# Heap Sort,
# Analysis of Heap Sort

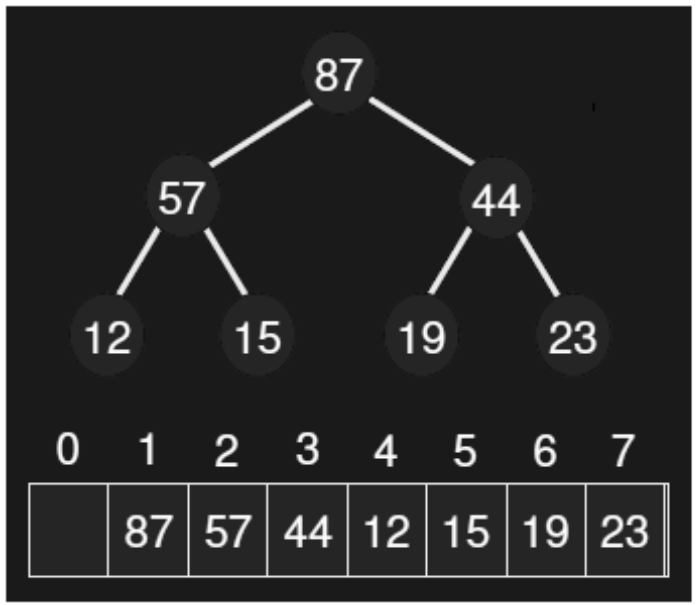
**(Class 12)**

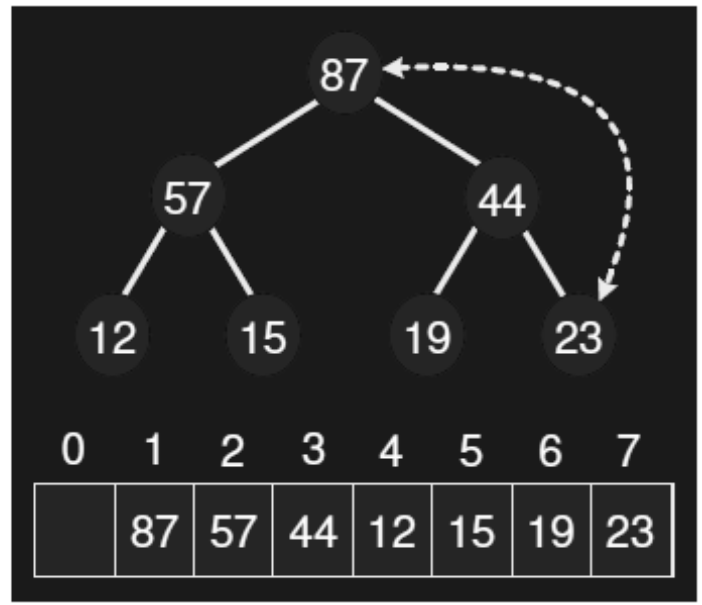
From Book's Page No 161 (Chapter 6)

