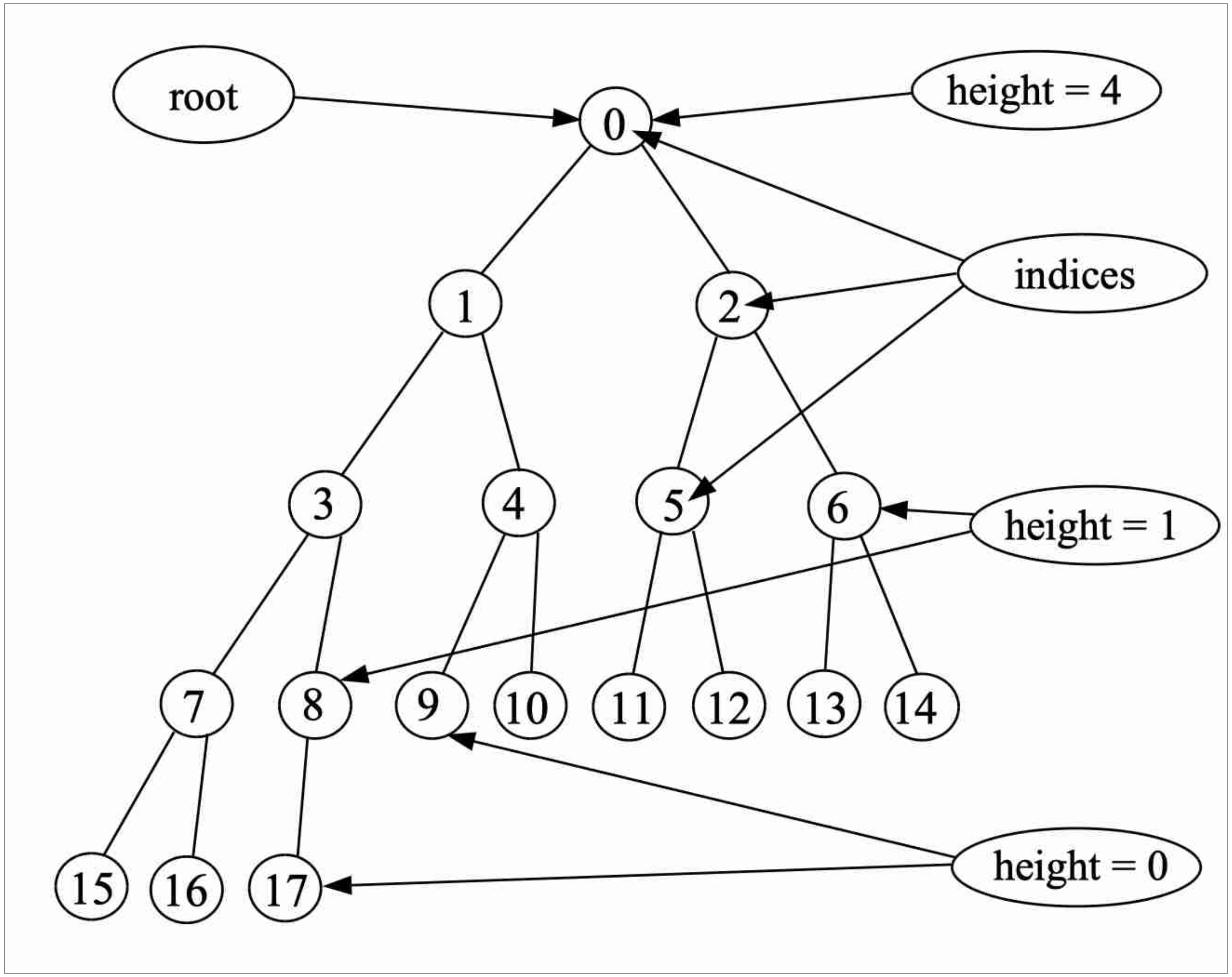
Heaps

These notes may contain undocumented references to “Introduction To Algorithms”, Cormen, Leiserson, Rivest.

- A heap is an array that can be viewed as a nearly complete binary tree
  - By a nearly complete binary tree, we mean a binary tree that is complete, except for possibly the lowest level, which is filled from the left up to a certain point
  - Image on the next slide
- In our heaps, we will assume that the array is indexed from 0 to n – 1, where the heap has n elements
- Each node in the tree corresponds to an array element

- The numbers in the nodes in the following image are the indices in the array that holds the values
- The height of the tree is the height of the root



- The root is the highest node (greatest height), and the nodes without any subnodes (children), are the lowest nodes (height of 0)
- For nodes other that the root, the parent of the i-th node is defined to be the (i-1)/2 node
  - That is, parent(i) = (i-1)/2
    - (i-1)/2 is the floor((i-1)/2), which is the largest integer ≤ (i-1)/2
  - The root node does not have a parent

- The left child of the i-th node is defined to be the (2i) +1 node
  - That is, left(i) = (2i)+1
  - If the (2i)+1 node is not defined ((2i)+1 > n – 1), because the heap is not that large, then the node does not have a left child
- The right child of the i-th node is defined to be the 2(i+1) node
  - That is, right(i) = 2(i+1)
  - If the 2(i+1) node is not defined (2(i+1) > n – 1), because of the size of the heap, then the node does not have a right child

- If a node has a right child, then is must have also a left child
- If a node has a left child, it does not necessarily have a right child

- The height of a node is the maximum number of edges that it takes to connect the node from a lowest node to the node
- To find the height of a node, we can count the number of left children recursively until there are no more left children
  - We can do it this way, because the heap is always filled from the left to the right
- The height of the heap is the height of the root node, which is the maximum height of any node in the tree
- A heap of height h will have between 2^h and 2^{h+1}-1 values
- A heap with n values has a height of floor of $\lg_2 n$

