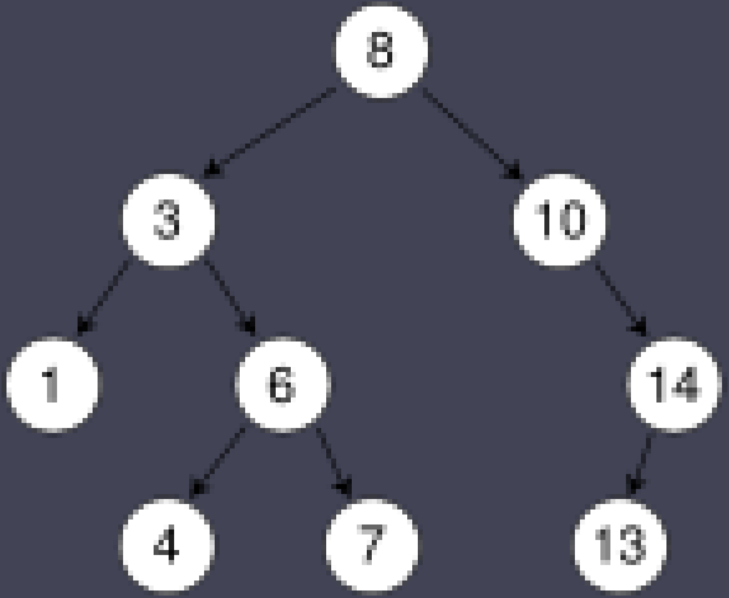
Introduction to Algorithms Lecture 6

By: Darmen Kariboz

Outline

- Binary Search Tree
- Heap Sort
- Storage array
- Build Heap
- Heapify



Binary Search Tree

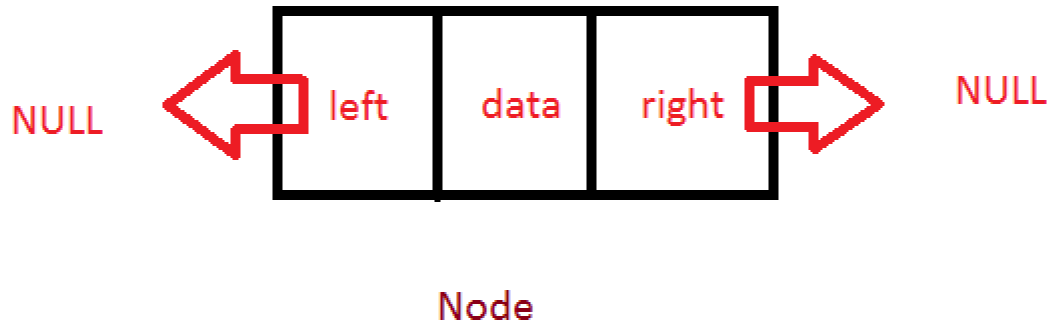
Smaller elements go left, bigger elements go right.

Binary Tree consists of Nodes

- Nodes are nothing but objects of a class and each node has data and a link to the left node and right node.

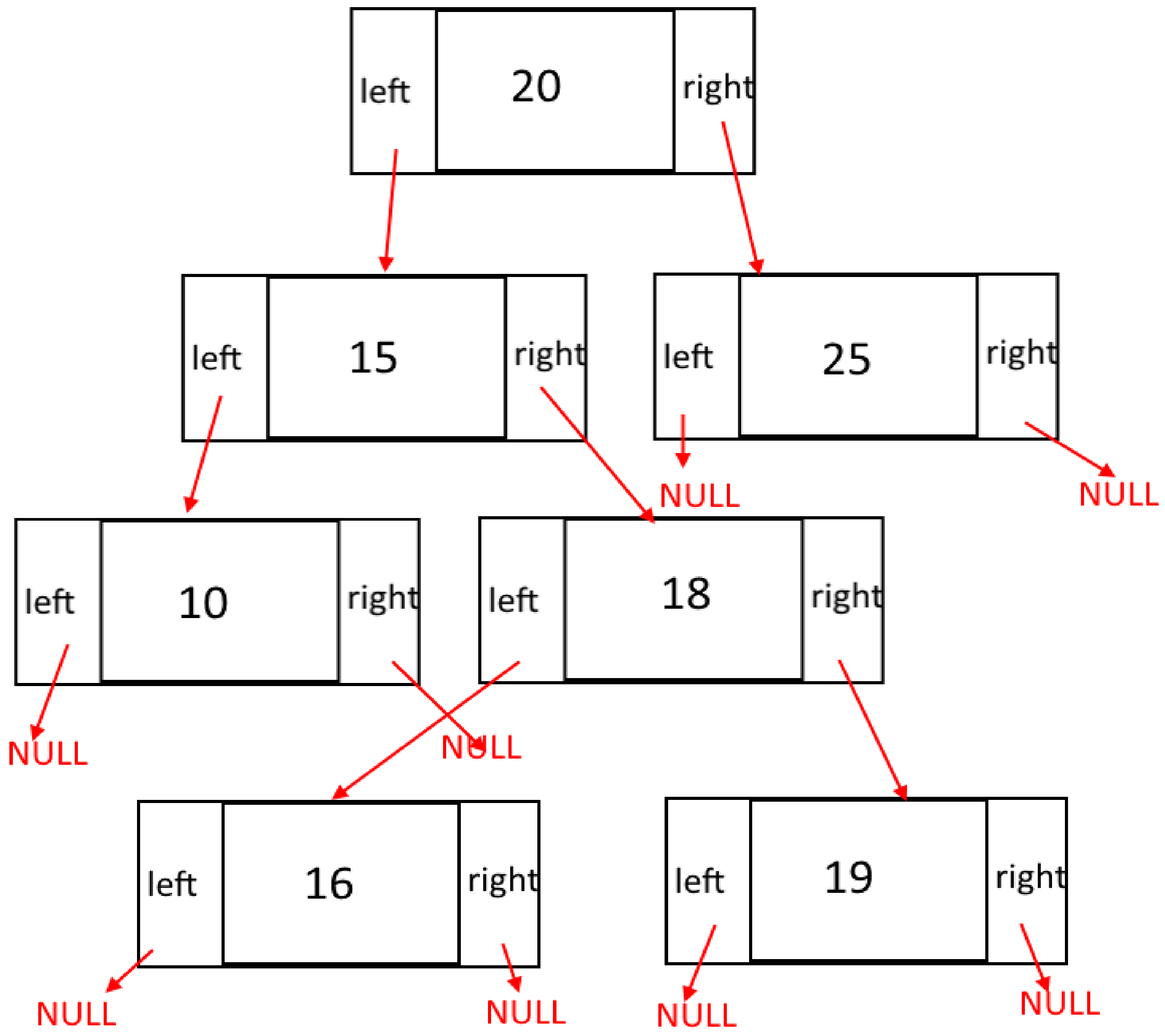
- Usually we call the starting node of a tree as *root*.

