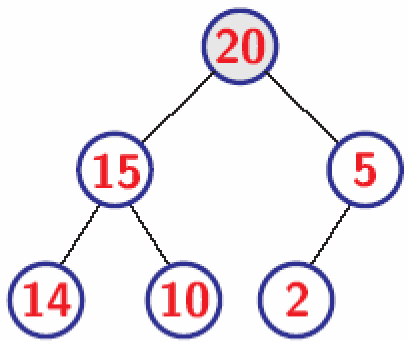
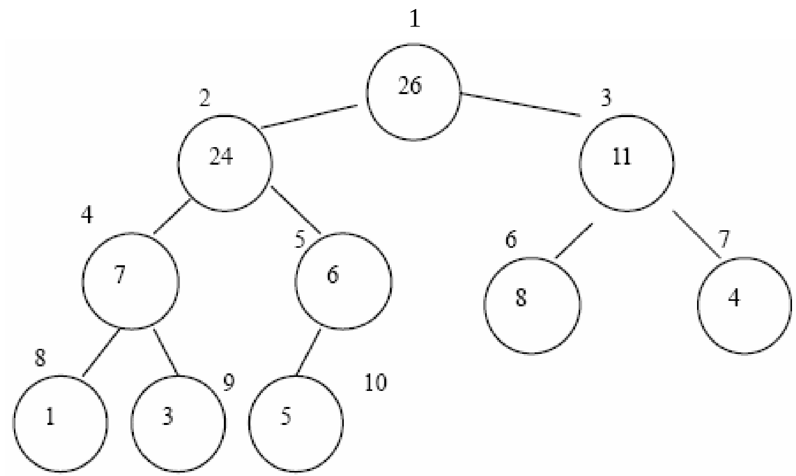
Binary Heap ADT

Complete Binary Tree

A binary tree
- which is full up to second last level and
- its last level is filled from left to right

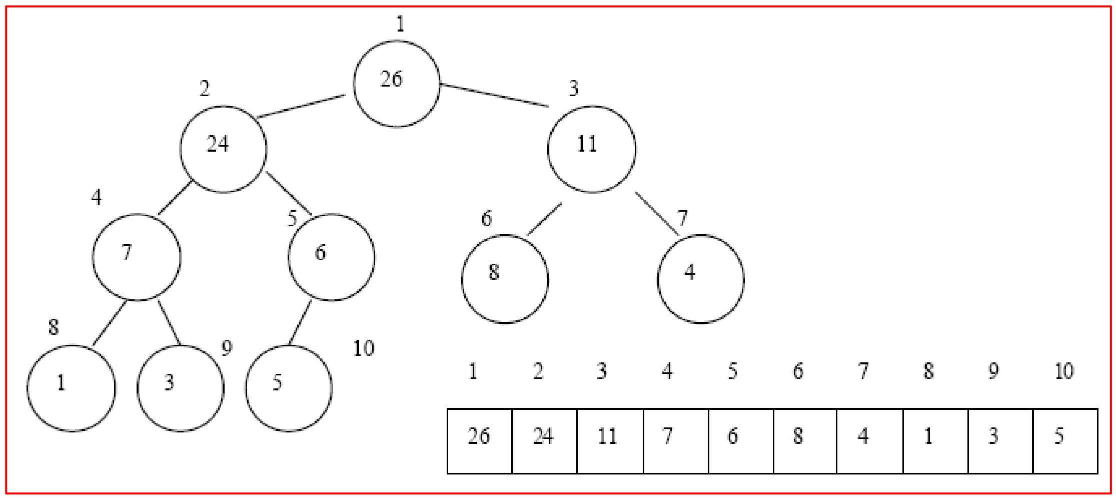
The following binary trees are complete





Complete Binary Tree

❖ It can be stored in an **array** using **level-order traversal**. For any element in array position i, its left child resides in position 2i and its right child resides in position 2i+1, and its parent resides in position $\lfloor i/2 \rfloor$



Advantages

- Its height is at most $\lfloor \log n \rfloor$
- Left and right pointers are not needed.
- Allows two directional traversal – root to leaves and leaves to root

Heap Order Property

The sequence of elements

$$r[1], r[2], r[3], \ldots, r[n]$$

is said to satisfy heap order property if

$r[i] < r[2i]$ and $r[i] < r[2i+1]$ (Min Heap Property)

or

$r[i] > r[2i]$ and $r[i] > r[2i+1]$ (Max Heap Property)

Binary Heap or Heap ADT

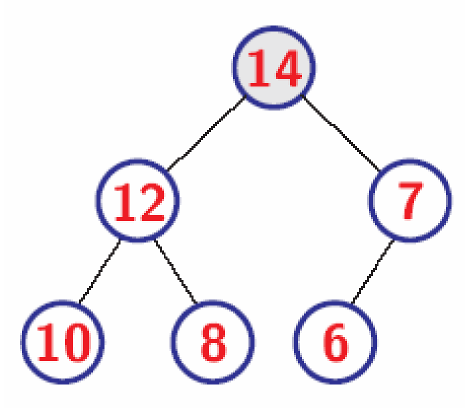
A complete binary tree that satisfies heap property i.e.

- key of each node is less than the keys of its children (Min Heap)

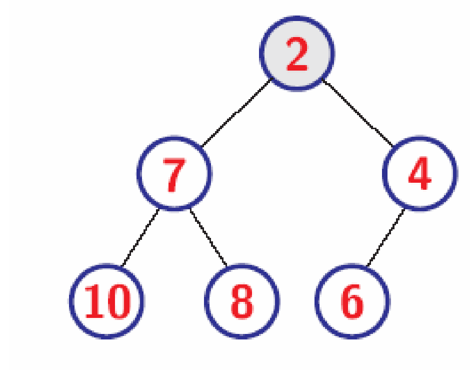
  OR

- key of each node is greater than the keys of its children (Max Heap)

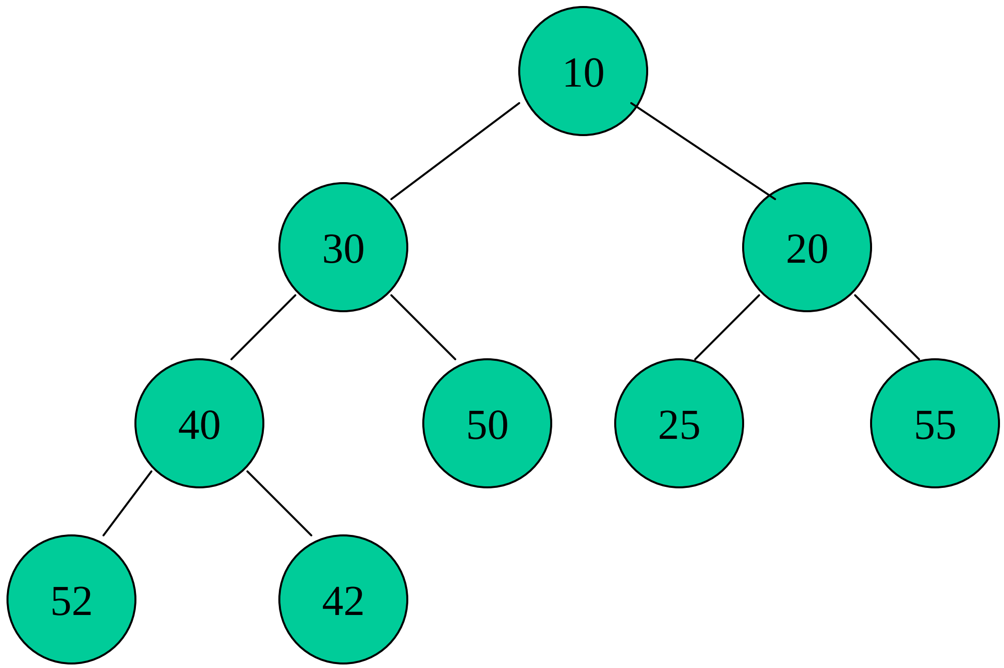
Max Heap



Min Heap

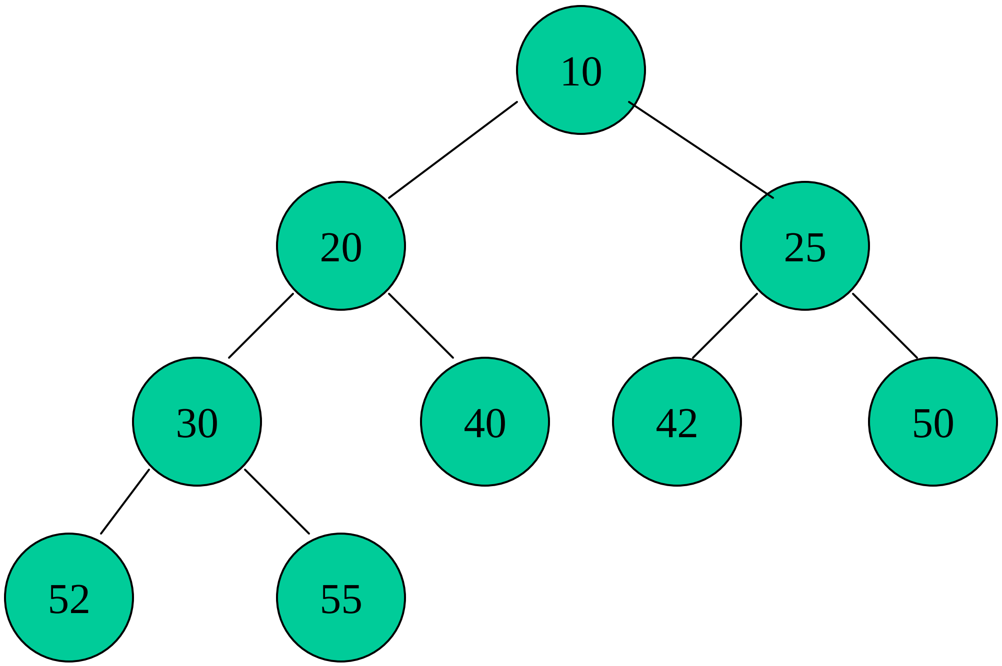


Min Heap



The key of any node is less than those of its left and right children

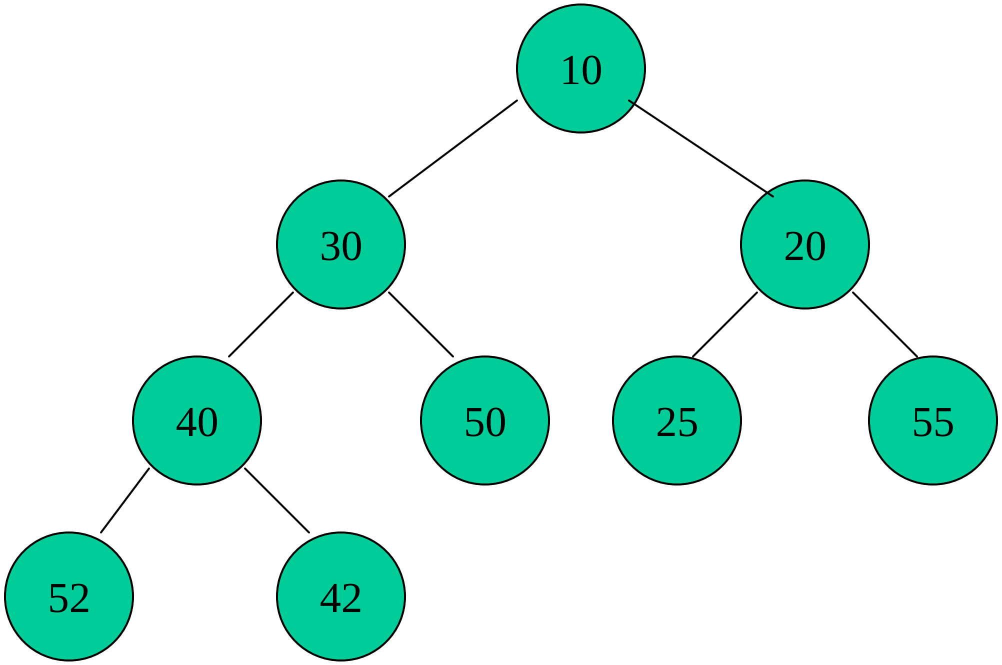
NOTE: For a given set of data, there may be different arrangements that satisfy heap conditions



