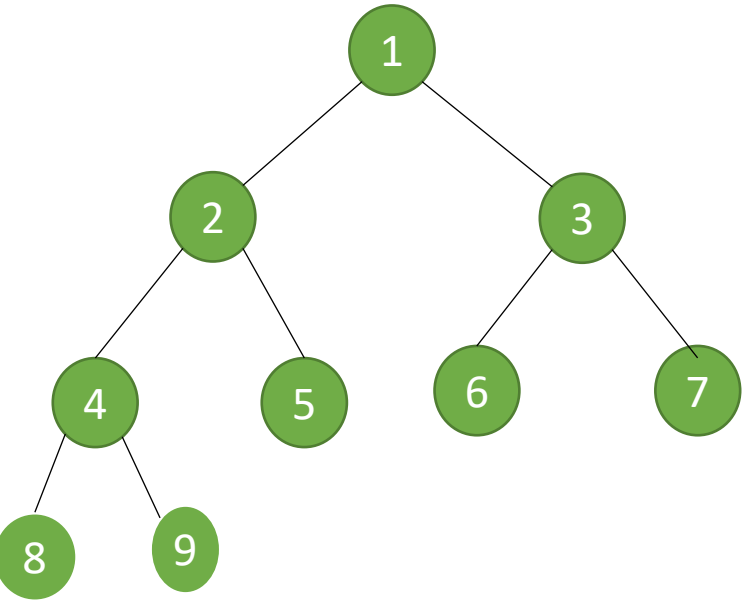
Heap
and
Heap sort

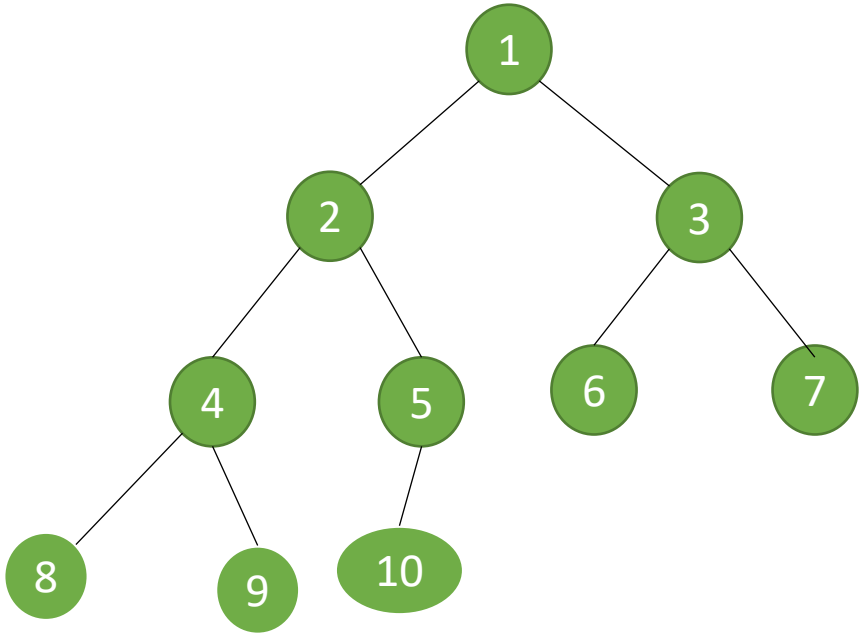
Some Special Types of Binary Tree :

1.



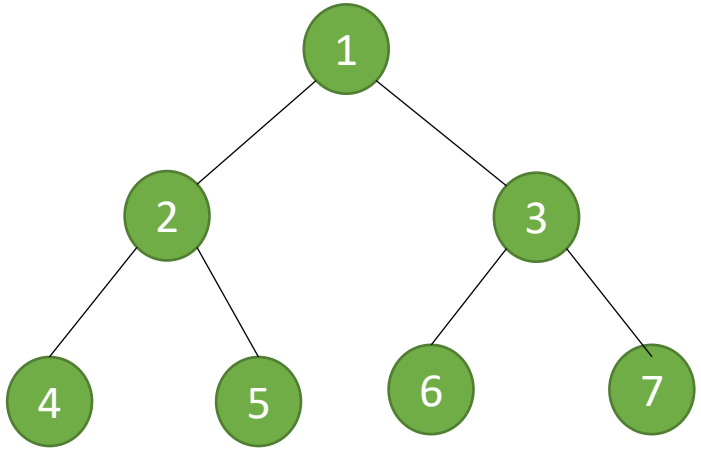
Full Binary Tree

2.



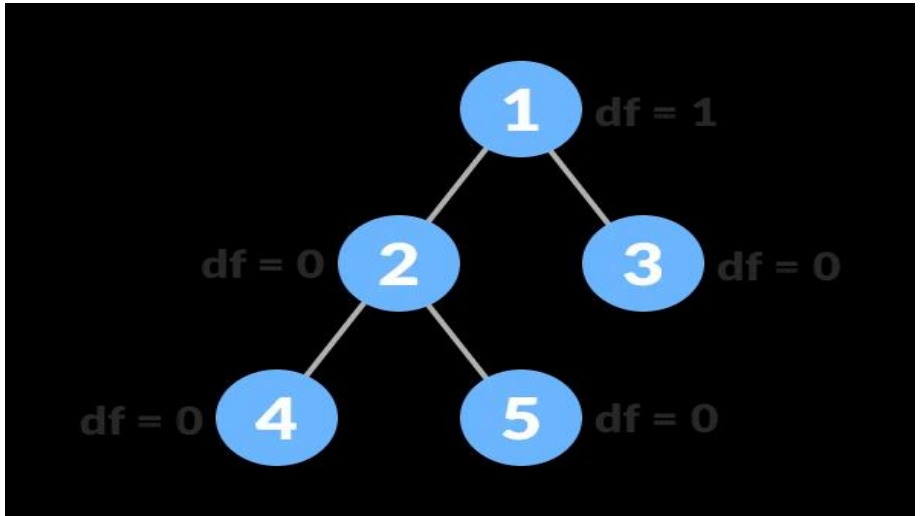
Almost Complete Binary Tree

3.