- Left and right node of a Leaf node points to NULL so you will know that you have reached to the end of the tree.



Binary Search Tree

- Parent have two children, left is smaller, right is bigger.
- Immediately sorted.
- In worst case it can be of depth N

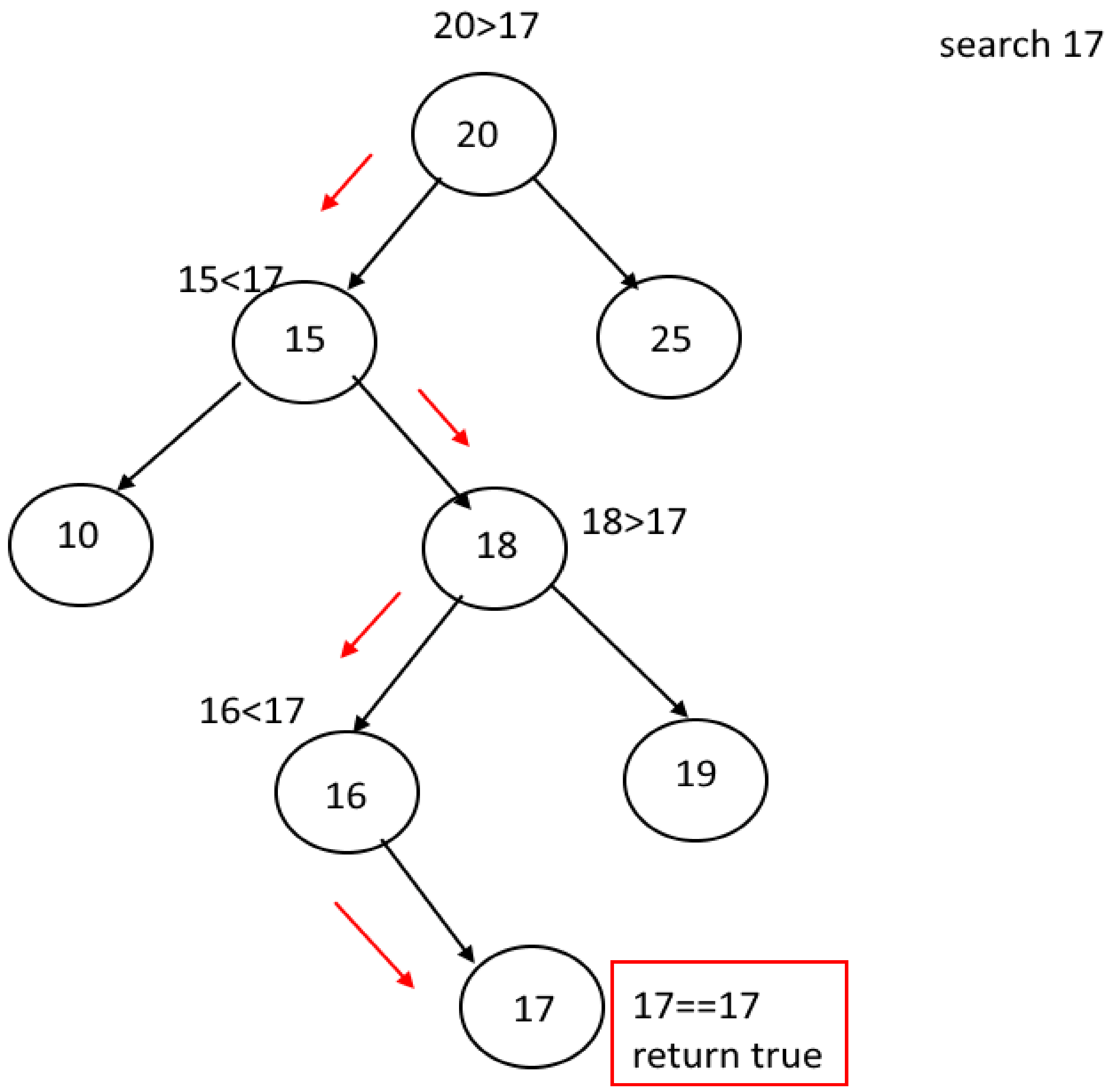

left
20
right
left
15
right
left
25
right
NULL
NULL
left
10
right
left
18
right
NULL
NULL
left
16
right
left
19
right
NULL
NULL
NULL
NULL

