

Heap Sort

Instructor: Krishna Venkatasubramanian

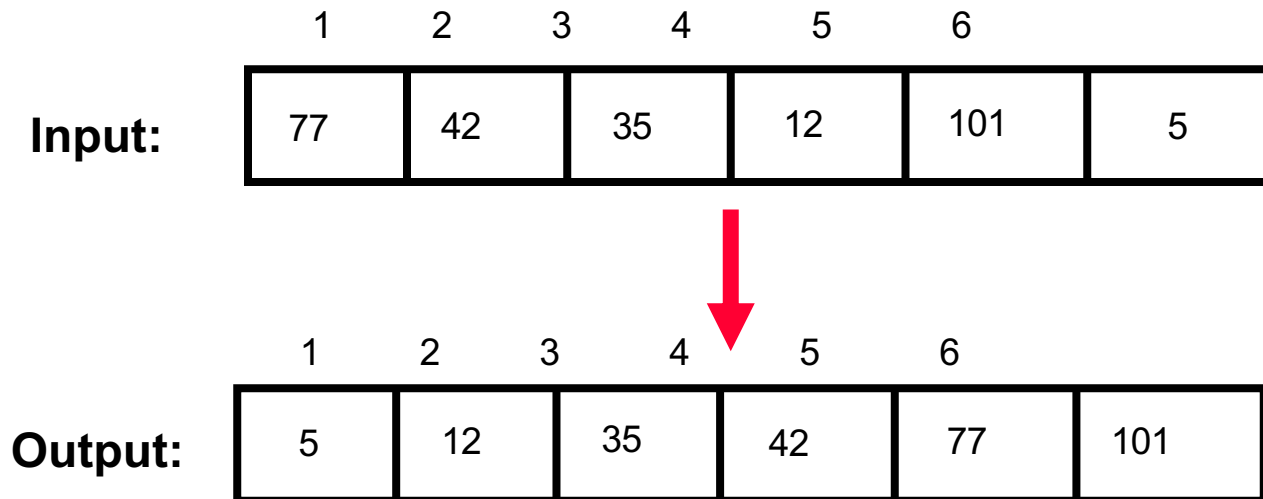
CSC 212

Announcements

- Repeated submissions for assignment 1
 - You can make multiple submissions on Gradescope, the last submission will determine your final grade
 - Make sure the assignment file is “assignment1.py” and the method names match the specifications.
- Assignment 1 deadline moved to Sunday (Oct 26) 11:59PM
- Quiz 3 on Tuesday (Oct 24)
 - Will cover material from classes on Oct 8, Oct 10, and Oct 22 (i.e., including today’s stuff)
- Assignment 2 OUT on Thursday

Sorting: Problem Definition

- **Sorting takes an unordered collection and makes it an ordered one.**



How can we sort an array using divide and conquer approach?

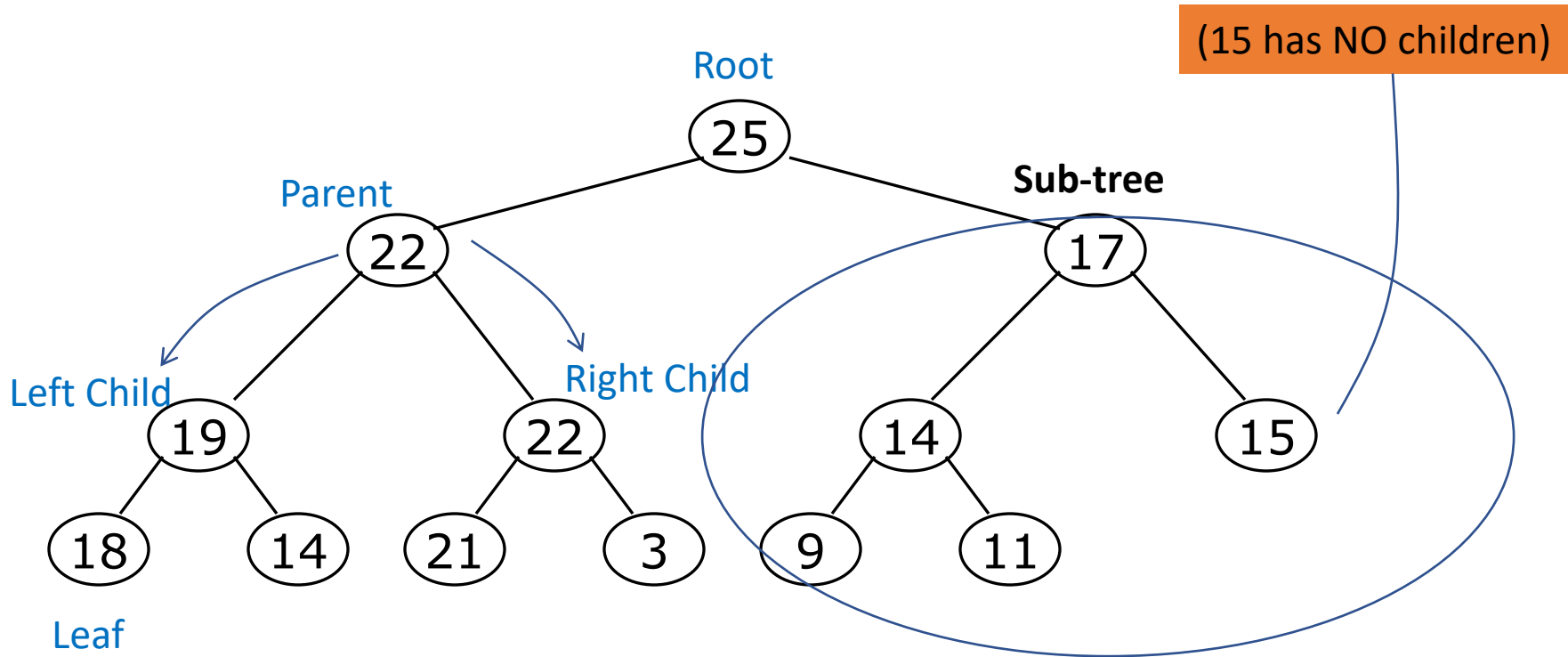
Sorting Algorithms

- *Insertion Sort --- covered already*
- *Bubble Sort --- covered already*
- *Selection Sort --- covered already*
- *Merge Sort --- covered already*
- *Quick Sort --- covered already*
- *Linear-Time Sort --- covered already*
- **Heap Sort**

Why study Heapsort?

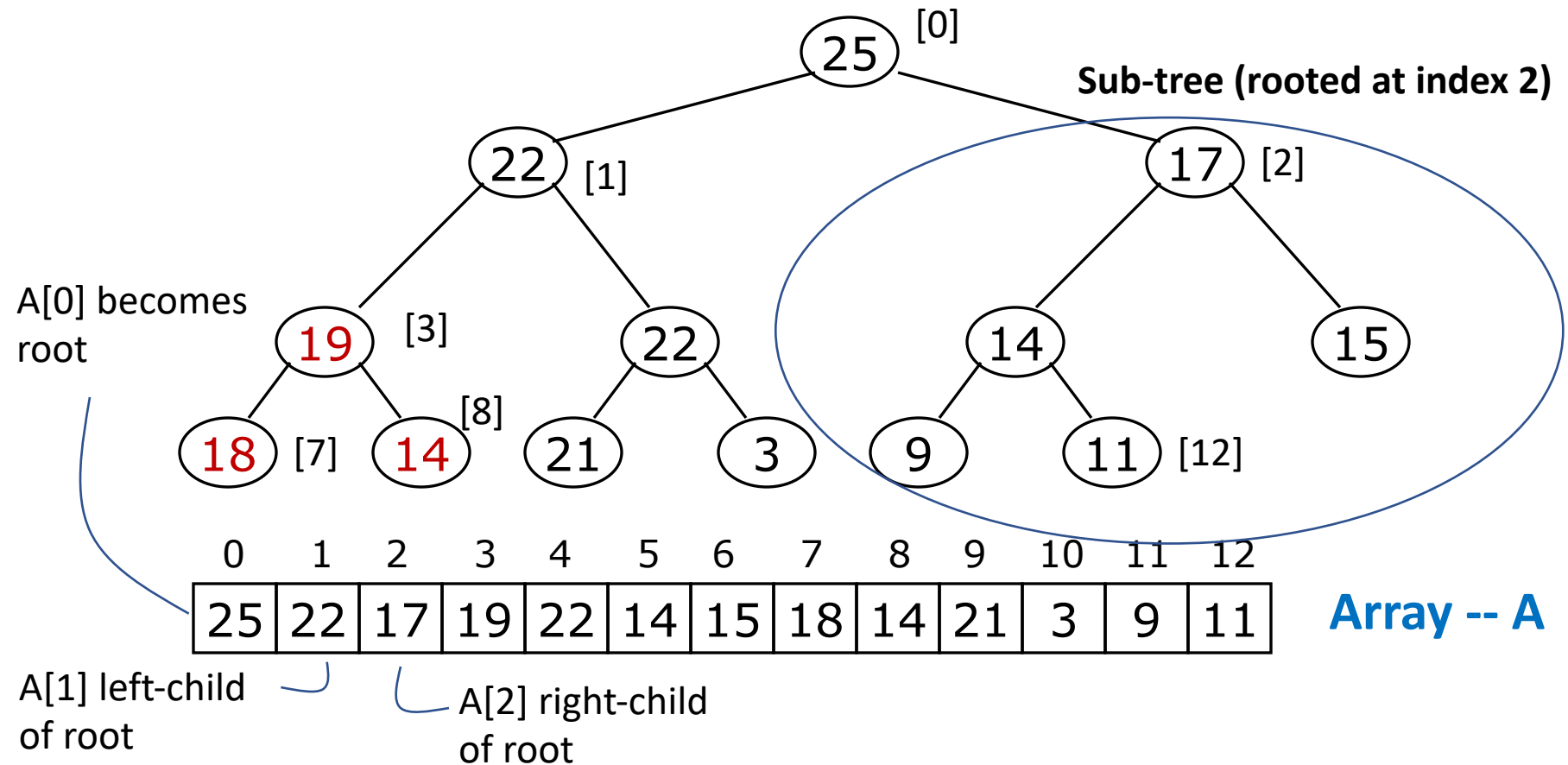
- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* **$O(n \log n)$**
 - Quicksort is usually $O(n \log n)$ but in the worst case slows to $O(n^2)$
 - **Quicksort is generally faster, but Heapsort has the guaranteed $O(n \log n)$ time** and can be used in time-critical applications
- Heapsort is a *really cool* algorithm!

We start with: A Binary Tree



Tree is called binary – because every element has two children
Leaf nodes have children which are basically Null pointers (empty).
We just don't show them.

An array and a Binary Tree



- The **left child** of index i is at index $2*i+1$
 - **[left child's parent]** is index $\text{floor}(i/2)$
- The **right child** of index i is at index $2*i+2$
 - **[right child's parent]** is index $\text{floor}(i/2)+1$
- Example: the children of node 3 (19) are 7 (18) and 8 (14)

HeapSort and Binary Trees

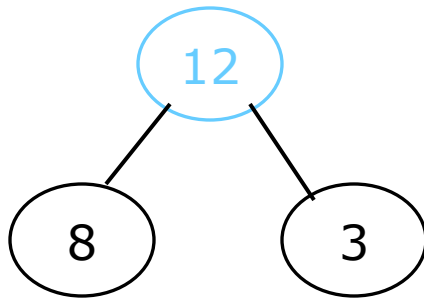
- HeapSort works on arrays (lists), but to do the actual sorting we use a **mental model of a binary tree**
- This means we view the array as a binary tree
 - Like the previous slide
- This DOES NOT mean that we are converting the array into a tree structure in memory
 - We just view it as such --- that is we track the parent/child indices
- **Once we view the array as a binary tree, we then understand the sorting through the tree structure**

Again: Things to Remember

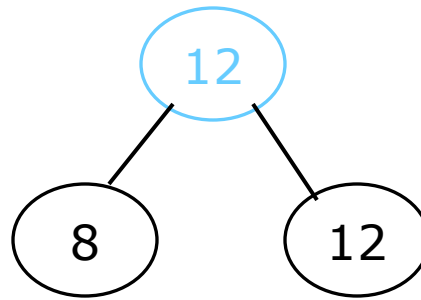
- The rest of the slides will **focus on the BinaryTree view of the array**
- **Even though we are manipulating the BinaryTree structure, it's not an actual tree structure in memory**
- In **reality we are changing the input array**
 - It's just easier to see what's happening when we view the array as a BinaryTree

The MAX-Heap property

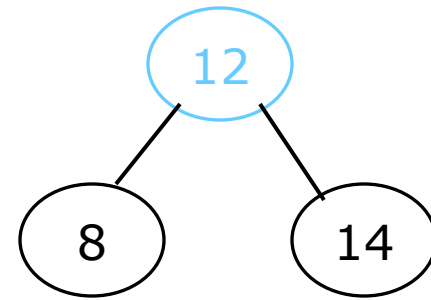
- A node has the **MAX-Heap property** if the **value in the node is \geq the values in its children**



Blue node has
MAX-heap
property



Blue node has
MAX-heap
property



Blue node does not
have MAX-heap
property

- All **leaf nodes automatically have the MAX-Heap property**
- A **binary tree is a MAX-heap if *all* nodes in it have the MAX-heap property**
- You can similarly have MIN-heap(with the smallest element as parent)

MAX-Heap and Min-Heaps are generally called HEAPS

HeapSort: Plan of attack

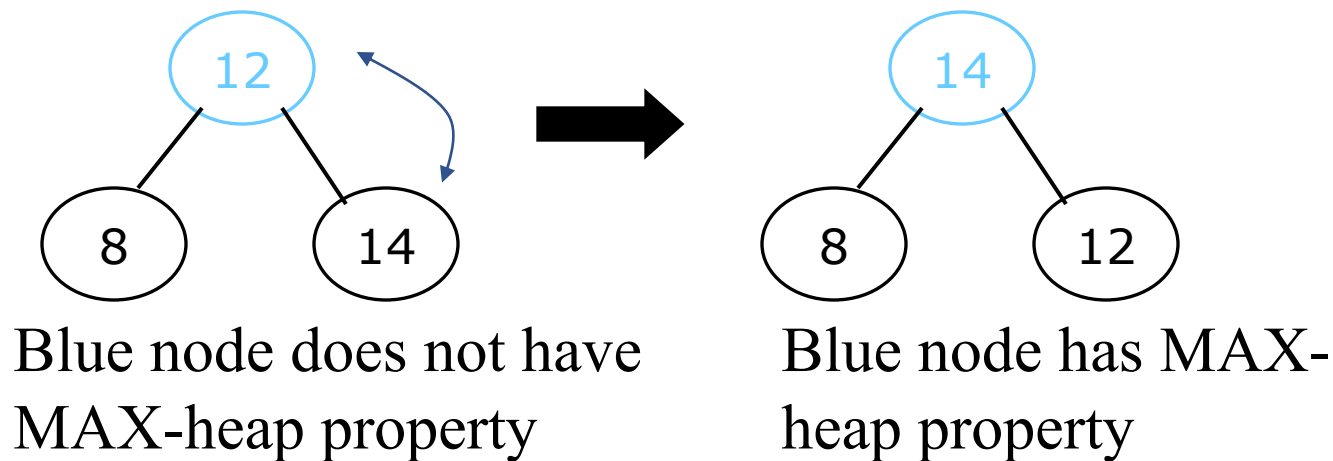
- First, we **turn any portion of a binary tree** (which we imagine our array to be) **into a MAX-heap (data structure)**
- Next, we will learn how to **turn a binary tree *back* into a MAX-heap after it has been changed** in a certain way
- Finally we will see how to use these ideas to sort an *array*

