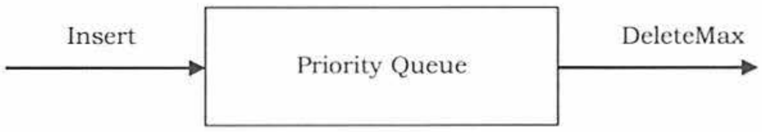
CHAPTER 7

PRIORITY QUEUES AND HEAPS

7.1 What is a Priority Queue?

In some situations we may need to find the minimum/maximum element among a collection of elements. We can do this with the help of Priority Queue ADT. A priority queue ADT is a data structure that supports the operations *Insert* and *DeleteMin* (which returns and removes the minimum element) or *DeleteMax* (which returns and removes the maximum element).

These operations are equivalent to *EnQueue* and *DeQueue* operations of a queue. The difference is that, in priority queues, the order in which the elements enter the queue may not be the same in which they were processed. An example application of a priority queue is job scheduling, which is prioritized instead of serving in first come first serve.



A priority queue is called an *ascending − priority* queue, if the item with the smallest key has the highest priority (that means, delete the smallest element always). Similarly, a priority queue is said to be a *descending − priority* queue if the item with the largest key has the highest priority (delete the maximum element always). Since these two types are symmetric we will be concentrating on one of them: ascending-priority queue.

7.2 Priority Queue ADT

The following operations make priority queues an ADT.

Main Priority Queues Operations

A priority queue is a container of elements, each having an associated key.

- Insert (key, data): Inserts data with *key* to the priority queue. Elements are ordered based on key.
- DeleteMin/DeleteMax: Remove and return the element with the smallest/largest key.
- GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- k^{th} −Smallest/k^{th} −Largest: Returns the k^{th} −Smallest/k^{th} −Largest key in priority queue.
- Size: Returns number of elements in priority queue.
- Heap Sort: Sorts the elements in the priority queue based on priority (key).

7.3 Priority Queue Applications

Priority queues have many applications – a few of them are listed below:

- Data compression: Huffman Coding algorithm
- Shortest path algorithms: Dijkstra's algorithm
- Minimum spanning tree algorithms: Prim's algorithm
- Event-driven simulation: customers in a line
- Selection problem: Finding k^{th}- smallest element

7.4 Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then deleting.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Unordered List Implementation

It is very similar to array implementation, but instead of using arrays, linked lists are used.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Ordered Array Implementation

Elements are inserted into the array in sorted order based on key field. Deletions are performed at only one end.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Ordered List Implementation

Elements are inserted into the list in sorted order based on key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Binary Search Trees Implementation

Both insertions and deletions take O($logn$) on average if insertions are random (refer to *Trees* chapter).

Balanced Binary Search Trees Implementation

Both insertions and deletion take O($logn$) in the worst case (refer to *Trees* chapter).

Binary Heap Implementation

In subsequent sections we will discuss this in full detail. For now, assume that binary heap implementation gives O($logn$) complexity for search, insertions and deletions and O(1) for finding the maximum or minimum element.

Comparing Implementations

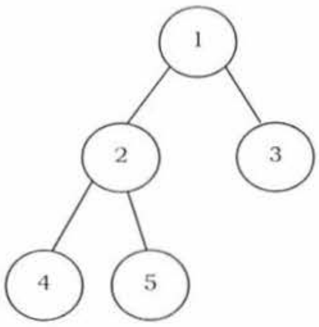
Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1

Binary Search Trees	*logn* (average)	*logn* (average)	*logn* (average)
Balanced Binary Search Trees	*logn*	*logn*	*logn*
Binary Heaps	*logn*	*logn*	1

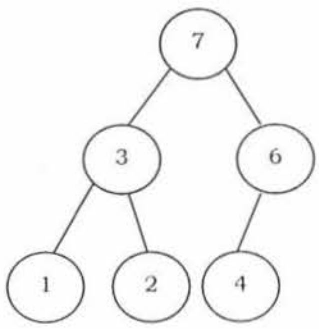
7.5 Heaps and Binary Heaps

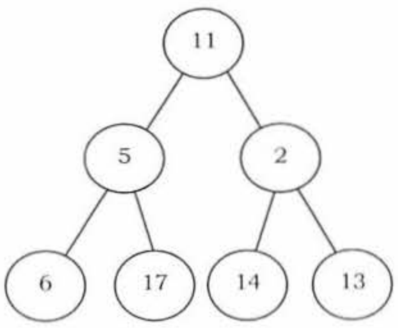
What is a Heap?

A heap is a tree with some special properties. The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called *heap property*. A heap also has the additional property that all leaves should be at h or $h - 1$ levels (where h is the height of the tree) for some $h > 0$ (*complete binary trees*). That means heap should form a *complete binary tree* (as shown below).



In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).

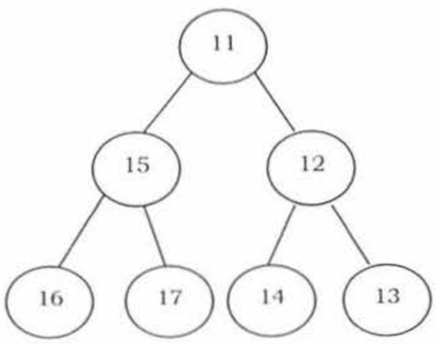




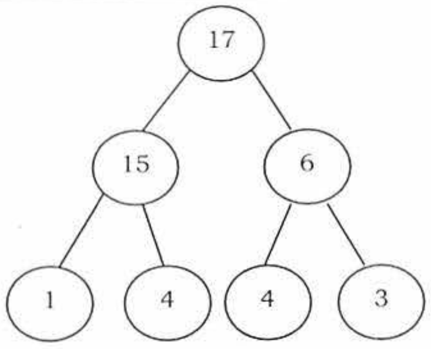
Types of Heaps?