Complete Binary Tree

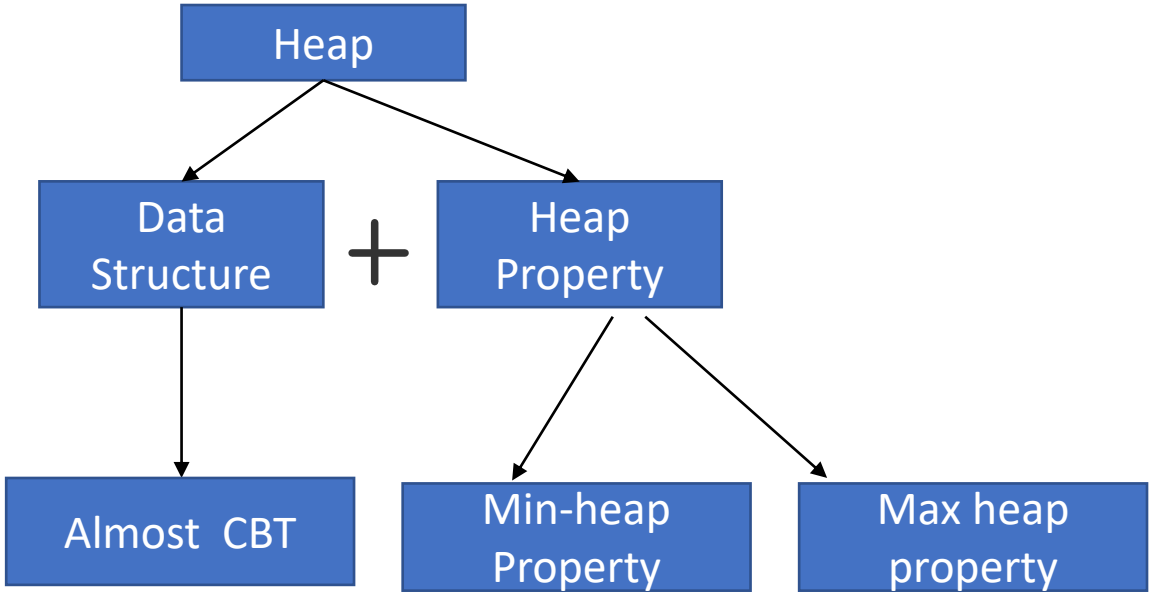
- Full Binary Tree:A Binary Tree is a full binary tree if every node has either 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.
- Almost Complete Binary Tree: A Binary Tree is an Almost Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.
- Complete Binary Tree: A Binary tree is a Complete Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.



df= height of the left child –height of right child

Heap

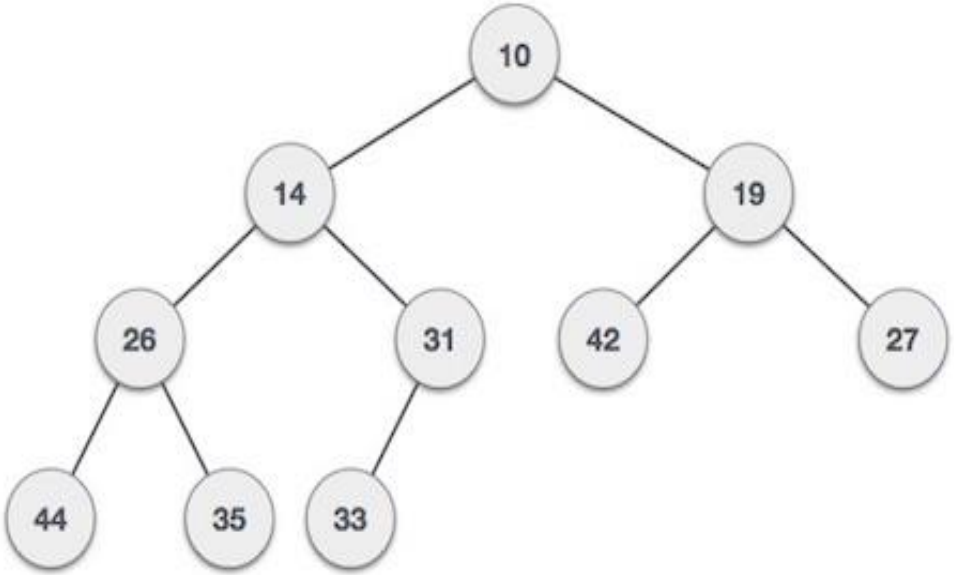
Heap is an almost complete binary tree with some specific property, i.e. heap property.

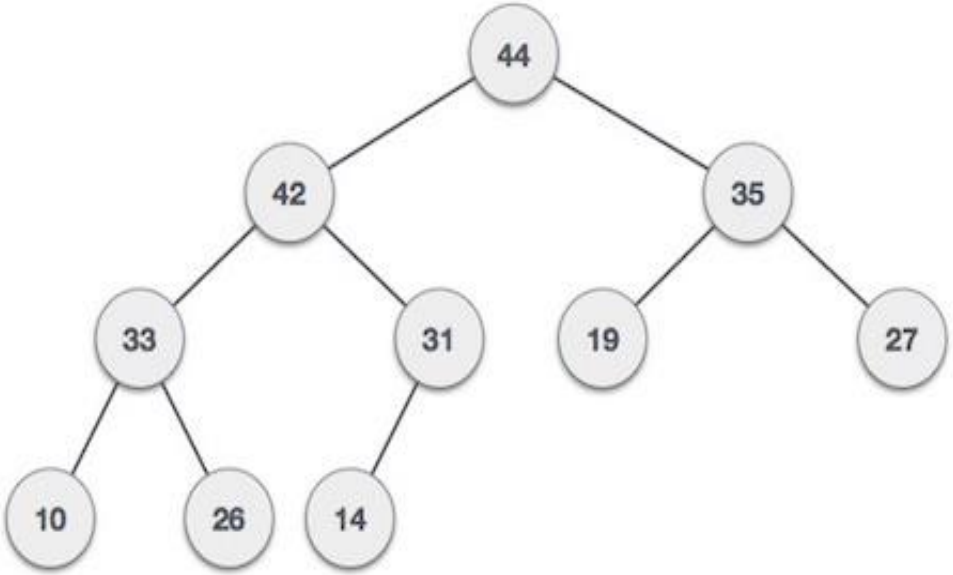


As the value of parent is greater than that either of its children, this property generates **Max Heap**

whereas the value of the root node is less than or equal to either of its children, generates **Min Heap**

Heap





Min-Heap − Where the value of the root node is less than or equal to either of its children

Max-Heap − Where the value of the root node is greater than or equal to either of its children.

