Priority Queues and Binary Heap Tree

Priority queues are useful for any application that involves processing elements not in the order they were received but based on some “priority”. Sometimes a printer shared by a number of computers is configured to always print smallest job in its queue first. Similarly in a multiuser environment, the operating system scheduler may decide to give time to the shortest job first, and take them off the queue. Priority queues also have applications in finding the kth smallest number in a given set of numbers.

A **binary heap tree** is very commonly used to implement a priority queue. Heaps have two properties, namely a *structure property* and a *heap-order property*. The need for the second property arises from the fact that we need to find the minimum item quickly.

Structure Property:

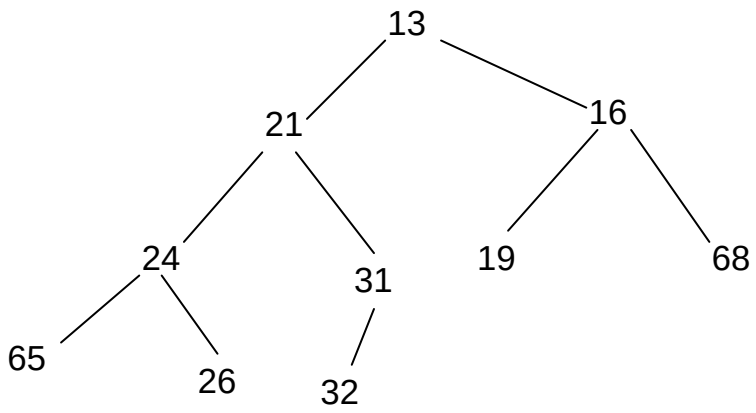
A heap is a binary tree that is completely filled, with the possible exception of the bottom level. Every level is filled from left to right. The height of a complete binary tree is log N. Since there are no nodes missing from such a tree, it is possible to represent it more coveniently using an array, instead of using the linked list structure.

If an array H[ ] is used to store a binary heap tree, then for an element with position H[i], the left child would be he element H[2i], and the right child would be H[2i+1]. Further, its parent would be the element H[i/2]. It is thus quite easy to traverse the tree.

Heap Order property:

The heap order property says that for every node in the tree, the value stored in that node is always less than the values stored in its child nodes. This implies that the root node will hold the smallest value.

Here is an example of a heap tree. More specifically it can be classified as a min-heap tree.



Similarly, we can construct a max-heap tree, in which the root holds the largest element.

The above heap tree can be implemented using an array as shown below

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
H[i]		13	21	16	24	31	19	68	65	26	32			

Note children of node 24 ,H[4] , are H[8] and H[9].

Parent of 68 H[7] is H[7/2] that is H[3].

Basic Heap Operations:

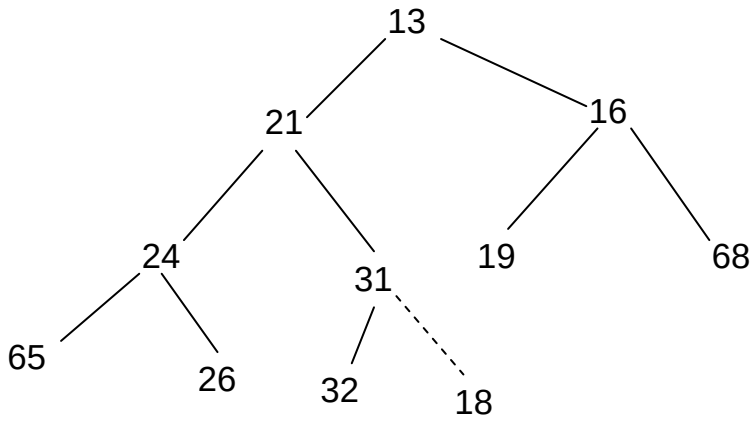
Insertion:

To start with, the new element to be inserted on the binary heap tree is placed next to the last element of the array containing the heap . This is called *the hole* position. Next it is moved up in the tree, to find its rightful place on the tree such that the heap order property is maintained. The reheapUp algorithm to do this is as follows:

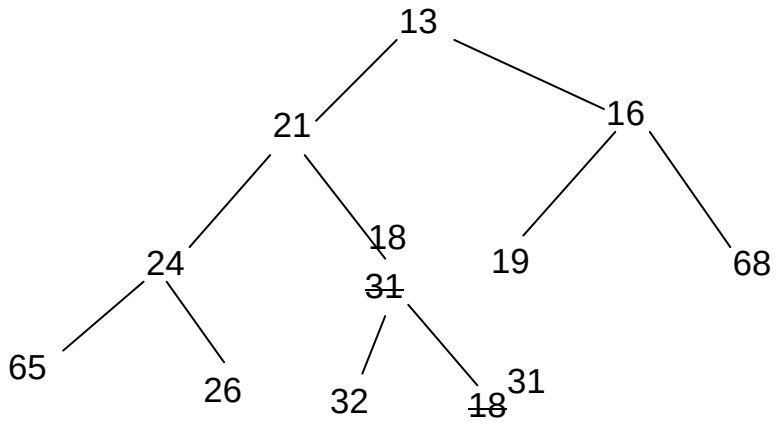
PercolateUp Algorithm

```
While (hole is not at the root        and     element < hole's parent) {
        Move hole's parent down
        Move hole up
}
Place element into final hole.
```