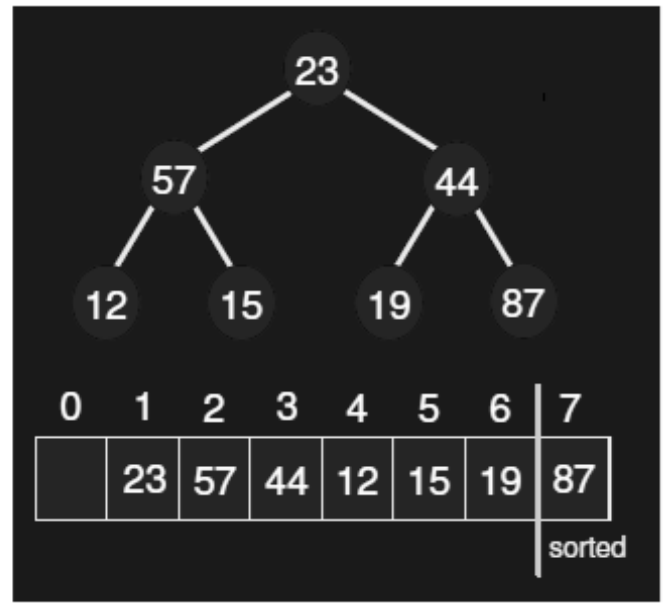
# Heapsort Algorithm

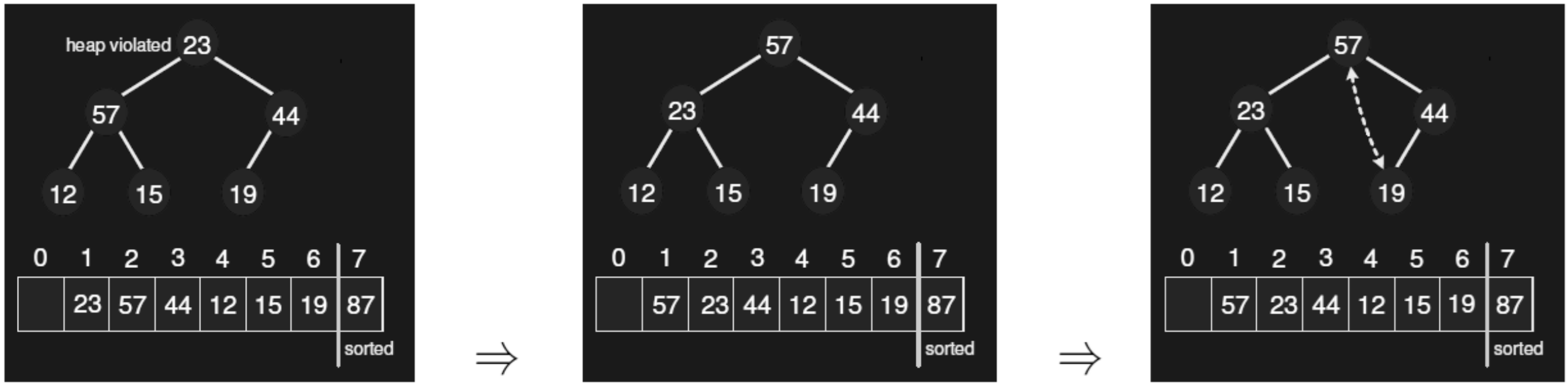
- We build a max heap out of the given array of numbers $A[1 \ldots n]$.
- We repeatedly extract the maximum item from the heap.
- Once the max item is removed, we are left with a hole at the root.
- To fix this, we will replace it with the last leaf in tree.
- But now the heap order will very likely be destroyed.
- We will apply a heapify procedure to the root to restore the heap.

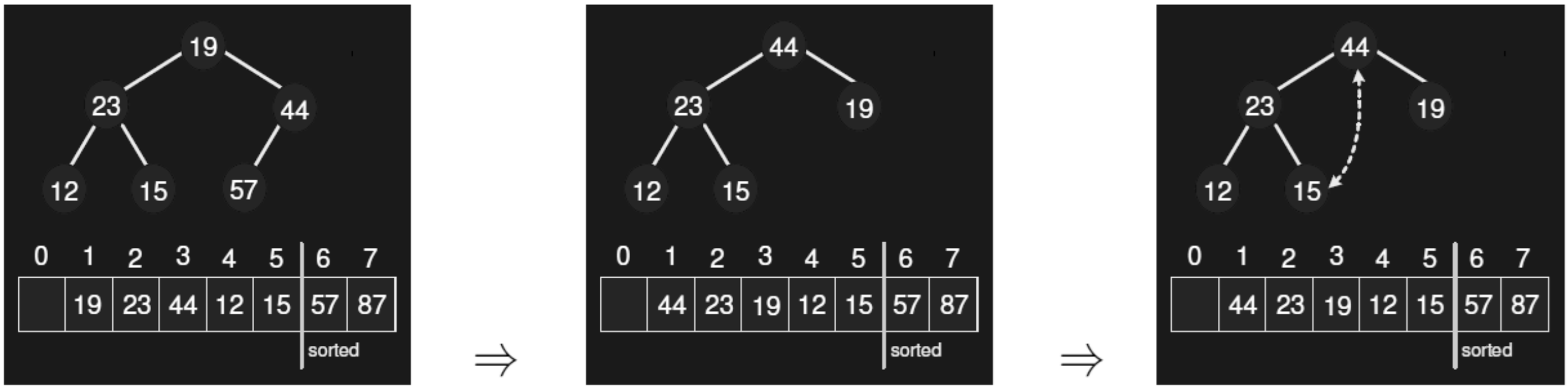
```
HEAPSORT(array A, int n)
1  BUILD-HEAP(A, n)
2  m ← n
3  while (m ≥ 2)
4     do SWAP(A[1], A[m])
5     m ← m-1
6     HEAPIFY(A, 1, m)
```

