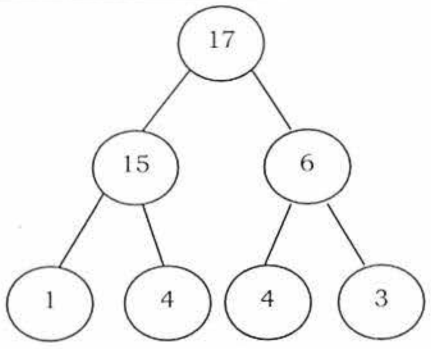


7.6 Binary Heaps

In binary heap each node may have up to two children. In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.

Representing Heaps: Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations.

For the discussion below let us assume that elements are stored in arrays, which starts at index 0. The previous max heap can be represented as:

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Note: For the remaining discussion let us assume that we are doing manipulations in max heap.

Declaration of Heap

```
class Heap:
    def __init__(self):
        self.heapList = [0]           # Elements in Heap
        self.size = 0                 # Size of the heap
```

Time Complexity: O(1).

Parent of a Node

For a node at i^{th} location, its parent is at $\frac{i-1}{2}$ location. In the previous example, the element 6 is at second location and its parent is at 0^{th} location.

```
def parent(self, index):
    """
    Parent will be at math.floor(index/2). Since integer division
    simulates the floor function, we don't explicitly use it
    """
    return index / 2
```

Time Complexity: O(1).

Children of a Node

Similar to the above discussion, for a node at i^{th} location, its children are at $2*i+1$ and $2*i+2$ locations. For example, in the above tree the element 6 is at second location and its children 2 and 5 are at 5 $(2*i+1 = 2*2+1)$ and 6 $(2*i+2 = 2*2+2)$ locations.

```
def leftChild(self, index):
    """ 1 is added because array begins at index 0 """
    return 2 * index + 1
```

Time Complexity: O(1).

```
def rightChild(self, index):
    return 2 * index + 2
```

Time Complexity: O(1).

Getting the Maximum Element

Since the maximum element in max heap is always at root, it will be stored at heapList[0].

```
#Get Maximum for MaxHeap
def getMaximum(self):
    if self.size == 0:
        return -1
    return self.heapList[0]
```

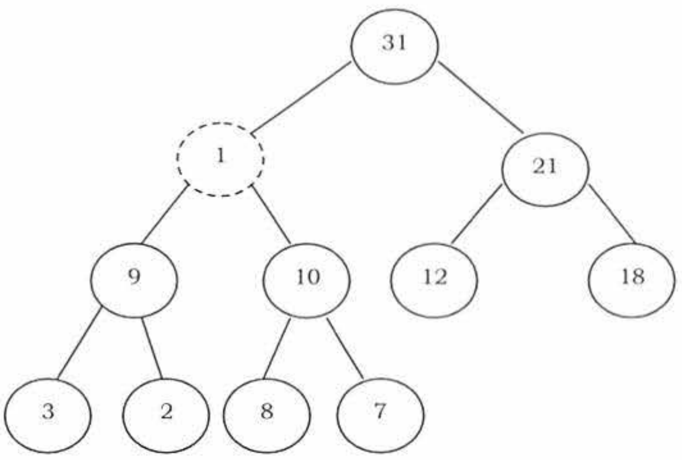
Time Complexity: O(1).

```
#Get Minimum for MinHeap
def getMinimum(self):
    if self.size == 0:
        return -1
    return self.heapList[0]
```

Time Complexity: O(1).

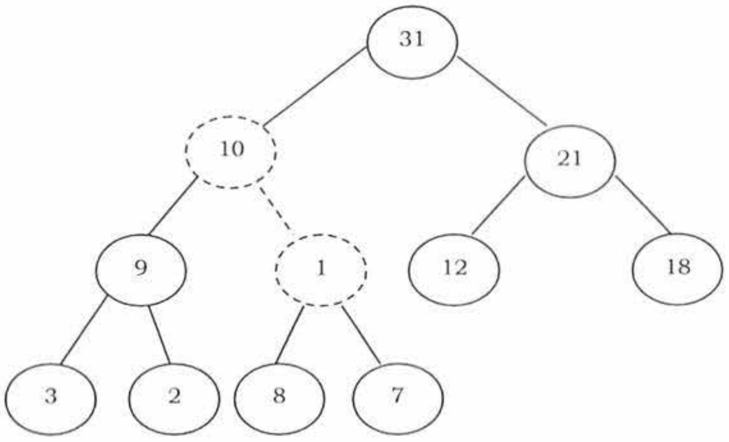
Heapifying an Element

After inserting an element into heap, it may not satisfy the heap property. In that case we need to adjust the locations of the heap to make it heap again. This process is called *heapifying*. In max-heap, to heapify an element, we have to find the maximum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node. In min-heap, to heapify an element, we have to find the minimum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node.