- There are two types of heaps
  - Maximum (max) heaps
  - Minimum (min) heaps

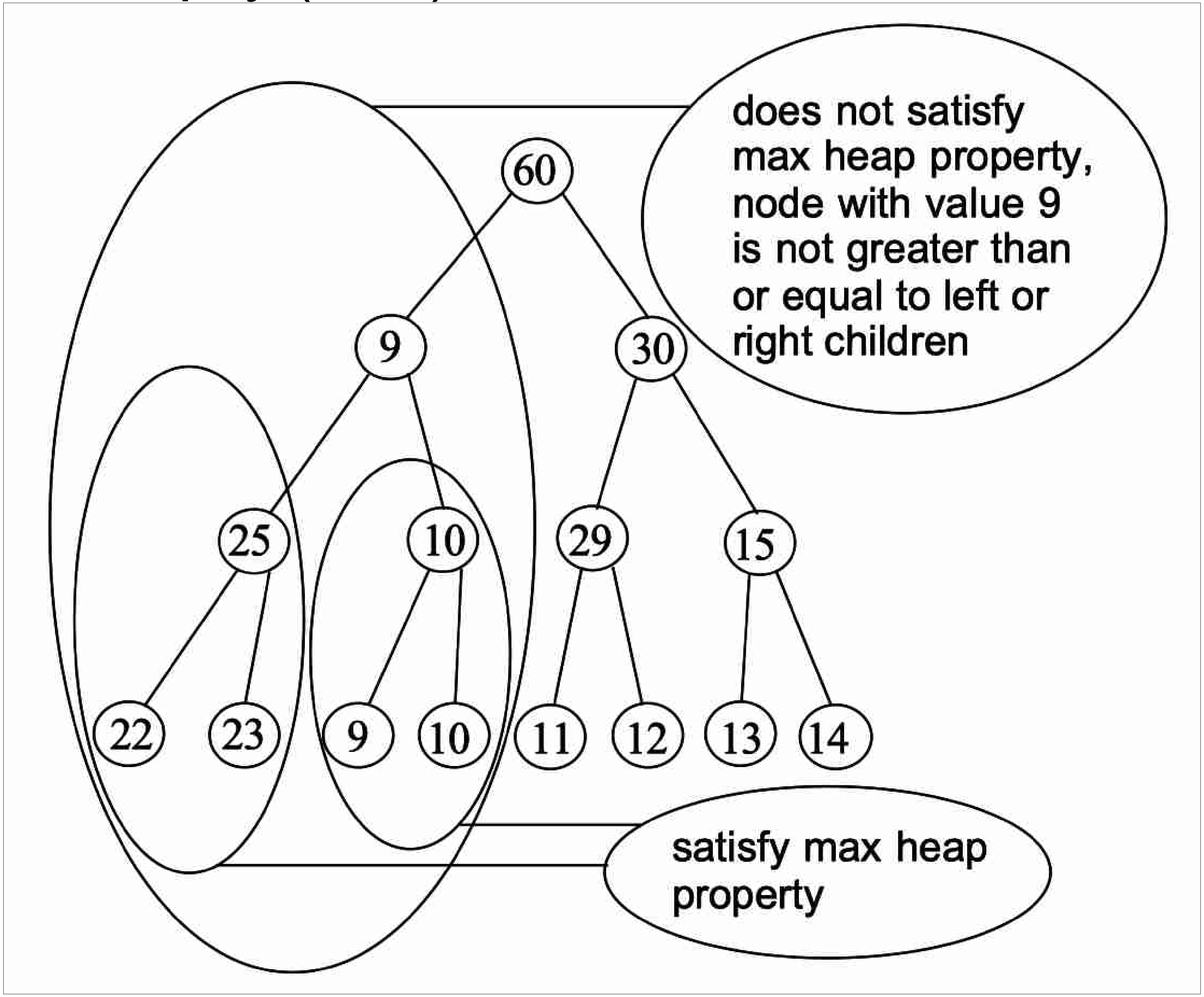
- The heap property
  - For a max heap, the heap property is A[parent(i)] ≥ A[i], where A is the array that stores the heap data and i is any node other than the root node
    - Since this is true for all non-root nodes of the heap, we have that the root node of the heap has the maximum value of the heap
  - For a min heap, the heap property is A[parent(i)] ≤ A[i], where A is the array that store the heap data and i is any node other than the root node
    - Since this is true for all non-root nodes of the heap, we have that the root node of the heap has the minimum value of the heap
  - All heaps must satisfy the min heap or max heap property
    - A heap that satisfies both has all values the same

- Sub trees
  - By the heap property, we have that any node is either the maximum (or minimum) of all of it's children and their children
  - Thus all sub trees of a heap are also heaps, since they also must satisfy the same heap property that the original heap satisfies