HeapSort: Plan of attack

- First, we **turn any portion of a binary tree** (which we imagine our array to be) **into a MAX-heap (data structure)**
- Next, we will learn how to turn a binary tree *back* into a MAX-heap after it has been changed in a certain way
- Finally we will see how to use these ideas to sort an *array*

MAX-Heapifying Step

- Given a node that does not have the MAX-heap property, you can give it the MAX-heap property by exchanging its value with the value of the larger child

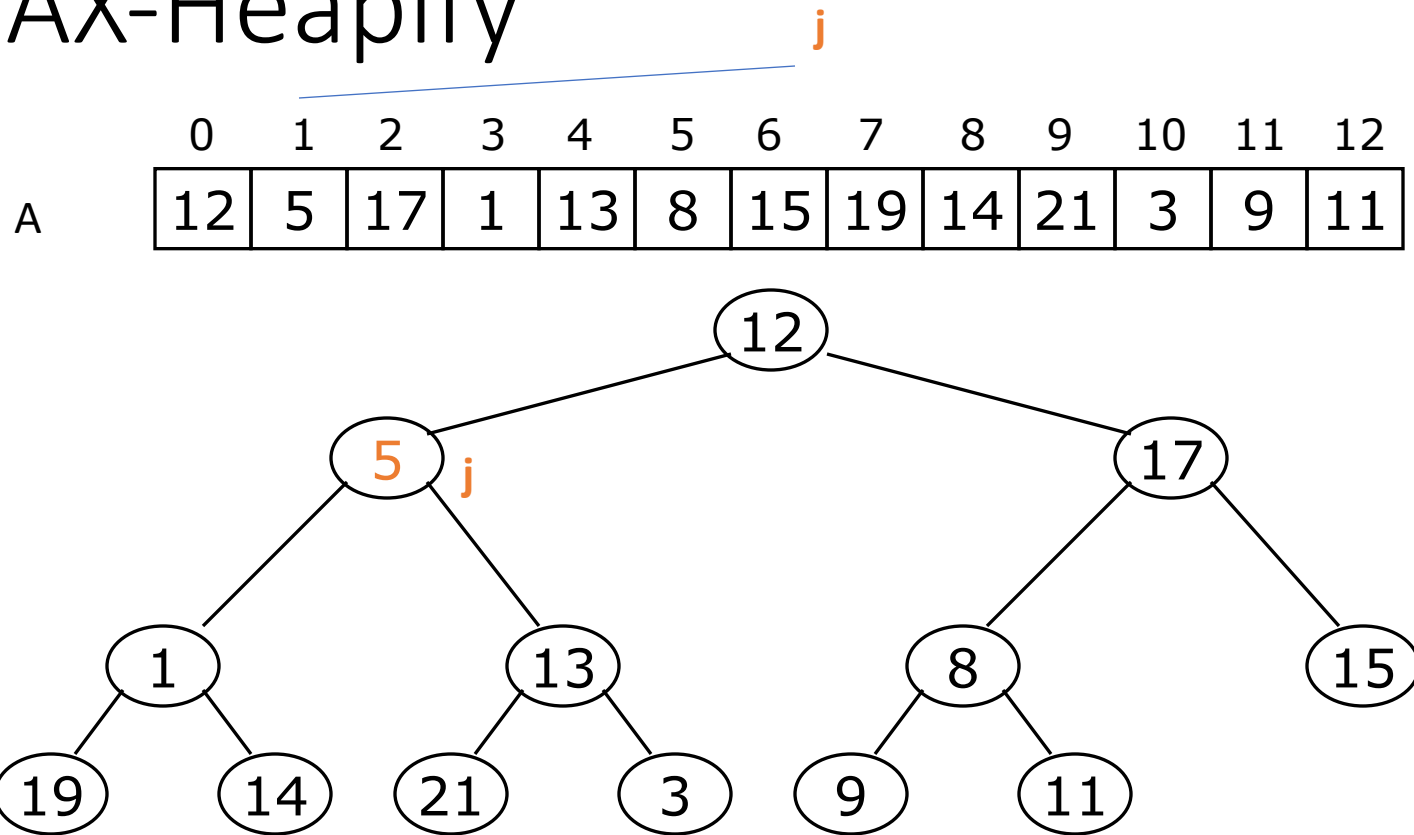


- Notice that the **child may have lost the MAX-heap property**
- We may to do this more than once**
- This is a single step in converting an binary tree into a MAX-Heap**

MAX-Heapify

- Given the MAX-Heapifying step (for a triad of parent and its two children)
- We now, present MAX-Heapify algorithm, that takes an **Array (A)** and an **index (j)** and produces **a MAX-Heap for the sub-tree rooted index j**

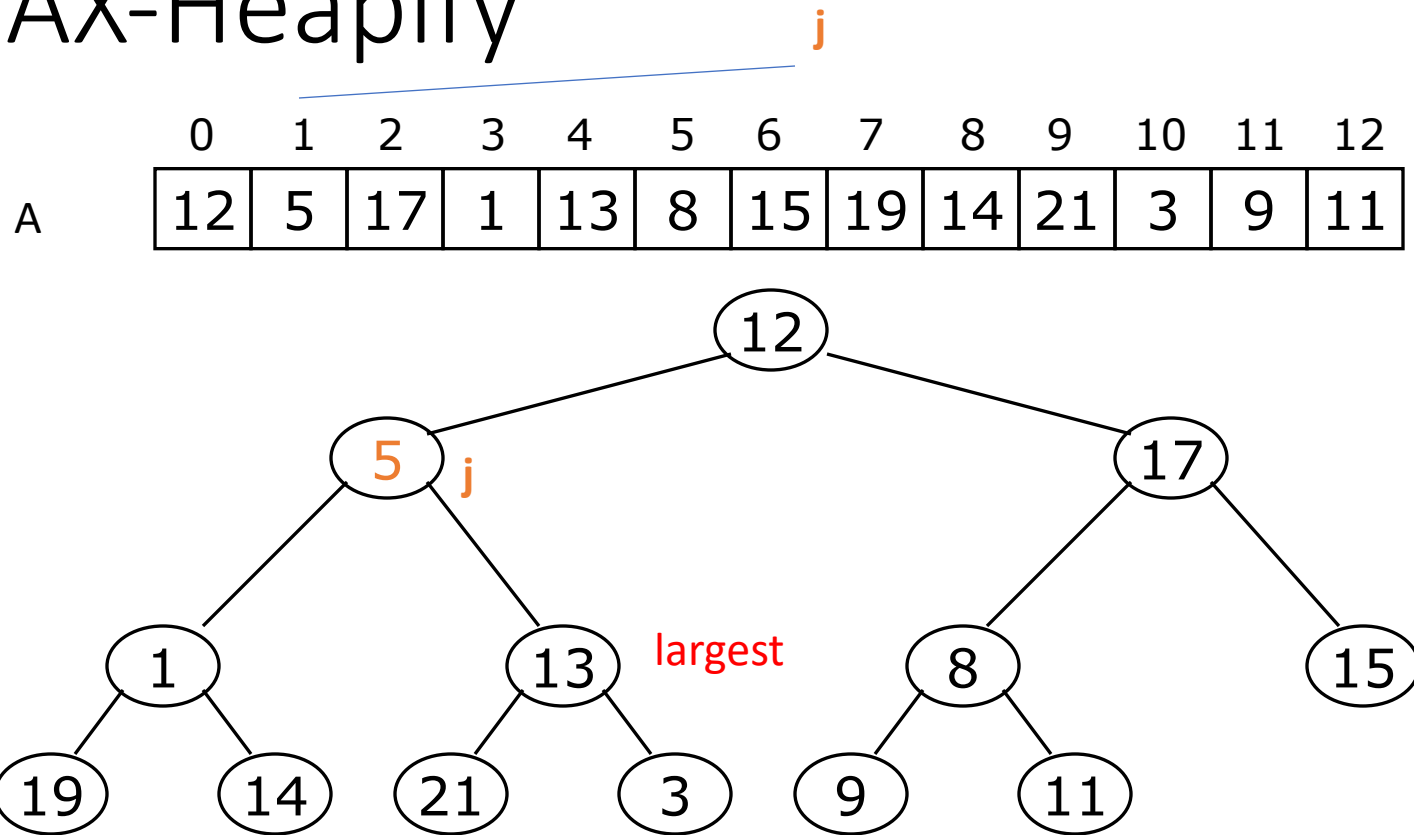
MAX-Heapify



Function call **MAX-Heapify(A, j)**

- Compare A[j] (if not leaf) and its children
 - find index **largest** of the three
- If **largest NOT j**, then **swap A[largest] and A[j]**
- **MAX-Heapify(A, largest)**

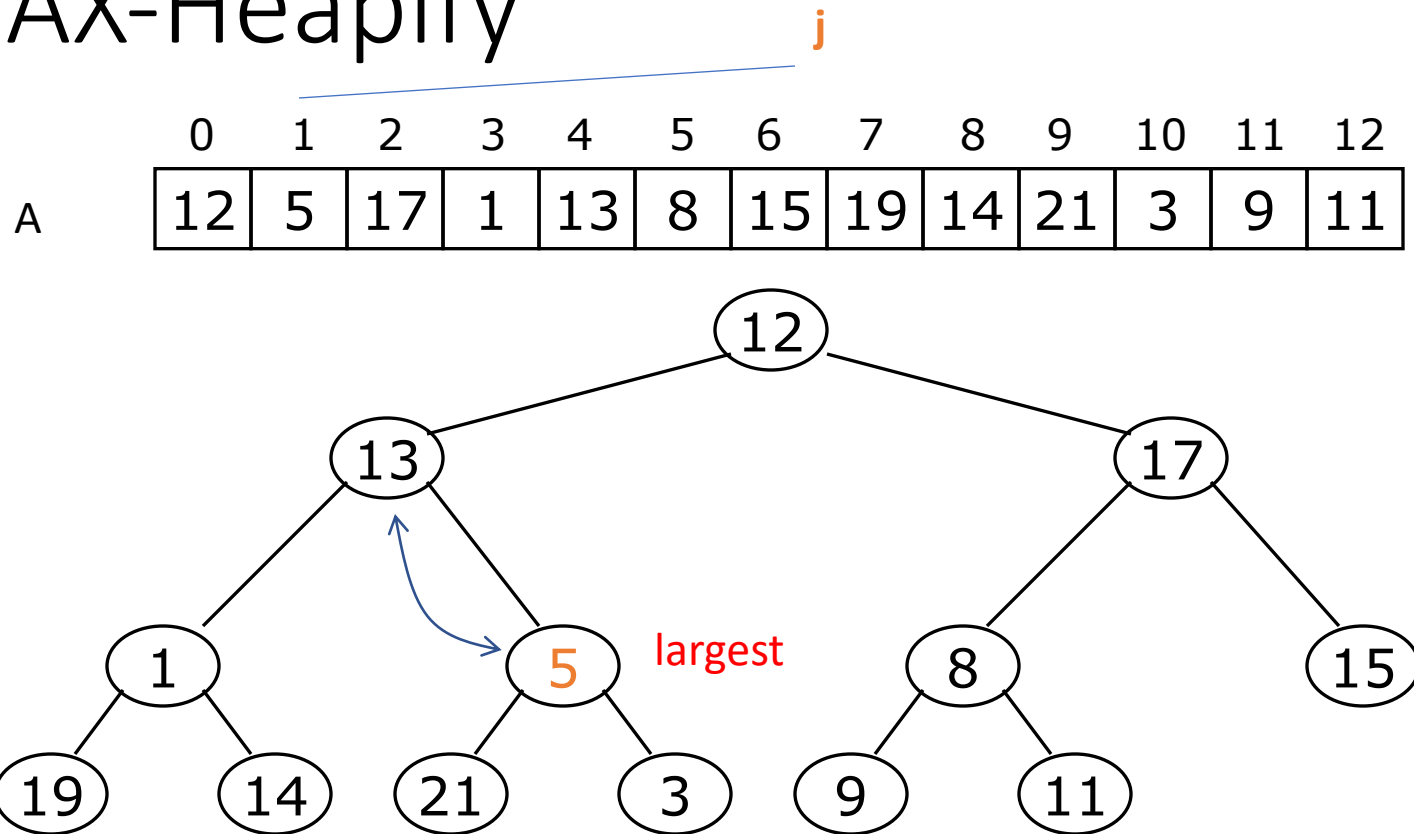
MAX-Heapify



Function call **MAX-Heapify(A, j)**

- Compare A[j] (if not leaf) and its children
 - find index **largest** of the largest of the three
- If **largest NOT j**, then **swap A[largest] and A[j]**
- **MAX-Heapify(A, largest)**