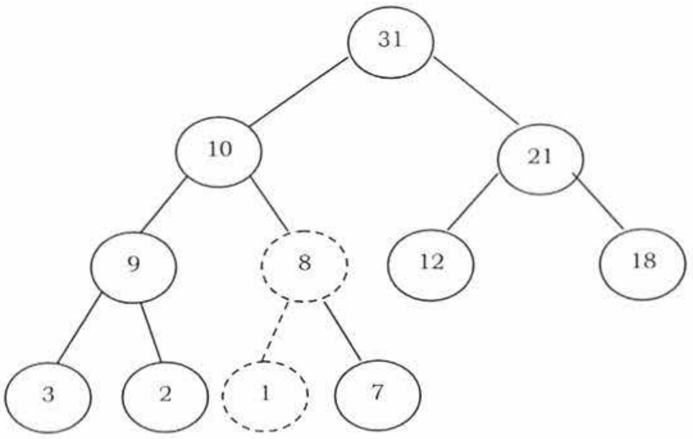


Observation: One important property of heap is that, if an element is not satisfying the heap property, then all the elements from that element to the root will have the same problem. In the example below, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly, if we heapify an element, then all the elements from that element to the root will also satisfy the heap property automatically. Let us go through an example. In the above heap, the element 1 is not satisfying the heap property. Let us try heapifying this element.

To heapify 1, find the maximum of its children and swap with that.



We need to continue this process until the element satisfies the heap properties. Now, swap 1 with 8.



Now the tree is satisfying the heap property. In the above heapifying process, since we are moving from top to bottom, this process is sometimes called *percolate down*. Similarly, if we start heapifying from any other node to root, we can that process *percolate up* as move from bottom to top.

```
def percolateDown(self,i):
  while (i * 2) <= self.size:
    minimumChild = self.minChild(i)
    if self.heapList[i] > self.heapList[minimumChild]:
      tmp = self.heapList[i]
      self.heapList[i] = self.heapList[minimumChild]
      self.heapList[minimumChild] = tmp
    i = minimumChild
def minimumChild(self,i):
  if i * 2 + 1 > self.size:
    return i * 2
  else:
    if self.heapList[i*2] < self.heapList[i*2+1]:
      return i * 2
    else:
      return i * 2 + 1
def percolateUp(self,i):
 while i // 2 > 0:
       if self.heapList[i] < self.heapList[i // 2]:
             tmp = self.heapList[i // 2]
             self.heapList[i // 2] = self.heapList[i]
             self.heapList[i] = tmp
       i = i // 2
```

Time Complexity: O(*logn*). Heap is a complete binary tree and in the worst case we start at the root and come down to the leaf. This is equal to the height of the complete binary tree. Space Complexity: O(1).

Deleting an Element

To delete an element from heap, we just need to delete the element from the root. This is the only operation (maximum element) supported by standard heap. After deleting the root element, copy the last element of the heap (tree) and delete that last element.

After replacing the last element, the tree may not satisfy the heap property. To make it heap again, call the *PercolateDown* function.

