Heap Trees

Heap Trees

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

  key(α) ≥ key(β)

  As the value of parent is greater than that of child, this property generates **Max Heap**.
- If α has child node β then –

  key(α) ≤ key(β)

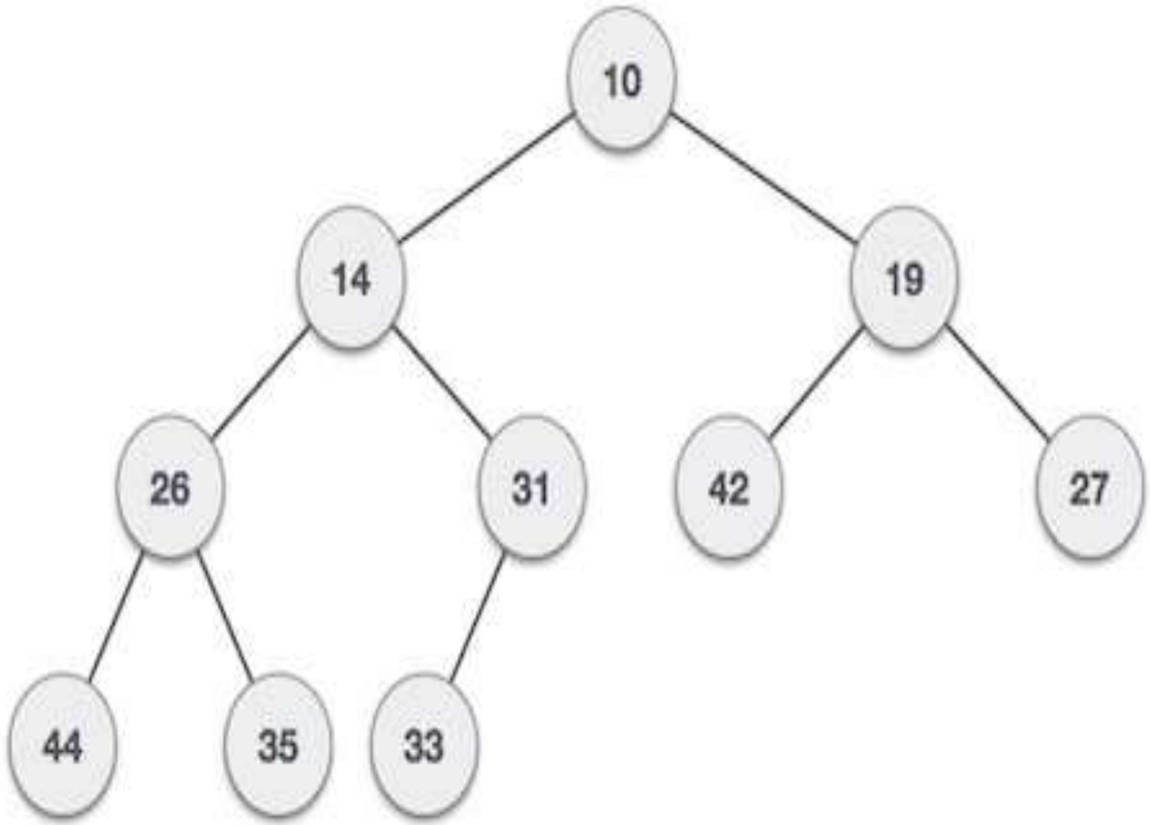
  As the value of parent is greater than that of child, this property generates **Min Heap**.

Heap Trees

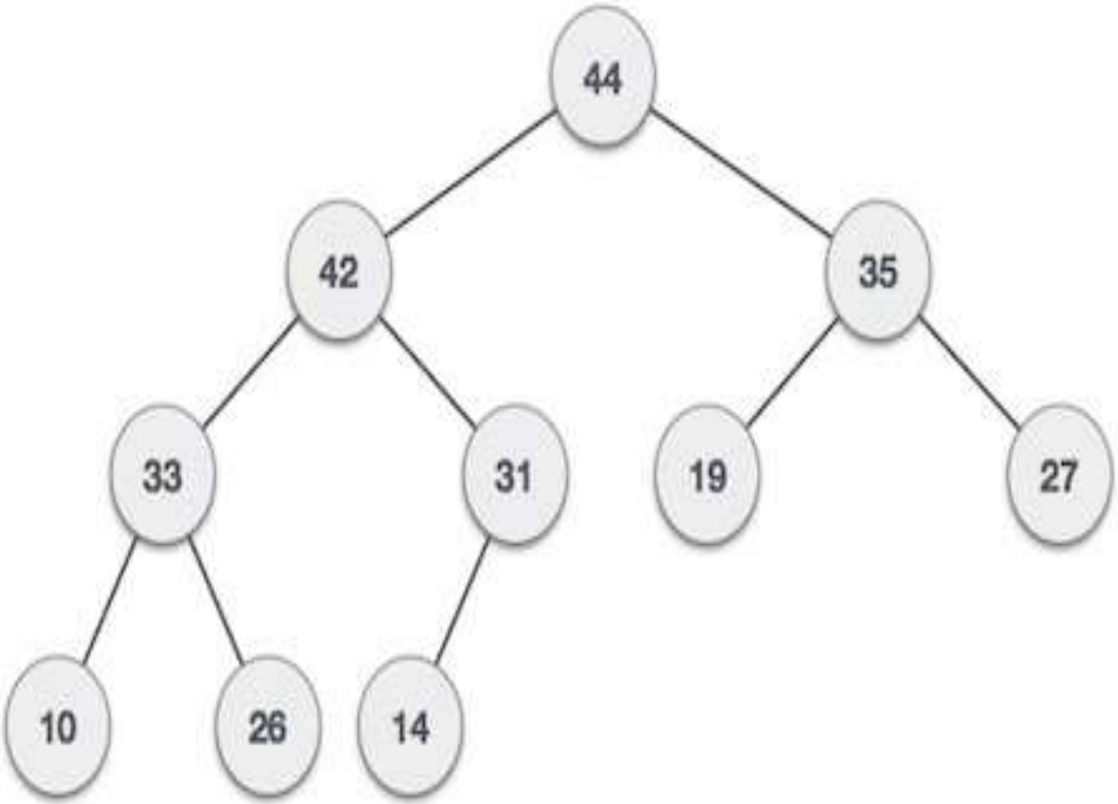
- It is a binary tree with the following properties:
  - *Property 1:* it is a complete binary tree
  - *Property 2:* the value stored at a node is greater or equal to the values stored at the children

    (**heap property**)