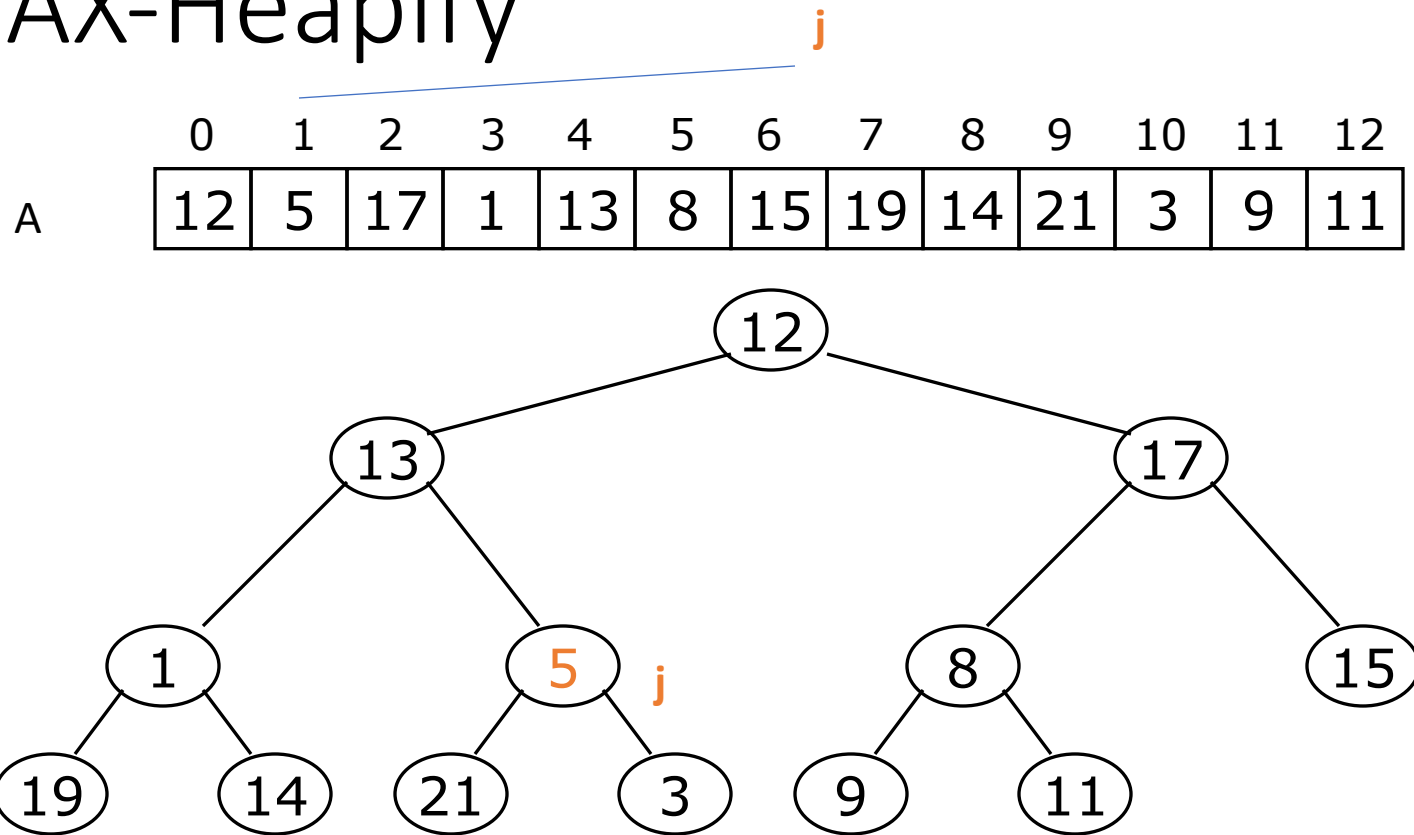
MAX-Heapify



Function call **MAX-Heapify(A, j)**

- Compare A[j] (if not leaf) and its children
 - find index **largest** of the three
- If **largest NOT j**, then **swap A[largest] and A[j]**
- **MAX-Heapify(A, largest)**

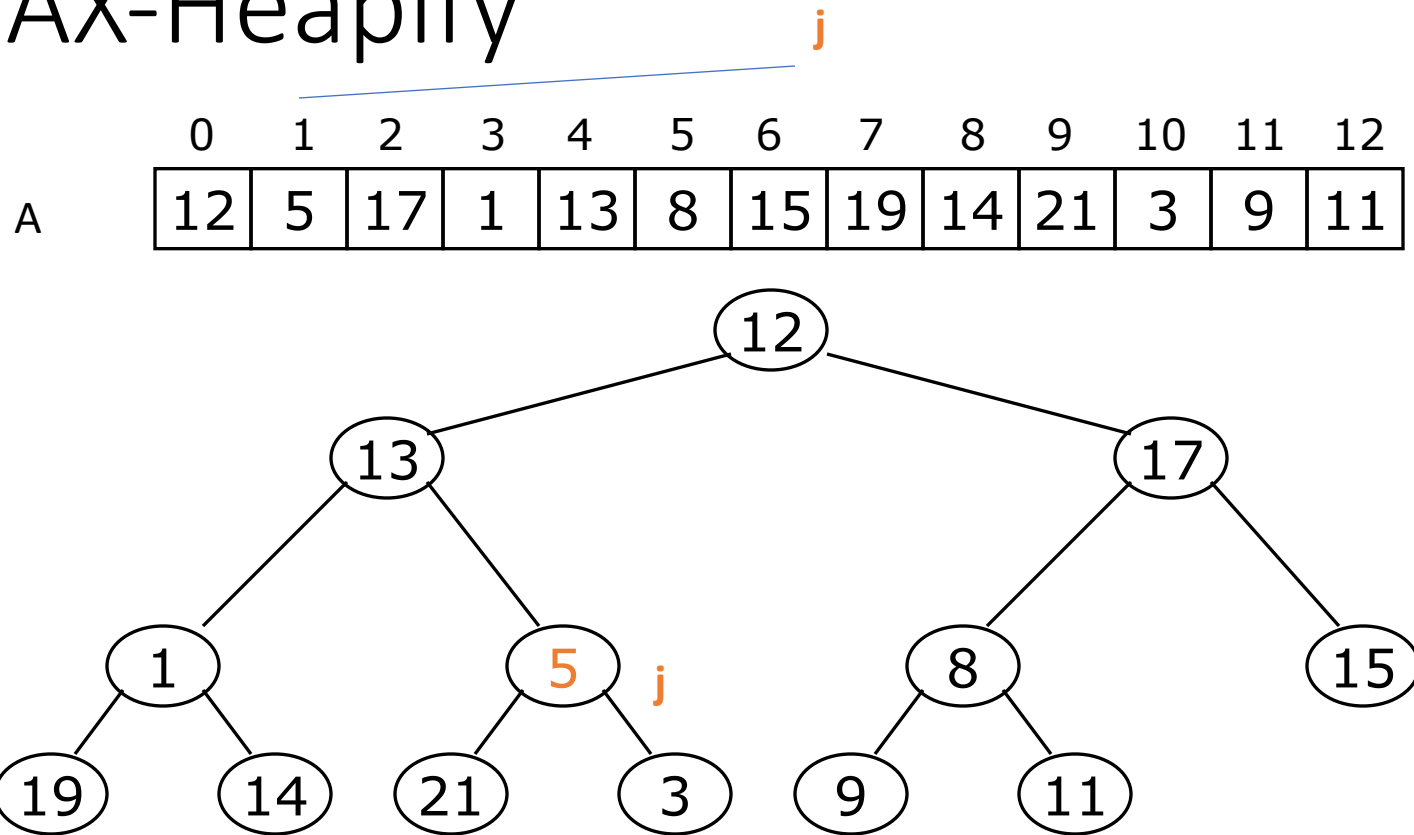
MAX-Heapify



Function call **MAX-Heapify**((A , j))

- Compare $A[j]$ (if not leaf) and its children
 - find index **largest** of the largest of the three
- If **largest NOT j** , then **swap $A[\text{largest}]$ and $A[j]$**
- **MAX-Heapify**($A, \text{largest}$)

MAX-Heapify

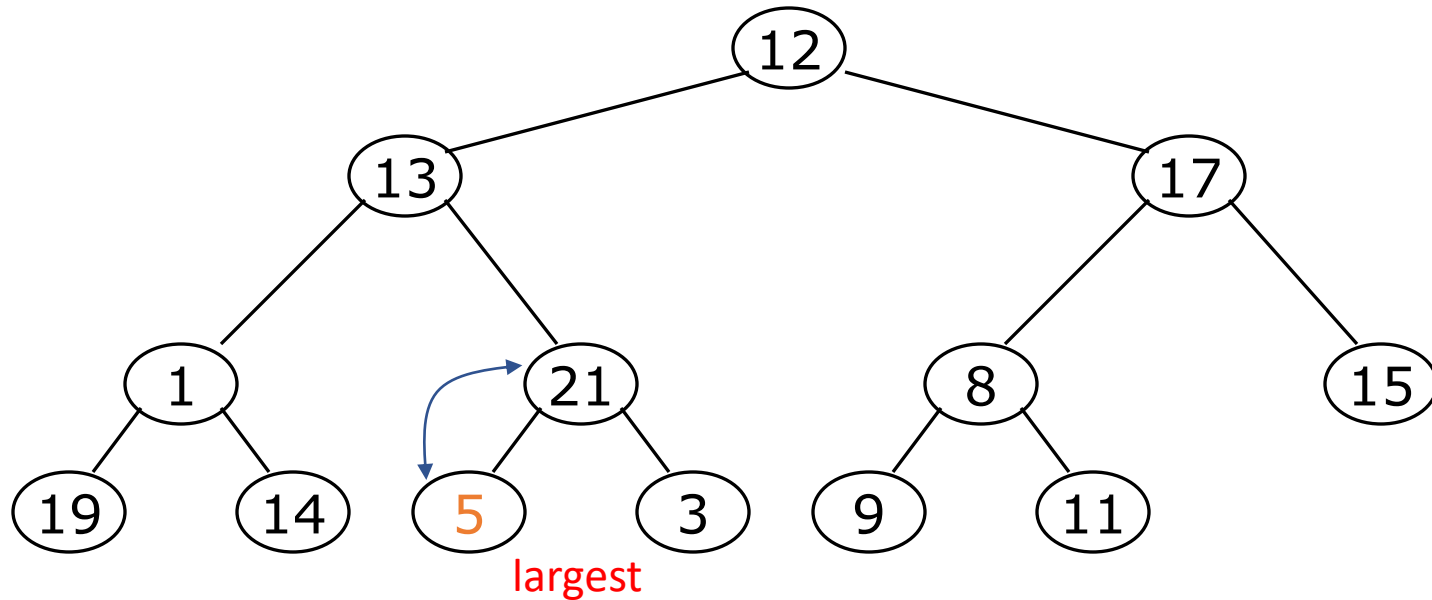


Function call **MAX-Heapify(A, j)**

- Compare A[j] (if not leaf) and its children
 - find index **largest** of the largest of the three
- If **largest NOT j**, then **swap A[largest] and A[j]**
- **MAX-Heapify(A, largest)**

MAX-Heapify

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	12	5	17	1	13	8	15	19	14	21	3	9	11



Function call **MAX-Heapify(A, j)**

- Compare $A[j]$ (if not leaf) and its children
 - find index **largest** of the largest of the three
- If **largest NOT j**, then **swap $A[\text{largest}]$ and $A[j]$**
- **MAX-Heapify(A, largest)**

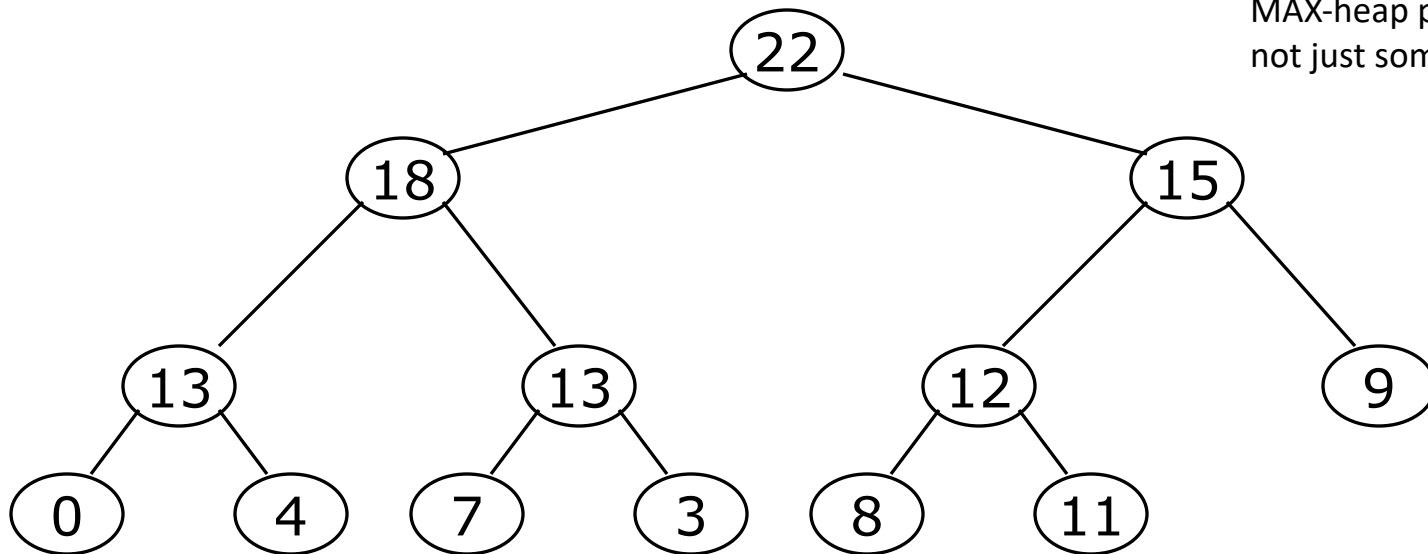
HeapSort: Plan of attack

- First, we turn any portion of a binary tree (which we imagine our array to be) into a **MAX-heap** (data structure)
- Next, we will learn how to **turn a binary tree *back* into a MAX-heap after it has been changed** in a certain way
- Finally we will see how to use these ideas to sort an *array*

A sample heap

- Here's a **sample binary tree after it has been MAX-HEAPIFIED**
(different tree than in previous slides)

The whole tree satisfies the MAX-heap property here, not just some sub-tree

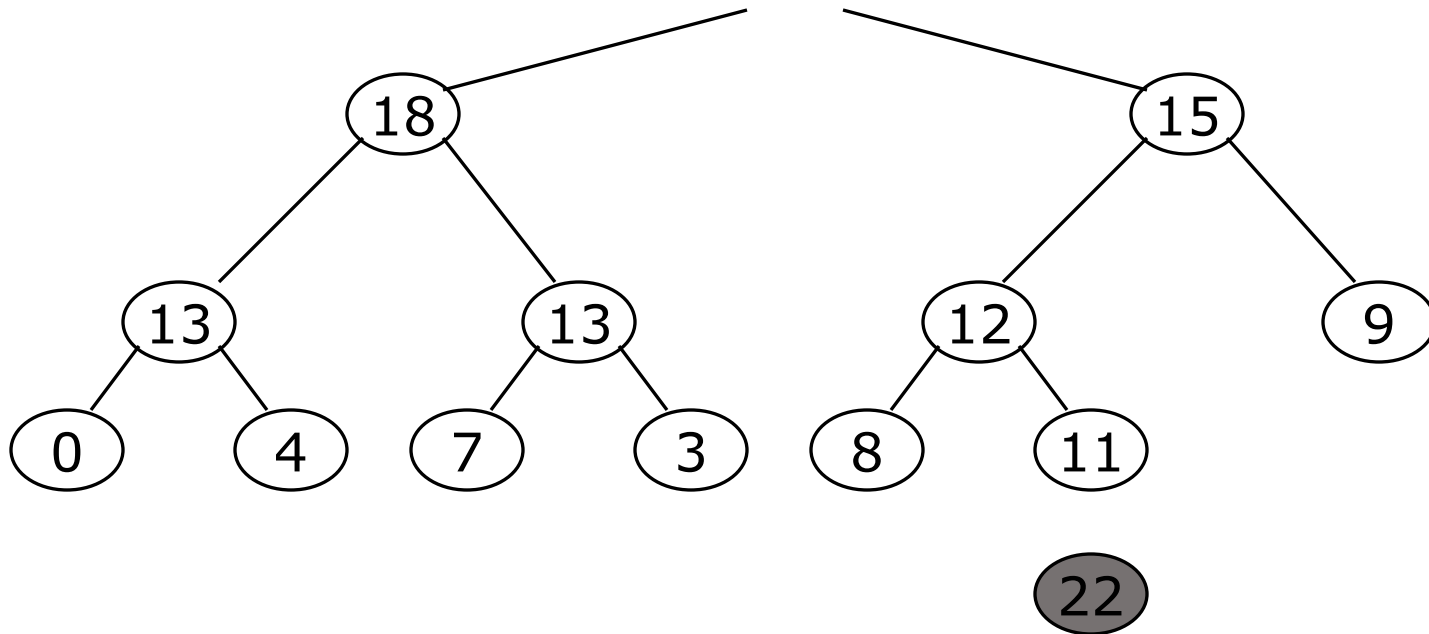


- Notice that **MAX-heapified does *not* mean sorted**
- MAX-Heapifying **does *not* change the shape of the binary tree**; this **binary tree is balanced** because it started out that way