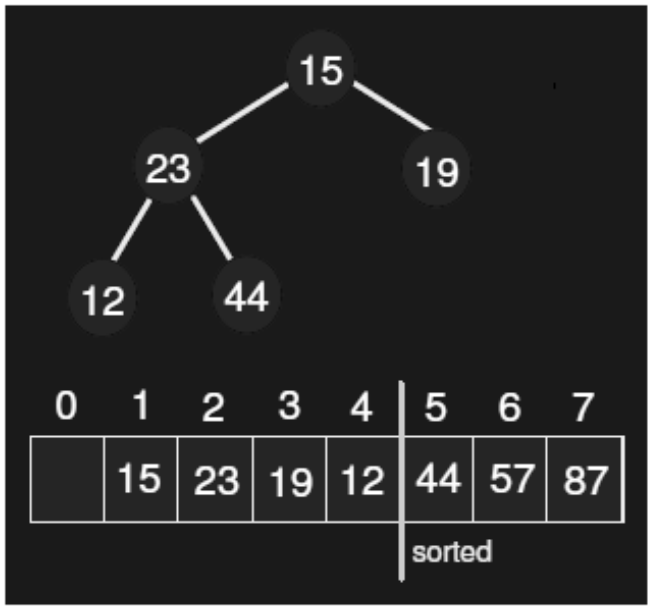

87
57
44
12
15
19
23
0 1 2 3 4 5 6 7
87 57 44 12 15 19 23

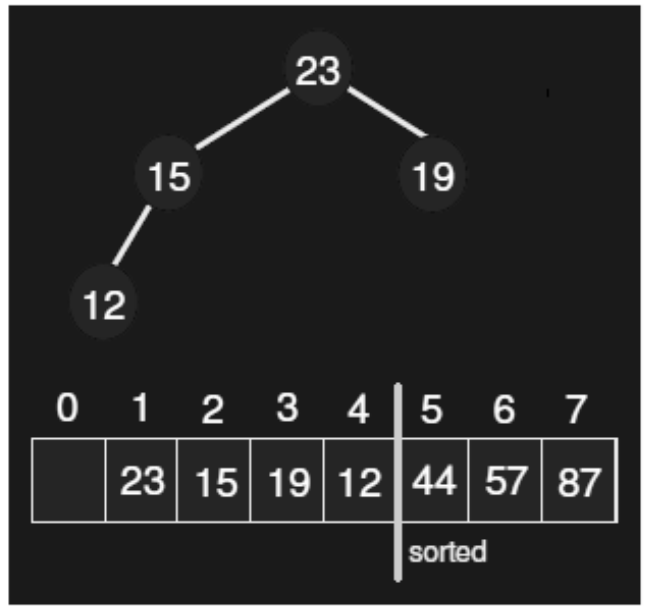

87
57
44
12
15
19
23
0 1 2 3 4 5 6 7
87 57 44 12 15 19 23

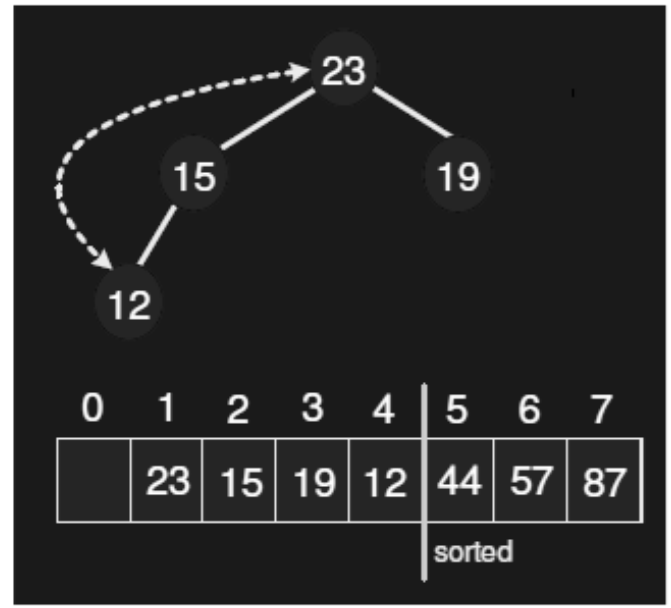

23
57
44
12
15
19
87
0 1 2 3 4 5 6 7
23 57 44 12 15 19 87
sorted

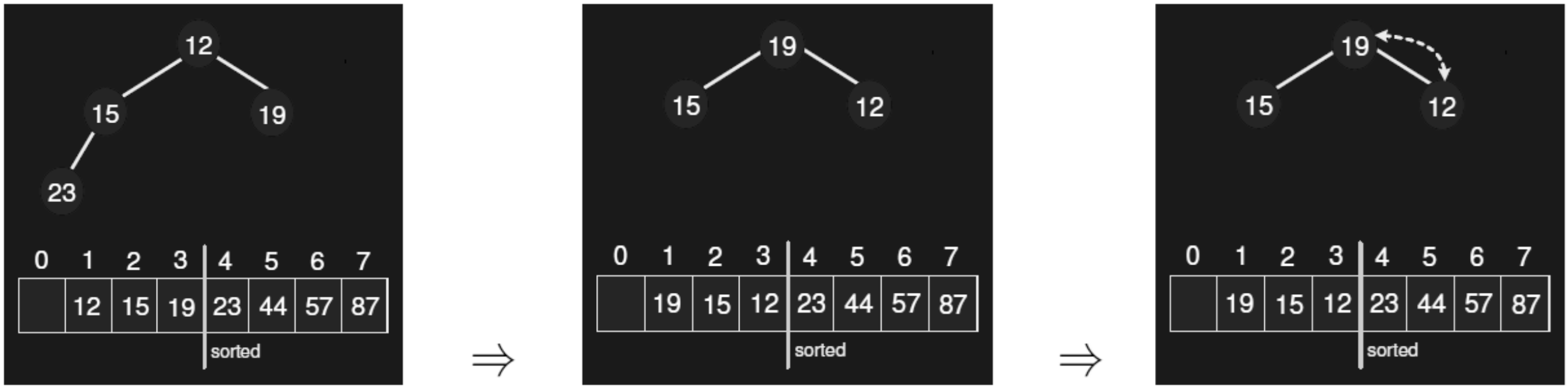

heap violated
23
57
44
12
15
19
0 1 2 3 4 5 6 7
23 57 44 12 15 19 87
sorted
57
23
44
12
15
19
0 1 2 3 4 5 6 7
57 23 44 12 15 19 87
sorted
57
23
44
12
15
19
0 1 2 3 4 5 6 7
57 23 44 12 15 19 87
sorted

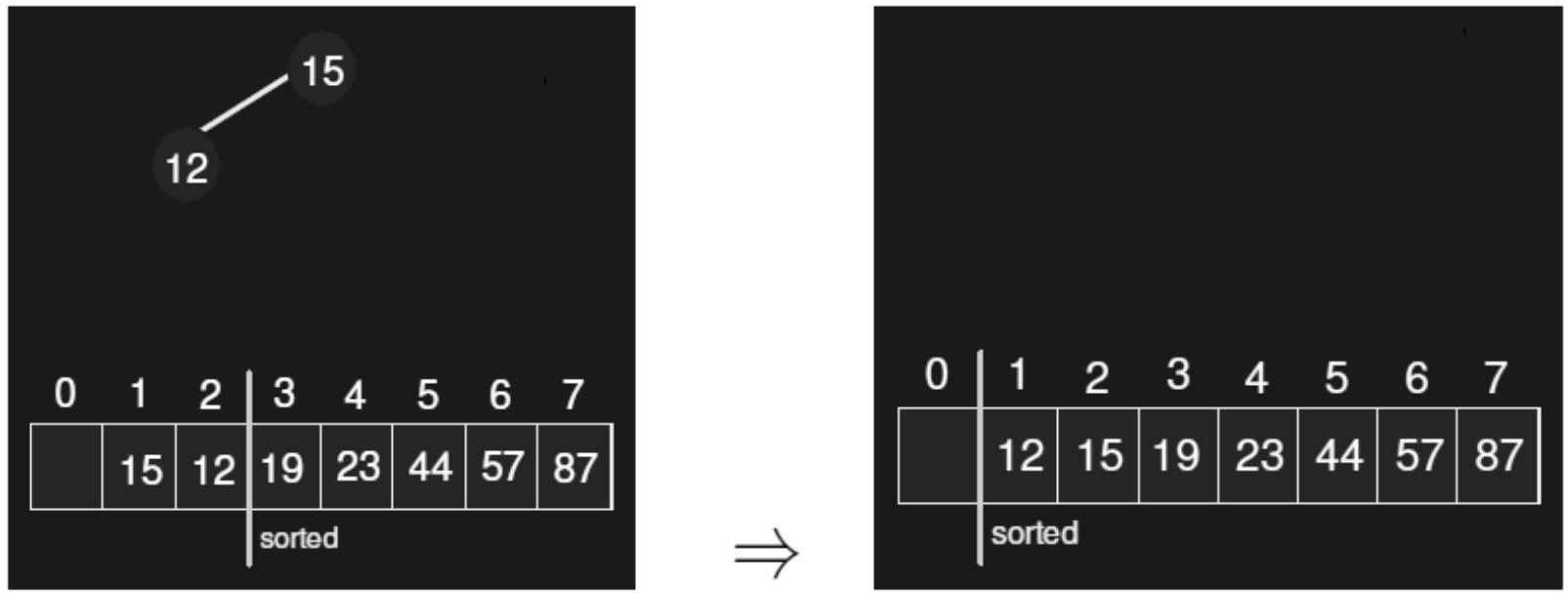

19
23
44
12
15
57
0 1 2 3 4 5 6 7
19 23 44 12 15 57 87
sorted
⇒
44
23
19
12
15
0 1 2 3 4 5 6 7
44 23 19 12 15 57 87
sorted
⇒
44
23
19
12
15
0 1 2 3 4 5 6 7
44 23 19 12 15 57 87
sorted


15
23
19
12
44
0 1 2 3 4 5 6 7
15 23 19 12 44 57 87
sorted


23
15
19
12
0 1 2 3 4 5 6 7
23 15 19 12 44 57 87
sorted


23
15
19
12
0 1 2 3 4 5 6 7
23 15 19 12 44 57 87
sorted


12
15
19
23
0 1 2 3 4 5 6 7
12 15 19 23 44 57 87
sorted
19
15
12
0 1 2 3 4 5 6 7
19 15 12 23 44 57 87
sorted
19
15
12
0 1 2 3 4 5 6 7
19 15 12 23 44 57 87
sorted


15
12
0 1 2 3 4 5 6 7
15 12 19 23 44 57 87
sorted
0 1 2 3 4 5 6 7
12 15 19 23 44 57 87
sorted

# Heapify Procedure

- There is one principal operation for maintaining the heap property.
- It is called Heapify (also called sifting down).
- The idea is that we are given an element of the heap which we suspect may not be in valid heap order.
- But we assume that all of other the elements in the subtree rooted at this element are in heap order.