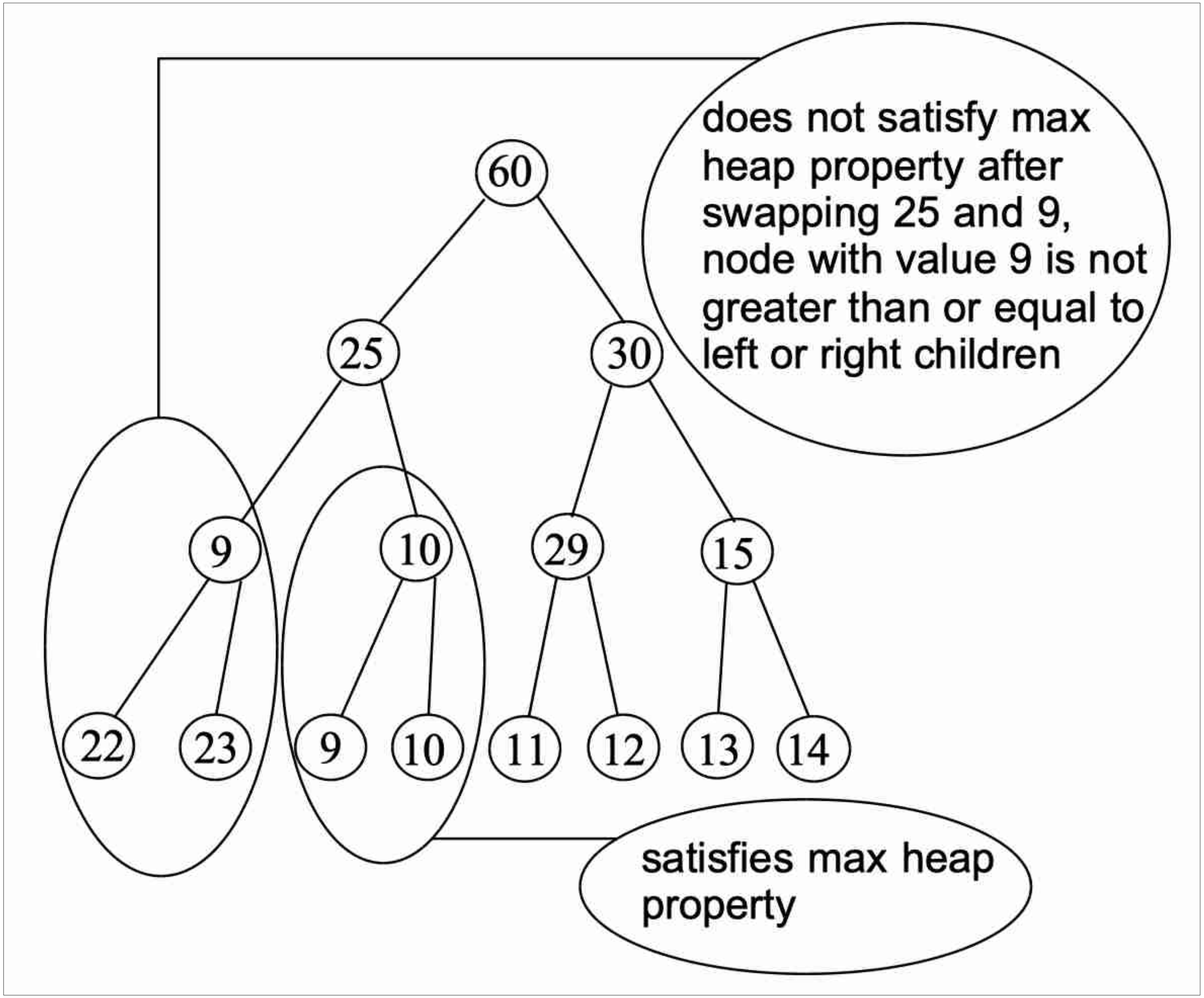
- Max heapify
  - The function max-heapify is used to maintain the heap property
  - The function takes two parameters, the heap and a node
  - Since we are maintaining the heap property, we are assuming the sub trees rooted at the left and right children of the node satisfy the heap property
  - We are not assuming anything about how the node compares to it's left and right children
  - Thus we are not assuming that the tree with root equal to the node satisfies the heap property, just it's sub trees

- Max heapify (cont)
  - The basic idea of the max-heapify function is to replace the node passed as a parameter with the larger of the node, and it's left and right children
  - If the node is greater than or equal to it's left and right children, then the sub tree rooted at the node satisfies the heap property
  - If one of the children is the maximum, then the node and the maximum child are swapped
  - When this happens, the sub tree that was rooted by the child may no longer satisfy the heap property, thus the function is recursively called on the child, which now has the original node as the root

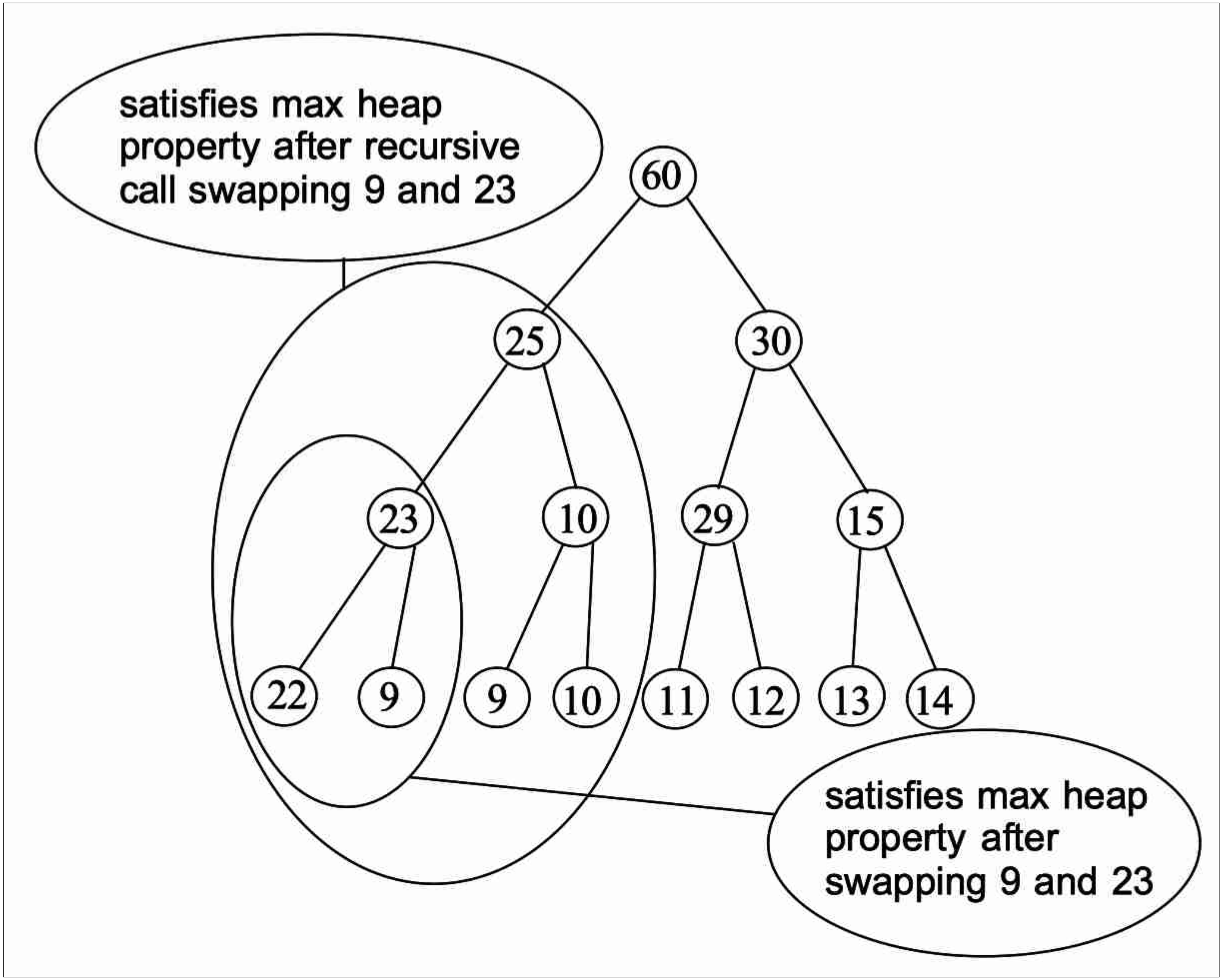
- Max heapify (cont)