Minimum Heap

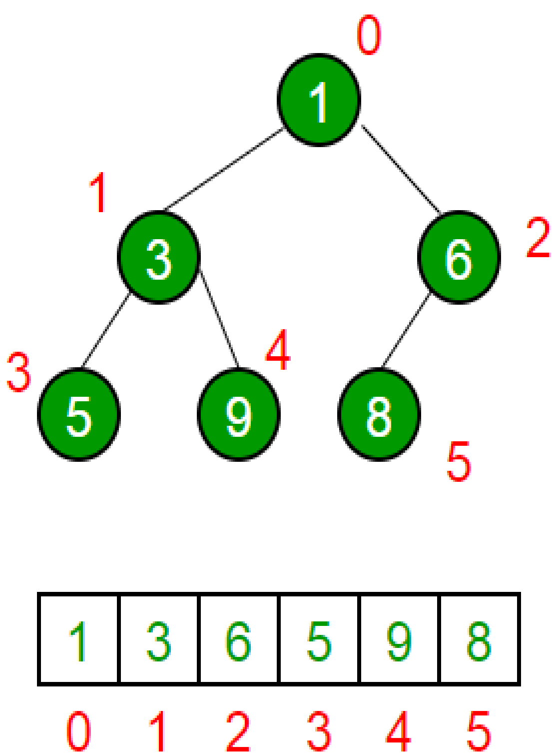


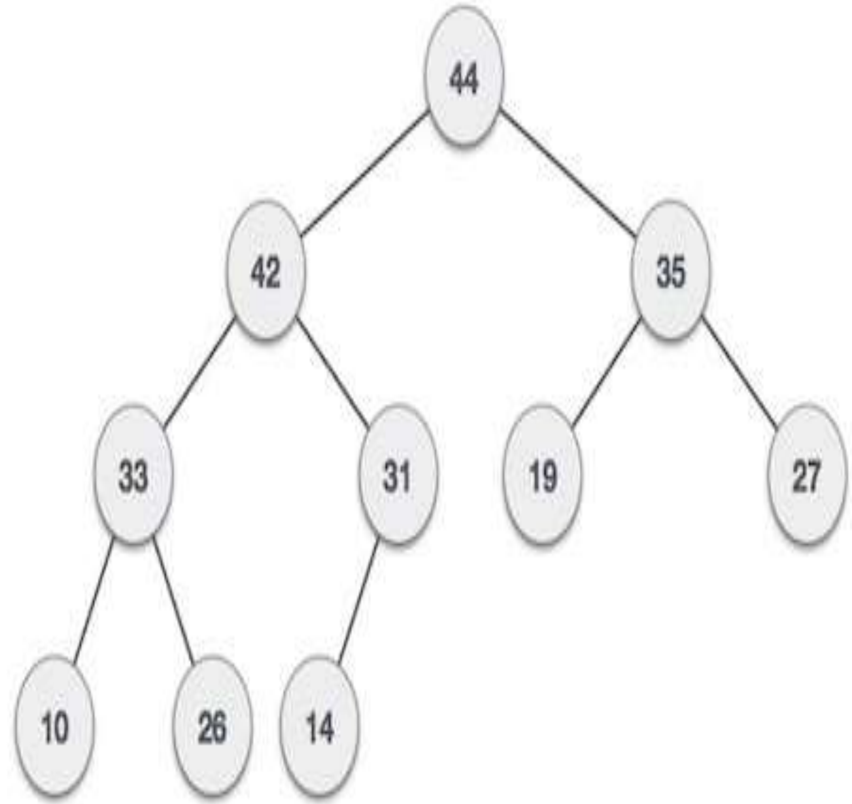
Maximum Heap



Tree Traversal

The traversal method use to achieve Array representation is **Level Order**



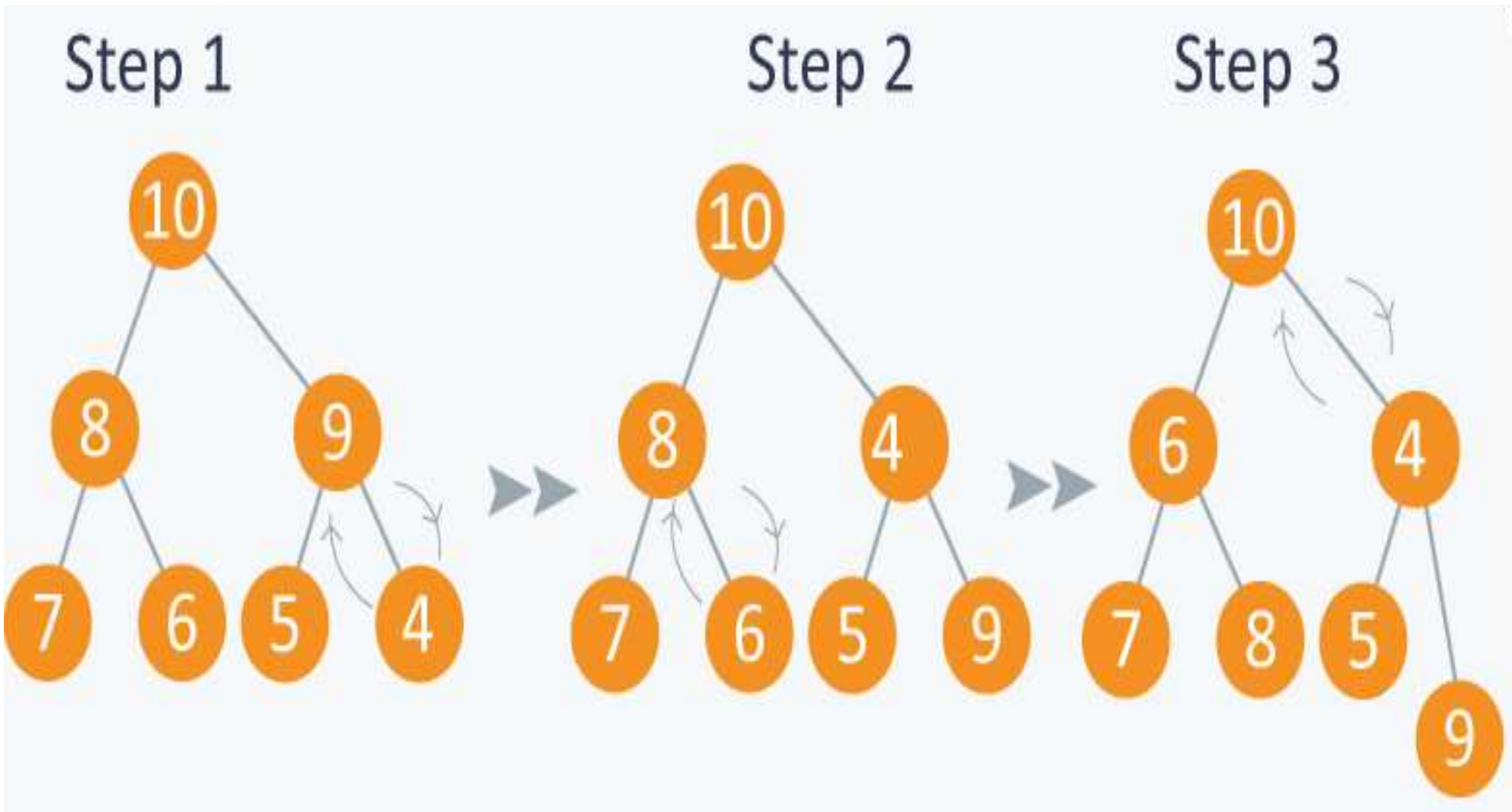


Heapify Procedure

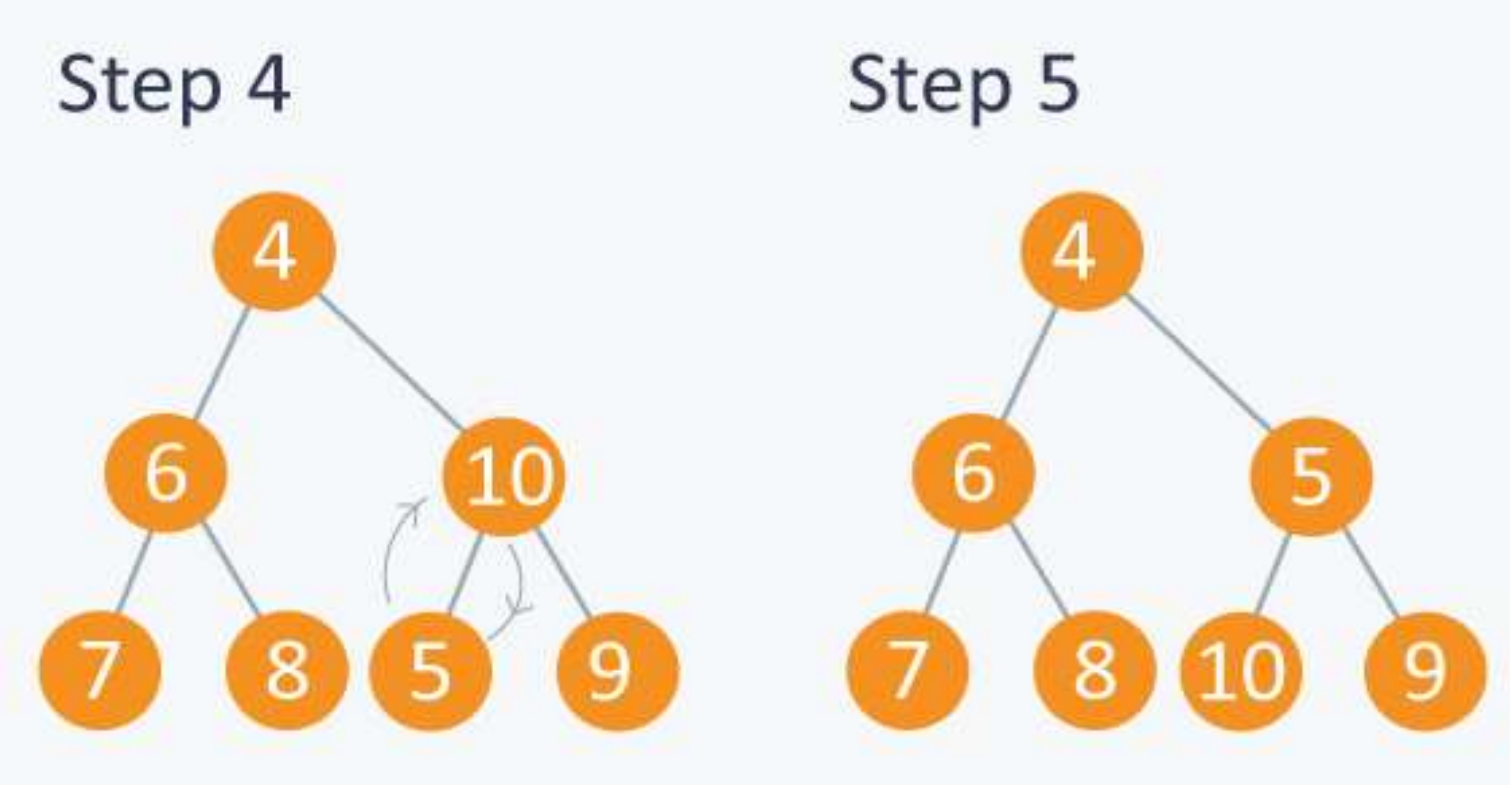
```
void build_minheap (int Arr[ ])
  {
      for( int i = N/2 ; i >= 1 ; i--)
      min_heapify (Arr, i);
  }
```

```
void min_heapify (int Arr[ ] , int i, int N)
  {
  int left  = 2*i;
  int right = 2*i+1;
  int smallest;
  if(left <= N and Arr[left] < Arr[ i ] )
        smallest = left;
  else
       smallest = i;
  if(right <= N and Arr[right] < Arr[smallest] )
       smallest = right;
  if(smallest != i)
  {
       swap (Arr[ i ], Arr[ smallest ]);
       min_heapify (Arr, smallest,N);
  }
  }
```