[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

For the heap shown

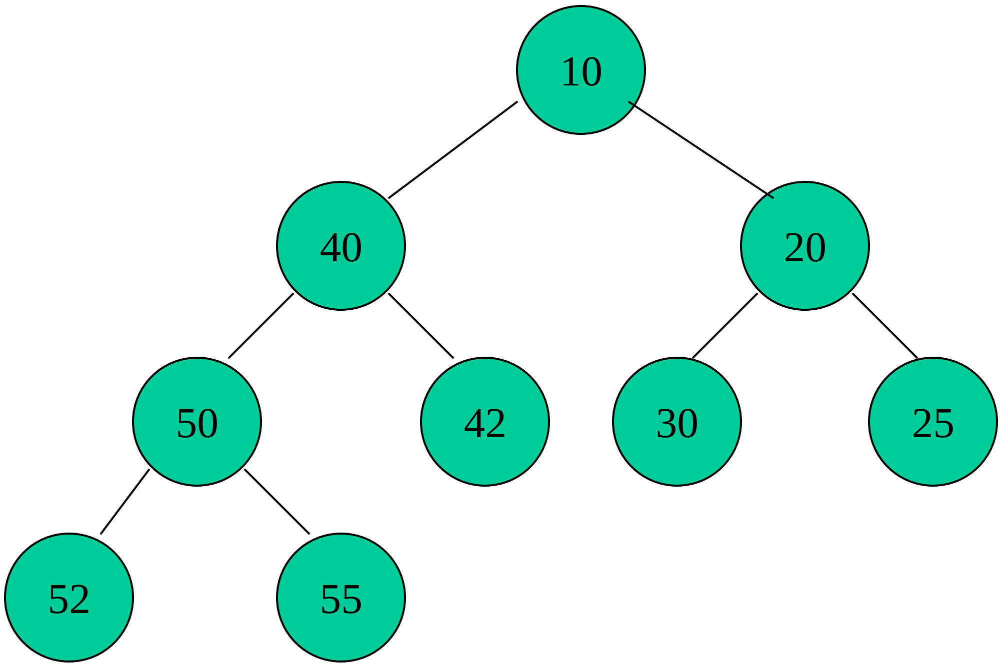
NOTE: For a given set of data, there may be different arrangements that satisfy heap condtions



[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

For the heap shown

NOTE: For a given set of data, there may be different arrangements that satisfy heap condtions



[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

For the heap shown

Why Heap?

Two important uses of heaps are

- Efficient implementation of priority queues
- Sorting $\Rightarrow$ Heapsort.

Representation of Heap ADT

Since heap is a complete binary tree, so it can be represented using Array H

Specification of Heap ADT

Elements: Any data type

Structure: A complete binary tree such that if N is any node in the tree then it is smaller (greater) than its children

Note: A node N is larger than the node M if key value of N is larger than that of M and vice versa.

Domain: Number of elements is bounded

Operations:

Operation	**Specification**
void **siftUp** (int n)	**Precondition**: Elements H[1], H[2], H[3], …, H[n-1] satisfy the heap conditions **Process:** Elements H[1], H[2], H[3], …, H[n] satisfy the heap conditions
void **siftDown** (int m, int n)	**Precondition**: Elements H[m+1], H[m+2], H[m+3], …, H[n] satisfy the heap conditions **Process:** Elements H[m], H[m+1], H[m+2], …, H[n] satisfy the heap conditions

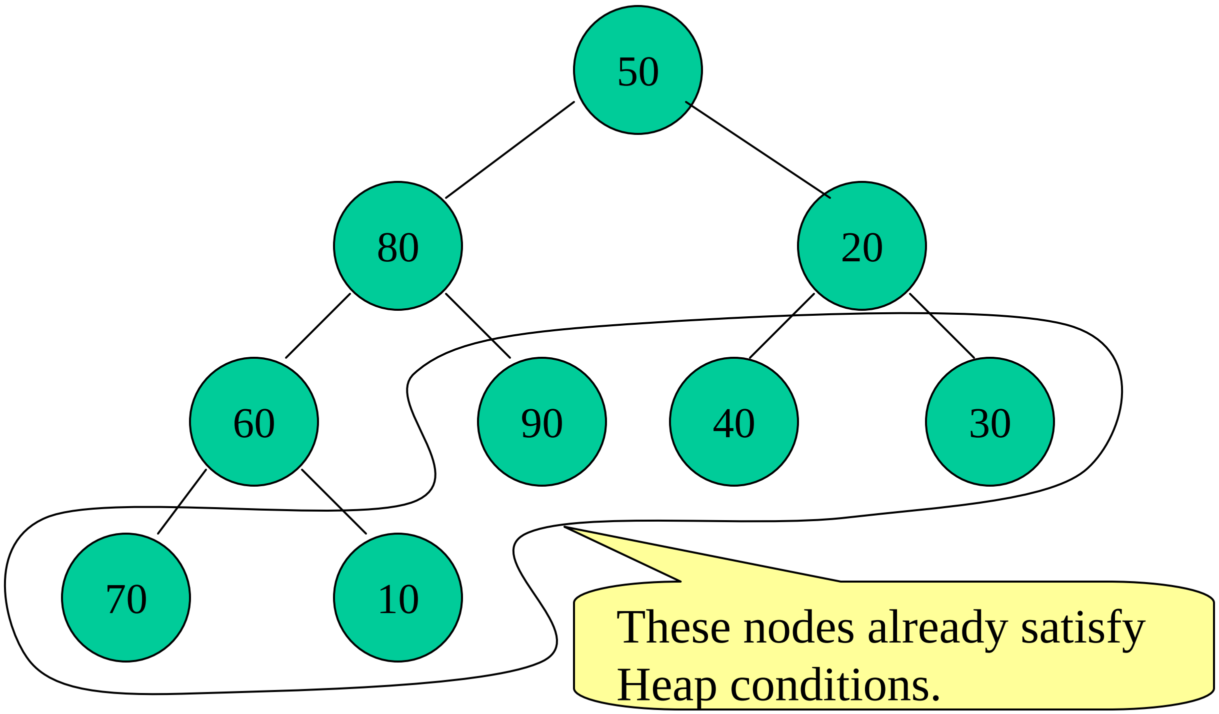
NOTE

- The two operations
  - SiftDown operation.
  - SiftUp operation.

  are used to construct a heap or to convert an arrangement of data stored in array into a heap
- **siftUp** operation is usually used when root node and nodes close to root node satisfy heap order property
- **siftDown** operation is usually used when leaf nodes and nodes close to leaf nodes satisfy heap order property

siftDown Operation

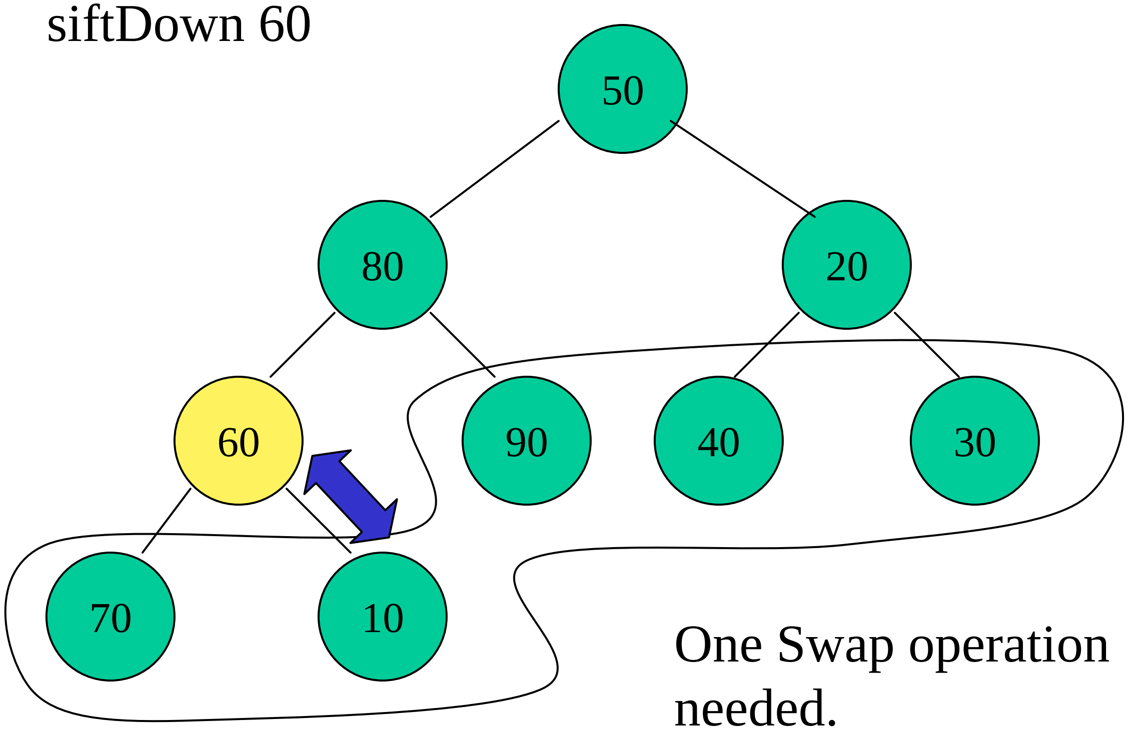
- The given arrangement of data is not a Heap
- To convert it into a min heap, we can employ **siftDown** operation or **siftUp** operation
- First, we apply **siftDown** Operation



[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftDown Operation

siftDown 60

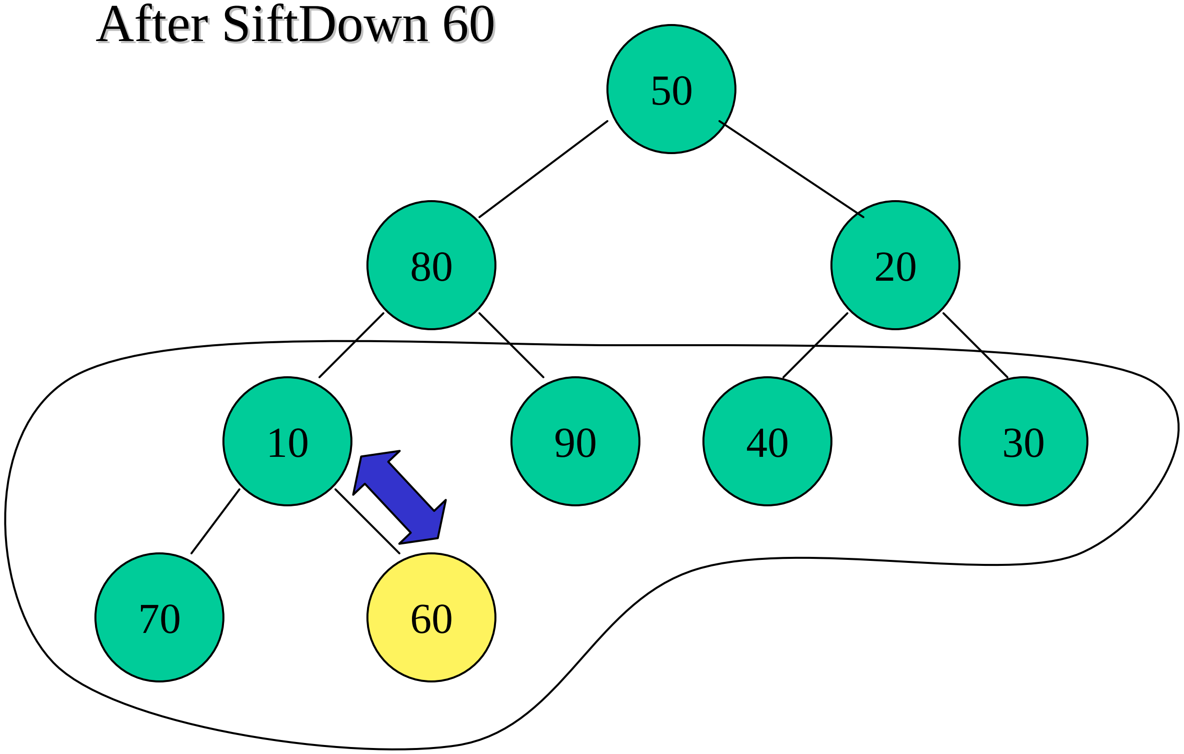


One Swap operation needed.