Operations:

- **Insert(int n) :** Add a node with value n. Its O(lgn)
- **Find(int n) :** Find a node with value n. Its O(lgn)
- **Delete (int n)** : Delete a node with value n. Its O(lgn)
- **Display**(): Prints the entire tree in increasing order. O(n).

Find(int n):

- Its very simple operation to perform.
- start from the root and compare root.data with n
- if root.data is greater than n that means we need to go to the left of the root.
- if root.data is smaller than n that means we need to go to the right of the root.
- if any point of time root.data is equal to the n then we have found the node, return true.
- if we reach to the leaves (end of the tree) return false, we didn’t find the element

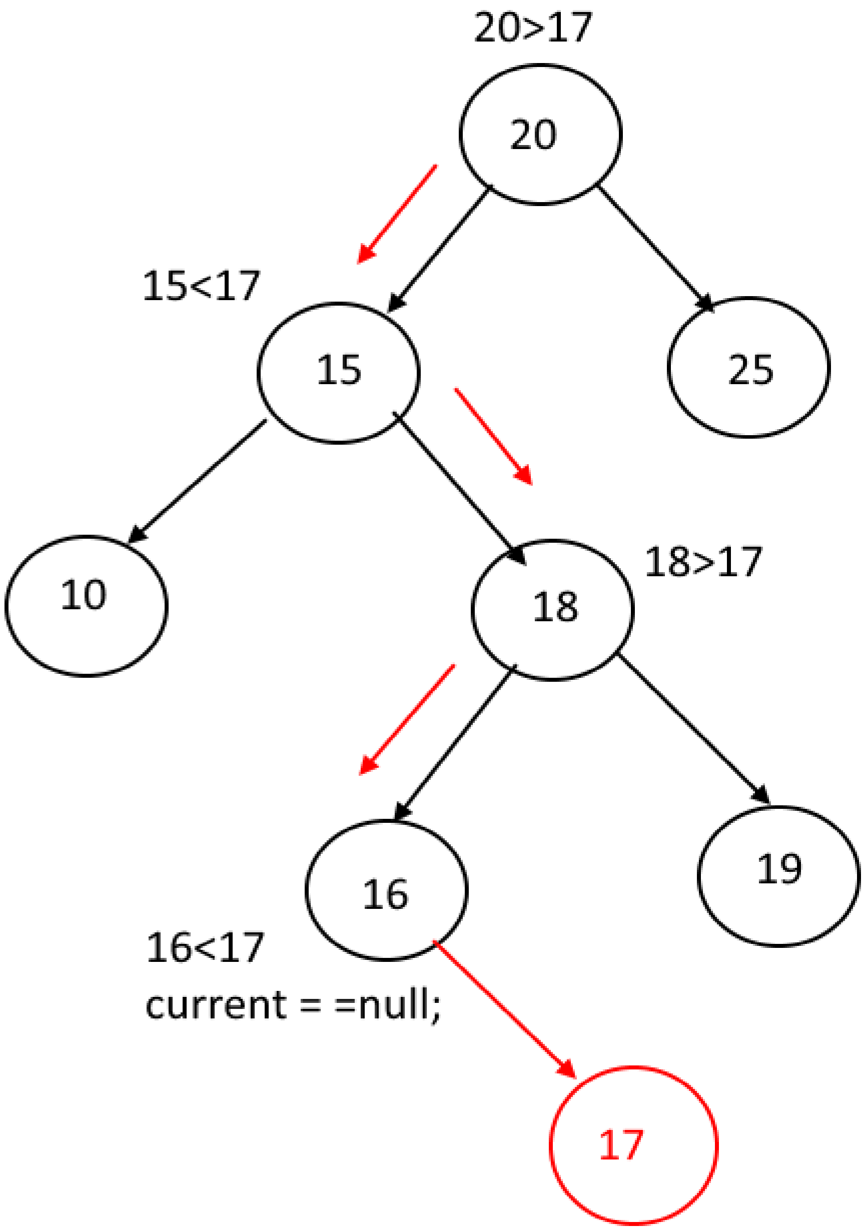


search 17
20>17
20
15<17
15
25
10
18
18>17
16<17
16
19
17
17==17
return true

Insert(int n):

- Very much similar to find() operation.
- To insert a node our first task is to find the place to insert the node.
- Take current = root .
- start from the current and compare root.data with n
- if current.data is greater than n that means we need to go to the left of the root.
- if current.data is smaller than n that means we need to go to the right of the root.
- if any point of time current is null that means we have reached to the leaf node, insert your node here with the help of parent node.