Applications of Heap Data Structure

Heaps are efficient for finding the min or max element in an array and are useful in order statistics and selection algorithms. The time complexity of getting the minimum/maximum value from a heap is O(1), (constant time complexity).

Priority queues are designed based on heap structures. It takes O(log(n)) time to insert (insert()) and delete (delete()) each element in the priority queue efficiently.

Heap-implemented priority queues are used in popular algorithms like:

- Prim’s algorithm
- Dijkstra’s algorithm
- Heapsort algorithm
- Finding the k-th smallest/largest element

Essential Operations in Heaps

The following are the essential operations you might use when implementing a heap data structure:

Build Heap: It is important to be able to construct a heap. Build-Heap is usually implemented using the Insert and Heapify function repeatedly. So starting from an empty heap, nodes are added with Insert and then Heapify is called to make sure the heap maintains the heap properties at each step.

heapify: rearranges the elements in the heap to maintain the heap property.
insert: adds an item to a heap while maintaining its heap property.
delete: removes an item in a heap.
extract: returns the value of an item and then deletes it from the heap.
isEmpty: boolean, returns true if boolean is empty and false if it has a node.
size: returns the size of the heap.
getmin(): returns the minimum value in a heap

Heap

Min Heap construction algorithm:

Step 1 − Create a new node
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 greater than child, then swap them.
Step 5 − Repeat step 3 & 4 until heap property holds.

Operations need to be carried out:

1. build-heap(A) or make-heap(A)------ using heap-insertion(A,x)
2. extract-min(A)
3. Heapsort(A)

Heap

Array Representation

A binary heap is represented as an array. The representation follows some property.

- The root of the tree will be at Arr[0].
- For any node at Arr[i], its left and right children will be at Arr[2*i + 1] and Arr[2*i+2] respectively.
- For any node at Arr[i], its parent node will be at Arr[(i-1)/2].

Why Array?

Since a Binary Heap is an Almost Complete Binary Tree, it can be easily represented as an array and array-based representation is space-efficient.

Heap as a Tree

root of tree: first element in the array, corresponding to i = 1
parent(i) =i/2: returns index of node's parent
left(i)=2i: returns index of node's left child
right(i)=2i+1: returns index of node's right child

Height of a binary heap is O(lg n)

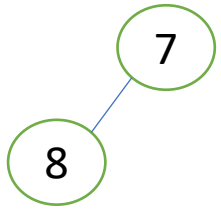
Min-Heap Construction

S[ ]

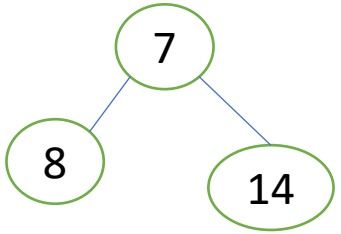
7	8	14	10	9	16	2	3	1	4

Step-1:

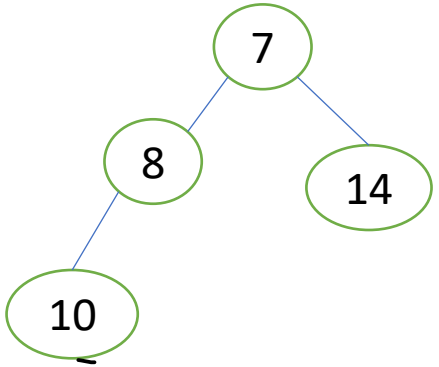
Step-2:



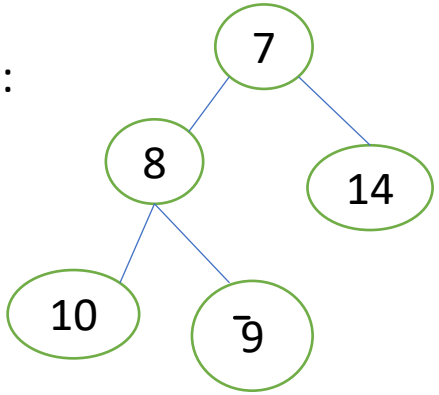
Step-3:



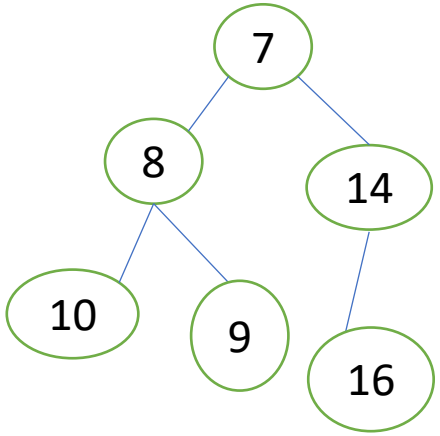
Step-4:



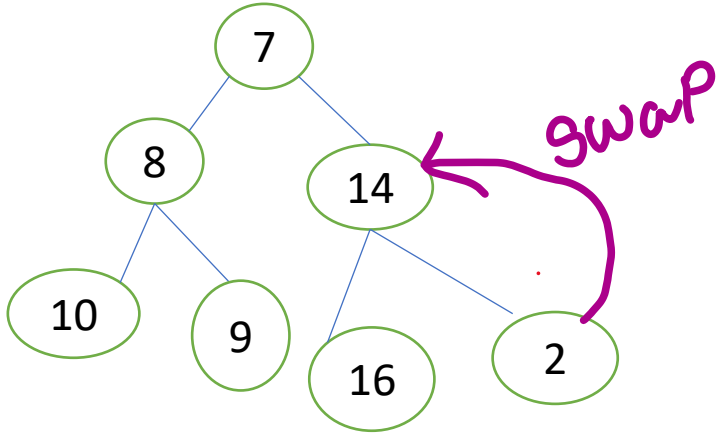
Step-5:



Step-6:

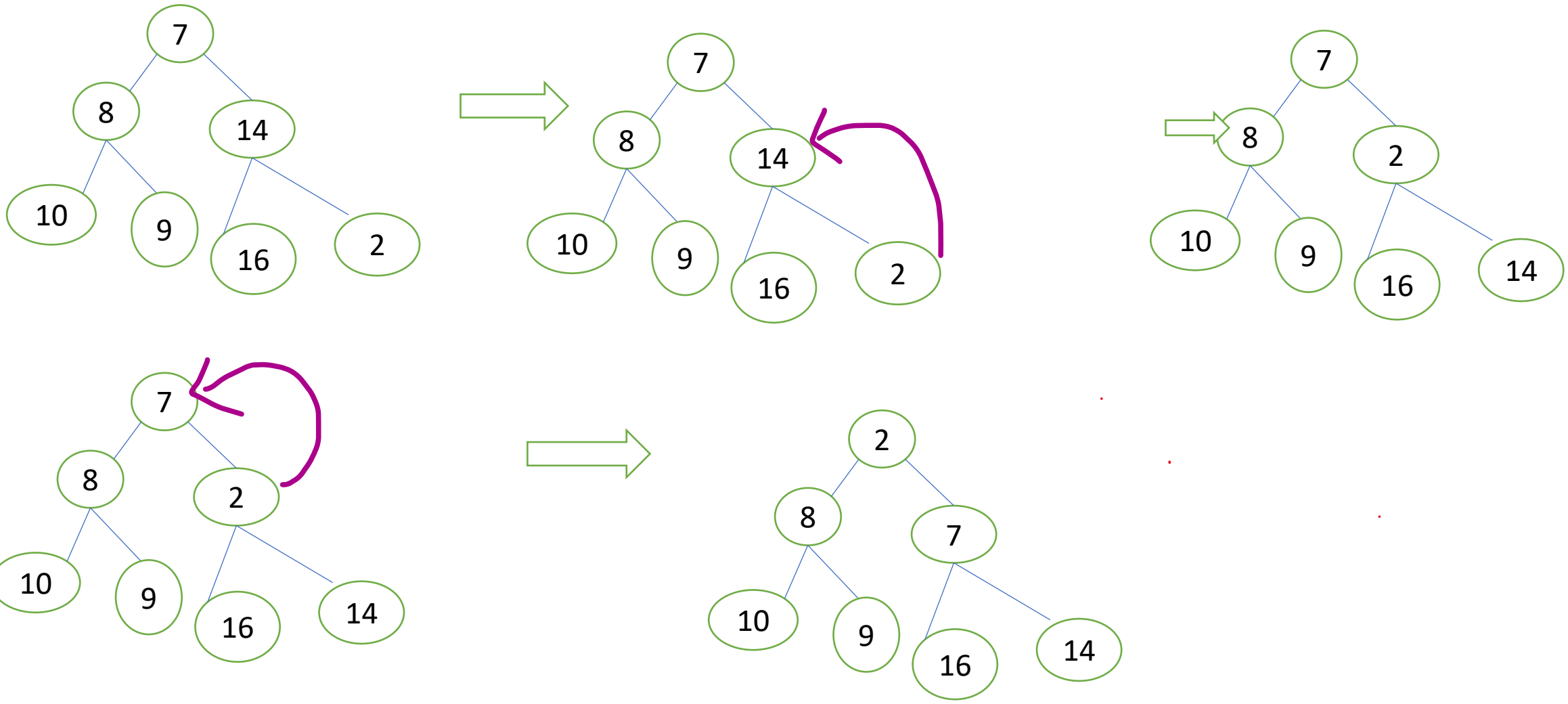


Step-7:



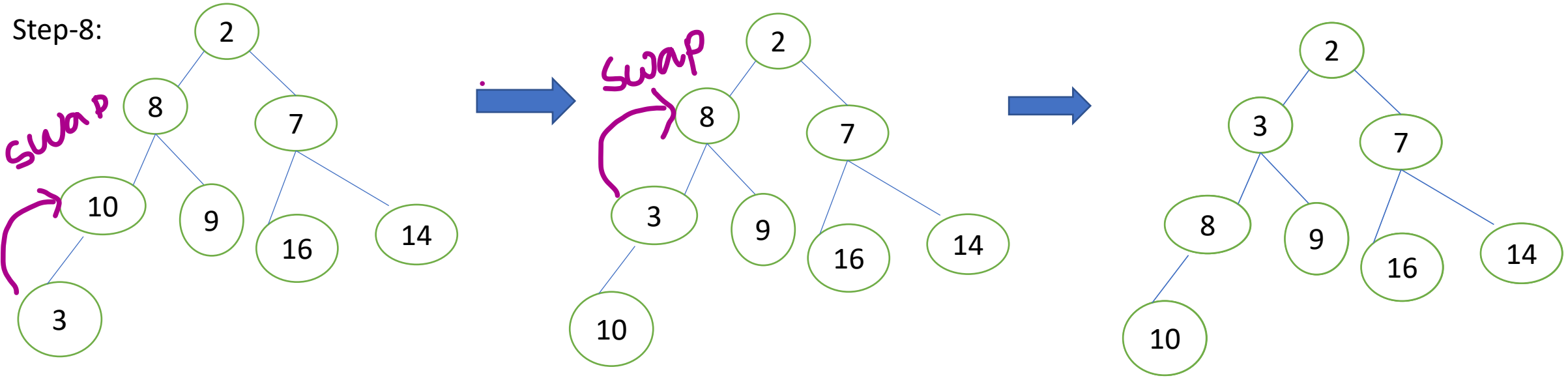
Min-Heap Construction

Step-7

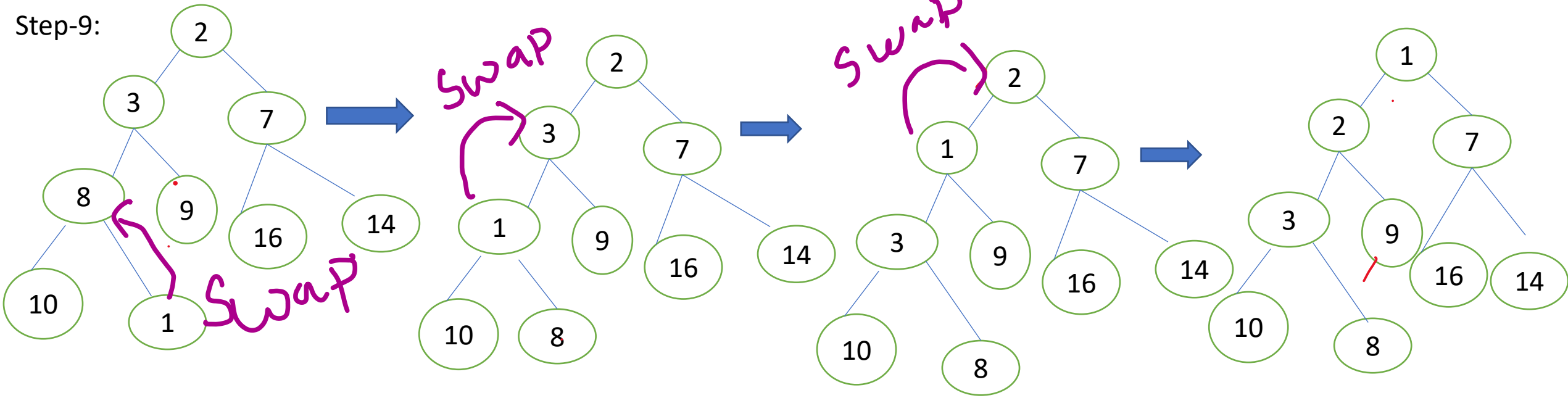


Min-Heap Construction

Step-8:

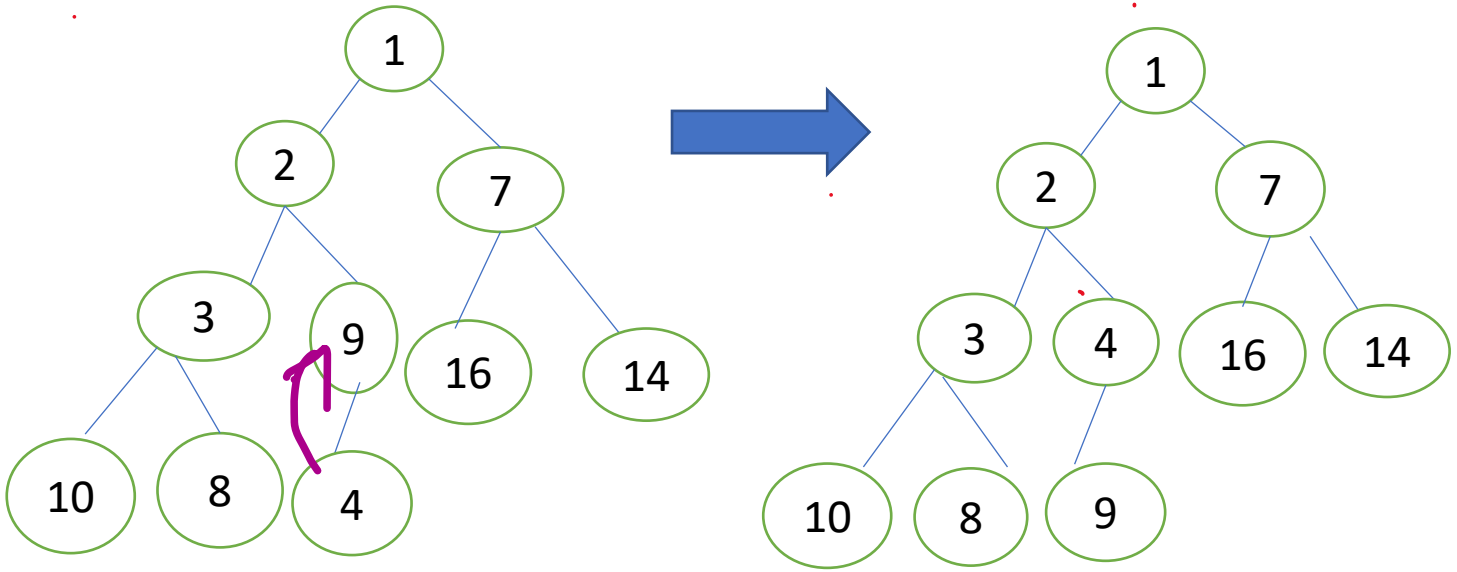


Step-9:



Min-Heap Construction

Step-10:



Min-heap

82, 90 , 10, 12 ,15 ,77, 55 ,23

Operation:

1. **Build Heap/make_heap**
2. **Insert**
3. **Heapify/bubble-up**

Min-Heap Construction

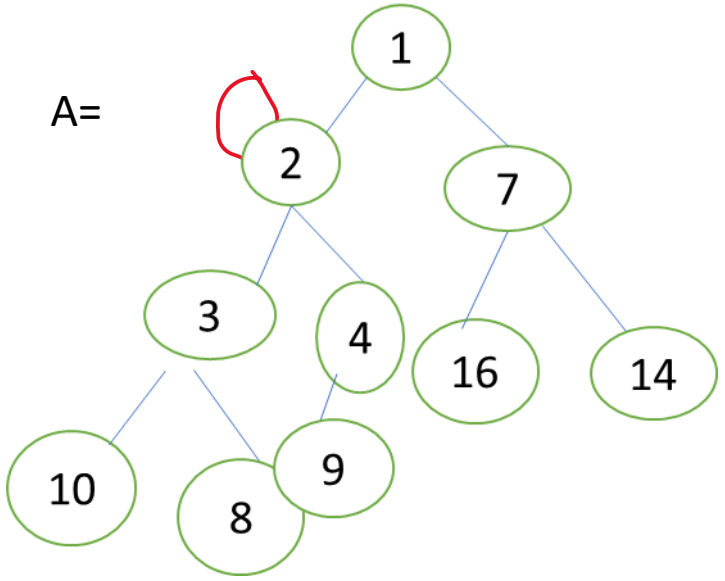
```
make_heap(A[],S[],n)
 {
      heapsize(A[])=0;
      length(A[])=n;
       for i=1 to length(A[]) do
            heap_insert(A,S[i]);
}
```