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children



- **Max heap:** The value of a node must be greater than or equal to the values of its children



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]
```

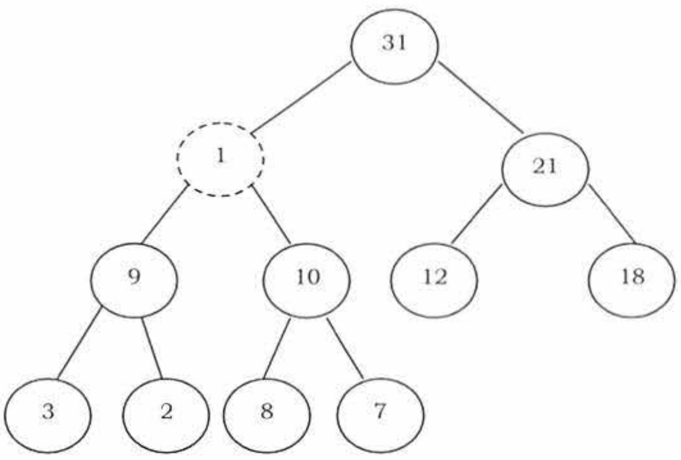
Time Complexity: O(1).

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

Time Complexity: O(1).

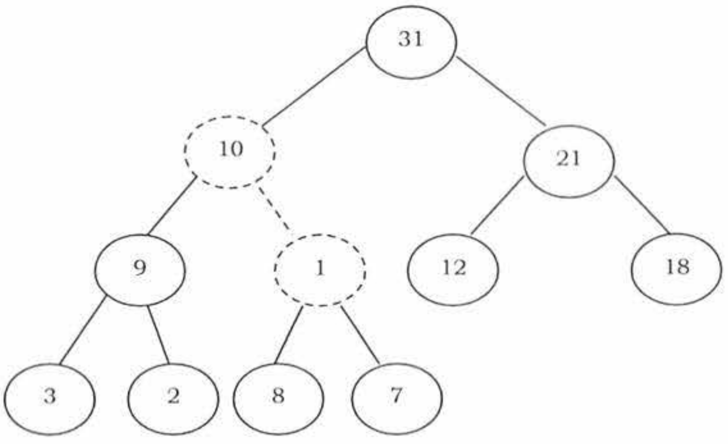
Heapifying an Element

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.

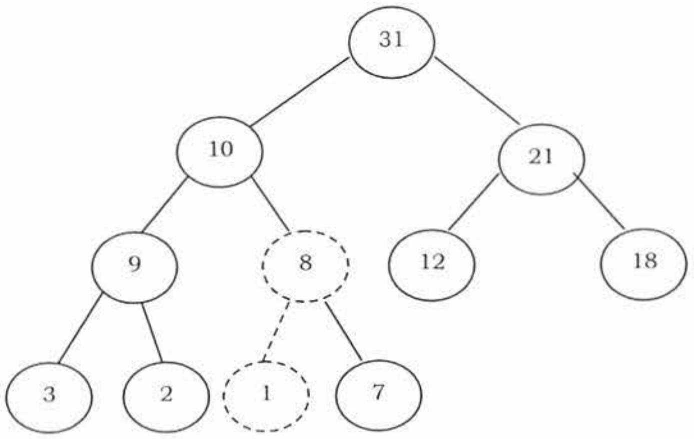


Observation: One important property of heap is that, if an element is not satisfying the heap property, then all the elements from that element to the root will have the same problem. In the example below, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly, 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).
```

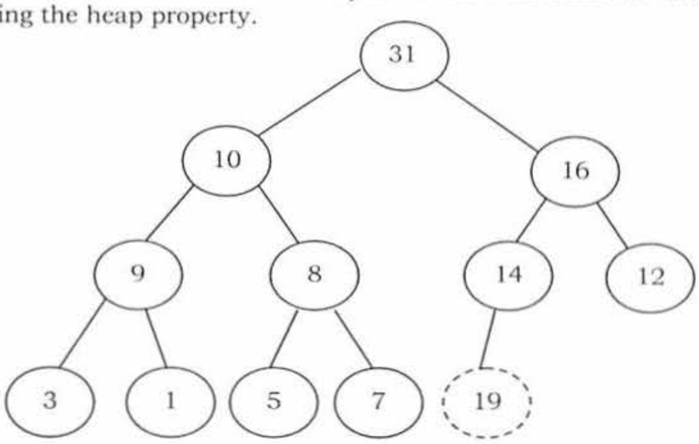
Note: Deleting an element uses *PercolateDown*, and inserting an element uses *PercolateUp*.
Time Complexity: same as *Heapify* function and it is O(*logn*)

Inserting an Element

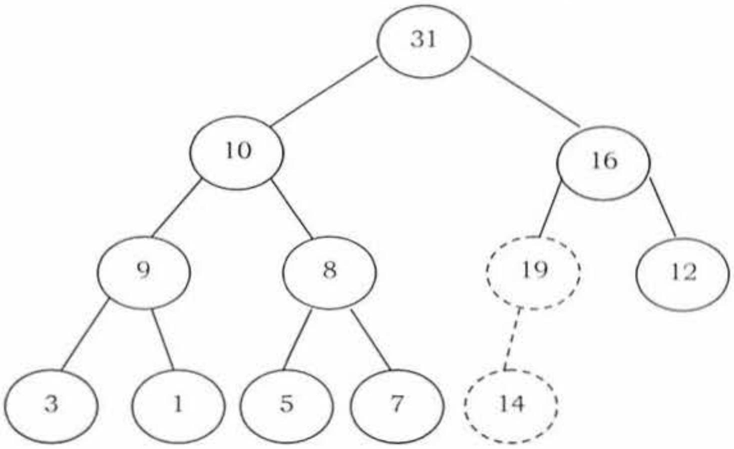
Insertion of an element is similar to the heapify and deletion process.

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

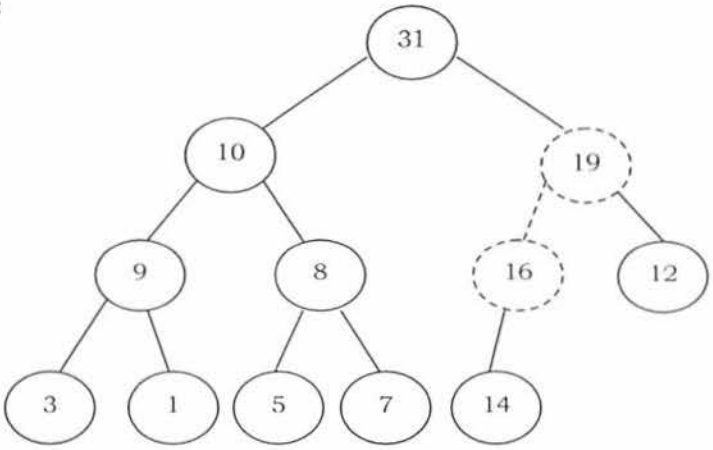
Before going through code, let us look at an example. We have inserted the element 19 at the end of the heap and this is not satisfying the heap property.



