Priority Queues and Heaps

Priority Queue
A priority queue is a queue where each element has a priority and the element with the highest priority is at the front of the queue and will be the first element to be removed. To be contrasted with the *stack* and *queue* data structures which use LIFO and FIFO respectively.

It can be implemented as an unsorted array or a fully sorted array. Complexities are:

	remove()	insert()
Sorted array	O(1)	O(n)
Unsorted array	O(n)	O(1)

So either one has no particular advantage, still get an O(n) operation, i.e. an operation which requires n steps.

Binary Tree
A binary tree is a recursive structure where each tree node may have 2 child nodes (or left and right subtrees).

A complete binary tree is a binary tree where new nodes are added from left to right on the same level and a new level is only added when the current level is full. There are no 'gaps' anywhere in the tree between the first and last nodes.

Heap
A heap is another way to implement a priority queue. It has the logical structure of a complete binary tree which satisfies the heap condition.

A heap can be implemented using pointers (each node 3 pointers – parent, left child and right child) or much more simply using a partially ordered array. In the array implementation where the highest priority key is in array position 1 we have:

$$parent(i) = i/2$$
$$lchild(i) = 2i$$
$$rchild(i) = 2i+1$$

Here *parent(i)* is the array index of the parent of node i.

A heap is **partially ordered** by satisfying the heap condition.

A fully sorted or ordered array could be used, but this would require more computation to maintain. Heaps make for a much more efficient implementation of a priority queue. We will see later that:

	remove()	insert()
Sorted array	O(1)	O(n)
Unsorted array	O(n)	O(1)
Heap	$O(\log_2 n)$	$O(\log_2 n)$

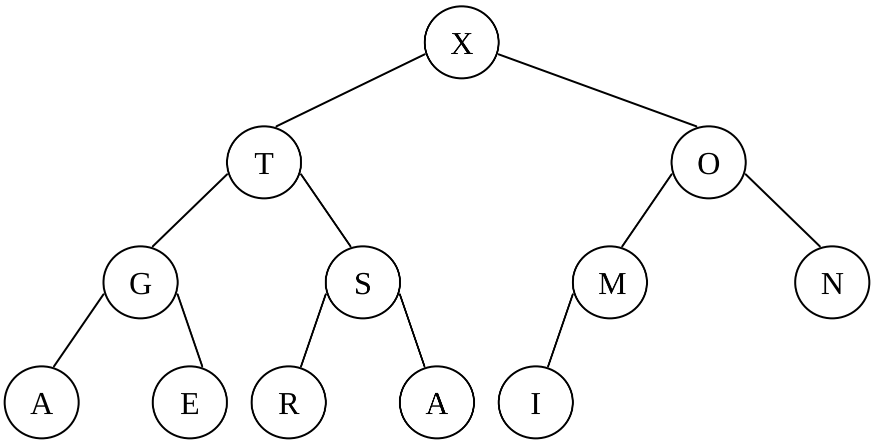
Heap Condition
Heap condition: the priority of any node is greater than or equal to that of its left child and right child if it has such child nodes.

Example

Heap array

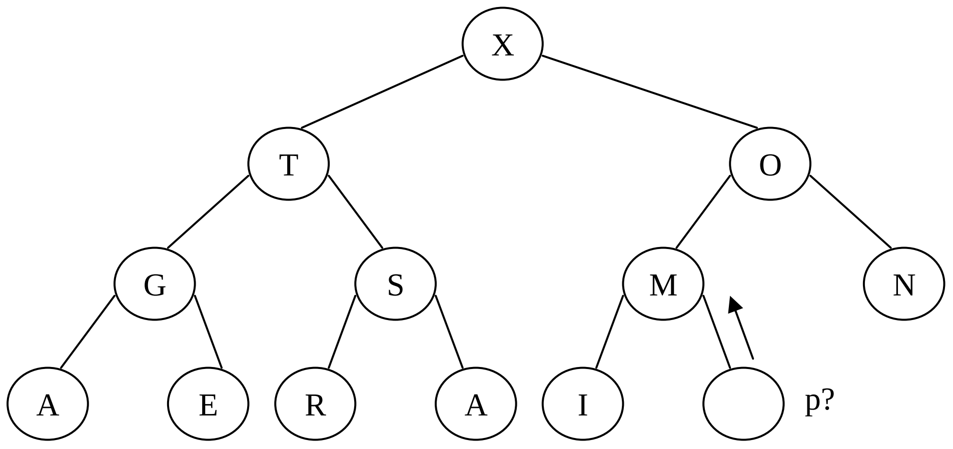
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a[k]		X	T	O	G	S	M	N	A	E	R	A	I		