The height difference between left sub-tree and right-subtree is at most 1

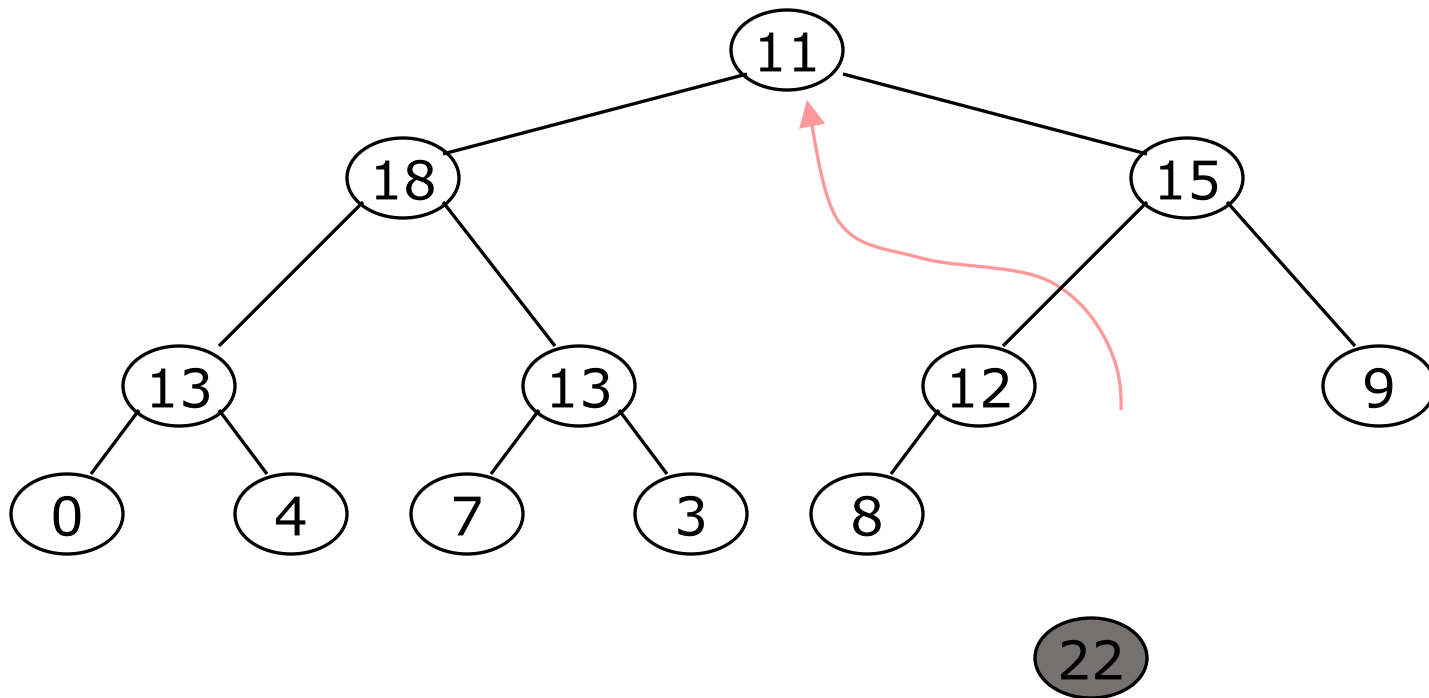
Removing the Root

- Notice that the largest number is now in the root
- Suppose we *discard* the root:



Replacing the Root

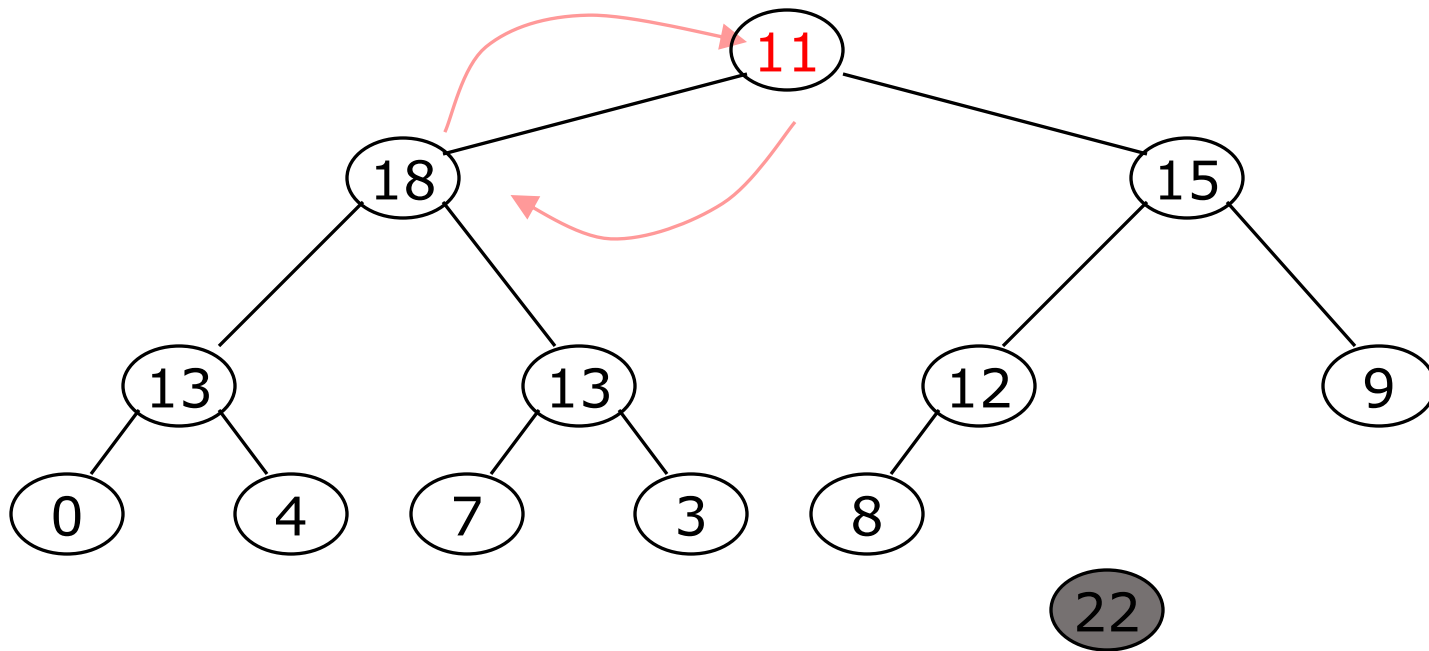
- Remove the rightmost leaf at the deepest level and use it for the new root



- The MAX-heap property of this “new” tree is now lost!

MAX-Heapify (I)

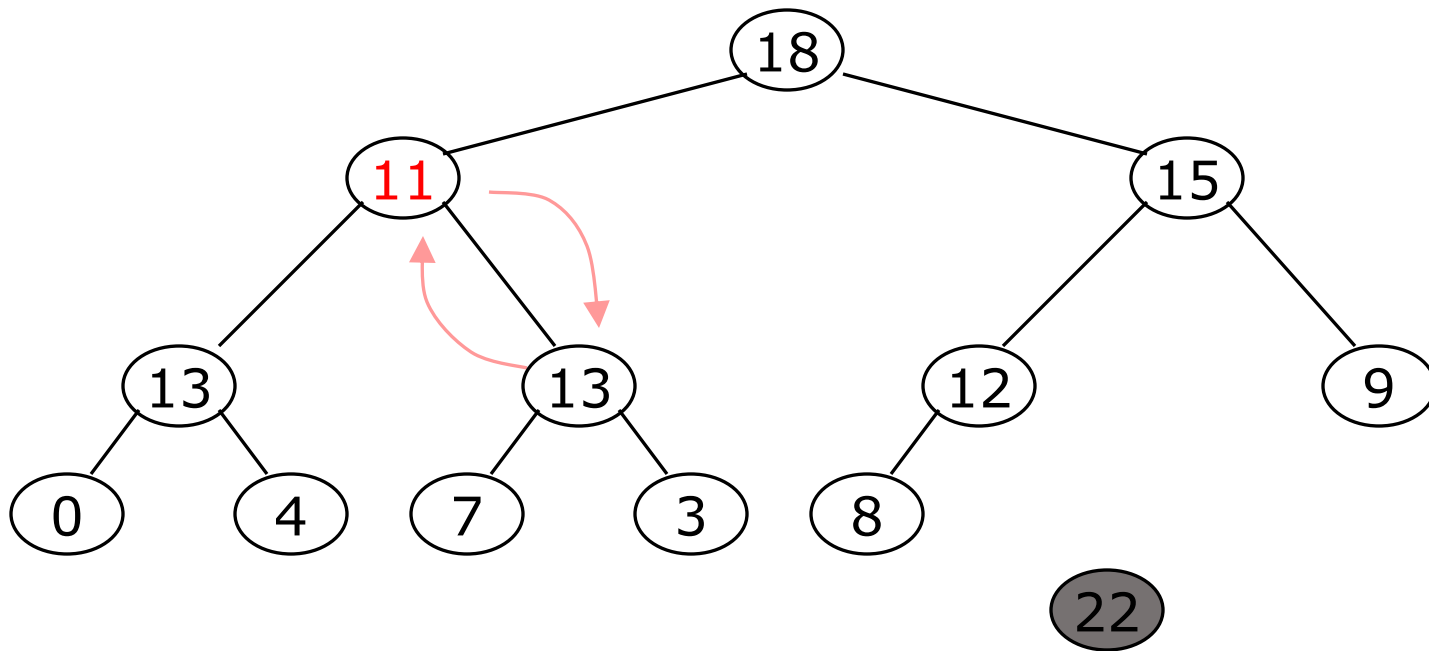
- However, *only the root* lacks the MAX-heap property



- We can MAX-Heapify() the root
- After doing this, one and only one of its children may have lost the MAX-heap property

MAX-Heapify (II)

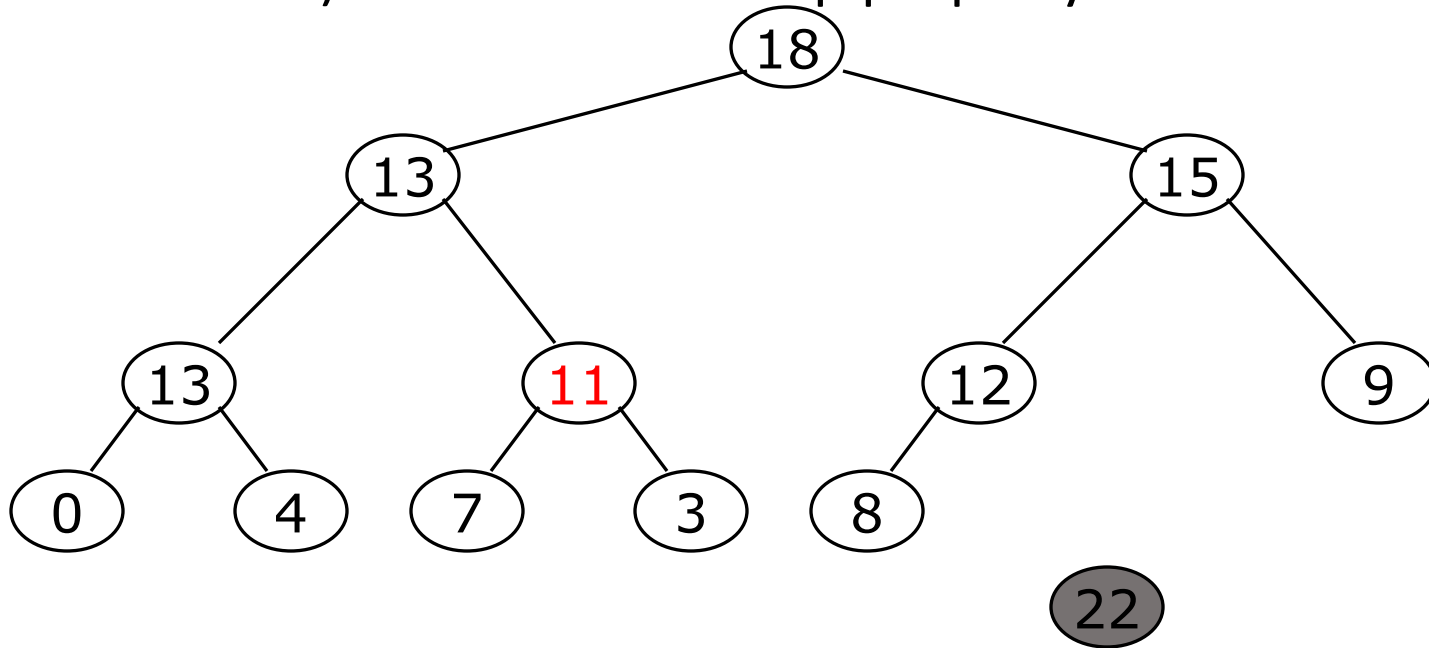
- Now the left child of the root (still the number 11) lacks the MAX-heap property



- We can MAX-Heapify() this node
- After doing this, one and only one of its children may have lost the MAX-heap property

MAX-Heapify (III)