60
9
30
25
10
29
15
22
23
9
10
11
12
13
14
does not satisfy max heap property, node with value 9 is not greater than or equal to left or right children
satisfy max heap property

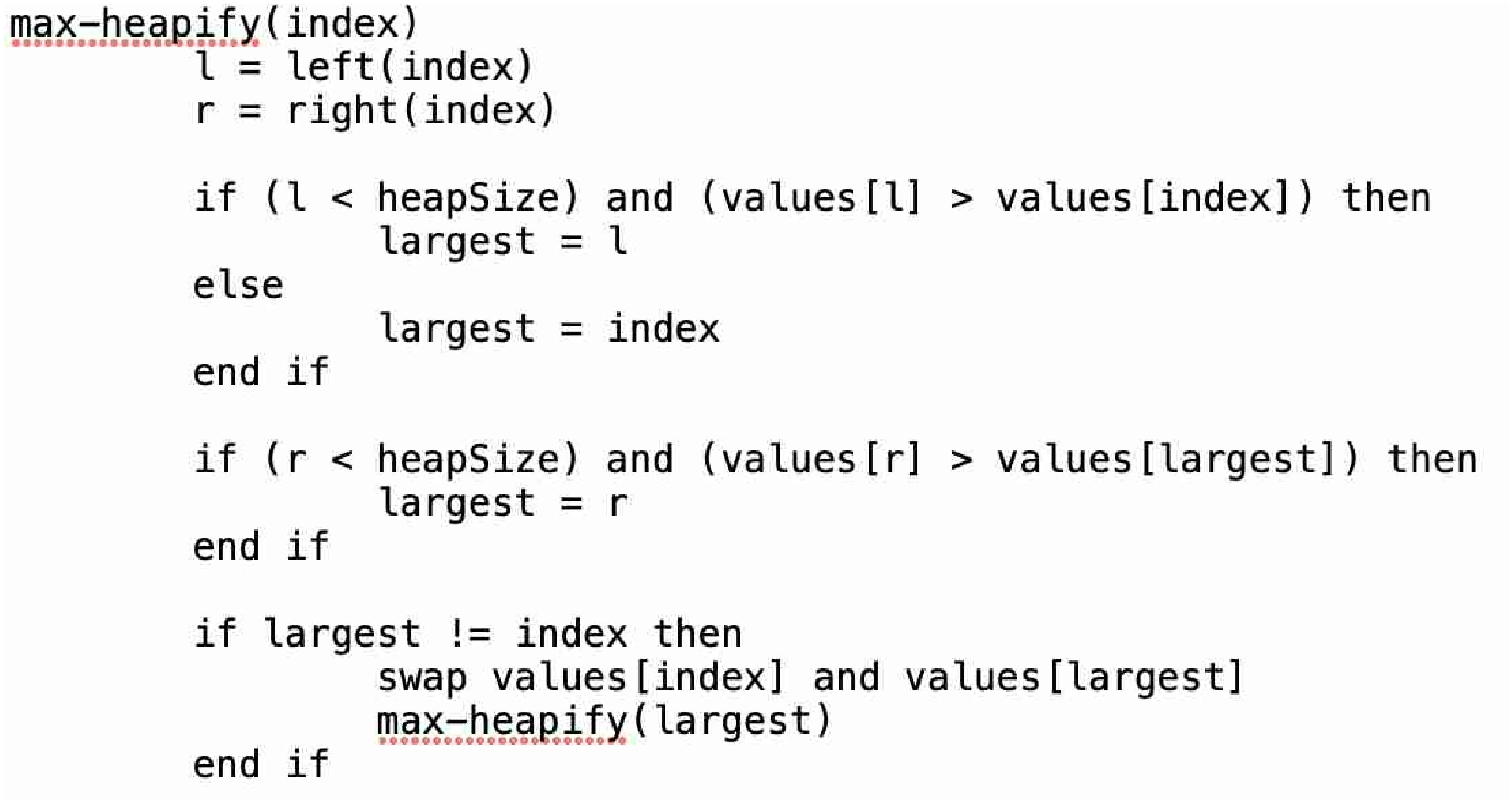
- Max heapify (cont)


does not satisfy max heap property after swapping 25 and 9, node with value 9 is not greater than or equal to left or right children
satisfies max heap property
60
25
30
9
10
29
15
22
23
9
10
11
12
13
14

- Max heapify (cont)


satisfies max heap property after recursive call swapping 9 and 23
60
25
30
23
10
29
15
22
9
9
10
11
12
13
14
satisfies max heap property after swapping 9 and 23

- Max heapify pseudo code



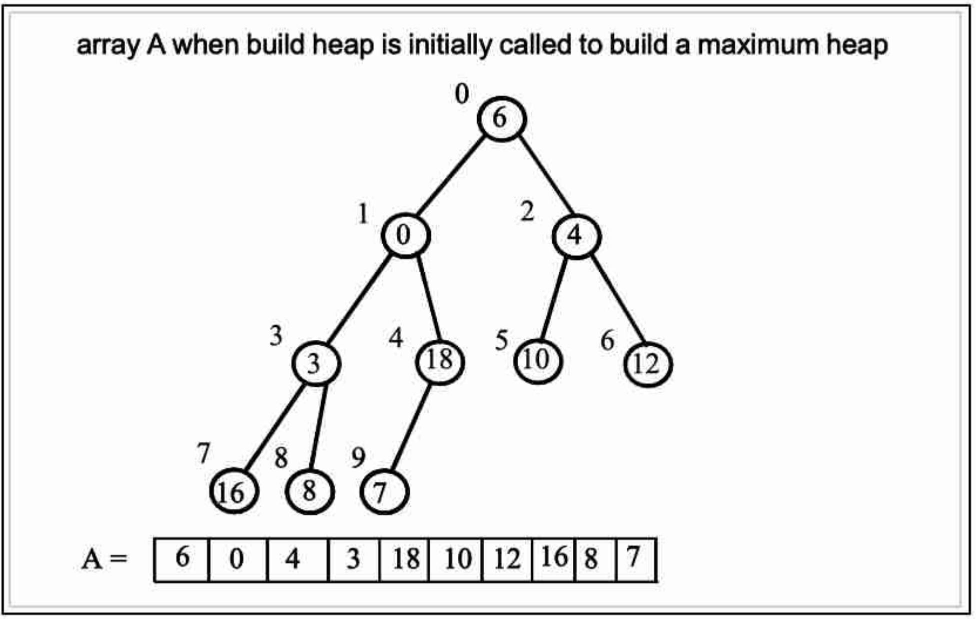
```
max-heapify(index)
        l = left(index)
        r = right(index)

        if (l < heapSize) and (values[l] > values[index]) then
                largest = l
        else
                largest = index
        end if

        if (r < heapSize) and (values[r] > values[largest]) then
                largest = r
        end if

        if largest != index then
                swap values[index] and values[largest]
                max-heapify(largest)
        end if
```