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)
```

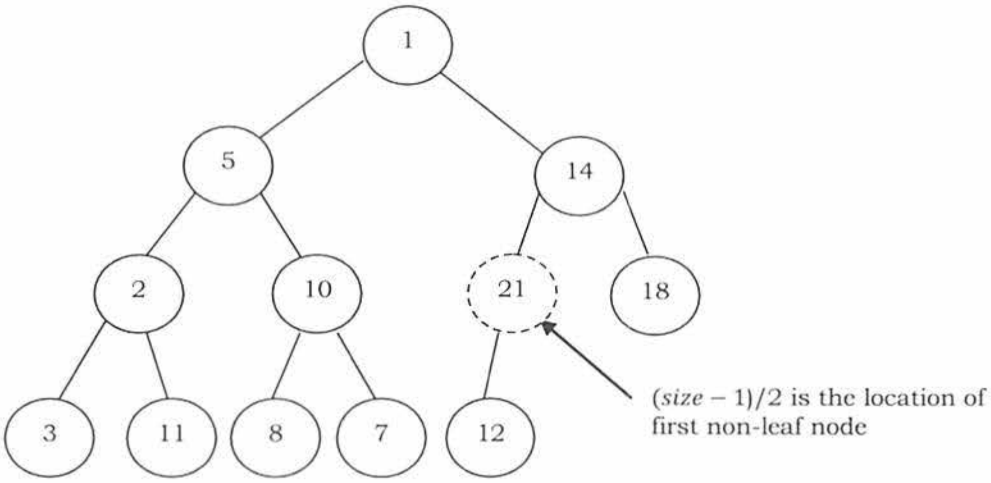
Time Complexity: $O(logn)$. The explanation is the same as that of the *Heapify* function.

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

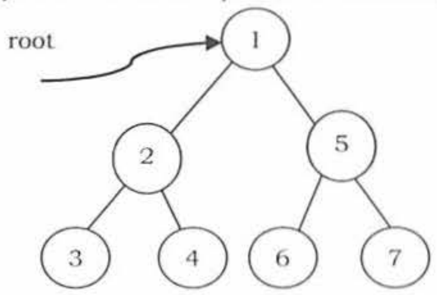
Problem-1 What are the minimum and maximum number of elements in a heap of height h?

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).

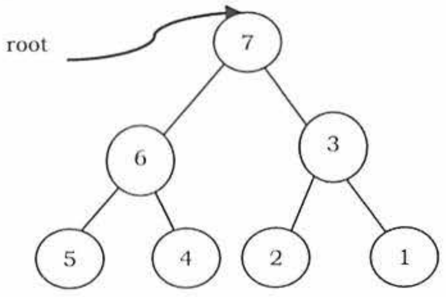
Problem-2 Is there a min-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted order?

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?

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

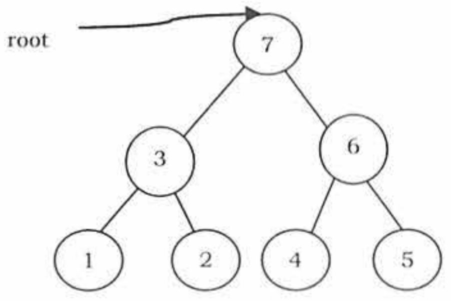


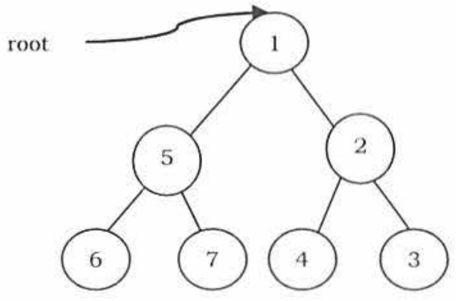
Problem-4 Is there a min-heap/max-heap with seven distinct elements so that the inorder traversal of it gives the elements in sorted order?

Solution: No. Since a heap must be either a min-heap or a max-heap, the root will hold the smallest element or the largest. An inorder traversal will visit the root of the tree as its second step, which is not the appropriate place if the tree's root contains the smallest or largest element.

Problem-5 Is there a min-heap/max-heap with seven distinct elements so that the postorder traversal of it gives the elements in sorted order?

Solution:





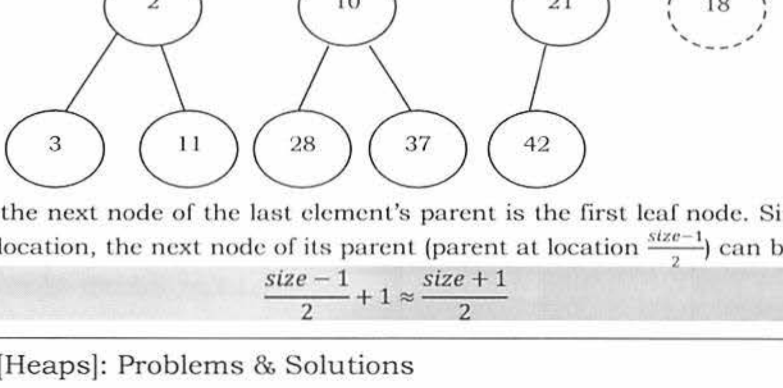
Yes, if the tree is a max-heap and we want descending order (below left), or if the tree is a min-heap and we want ascending order (below right).

Problem-6 Show that the height of a heap with n elements is $logn$?

Solution: A heap is a complete binary tree. All the levels, except the lowest, are completely full. A heap has at least 2^h elements and at most elements $2^h \leq n \leq 2^{h+1} - 1$. This implies, $h \leq logn \leq h + 1$. Since h is an integer, $h = logn$.

Problem-7 Given a min-heap, give an algorithm for finding the maximum element.

Solution: For a given min heap, the maximum element will always be at leaf only. Now, the next question is how to find the leaf nodes in the tree.



If we carefully observe, the next node of the last element's parent is the first leaf node. Since the last element is always at the $size - 1^{th}$ location, the next node of its parent (parent at location $\frac{size-1}{2}$) can be calculated as:

$$\frac{size-1}{2} + 1 \approx \frac{size+1}{2}$$

Now, the only step remaining is scanning the leaf nodes and finding the maximum among them.

```
def findMaxInMinHeap(self):
    max = -1
    for i in range((self.size+1)//2, self.size):
            if(self.array[i] > max):
                  max = self.array[i]
return max
```

Time Complexity: $O(\frac{n}{2}) \approx O(n)$.

Problem-8 Give an algorithm for deleting an arbitrary element from min heap.

Solution: To delete an element, first we need to search for an element. Let us assume that we are using level order traversal for finding the element. After finding the element we need to follow the DeleteMin process.

Time Complexity = Time for finding the element + Time for deleting an element
$= O(n) + O(logn) \approx O(n)$. //Time for searching is dominated.

Problem-9 Give an algorithm for deleting the i^{th} indexed element in a given min-heap.

Solution: Delete the i^{th}elemenet and perform heapify at i^{th}position.

