Heaps

- Heap as a Data Structure
- Types of heap – Min heap and Max heap
- Operations on Heap
- Heap Sort
- Applications of Heap

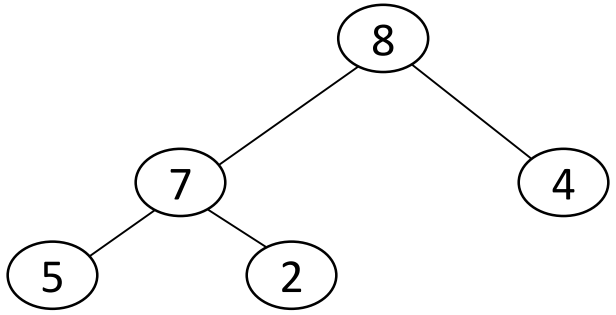
The Heap Data Structure

Def: A **max** (min) heap is a tree in which the **key value** in each node is **no smaller** (larger) than the key values in its **children** (if any).

- A max heap is a complete binary tree that is also a max tree.
- A min heap is a complete binary tree that is also a min tree.

Order (heap) property: for any node x (for Max heap)

Parent(x) ≥ x



Heap

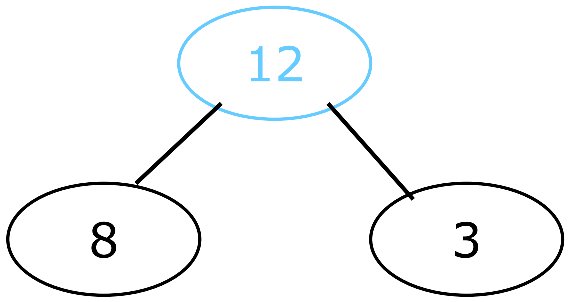
Example of Max heap

"The root is the maximum element of the heap!"

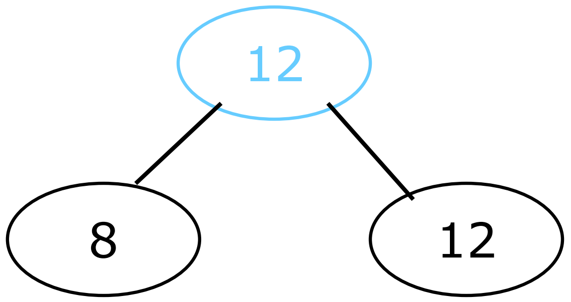
A heap is a binary tree that is filled in order

The heap property **(for Max heap)**

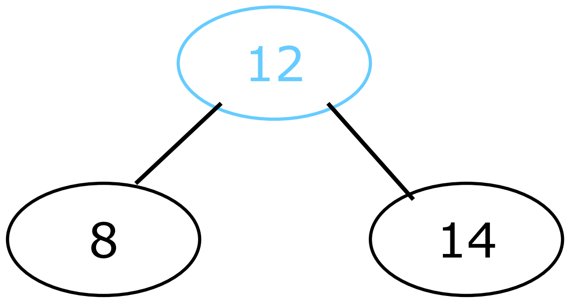
A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property



Blue node does not have heap property

All leaf nodes automatically have the heap property

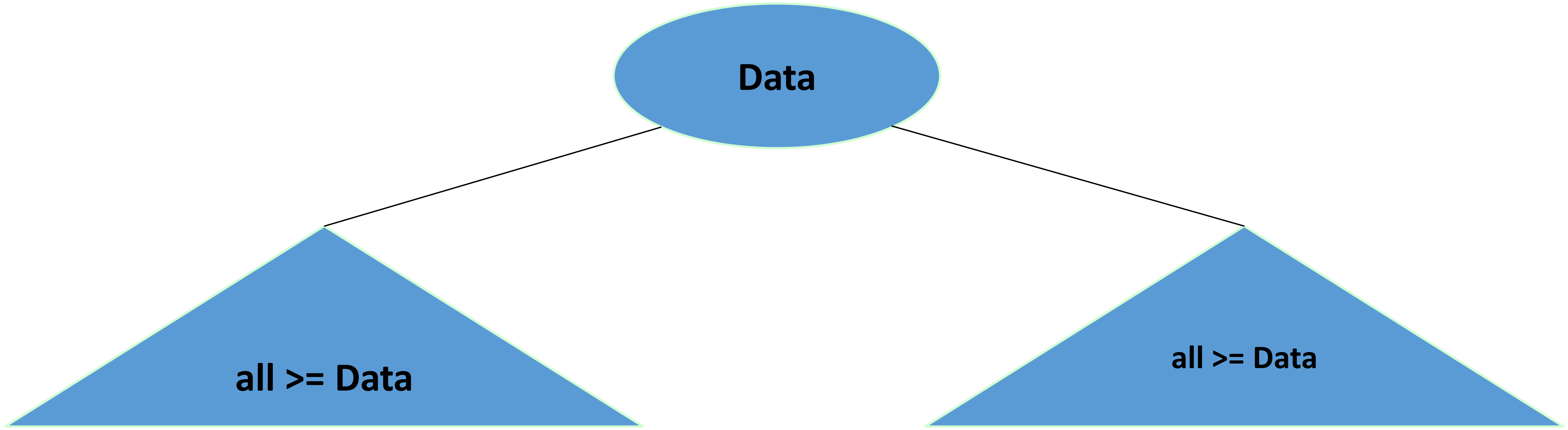
A binary tree is a heap if *all* nodes in it have the heap property

Types of Heap

- **Min-heap**

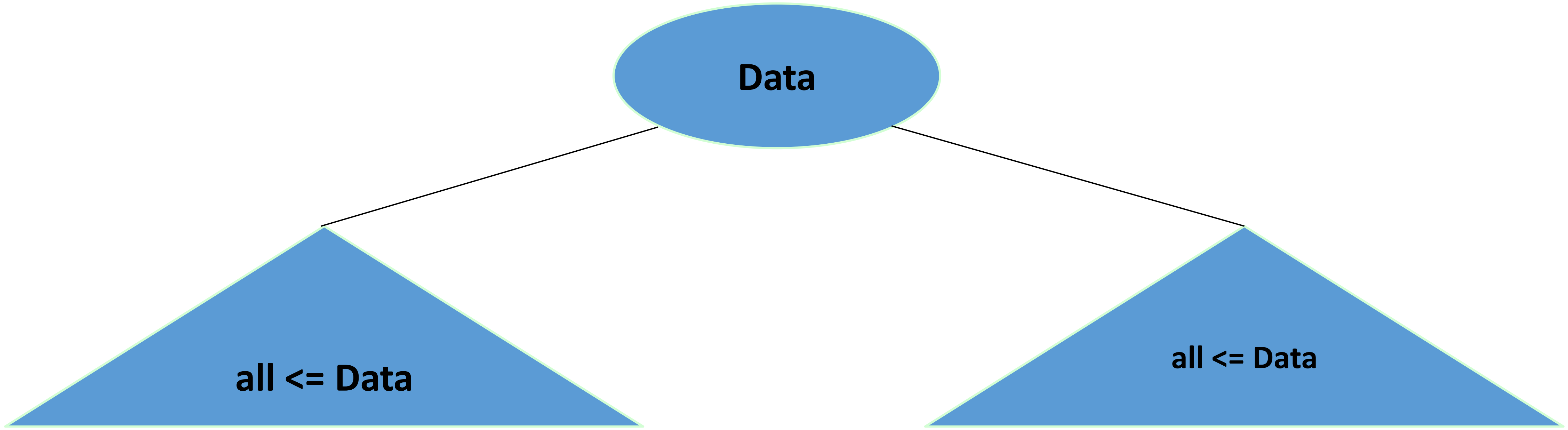
- **Max-heap**

Min Heap



- **In min-heap, the key value of each node is lesser than or equal to the key value of its children**
- **In addition, every path from root to leaf should be sorted in ascending order**

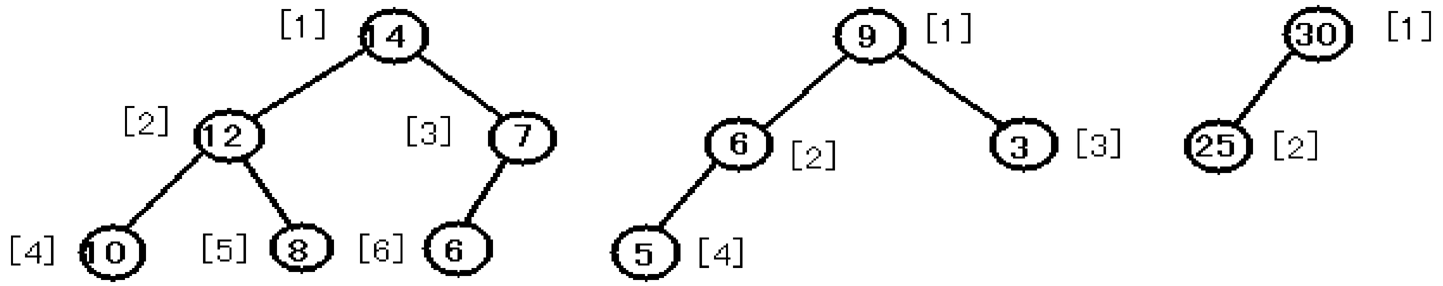
Max Heap

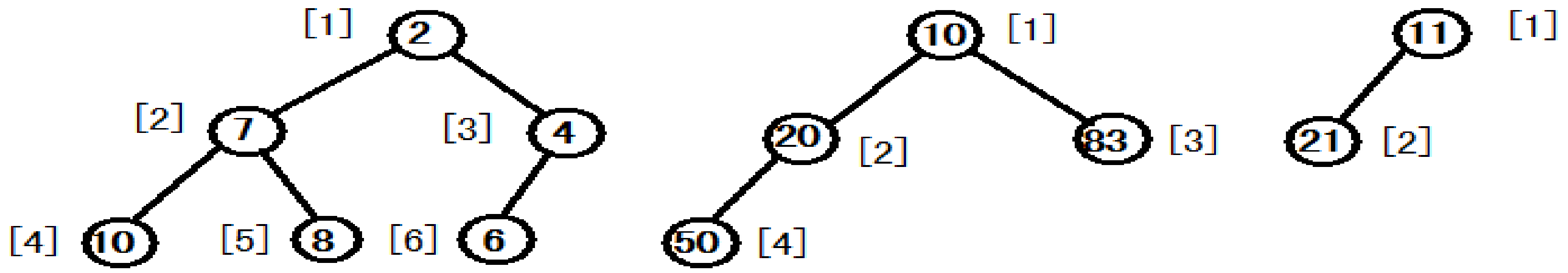


❖ **A max-heap is where the key value of a node is greater than the key values in of its children**

