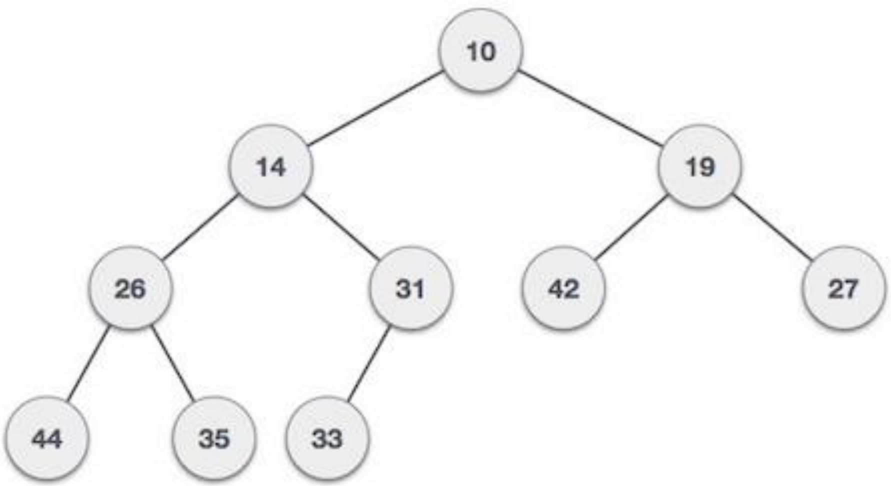
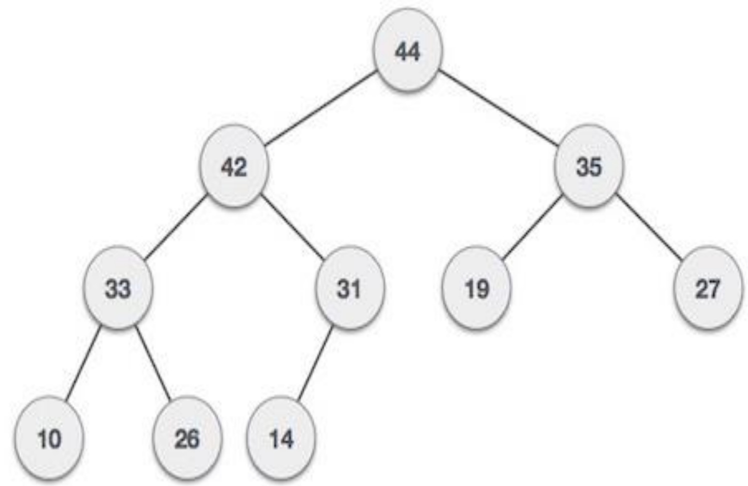
HEAP

HEAP

- Heap is a special case of balanced binary tree data structure where root-node key is compared with its children and arranged accordingly
- If α has child node β then –

  key(α) >= key(β) or key(α) <= key(β)
- Heap is a binary tree that stores priorities pair at node





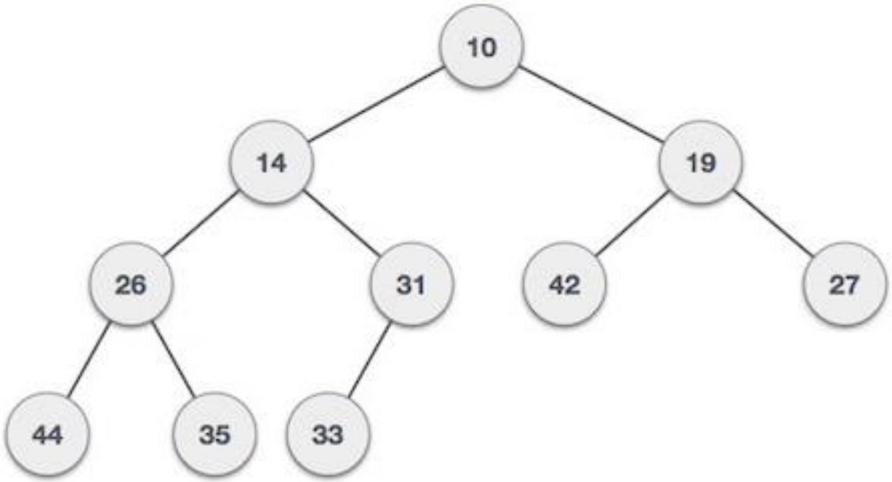
Heap Property

- **Heap has two property :**
  - Structural property :

    All levels are full except last levels . Last level is left filled.

    A complete or nearly complete binary tree.
  - Heap Property :

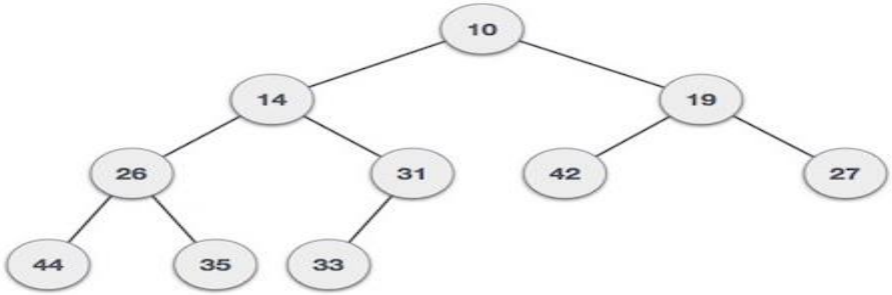
    Priority of a node is as large or small as that of its parent
- Can be represented in an array and no pointers are necessary.