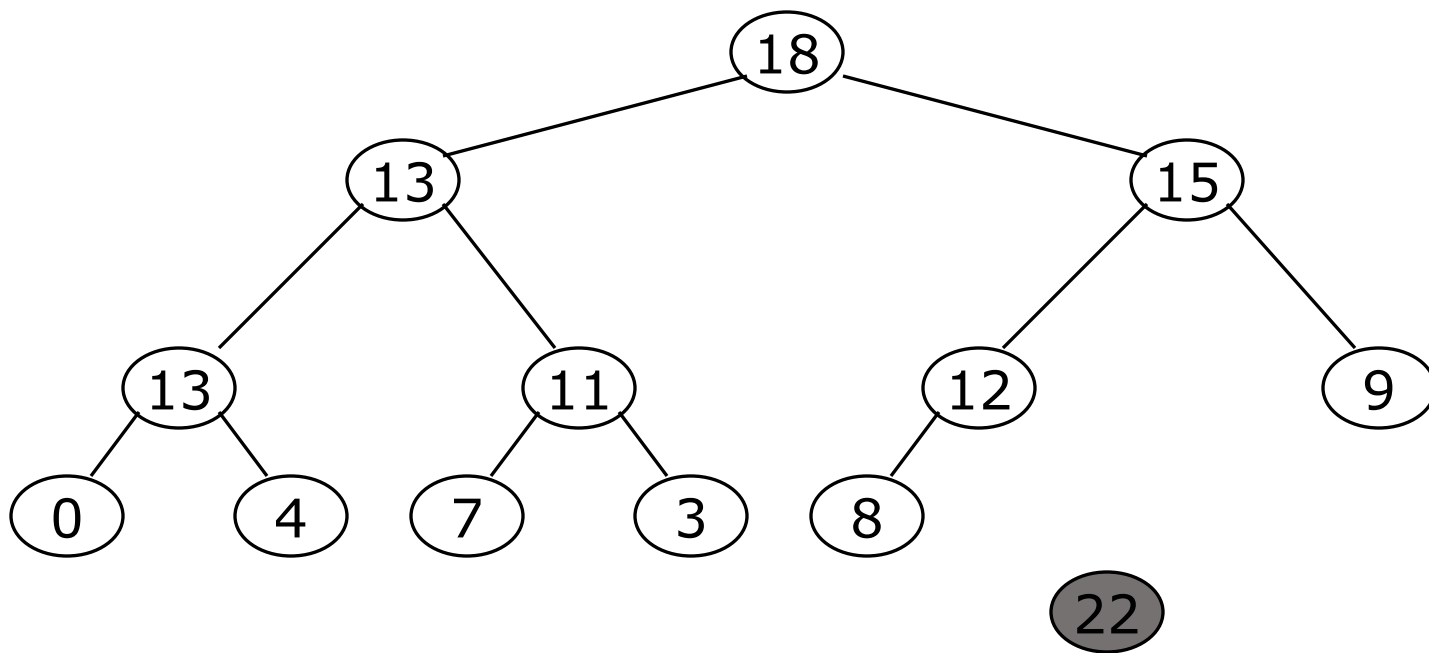
- Now the right child of the left child of the root (still the number 11) HAS the MAX-heap property:



- At this stage the MAX-heap property is satisfied.

New Heap

- Our tree is once again a heap, because every node in it has the MAX-heap property



- Once again, the largest (or *a* largest) value is in the root
- **We can repeat this process until the tree becomes empty**
 - This **produces a sequence of values in order largest to smallest**

HeapSort: Plan of attack

- First, we turn any portion of a binary tree (which we imagine our array to be) into a MAX-heap (data structure)
- Next, we will learn how to turn a binary tree *back* into a MAX-heap after it has been changed in a certain way
- Finally we will see how to use these ideas to sort an *array*

HeapSort

To sort:

Build MAX-HEAP

while the array isn't empty

remove the root

replace the root with the last leaf node

MAX-HEAPIFY(A,0)

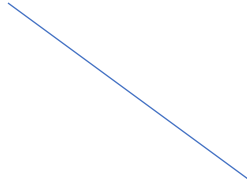
Build MAX-HEAP from an array (A):

if A.length() %2 == 0 **then** MID = floor(A.length()/2)-1

else MID = floor(A.length()/2)

for i in MID downto 1:

MAX-Heapify(A,i)



This is called initially
when a new array is
received for sorting via
HeapSort

HeapSort

To sort:

```
Build MAX-HEAP
while the array isn't empty
    remove the root
    replace the root with the last leaf node
    MAX-HEAPIFY(A,0)
```

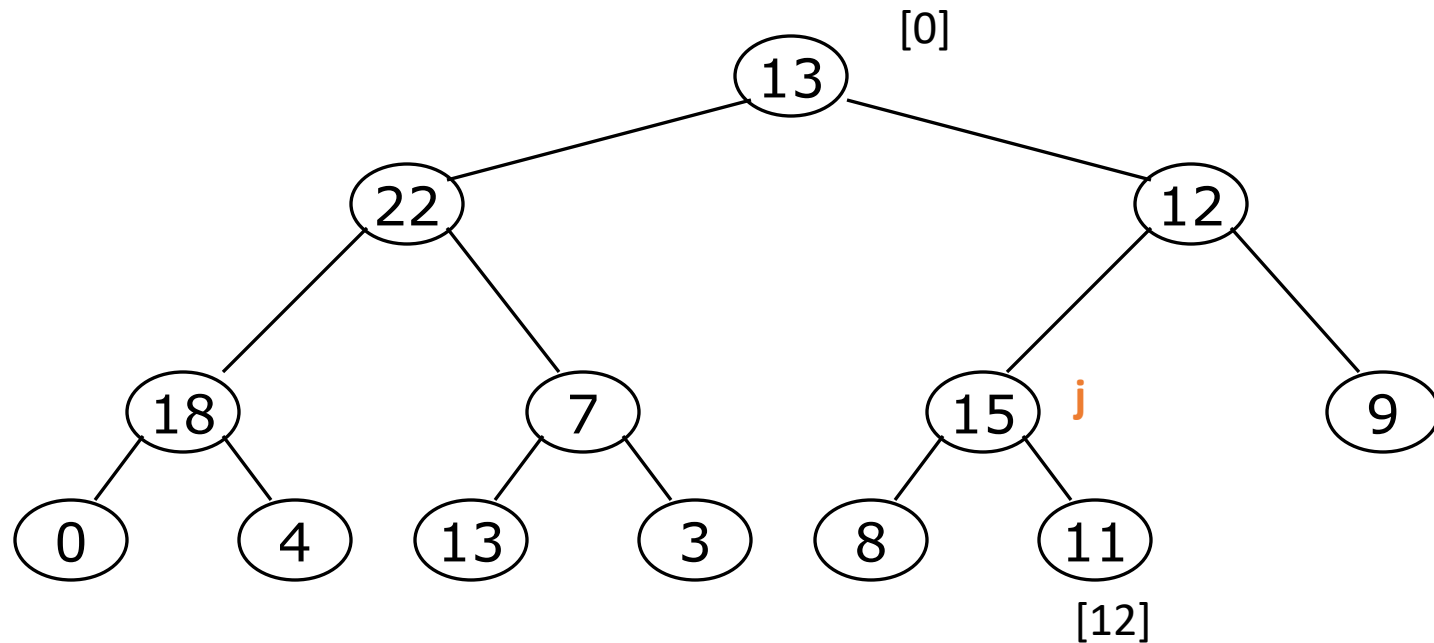
Build MAX-HEAP from an array (A):

```
if A.length() %2 == 0 then MID = floor(A.length()/2)-1
else MID = floor(A.length()/2)
for i in MID downto 1:
    MAX-Heapify(A,i)
```