[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftDown Operation

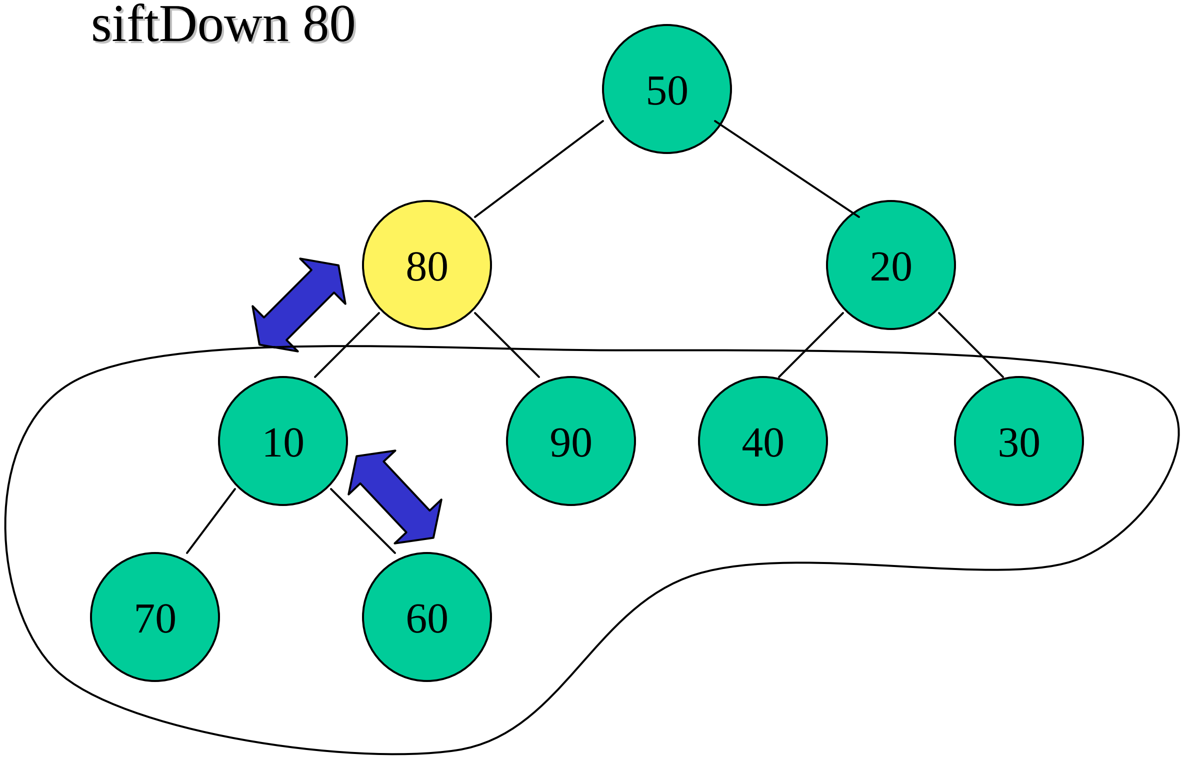
After SiftDown 60



[1]	50
[2]	80
[3]	20
[4]	10
[5]	90
[6]	40
[7]	30
[8]	70
[9]	60

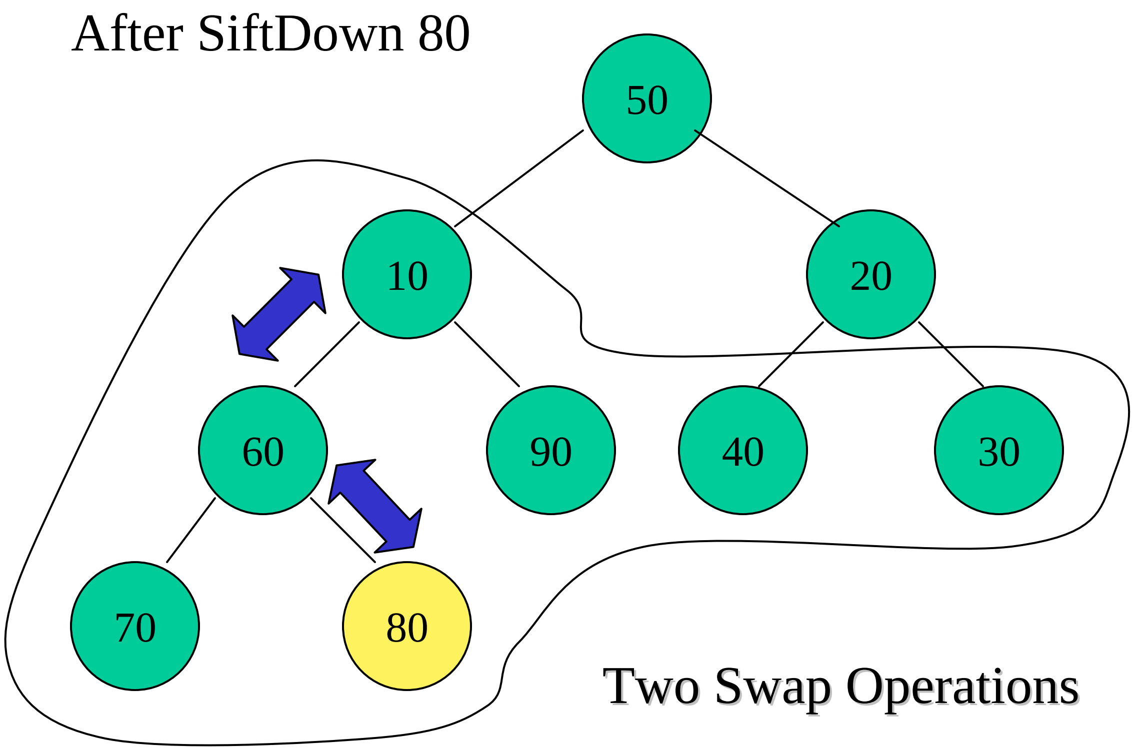
siftDown Operation

siftDown 80



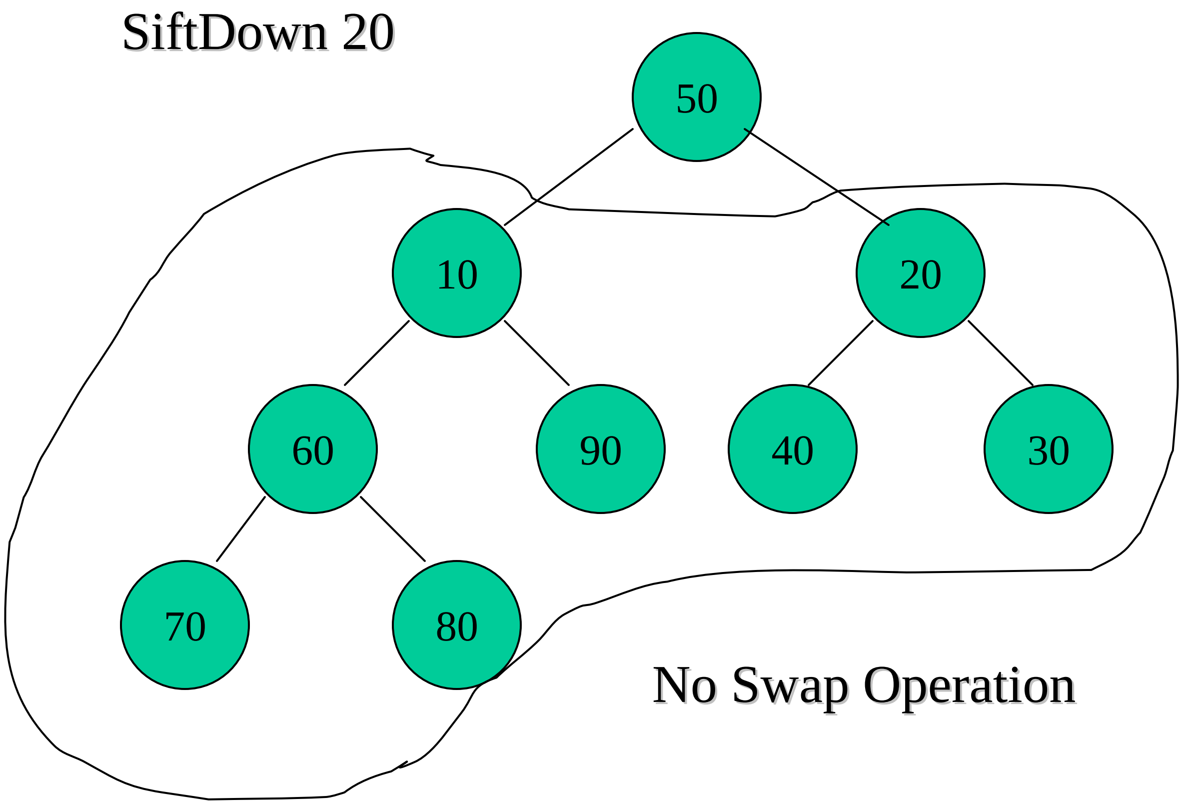
[1]	50
[2]	80
[3]	20
[4]	10
[5]	90
[6]	40
[7]	30
[8]	70
[9]	60

siftDown Operation



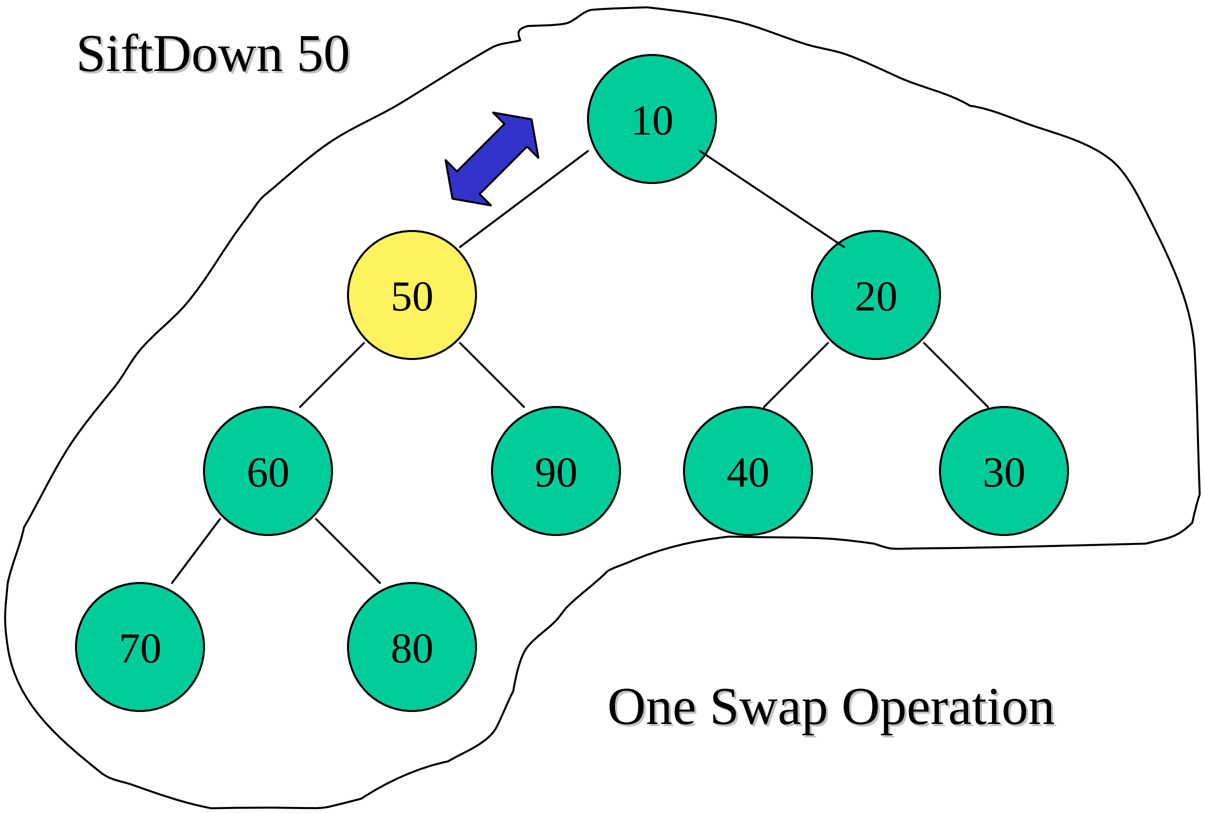
[1]	50
[2]	10
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	80