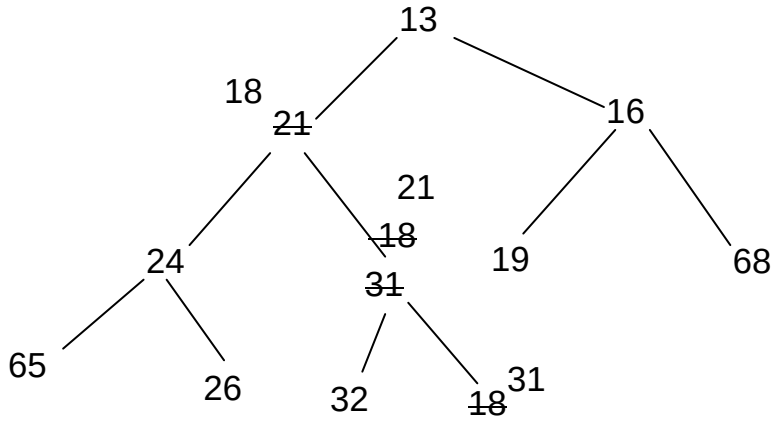
Example.
Let us insert the element 18 in the above binary heap tree. To start with place the element in a hole next to the last position that is in H[11].



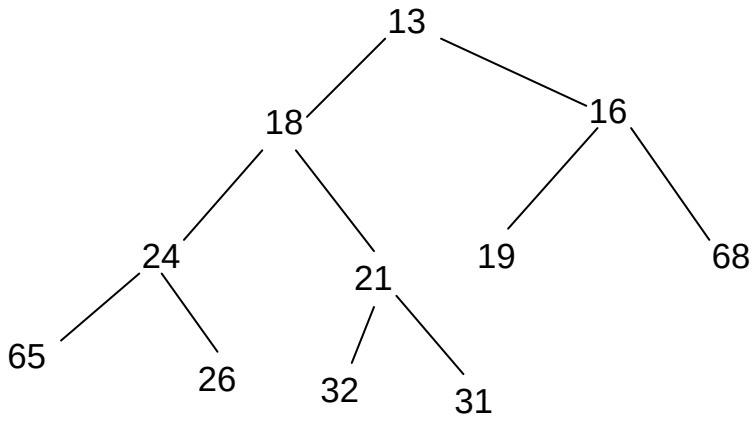
The value in the hole (18) is less than its parent value, so it is moved up and the parent is moved down



In the current heap tree, 18 is still not in right place as it is less than the value stored in its parent. Therefore the parent is moved down and the hole is moved up again to result in the following tree



Now the hole is in proper place, as the hole value is not less than the parent value. The insertion algorithm terminates here. The final tree is shown below:



It is easily seen that irrespective of the value to be inserted, the maximum hole movement is limited to the height of the heap tree, and thus the insertion takes O(log N) operations.

Creation:

The binary heap tree can be created by inserting each element in an empty tree following the above algorithm. The time complexity of the operations for every node is bounded by O(log i), which sums up to O(N) for N elements.

Deletion:

In a binary heap tree, we are not interested in removing a random element in the tree, but we want to delete the node with minimum value which is sitting at the root. Deleting the root node from a tree having N elements, leaves a hole at the root. Now the tree must be readjusted by moving the hole down. This is done by PercolateDown algorithm.

Delete_min( )

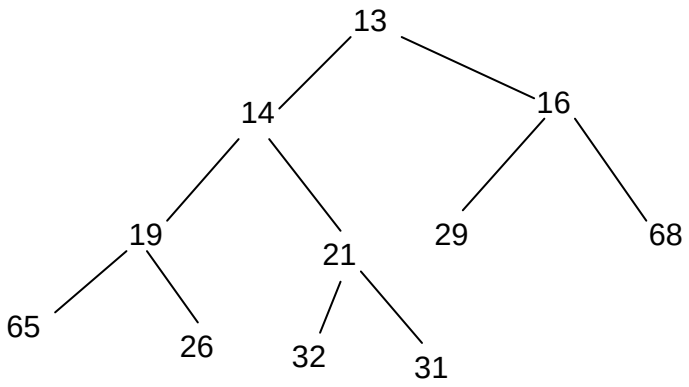
Put the last element in the root (That is H [1] = H [N--]);
PercolateDown (1);