i

S[]	7	8	14	10	9	16	2	3	1	4

heapsize =0(number of element in a heap structure)
length =n

A=

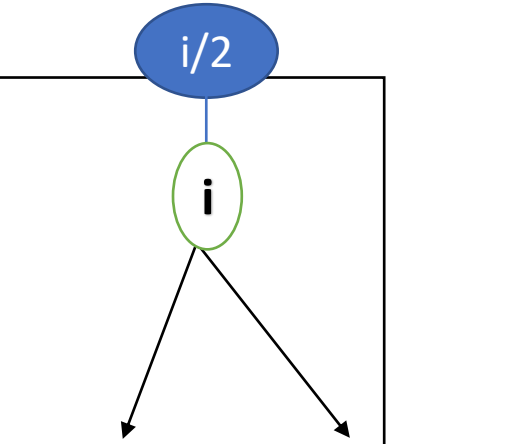



```
heap_insert(A[],x)
{
  if(heapsize(A[]>=length(A[])) then
        {  print "heap is full hence overflow occurs";
         return;
         }
else {
          heapsize(A[])=heapsize(A[])+1;
         A[heapsize(A[])]=x;
          bubble_up(A[],heapsize(A[]));  }
 }
```

Min-Heap Construction

```
make_heap(A[],S[],n)
 {     heapsize(A[])=0;
       length(A[])=n;
        for i=1 to length(A[]) do
             heap_insert(A,S[i]);
}
heap_insert(A[],x)
{  if(heapsize(A[]>=length(A[])) then
          {  print "heap is full hence overflow occurs";
             return;
           }
    else {   heapsize(A[])=heapsize(A[])+1;
             A[heapsize(A[])]=x;
             bubble_up(A[],heapsize(A[]));  }
 }
```

```
bubble_up(A[],i)
{
  if (parent(i)==0) then return;
  if(A[parent(i)>A[i]) then
  {
    swap(A[i],A[parent(i)]
    bubble_up(A[],parent(i));
  }
}
```



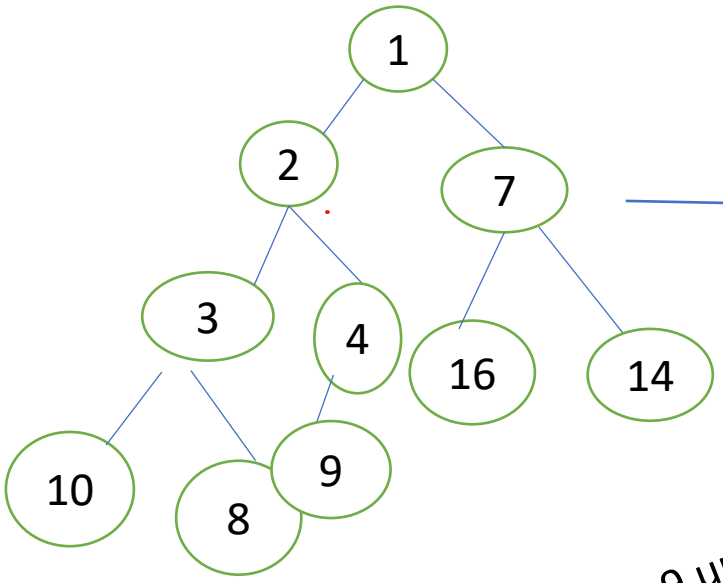
T(n)=T(n/2)+c

Heap-Sort

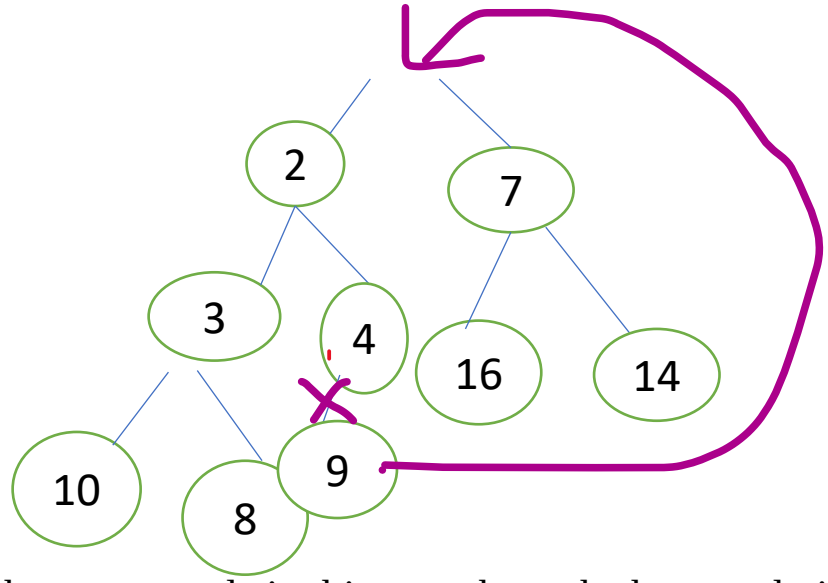
Step-by-Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

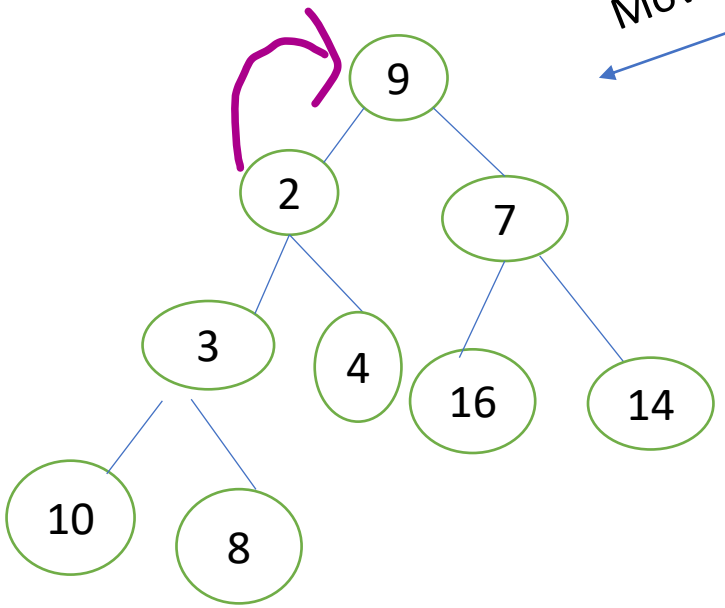
- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min-Heap.**
- **Step 3 -** Delete the root element from Min-Heap
- **Step 4 -** Put the deleted element into the sorted list.
- **Step 5 -** Repeat the same until Min-Heap becomes empty.
- **Step 6 -** Display the sorted list.