Insert 17



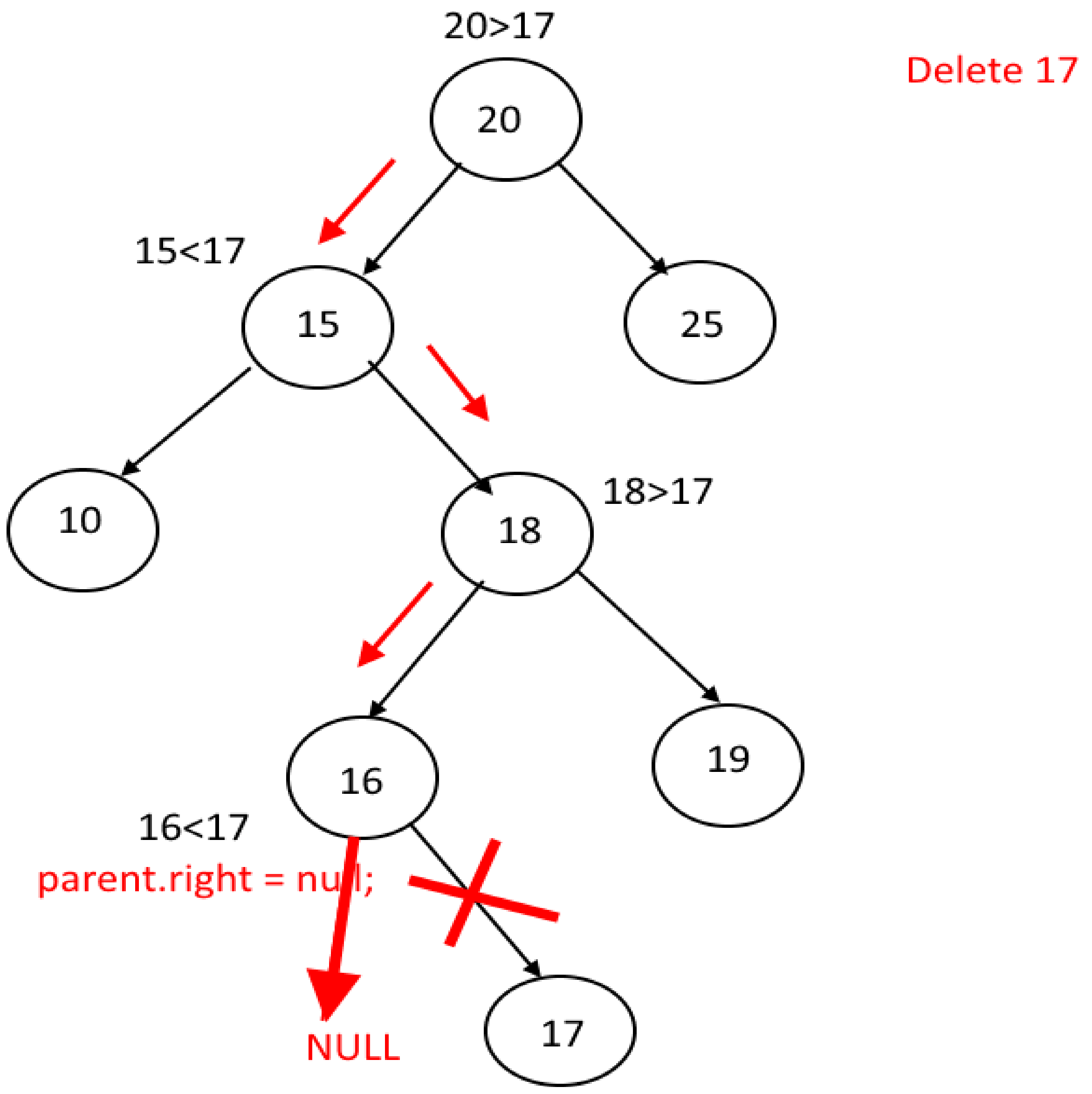
Delete(int n):

Complicated than Find() and Insert() operations. Here we have to deal with 3 cases.

- Node to be deleted is a leaf node (No Children).
- Node to be deleted has only one child.
- Node to be deleted has two childrens.

Node to be deleted is a leaf node (No Children).