```
def Delete(self, i):
    if(self.size < i):
          print("Wrong position")
          return
    key = self.array[i]
    self.array[i]= self.array[self.size-1]
    self.size -= 1
    seld.percolateDown(i)
    return key
```

Time Complexity = $O(logn)$.

Problem-10 Prove that, for a complete binary tree of height h the sum of the height of all nodes is $O(n - h)$.

Solution: A complete binary tree has 2^i nodes on level i. Also, a node on level i has depth i and height $h - i$. Let us assume that S denotes the sum of the heights of all these nodes and S can be calculated as:

$$S = \sum_{i=0}^{h} 2^i(h-i)$$
$$S = h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}(1)$$

Multiplying with 2 on both sides gives: $2S = 2h + 4(h-1) + 8(h-2) + \cdots + 2^h(1)$

Now, subtract S from $2S$: $2S - S = -h + 2 + 4 + \cdots + 2^h \Rightarrow S = (2^{h+1} - 1) - (h-1)$

But, we already know that the total number of nodes n in a complete binary tree with height h is $n = 2^{h+1} - 1$. This gives us: $h = \log(n+1)$.

Finally, replacing $2^{h+1} - 1$ with n, gives: $S = n - (h-1) = O(n - logn) = O(n - h)$.

Problem-11 Give an algorithm to find all elements less than some value of k in a binary heap.

Solution: Start from the root of the heap. If the value of the root is smaller than k then print its value and call recursively once for its left child and once for its right child. If the value of a node is greater or equal than k then the function stops without printing that value.

The complexity of this algorithm is $O(n)$, where n is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than k, so the function has to call each node of the heap.

Problem-12 Give an algorithm for merging two binary max-heaps. Let us assume that the size of the first heap is $m + n$ and the size of the second heap is n.

Solution: One simple way of solving this problem is:

- Assume that the elements of the first array (with size $m + n$) are at the beginning. That means, first m cells are filled and remaining n cells are empty.
- Without changing the first heap, just append the second heap and heapify the array.
- Since the total number of elements in the new array is $m + n$, each heapify operation takes $O(log(m + n))$.

The complexity of this algorithm is : $O((m + n)log(m + n))$.

Problem-13 Can we improve the complexity of Problem-12?

Solution: Instead of heapifying all the elements of the $m + n$ array, we can use the technique of "building heap with an array of elements (heapifying array)". We can start with non-leaf nodes and heapify them. The algorithm can be given as:

- Assume that the elements of the first array (with size $m + n$) are at the beginning. That means, the first m cells are filled and the remaining n cells are empty.
- Without changing the first heap, just append the second heap.
- Now, find the first non-leaf node and start heapifying from that element.

In the theory section, we have already seen that building a heap with n elements takes $O(n)$ complexity. The complexity of merging with this technique is: $O(m + n)$.

Problem-14 Is there an efficient algorithm for merging 2 max-heaps (stored as an array)? Assume both arrays have n elements.

Solution: The alternative solution for this problem depends on what type of heap it is. If it's a standard heap where every node has up to two children and which gets filled up so that the leaves are on a maximum of two different rows, we cannot get better than $O(n)$ for the merge.

There is an $O(logm \times logn)$ algorithm for merging two binary heaps with sizes m and n. For $m = n$, this algorithm takes $O(log^2n)$ time complexity. We will be skipping it due to its difficulty and scope.

For better merging performance, we can use another variant of binary heap like a *Fibonacci-Heap* which can merge in O(1) on average (amortized).

Problem-15 Give an algorithm for finding the k^{th} smallest element in min-heap.

Solution: One simple solution to this problem is: perform deletion k times from min-heap.

```
def kthSmallest(collection, k):
    """Return kth smallest element in collection for valid k >=1 """
    A = collection[:k]
    buildHeap(A)
    for i in range(k, len(collection)):
        if collection[i] < A[0]:
            A[0] = collection[i]
            heapify(A, 0, k)
    return A[0]
def buildHeap(A):
    n = len(A)
    for i in range(n/2-1, -1, -1):
        heapify(A, i, n)
def heapify (A, index, maxIndex):
    """Ensure structure rooted at A[index] is a heap"""
    left = 2*index+1
    right = 2*index+2
    if left < maxIndex and A[left] > A[index]:
        largest = left
    else:
        largest = index
    if right < maxIndex and A[right] > A[largest]:
        largest = right
    if largest != index:
        A[index],A[largest] = A[largest],A[index]
        heapify(A, largest, maxIndex)
print kthSmallest(range(10),3)
print kthSmallest(range(10),1)
print kthSmallest(range(10),10)
```

Time Complexity: $O(klogn)$. Since we are performing deletion operation k times and each deletion takes $O(logn)$.

Problem-16 For Problem-15, can we improve the time complexity?

Solution: Assume that the original min-heap is called *HOrig* and the auxiliary min-heap is named *HAux*. Initially, the element at the top of *HOrig*, the minimum one, is inserted into *HAux*. Here we don't do the operation of DeleteMin with *HOrig*.

Every while-loop iteration gives the k^{th} smallest element and we need k loops to get the k^{th} smallest elements. Because the size of the auxiliary heap is always less than k, every while-loop iteration the size of the auxiliary heap increases by one, and the original heap $HOrig$ has no operation during the finding, the running time is O($klogk$).

Note: The above algorithm is useful if the k value is too small compared to n. If the k value is approximately equal to n, then we can simply sort the array (let's say, using *couting* sort or any other linear sorting algorithm) and return k^{th} smallest element from the sorted array. This gives O(n) solution.