Heap Property

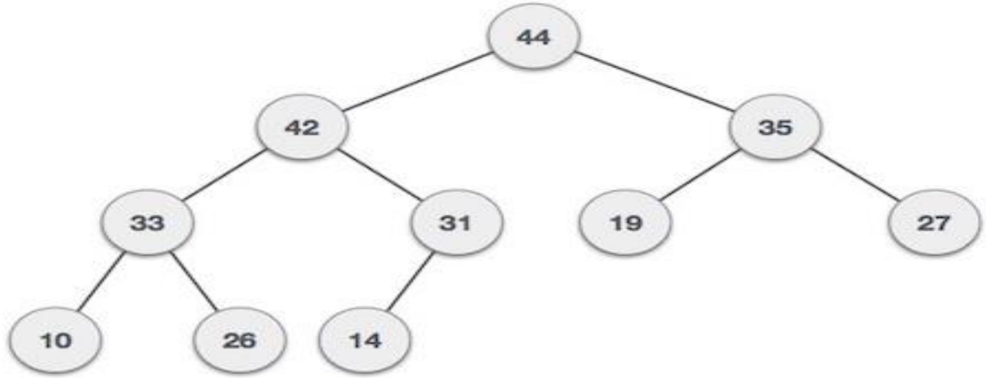
- Based on this criteria a heap can be of two types:
  - **Min Heap**

    where the value of root node is less than or equal to either of its children.



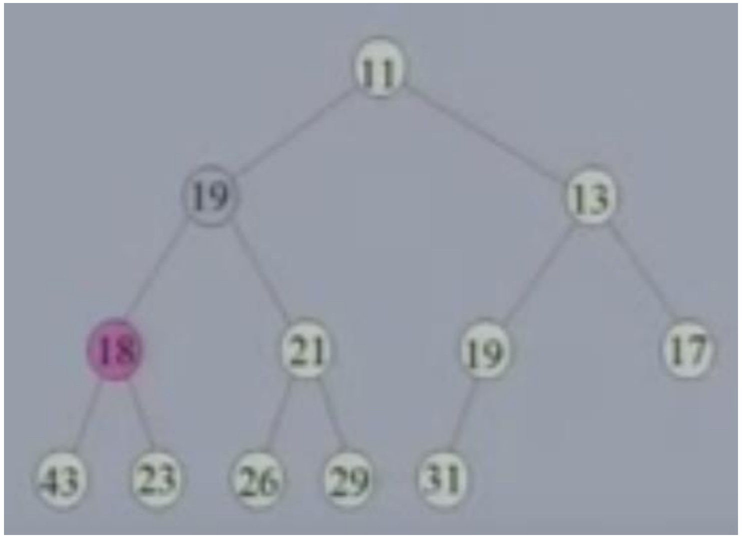
  - **Max Heap**

    where the value of root node is greater than or equal to either of its children. Max heap is often called as heap



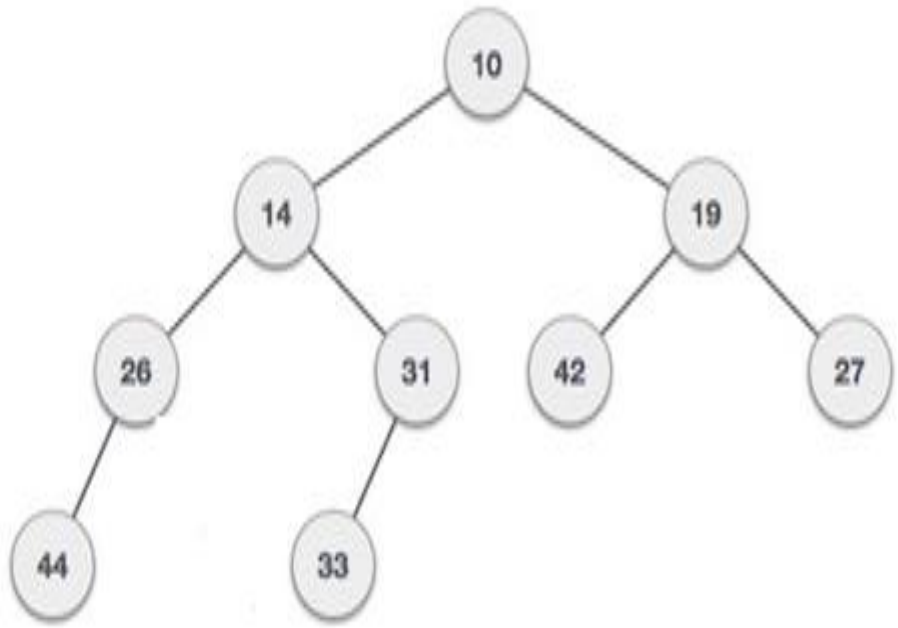
Example of Non-Heaps

- **Heap Property Violated** :