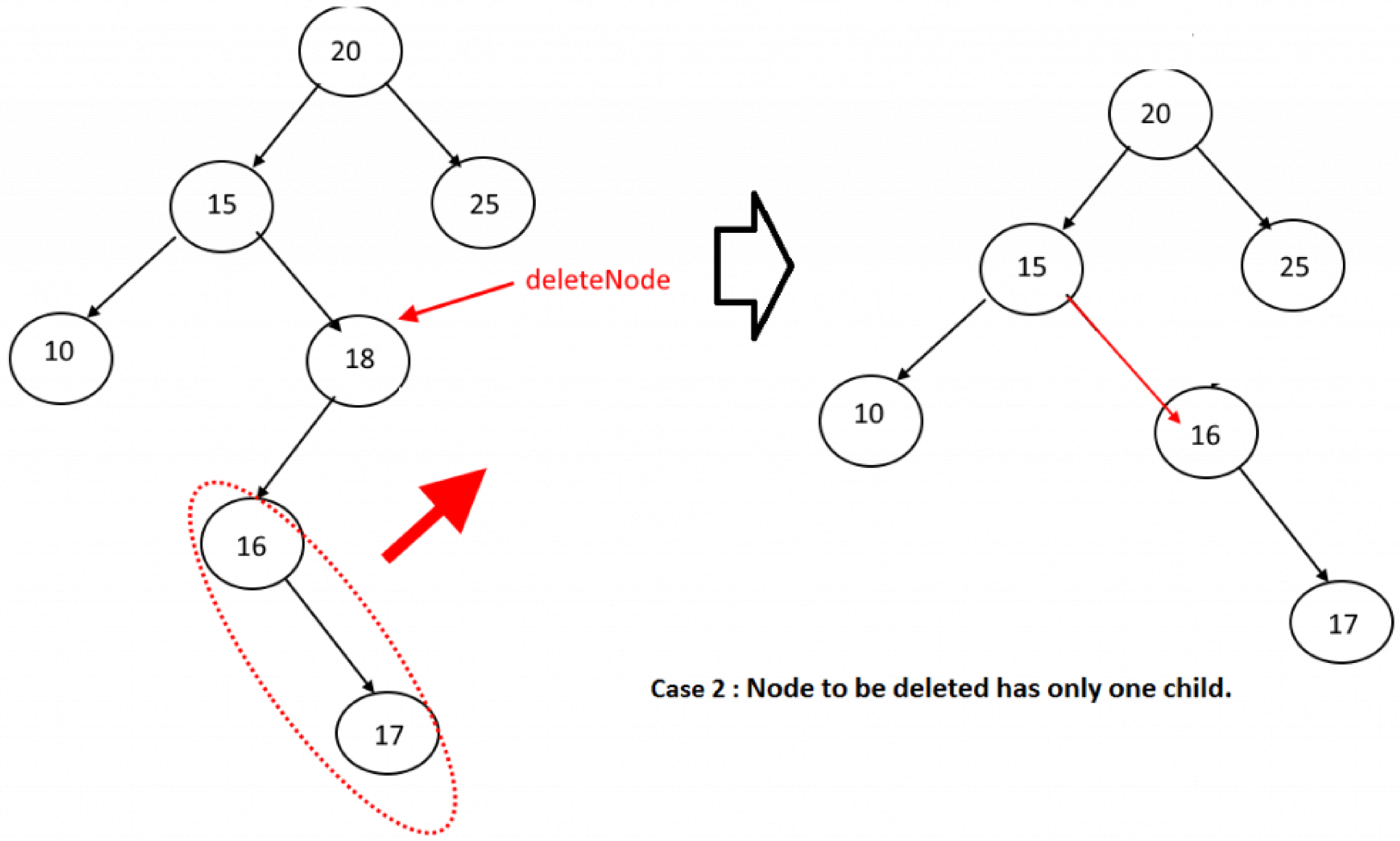
- its a very simple case, if a node to be deleted has no children then just traverse to that node, keep track of parent node and the side in which the node exist (left or right) and set ***parent.left* = *null or parent.right* = *null;***



Case 1 : Node to be deleted is a leaf node (No Children).

Node to be deleted has only one child.

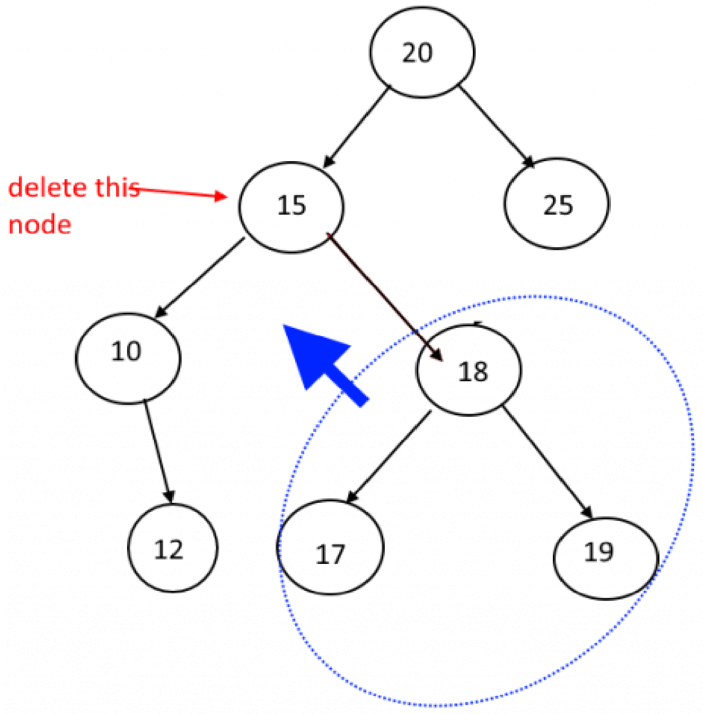
- its a slightly complex case. if a node to be deleted (deleteNode) has only one child then just traverse to that node, keep track of parent node and the side in which the node exist(left or right).
- check which side child is null (since it has only one child).
- Say node to be deleted has child on its left side . Then take the entire sub tree from the left side and add it to the parent and the side on which deleteNode exist