Heapify Procedure – Minheap

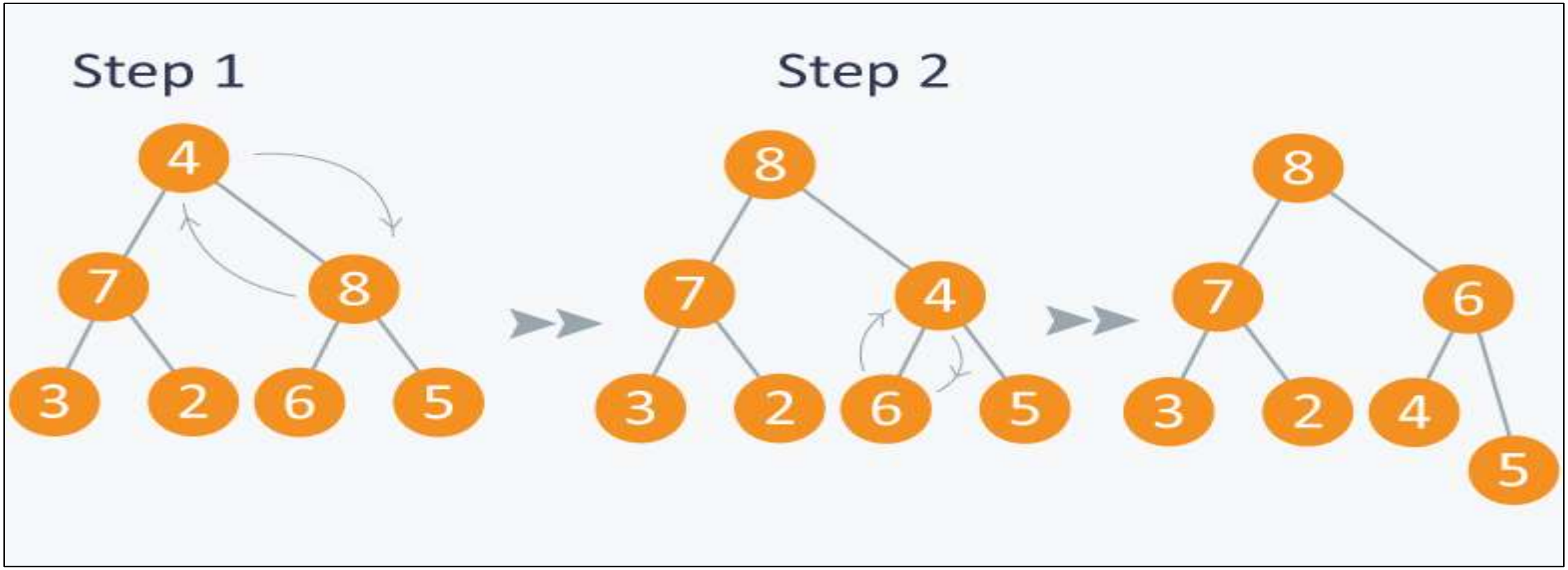


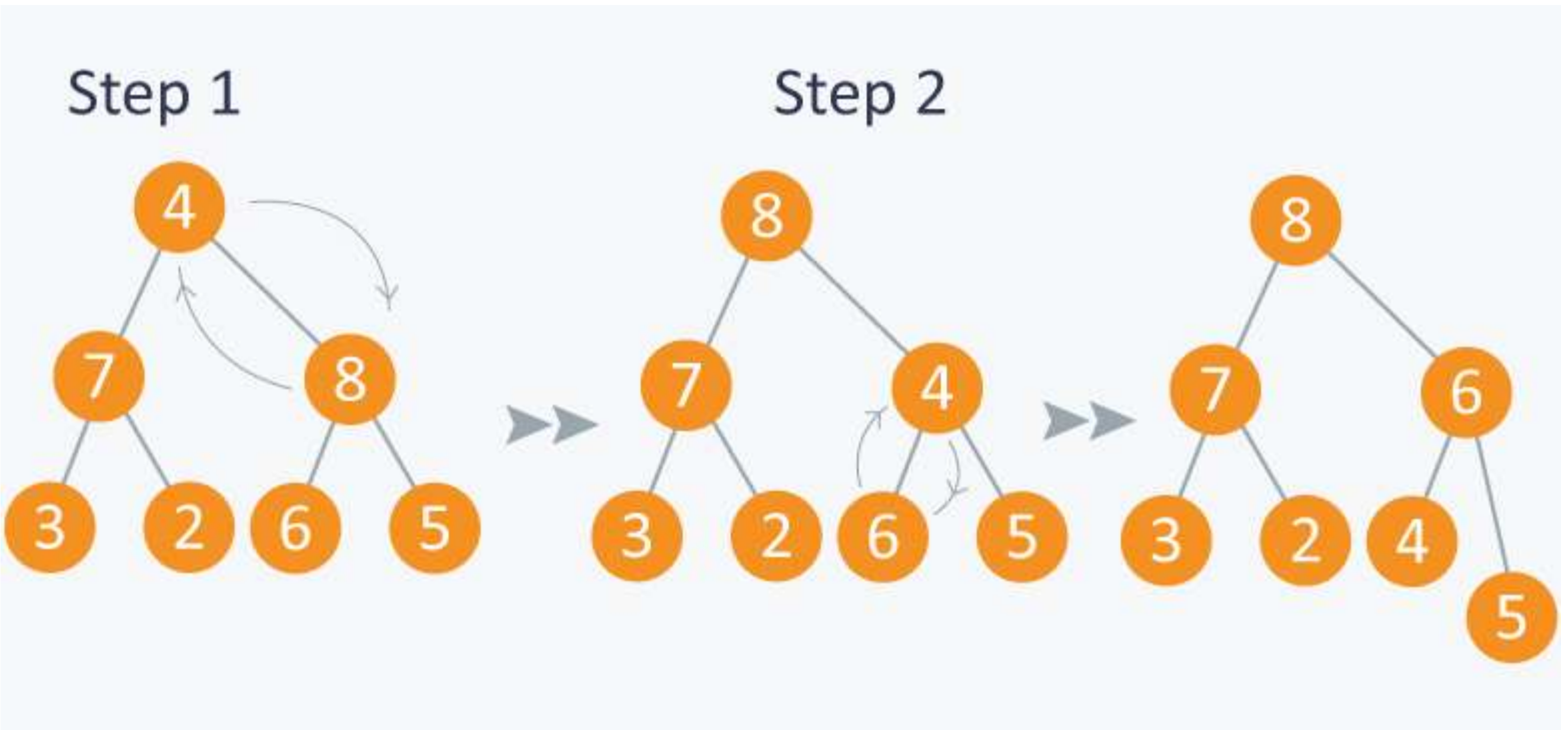
Heapify Procedure – Minheap



Heapify Procedure – Maxheap

```
void max_heapify (int Arr[ ], int i, int N)
      {
            int left = 2*i                            //left child
            int right = 2*i +1                   //right child
            if(left<= N and Arr[left] > Arr[i] )
                     largest = left;
            else
                   largest = i;
            if(right <= N and Arr[right] > Arr[largest] )
                  largest = right;
            if(largest != i )
            {
                  swap (Arr[i] , Arr[largest]);
                  max_heapify (Arr, largest,N);
            }
       }
```




Step 1
4
7
8
3
2
6
5
Step 2
8
7
4
3
2
6
5
8
7
6
3
2
4
5

Max Heap Construction Algorithm

Step 1 – Create a new node at the end of heap.