siftDown Operation



[1]	50
[2]	10
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	80

siftDown Operation

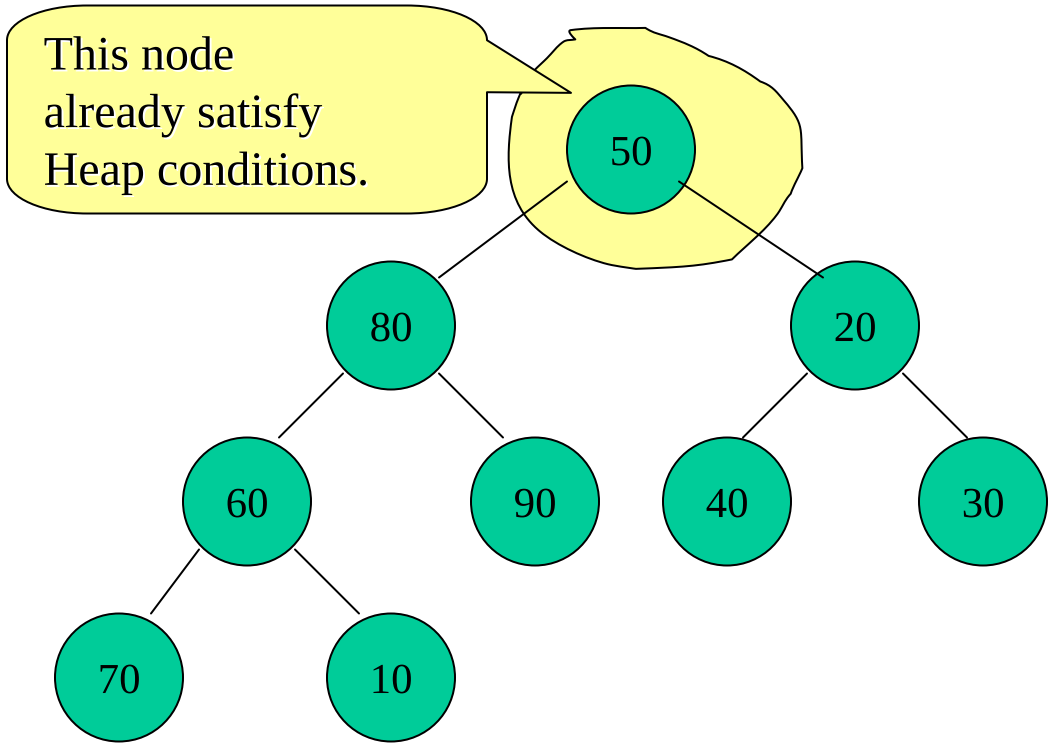


[1]	10
[2]	50
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	80

It is a heap now

siftUp Operation

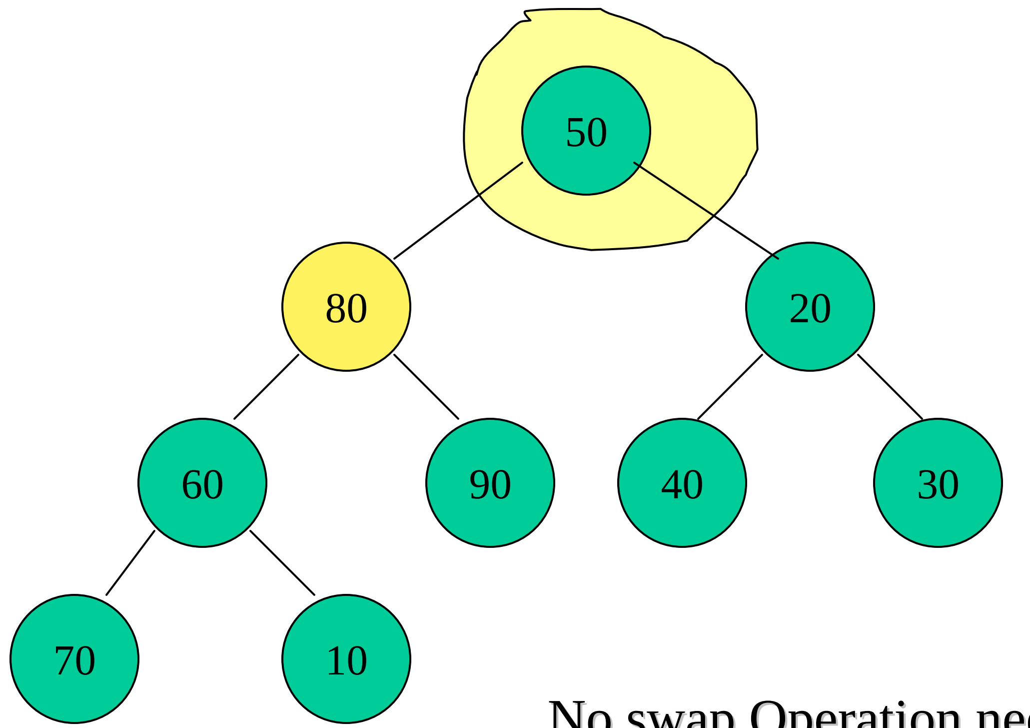
- The following is the same arrangement, which is not a Heap
- Now we employ **siftUp** operation to convert it into a heap