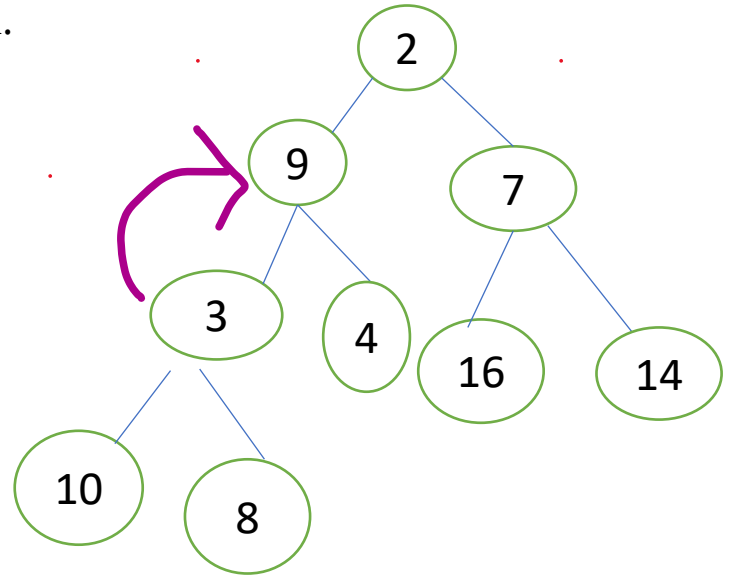
Remove 1 ,



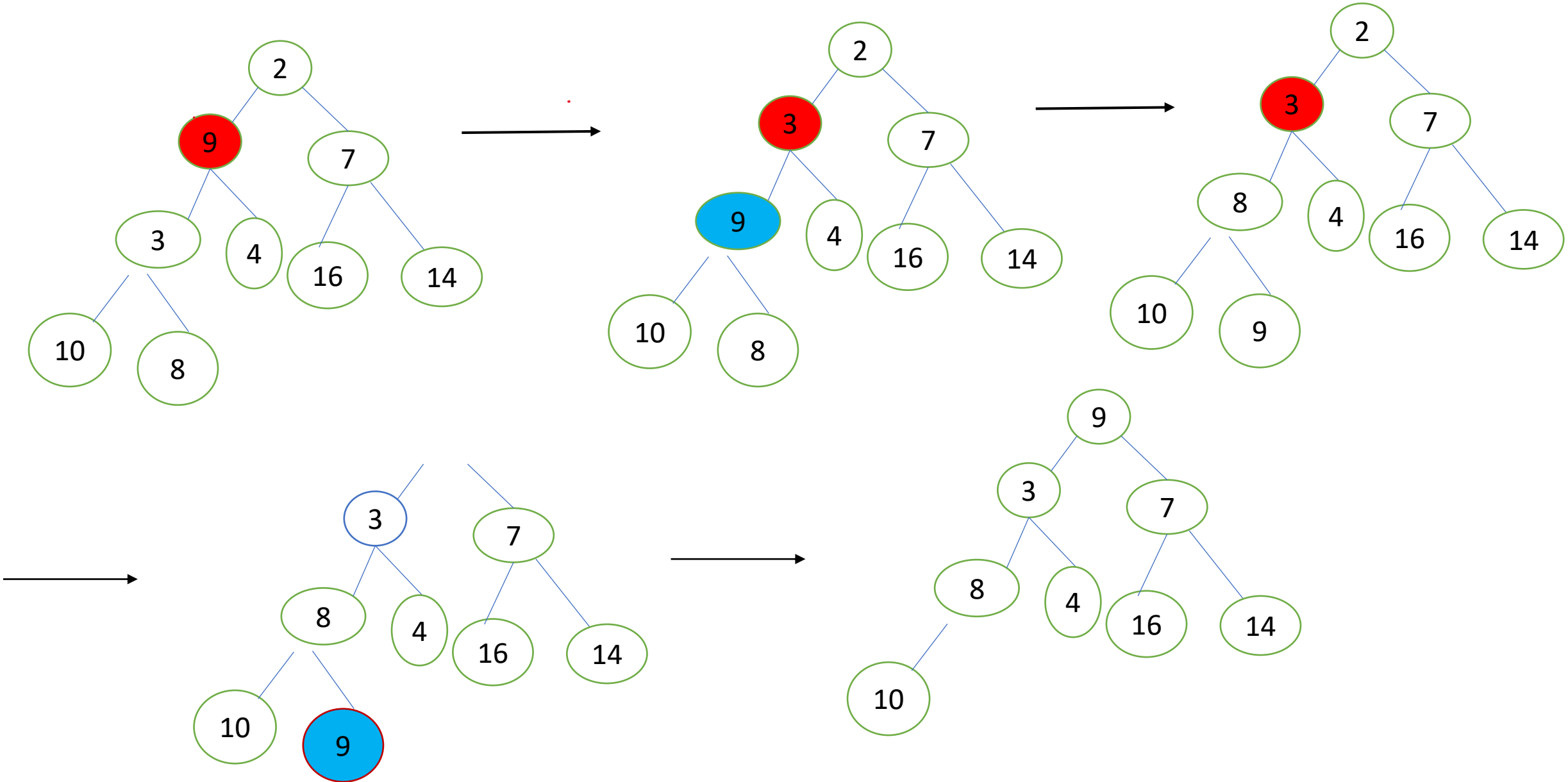
Move 9 upto the heap



Notice that every node in this tree obeys the heap-ordering property except for the root itself. If we swap the 9 with one of its kids, we can push the problem down a level.



