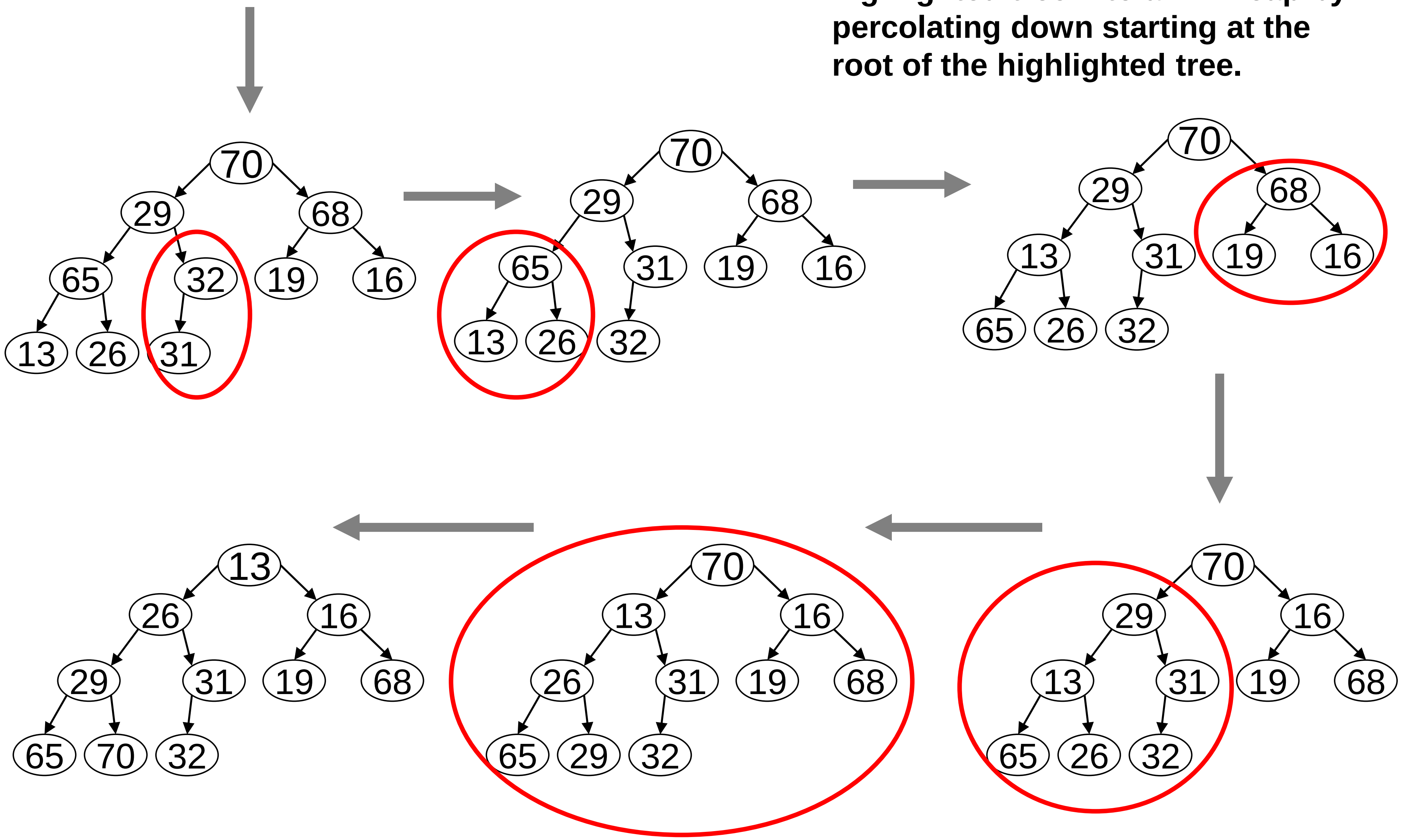
- The above algorithm can be refined to the following method of the BinaryHeap class:

```
private void buildHeapBottomUp()
{
 for(int i = count / 2; i >= 1; i--)
           percolateDown(i);
}
```

Converting an array into a MinHeap (Example)

70	29	68	65	32	19	16	13	26	31

At each stage convert the highlighted tree into a MinHeap by percolating down starting at the root of the highlighted tree.



Heap Application: Heap Sort

- A MinHeap or a MaxHeap can be used to implement an efficient sorting algorithm called Heap Sort.
- The following algorithm uses a MinHeap:

```
public  static void  heapSort(Comparable[] array){
   BinaryHeap  heap  =  new BinaryHeap(array) ;
   for(int i = 0; i < array.length; i++)
      array[i] = heap.dequeueMin() ;
}
```

- Because the dequeueMin algorithm is O(log n), heapSort is an O(n log n) algorithm.
- Apart from needing the extra storage for the heap, heapSort is among efficient sorting algorithms.

Heap Applications: Priority Queue

- A heap can be used as the underlying implementation of a priority queue.
- A priority queue is a data structure in which the items to be inserted have associated priorities.
- Items are withdrawn from a priority queue in order of their priorities, starting with the highest priority item first.
- Priority queues are often used in resource management, simulations, and in the implementation of some algorithms (e.g., some graph algorithms, some backtracking algorithms).
- Several data structures can be used to implement priority queues. Below is a comparison of some:

Data structure	Enqueue	Find Min	Dequeue Min
Unsorted List	O(1)	O(n)	O(n)
Sorted List	O(n)	O(1)	O(1)
AVL Tree	O(log n)	O(log n)	O(log n)
MinHeap	O(log n)	O(1)	O(log n)

Heaps

A **heap** is a certain kind of complete binary tree.

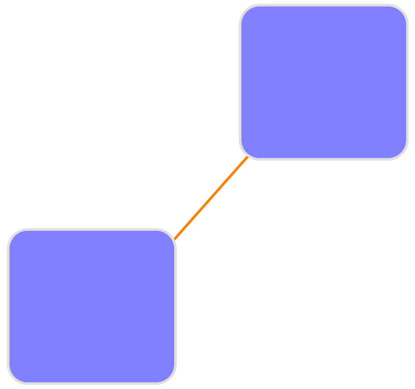
Heaps

A **heap** is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

Heaps

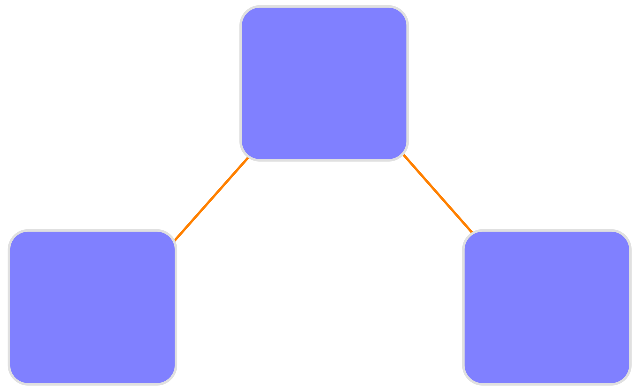
Complete binary tree.



The second node is always the left child of the root.

Heaps

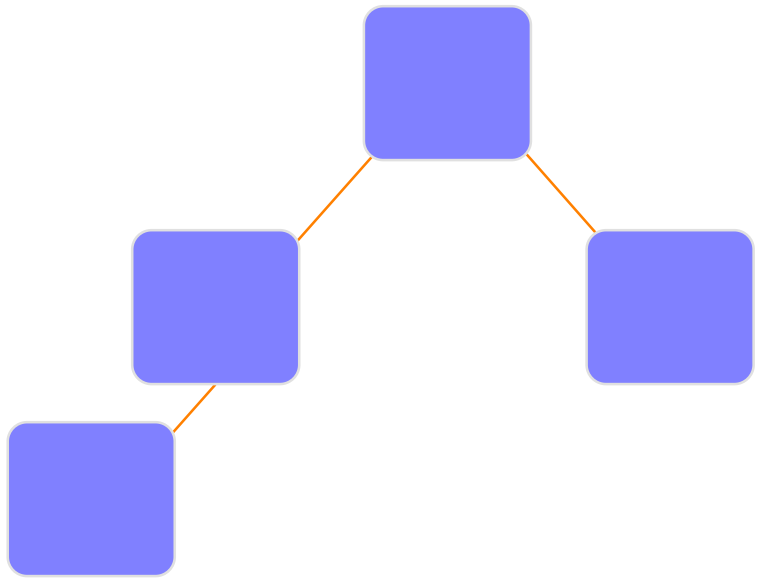
Complete binary tree.



The third node is always the right child of the root.

Heaps

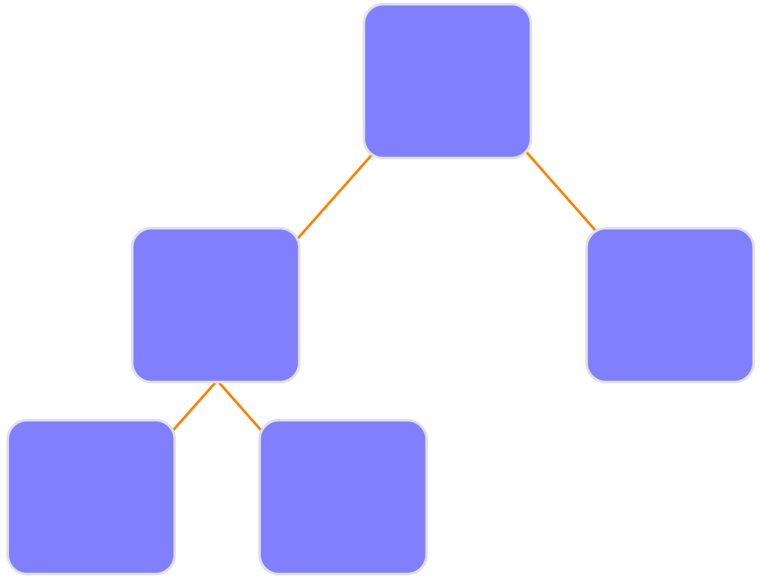
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

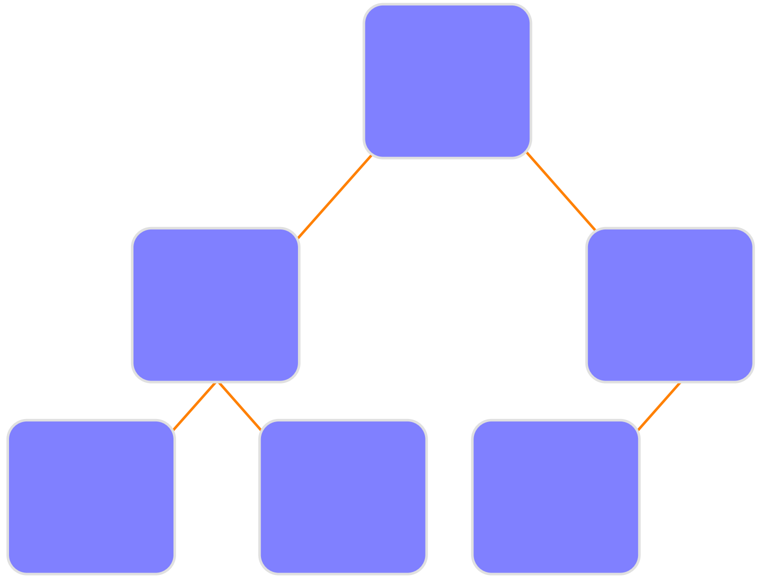
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