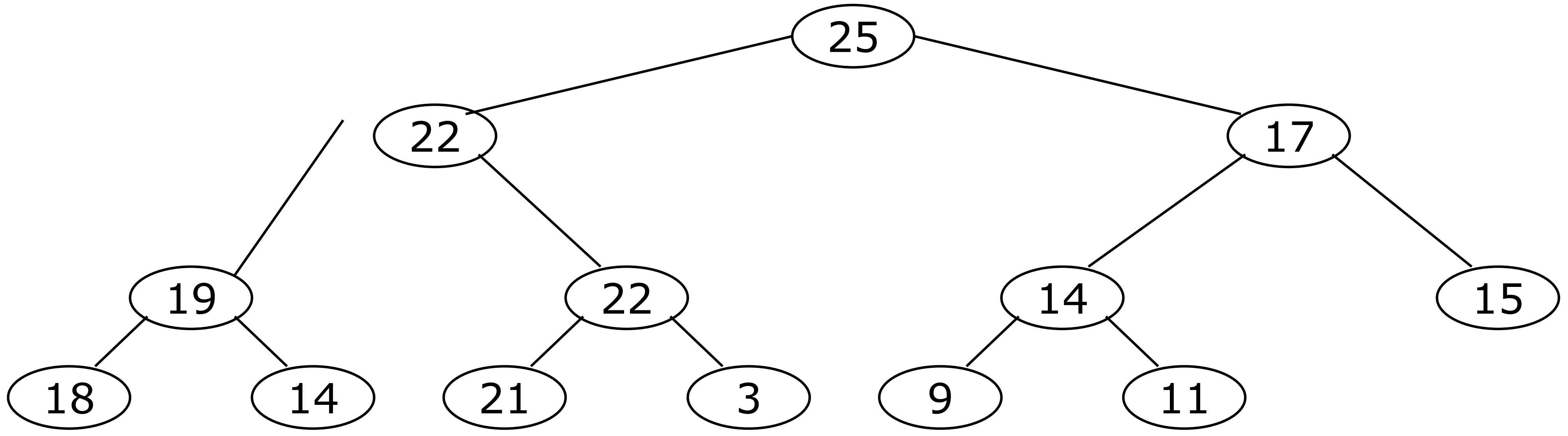
Example of Heap

Max-Heap





Mapping into an array



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Notice:

- The left child of index i is at index 2*i+1(index starting from 0)
- The right child of index i is at index 2*i+2
- Example: the children of node value (19) [index 3] are at [index 7] (18) and [index 8] (14)

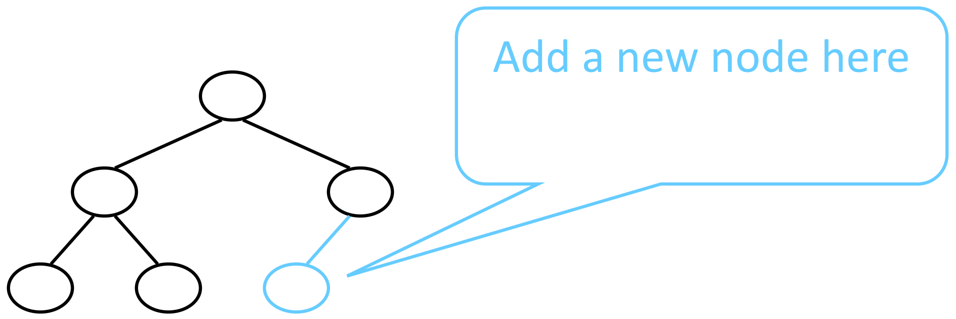
MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

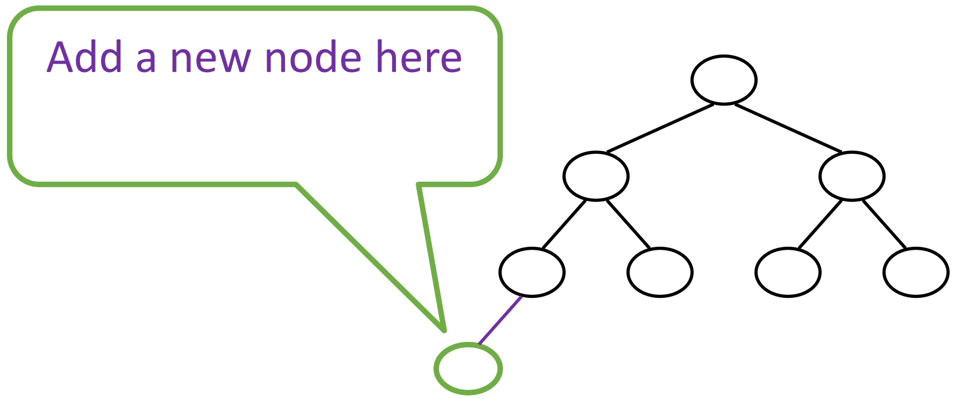
Operations on Heaps

- **Create—To create an empty heap to which ‘root’ points**
- **Insert—To insert an element into the heap**
- **Delete—To delete max (or min) element from the heap**
- **ReheapUp—To rebuild heap when we use the insert() function**
- **ReheapDown—To build heap when we use the delete() function**

Constructing a (max) heap

- ❖ A tree consisting of a single node is automatically a heap
- ❖ We construct a heap by adding nodes one at a time:
  - ❖ Add the node just to the right of the rightmost node in the deepest level
  - ❖ If the deepest level is full, start a new level
- ❖ Examples:



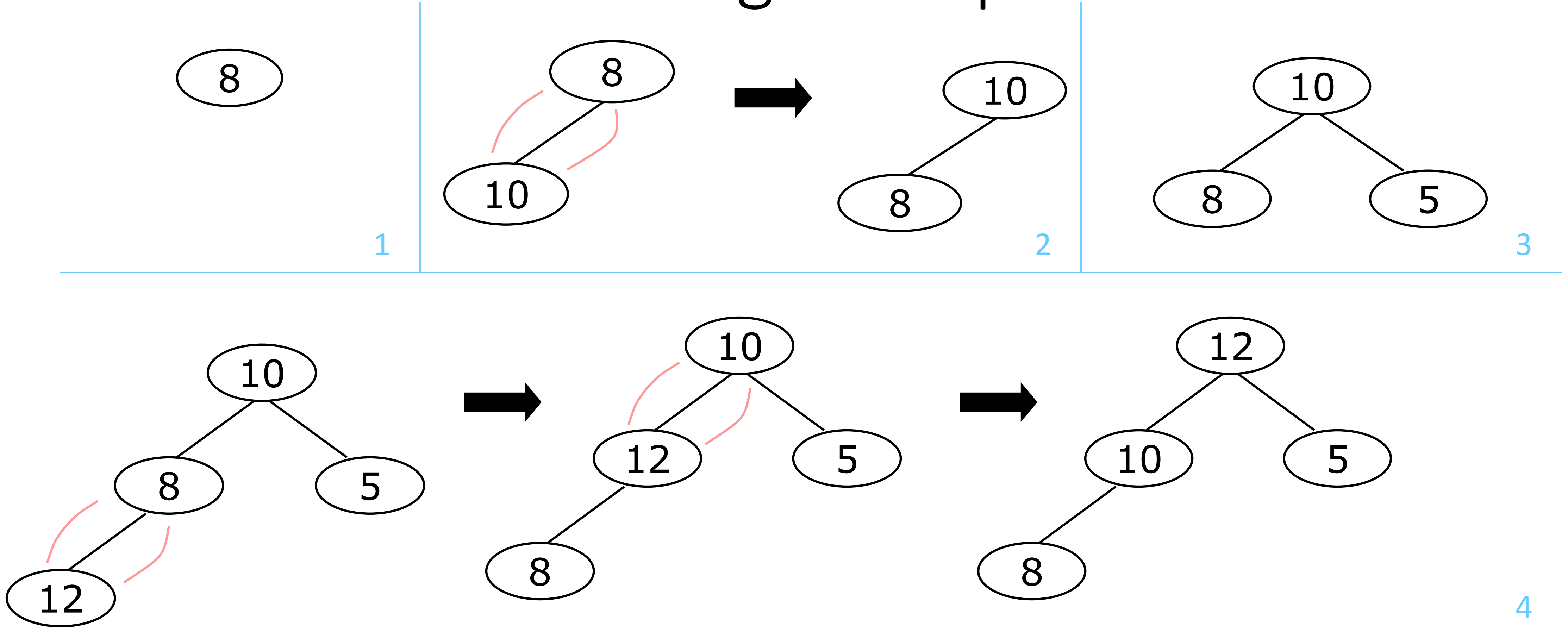


Constructing a (max)heap contd…

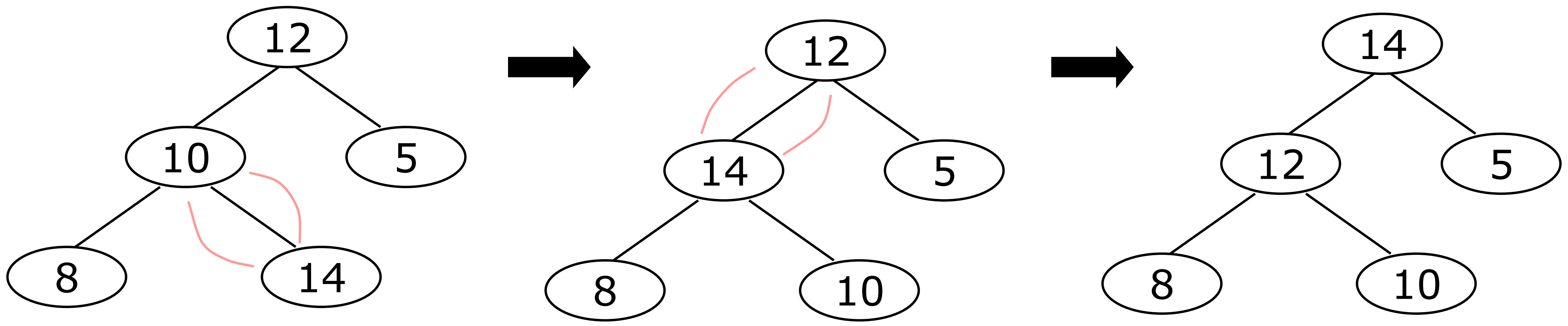
- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we shift up
- But each time we shift up, the value of the topmost node in the shift may increase, and this may destroy the heap property of *its* parent node
- We repeat the shifting up process, moving up in the tree, until either
    - We reach nodes whose values don’t need to be swapped (because the parent is *still* larger than both children), or
    - We reach the root