Example of Non-Heaps

- **Structural Property Violated :**



Height of Heaps

- Recall from complete binary tree :

  if height of the tree h, then number of nodes n $=2^{h+1}-1$

- Hence, $n+1= 2^{levels} = 2^{h+1}$

  $\Rightarrow \log_2 (n+1) = h+1$

  $\Rightarrow h = \log_2 (n+1) -1$

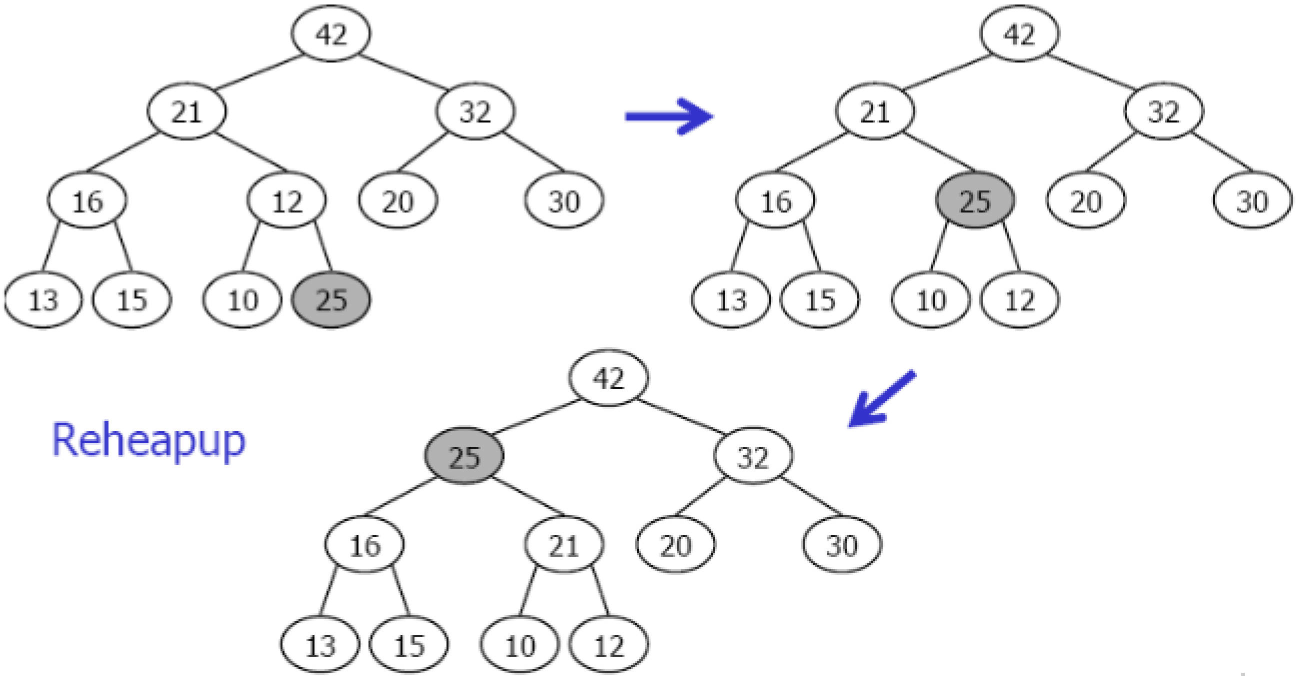
  $\Rightarrow h = \lfloor \log_2 (n+1)-1 \rfloor$

Maintenance Operations

- Two basic maintenance operations are performed on heap.
  - Insertion
  - Deletion
- To implement this basic operation we need two algorithm :
  - Reheap Up
  - Reheap down