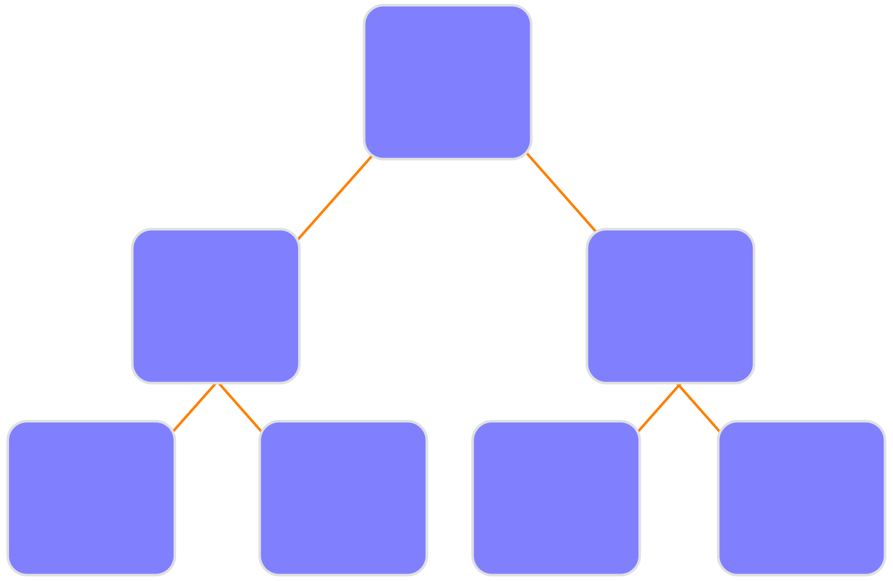
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

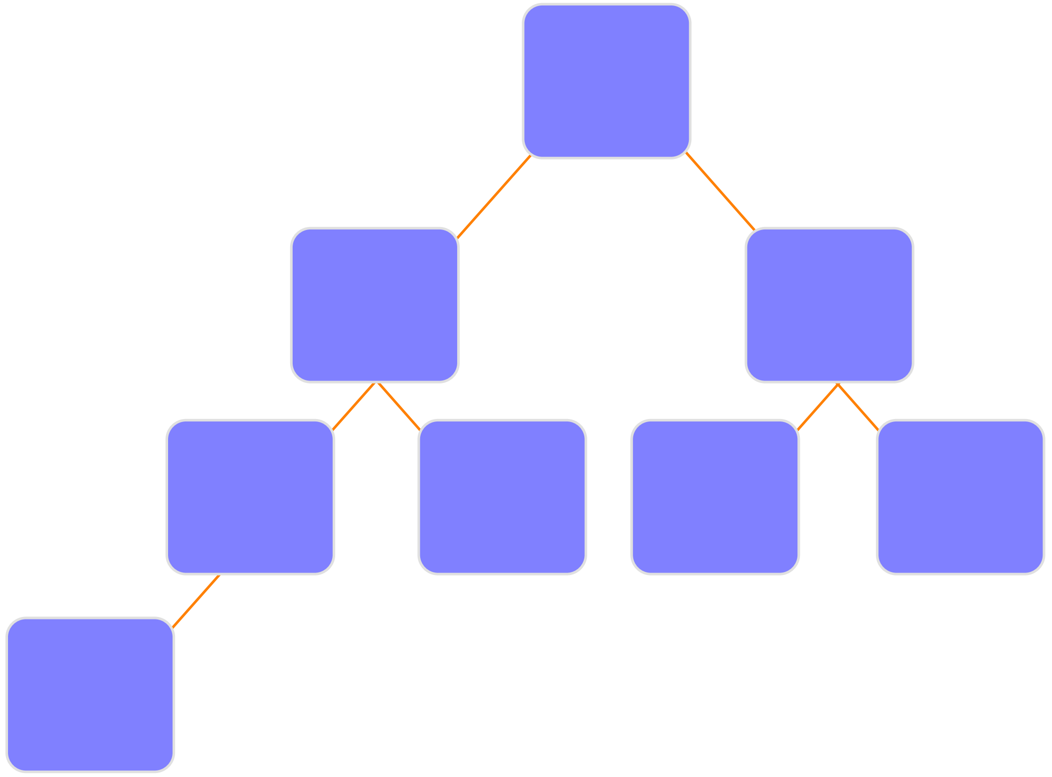
Complete binary tree.



The next nodes always fill the next level from left-to-right.

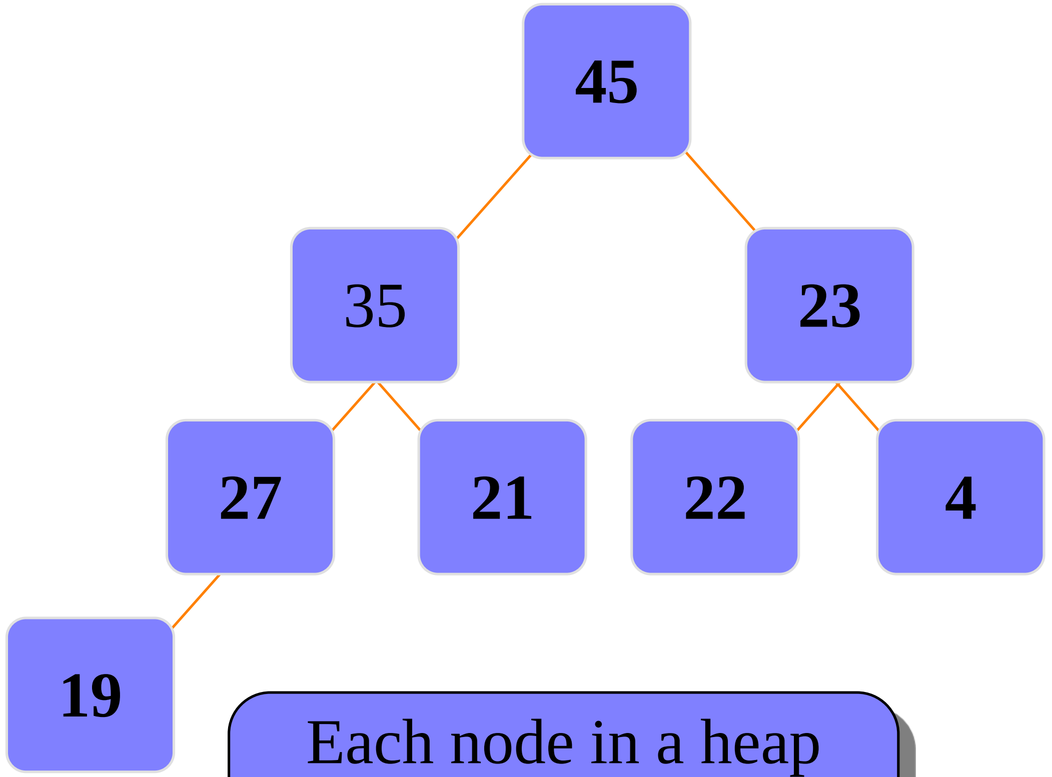
Heaps

Complete binary tree.



Heaps

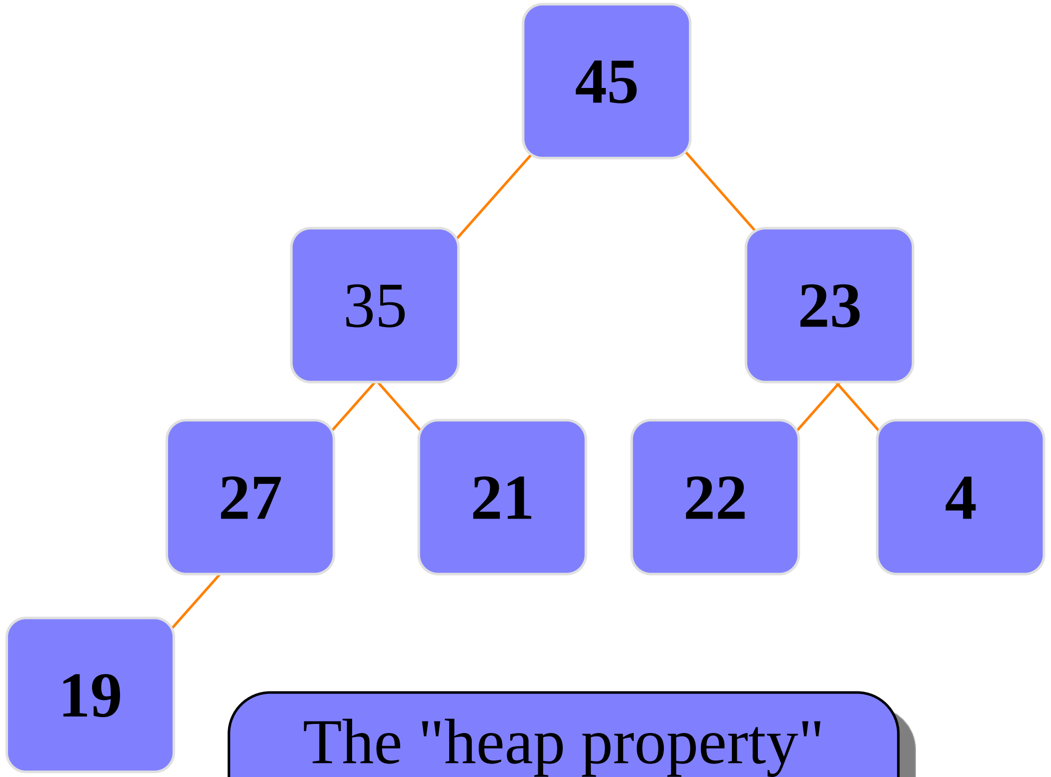
A heap is a **certain** kind of complete binary tree.



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

Heaps

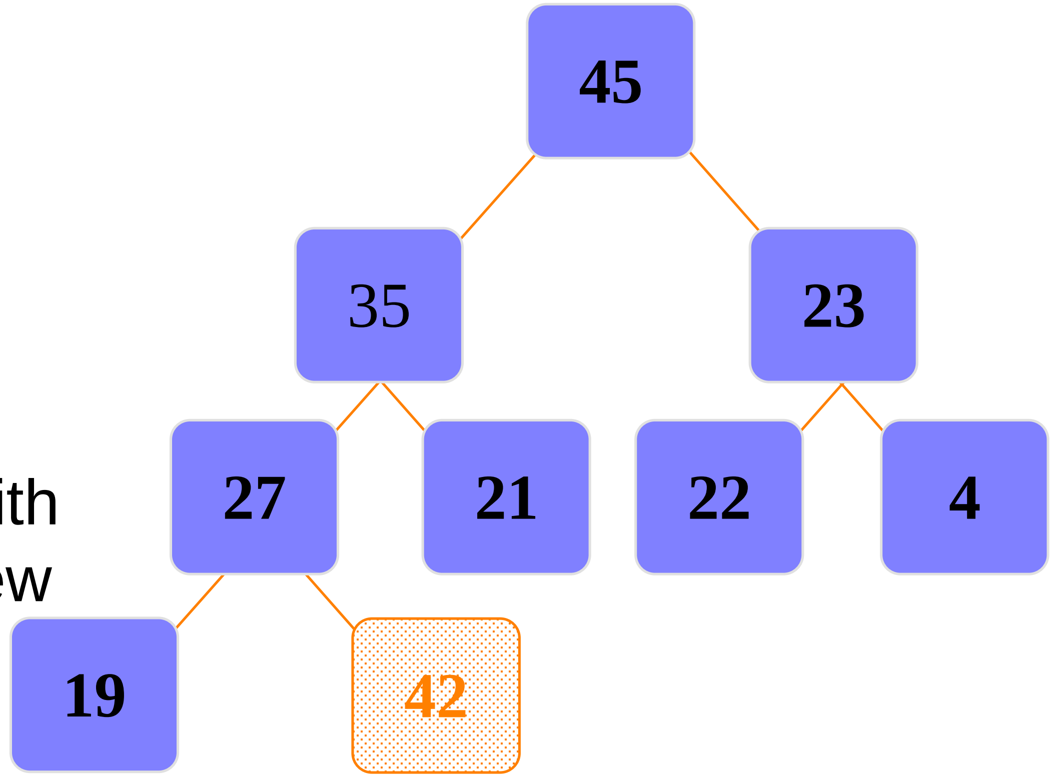
A heap is a **certain** kind of complete binary tree.