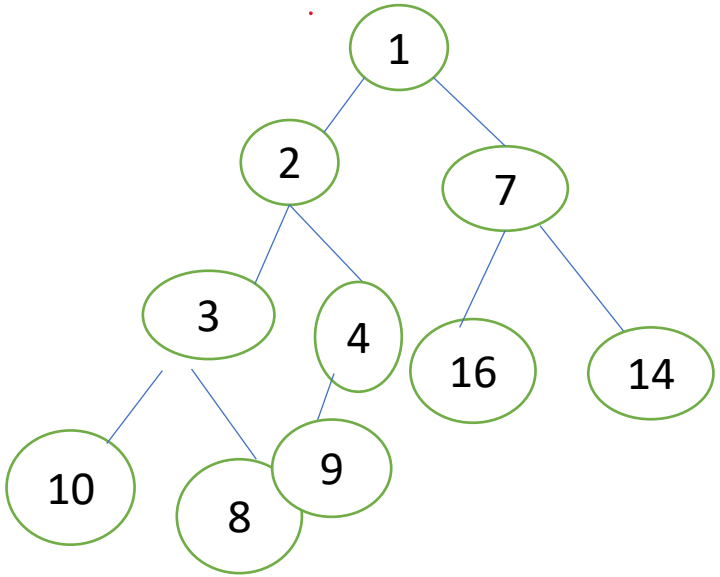
•While the item is smaller than either of its children, swap it with the smallest of its children. (This step is often known as the “bubble down” operation.)



Heap-Sort

```
heapsort(S[],n)
{ make_heap(A[], S[], n);
  for i=1 to n do ——————— O(n)
      S[i] = extract_min(A[]); ——— O(n) × O(log n)
}                                  = O(n log n)
```

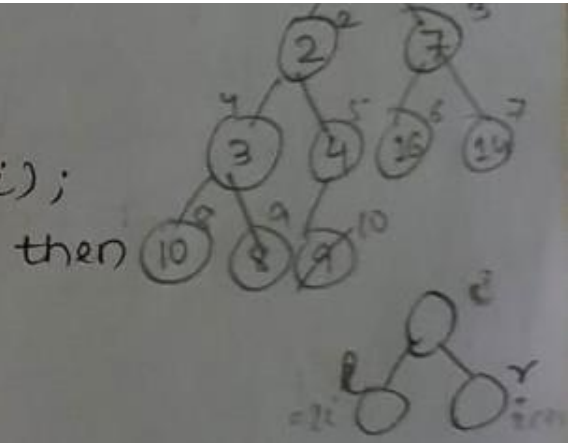
```
extract_min(A[])
{ min = -1;
  if (heapsize(A[]) ≤ 0) then print "empty heap";
  else                                    O(log n)
     { min = A[1];
       A[1] = A[heapsize(A[])];
       heapsize(A[]) = heapsize(A[]) - 1;
       bubble_down(A[], 1);
     }
  return (min);
}
```

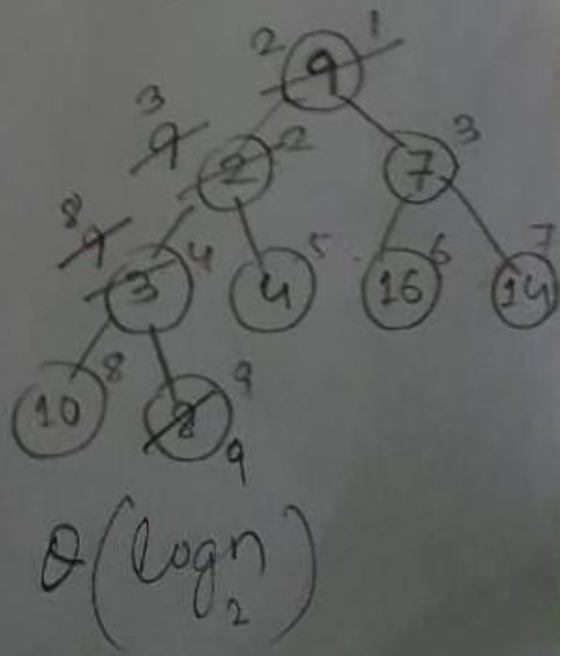


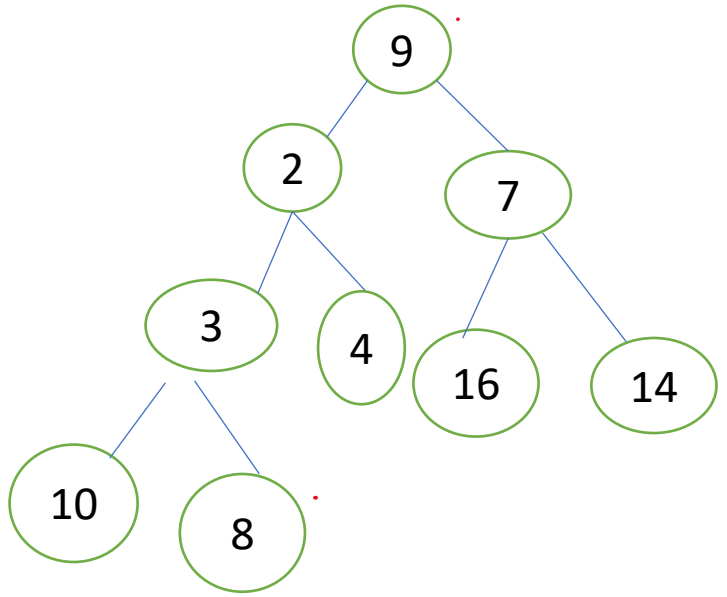
min=-1
heapsize(A[])=n <=0
else
min=A[1] =1
A[1]=A[n]
heapsize(A[])= n-1
bubble_down(A[],1) (root=1)

Heap-Sort

```
bubble_down (A[], i)
{ l = leftchild(i) ; r = rightchild(i) ;
  if(l ≤ heapsize(A[]) and A[l] < A[i]) then
          smallest = l ;
  else
          smallest = i ;
  if(r ≤ heapsize(A[]) and A[r] < A[smallest]) then
          smallest = r ;
  if (smallest ≠ i) then
      { swap (A[i], A[smallest]) ;
        bubble_down(A[], smallest) ;
      }
}
```







i=1
l=leftchid r=rightchild
l<= heapsize(A[]) 2< 9
 smallest=l
r<= heapsize(A[]) 7<2
 swap(9,2)
bubble_down(A[], smallest)