Constructing a heap

contd…



Other children are not affected



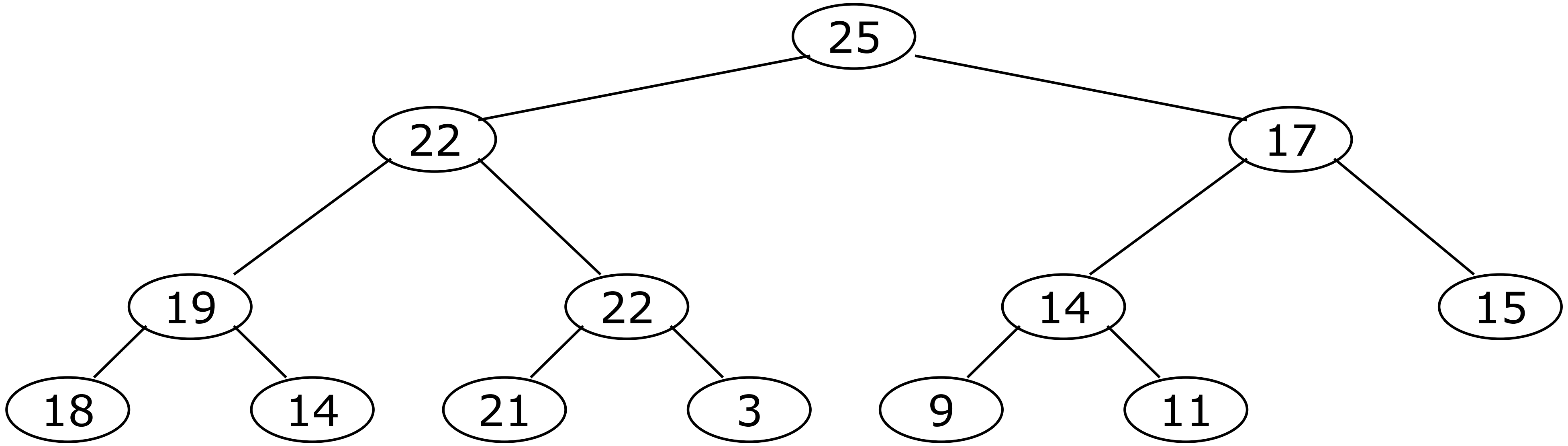
❖ The node containing 8 is not affected because its parent gets larger, not smaller

❖ The node containing 5 is not affected because its parent gets larger, not smaller

❖ The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

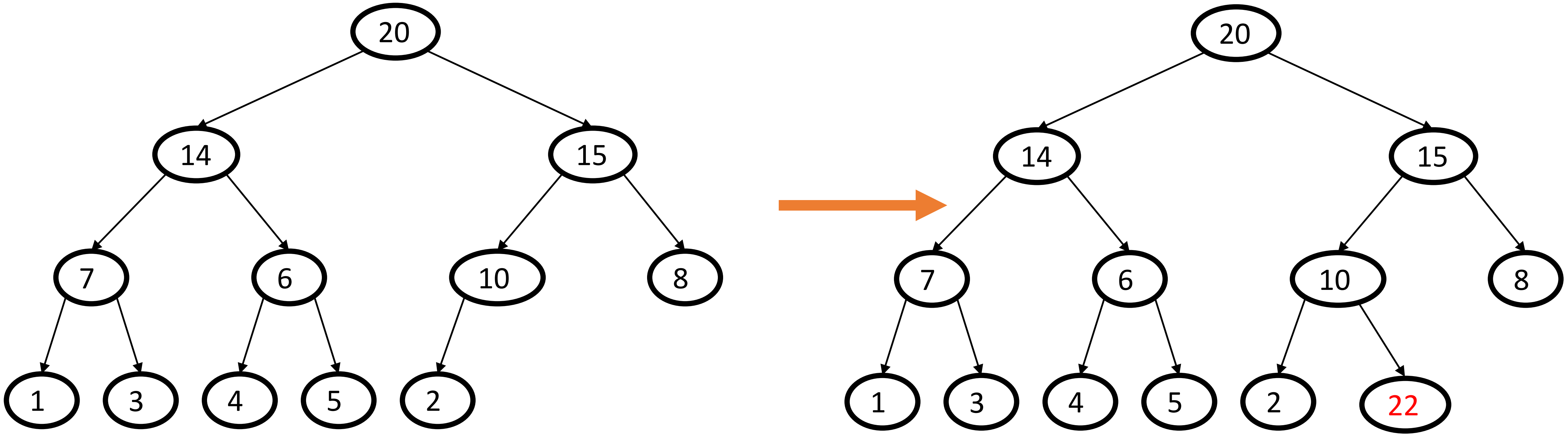
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

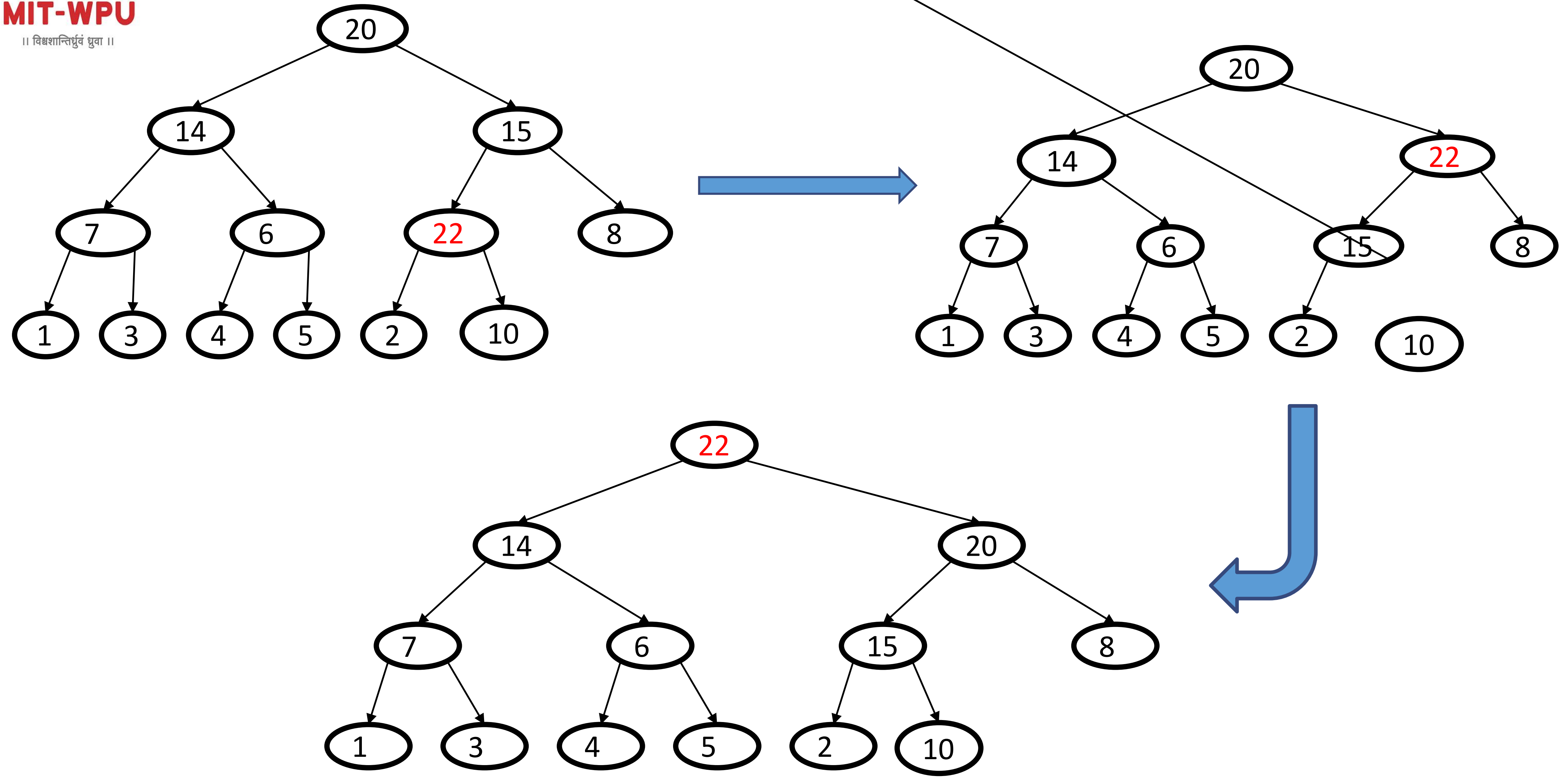
Insert (max heap)

`insert(22)`



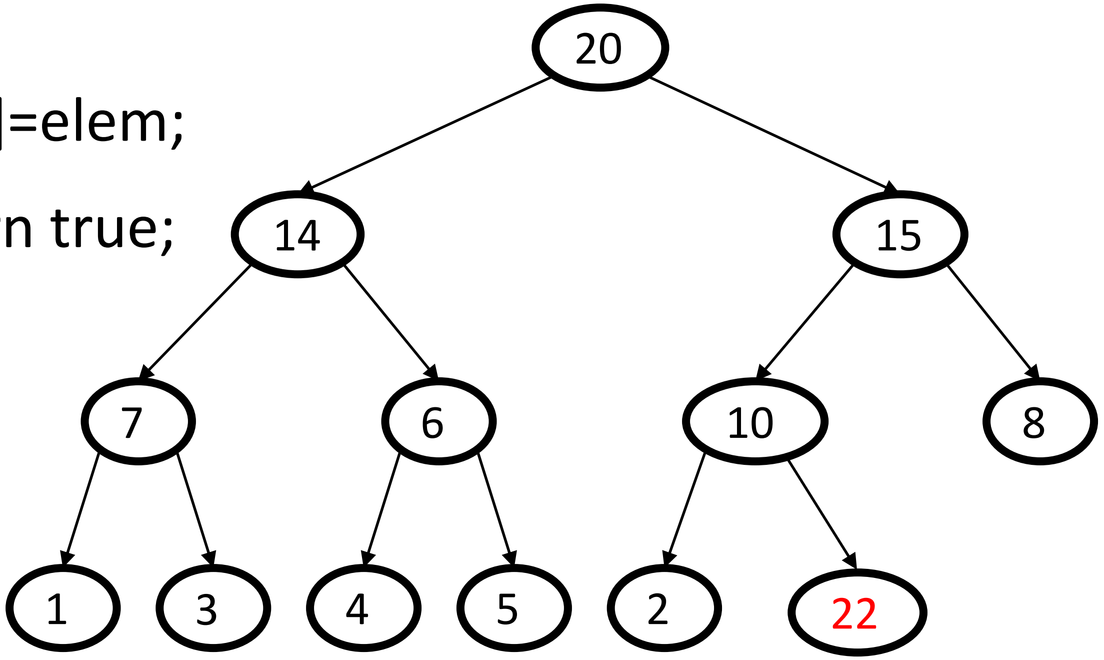
Insert (max heap)