- Copy the first element into some variable
- Copy the last element into first element location
- *PercolateDown* the first element

```
#Delete Maximum for MaxHeap
def deleteMax(self):
  retval = self.heapList[1]
  self.heapList[1] = self.heapList[self.size]
  self.size = self.size - 1
  self.heapList.pop()
  self.percolateDown(1)
```

```
#Delete Minimum for MinHeap
def deleteMin(self):
  retval = self.heapList[1]
  self.heapList[1] = self.heapList[self.size]
  self.size = self.size - 1
  self.heapList.pop()
  self.percolateDown(1)
```

```
        return retval
    Time Complexity: O(logn).
```

```
        return retval
    Time Complexity: O(logn).
```

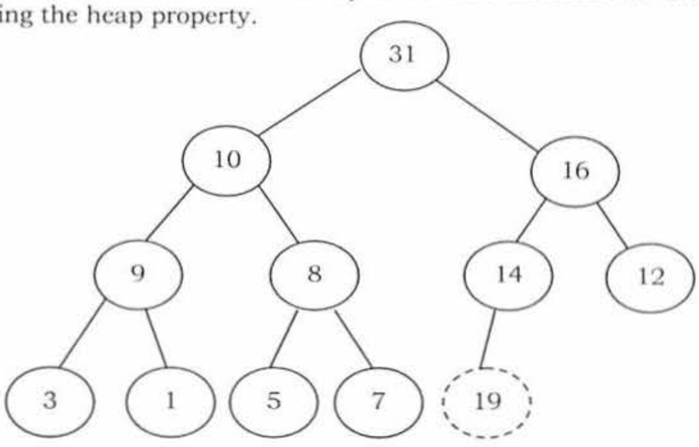
Note: Deleting an element uses *PercolateDown*, and inserting an element uses *PercolateUp*.
Time Complexity: same as *Heapify* function and it is O(*logn*)

Inserting an Element

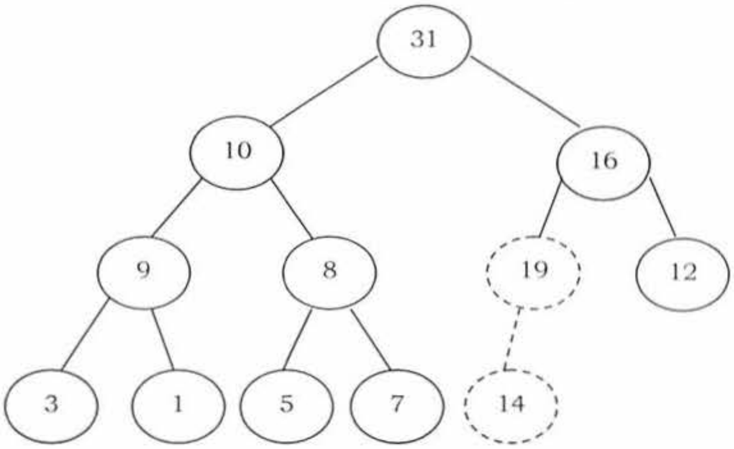
Insertion of an element is similar to the heapify and deletion process.

- Increase the heap size
- Keep the new element at the end of the heap (tree)
- Heapify the element from bottom to top (root)

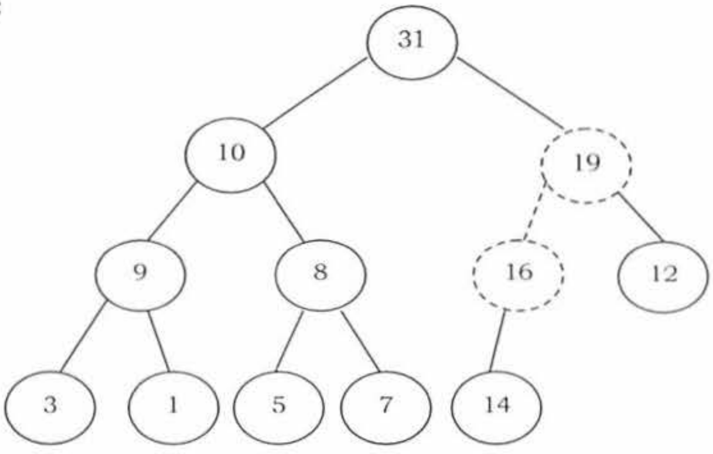
Before going through code, let us look at an example. We have inserted the element 19 at the end of the heap and this is not satisfying the heap property.



In order to heapify this element (19), we need to compare it with its parent and adjust them. Swapping 19 and 14 gives:



Again, swap 19 and16:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach we sometimes call this process *percolate up*.

```
def insert(self,k):
  self.heapList.append(k)
  self.size = self.size + 1
  self.percolateUp(self.size)
```

Time Complexity: $O(logn)$. The explanation is the same as that of the *Heapify* function.