This is called initially
when a new array is
received for sorting via
HeapSort

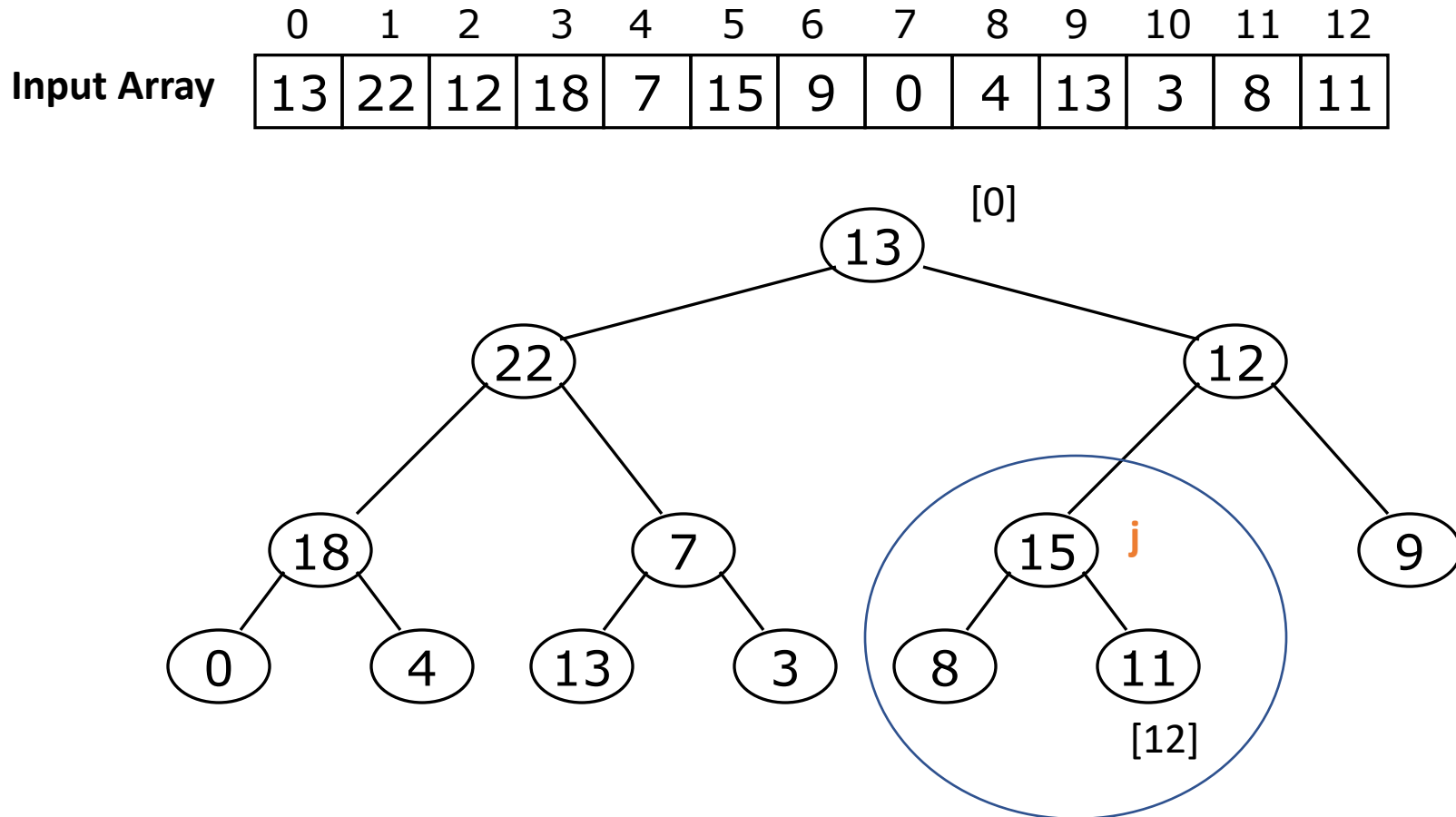
Build MAX-HEAP

	0	1	2	3	4	5	6	7	8	9	10	11	12
Input Array	13	22	12	18	7	15	9	0	4	13	3	8	11



Binary Tree view of the input array

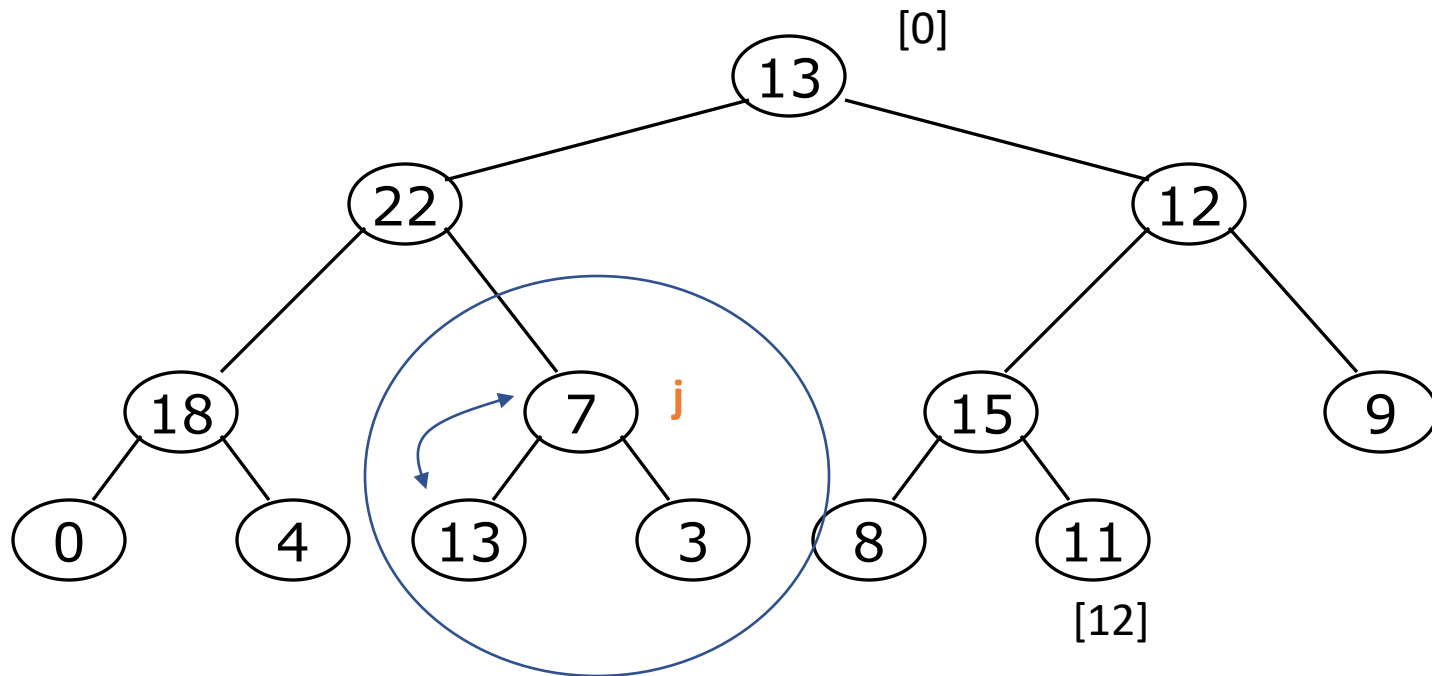
Build MAX-HEAP



j is the index of “mid” point of the array, given length A is even

Build MAX-HEAP

	0	1	2	3	4	5	6	7	8	9	10	11	12
Input Array	13	22	12	18	7	15	9	0	4	13	3	8	11

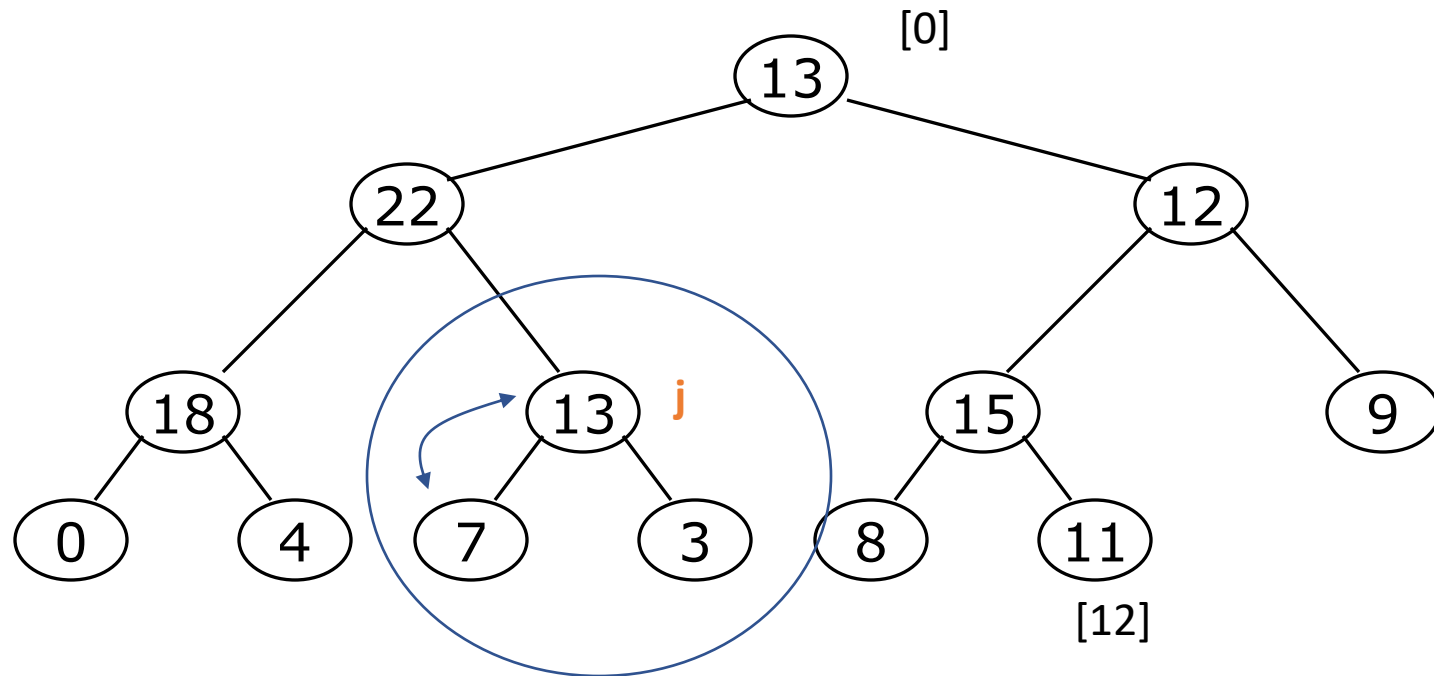


Run the for loop in Build-MAX Heap by considering the previous element to 15

Build MAX-HEAP

Input Array

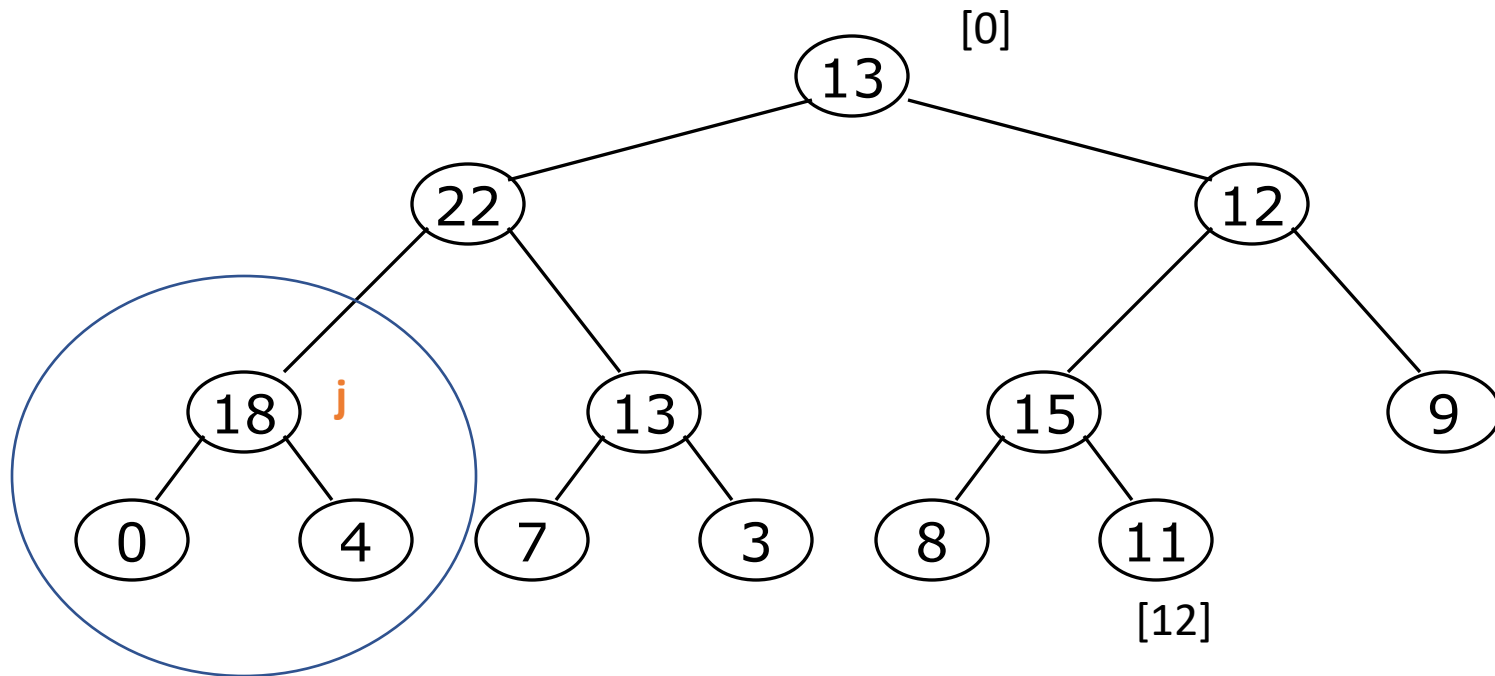
0	1	2	3	4	5	6	7	8	9	10	11	12
13	22	12	18	13	15	9	0	4	7	3	8	11



Build MAX-HEAP

Input Array

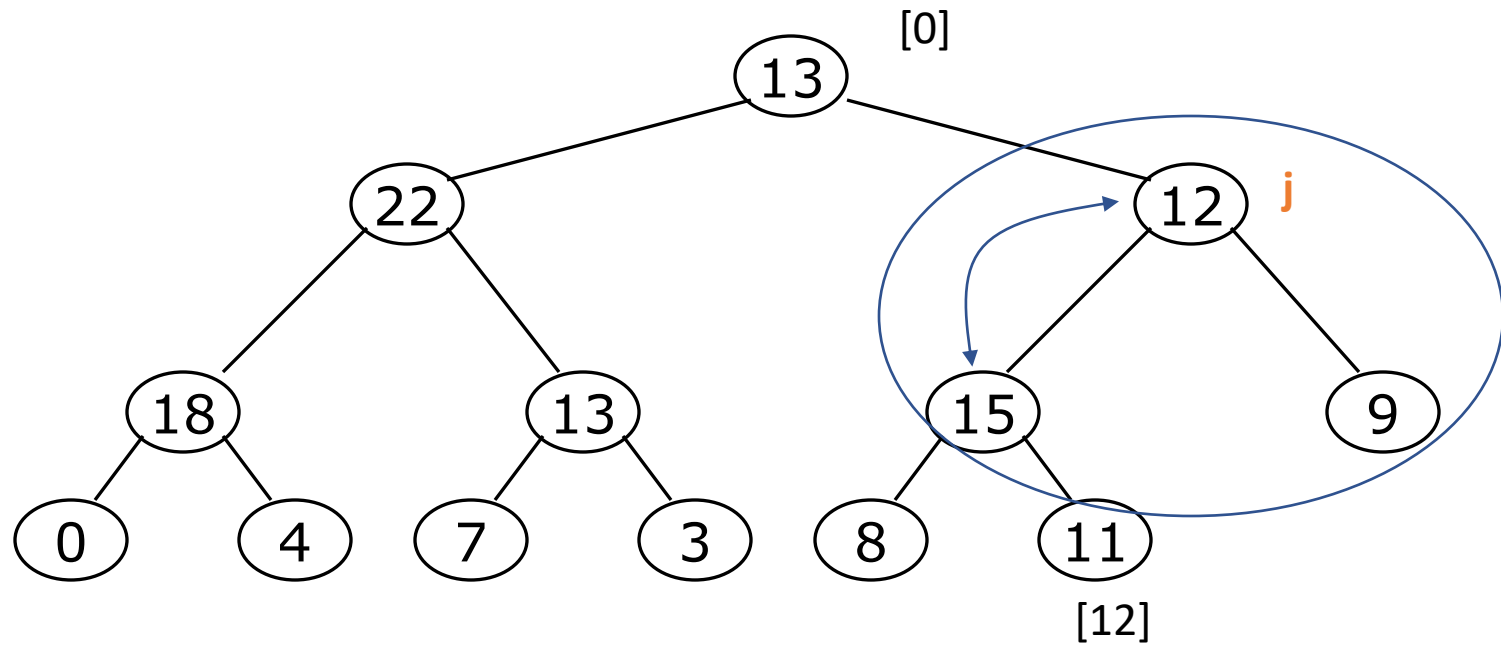
0	1	2	3	4	5	6	7	8	9	10	11	12
13	22	12	18	13	15	9	0	4	7	3	8	11



Build MAX-HEAP

Input Array

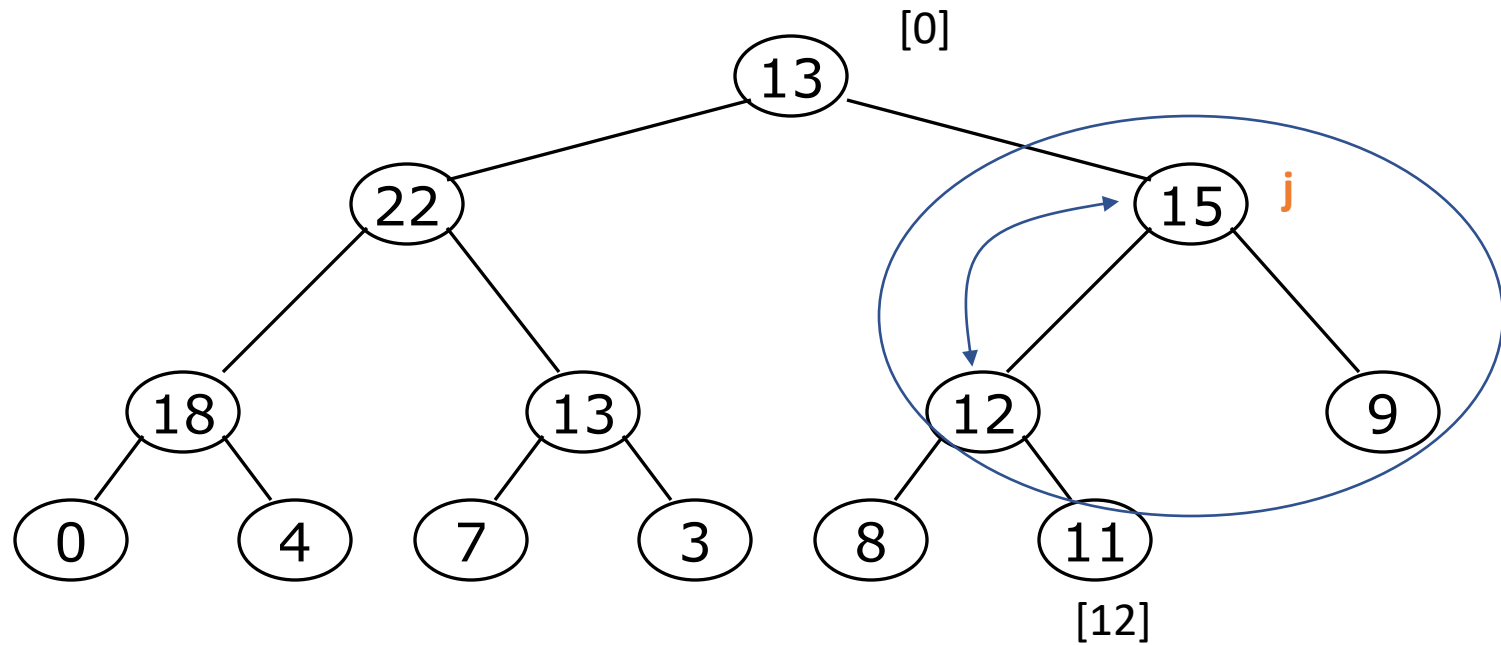
0	1	2	3	4	5	6	7	8	9	10	11	12
13	22	12	18	13	15	9	0	4	7	3	8	11