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

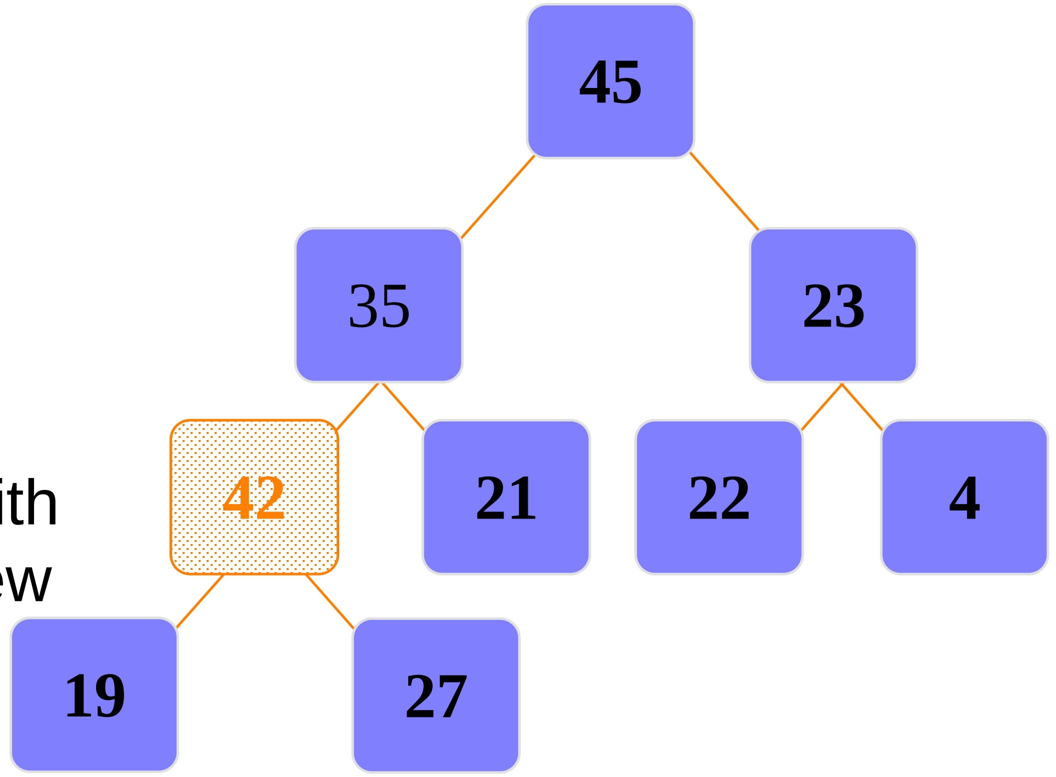
Adding a Node to a Heap

- ❑ 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.



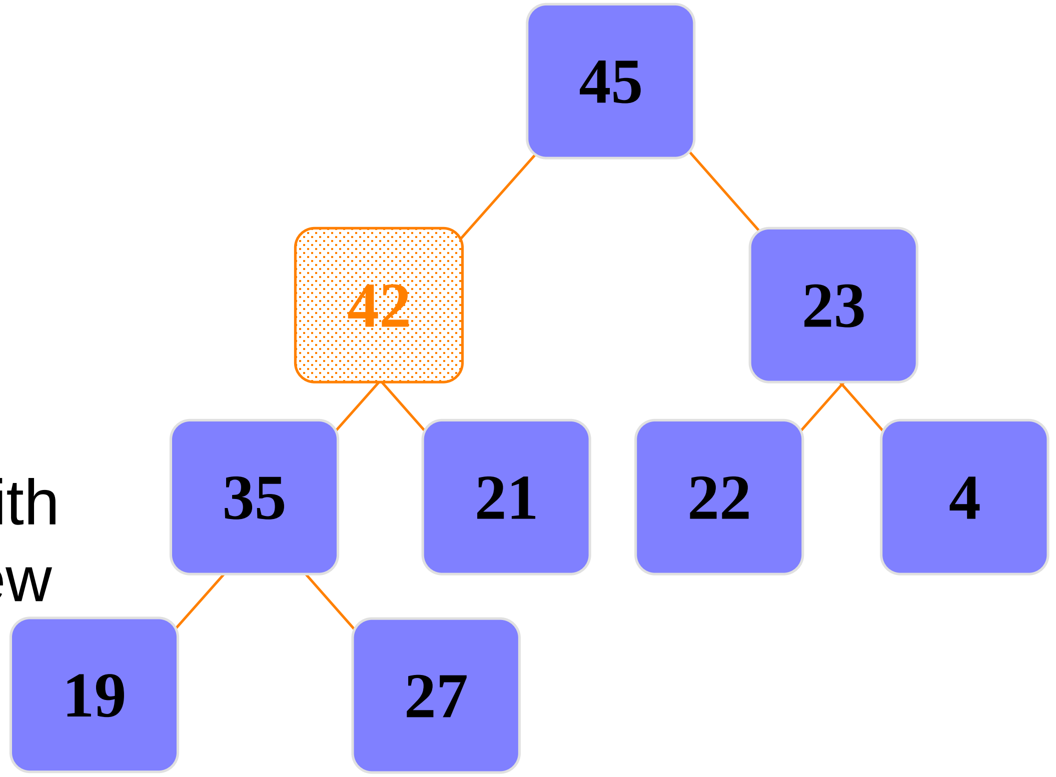
Adding a Node to a Heap

- ❑ 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.



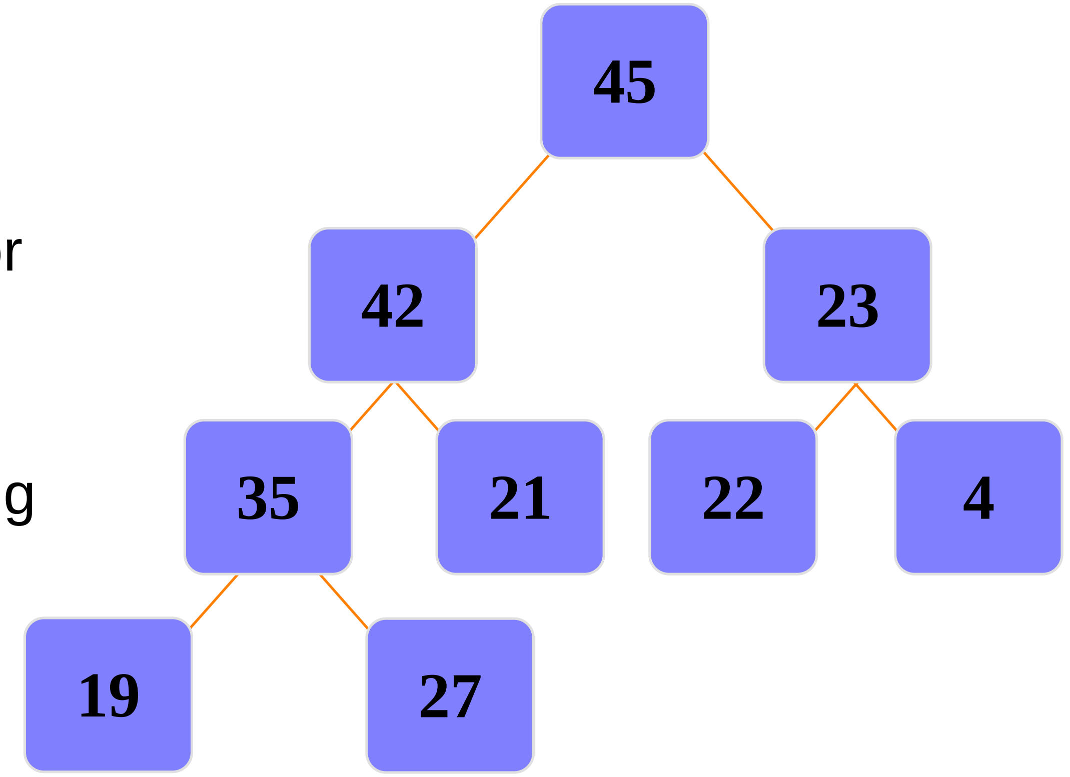
Adding a Node to a Heap

- ❑ 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.



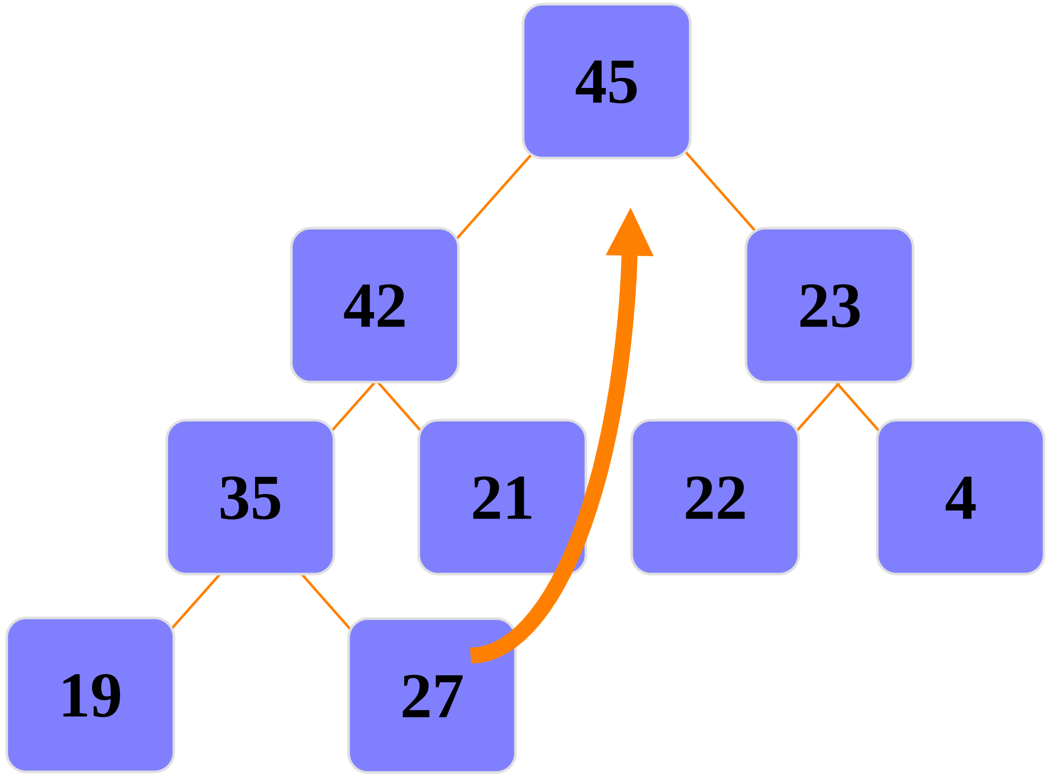
Adding a Node to a Heap

- 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**.