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

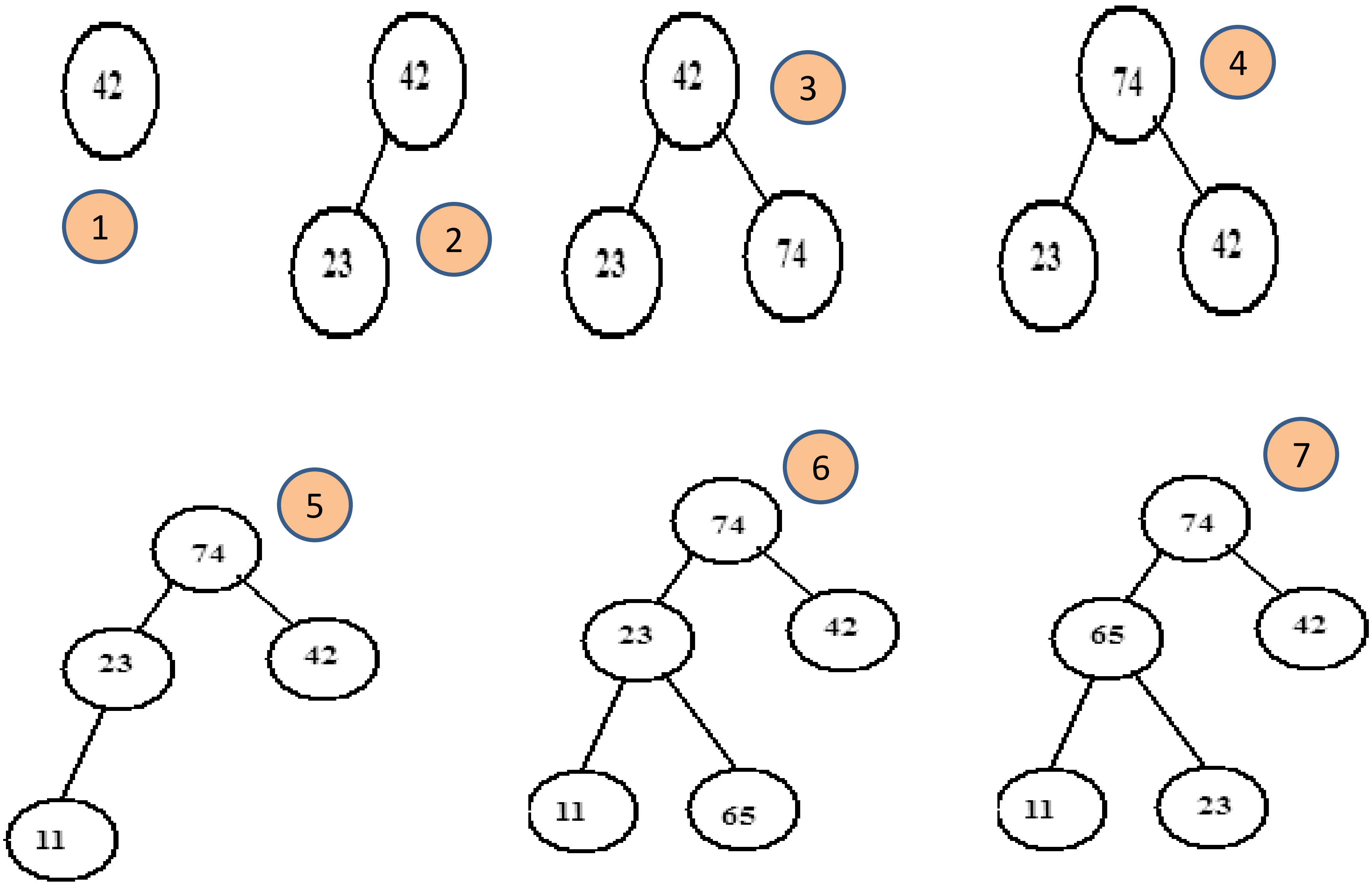
Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

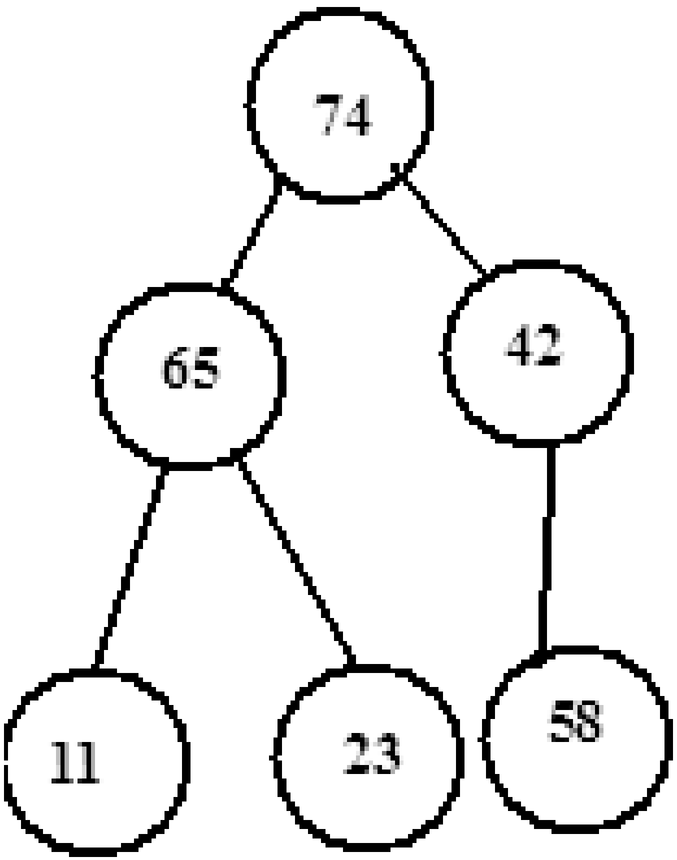
Insert Procedure

```
INSERT( A[ ], T, k )
 N = T
N = N + 1
A[N] = k
 While N! = 1
     If A[N] > A[N/2]
           N = N/2
     Else
           Break
     End if
End while
T = T + 1
End INSERT
```