- In particular this root element may be too small.
- To fix this we “sift” it down the tree by swapping it with one of its children.
- Which child? We should take the larger of the two children to satisfy the heap ordering property.
- This continues recursively until the element is either larger than both its children and until its falls all the way to the leaf level.

- Here is the algorithm.
- It is given the heap in the array $A$, and the index $i$ of the suspected element, and $m$ the current active size of the heap.
- The element $A[max]$ is set to the maximum of $A[i]$ and its two children.
- If $max \neq i$ then we swap $A[i]$ and $A[max]$ and then recurse on $A[max]$.

```
HEAPIFY (array A, int i, int m)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  max ← i
4  if (l ≤ m) and (A[l] > A[max])
5    then max ← l
6  if (r ≤ m) and (A[r] > A[max])
7    then max ← r
8  if (max ≠ i)
9    then SWAP(A[i],A[max])
10 HEAPIFY(A,max,m)
```

# Analysis of Heapify

- We call heapify on the root of the tree.
- The maximum levels an element could move up is $O(\log n)$ levels. (Due to complete binary tree)
- At each level, we do simple comparison which is $O(1)$.
- The total time for heapify is thus:

$$1.\log n = O(\log n)$$

- And the best-case running time of this heapify procedure is $O(1)$.

# BuildHeap Procedure

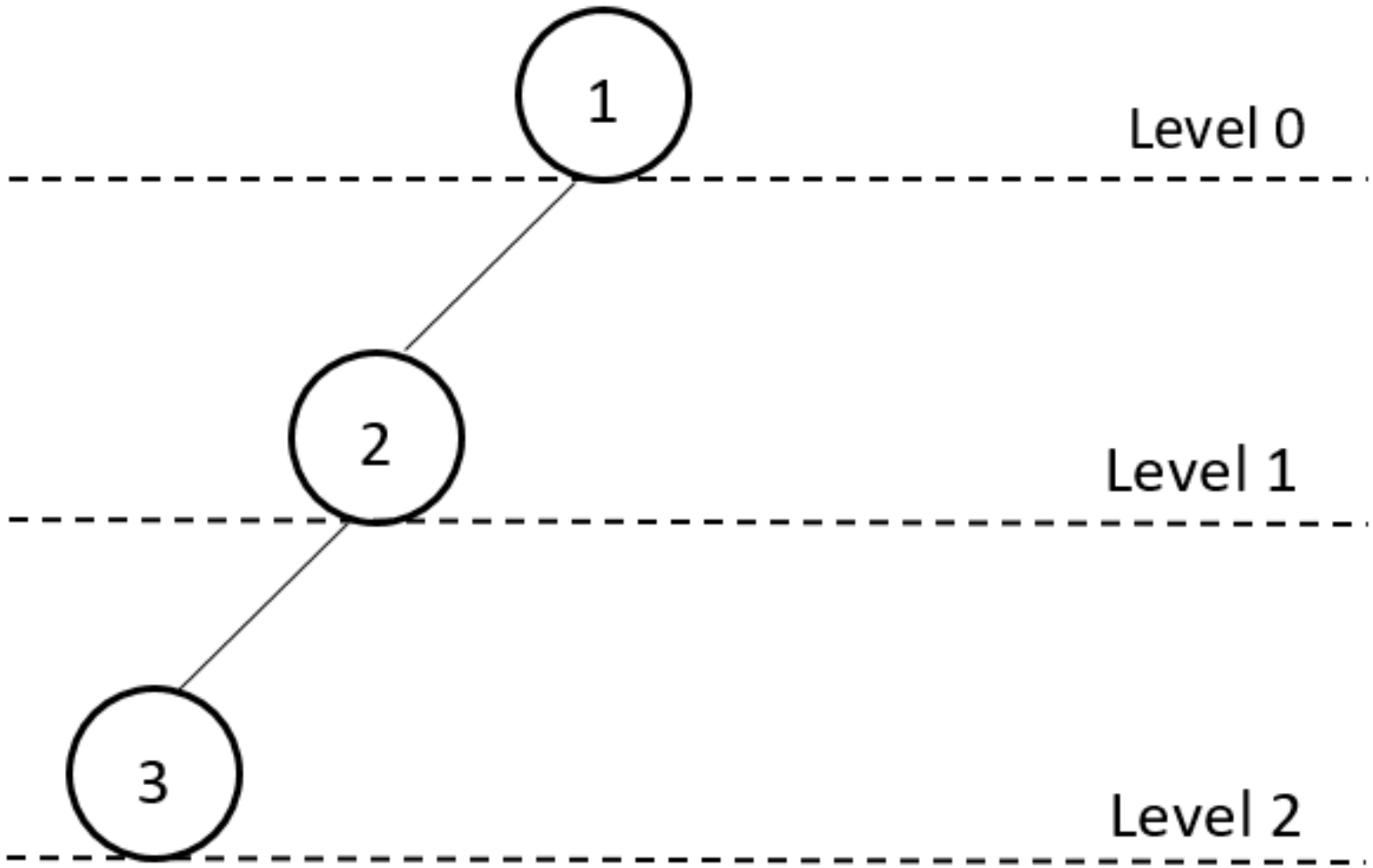
- We can use Heapify to build a heap as follows:
- First, we start with a heap in which the elements are not in heap order.
- They are just in the same order that they were given to us in the array $A$.
- We build the heap by starting at the leaf level and then invoke Heapify on each node.