```
import heapq
class Heap:
    def __init__(self):
            self.heapList = [0]                    # Elements in Heap
            self.size = 0                          # Size of the heap
    def parent(self, index):
            return index // 2
    def leftChildIndex(self, index):
            return 2 * index
    def rightChildIndex(self, index):
            return 2 * index + 1
    def leftChild(self, index):
            if 2 * index  <= self.size:
                    return self.heapList[2 * index ]
            return -1
    def rightChild(self, index):
            if 2 * index + 1 <= self.size :
                    return self.heapList[2 * index + 1]
            return -1
    def searchElement(self,itm):
            i = 1
            while (i <= self.size):
                    if itm == self.heapList[i]:
                            return i
                    i += 1
    def getMinimum(self):
        if self.size == 0:
            return -1
        return self.heapList[1]
    def percolateDown(self,i):
        while (i * 2) <= self.size:
            minimumChild = self.minimumChild(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
    #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
    def insert(self,k):
            self.heapList.append(k)
            self.size = self.size + 1
            self.percolateUp(self.size)
    def printHeap(self):
            print self.heapList[1:]
def FindKthLargestEle(HOrig, k):
    count=1
    HAux = Heap()
    itm = HOrig.getMinimum()
    HAux.insert(itm)
    if count == k:
            return itm
    while (HAux.size>=1):
            itm  = HAux.deleteMin()
            count +=  1
            if count == k:
                    return itm
            else:
                    if HOrig.rightChild(HOrig.searchElement(itm)) != -1:
                            HAux.insert(HOrig.rightChild(HOrig.searchElement(itm)))
                    if HOrig.leftChild(HOrig.searchElement(itm)) != -1:
                            HAux.insert(HOrig.leftChild(HOrig.searchElement(itm)))
HOrig = Heap()
# add some test data:
HOrig.insert(1)
HOrig.insert(20)
HOrig.insert(5)
HOrig.insert(100)
HOrig.insert(1000)
HOrig.insert(12)
HOrig.insert(18)
HOrig.insert(16)
print FindKthLargestEle(HOrig,6)
print FindKthLargestEle(HOrig,3)
```

Problem-17 Find k max elements from max heap.

Solution: One simple solution to this problem is: build max-heap and perform deletion k times.

$$T(n) = \text{DeleteMin from heap } k \text{ times} = \Theta(klogn).$$

Problem-18 For Problem-17, is there any alternative solution?

Solution: We can use the Problem-16 solution. At the end, the auxiliary heap contains the k-largest elements. Without deleting the elements we should keep on adding elements to $HAux$.

Problem-19 How do we implement stack using heap?

Solution: To implement a stack using a priority queue PQ (using min heap), let us assume that we are using one extra integer variable c. Also, assume that c is initialized equal to any known value (e.g., 0). The implementation of the stack ADT is given below. Here c is used as the priority while inserting/deleting the elements from PQ.

```
def Push(element):
    PQ.Insert(c, element)
    c -= 1
def Pop():
    return PQ.DeleteMin()
def Top():
    return PQ.Min()
def Size():
        return PQ.Size()
    def IsEmpty():
        return PQ.IsEmpty()
```

We could also increment c back when popping.

Observation: We could use the negative of the current system time instead of c (to avoid overflow). The implementation based on this can be given as:

```
    def Push(element):
        PQ.insert(-gettime(),element)
```

Problem-20 How do we implement Queue using heap?

Solution: To implement a queue using a priority queue PQ (using min heap), as similar to stacks simulation, let us assume that we are using one extra integer variable, c. Also, assume that c is initialized equal to any known value (e.g., 0). The implementation of the queue ADT is given below. Here the c is used as the priority while inserting/deleting the elements from PQ.

```
    def Push(element):
        PQ.Insert(c, element)
        c += 1
    def Pop():
        return PQ.DeleteMin()
    def Top():
        return PQ.Min()
    def Size():
        return PQ.Size()
    def IsEmpty() {
        return PQ.IsEmpty()
```

Note: We could also decrement c when popping.

Observation: We could use just the negative of the current system time instead of c (to avoid overflow). The implementation based on this can be given as:

```
    void Push(int element) {
        PQ.insert(gettime(),element);
    }
```

Note: The only change is that we need to take a positive c value instead of negative.

Problem-21 Given a big file containing billions of numbers, how can you find the 10 maximum numbers from that file?

Solution: Always remember that when you need to find max n elements, the best data structure to use is priority queues. One solution for this problem is to divide the data in sets of 1000 elements (let's say 1000) and make a heap of them, and then take 10 elements from each heap one by one. Finally heap sort all the sets of 10 elements and take the top 10 among those. But the problem in this approach is where to store 10 elements from each heap. That may require a large amount of memory as we have billions of numbers.

Reusing the top 10 elements (from the earlier heap) in subsequent elements can solve this problem. That means take the first block of 1000 elements and subsequent blocks of 990 elements each. Initially, Heapsort the first set of 1000 numbers, take max 10 elements, and mix them with 990 elements of the 2^{nd} set. Again, Heapsort these 1000 numbers (10 from the first set and 990 from the 2^{nd} set), take 10 max elements, and mix them with 990 elements of the 3^{rd} set. Repeat till the last set of 990 (or less) elements and take max 10 elements from the final heap. These 10 elements will be your answer.

Time Complexity: $O(n) = n/1000 \times$(complexity of Heapsort 1000 elements) Since complexity of heap sorting 1000 elements will be a constant so the $O(n) = n$ i.e. linear complexity.

Problem-22 **Merge k sorted lists with total of n elements:** We are given k sorted lists with total n inputs in all the lists. Give an algorithm to merge them into one single sorted list.

Solution: Since there are k equal size lists with a total of n elements, the size of each list is $\frac{n}{k}$. One simple way of solving this problem is:

- Take the first list and merge it with the second list. Since the size of each list is $\frac{n}{k}$, this step produces a sorted list with size $\frac{2n}{k}$. This is similar to merge sort logic. The time complexity of this step is: $\frac{2n}{k}$. This is because we need to scan all the elements of both the lists.

- Then, merge the second list output with the third list. As a result, this step produces a sorted list with size $\frac{3n}{k}$. The time complexity of this step is: $\frac{3n}{k}$. This is because we need to scan all the elements of both lists (one with size $\frac{2n}{k}$ and the other with size $\frac{n}{k}$).
- Continue this process until all the lists are merged to one list.

Total time complexity: $= \frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \cdots . \frac{kn}{k} = \sum_{i=2}^{n} \frac{in}{k} = \frac{n}{k} \sum_{i=2}^{n} i \approx \frac{n(k^2)}{k} \approx O(nk)$.
Space Complexity: $O(1)$.

Problem-23 For Problem-22, can we improve the time complexity?

Solution:

1 Divide the lists into pairs and merge them. That means, first take two lists at a time and merge them so that the total elements parsed for all lists is $O(n)$. This operation gives $k/2$ lists.
2 Repeat step-1 until the number of lists becomes one.

Time complexity: Step-1 executes $logk$ times and each operation parses all n elements in all the lists for making $k/2$ lists. For example, if we have 8 lists, then the first pass would make 4 lists by parsing all n elements. The second pass would make 2 lists by again parsing n elements and the third pass would give 1 list by again parsing n elements. As a result the total time complexity is $O(nlogn)$. Space Complexity: $O(n)$.

Problem-24 For Problem-23, can we improve the space complexity?

Solution: Let us use heaps for reducing the space complexity.

1. Build the max-heap with all the first elements from each list in $O(k)$.
2. In each step, extract the maximum element of the heap and add it at the end of the output.
3. Add the next element from the list of the one extracted. That means we need to select the next element of the list which contains the extracted element of the previous step.
4. Repeat step-2 and step-3 until all the elements are completed from all the lists.

Time Complexity = $O(nlogk)$. At a time we have k elements max-heap and for all n elements we have to read just the heap in $logk$ time, so total time = $O(nlogk)$.
Space Complexity: $O(k)$ [for Max-heap].

Problem-25 Given 2 arrays A and B each with n elements. Give an algorithm for finding largest n pairs $(A[i], B[j])$.

Solution:

Algorithm:

- Heapify A and B. This step takes $O(2n) \approx O(n)$.
- Then keep on deleting the elements from both the heaps. Each step takes $O(2logn) \approx O(logn)$.

Total Time complexity: $O(nlogn)$.

Problem-26 **Min-Max heap:** Give an algorithm that supports min and max in $O(1)$ time, insert, delete min, and delete max in $O(logn)$ time. That means, design a data structure which supports the following operations:

Operation	Complexity
Init	$O(n)$
Insert	$O(logn)$
FindMin	$O(1)$
FindMax	$O(1)$
DeleteMin	$O(logn)$
DeleteMax	$O(logn)$

Solution: This problem can be solved using two heaps. Let us say two heaps are: Minimum-Heap H_{min} and Maximum-Heap H_{max}. Also, assume that elements in both the arrays have mutual pointers. That means, an element in H_{min} will have a pointer to the same element in H_{max} and an element in H_{max} will have a pointer to the same element in H_{min}.

Init	Build H_{min} in $O(n)$ and H_{max} in $O(n)$
Insert(x)	Insert x to H_{min} in $O(logn)$. Insert x to H_{max} in $O(logn)$. Update the pointers in $O(1)$
FindMin()	Return root(H_{min}) in $O(1)$
FindMax	Return root(H_{max}) in $O(1)$
DeleteMin	Delete the minimum from H_{min} in $O(logn)$. Delete the same element from H_{max} by using the mutual pointer in $O(logn)$
DeleteMax	Delete the maximum from H_{max} in $O(logn)$. Delete the same element from H_{min} by using the mutual pointer in $O(logn)$

Problem-27 Dynamic median finding. Design a heap data structure that supports finding the median.

Solution: In a set of n elements, median is the middle element, such that the number of elements lesser than the median is equal to the number of elements larger than the median. If n is odd, we can find the median by sorting the set and taking the middle element. If n is even, the median is usually defined as the average of the two middle elements. This algorithm works even when some of the elements in the list are equal. For example, the median of the multiset {1, 1, 2, 3, 5} is 2, and the median of the multiset {1, 1, 2, 3, 5, 8} is 2.5.

"*Median heaps*" are the variant of heaps that give access to the median element. A median heap can be implemented using two heaps, each containing half the elements. One is a max-heap, containing the smallest elements; the other is a min-heap, containing the largest elements. The size of the max-heap may be equal to the size of the min-heap, if the total number of elements is even. In this case, the median is the average of the maximum element of the max-heap and the minimum element of the min-heap. If there is an odd number of elements, the max-heap will contain one more element than the min-heap. The median in this case is simply the maximum element of the max-heap.

Problem-28 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output**: An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]
Requirement: Find a good optimal way to get B[i]

Solution: Brute force solution is, every time the window is moved we can search for a total of w elements in the window. Time complexity: $O(nw)$.

Problem-29 For Problem-28, can we reduce the complexity?

Solution: Yes, we can use heap data structure. This reduces the time complexity to $O(nlogw)$. Insert operation takes $O(logw)$ time, where w is the size of the heap. However, getting the maximum value is cheap; it merely takes constant time as the maximum value is always kept in the root (head) of the heap. As the window slides to the right, some elements in the heap might not be valid anymore (range is outside of the current window). How should we remove them? We would need to be somewhat careful here. Since we only remove elements that are out of the window's range, we would need to keep track of the elements' indices too.

Problem-30 For Problem-28, can we further reduce the complexity?

Solution: Yes, The double-ended queue is the perfect data structure for this problem. It supports insertion/deletion from the front and back. The trick is to find a way such that the largest element in the window would always appear in the front of the queue. How would you maintain this requirement as you push and pop elements in and out of the queue?

Besides, you will notice that there are some redundant elements in the queue that we shouldn't even consider. For example, if the current queue has the elements: [10 5 3], and a new element in the window has the element 11. Now, we could have emptied the queue without considering elements 10, 5, and 3, and insert only element 11 into the queue.

Typically, most people try to maintain the queue size the same as the window's size. Try to break away from this thought and think out of the box. Removing redundant elements and storing only elements that need to be considered in the queue is the key to achieving the efficient $O(n)$ solution below. This is because each element in the list is being inserted and removed at most once. Therefore, the total number of insert + delete operations is $2n$.

```
from collections import deque
def MaxSlidingWindow(A, k):
    D = deque()
    res, i = [], 0
    for i in xrange(len(A)):
        while D and D[-1][0] <= A[i]:
            D.pop()
        D.append((A[i], i+k-1))
        if i >= k-1: res.append(D[0][0])
        if i == D[0][1]: D.popleft()
    return res