[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 80

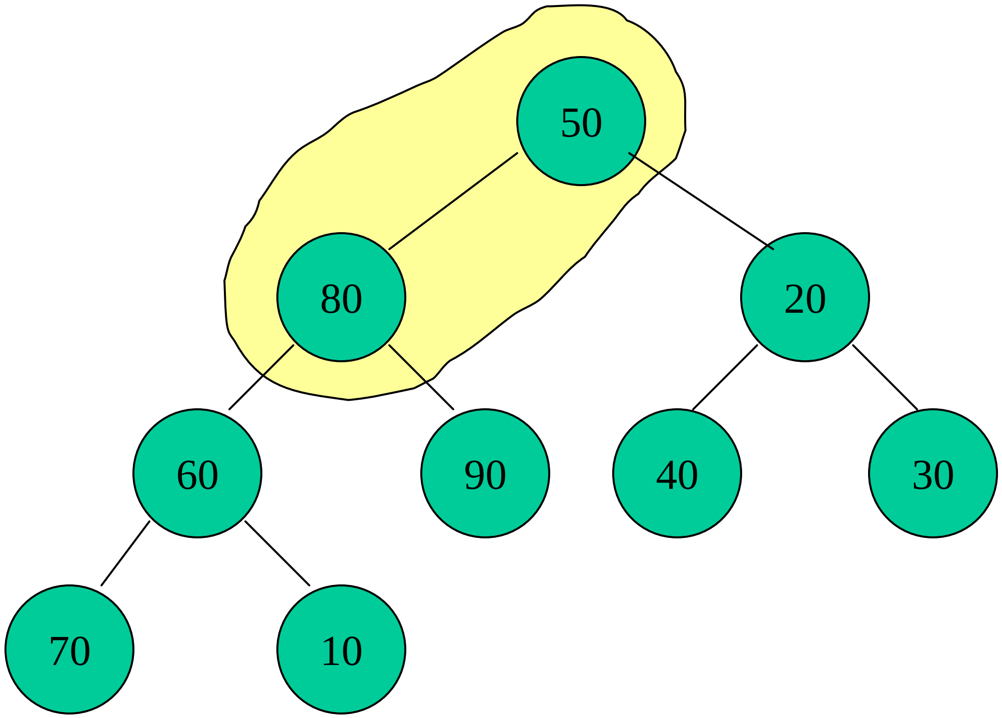


No swap Operation needed

[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

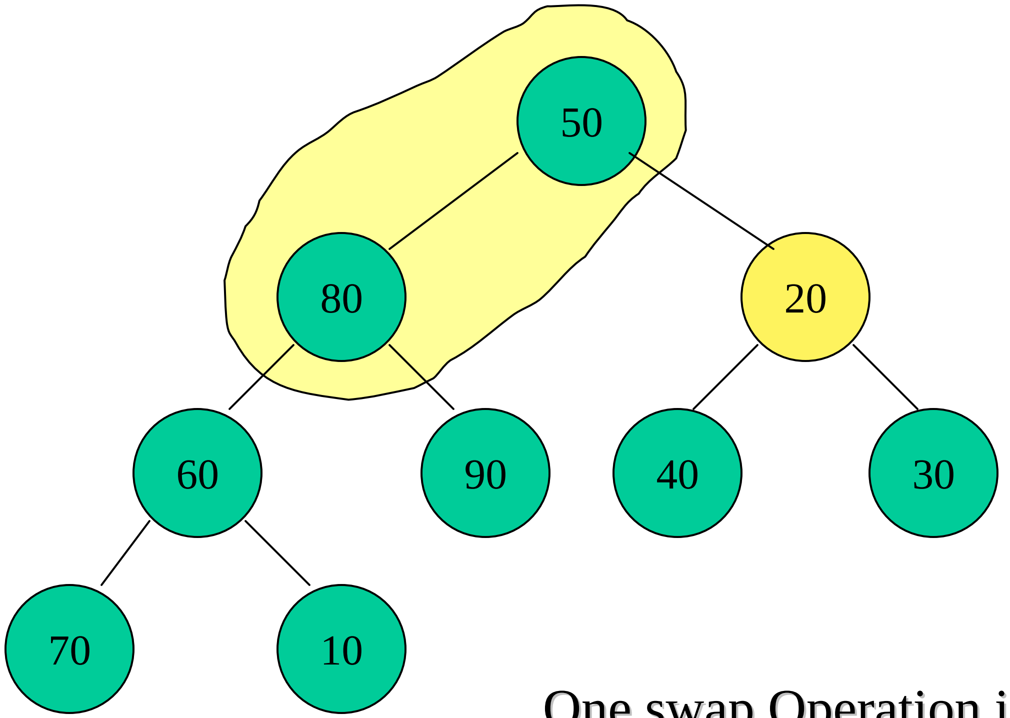
After siftUp 80



[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 20

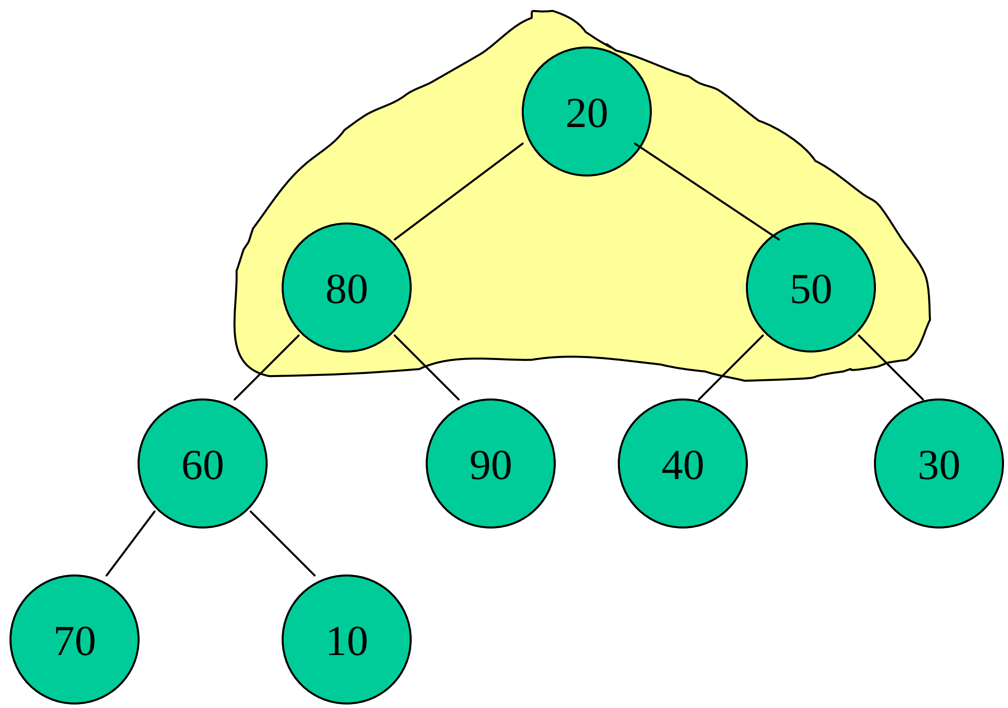


[1]	50
[2]	80
[3]	20
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

One swap Operation is needed

siftUp Operation

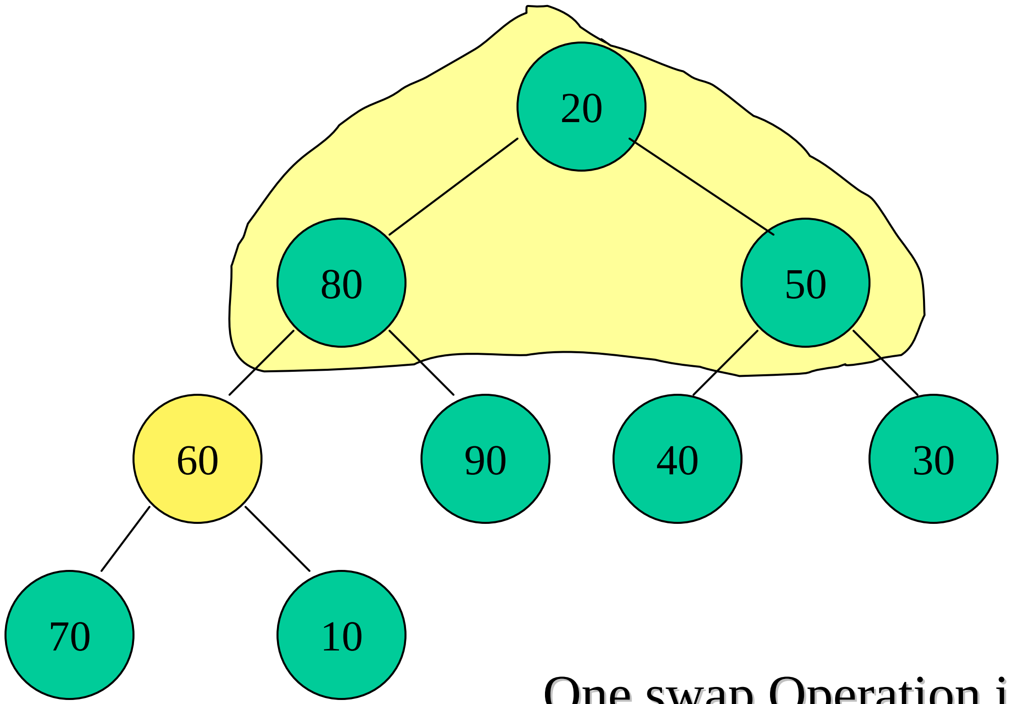
After siftUp 20



[1]	20
[2]	80
[3]	50
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 60

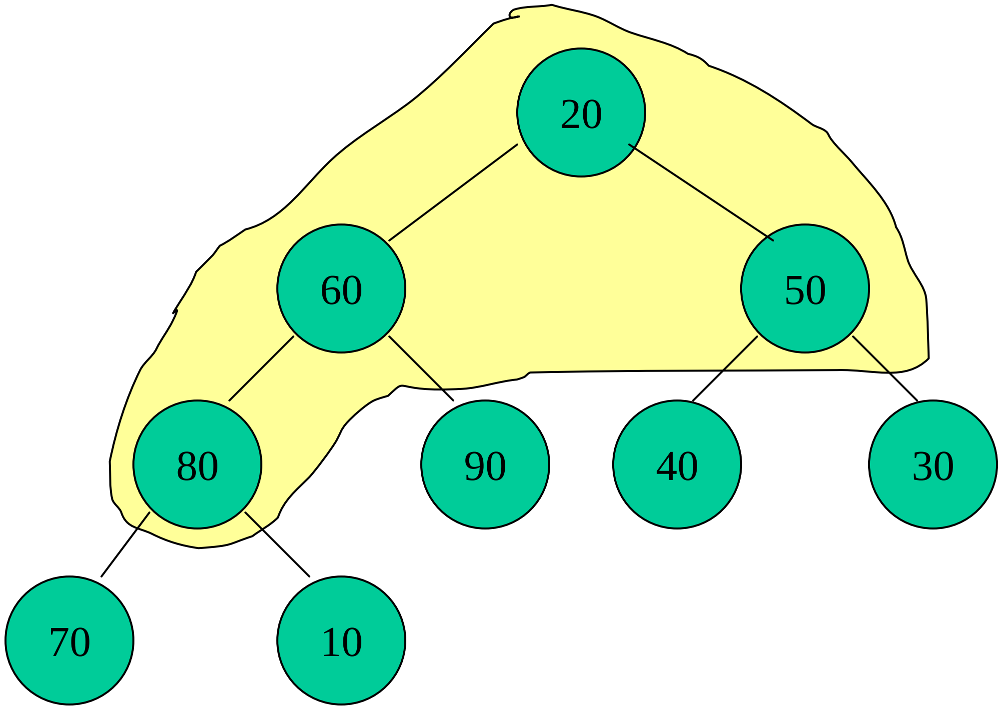


[1]	20
[2]	80
[3]	50
[4]	60
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

One swap Operation is needed

siftUp Operation

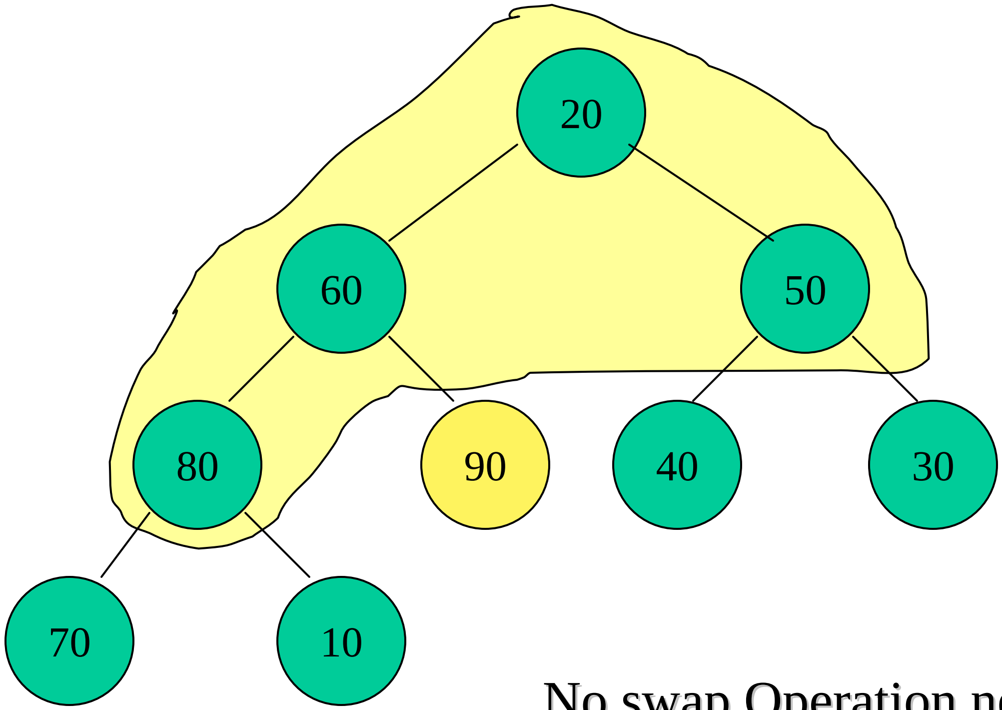
After siftUp 60



[1]	20
[2]	60
[3]	50
[4]	80
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 90

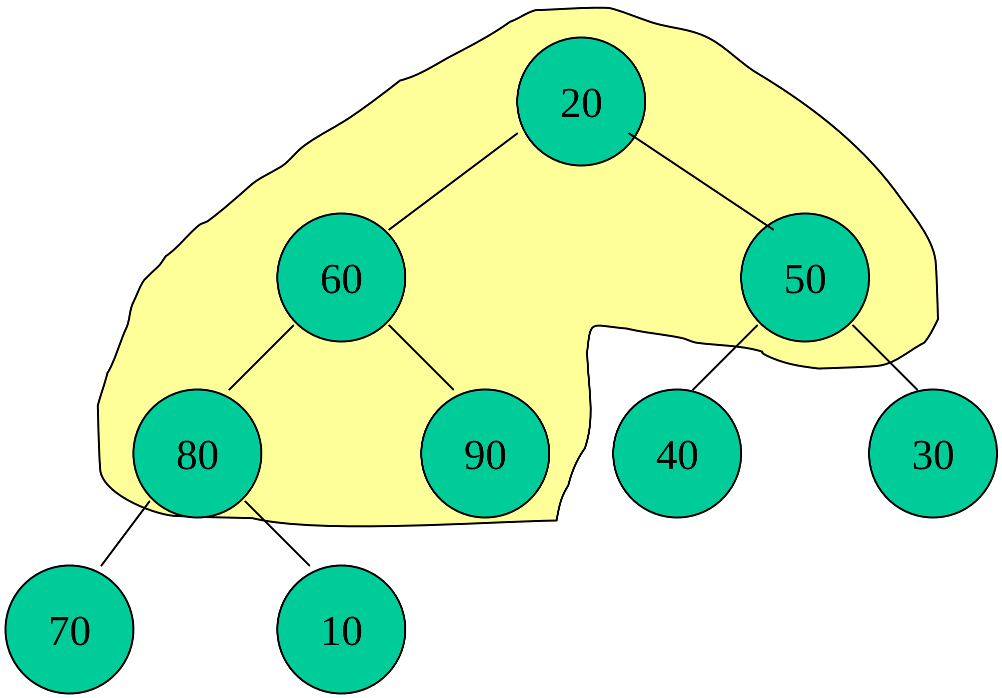


No swap Operation needed

[1]	20
[2]	60
[3]	50
[4]	80
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

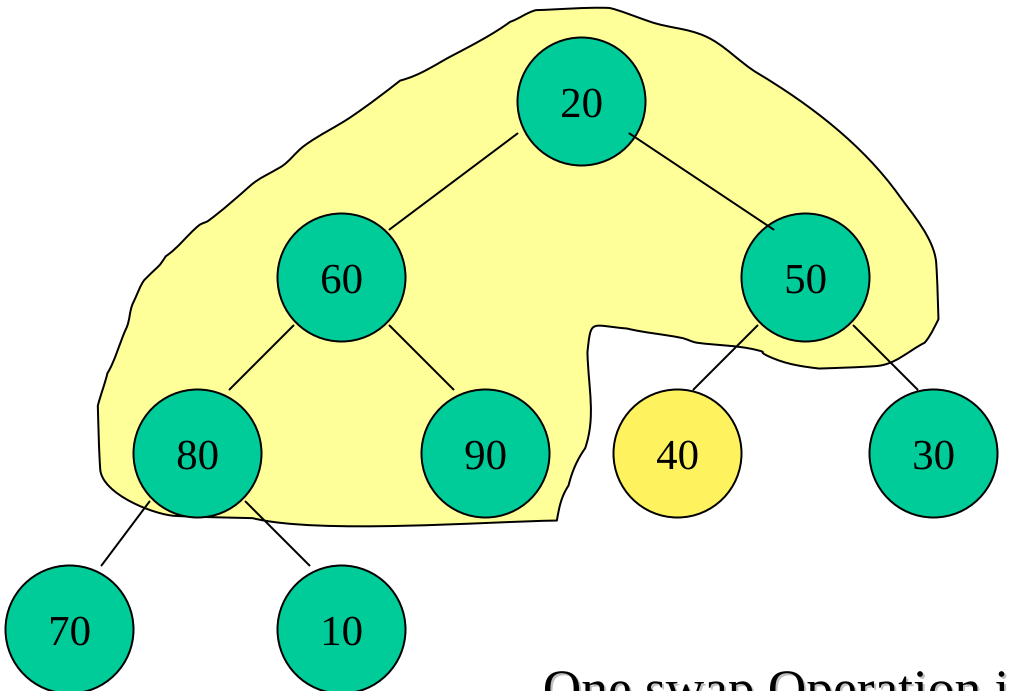
After siftUp 90



[1]	20
[2]	60
[3]	50
[4]	80
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 40