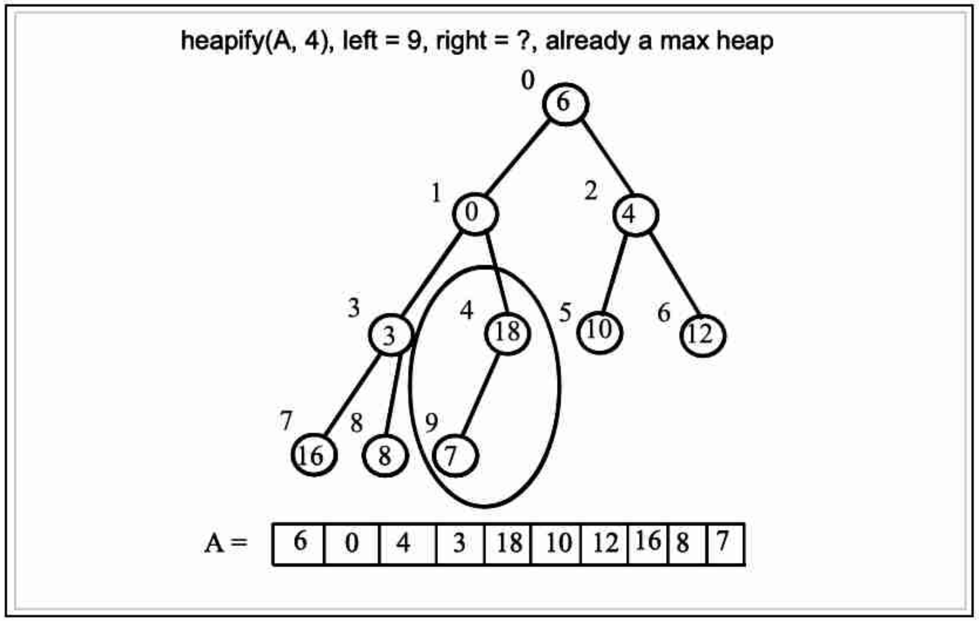
- Building a max heap
- Assuming we are given an array of size n, we wish to make a heap of size n
  - We can use the heapify function to do this
- The nodes at indices n/2, n/2 + 1, ... , n-1 are leaves
- Since a leaf node is already a heap, we can call heapify on the parents of the leaves, then call heapify on the parents of the parents, until we finally reach the root of the heap
- Once we have called heapify on the root of the heap, all of the nodes satisfy the heap property and we have a heap