Build MAX-HEAP

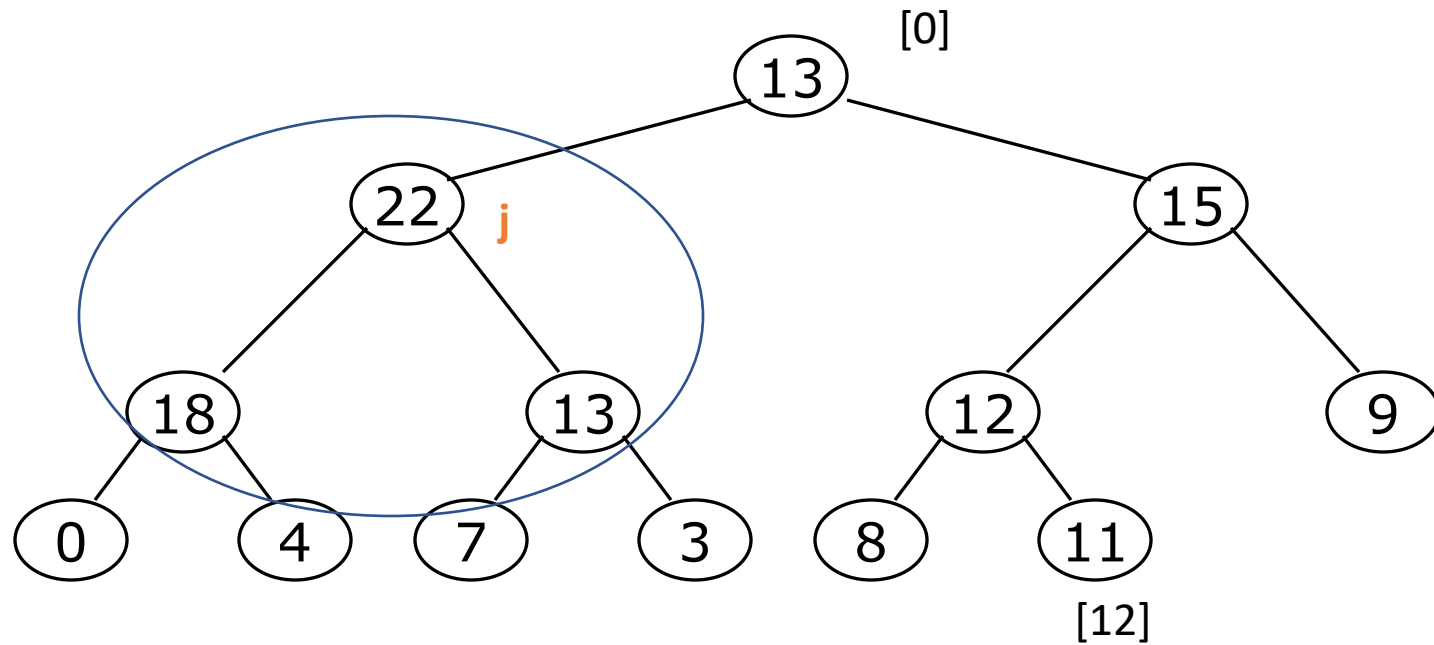
Input Array

0	1	2	3	4	5	6	7	8	9	10	11	12
13	22	15	18	13	12	9	0	4	7	3	8	11

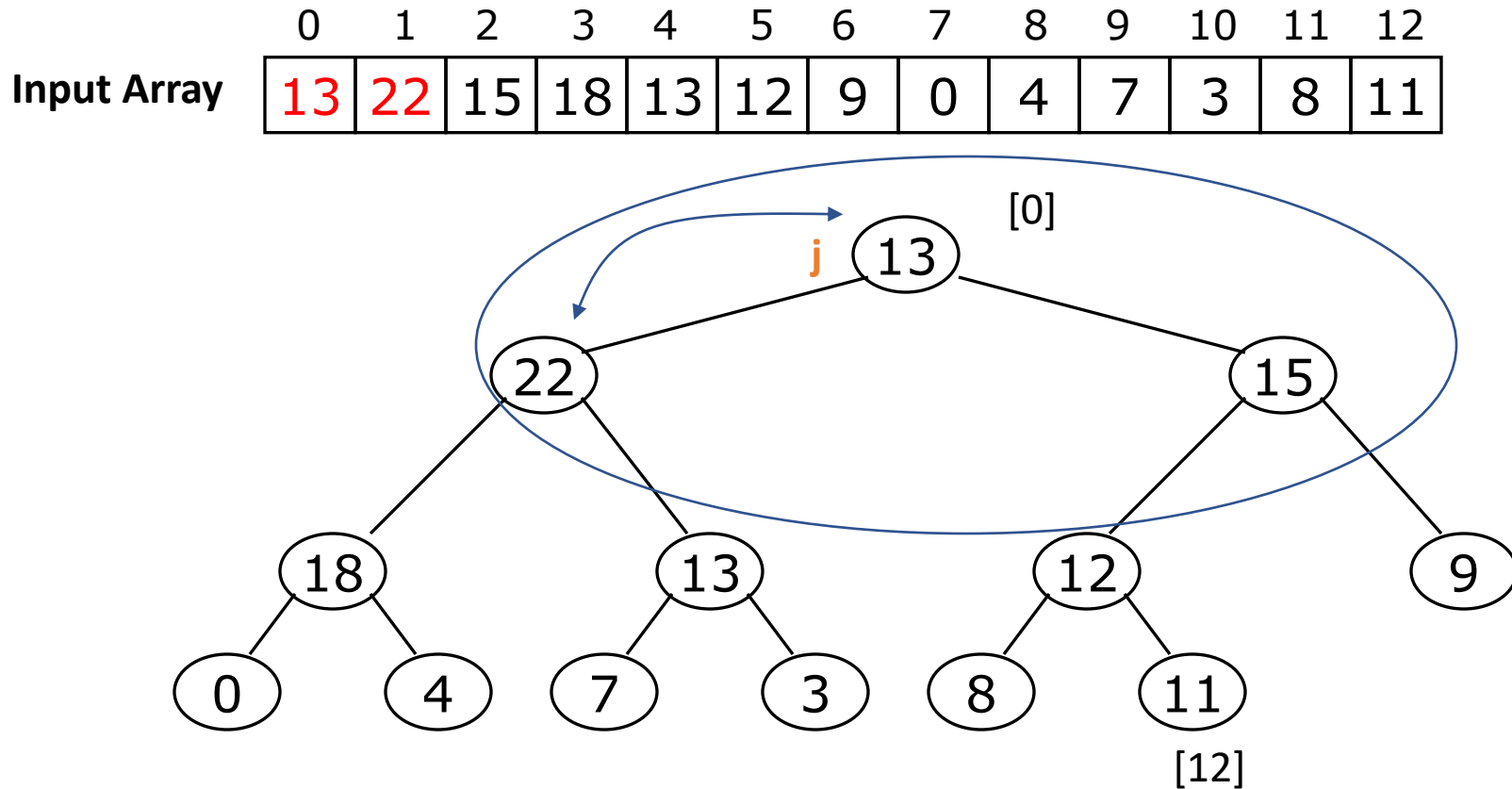


Build MAX-HEAP

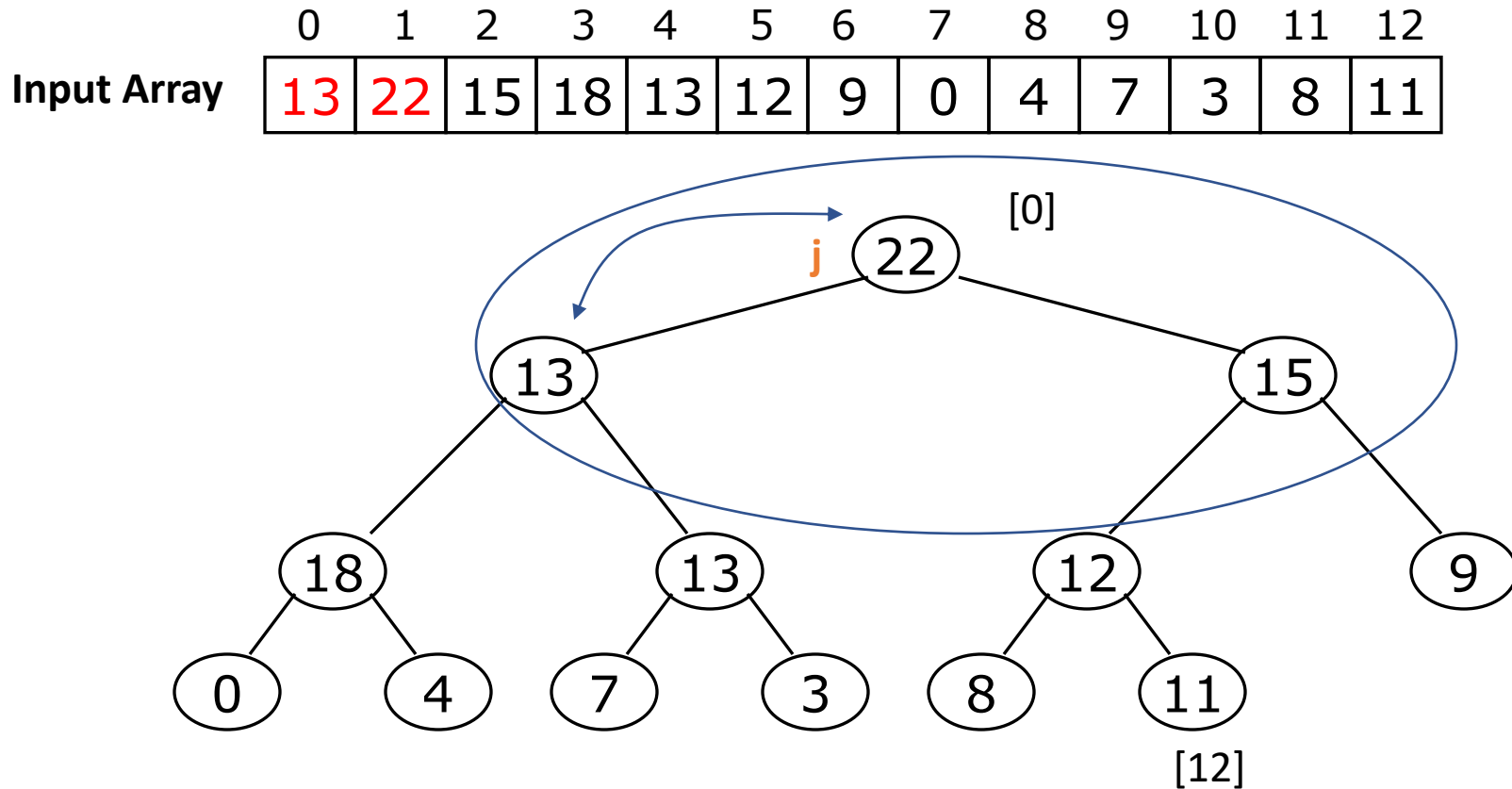
	0	1	2	3	4	5	6	7	8	9	10	11	12
Input Array	13	22	15	18	13	12	9	0	4	7	3	8	11



Build MAX-HEAP



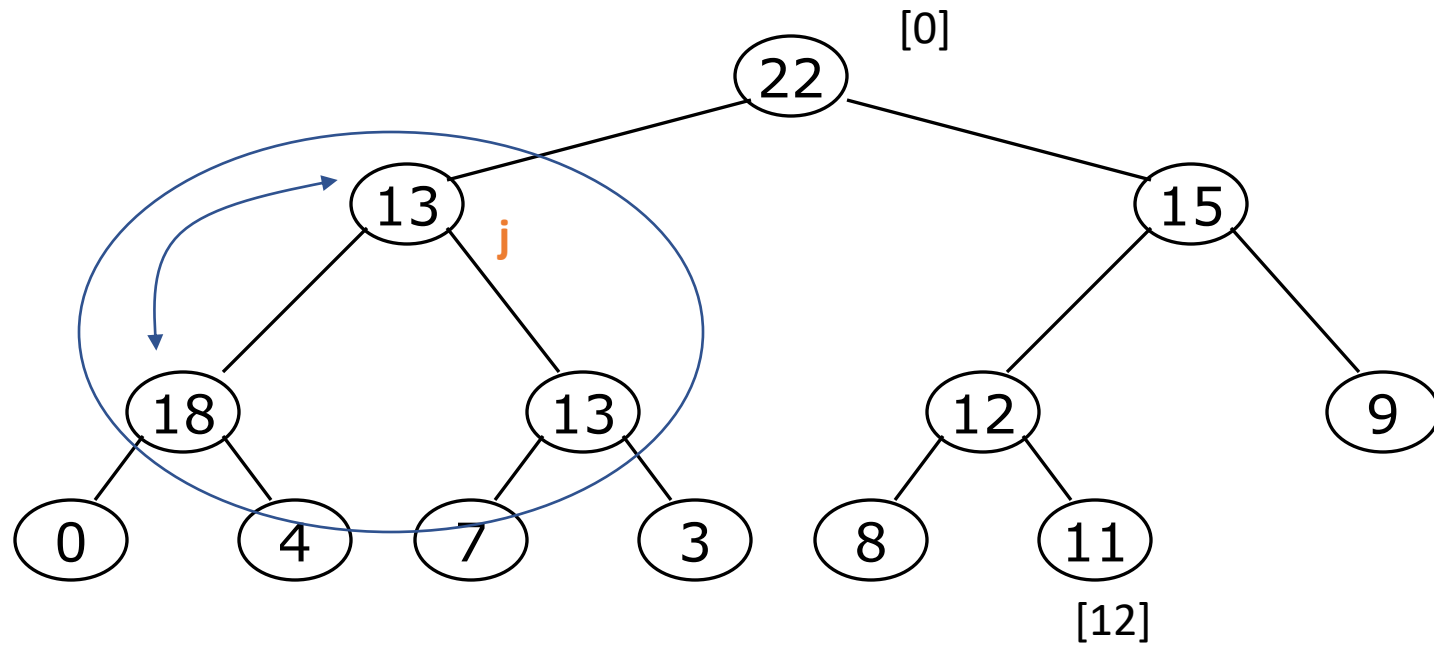
Build MAX-HEAP



Build MAX-HEAP

Input Array

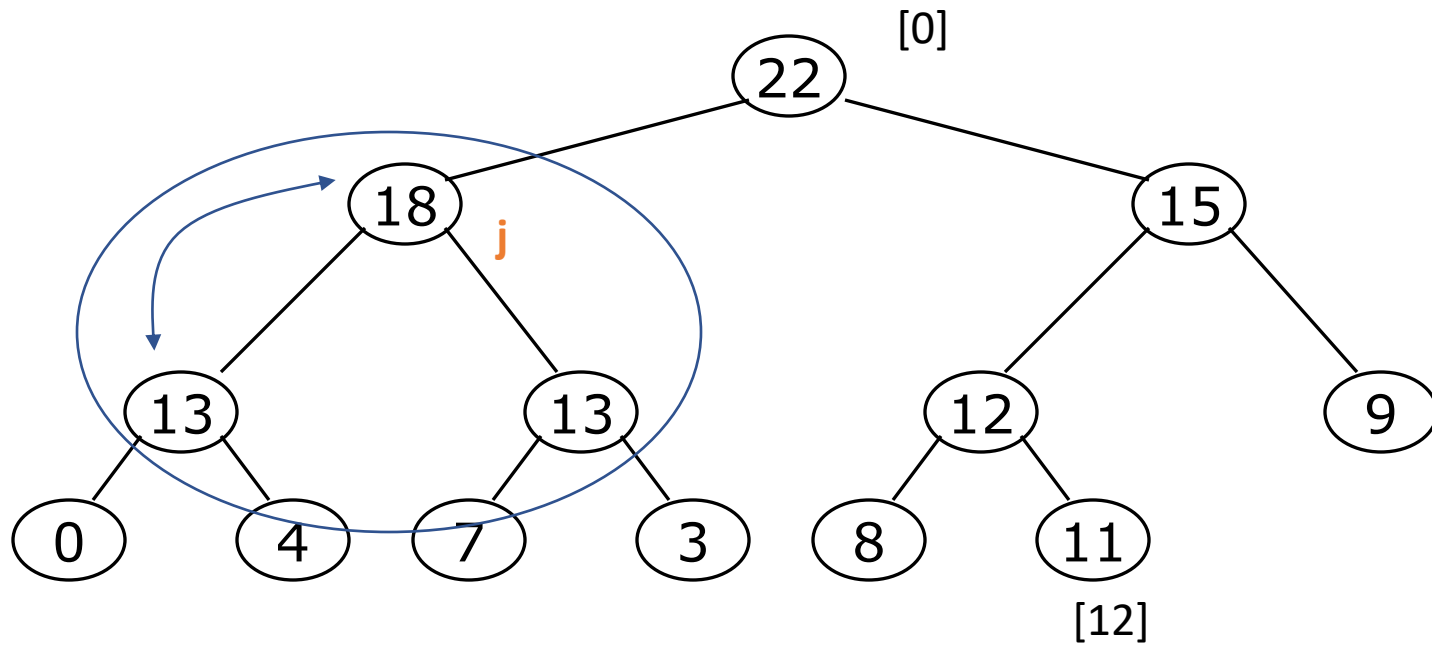
0	1	2	3	4	5	6	7	8	9	10	11	12
22	13	15	18	13	12	9	0	4	7	3	8	11