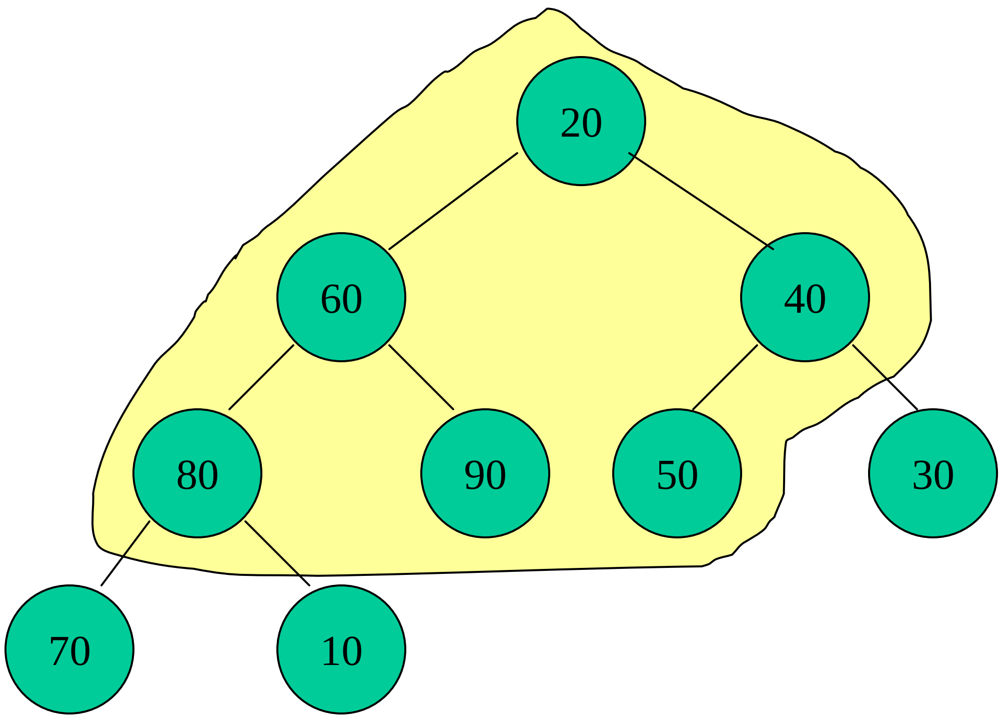


[1]	20
[2]	60
[3]	50
[4]	80
[5]	90
[6]	40
[7]	30
[8]	70
[9]	10

One swap Operation is needed

siftUp Operation

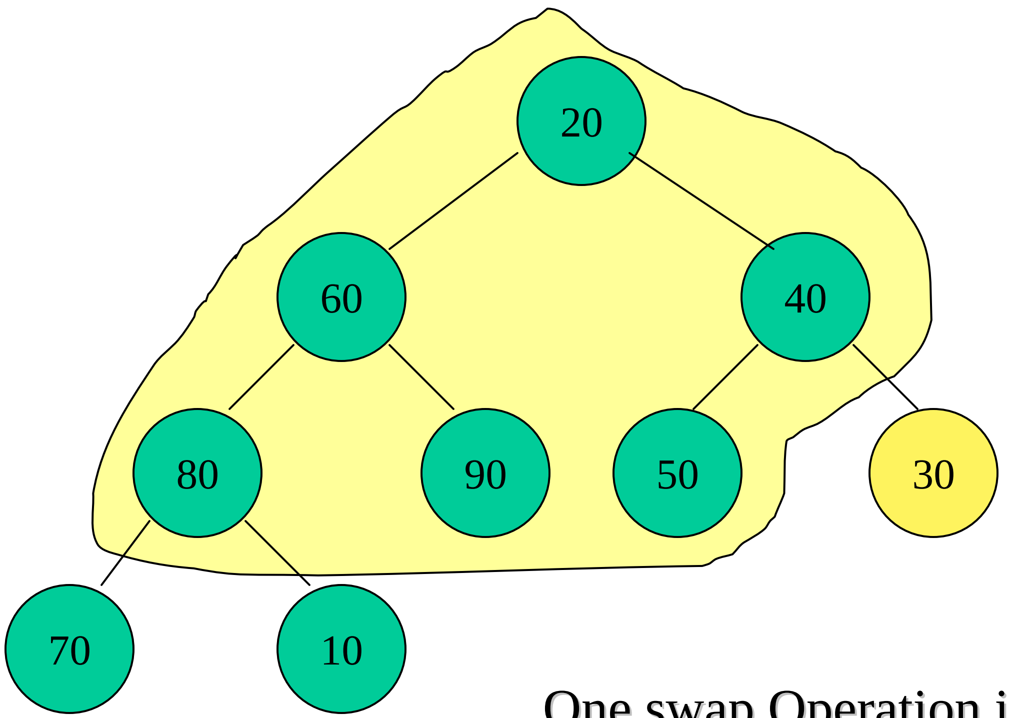
After siftUp 40



[1]	20
[2]	60
[3]	40
[4]	80
[5]	90
[6]	50
[7]	30
[8]	70
[9]	10

siftUp Operation

siftUp 30

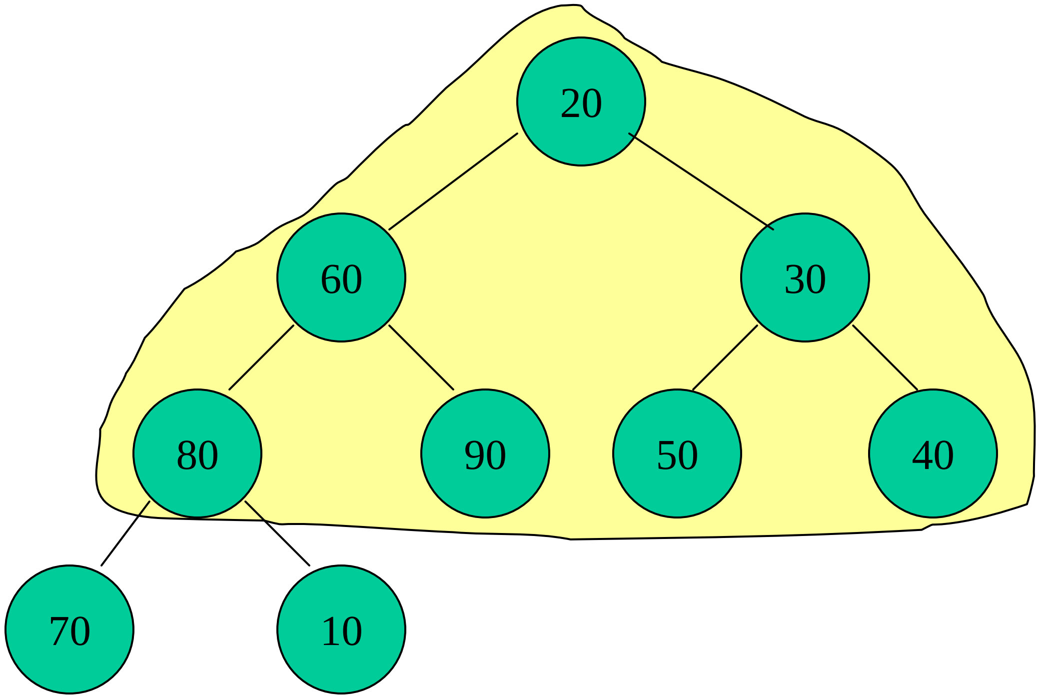


[1]	20
[2]	60
[3]	40
[4]	80
[5]	90
[6]	50
[7]	30
[8]	70
[9]	10

One swap Operation is needed

siftUp Operation

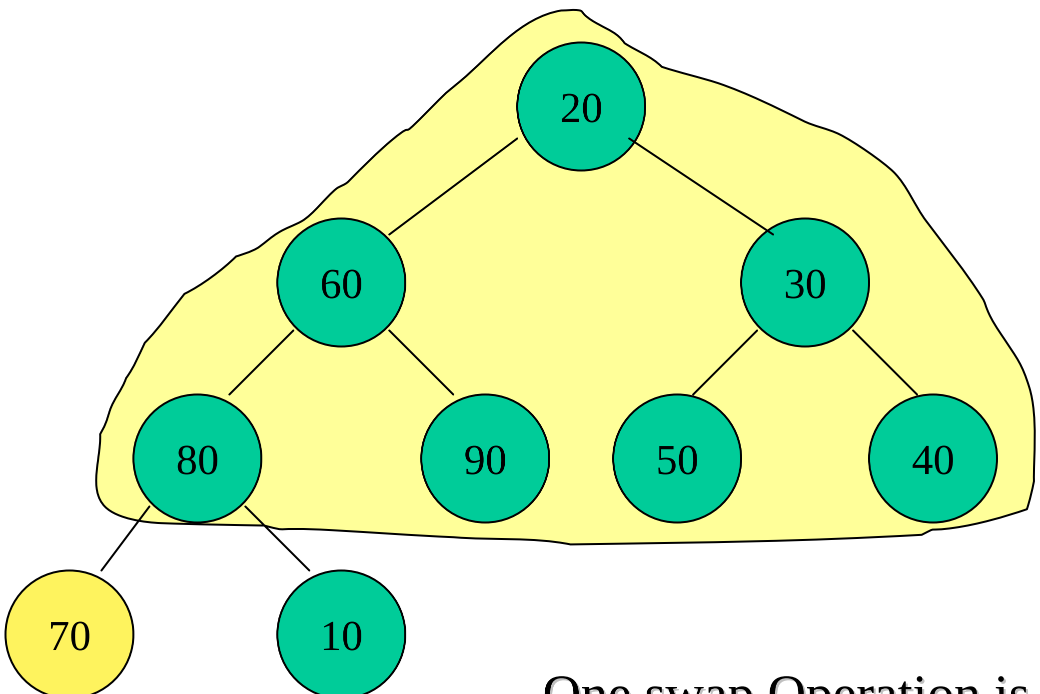
After siftUp 30



[1]	20
[2]	60
[3]	30
[4]	80
[5]	90
[6]	50
[7]	40
[8]	70
[9]	10

siftUp Operation

siftUp 70

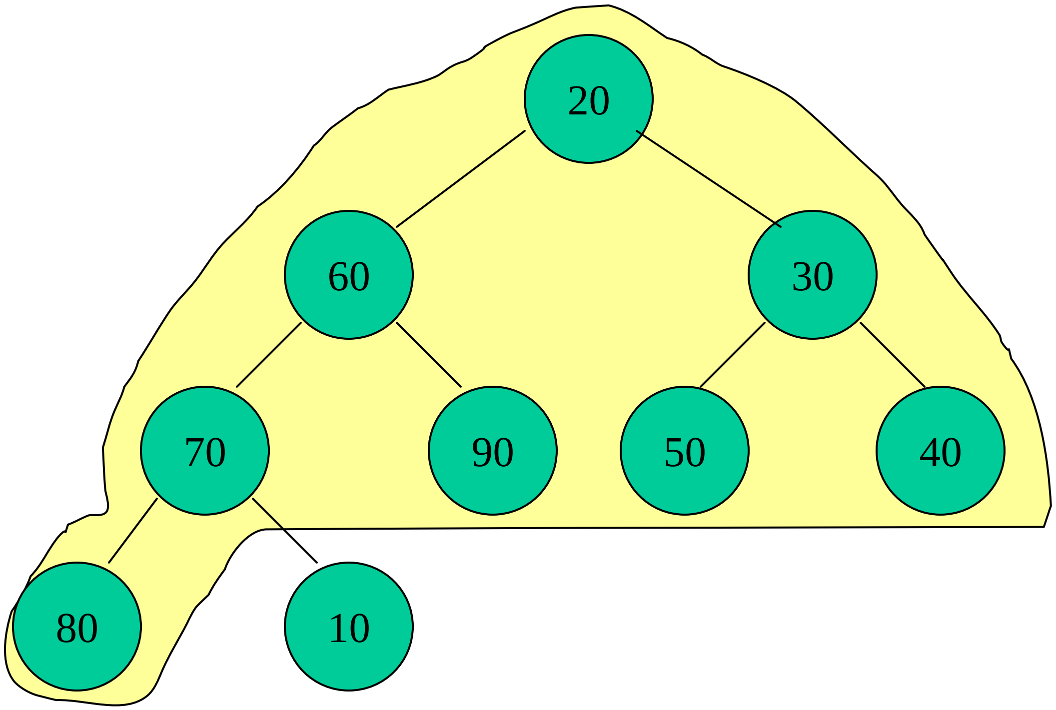


[1]	20
[2]	60
[3]	30
[4]	80
[5]	90
[6]	50
[7]	40
[8]	70
[9]	10

One swap Operation is needed

siftUp Operation

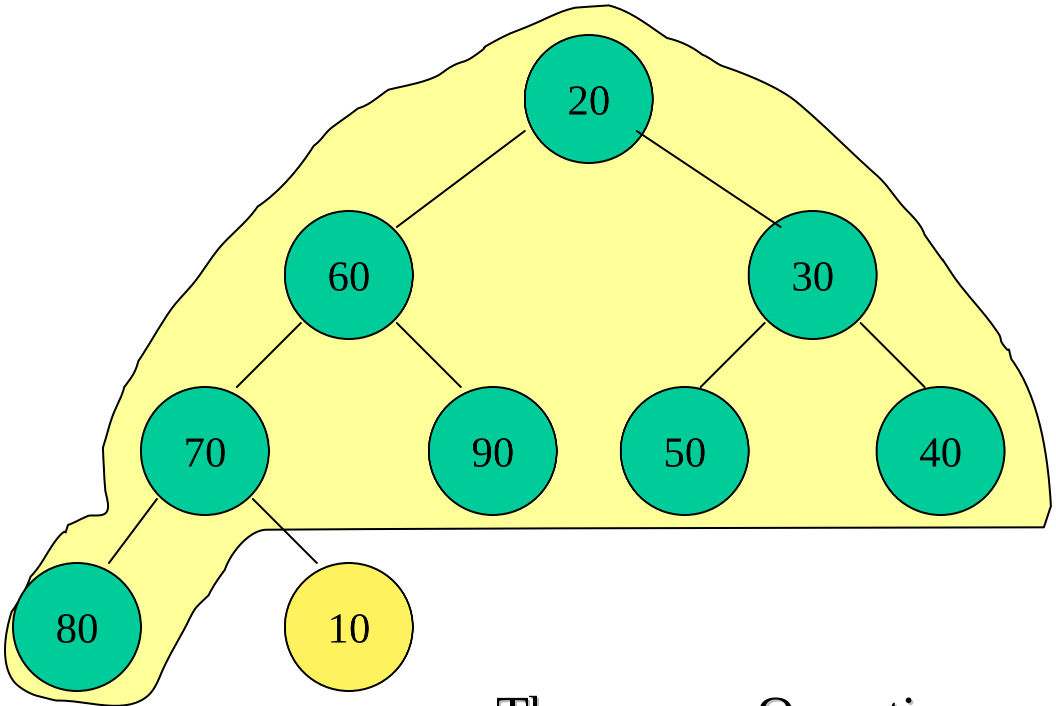
After siftUp 70



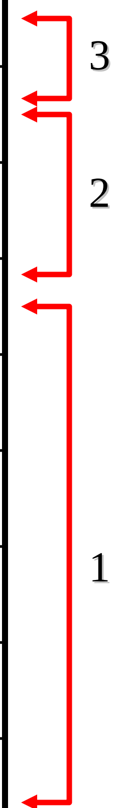
[1]	20
[2]	60
[3]	30
[4]	70
[5]	90
[6]	50
[7]	40
[8]	80
[9]	10

siftUp Operation

siftUp 10



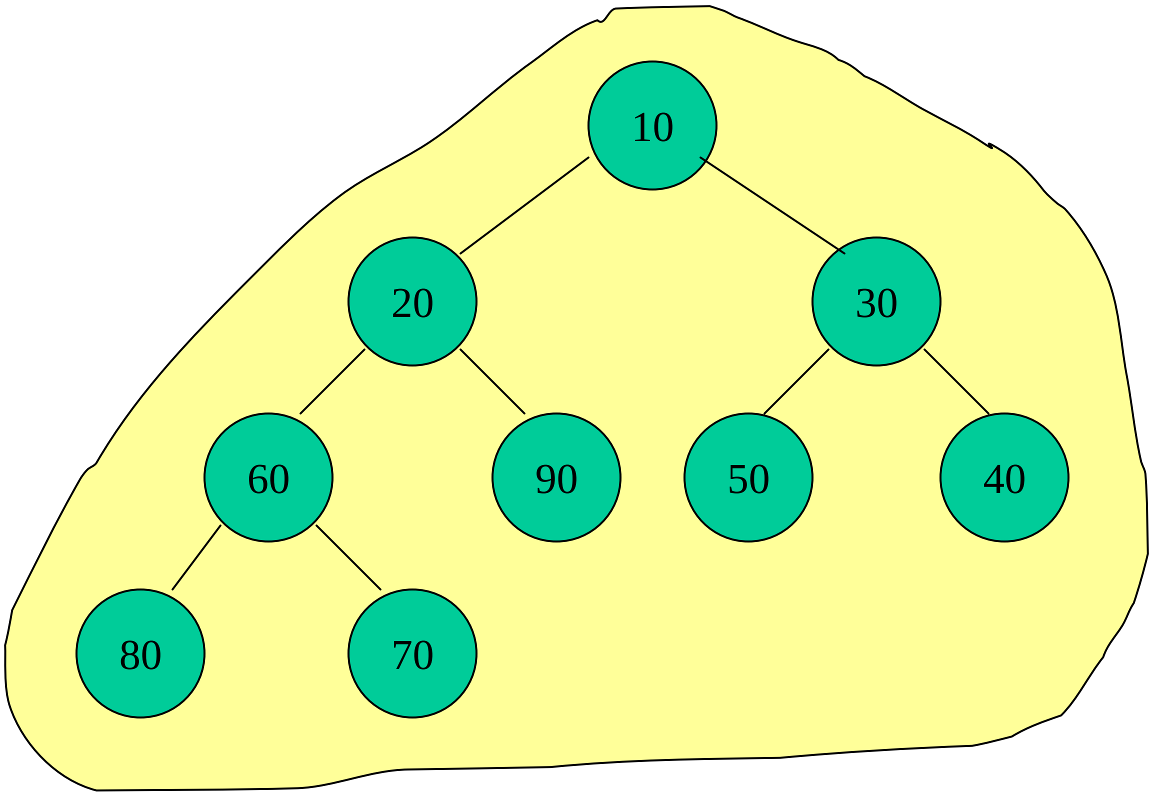
[1]	20
[2]	60
[3]	30
[4]	70
[5]	90
[6]	50
[7]	40
[8]	80
[9]	10



Three swap Operations are needed

siftUp Operation

After siftUp 10



[1]	10
[2]	20
[3]	30
[4]	60
[5]	90
[6]	50
[7]	40
[8]	80
[9]	70

Now it is a heap

Implementation of Heap ADT

```
public class HeapElement<T>
{
      T data;
Priority p;
      public HeapElement(T e, Priority pty) {
            data = e;
      p = pty;}
      public T get_data(){ return data;}
      public void set_data(T e) { data = e;}
      public Priority get_priority(){ return p;}
      public void set_priority(Priority pty){ p =
pty;}
}
```

Implementation of Heap ADT

```
public class Heap<T> {
    int maxsize;
    int size;
    HeapElement<T>[] heap;
  private void swap (HeapElement<T> h[], int
  i, int j)
  private int min(int i, int j)
  public Heap(int n)
  public Heap(T a[], int n)
  public void Display_Heap()
  public void SiftUp(HeapElement<T> e)
  public void SiftDown(int m)
}
```

Implementation of Operations

```
private void swap (HeapElement<T> h[], int i, int j){
        HeapElement<T> tmp;
        tmp = h[i];
        h[i]= h[j];
        h[j]=tmp;
}
```

```
private int min(int i, int j){
        if (i<=j)
            return i;
        return j;
}
```

Implementation of Operations

```
public Heap(int n) {
        maxsize = n;
        size = 0;
        heap = (HeapElement<T>[]) Array.newInstance(HeapElement.class,