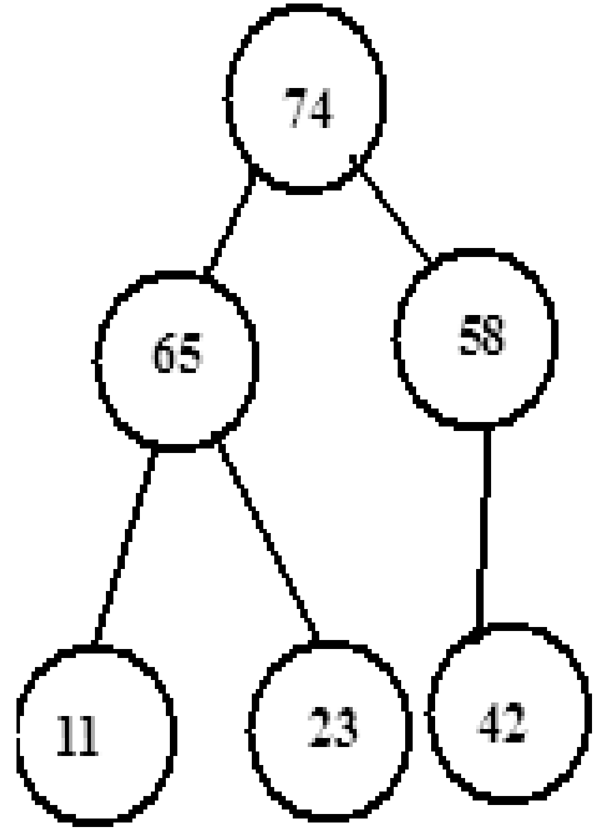
Insert operation in a Heap



Insert operation in a Heap



8



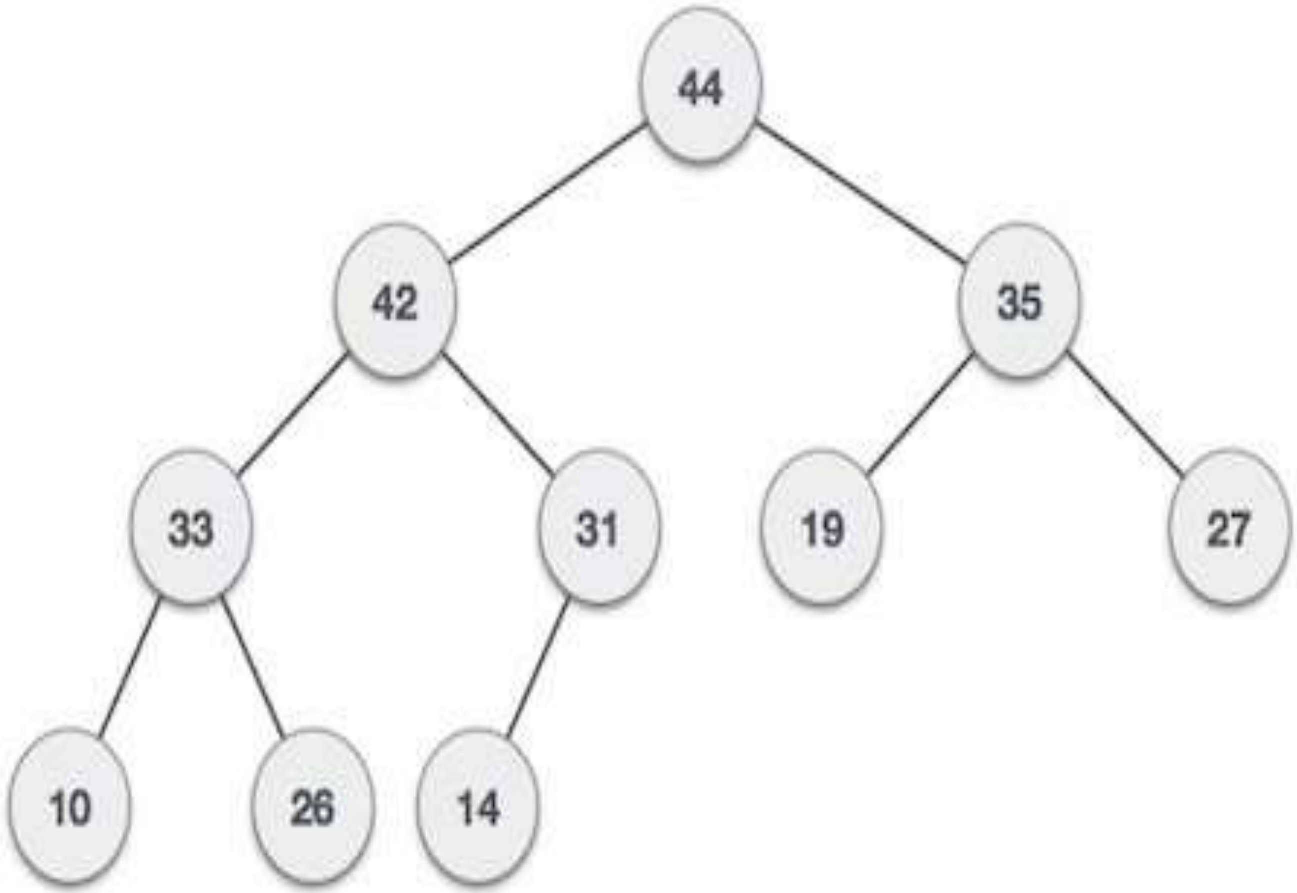
9

Insert operation in a Heap

- 35, 33, 42, 10, 14, 19, 27, 44, 26, 31

Example

- 35, 33, 42, 10, 14, 19, 27, 44, 26, 31



Max Heap Deletion Algorithm

Step 1 – Remove root node.

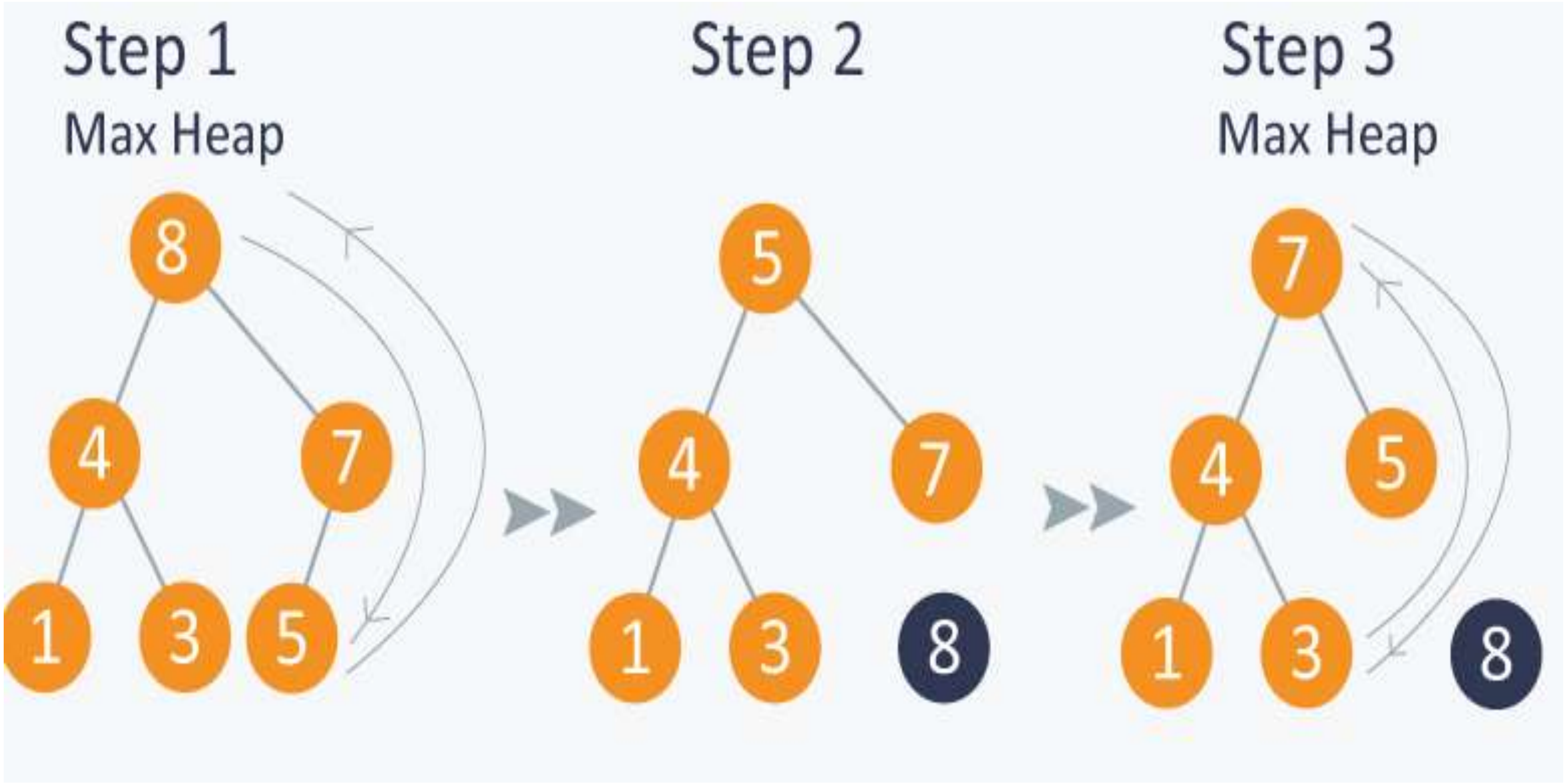
Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