print MaxSlidingWindow([4, 3, 2, 1, 5, 7, 6, 8, 9], 3)
```

Problem-31 A priority queue is a list of items in which each item has associated with it a priority. Items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first. If the maximum priority item is required, then a heap is constructed such than priority of every node is greater than the priority of its children.

Design such a heap where the item with the middle priority is withdrawn first. If there are n items in the heap, then the number of items with the priority smaller than the middle priority is $\frac{n}{2}$ if n is odd, else $\frac{n}{2} \mp 1$.

Explain how withdraw and insert operations work, calculate their complexity, and how the data structure is constructed.

Solution: We can use one min heap and one max heap such that root of the min heap is larger than the root of the max heap. The size of the min heap should be equal or one less than the size of the max heap. So the middle element is always the root of the max heap.

For the insert operation, if the new item is less than the root of max heap, then insert it into the max heap; else insert it into the min heap. After the withdraw or insert operation, if the size of heaps are not as specified above than transfer the root element of the max heap to min heap or vice-versa.

With this implementation, insert and withdraw operation will be in O(*logn*) time.

Problem-32 Given two heaps, how do you merge (union) them?

Solution: Binary heap supports various operations quickly: Find-min, insert, decrease-key. If we have two min-heaps, H1 and H2, there is no efficient way to combine them into a single min-heap.

For solving this problem efficiently, we can use mergeable heaps. Mergeable heaps support efficient union operation. It is a data structure that supports the following operations:

- Create-Heap(): creates an empty heap
- Insert(H,X,K) : insert an item x with key K into a heap H
- Find-Min(H) : return item with min key
- Delete-Min(H) : return and remove
- Union(H1, H2) : merge heaps H1 and H2

Examples of mergeable heaps are:

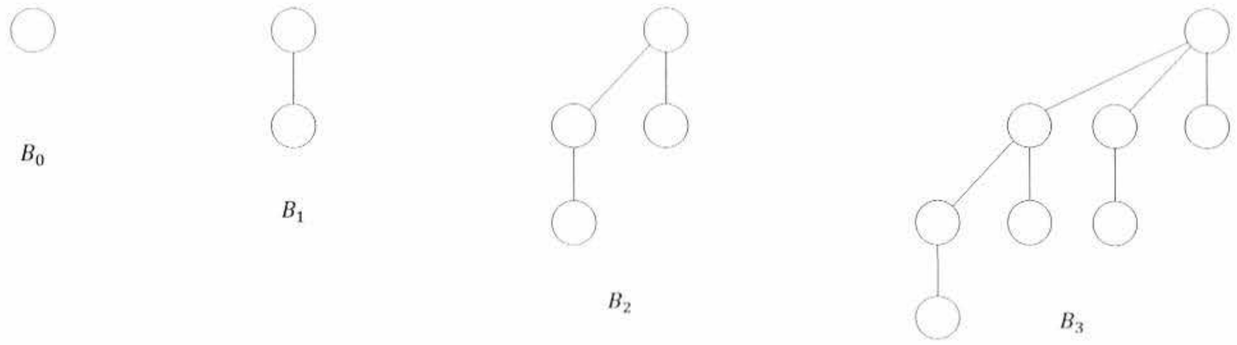
- Binomial Heaps
- Fibonacci Heaps

Both heaps also support:

- Decrease-Key(H,X,K): assign item Y with a smaller key K
- Delete(H,X) : remove item X

Binomial Heaps: Unlike binary heap which consists of a single tree, a *binomial* heap consists of a small set of component trees and no need to rebuild everything when union is performed. Each component tree is in a special format, called a *binomial tree*.

Example:



A binomial tree of order k, denoted by B_k is defined recursively as follows:

- B_0 is a tree with a single node
- For $k \geq 1$, B_k is formed by joining two B_{k-1}, such that the root of one tree becomes the leftmost child of the root of the other.

Fibonacci Heaps: Fibonacci heap is another example of mergeable heap. It has no good worst-case guarantee for any operation (except Insert/Create-Heap). Fibonacci Heaps have excellent amortized cost to perform each operation. Like *binomial* heap, *fibonacci* heap consists of a set of min-heap ordered component trees. However, unlike binomial heap, it has

- No limit on number of trees (up to O(n)), and
- No limit on height of a tree (up to O(n))

Also, *Find-Min, Delete-Min, Union, Decrease-Key, Delete* all have worst-case O(n) running time. However, in the amortized sense, each operation performs very quickly.

Operation	Binary Heap	Binomial Heap	Fibonacci Heap
Create-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Find-Min	$\Theta(1)$	$\Theta(logn)$	$\Theta(1)$
Delete-Min	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$
Insert	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
Delete	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$
Decrease-Key	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
Union	$\Theta(n)$	$\Theta(logn)$	$\Theta(1)$

Problem-33 Median in an infinite series of integers

Solution: Median is the middle number in a sorted list of numbers (if we have odd number of elements). If we have even number of elements, median is the average of two middle numbers in a sorted list of numbers.

We can solve this problem efficiently by using 2 heaps: One MaxHeap and one MinHeap.

1. MaxHeap contains the smallest half of the received integers
2. MinHeap contains the largest half of the received integers

The integers in MaxHeap are always less than or equal to the integers in MinHeap. Also, the number of elements in MaxHeap is either equal to or 1 more than the number of elements in the MinHeap.

In the stream if we get $2n$ elements (at any point of time), MaxHeap and MinHeap will both contain equal number of elements (in this case, n elements in each heap). Otherwise, if we have received $2n + 1$ elements, MaxHeap will contain $n + 1$ and MinHeap n.

Let us find the Median: If we have $2n + 1$ elements (odd), the Median of received elements will be the largest element in the MaxHeap (nothing but the root of MaxHeap). Otherwise, the Median of received elements will be the average of largest element in the MaxHeap (nothing but the root of MaxHeap) and smallest element in the MinHeap (nothing but the root of MinHeap). This can be calculated in O(1).

Inserting an element into heap can be done in O($logn$). Note that, any heap containing $n + 1$ elements might need one delete operation (and insertion to other heap) as well.

Example:

Insert 1: Insert to MaxHeap.
MaxHeap: {1}, MinHeap:{}

Insert 9: Insert to MinHeap. Since 9 is greater than 1 and MinHeap maintains the maximum elements.
MaxHeap: {1}, MinHeap:{9}

Insert 2: Insert MinHeap. Since 2 is less than all elements of MinHeap.
MaxHeap: {1,2}, MinHeap:{9}

Insert 0: Since MaxHeap already has more than half; we have to drop the max element from MaxHeap and insert it to MinHeap. So, we have to remove 2 and insert into MinHeap. With that it becomes:
MaxHeap: {1}, MinHeap:{2,9}
Now, insert 0 to MaxHeap.

Total Time Complexity: O($logn$).