contd…



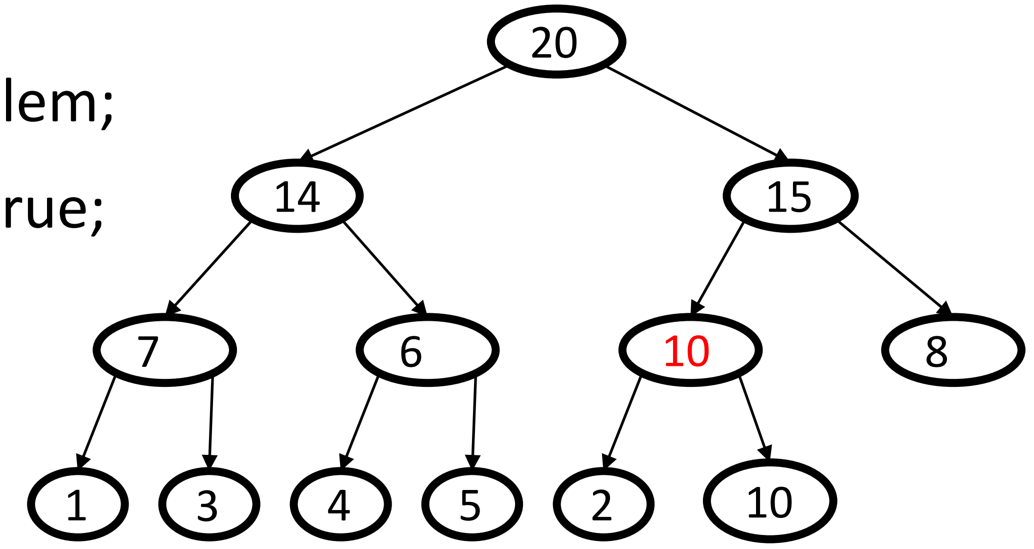
```
Algorithm add()
{     //index of a starting from 0
      j=0 ; a[20];
      Get Choice
     Repeat
     {
      a[j]=elem;
       insert(a,j+1);
       j++;
      } until(choice=='n');
}
```

```
Algorithm Insert(a,n)
{
     i=n;  elem=a[n-1];
     if(i!=1){
          while((i>0)&&(a[(i/2)-1]<elem){
                    a[i-1]=a[(i/2)-1];
                    i=(i/2);
          }
          a[i-1]=elem;
          return true;
     }
}
```



```
Algorithm add()
{     //index of a starting from 0
      j=0 ; a[20];
      Get Choice
     Repeat
     {
      a[j]=elem;
        insert(a,j+1);
        j++;
      } until(choice==‘n’);
}
```

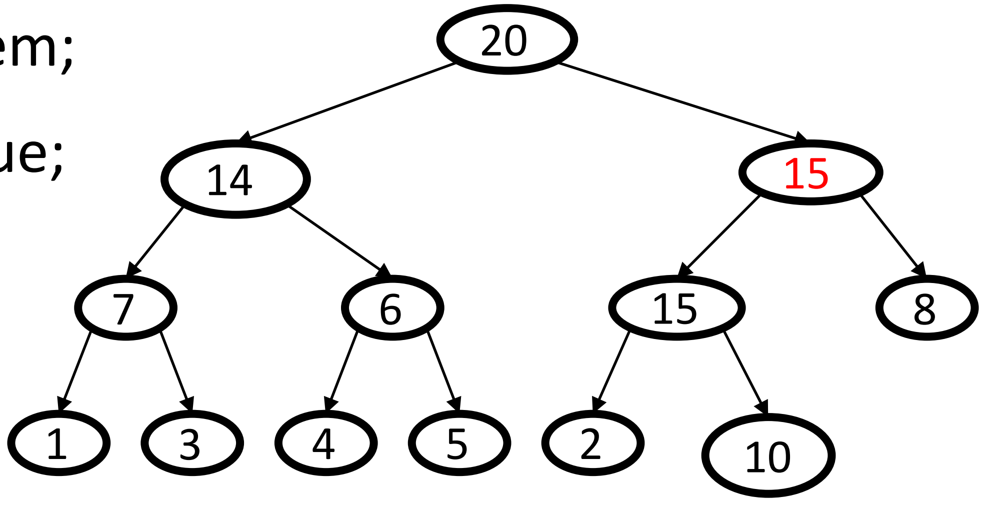
```
Algorithm Insert(a,n)
{
     i=n;  elem=a[n-1];
     if(i!=1){
          while((i>0)&&(a[(i/2)-1]<elem){
                    a[i-1]=a[(i/2)-1];
                    i=(i/2);
          }
          a[i-1]=elem;
          return true;
     }
}
```



elem = 22

```
Algorithm add()
{     //index of a starting from 0
      j=0 ; a[20];
     Get Choice
    Repeat
    {
     a[j]=elem;
       insert(a,j+1);
       j++;
     } until(choice=='n');
}
```

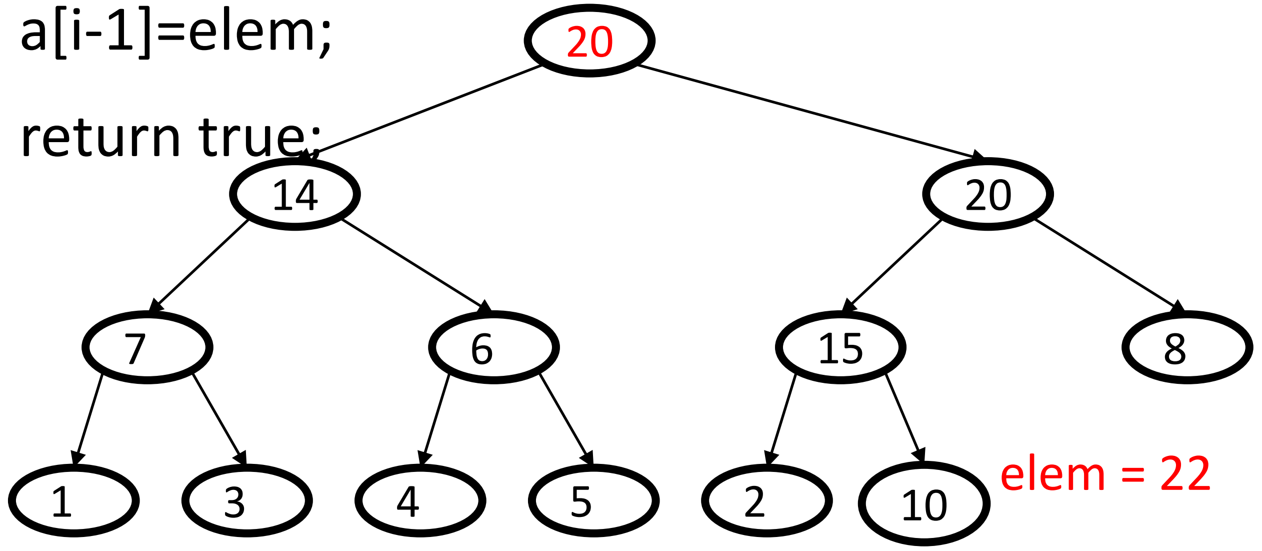
```
Algorithm Insert(a,n)
{
     i=n;  elem=a[n-1];
     if(i!=1){
          while((i>0)&&(a[(i/2)-1]<elem){
                  a[i-1]=a[(i/2)-1];
                  i=(i/2);
          }
          a[i-1]=elem;
          return true;
     }
}
```



elem = 22

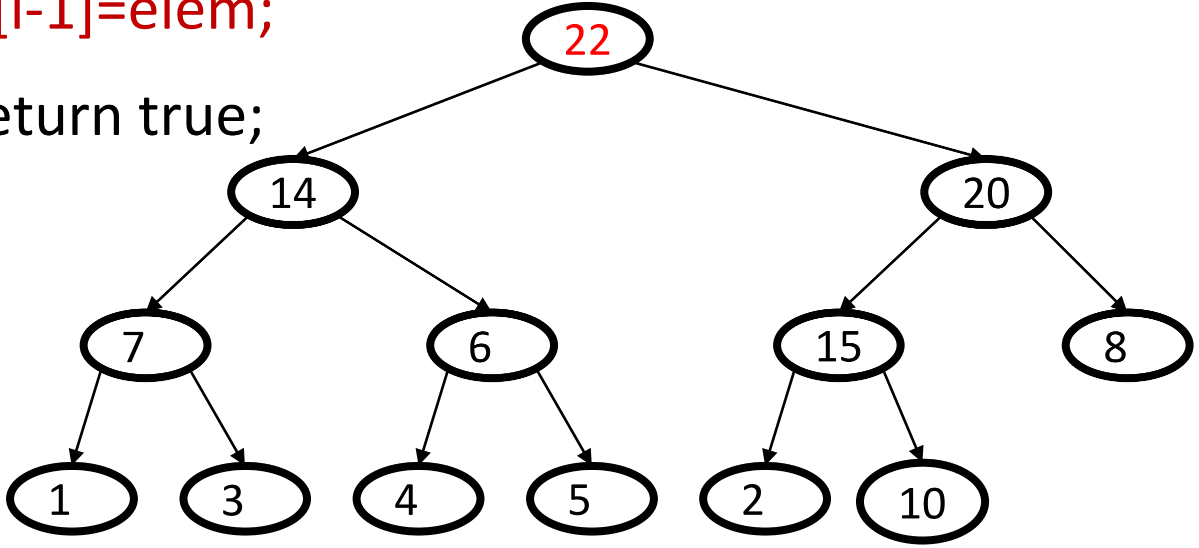
```
Algorithm add()
{     //index of a starting from 0
      j=0 ; a[20];
      Get Choice
     Repeat
     {
      a[j]=elem;
        insert(a,j+1);
        j++;
      } until(choice==‘n’);
}
```

```
Algorithm Insert(a,n)
{
     i=n;  elem=a[n-1];
     if(i!=1){
          while((i>0)&&(a[(i/2)-1]<elem){
                    a[i-1]=a[(i/2)-1];
                    i=(i/2);
          }
          a[i-1]=elem;
          return true;
     }
}
```



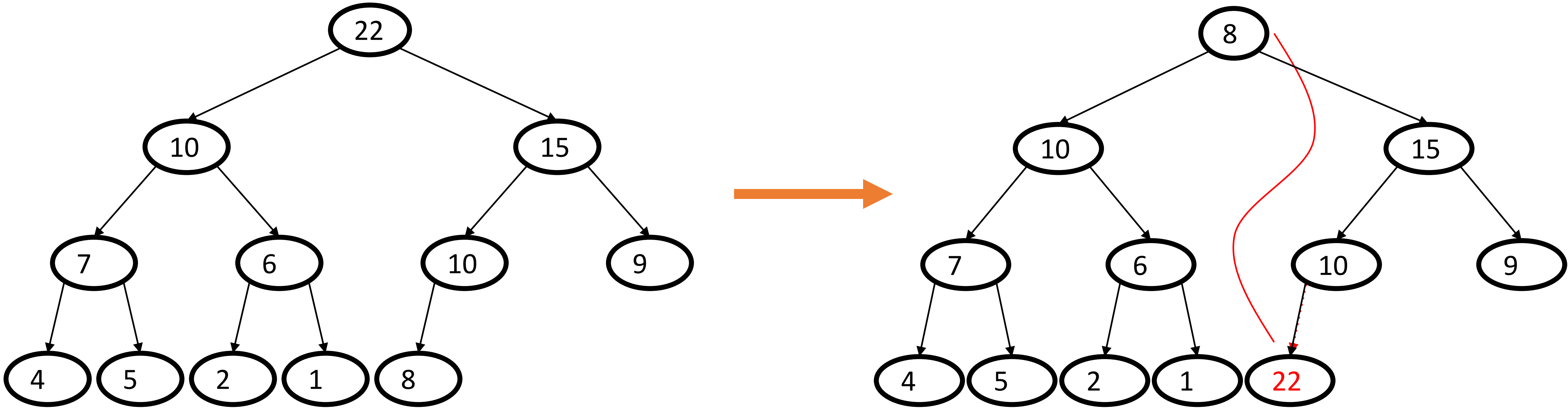
```
Algorithm add()
{     //index of a starting from 0
      j=0 ; a[20];
      Get Choice
     Repeat
     {
      a[j]=elem;
        insert(a,j+1);
        j++;
      } until(choice=='n');
}
```

```
Algorithm Insert(a,n)
{
      i=n;  elem=a[n-1];
      if(i!=1){
           while((i>0)&&(a[(i/2)-1]<elem){
                    a[i-1]=a[(i/2)-1];
                    i=(i/2);
           }
           a[i-1]=elem;
           return true;
      }
}
```