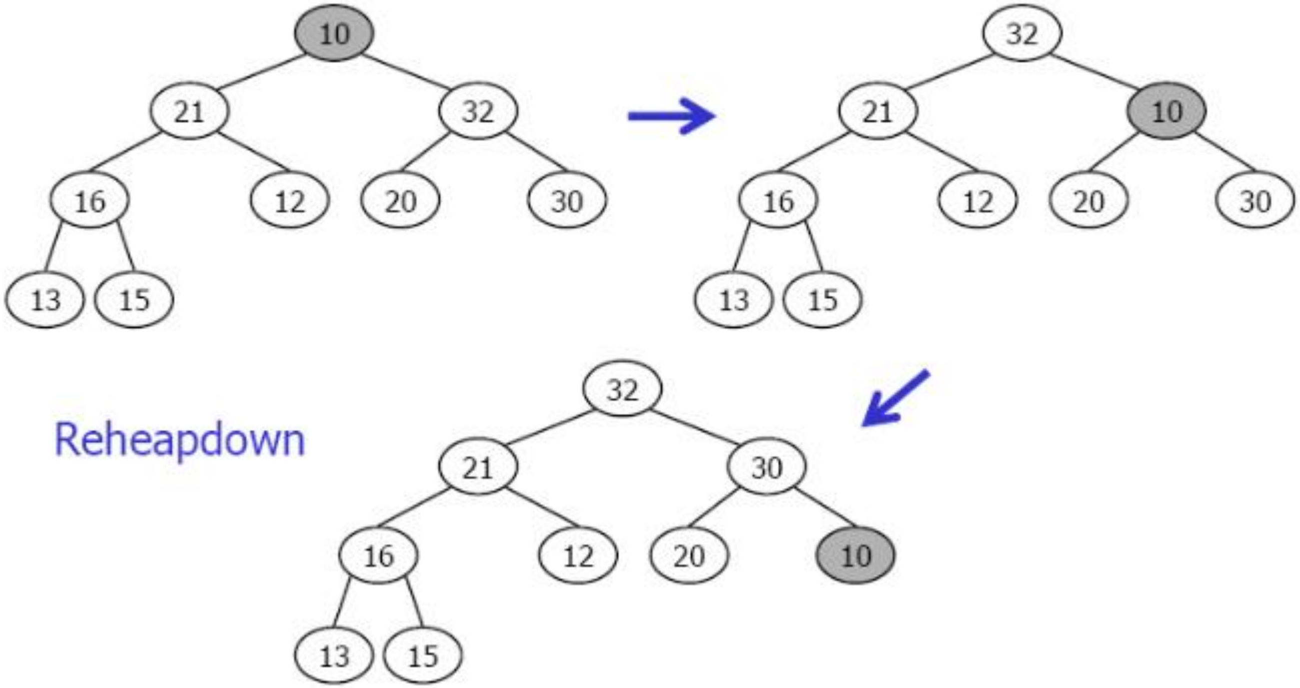
Reheap Up

- Suppose we have a nearly complete tree with N elements whose N-1 element satisfy the heap property but the last element does not
- The reheap Up operation repairs the structure so that it is a heap by floating the last element up the tree until that element is in its correct location in the tree.

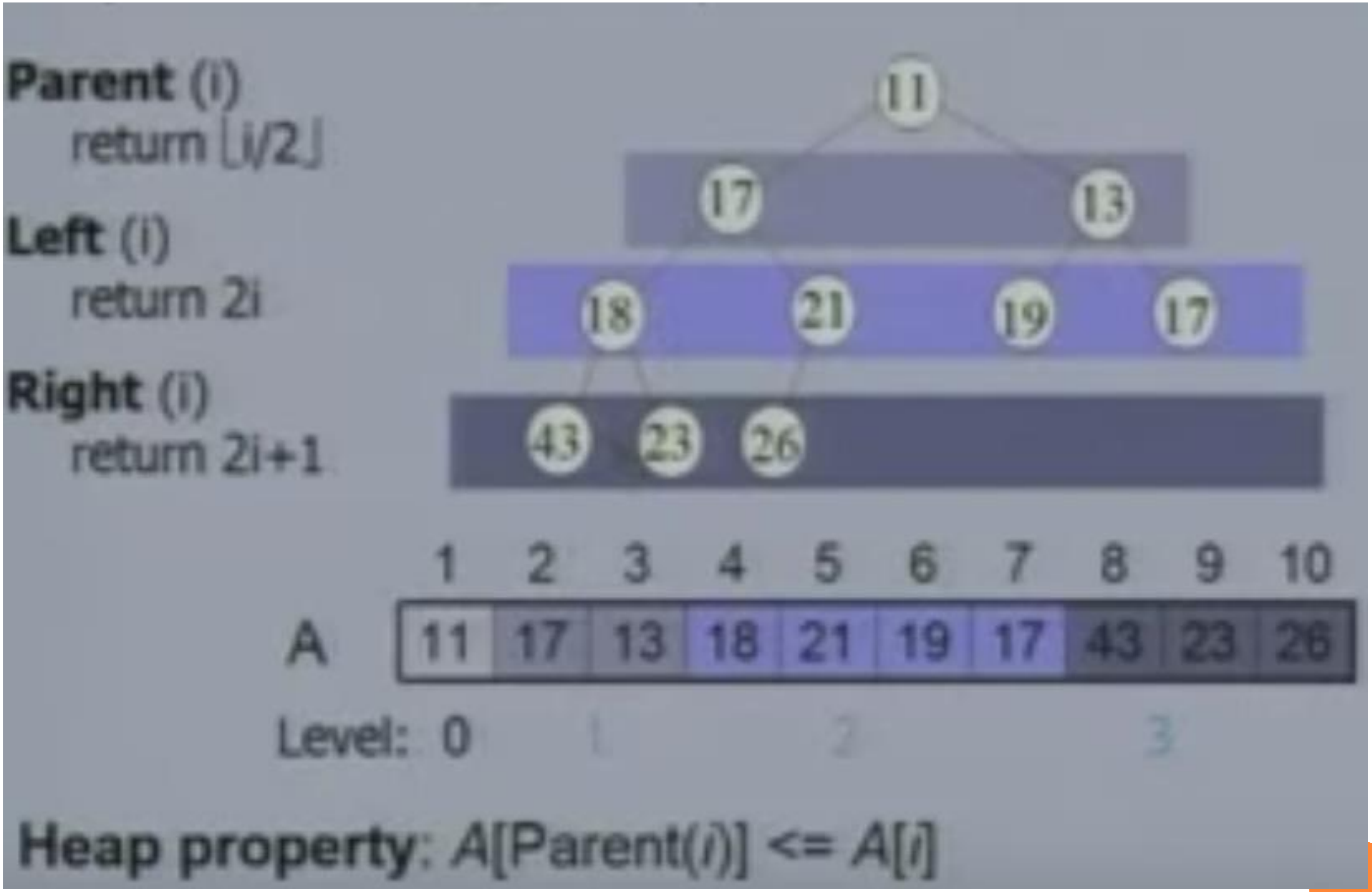


Reheap Down

- Repairs a "broken" heap by pushing the root of the subtree down until it is in its correct location.