n+1);
}
```

```
public Heap(T a[], int n) {
      T x;
      size = n;
      maxsize = n;
      heap = (HeapElement<T>[]) Array.newInstance(HeapElement.class,
      n+1);
      for (int i = 0; i < n; ++i){
        x = (T) new Object();
        heap[i+1] = new HeapElement<T>(x, new Priority(a[i]));
      }
      for (int m = size/2; m >= 1; --m)
      SiftDown(m);
}
```

Implementation of Operations

```
public void Display_Heap(){
  for(int i = 1; i<=size; i++){
    System.out.println(heap[i].get_priority().get_value());
  }
}
```

```
public void SiftUp(HeapElement<T> e){
     heap[++size] = e;
     int i = size;
     while (i > 1 &&
     heap[i/2].get_priority().get_value() >
     e.get_priority().get_value()) {
          heap[i] = heap[i/2];
          i = i/2;
     }
     heap[i] = e;
}
```

Implementation of Operations

```
public void SiftDown(int m){
      int i, k;
      i = m;
      while ((i<=size/2 && heap[i].get_priority().get_value() >
      heap[2*i].get_priority().get_value()) ||
      (2*i+1<=size && heap[i].get_priority().get_value() >
      heap[2*i+1].get_priority().get_value()))
      {
      if (2*i+1<=size && heap[2*i].get_priority().get_value() >
      heap[2*i+1].get_priority().get_value())
                k = 2*i+1;
      else
                k = 2*i;
      swap(heap, i, k);
      i = k;
   }
}
```

Implementation of Priority Queue using Heap

Specification

Elements: Any data type

Structure: any structure that allows to insert data elements in any order but removal of the data element with the highest priority

Domain: Number of elements is bounded

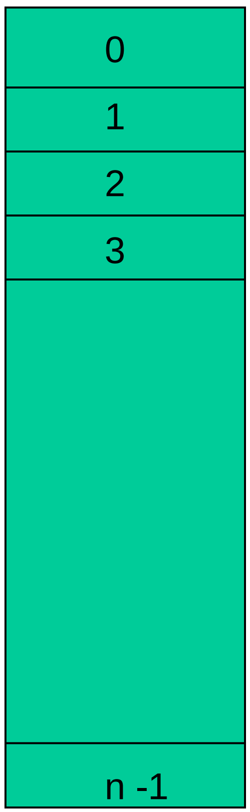
Operations :

Operation	Specification
bool full()	**Precondition/Requires:** none **Processing/Results:** returns true if the priority queue is full otherwise false.
int length()	**Precondition/Requires:** none **Processing/Results:** returns the number of elements currently in the priority queue.
void enqueue(Typ, int)	**Precondition/Requires:** priority queue is not full. **Processing/Results:** inserts a given element into the queue according to its priority.
Typ serve(int &p) or Typ deque(int &p)	**Precondition/Requires:** priority queue is not empty **Processing/Results:** removes the element at the front or head of the priority queue and returns it and its priority as p.