DeleteMax

`pqueue.deleteMax()`

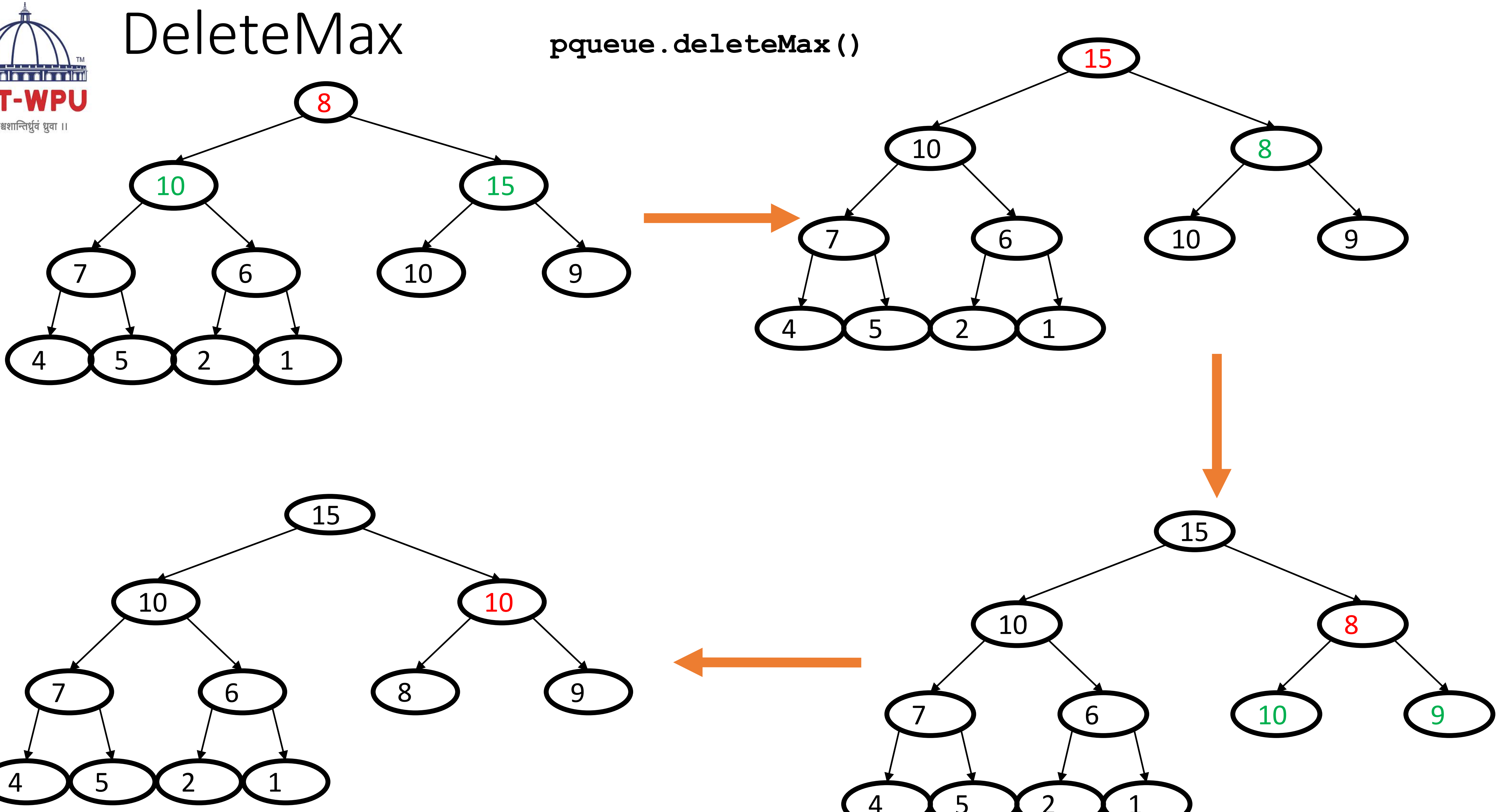


MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

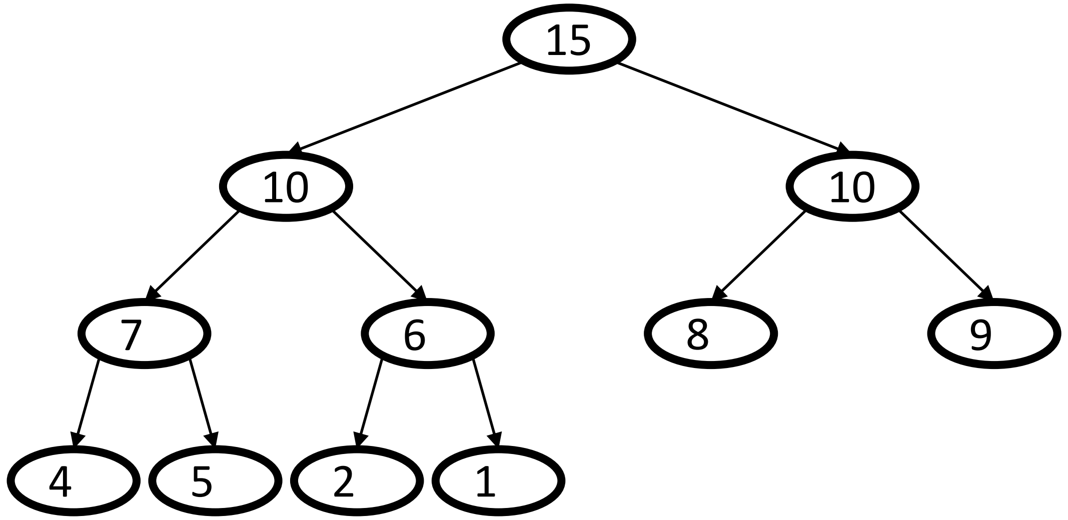
DeleteMax

```
pqueue.deleteMax()
```



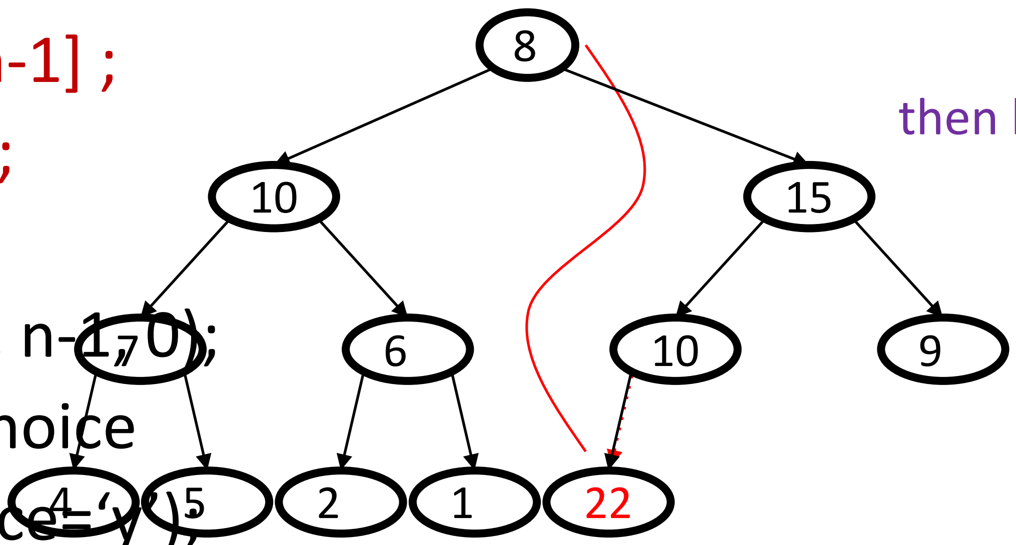
DeleteMax (Final)

`pqueue.deleteMax()`



Delete (Max) element Algorithm(index starting from 0)

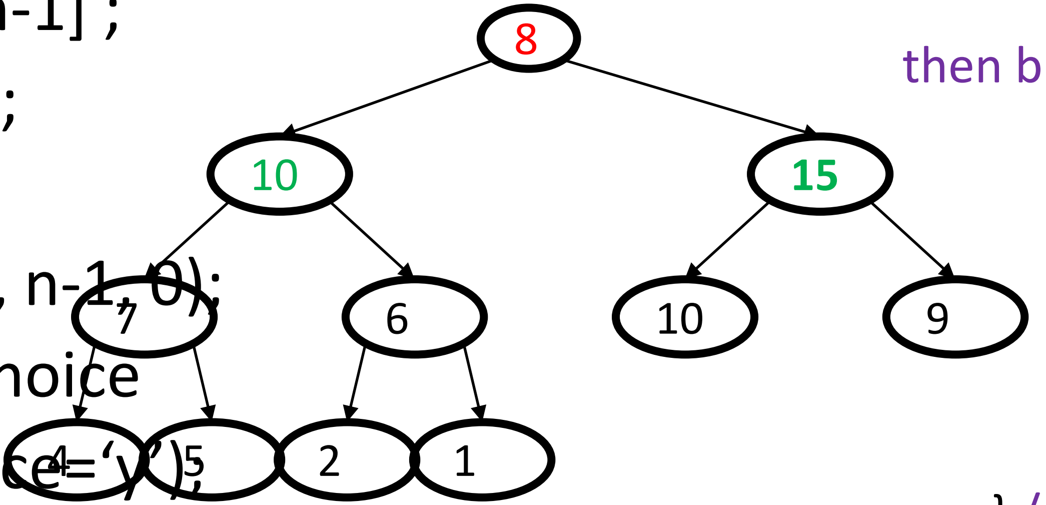
```
Algorithm  Deleteheap(a,n) //n are the no.
 of elements in array
{
      // Interchange the maximum with the
 element at the end of array
 repeat{
      t=a[0];
      a[0]=a[n-1] ;
      a[n-1]=t;
      n --;
     Adjust(a, n-1,0);
     accept choice
   }until(choice='y');
}
```



```
Algorithm Adjust(a,n,i)
{
      while(2*i+1<=n) do
      {
            j=2*i+1;       //index of left child
// compare left and right child and let j be the
 larger child
       if((j+1 <= n) and (a[j+1] > a[j]))
                  j=j+1
            If a[i] >= a[j])
               then break;  //if parent > children
 then break
            else
            {
               //swap a[i] with a[j]
              temp=a[i]; a[i]=a[j];  a[j]=temp;
               i=j;
            }
      } //end of while
}
```