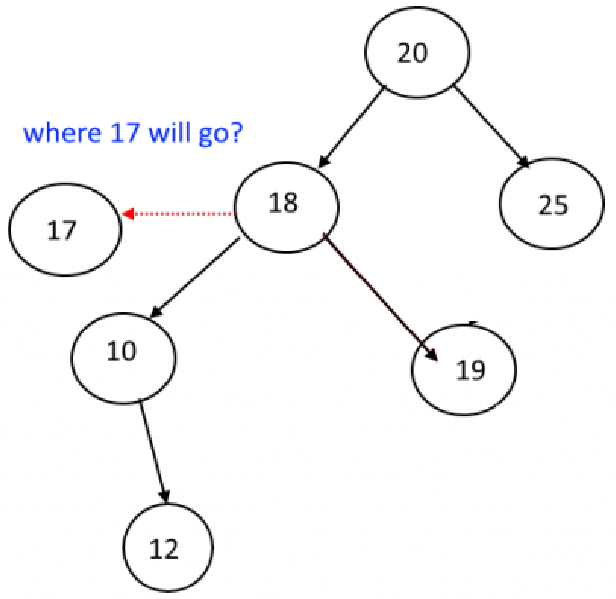


Case 2 : Node to be deleted has only one child.

Node to be deleted has two children.

- You just cannot replace the deleteNode with any of its child, Why? Lets try out a example.