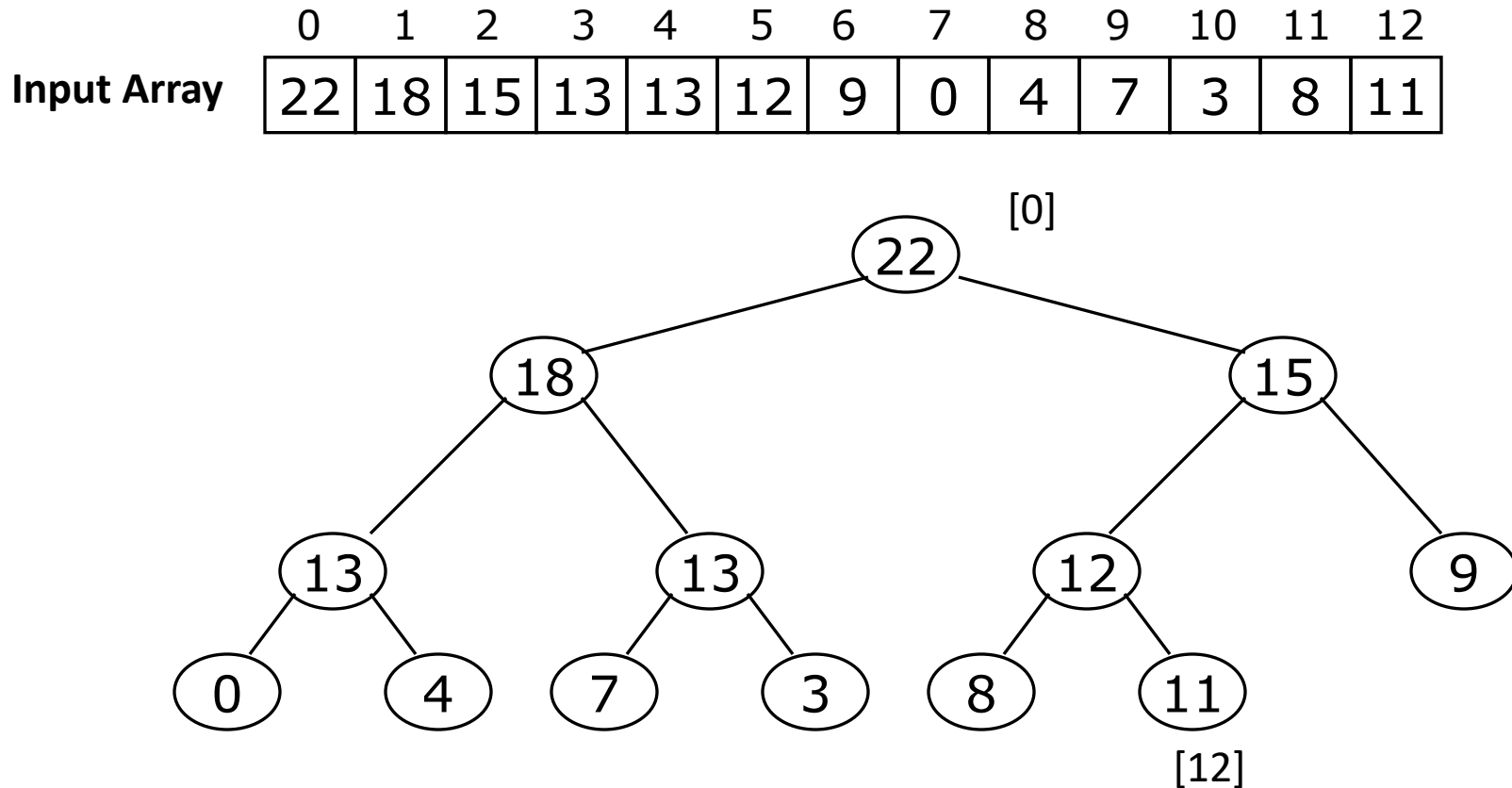
Build MAX-HEAP

Input Array

0	1	2	3	4	5	6	7	8	9	10	11	12
22	18	15	13	13	12	9	0	4	7	3	8	11



Build MAX-HEAP



The **array is a MAX-HEAP**, but not sorted, which we do next

HeapSort

To sort:

Build MAX-HEAP



while the array isn't empty

remove the root

replace the root with the **last** leaf node

MAX-HEAPIFY(A,0)

Build MAX-HEAP from an array (A):

if A.length() % 2 == 0 then MID = floor(A.length()/2)-1

else MID = floor(A.length()/2)

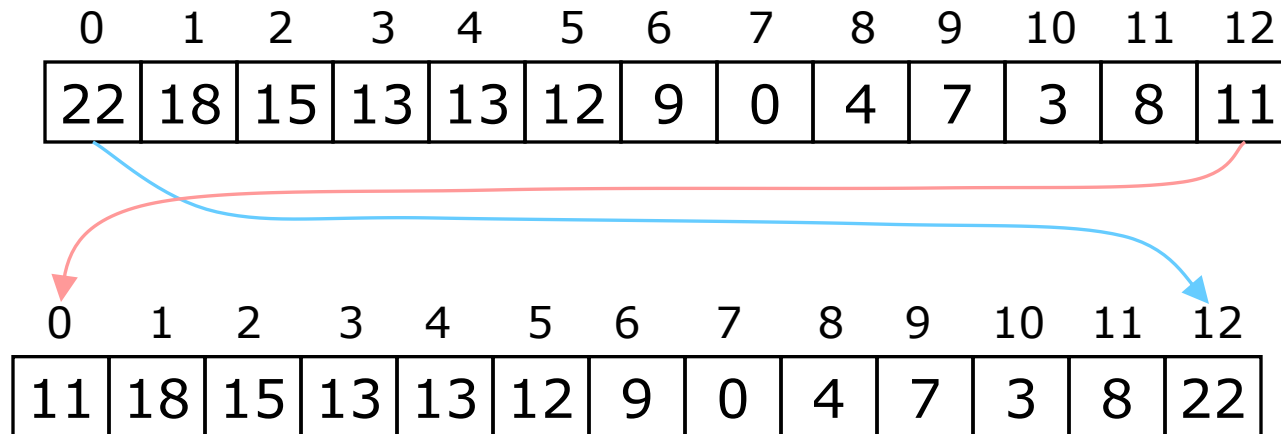
for i in MID downto 1:

MAX-Heapify(A,i)

This is called initially
when a new array is
received for sorting via
HEapSort

Removing and replacing the root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists—that is, the “last index” is **11** (9)
- **We reduce the length of the array by 1, as 22 is in its correct position**

Repeat and MAX-Heapify

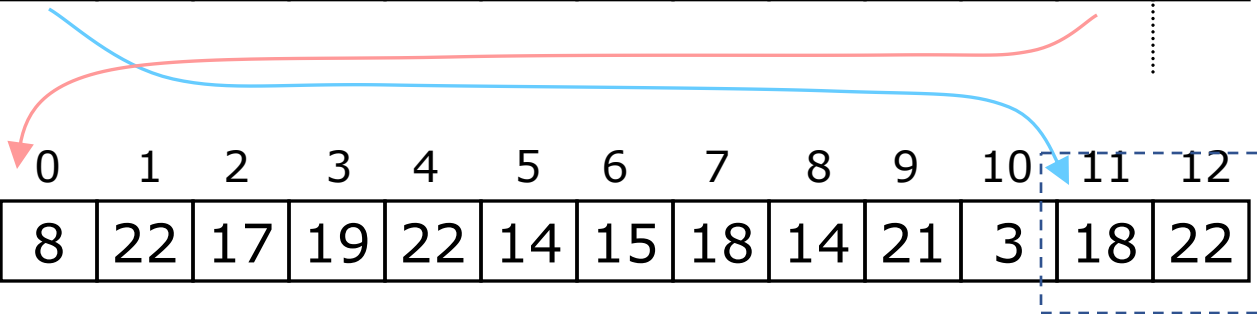
- MAX-Heapify the root node (index 0, containing 11)...

0	1	2	3	4	5	6	7	8	9	10	11	12
11	18	15	13	13	12	9	0	4	7	3	8	22



MAX-Heapification

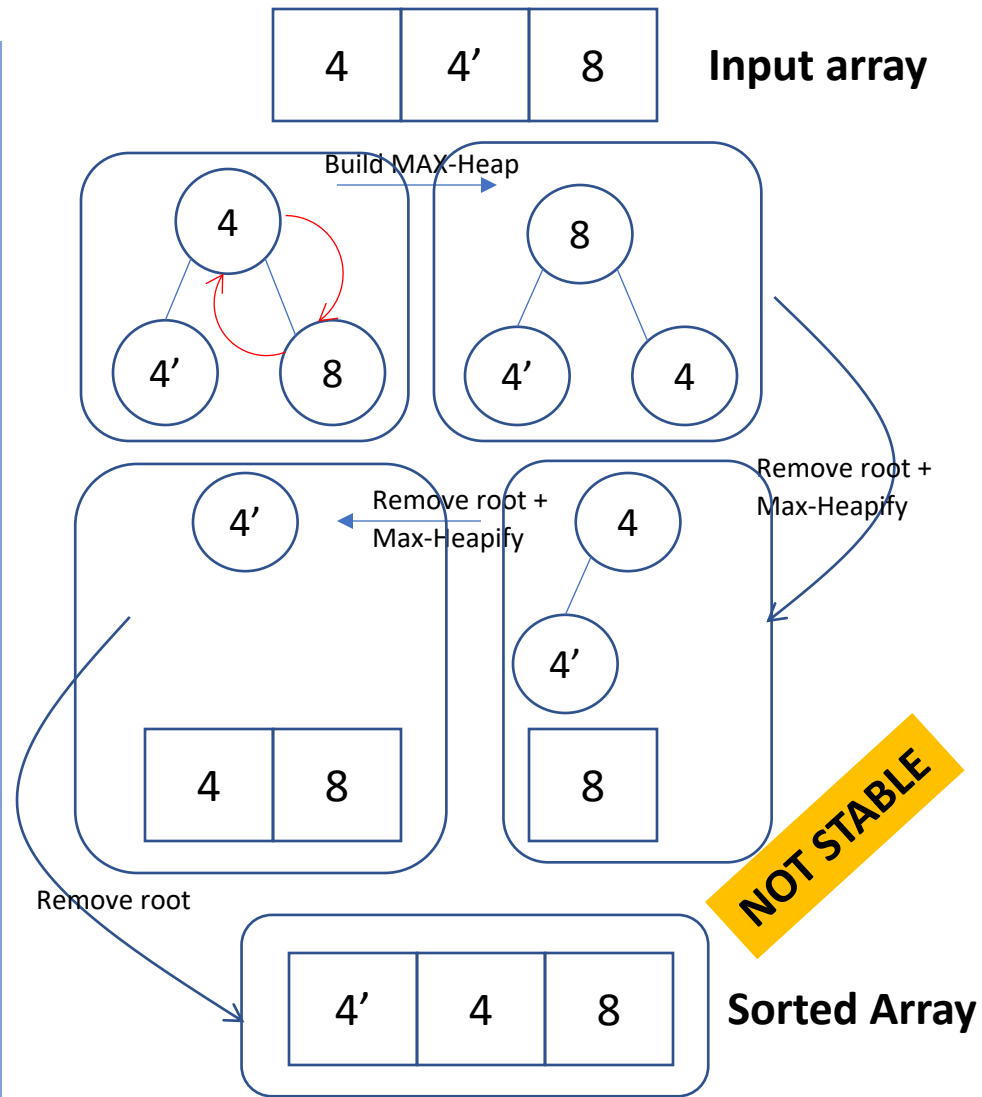
0	1	2	3	4	5	6	7	8	9	10	11	12
18	13	15	13	11	12	9	0	4	7	3	8	22



- ...And again, remove and replace the root node
- Remember, though, that the “last” array index is changed
- Repeat until the last becomes first, and the array is sorted!

Memory Requirements + Stability

- HeapSort can sort the array **in-place**
- Extra memory required during the sorting process is **$O(1)$**
 - **Only small amount needed to swap variables**



Analysis: Build MAX-HEAP

- Build MAX-Heap for the array
 - We start at the middle of the array and MAX-heapify at every node until first element of the array
 - The MAX-Heapifying itself takes
 - $O(\lg n)$ time
 - Since we do this (aprox.) $n/2$ times (i.e., $O(n)$)
 - MAX-Heapify takes $O(n) * O(\lg n)$ time
- **Running time of Build MAX-HEAP is**
 $O(n \lg n)$

Analysis: MAX-Heapify

- To MAX-Heapify the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- Therefore, this path is $O(\lg n)$ long
 - And we only do $O(1)$ operations at each node
- Therefore, MAX-Heapify takes $O(\lg n)$

Analysis: HeapSort

- Here's the rest of the algorithm:
 - while** the array isn't empty
 - remove the root
 - replace the root with the last leaf node
 - MAX-HEAPIFY(A,0)
- We do the while loop n times (actually, $n-1$ times), because we remove one of the n nodes each time
- Removing and replacing the root takes $O(1)$ time
- Therefore, the total time is n times however long it takes the **MAX-Heapify** method
 - $O(n \lg n)$

Analysis

- Here's the algorithm again:

Build MAX-HEAP

while the array isn't empty

remove the root

replace the root with the last leaf node

MAX-Heapify(A,0)

- We have seen that Build MAX-HEAP takes $O(n \lg n)$ time
 - The while loop (with MAX-Heapify) takes $O(n \lg n)$ time
 - The total time is therefore $O(n \lg n) + O(n \lg n)$
-
- **HeapSort therefore takes: $O(n \log n)$ time**



That's all Folks!
Any Question?