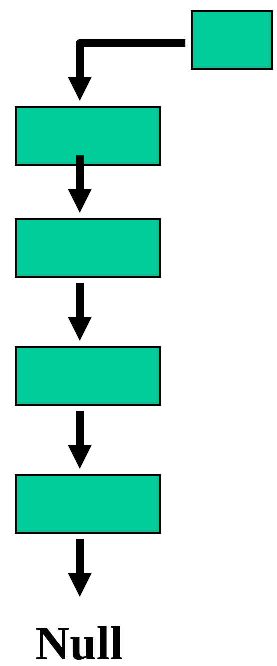
Representation of Priority Queue ADT

Queue ADT can be represented as

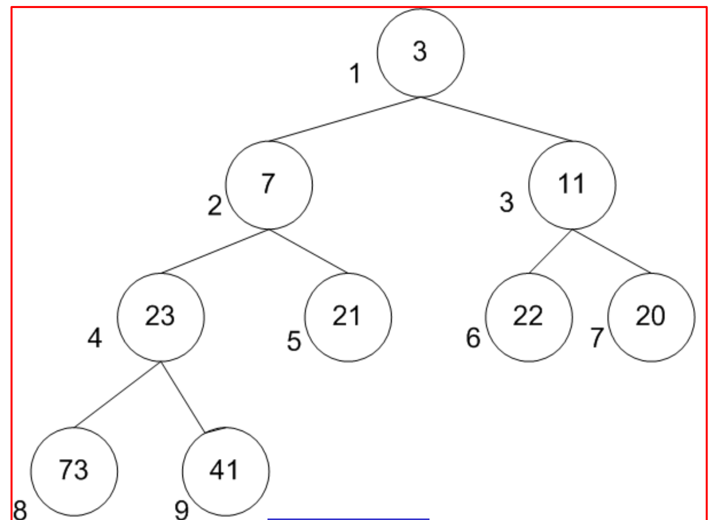
- **Array**
- **Linked List**
- **Heap**



Array



Linked List



Heap

Heap based Implementation of Priority Queue

```
public class HeapPQ<T>
{
     //Data Members
     Heap<T> pq;;
     //Operations
     public HeapPQ(int n)
     public int length ()
     public boolean full ()
     public void enqueue(T e, Priority pty)
     public T serve (Priority pty)
}
```

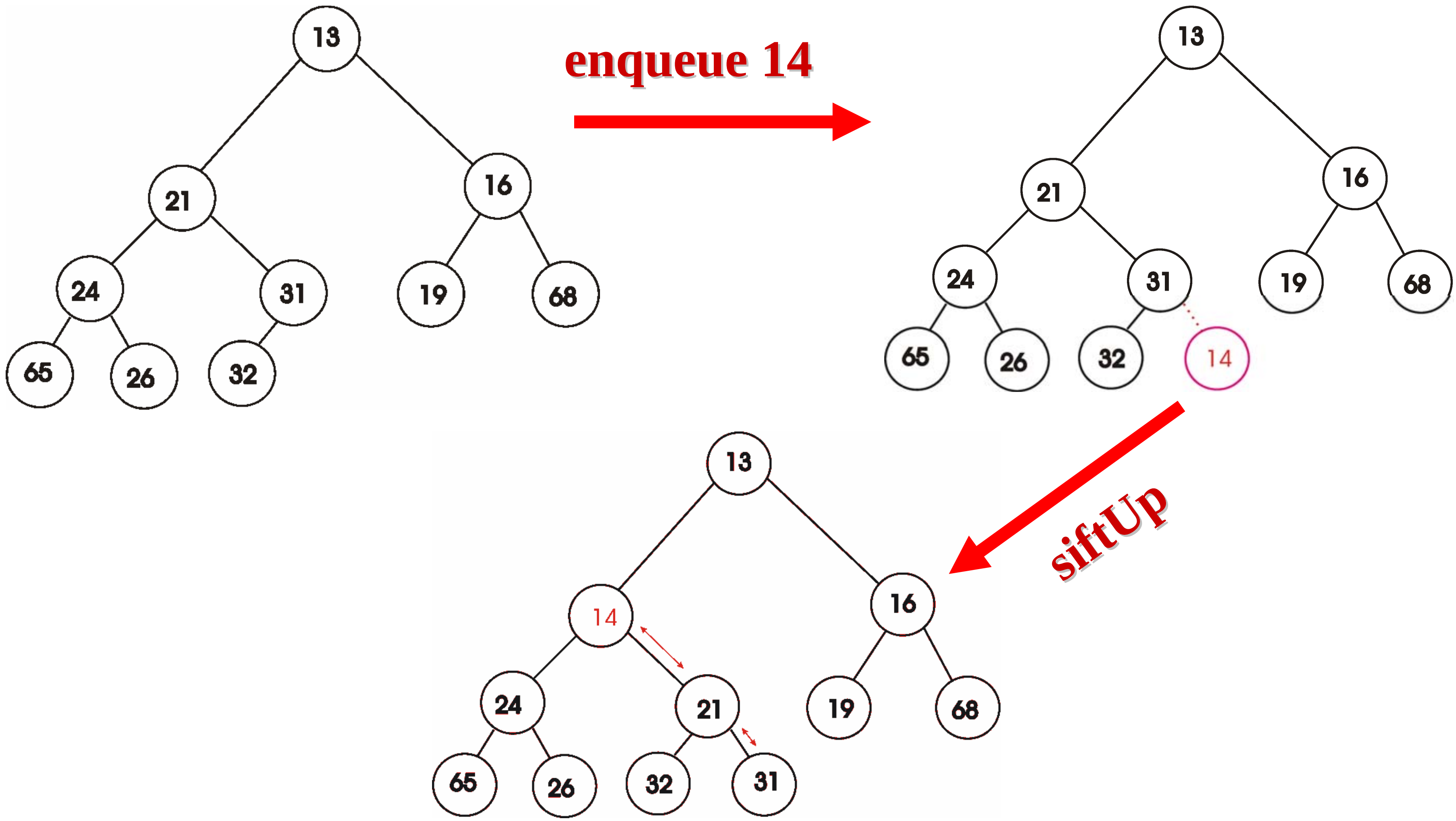
Implementation of Operations

```
public HeapPQ(int n)
{
    pq = new Heap(n);
}
```

```
public int length ()
{
        return pq.size;;
}
```

```
public boolean full ()
{
          return false;
}
```

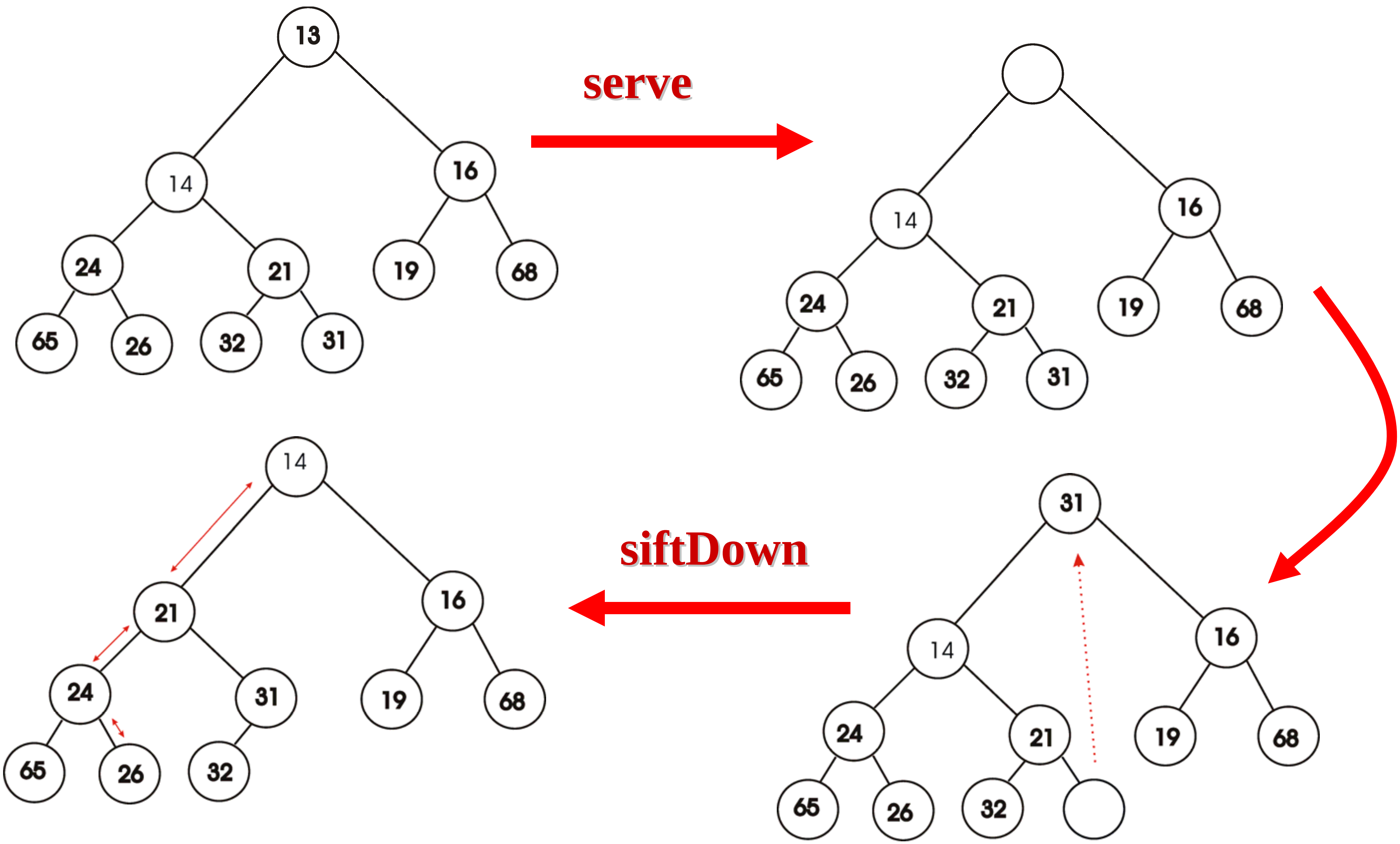
enqueue operation



Implementation of Operations

```
public void enqueue(T e, Priority pty)
{
    HeapElement<T> x = new HeapElement<T> (e, pty);
    pq.SiftUp(x);
}
```

serve operation of Priority Queue ADT



Implementation of Operations

```
public T serve (Priority pty)
{
         T e;
        Priority p;
        e = pq.heap[1].get_data();
        p = pq.heap[1].get_priority();
        pty.set_value(p.get_value());
        pq.heap[1] = pq.heap[pq.size];
        pq.size--;
        pq.SiftDown(1);
        return(e);
}
```

HeapSort

- An other popular application of Heap is in sorting - Heapsort
- HeapSort is based on the idea that heap always has the smallest or largest element at the root.
- Given set of data can be sorted using Heap following these steps:
  - Step 1: Insert each item into an array – H
  - Step 2: Convert it into a heap
  - Step 2: Swap the element with smallest (largest key) H[1] with H[size - 1], reduce size by 1, and apply siftDown operation with m = 1
  - Step 3: Continue Step 2 until the size is zero

Implementation of HeapSort

```
public void HeapSort(T a[], int n){
      T x;
      size = n;
      maxsize = n;
      heap = (HeapElement<T>[])
      Array.newInstance(HeapElement.class, n+1);
      for (int i = 0; i < n; ++i){
            x = (T) new Object();
       heap[i+1] = new HeapElement<T>(x, new Priority(a[i]));
      }
      for (int m = size/2; m >= 1; --m)
      SiftDown(m);
      while (size>1){
      swap(heap, 1, size);
      size--;
      SiftDown(1);
      }
      for(int i = 1; i<=maxsize; i++){
      System.out.println(heap[i].get_priority().get_value());
      }
}
```

Example:

Given the integers 24, 65, 32, 14, 68, 21, 19, 16, 26, 13, 31. Sort them in descending order

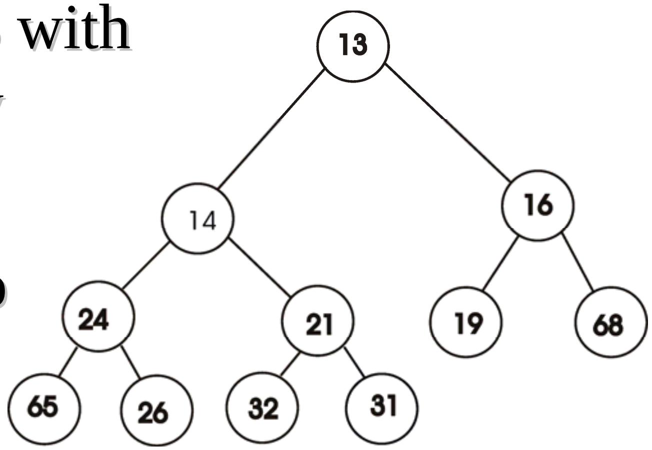
- Put them in an array - H

24	**65**	**32**	**14**	**68**	**21**	**19**	**16**	**26**	**31**	**31**

- Convert it into a min-heap

13	**14**	**16**	**24**	**21**	**19**	**68**	**65**	**26**	**32**	**31**

- Swap the smallest element i.e. H[1] = 13 with H[size] = 31, reduce size by 1, and apply siftDown operation



- Continue the above step until size is zero

The array is in sorted order

68	**65**	**32**	**31**	**26**	**24**	**21**	**19**	**16**	**14**	**13**

NOTE:

- For sorting in descending order , construct min-heap
- For sorting in ascending order, construct max-heap