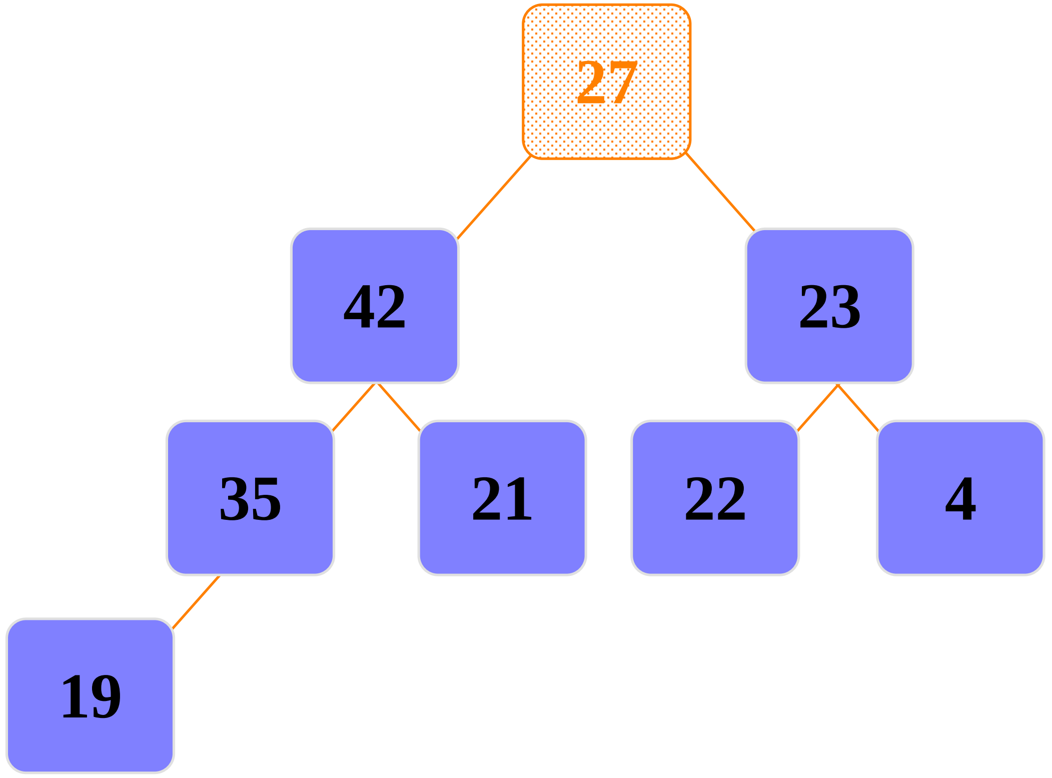
Removing the Top of a Heap

- Move the last node onto the root.



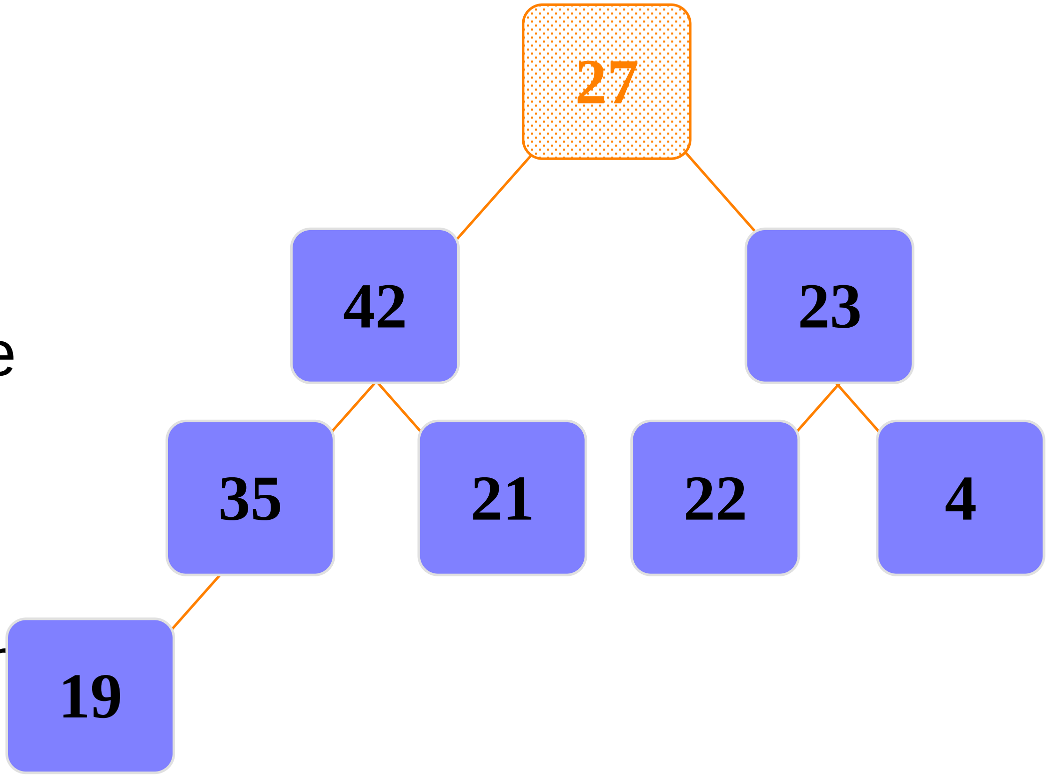
Removing the Top of a Heap

- Move the last node onto the root.



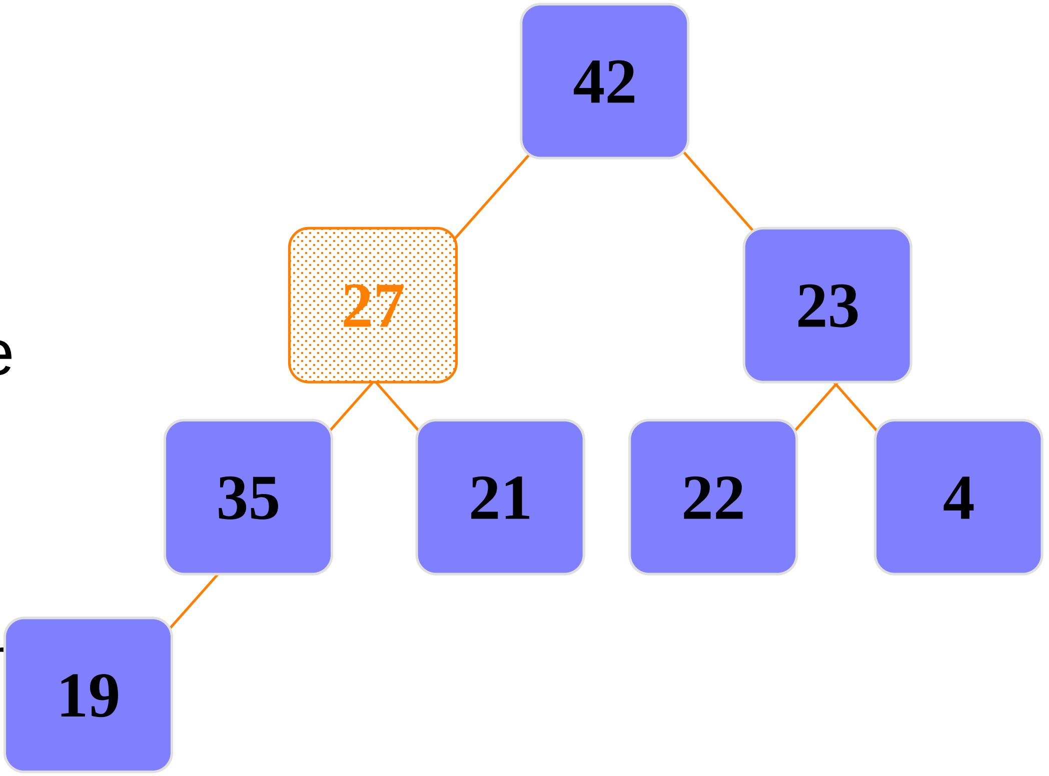
Removing the Top of a Heap

- ❑ 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.



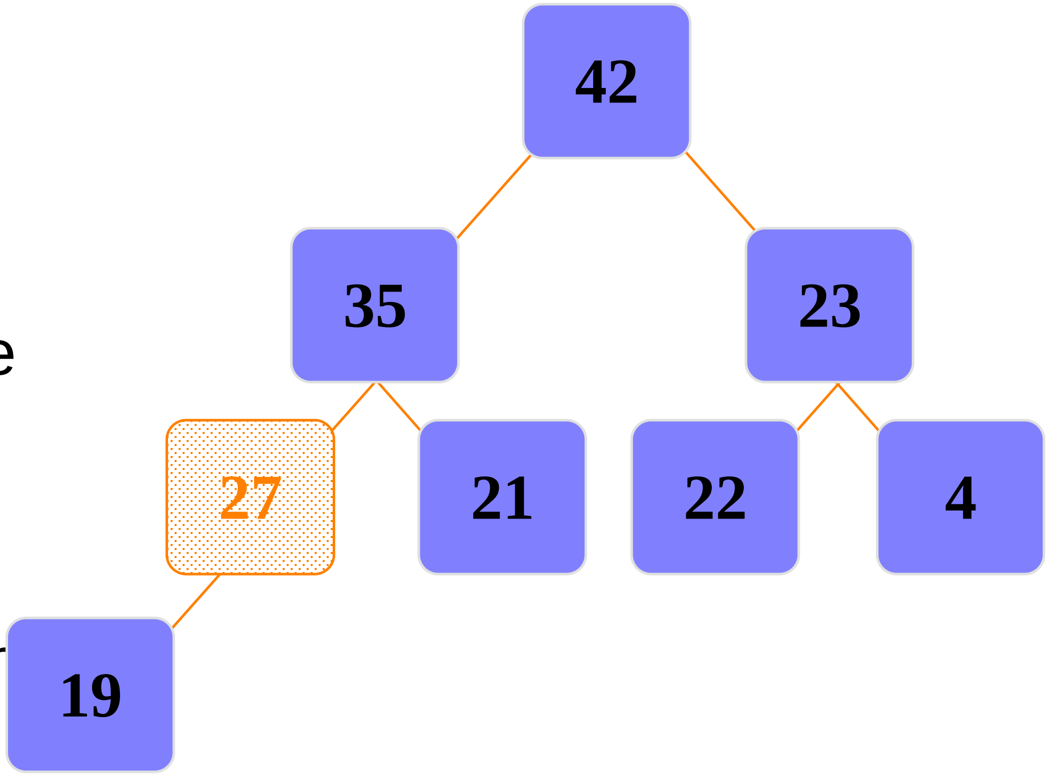
Removing the Top of a Heap

- 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.