- Note: We cannot start at the top of the tree.
- Because the precondition which Heapify assumes is that the entire tree rooted at node $i$ is already in heap order, except for $i$.
- Actually, we can be a bit more efficient. Since we know that each leaf is already in heap order, we may as well skip the leaves and start with the first non-leaf node.
- This will be in position $\frac{n}{2}$.

- Here is the algorithm.
- Since we will work with the entire array, the parameter $m$ for Heapify, which indicates the current heap size will be equal to $n$, the size of array $A$, in all the calls.

```
BUILDHEAP (array A, int n)
1  for I ← n/2 downto 1
2     do
3        HEAPIFY(A, i, n)
```

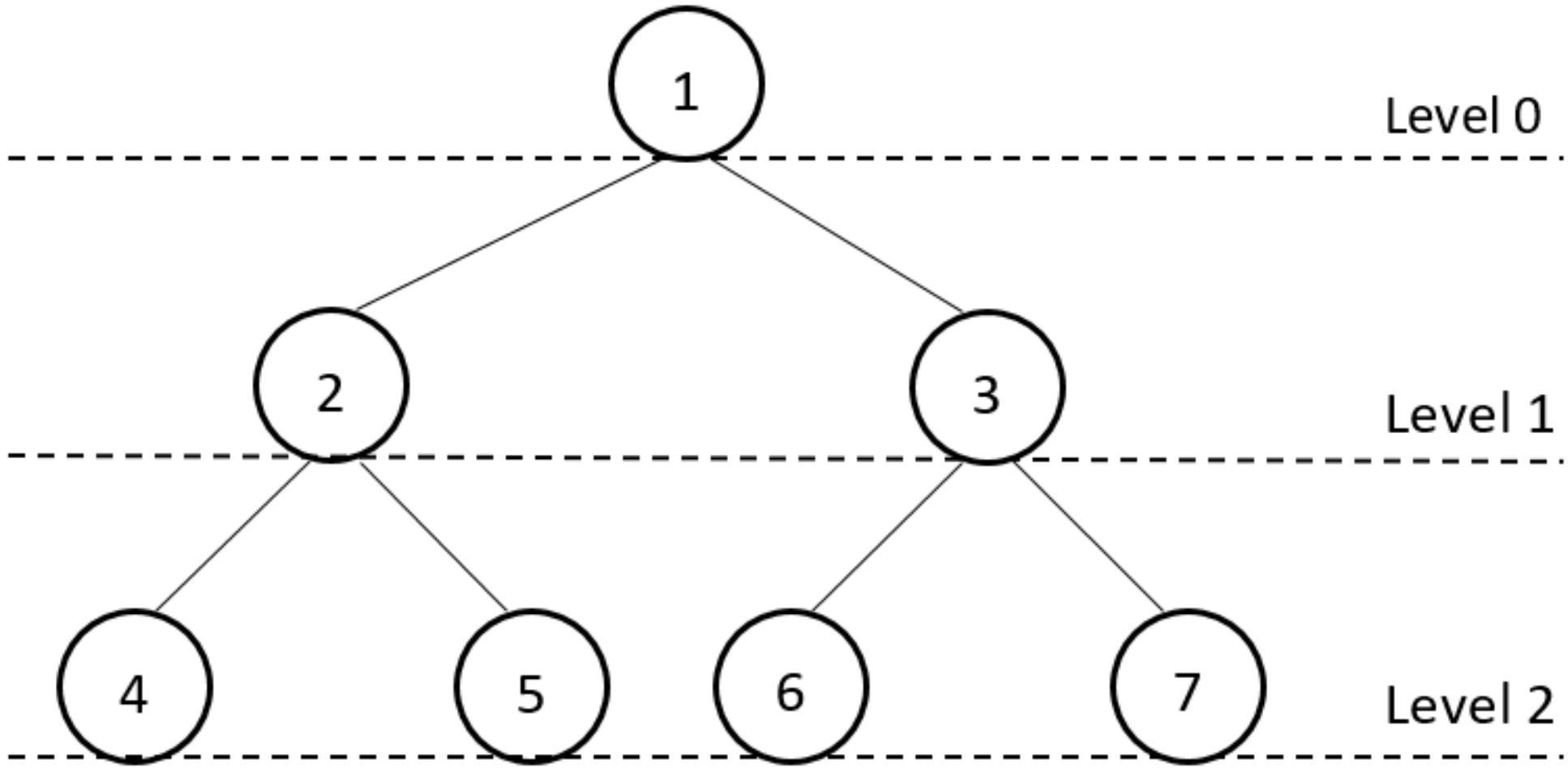
# Minimum Number of Nodes in Binary Tree