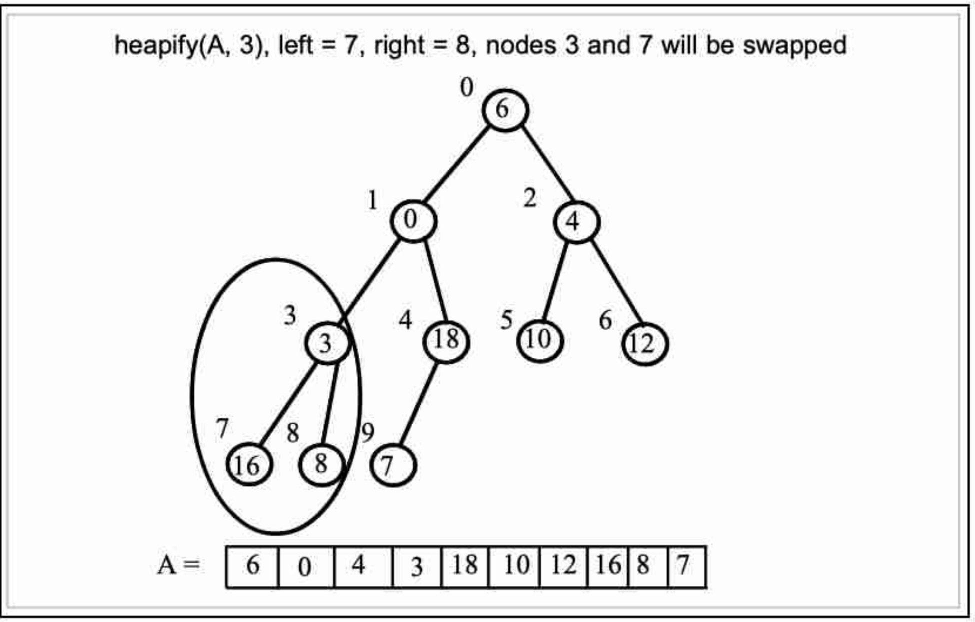
- Pseudo code for build max heap

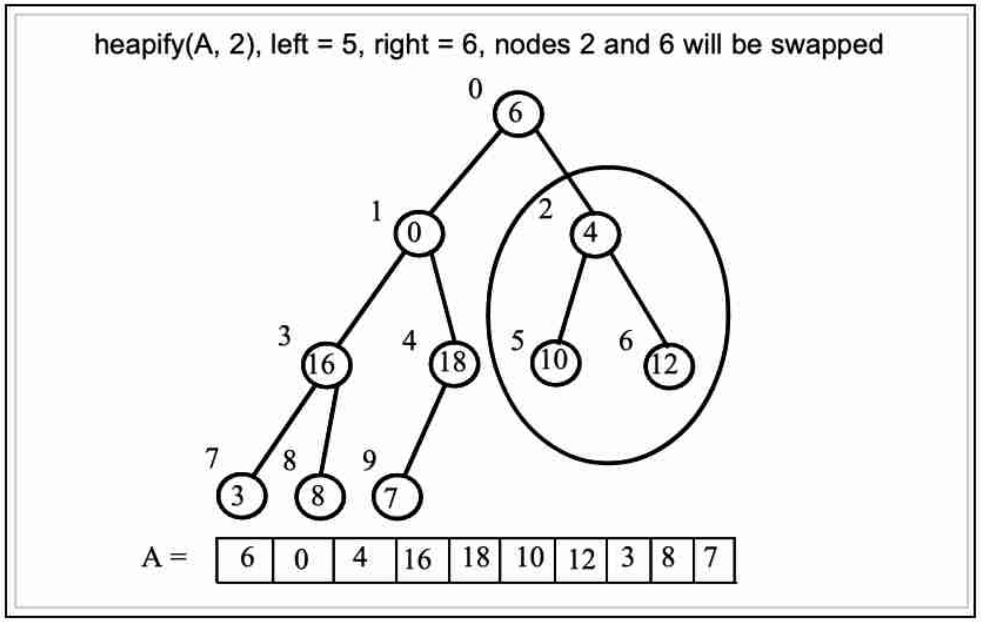
```
buildMaxHeap(A)
        heapSize = A.length()
        i = (A.length/2) – 1
        while( i ≥ 0 )
                max-heapify(A, i)
                i = i – 1
        while end
```

- Example of build max heap

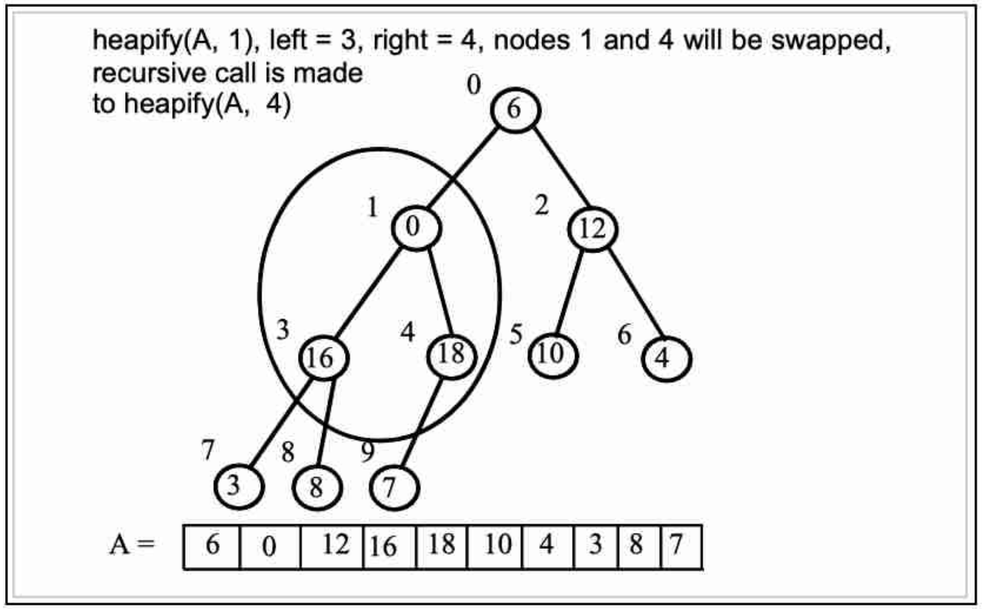

array A when build heap is initially called to build a maximum heap
A = 6 0 4 3 18 10 12 16 8 7


heapify(A, 4), left = 9, right = ?, already a max heap
A = 6 0 4 3 18 10 12 16 8 7


heapify(A, 3), left = 7, right = 8, nodes 3 and 7 will be swapped
A = 6 0 4 3 18 10 12 16 8 7

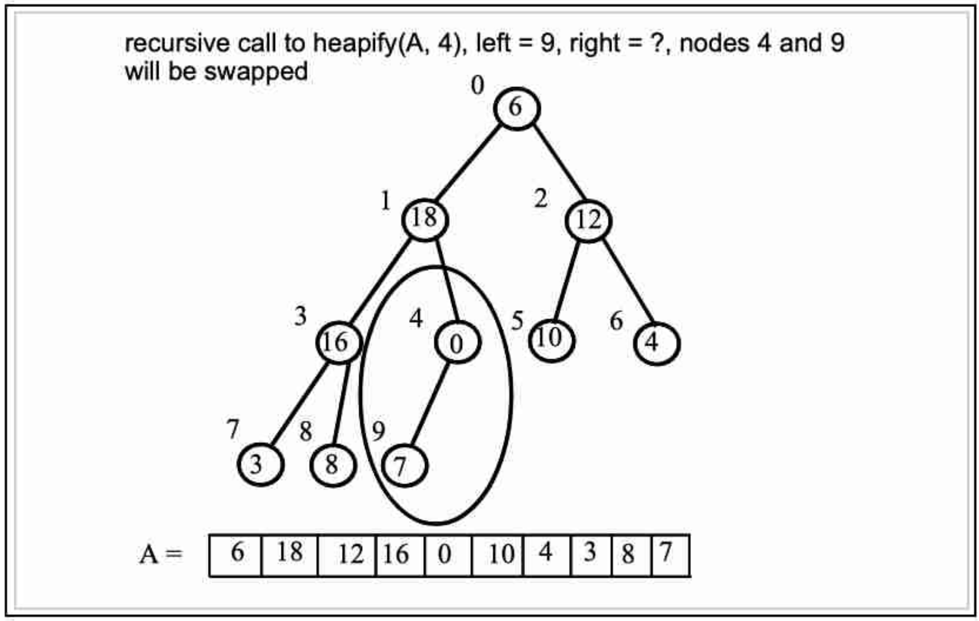

heapify(A, 2), left = 5, right = 6, nodes 2 and 6 will be swapped
A = 6 0 4 16 18 10 12 3 8 7