Implementing Heaps



Implementing Heaps (cont…)

- The implicit tree link : children of node i are 2i and 2i+1
- Why is this important?
  - In a binary representation a multiplication and division by 2 is a left/right shifts
  - Adding 1 can be done by adding the lowest bit

Reheap Up Algorithm

```
Algorithm ReheapUp(position )
    If ( position  > 1)
      parent = (position ) / 2;
      if (data[ position] > data[parent])
         swap( position , parent )
         ReheapUp (parent)
         return;
  End  ReheapUp
```

Reheap Down Algorithm

```
Algorithm Reheapdown (position, lastPosition)
  leftChild =  position*2
  rightChild =  position*2 + 1
  largest = position
  if ((leftChild <=  lastPosition) AND (data [leftChild ] > data[largest])
         largest =  leftChild
  if ((rightChild <=  lastPosition) AND (data [rightChild ] > data[largest])
        largest = rightChild
  if (largest!=position)
        swap(largest, position)
        ReheapDown (largest,  lastPosition )
End  ReheapDown
```

Build heap

- Suppose, given a filled array, to build the heap we need to rearrange the data so that each node in the heap is greater than or less than its children.
- We consider two parts of array, one is heap and other part contains element to be inserted into the array
- At the beginning, first node of array is in heap and rest of the array are data to be inserted.
- Then take the next element and check if it satisfies the heap property i.e parent root has value greater or smaller than the children. If heap property violate then call **reheap up** operation to solve the problem.
- This process sometimes referred to as **heapify**