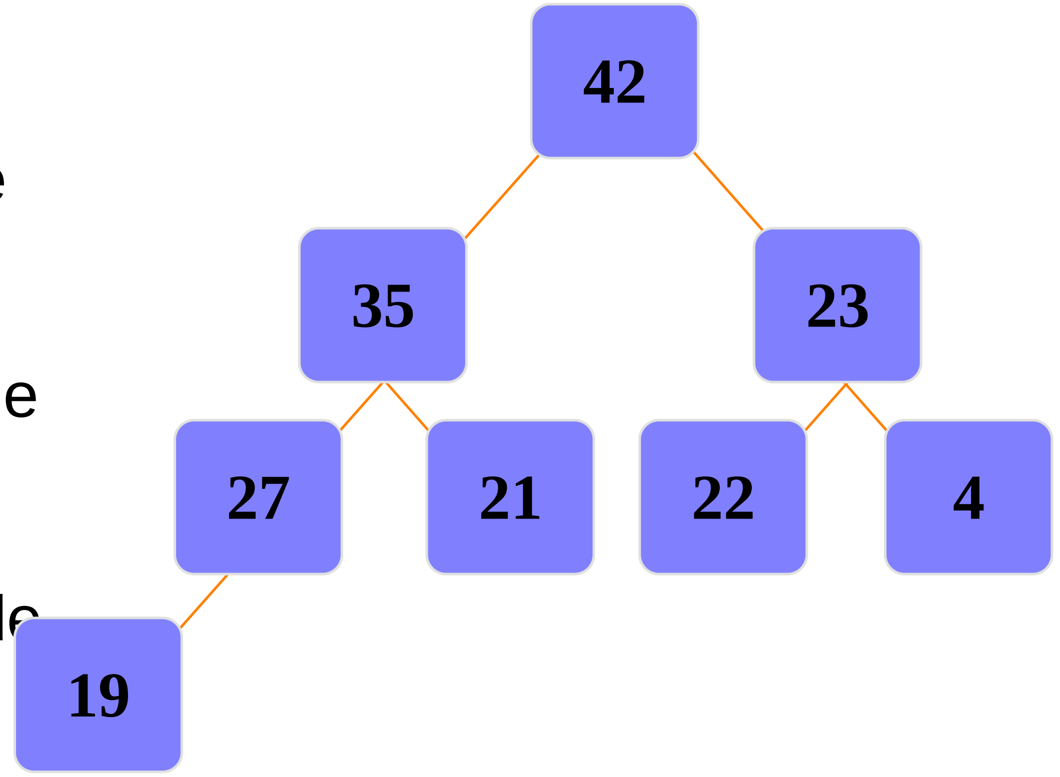
Removing the Top of a Heap

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



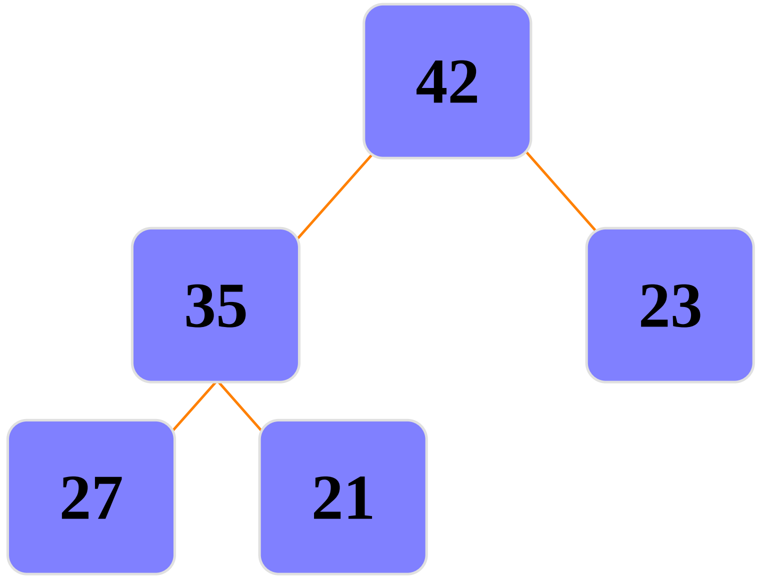
Removing the Top of a Heap

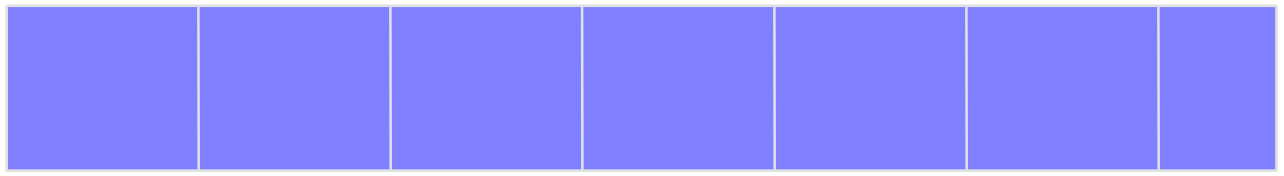
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

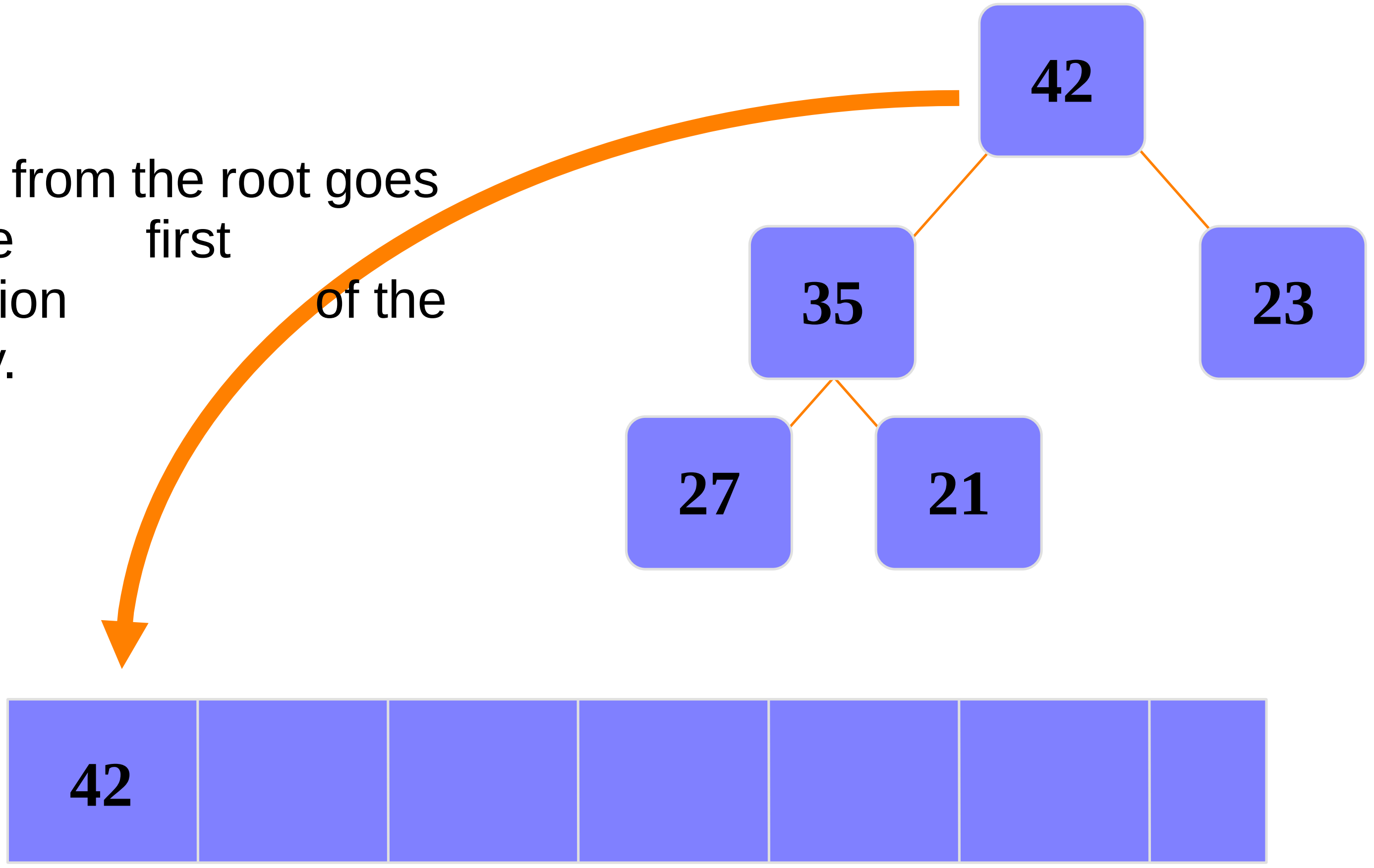
- We will store the data from the nodes in a partially-filled array.