• Example of build max heap (cont)


heapify(A, 1), left = 3, right = 4, nodes 1 and 4 will be swapped, recursive call is made to heapify(A, 4)

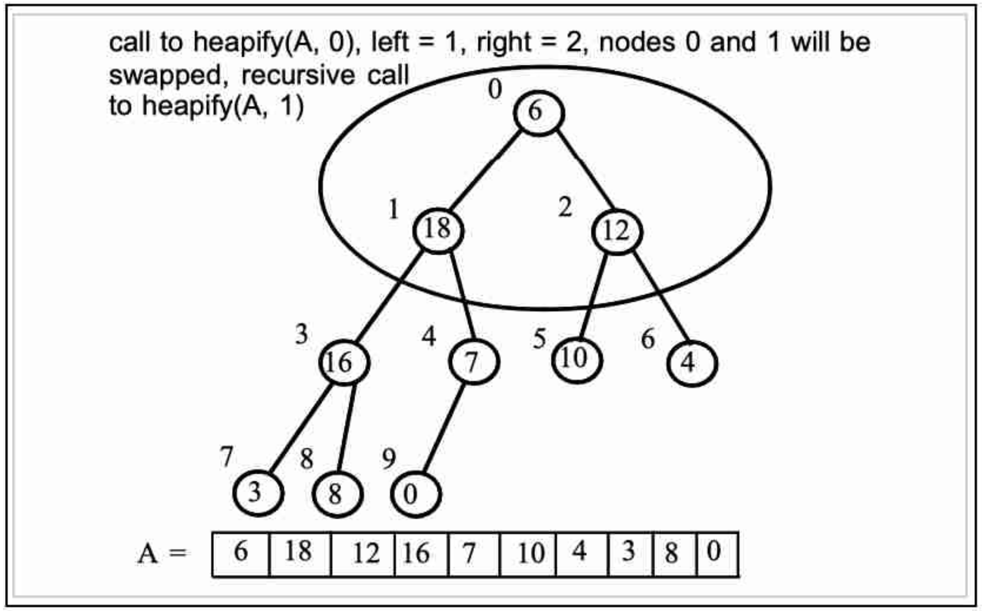
A =

6	0	12	16	18	10	4	3	8	7


recursive call to heapify(A, 4), left = 9, right = ?, nodes 4 and 9 will be swapped

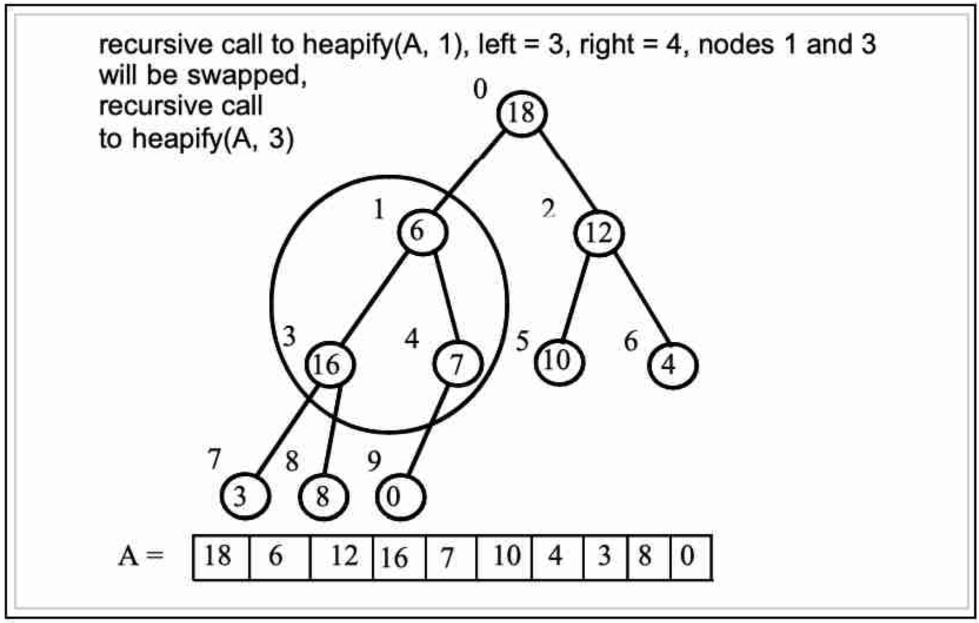
A =

6	18	12	16	0	10	4	3	8	7


call to heapify(A, 0), left = 1, right = 2, nodes 0 and 1 will be swapped, recursive call to heapify(A, 1)

A =

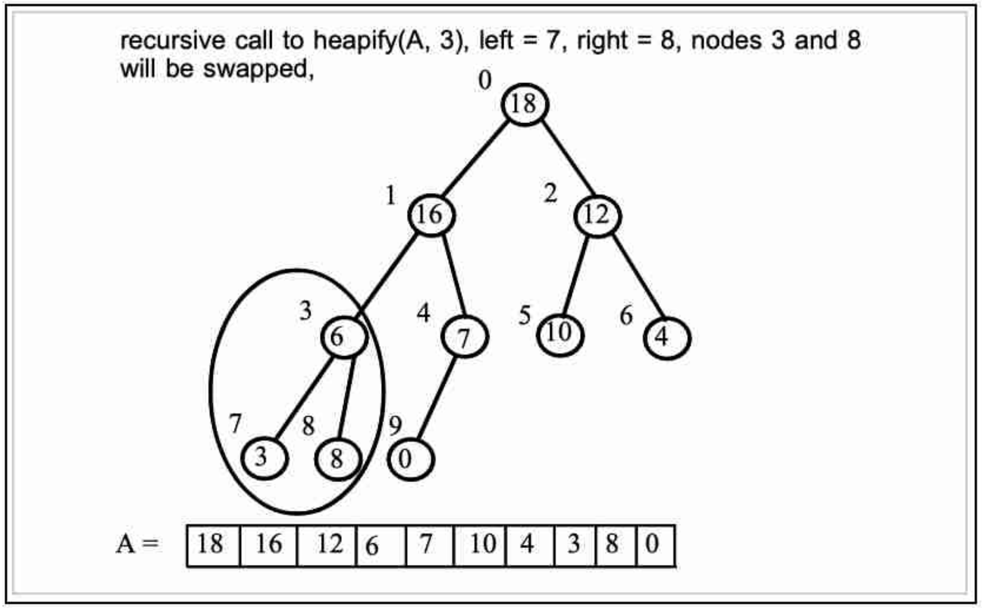
6	18	12	16	7	10	4	3	8	0


recursive call to heapify(A, 1), left = 3, right = 4, nodes 1 and 3 will be swapped, recursive call to heapify(A, 3)