Build max heap example

67 |

---

67 12 |

---

89 12 67 |

---

89 26 67 12 |

---

89 36 67 12 26 |

---

89 36 67 12 26 45 |

---

89 36 67 12 26 45 22 |

---

89 79 67 36 26 45 22 12 |

---

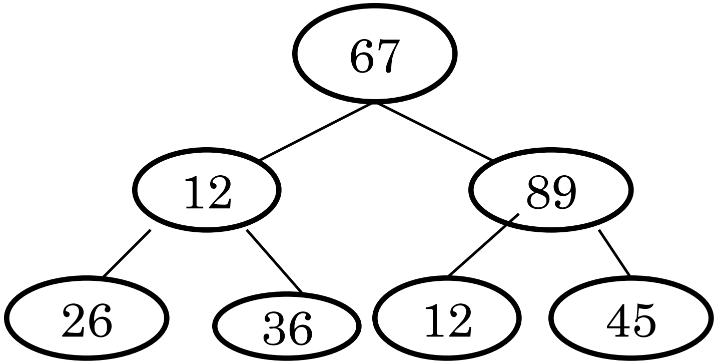
89 79 67 53 26 45 22 12 36 |

---

89 79 67 53 26 45 22 12 36 9 |

---

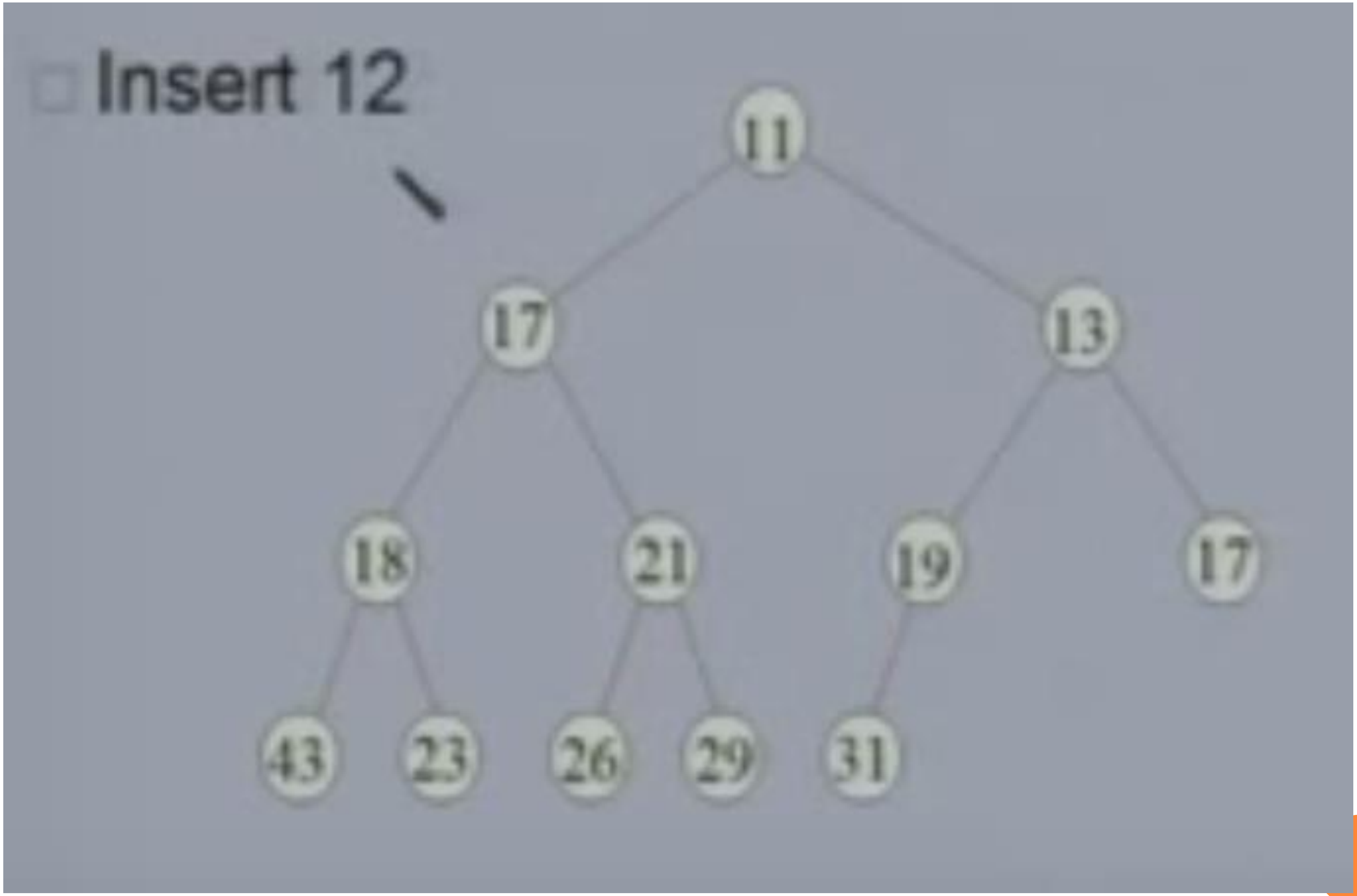
89 79 67 53 61 45 22 12 36 9 26



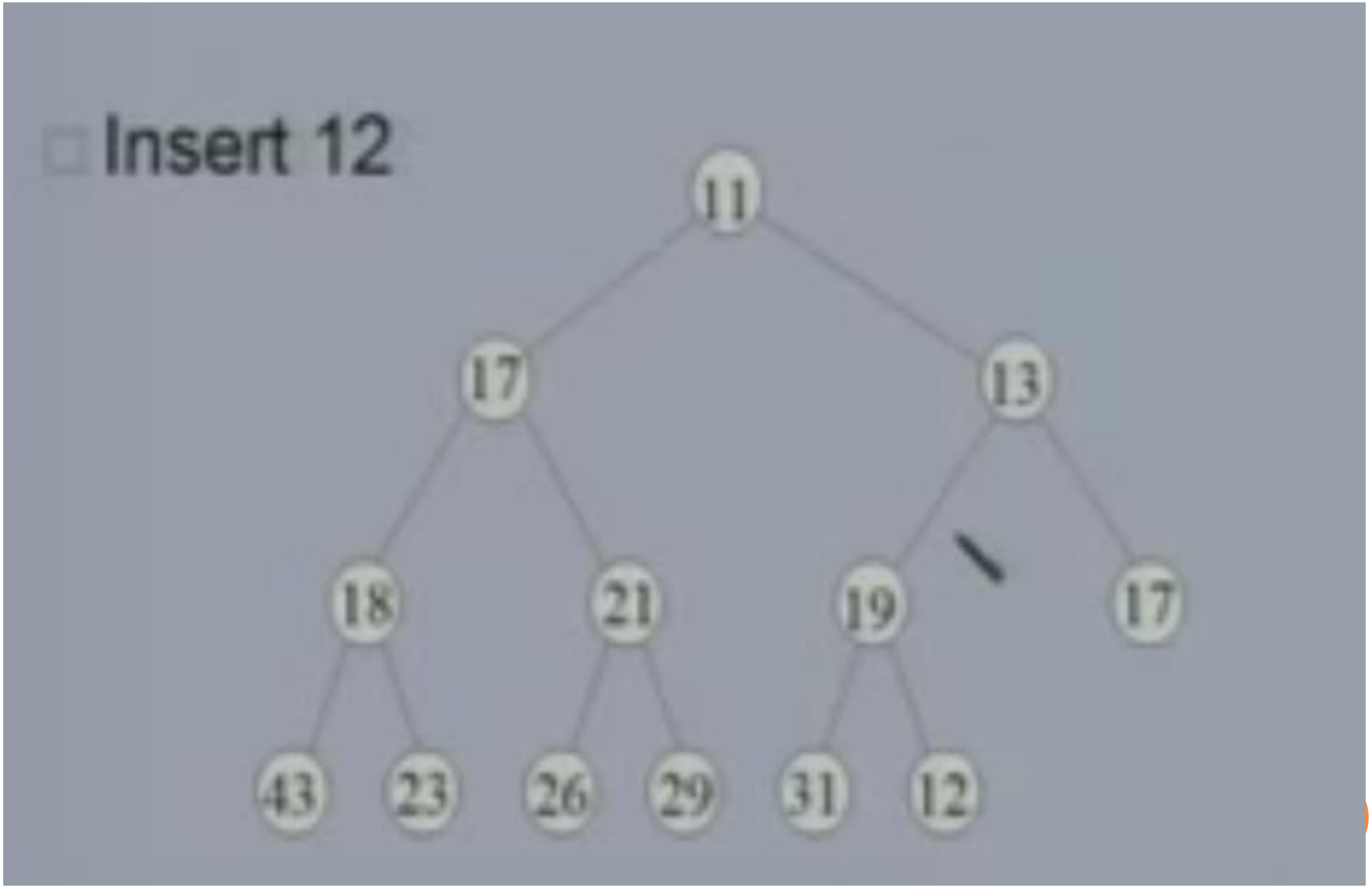
BUILD HEAP

```
Algorithm BuildHeap (listOfData)
    count = 1
     loop(count <= listOfData.Size())
        data[count] =  listOfData[count]
        ReheapUp( count );
        count = count + 1
     end loop;
End BuildHeap
```