PercolateDown algorithm:

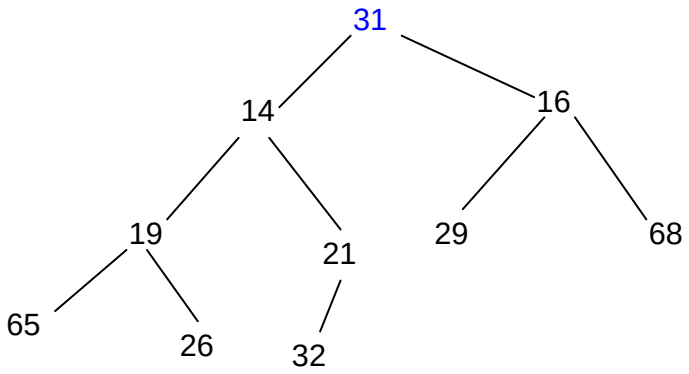
As long as current size is not exceeded
    If the hole is greater than its children,
        swap it with its smaller of the two children

The worst case running time of delete_min is O(log N) as the hole moves from the root to its rightful place in the tree and this movement is limited by the height of the tree.

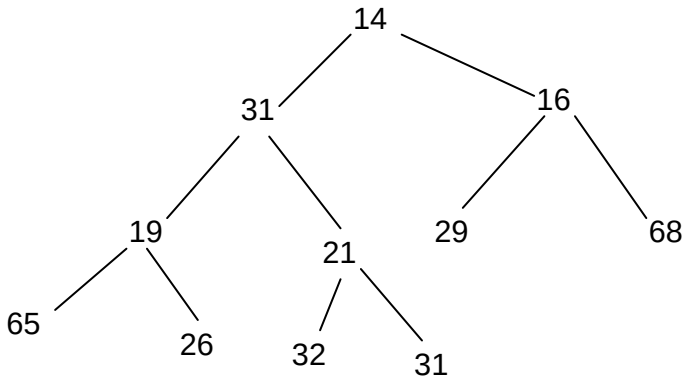
Example:
Consider the following heap on which we want to carry out delete_min operation.



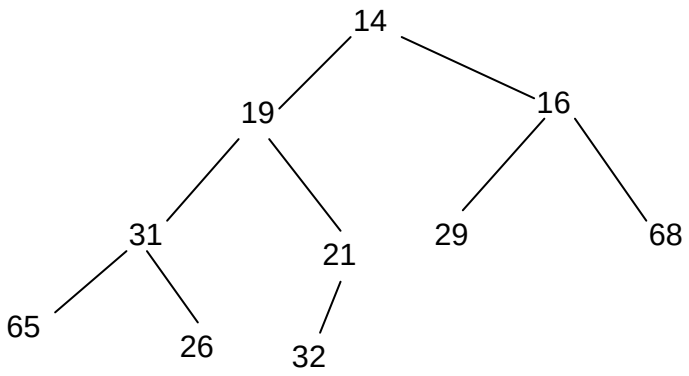
The root value 13 is removed leaving a hole. The hole is percolated down with the last element of the heap inside it.



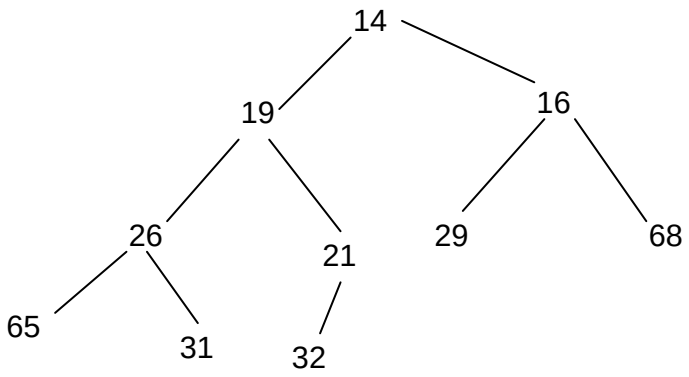
This violates the heap order. So hole is moved down.



This also violates the heap property. The hole is replaced by its smaller child node.



This also violates the heap property. The hole is moved further in the array.



The hole does not have children. The heap property is satisfied for this tree. The algorithm terminates at this point.

Application of Binary Heap in array sorting:

The array elements can be put on a heap tree, such as a min-heap tree. Now the smallest item is at the root node. Delete this value and swap it with last element of the array. As soon as it is deleted, the heap tree has the next smallest element as its root. This can be placed next to the smallest element placed earlier. The process can be repeated till every element has been deleted from the heap tree. Meanwhile the array will be organized automatically in descending order.

If it is desired to sort the array in increasing order, the same process as above can be repeated with the difference that now you make use of a MAX-HEAP tree instead of a min-heap tree. Now the largest element is at the root, which can be removed to the last position of the array and remaining elements put on the heap tree.