A =

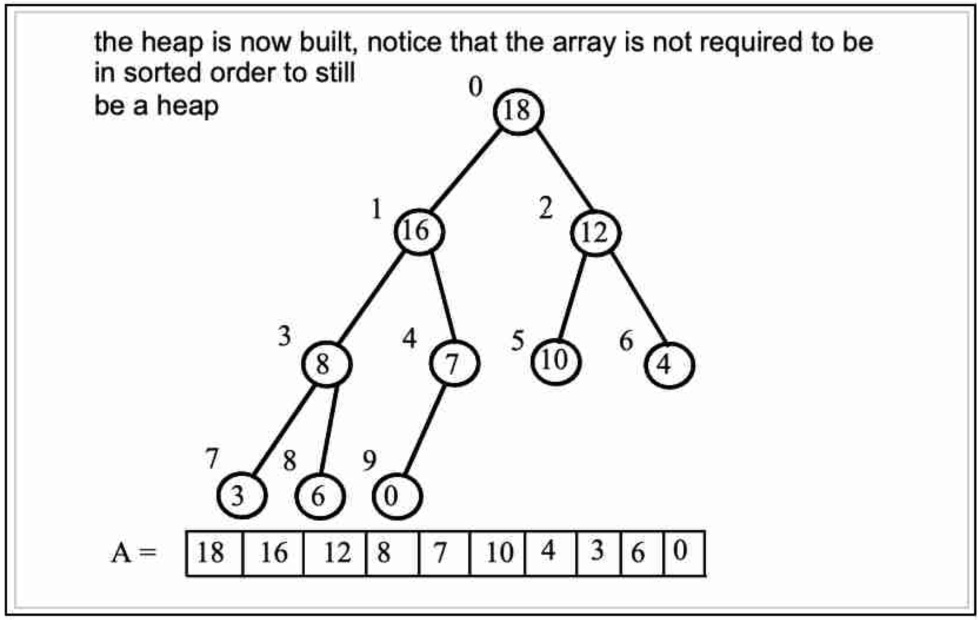
18	6	12	16	7	10	4	3	8	0

- Example of build max heap (cont)



recursive call to heapify(A, 3), left = 7, right = 8, nodes 3 and 8 will be swapped,
0
18
1
16
2
12
3
6
4
7
5
10
6
4
7
3
8
8
9
0
A =

18	16	12	6	7	10	4	3	8	0



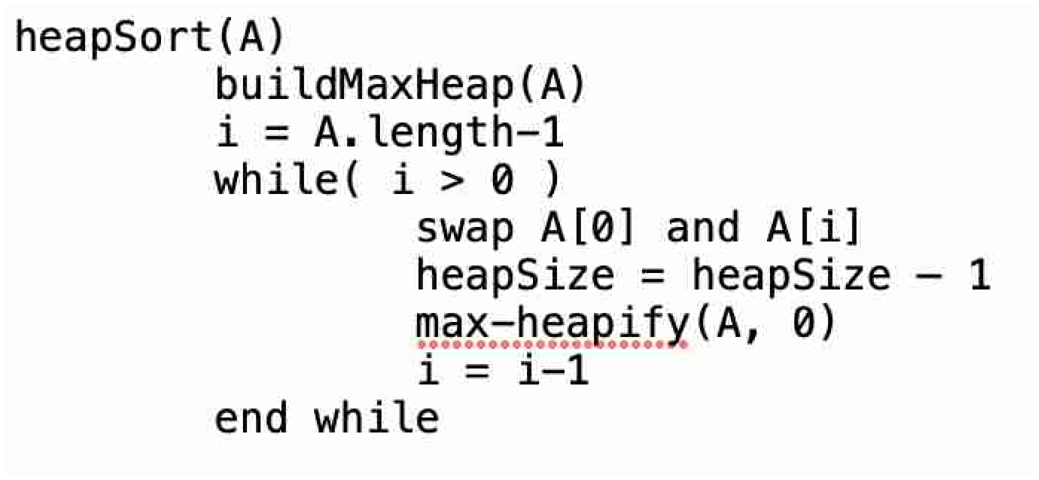
the heap is now built, notice that the array is not required to be in sorted order to still be a heap
0
18
1
16
2
12
3
8
4
7
5
10
6
4
7
3
8
6
9
0
A =

18	16	12	8	7	10	4	3	6	0

- Heap sort

- The heapsort algorithm starts with an array A[0..n-1], where n is the length of A
- We build a max heap using the buildMaxHeap function
- Once we have a heap, we have the element with the maximum value at A[0]
- By swapping the the values/nodes at indexes 0 and n-1, we now have the maximum value at index n-1 (the correct position) and an arbitrary element at index 0
- At the same time we reduce the heapSize by 1
- Now, all subtrees are still heaps, so all we need to do is call heapify(A, 0) (heapify on the root)
- Once we have done this we have a (max) heap of size n-1
- Once again, we can swap the values/nodes at indices 0 and n-2
- This will put the new largest value at index n-2, and an arbitrary element at index 0
- We also reduce the heapSize by 1
- We continue this process until we have a heap of size 1, at this point, we have sorted A in ascending order and we are done

- Heap sort pseudo code



```
heapSort(A)
        buildMaxHeap(A)
        i = A.length-1
        while( i > 0 )
                swap A[0] and A[i]
                heapSize = heapSize - 1
                max-heapify(A, 0)
                i = i-1
        end while
```

- Heap sort has O(n lg n)

- Priority queue
  - A heap can be used as a priority queue
    - Min heap for a min priority queue
    - Max heap for a max priority queue
  - Getting the minimum or maximum value is constant time (the value at index 0)
  - To extract the minimum or maximum value takes O(lg n)
    - Return the value at index 0
    - Replace the value at index 0 with the value at index n-1
    - Reduce the heap size
    - Call heapify on the new root