Delete (Max) element Algorithm(index starting from 0)

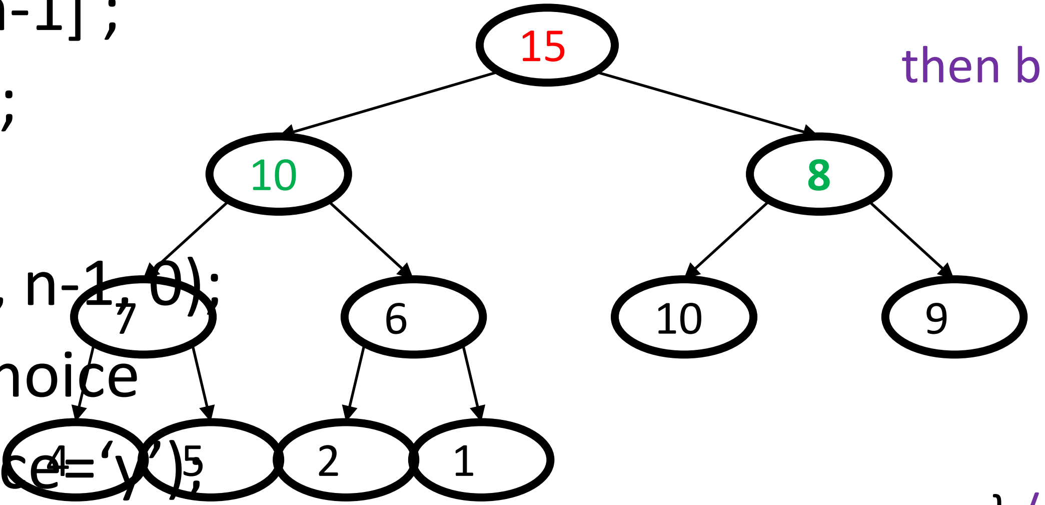
```
Algorithm  Deleteheap(a,n) //n are the no. of elements in array
{
        // Interchange the maximum with the element at the end of array
 repeat{
        t=a[0];
        a[0]=a[n-1] ;
        a[n-1]=t;
        n --;
       Adjust(a, n-1, 0);
       accept choice
    }until(choice='y');
}
```



```
Algorithm Adjust(a,n,i)
{
        while(2*i+1<=n) do
        {
                j=2*i+1;        //index of left child
// compare left and right child and let j be the larger child
                if((j+1 <= n) and (a[j+1] > a[j]))
                        j=j+1
                If a[i] >= a[j])
                   then break;  //if parent > children then break
                else
                {
                    //swap a[i] with a[j]
                    temp=a[i]; a[i]=a[j];  a[j]=temp;
                     i=j;
                }
        } //end of while
}
```

Delete (Max) element Algorithm(index starting from 0)

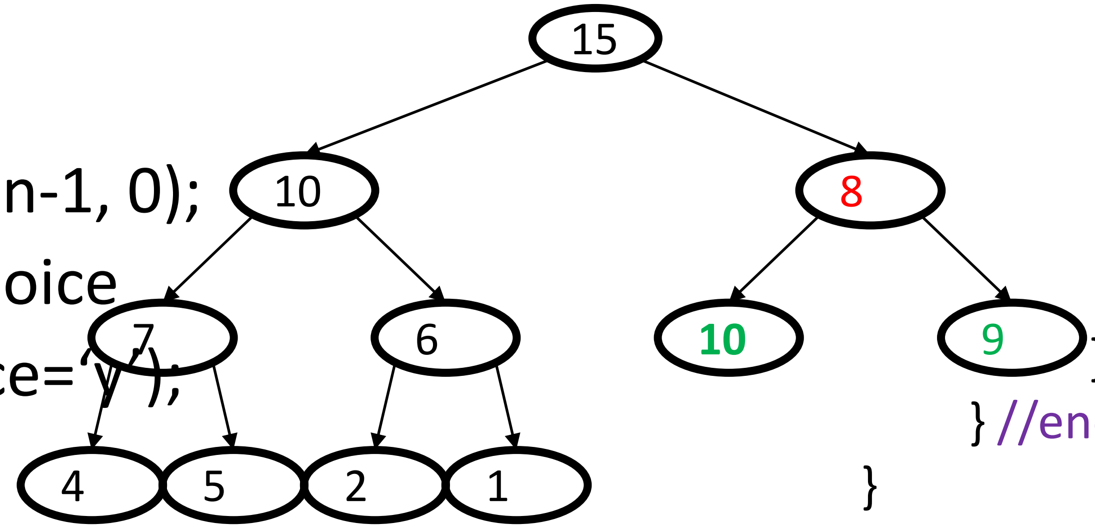
```
Algorithm  Deleteheap(a,n) //n are the no.
 of elements in array
{
       // Interchange the maximum with the
 element at the end of array
 repeat{
        t=a[0];
       a[0]=a[n-1] ;
       a[n-1]=t;
       n --;
      Adjust(a, n-1, 0);
      accept choice
    }until(choice='y');
}
```



```
Algorithm Adjust(a,n,i)
{
      while(2*i+1<=n) do
      {
            j=2*i+1;       //index of left child
// compare left and right child and let j be the
 larger child
         if((j+1 <= n) and (a[j+1] > a[j]))
                        j=j+1
            If a[i] >= a[j])
                then break;  //if parent > children
 then break
            else
            {
               //swap a[i] with a[j]
               temp=a[i]; a[i]=a[j];  a[j]=temp;
                i=j;
            }
      } //end of while
}
```

Delete (Max) element Algorithm(index starting from 0)

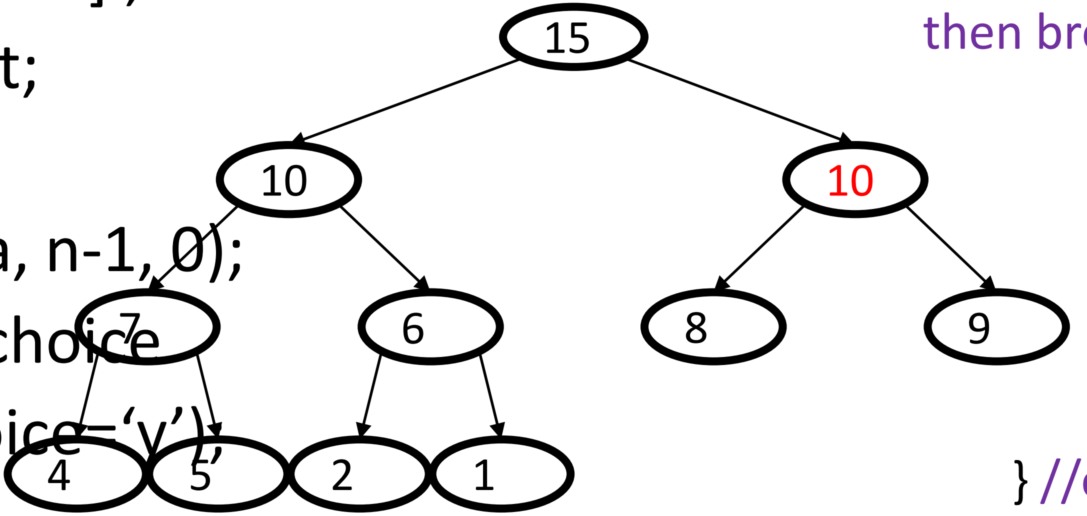
```
Algorithm  Deleteheap(a,n) //n are the no.
 of elements in array
{
        // Interchange the maximum with the
 element at the end of array
 repeat{
        t=a[0];
        a[0]=a[n-1] ;
        a[n-1]=t;
        n --;
       Adjust(a, n-1, 0);
       accept choice
    }until(choice='y');
}
```



```
Algorithm Adjust(a,n,i)
{
        while(2*i+1<=n) do
        {
                j=2*i+1;        //index of left child
// compare left and right child and let j be the
 larger child
                if((j+1 <= n) and (a[j+1] > a[j]))
                        j=j+1
                If a[i] >= a[j])
                   then break;  //if parent > children
 then break
                else
                {
                     //swap a[i] with a[j]
                     temp=a[i]; a[i]=a[j];  a[j]=temp;
                      i=j;
                }
        } //end of while
}
```

Delete (Max) element Algorithm(index starting from 0)

```
Algorithm  Deleteheap(a,n) //n are the no.
 of elements in array
{
        // Interchange the maximum with the
 element at the end of array
 repeat{
         t=a[0];
        a[0]=a[n-1] ;
        a[n-1]=t;
        n --;
       Adjust(a, n-1, 0);
       accept choice
    }until(choice='y');
}
```



```
Algorithm Adjust(a,n,i)
{
      while(2*i+1<=n) do
      {
            j=2*i+1;        //index of left child
// compare left and right child and let j be the
 larger child
            if((j+1 <= n) and (a[j+1] > a[j]))
                        j=j+1
            If a[i] >= a[j])
               then break;  //if parent > children
 then break
            else
            {
                //swap a[i] with a[j]
                temp=a[i]; a[i]=a[j];  a[j]=temp;
                 i=j;
            }
      } //end of while
}
```

Sorting

- Other than as a priority queue, the heap has one other important usage: heap sort

- Heap sort is one of the fastest sorting algorithms, achieving speed as that of the quicksort and merge sort algorithms

- The advantages of heap sort are that it does not use recursion and it is efficient for any data order

- There is no worse-case scenario in the case of heap sort

Heap Sort

- ❖ Steps for heap sort (ascending order) are as follows:

1. Build the heap tree(for given array as it may not be in heap tree form)
2. Start Delete Heap operations, storing each deleted element at the end of the heap array
3. After performing step 2, again adjust the heap tree (ReHeapDown)
4. Repeat step 2 and 3 for n-1 times to sort the complete array