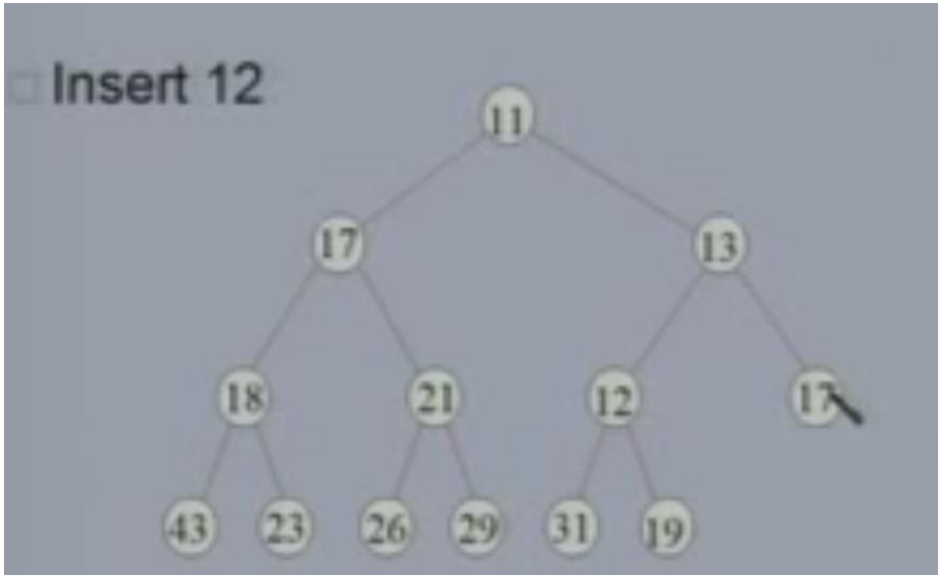
Insertion into Heaps



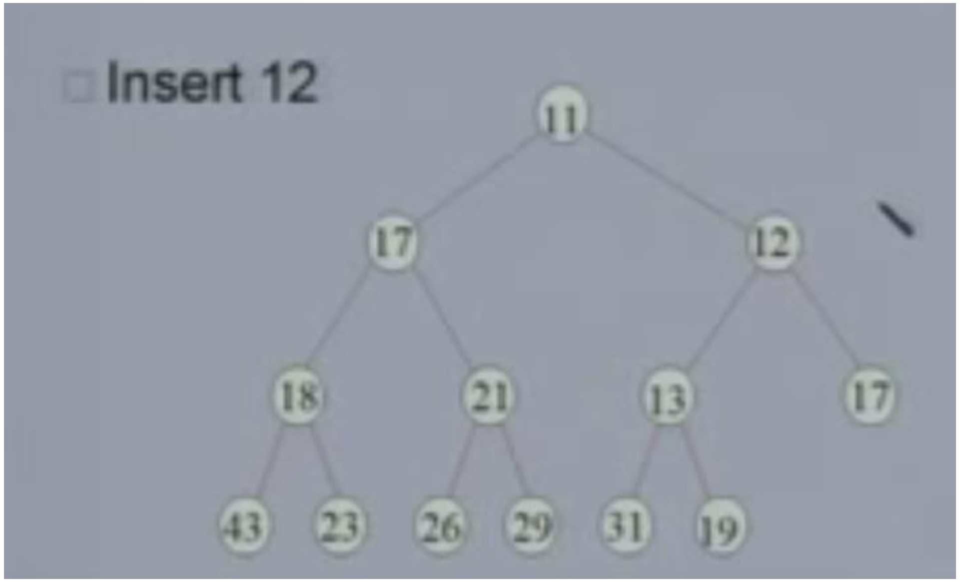
Insertion into Heaps



Insertion into Heaps



Insertion into Heaps



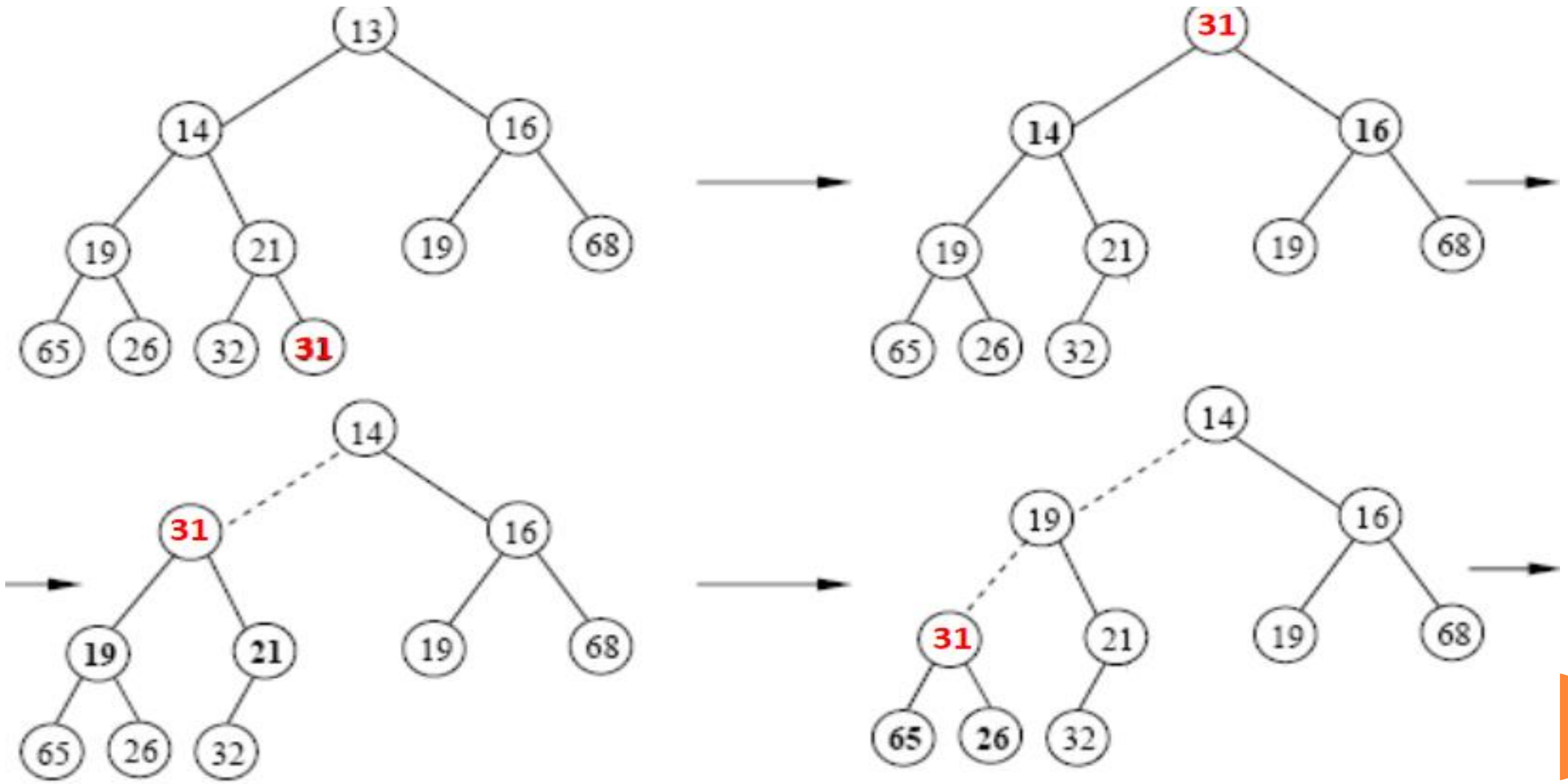
Insertion into Heaps

Recursive

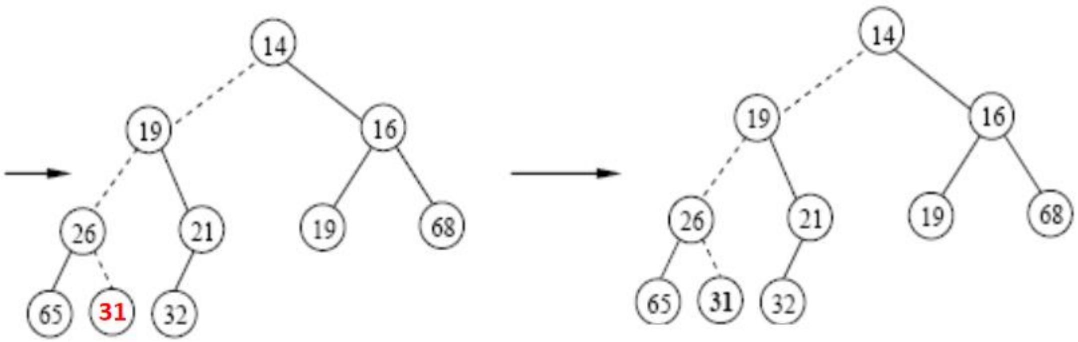
InsertHeap(DataIn)
    if(heap is full)
    return overflow
    else
    data[count] = DataIn ;
     ReheapUp(count)
     count = count+ 1
     end if;
   End InsertHeap

DELETE MIN HEAP

- Delete the top element
- The element in the last position is put to the position of the root, and Reheap Down is called for that position



Delete min Heap



Delete Heap

```
if (heap is empty)
  return underflow
else
  DeleteData = Data[1]
  Data[1] = Data[count]
  count =  count - 1
  ReheapDown(1, count )
end if;
```