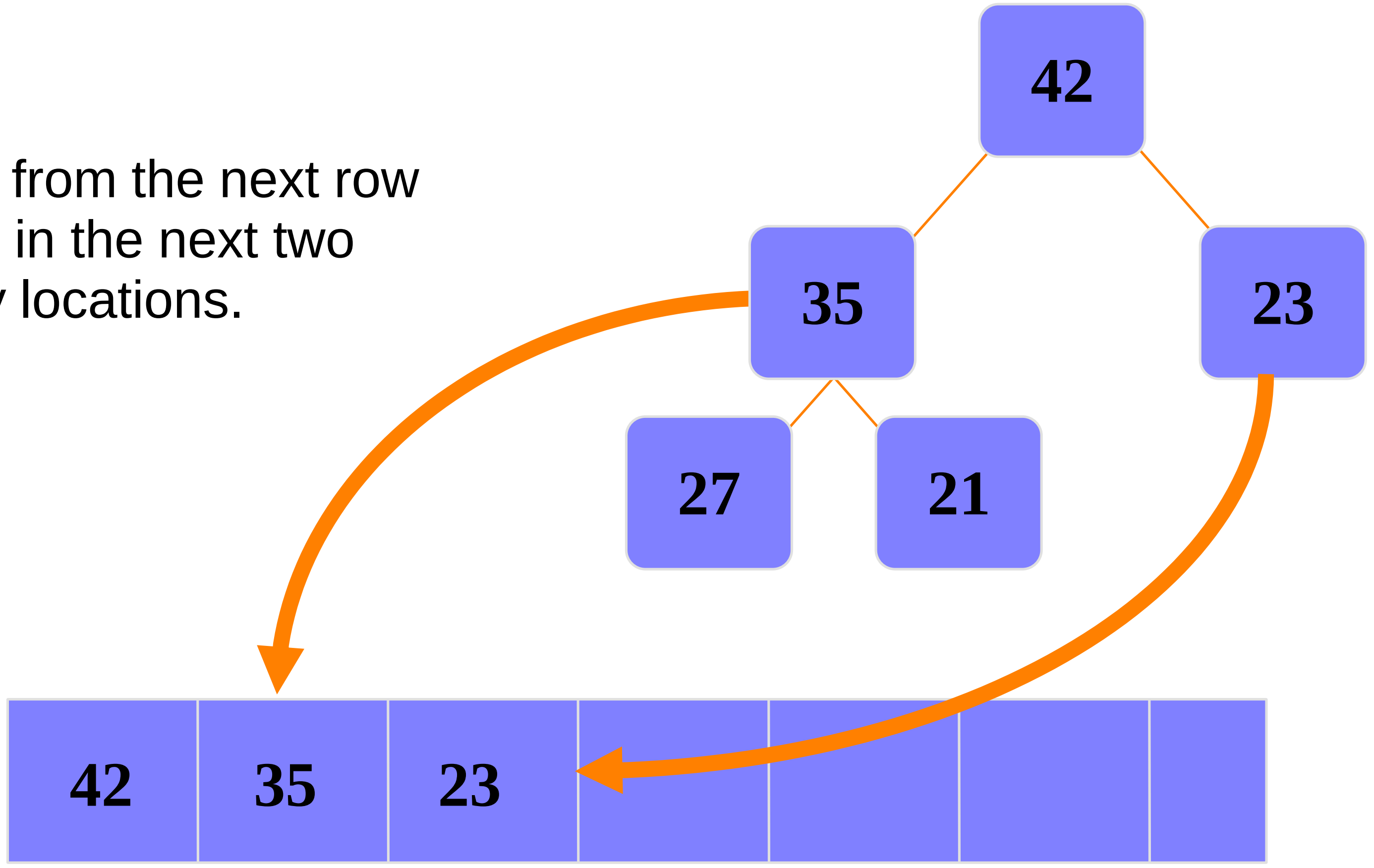
Implementing a Heap

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



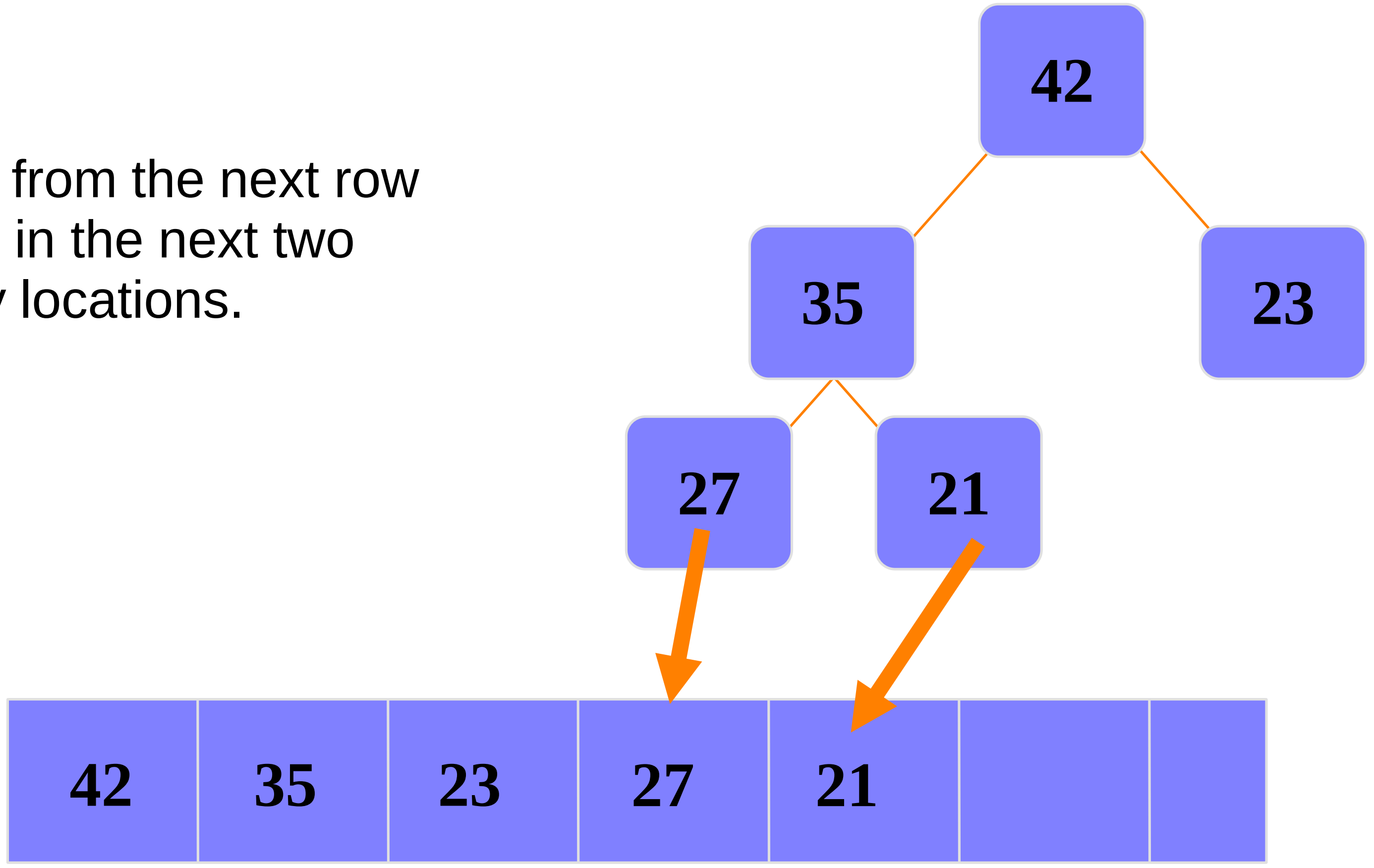
Implementing a Heap

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



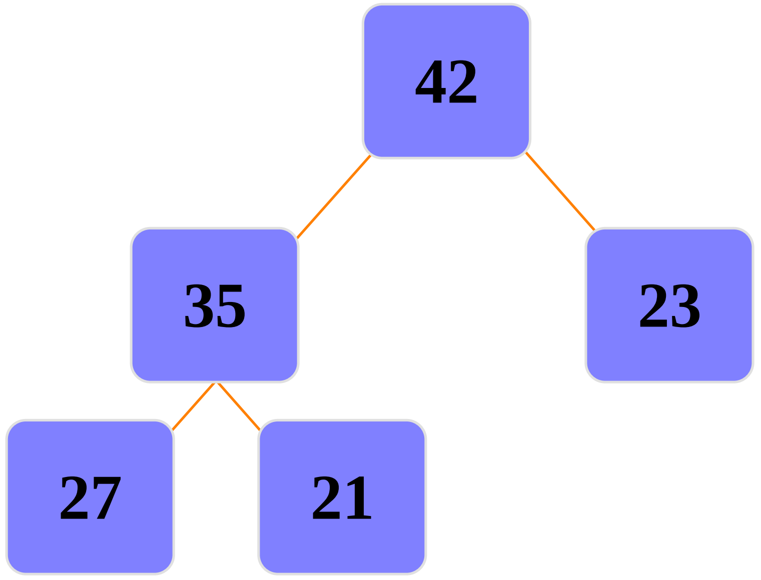
Implementing a Heap

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



Important Points about the Implementation

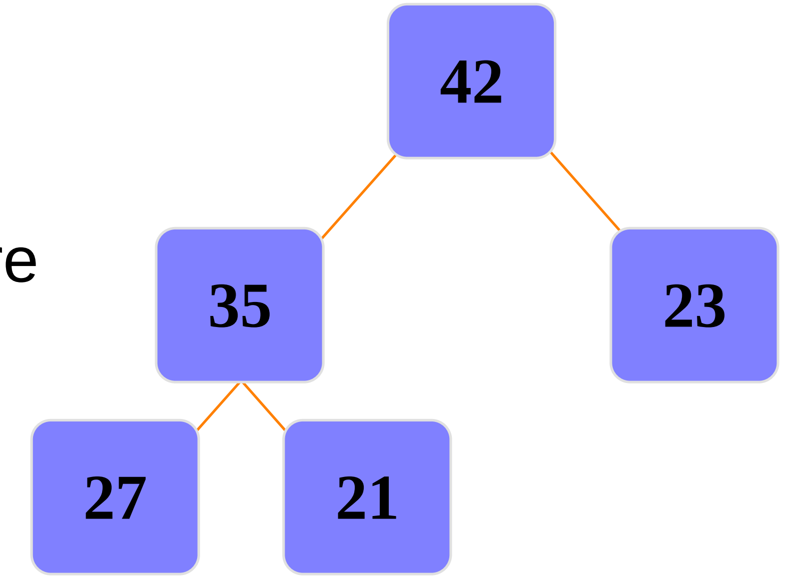
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

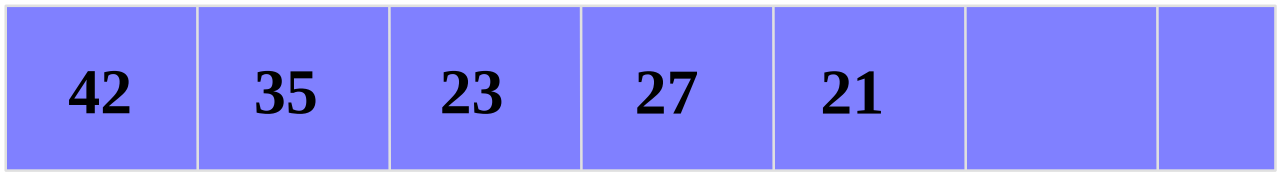


42	35	23	27	21		

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. Formulas are given in the starting slides.





Summary

- **MaxHeap:**
- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

Heap Sort

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap.

Procedures on Heap

- Heapify
- Build Heap
- Heap Sort

HEAPIFY

- Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

BUILD HEAP

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

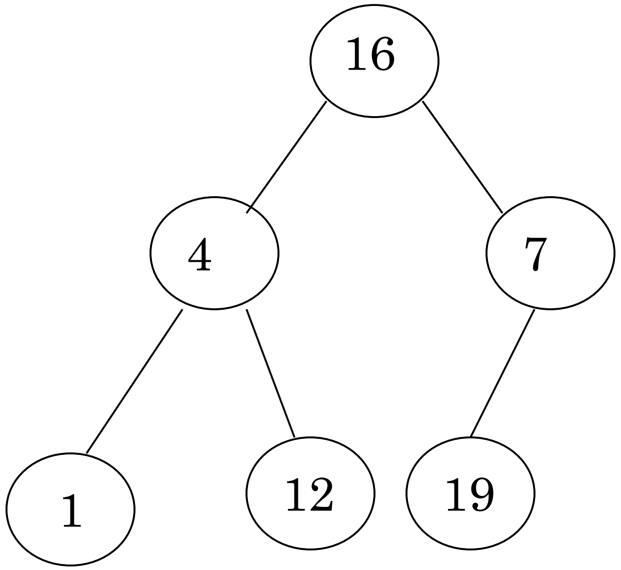
Heap Sort Algorithm

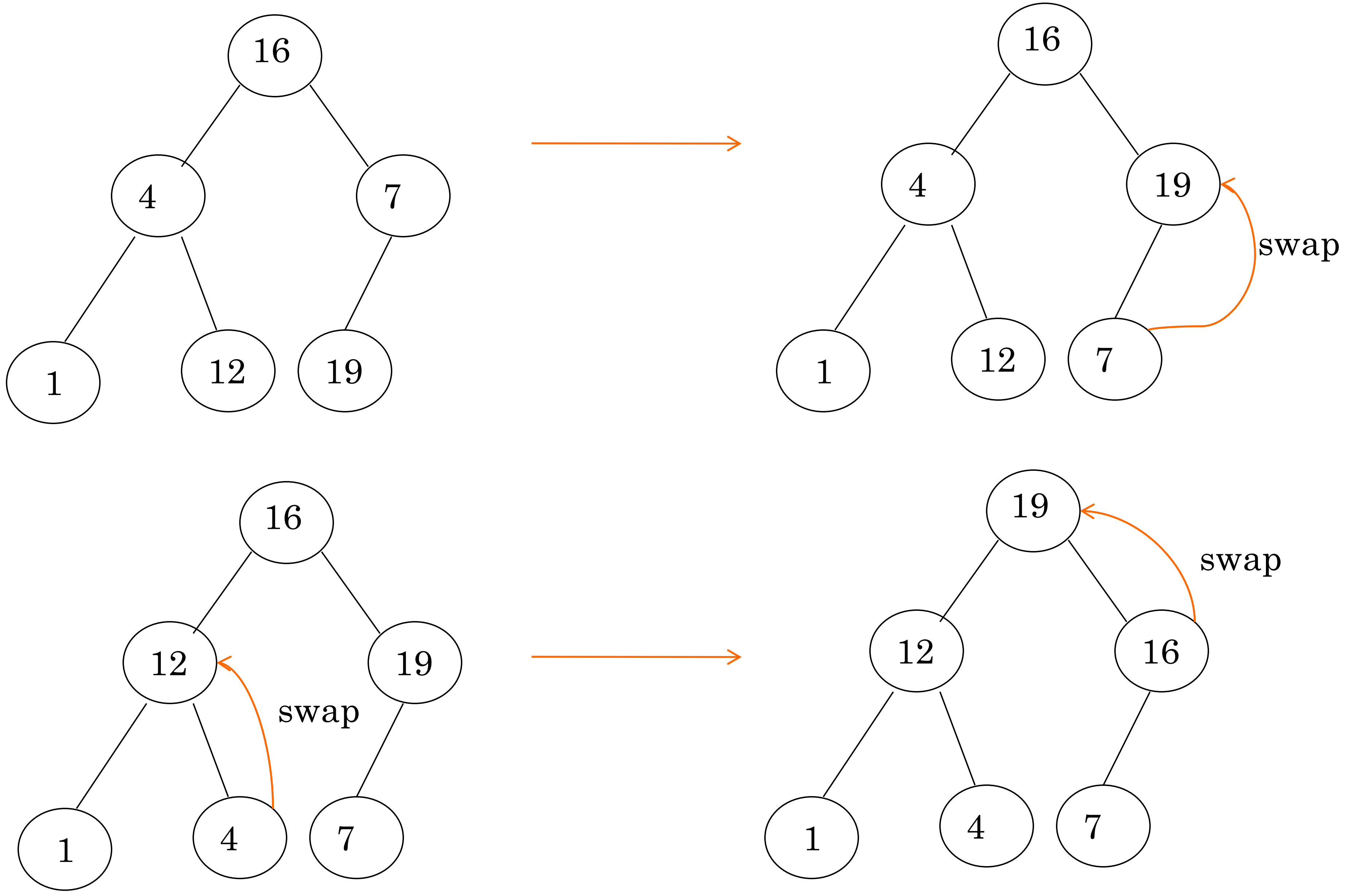
- The heap sort algorithm starts by using procedure **BUILD-HEAP to build a heap on the input array A[1 . . *n*].** Since the maximum element of the array stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$ (the last element in A). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