Heapifying the Array

One simple approach for building the heap is, take n input items and place them into an empty heap. This can be done with n successive inserts and takes $O(nlogn)$ in the worst case. This is due to the fact that each insert operation takes $O(logn)$.

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately $O(logn)$ operations. However, remember that inserting an item in the middle of the list may require $O(n)$ operations to shift the rest of the list over to make room for the new key. Therefore, to insert n keys into the heap would require a total of $O(nlogn)$ operations. However, if we start with an entire list then we can build the whole heap in $O(n)$ operations.

Observation: Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the end and to heapify the given array it should be enough if we heapify the non-leaf nodes. Now let us concentrate on finding the first non-leaf node. The last element of the heap is at location $h \rightarrow count - 1$, and to find the first non-leaf node it is enough to find the parent of the last element.

```
def buildHeap(self,A):
  i = len(A) // 2
  self.size = len(A)
  self.heapList = [0] + A[:]
  while (i > 0):
        self.percolateDown(i)
        i = i - 1
```

Time Complexity: The linear time bound of building heap can be shown by computing the sum of the heights of all the nodes. For a complete binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $n - h - 1 = n - logn - 1$ (for proof refer to *Problems Section*). That means, building the heap operation can be done in linear time ($O(n)$) by applying a *PercolateDown* function to the nodes in reverse level order.

7.7 Heapsort

One main application of heap ADT is sorting (heap sort). The heap sort algorithm inserts all elements (from an unsorted array) into a heap, then removes them from the root of a heap until the heap is empty. Note that heap sort can be done in place with the array to be sorted. Instead of deleting an element, exchange the first element (maximum) with the last element and reduce the heap size (array size). Then, we heapify the first element. Continue this process until the number of remaining elements is one.

```
def heapSort( A ):
  # convert A to heap
  length = len( A ) - 1
  leastParent = length / 2
  for i in range ( leastParent, -1, -1 ):
    percolateDown( A, i, length )

  # flatten heap into sorted array
  for i in range ( length, 0, -1 ):
    if A[0] > A[i]:
      swap( A, 0, i )
      percolateDown( A, 0, i - 1 )

#Modfied percolateDown to skip the sorted elements
def percolateDown( A, first, last ):
  largest = 2 * first + 1
  while largest <= last:
    # right child exists and is larger than left child
    if ( largest < last ) and ( A[largest] < A[largest + 1] ):
      largest += 1

    # right child is larger than parent
    if A[largest] > A[first]:
      swap( A, largest, first )
      # move down to largest child
      first = largest;
      largest = 2 * first + 1
    else:
      return # force exit

def swap( A, x, y ):
  temp = A[x]
  A[x] = A[y]
  A[y] = temp
```

Time complexity: As we remove the elements from the heap, the values become sorted (since maximum elements are always *root* only). Since the time complexity of both the insertion algorithm and deletion algorithm is $O(logn)$ (where n is the number of items in the heap), the time complexity of the heap sort algorithm is $O(nlogn)$.

7.8 Priority Queues [Heaps]: Problems & Solutions

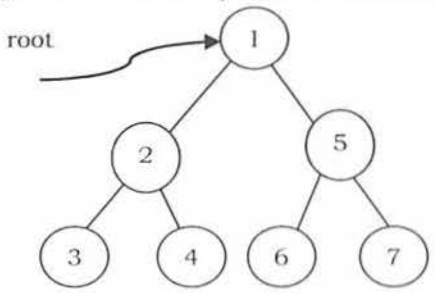
Problem-1 What are the minimum and maximum number of elements in a heap of height h?

Solution: Since heap is a complete binary tree (all levels contain full nodes except possibly the lowest level), it has at most $2^{h+1} - 1$ elements (if it is complete). This is because, to get maximum nodes, we need to fill all the h levels completely and the maximum number of nodes is nothing but the sum of all nodes at all h levels.

To get minimum nodes, we should fill the $h - 1$ levels fully and the last level with only one element. As a result, the minimum number of nodes is nothing but the sum of all nodes from $h - 1$ levels plus 1 (for the last level) and we get $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and all the other levels are complete).

Problem-2 Is there a min-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted order?

Solution: **Yes**. For the tree below, preorder traversal produces ascending order.



Problem-3 Is there a max-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted order?