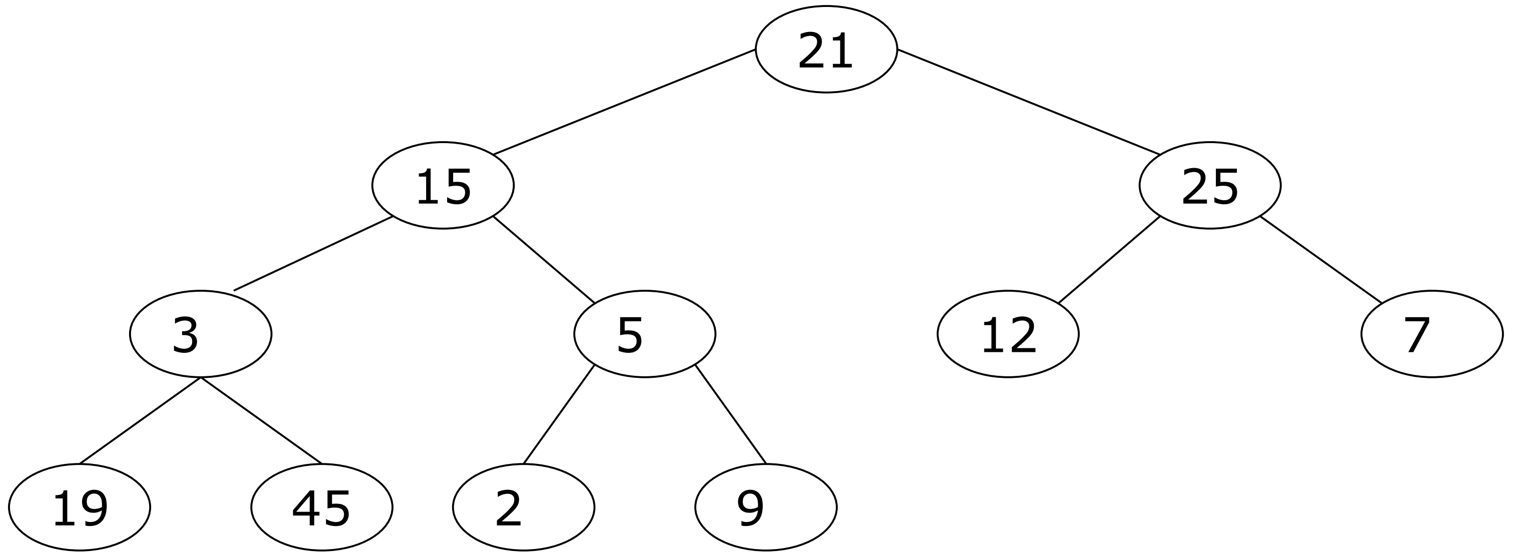
Sample Run

- Start with unordered array of data

Array representation:

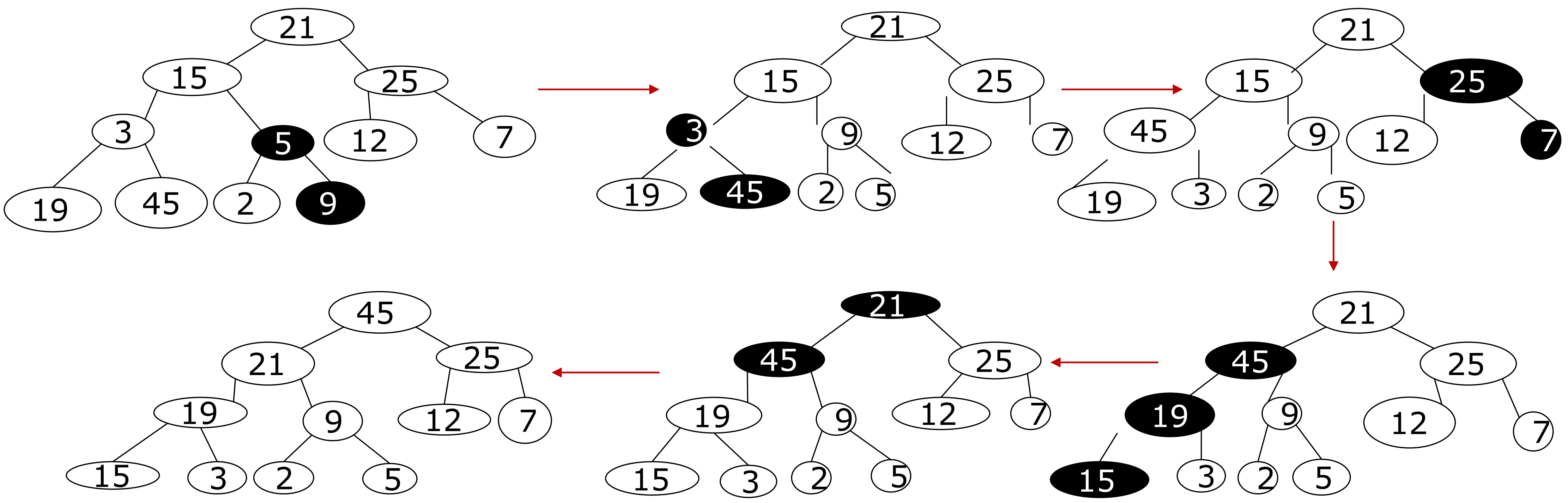
21	15	25	3	5	12	7	19	45	2	9

Binary tree representation:

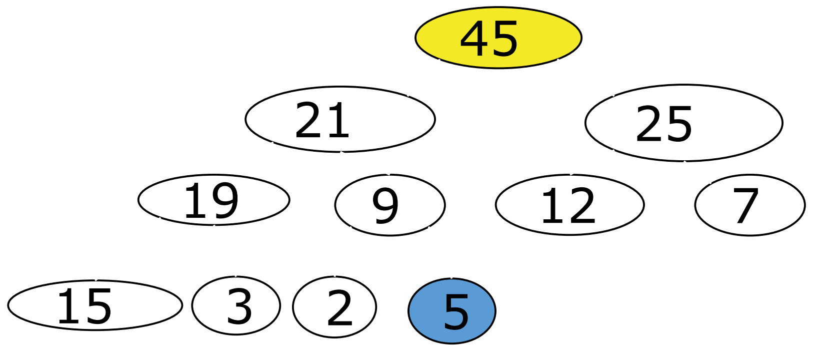


Sample Run

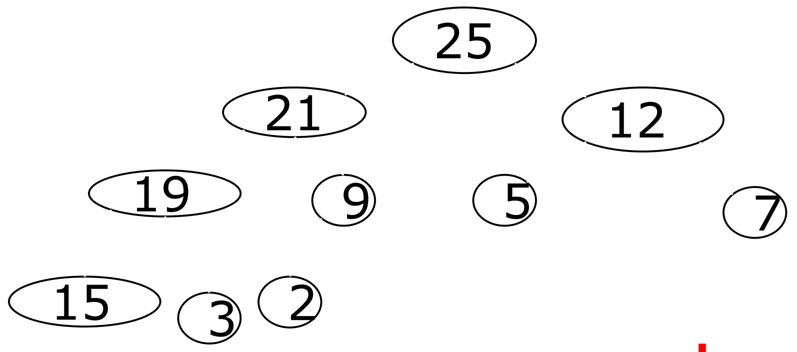
- Heapify the binary tree –(from (n/2)-1 to 0)



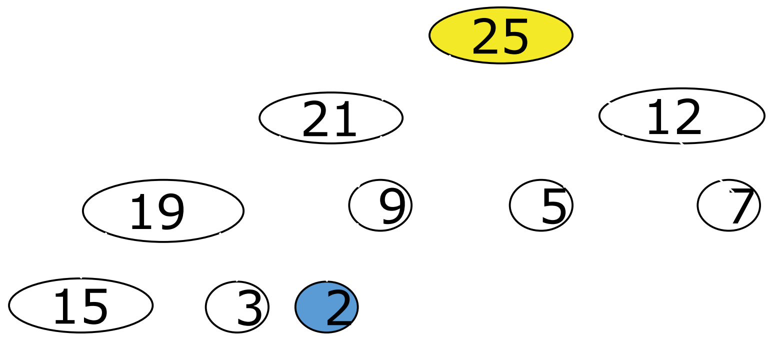
Step 2 – perform n – 1 deleteMax(es) and replace last element in heap with first, then re-heapify. Place deleted element in the last node’s position.