```
class StreamMedian:
    def __init__(self):
            self.minHeap, self.maxHeap = [], []
            self.n=0
    def insert(self, num):
            if self.n%2==0:
                    heapq.heappush(self.maxHeap, -1*num)
                    self.n+=1
                    if len(self.minHeap)==0:
                            return
                    if -1*self.maxHeap[0]>self.minHeap[0]:
                            toMin=-1*heapq.heappop(self.maxHeap)
                            toMax=heapq.heappop(self.minHeap)
                            heapq.heappush(self.maxHeap, -1*toMax)
                            heapq.heappush(self.minHeap, toMin)
                    else:
                            toMin=-1*heapq.heappushpop(self.maxHeap, -1*num)
                            heapq.heappush(self.minHeap, toMin)
                            self.n +=1
    def getMedian(self):
            if self.n%2==0:
                    return (-1*self.maxHeap[0]+self.minHeap[0])/2.0
            else:
                    return -1*self.maxHeap[0]
```

Problem-34 Given a string inputStr and a string pattern, find the minimum window in inputStr which will contain all the characters in pattern in complexity $O(n)$. For example, inputStr = "XFDOYEZODEYXNZD" pattern = "XYZ" Minimum window is "XFDOYEZ". If there is no such window in inputStr that covers all characters in pattern, return the emtpy string "". If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in inputStr.

Solution:

```
def minWindowSubstr(inputStr, pattern):
    if inputStr == '' or pattern == '': return ''
    last_seen = {}
    start = 0; end = len(inputStr)-1
    pattern = set(pattern)
    # find such a substring ended at i-th character.
    for i, ch in enumerate(inputStr):
        if ch not in pattern: continue
        last_seen[ch] = i
       if len(last_seen) == len(pattern):
           # all chars have been seen
           first = min(last_seen.values()) #**We can use a priority queue, O(logn)
           if i-first+1 < end-start+1:
              start = first; end = i
    window = inputStr[start:end+1] if len(last_seen) == len(pattern) else ""
    #print window, len(window)
    return window
print minWindowSubstr("XFDOYEZODEYXNZD", "XYZF")
print minWindowSubstr("XXXYDFYFFHGKOXXFDOPPQDQPFVZZDEZ", "XZD")
print minWindowSubstr("XXXYYYY", "XY")
print minWindowSubstr("", "")
```

Time Complexity: $O(mlogn)$, where $m = len(inputStr)$ and $n = len(pattern)$.

Problem-35 Given a maxheap, give an algorithm to check whether the k^{th} largest item is greater than or equal to x. Your algorithm should run in time proportional to k.

Solution: If the key in the node is greater than or equal to x, recursively search both the left subtree and the right subtree. Stop when the number of node explored is equal to k (the answer is yes) or there are no more nodes to explore (no).

Problem-36 You have k lists of sorted integers. Find the smallest range that includes at least one number from each of the k lists.

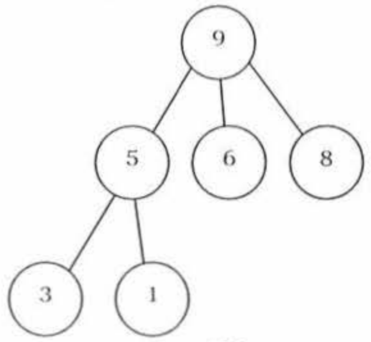
Solution:

```
import heapq
def KListsOneElementFromEach(Lst):
    heap = []
    end = False
    for l in Lst :
        thisRange = max(l) - min(l)
        heap.append(min(l))
        heapq.heapify(heap)
    while not end:
        elem = heapq.heappop(heap)
        print elem
        for l in Lst :
            if elem in l:
                #print l
                l.remove(elem)
                #print l
                if len(l) == 0:
                    end = True
                    break
                heapq.heappush(heap,l[0])
    print heap
def minL(l):
    m = min(float(s) for s in l)
    return m
def maxL(l):
    m = max(float(s) for s in l)
    return m
Lst = [[4, 10, 15, 24, 26],[0, 19, 12, 20],[15, 18, 28, 30],]
KListsOneElementFromEach(Lst)
```

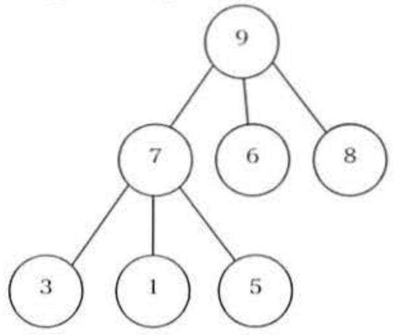
Problem-37 Suppose the elements 7, 2, 10 and 4 are inserted, in that order, into the valid 3-ary max heap found in the above question, Which one of the following is the sequence of items in the array representing the resultant heap?

(A) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4
(B) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
(C) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3
(D) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

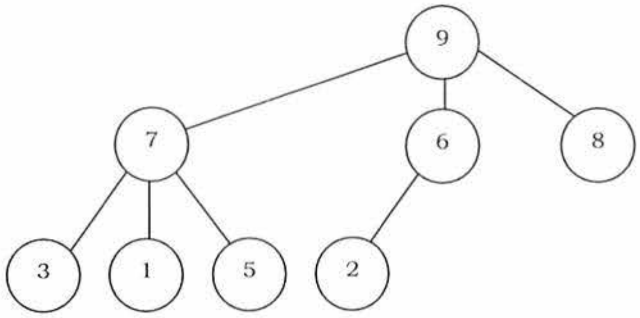
Solution: The 3-ary max heap with elements 9, 5, 6, 8, 3, 1 is:



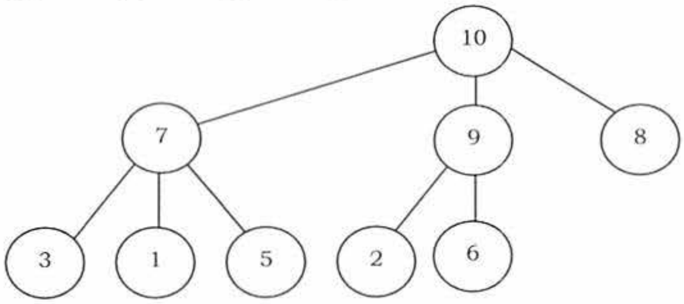
After Insertion of 7:



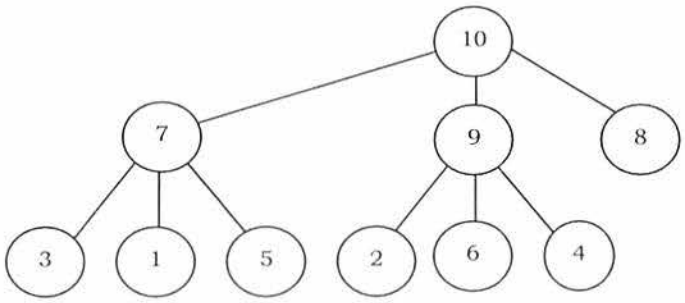
After Insertion of 2:



After Insertion of 10:



After Insertion of 4:



Problem-38 A complete binary min-heap is made by including each integer in [1,1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus, the root is at depth 0. The maximum depth at which integer 9 can appear is___

Solution: As shown in the figure below, for a given number i, we can fix the element i at i^{th} level and arrange the numbers 1 to $i-1$ to the levels above. Since the root is at depth *zero*, the maximum depth of the i^{th} element in a min-heap is $i-1$. Hence, the maximum depth at which integer 9 can appear is 8.