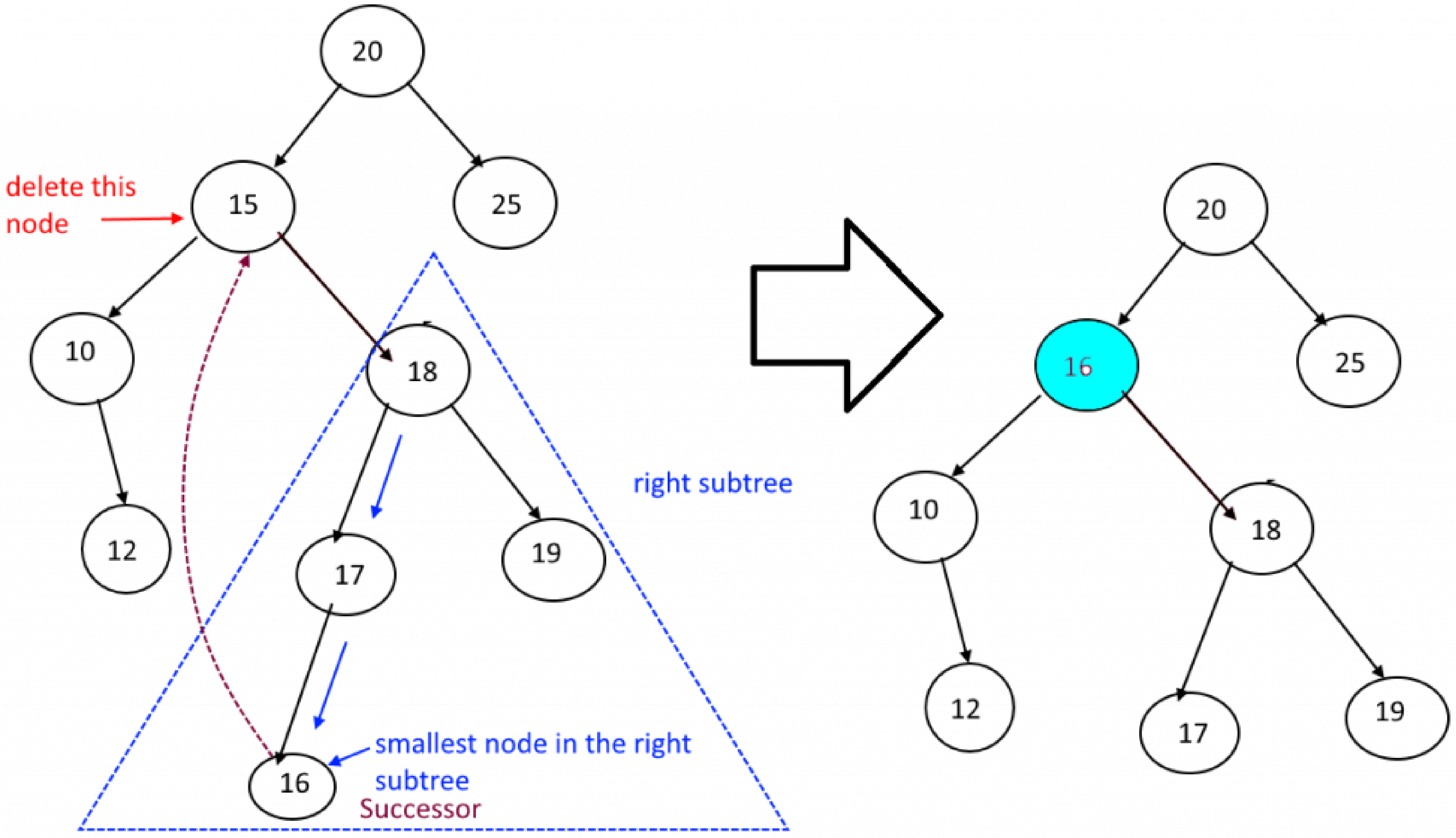




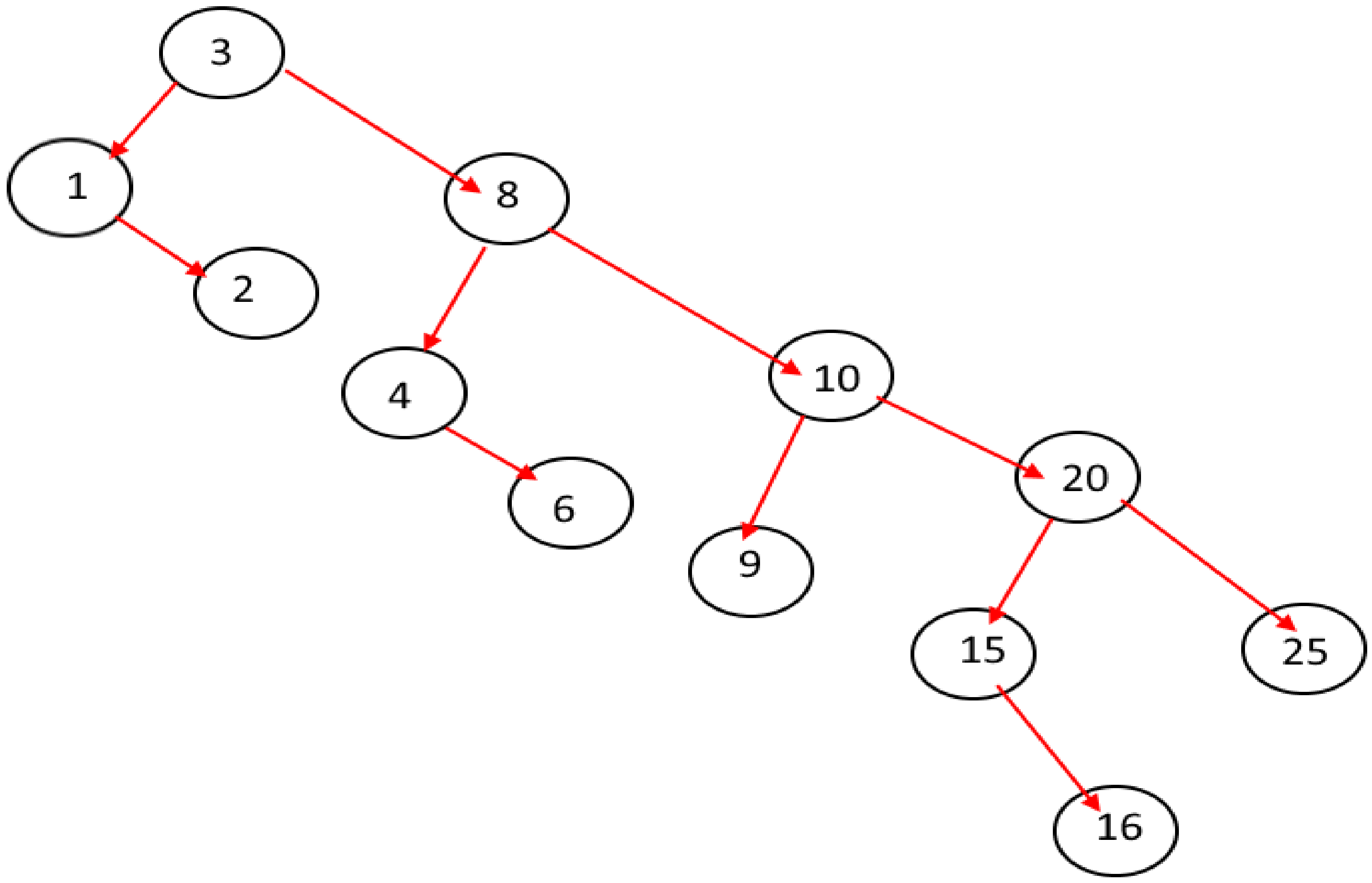
What to do now?????

Find The Successor:

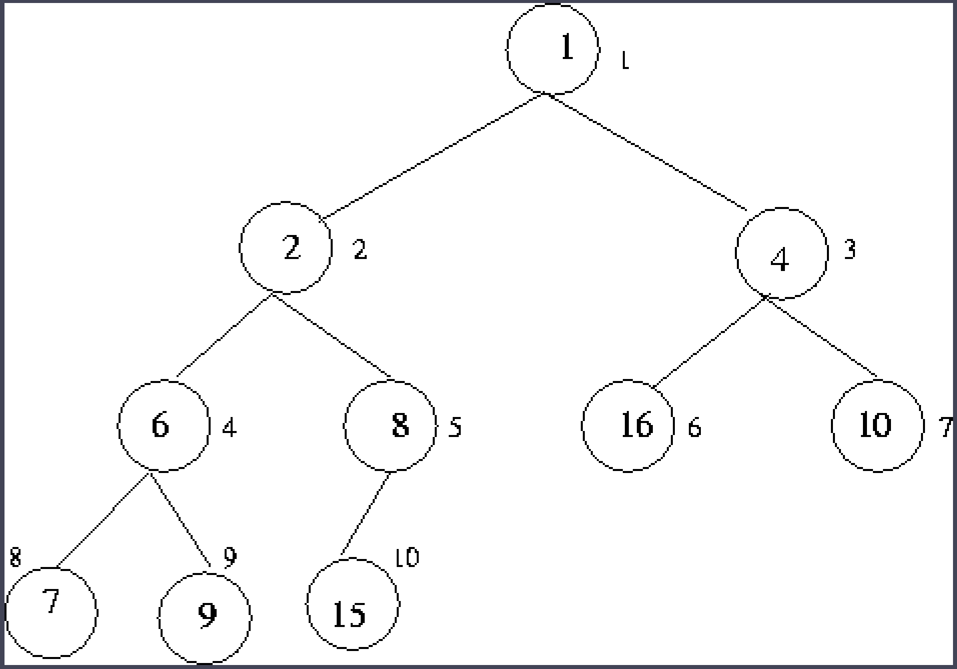
- Successor is the node which will replace the deleted node. Now the question is to how to find it and where to find it.
- *Successor is the smaller node in the right sub tree of the node to be deleted.*


20
15
25
delete this node
10
18
12
17
19
right subtree
16
smallest node in the right subtree
Successor
20
16
25
10
18
12
17
19

Display()



Heap Sort



Heap sort stores elements in binary tree form. Tree called heap. In min-heap parent always smaller than children.