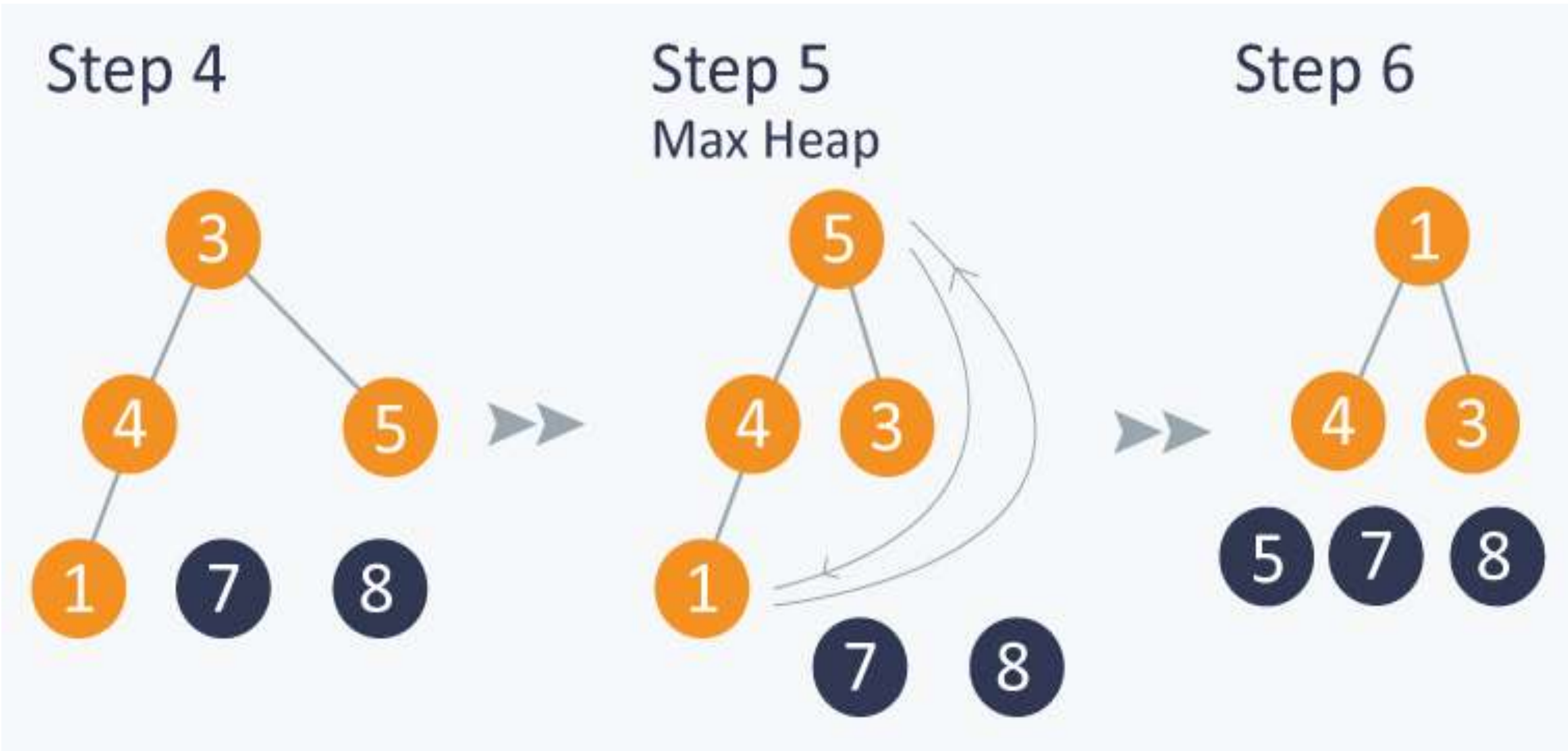
Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