- It’s easy to see that we need at least one node for each level to construct a binary tree with level $n$.
- Therefore, the minimum number of nodes of a binary tree with level $n$ is $n + 1$.
- This binary tree behaves like a linked list data structure.

# Maximum Number of Nodes in Binary Tree

- To construct a binary tree of level $n$ with the maximum number of nodes, we need to make sure all the internal nodes have two children.
- In addition, all the leaf nodes must be at the level $n$.


1
Level 0
2
3
Level 1
4
5
6
7
Level 2

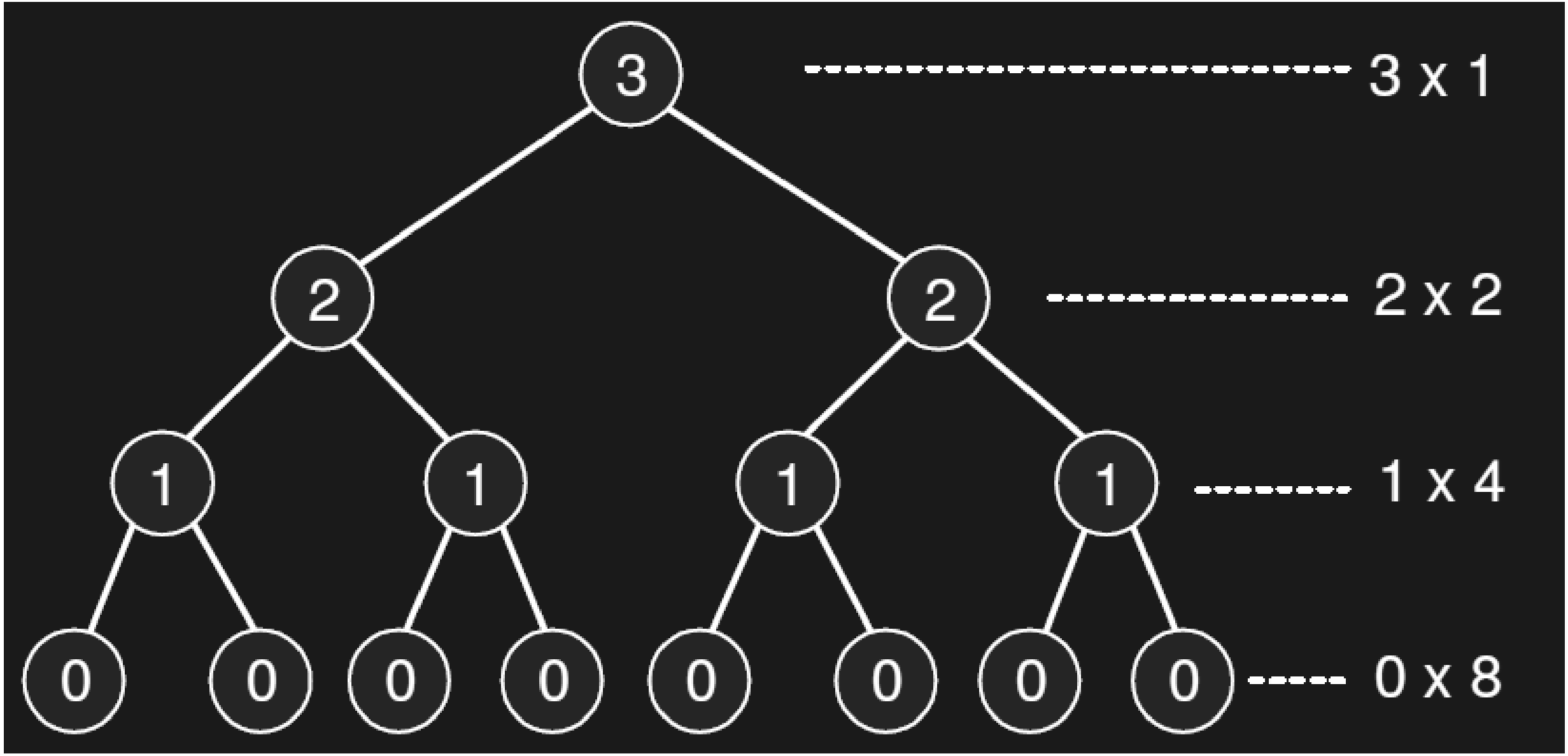
- Based on this observation, we can see that each level doubles the number of nodes from its previous level.
- This is because every internal node has two children.
- Therefore, the maximum number of nodes of a level $n$ binary tree is:

$$1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

- We also call this type of binary tree a full binary tree.

# Analysis of BuildHeap Procedure

- For convenience, we will assume $n = 2^{h+1} - 1$
- where $h$ is the height of tree.
- The heap is a left-complete binary tree.
- Thus, at each level $j$, $j < h$, there are $2^j$ nodes in the tree.
- At level $h$, there will be $2^h$ or less nodes.
- How much work does buildHeap carry out?
- Here for the convenience, we call the bottom most level as 0 level and then increase the level numbering towards top levels.


3
2
2
1
1
1
1
0
0
0
0
0
0
0
0
3 x 1
2 x 2
1 x 4
0 x 8

- At the bottom most level, there are $2^h$ nodes but we do not heapify these.
- At the next level up, there are $2^{h-1}$ nodes and each might shift down 1.
- In general, at level $j$, there are $2^{h-j}$ nodes and each may shift down $j$ levels.
- In simple words, how many nodes are there in a level and how much each node can go downwards?

- So, if count from bottom to top, level-by-level, the total time is:

$$T(\mathrm{n}) = \sum_{j=0}^{h} j 2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j}$$

- We can factor out the $2^h$ term:

$$\mathrm{T(n)} = 2^{\mathrm{h}} \sum_{j=0}^{\mathrm{h}} \frac{j}{2^j}$$

- How do we solve this sum? Recall the geometric series, for any constant $x < 1$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

- Take the derivative with respect to $x$ and multiply by $x$

$$\sum_{j=0}^{\infty} j x^{j-1} = \frac{1}{(1-x)^2}$$

$$\sum_{j=0}^{\infty} j x^{j} = \frac{x}{(1-x)^2}$$

- We plug $x = \frac{1}{2}$ and we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

- In our case, we have a bounded sum, but since the infinite series is bounded, we can use it as an easy approximation:

$$\mathrm{T(n)} = 2^{\mathrm{h}} \sum_{\mathrm{j}=0}^{\mathrm{h}} \frac{j}{2^j}$$

$$\leq 2^{\mathrm{h}} \sum_{j=0}^{\infty} \frac{j}{2^j}$$

$$\leq 2^{\mathrm{h}}.2 = 2^{\mathrm{h}+1}$$

- Recall that $\mathrm{n} = 2^{\mathrm{h}+1} - 1$. Therefore

$$\mathrm{T(n)} \leq \mathrm{n} + 1$$

$$\boldsymbol{T(n) \in O(n)}$$

- The algorithm takes at least $\Omega(n)$ time since it must access every element at once.
- So, the total time for BuildHeap is $\theta(n)$.
- BuildHeap is a relatively complex algorithm. Yet, the analysis yield that it takes $\theta(n)$ time.
- An intuitive way to describe why it is so is to observe an important fact about binary trees.

- The fact is that the vast majority of the nodes are at the lowest level of the tree.
- For example, in a complete binary tree of height $h$, there is a total of $n \approx 2^{h+1}$ nodes.
- The number of nodes at the bottom three levels alone is:

$$2^{\mathrm{h}} + 2^{\mathrm{h}-1} + 2^{\mathrm{h}-2} = \frac{\mathrm{n}}{2} + \frac{\mathrm{n}}{4} + \frac{\mathrm{n}}{8} = \frac{7\mathrm{n}}{8} = 0.875\mathrm{n}$$

- Almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels.
- Thus, algorithms that operate on trees should be efficient (as BuildHeap is) on the bottom-most levels since that is where most of the weight of the tree resides.

# Analysis of Heapsort

- Heapsort calls *BuildHeap* procedure once at the start of algorithm. This takes $O(n)$.
- Heapsort then extracts roughly $n$ maximum elements from the heap.
- Each extract requires a constant amount of work (swap) and $O(\log n)$ *Heapify*.
- Heapsort is thus $O(n \log n)$.
- Is HeapSort also $\Omega(n \log n)$? The answer is yes.
- In fact, later we will show that comparison-based sorting algorithms cannot run faster than $(n \log n)$.
- Heapsort is such an algorithm and so is Merge-sort that we saw earlier.