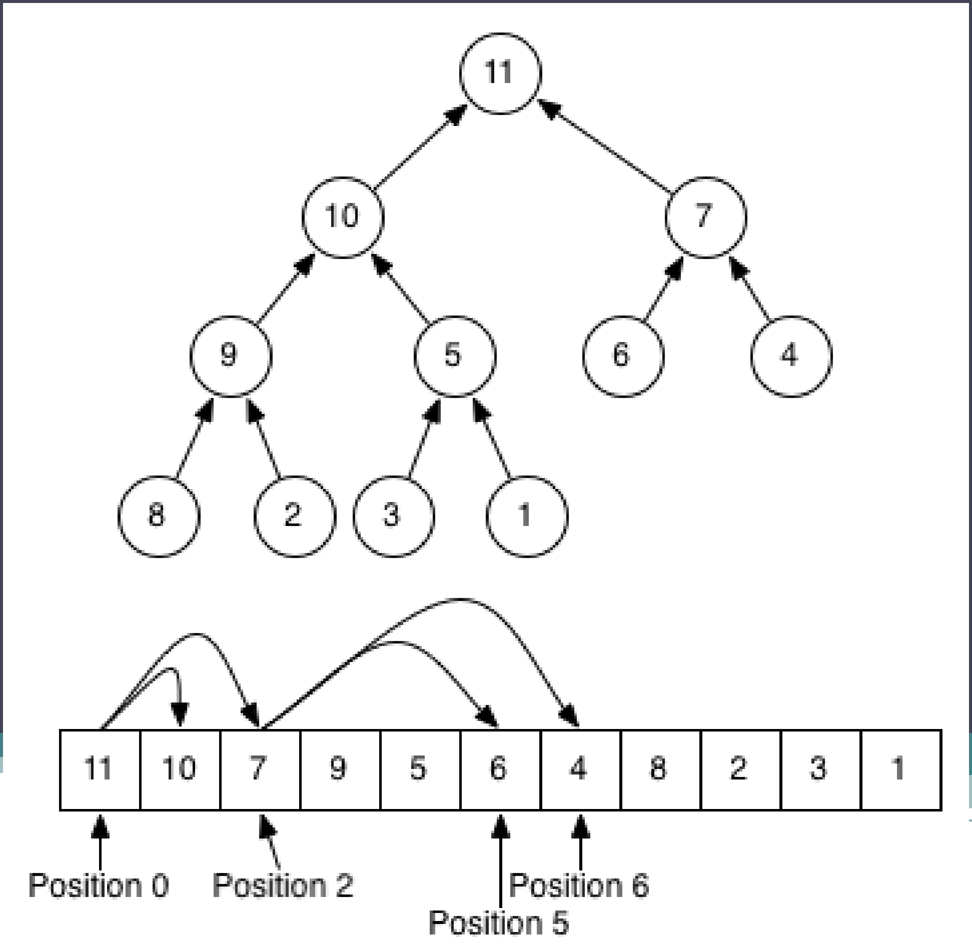
Array structure

Heap sort uses one dimensional array to store data. Each level can be found by formulae:

Parent = (CurIndex-1)/2

Left = CurIndex*2 + 1

Right = CurIndex*2 + 2



Heap sort - steps

1. Build min-Heap from given numbers
2. Take first element to sorted array
3. Swap first and last element.
4. Remove last element
5. Fix heap
6. Repeat 2-5 until all elements removed.

Heap Sort (indices starts from 0)

```
function heapSort(A)
    (first build min-heap)
    for i = (length(A)-2)/2 downto 0 do
        minHeapify(a, i)
    create array SORTED
    while length(A) > 0 do
        (swap the root(minimum value) of the heap with the last element of the heap)
        swap(A[length(A)-1], a[0]) (put the heap back in min-heap order)
        SORTED ← a[end]
        removeLast(A) (decrease size of A)
        minHeapify(a, 0)
    A ← SORTED
```

Heapify – important property of min-heap

- Check left and right children
- Select minimum that should be on current place.
- Swap their places.
- And heapify swapped child.

Heapify

```
function heapify(A , index)
    left = 2*index + 1
    right = 2*index + 2
    smallest = index
    if left < length(A) and A[left] < A[smallest] then
        smallest = left
    if right < length(A) and A[right] < A[smallest] then
        smallest = right
    if smallest ≠ index then
        swap(A[index],A[smallest])
         minHeapify(a, smallest)
```

Home Work

- **<u>Realize Heap Sort</u>**
- So that it will sort in decreasing order.

- **<u>Realize Binary Search Tree</u>**
- Realize the operations: find, insert, display.
- delete operation is not necessary, but you may do it.