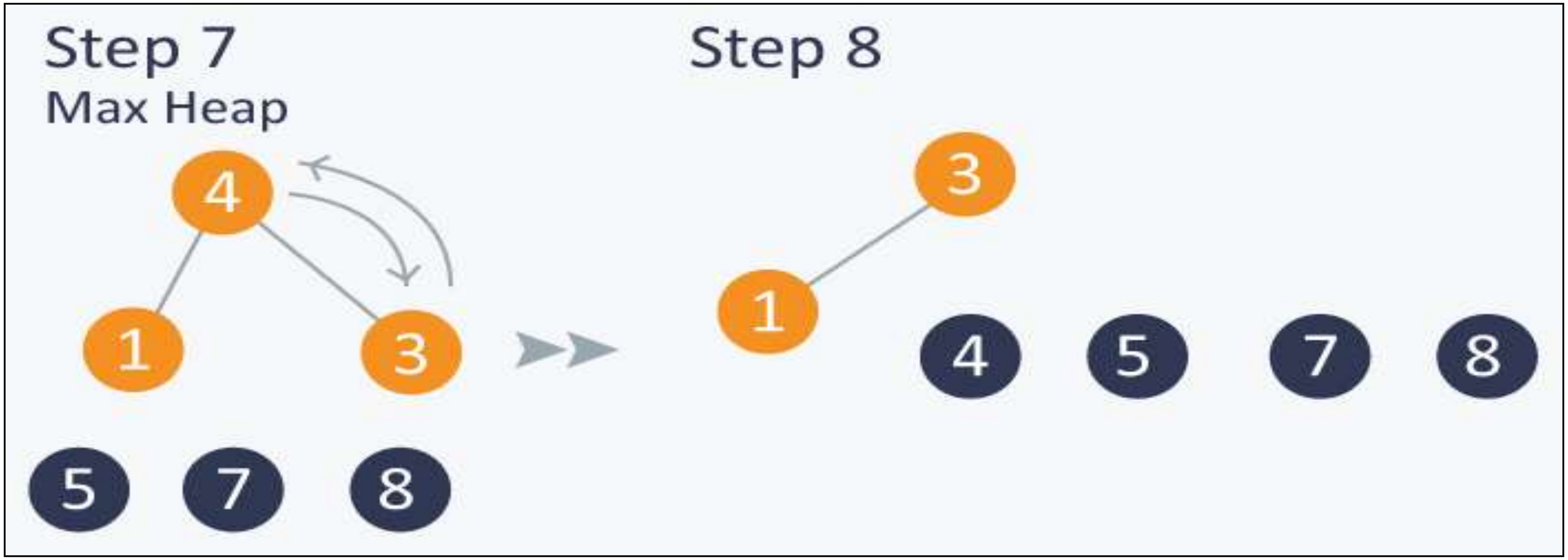
Max Heap Deletion



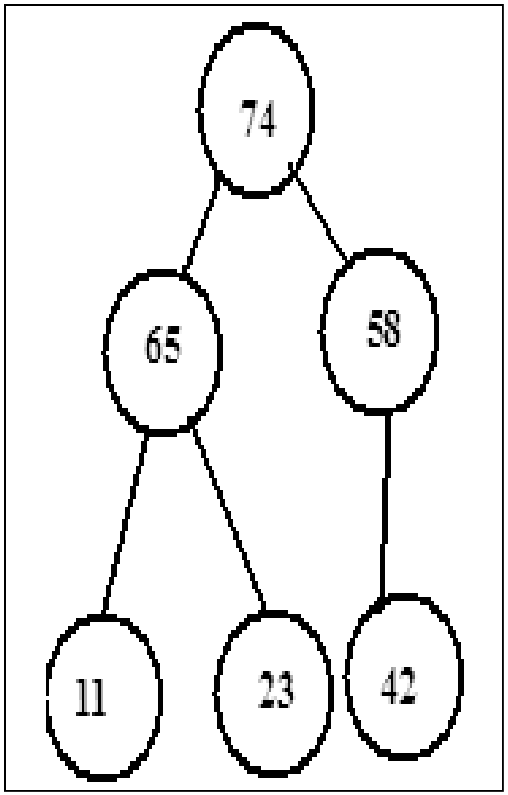
Max Heap Deletion



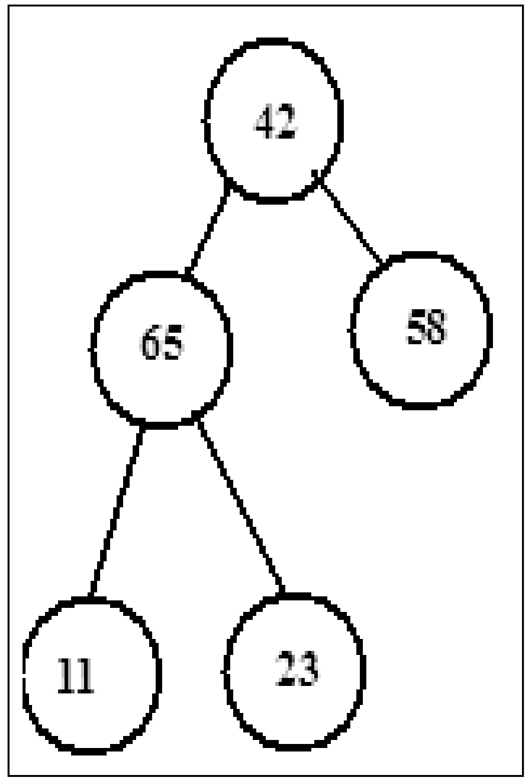
Max Heap Deletion



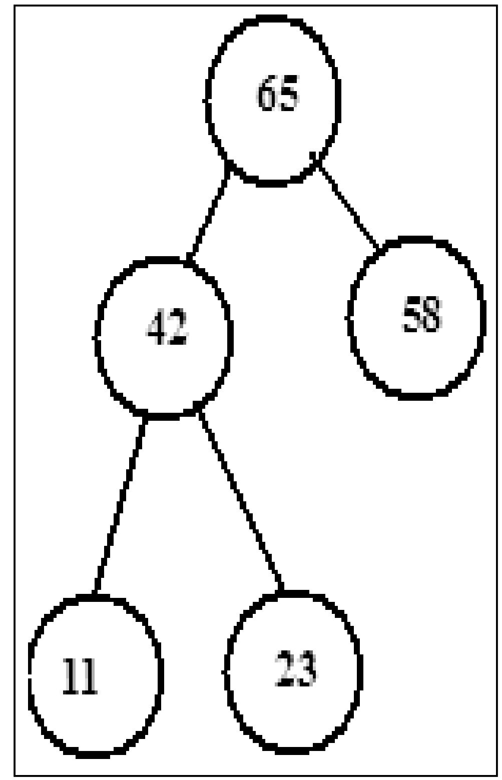
Max Heap Deletion



1

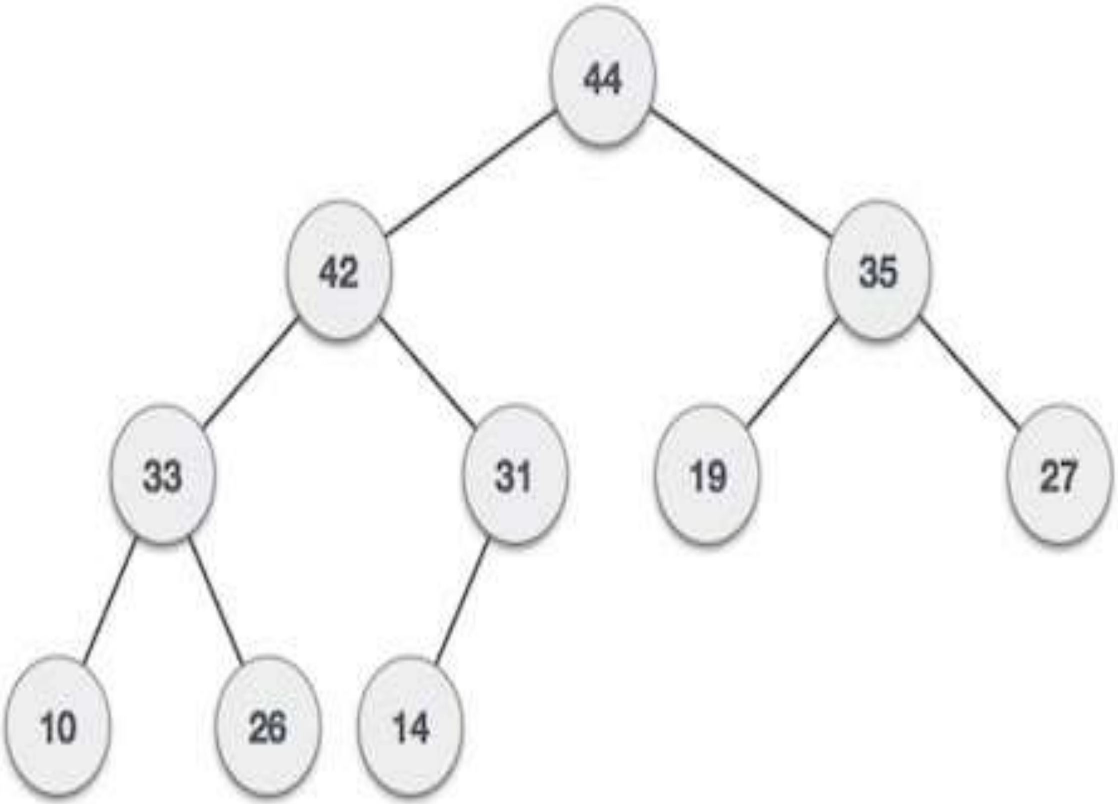


2



3

Delete 44



Applications

- Sorting – Heap sort
- Extract Maximum, Minimum – O(logn)
- Priority Queues