Example: Convert the following array to a heap

16	4	7	1	12	19

Picture **the array as a complete binary tree:**

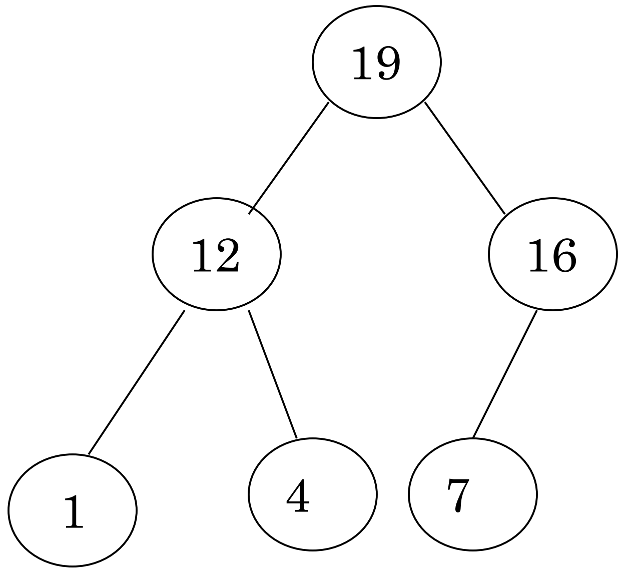



16
4
7
1
12
19
16
4
19
swap
1
12
7
16
12
19
swap
1
4
7
19
12
16
swap
1
4
7

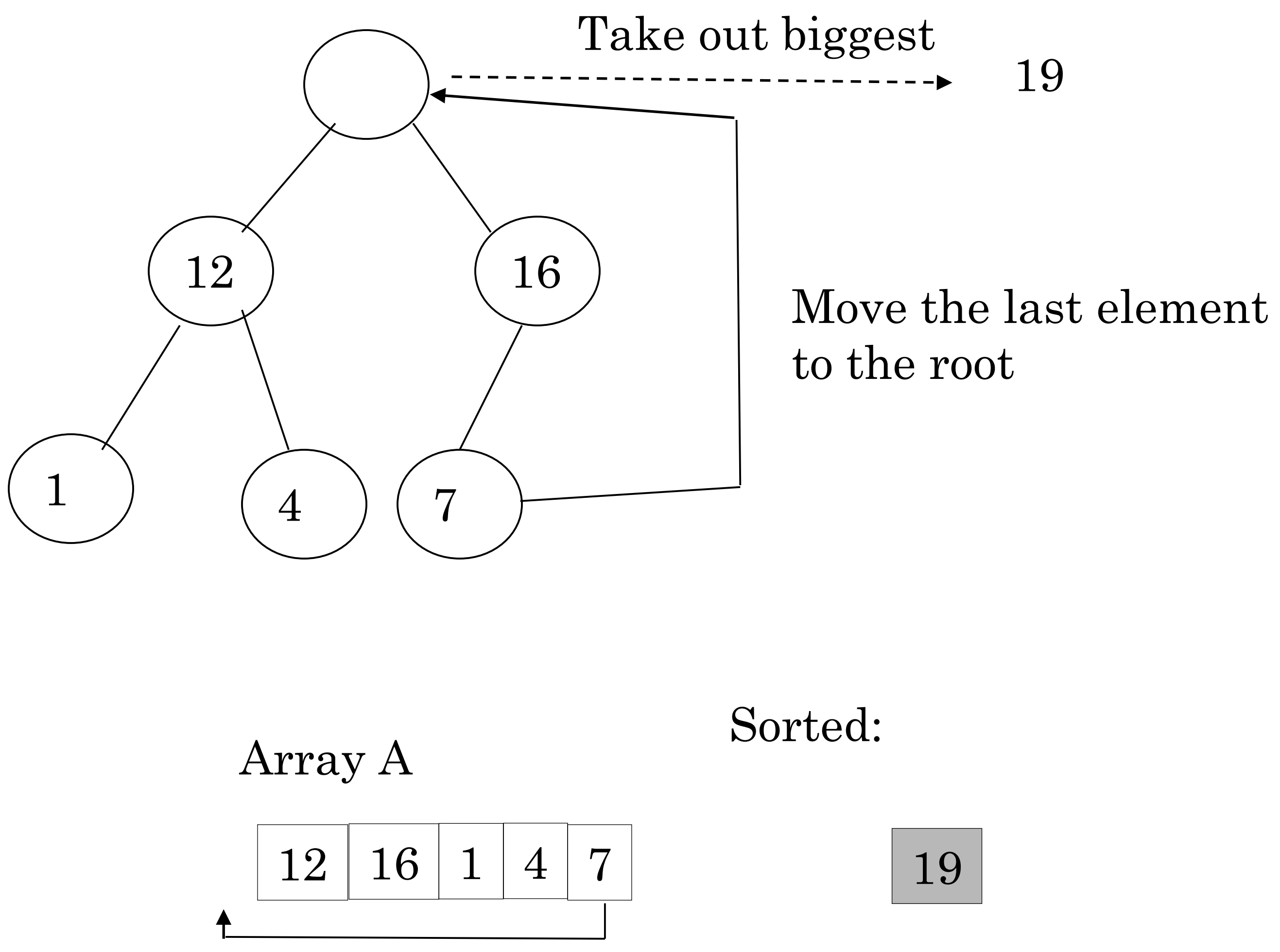
Heap Sort

- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

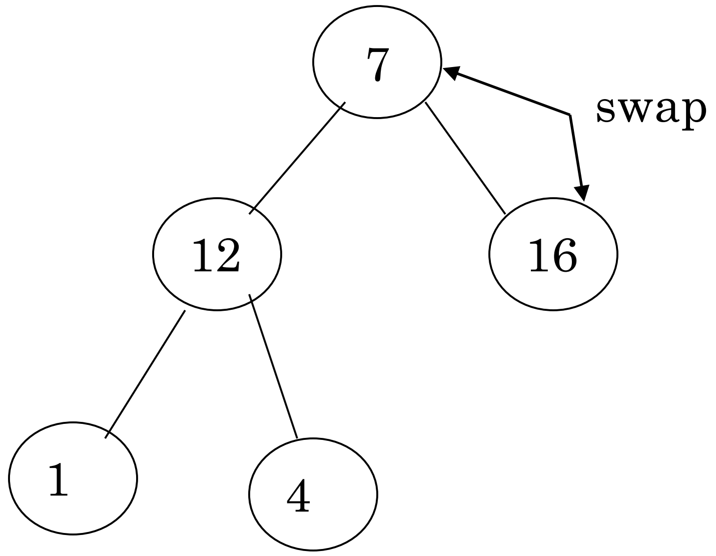
- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



Example of Heap Sort



HEAPIFY()

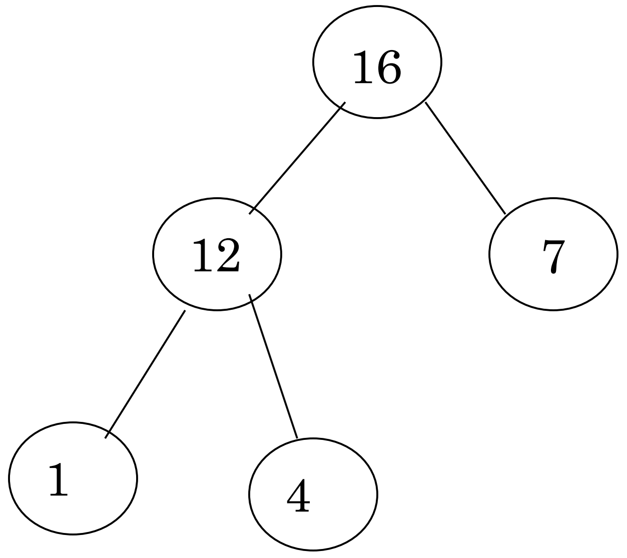


Array A

7	12	16	1	4

Sorted:

19

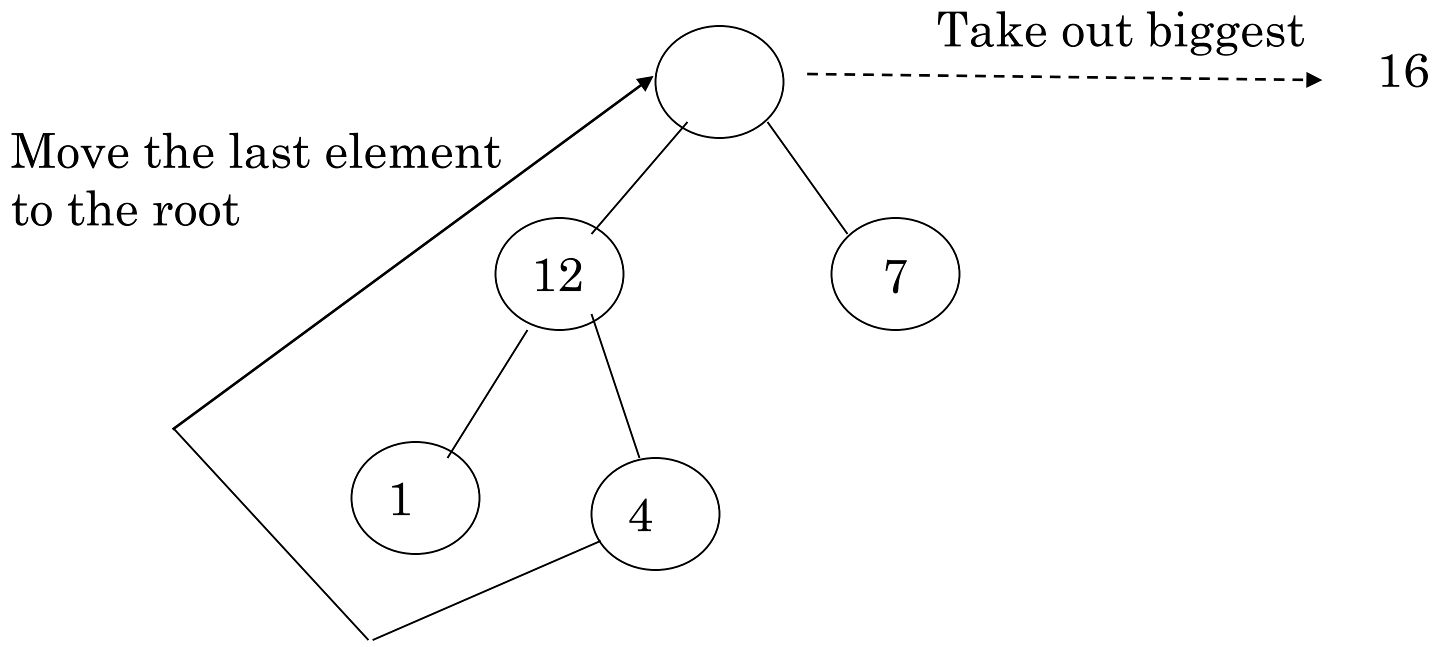


Array A

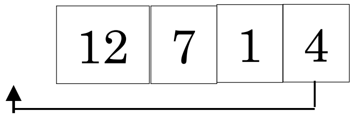
16	12	7	1	4

Sorted:

19

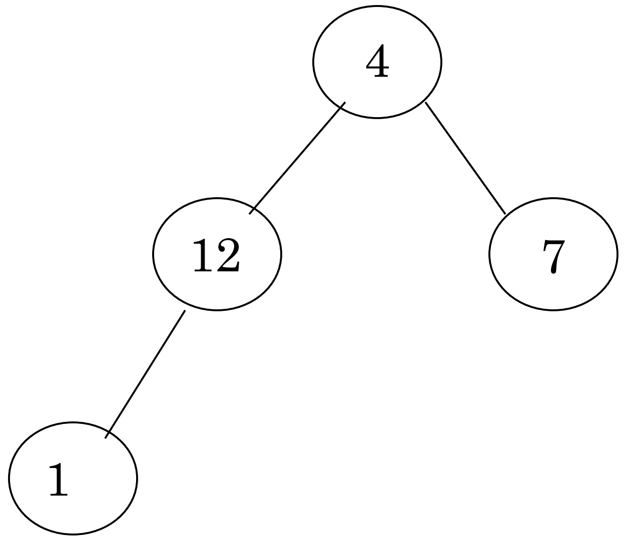


Array A



Sorted:

16	19



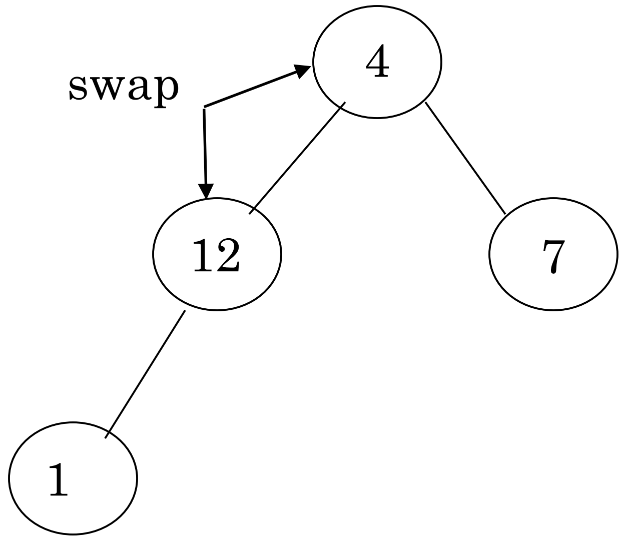
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

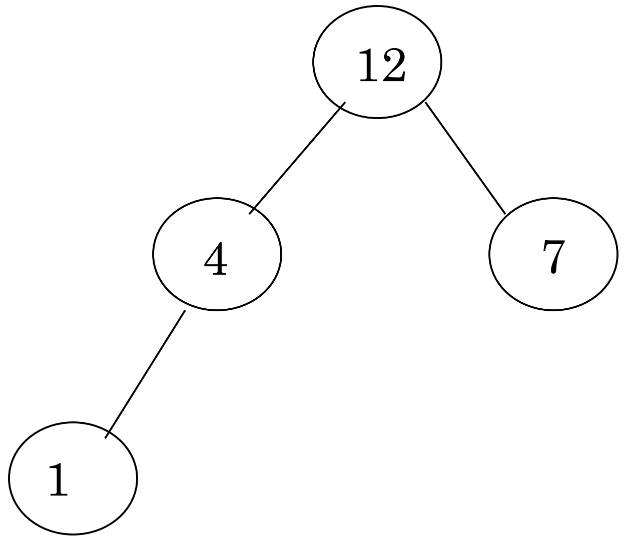


Array A

4	12	7	1

Sorted:

16	19

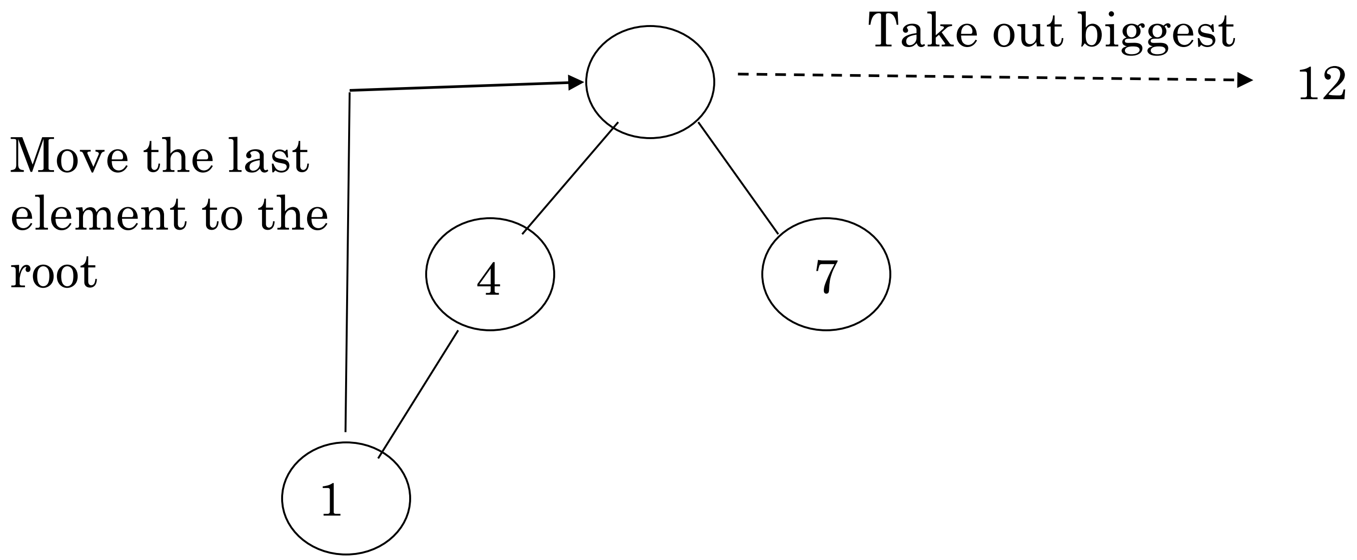


Array A

12	4	7	1

Sorted:

16	19

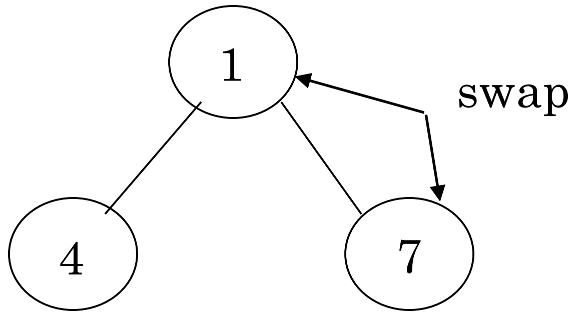


Array A

4	7	1

Sorted:

12	16	19

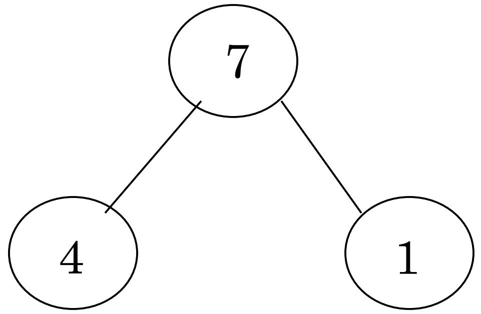


Array A

1	4	7

Sorted:

12	16	19

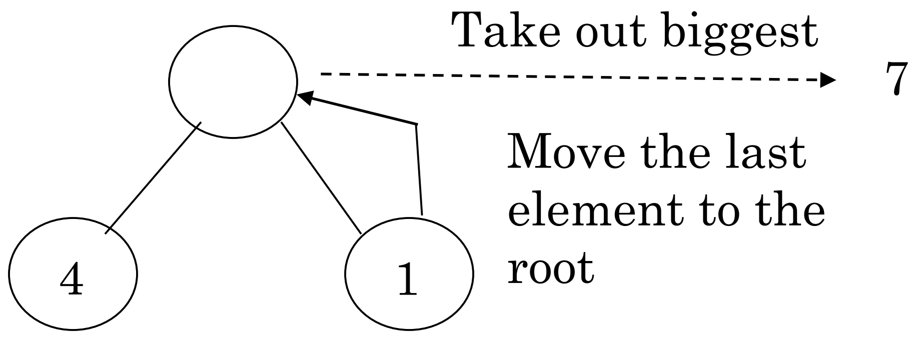


Array A

7	4	1

Sorted:

12	16	19



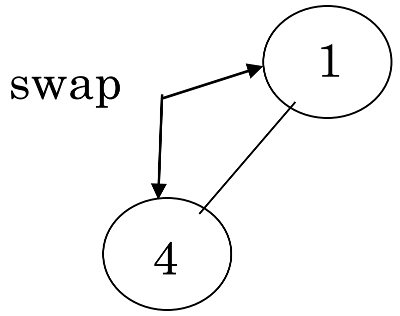
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()

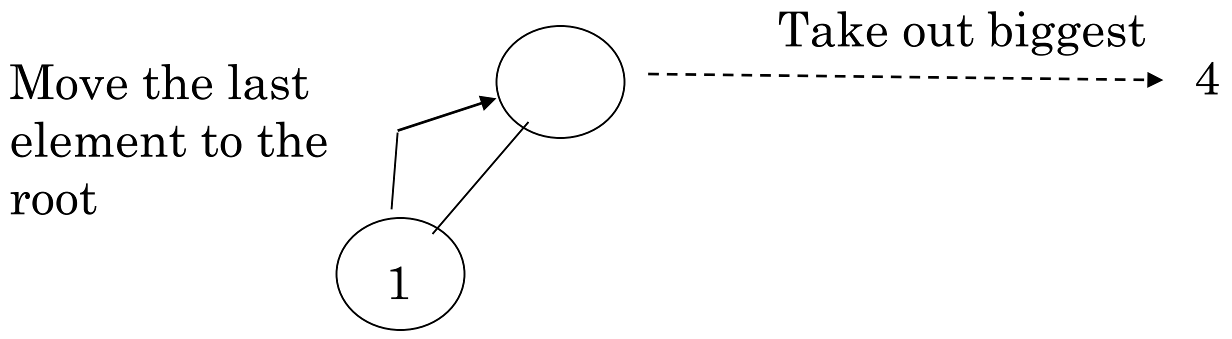


Array A

4	1

Sorted:

7	12	16	19

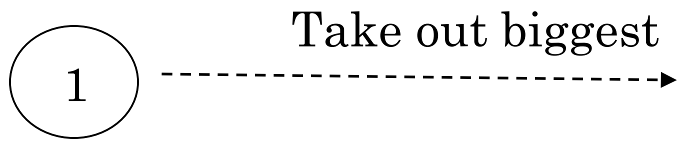


Array A

1

Sorted:

4	7	12	16	19



Sorted:

Array A

1	4	7	12	16	19

Sorted:

1	4	7	12	16	19

Time Analysis

- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

Possible Application

- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a lists of certain task with start and finish times and we want to do as many tasks as possible
- Sorting a list of elements that needs and efficient sorting algorithm

Conclusion

- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is O(n log n). The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.
- The heap sort algorithm has two major steps. The first major step involves transforming the complete tree into a heap. The second major step is to perform the actual sort by extracting the largest element from the root and transforming the remaining tree into a heap.