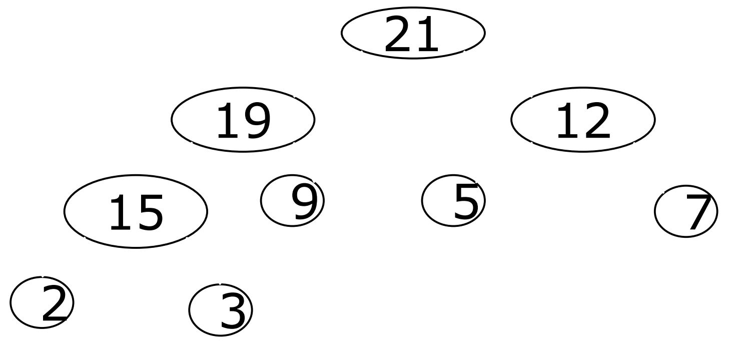
45	21	25	19	9	12	7	15	3	2	5



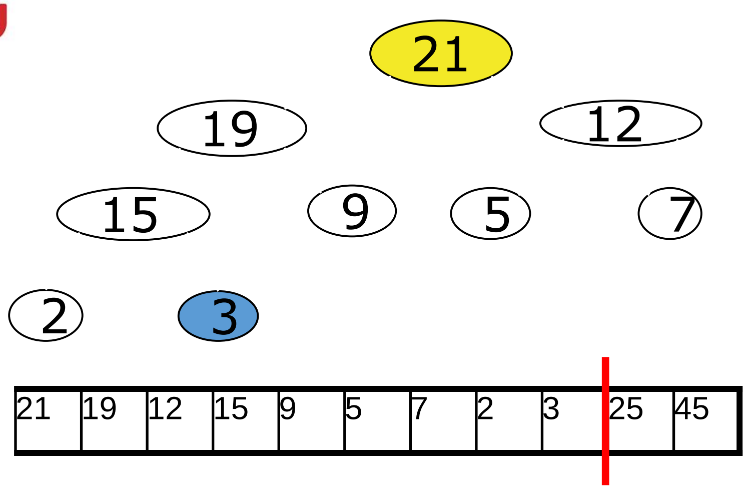
25	21	12	19	9	5	7	15	3	2	45

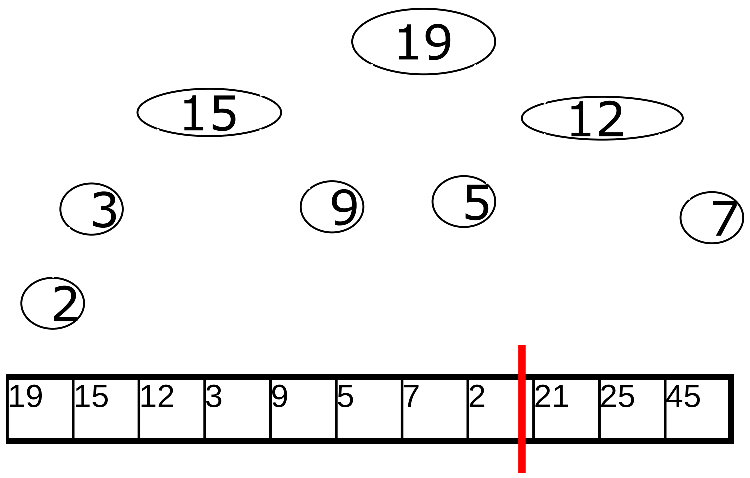


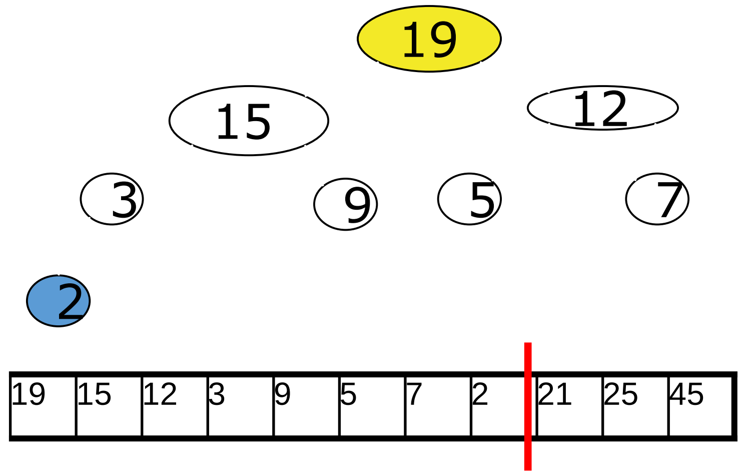
25	21	12	19	9	5	7	15	3	2	45

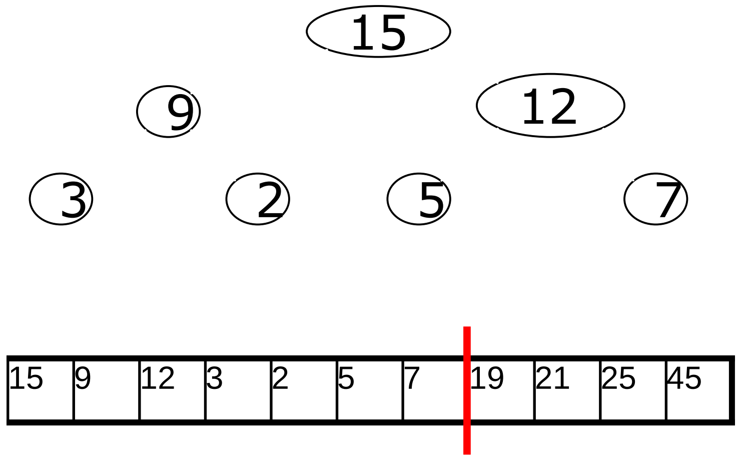


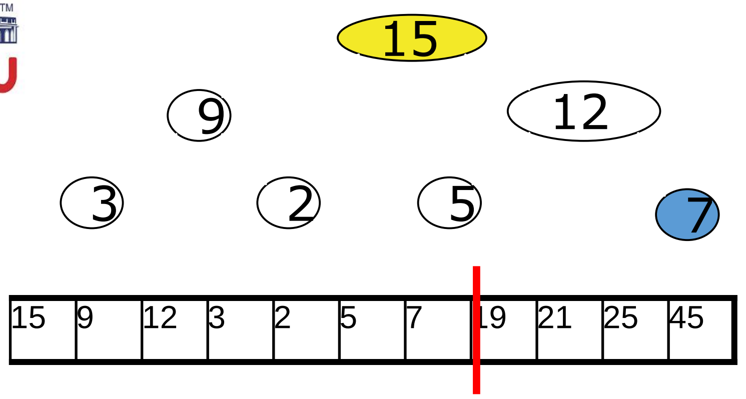
21	19	12	15	9	5	7	2	3	25	45

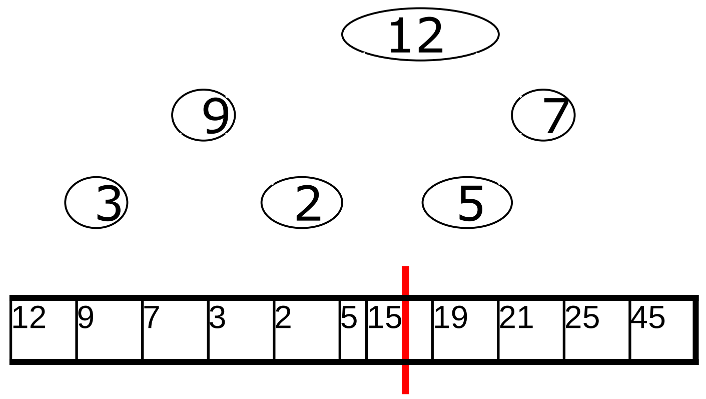

21
19
12
15
9
5
7
2
3
21 19 12 15 9 5 7 2 3 25 45

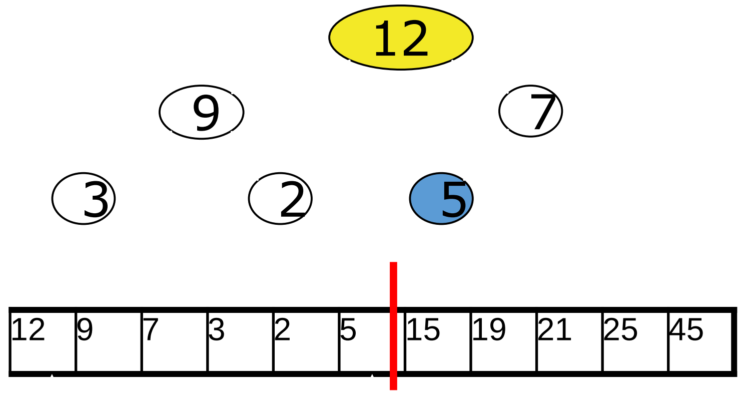

19
15
12
3
9
5
7
2
19 15 12 3 9 5 7 2 21 25 45

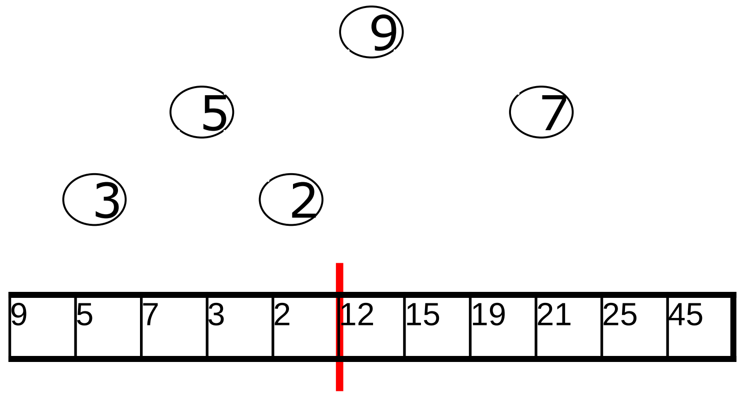

19
15
12
3
9
5
7
2
19 15 12 3 9 5 7 2 21 25 45


15
9
12
3
2
5
7
15 9 12 3 2 5 7 19 21 25 45

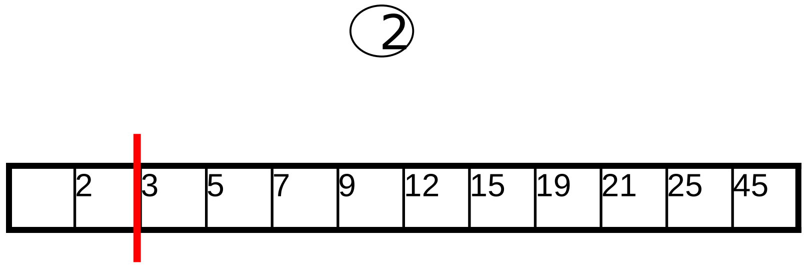








This continues till only one is left



and finally

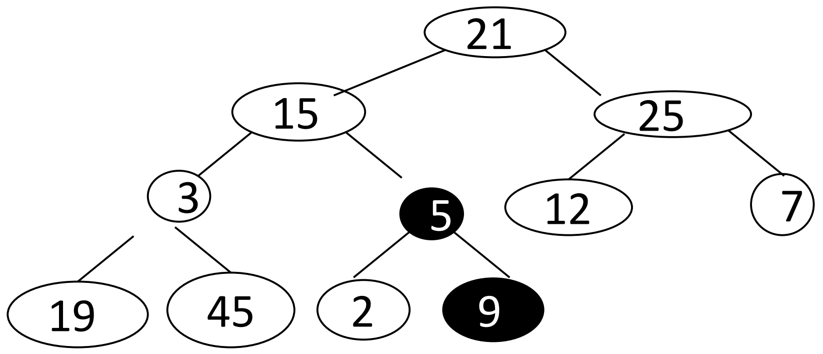
Heap sort Algorithm (index starting from 0)

```
Algorithm HeapSort(a,n)
//here n is the total no. of elements in the array
{

    for i = (n/2)-1  to 0  step -1 do
        Adjust(a,n-1,i)
   // Interchange the new maximum with
the element at the end of array
   while(n>0)
   {
      t=a[0];
      a[0]=a[n-1] ;
      a[n-1]=t;
      n --;
     Adjust(a, n-1, 0);
   }
}
```

```
Algorithm Adjust(a,n,i)
{
    while(2*i+1<=n) do
    {
        j=2*i+1;       //index of left child
        if((j+1 <= n) and (a[j+1] > a[j]))
                j=j+1;   // compare left and right child and let j be the larger child
        If a[i] >= a[j])
          then break;  //if root >children then break
        else
        {
            //swap a[i] with a[j]
            temp=a[i]; a[i]=a[j];  a[j]=temp;
             i=j;
        }
    } //end of while
}
```

```
Algorithm Adjust(a,n,i)
{
    while(2*i+1<=n) do
    {
        j=2*i+1;        //index of left child
        if((j+1 <= n) and (a[j+1] > a[j]))
                j=j+1;    // compare left and right child and let j be the larger child
        If a[i] >= a[j])
            then break;  //if root >children then break
        else
        {
            //swap a[i] with a[j]
            temp=a[i]; a[i]=a[j];  a[j]=temp;
            i=j;
        }
    } //end of while
}
```



Heap Applications

- ❖ Selection algorithm
- ❖ Scheduling and prioritizing (priority queue)
- ❖ Sorting

Selection Problem

- For the solution to the problem of determining the kth element, we can create the heap and delete k − 1 elements from it, leaving the desired element at the root.
- So the selection of the kth element will be very easy as it is the root of the heap
- For this, we can easily implement the algorithm of the selection problem using heap creation and heap deletion operations
- This problem can also be solved in O(nlogn) time using priority queues

Scheduling and prioritizing (priority queue)

- The heap is usually defined so that only the largest element (that is, the root) is removed at a time.

- This makes the heap useful for scheduling and prioritizing

- In fact, one of the two main uses of the heap is as a prioriy queue, which helps systems decide what to do next

Applications of priority queues where heaps are implemented

- ❖ CPU scheduling
- ❖ I/O scheduling
- ❖ Process scheduling.