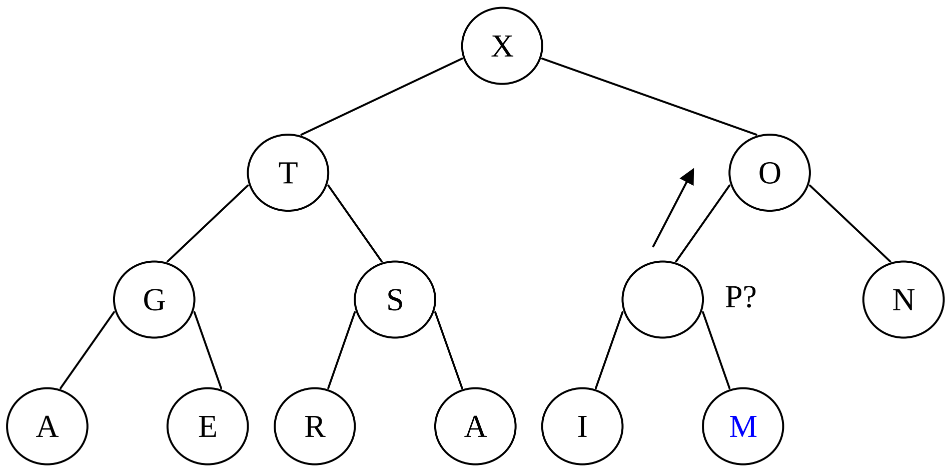
Tree structure

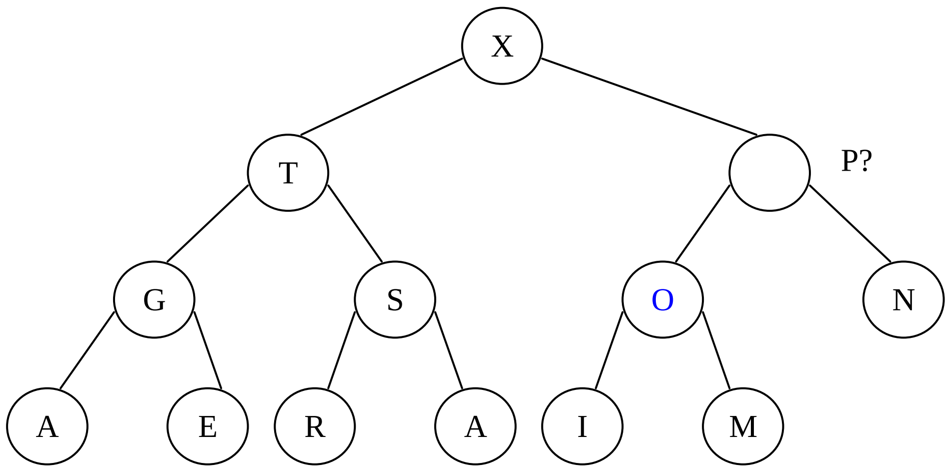


Inserting node with key P into heap

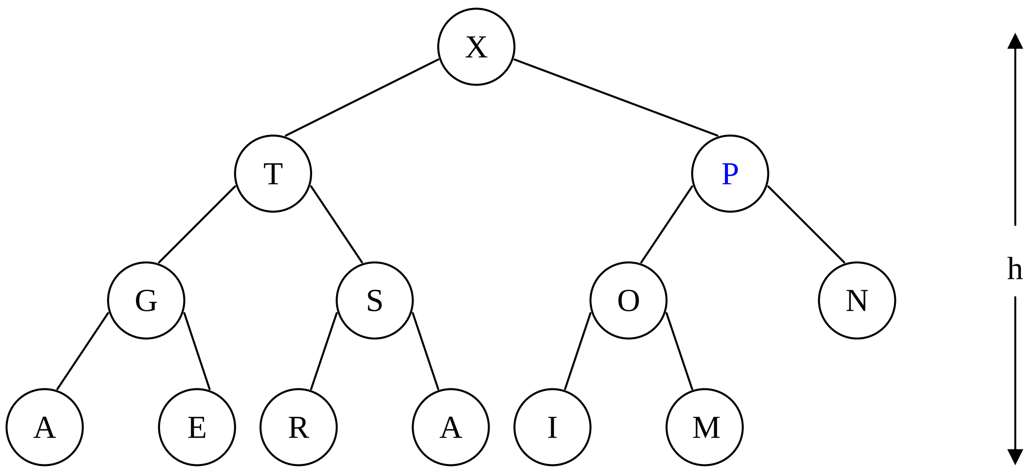
P added at bottom right of tree and sifted up to correct position.





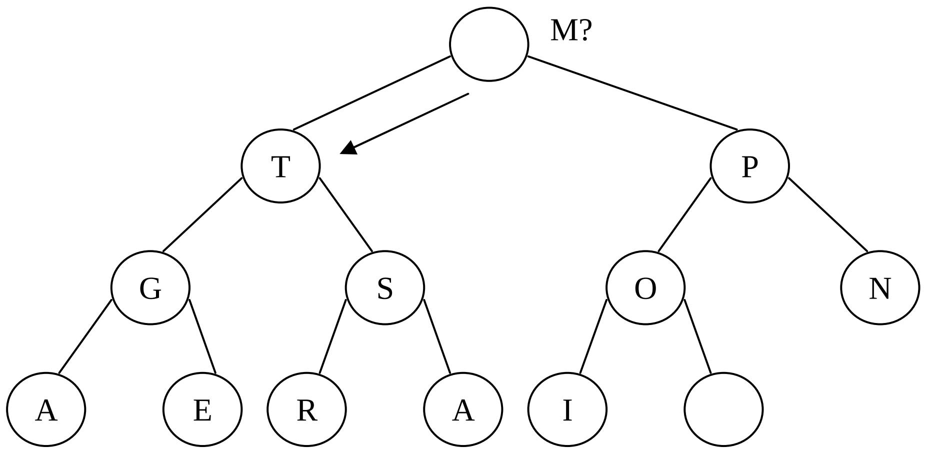


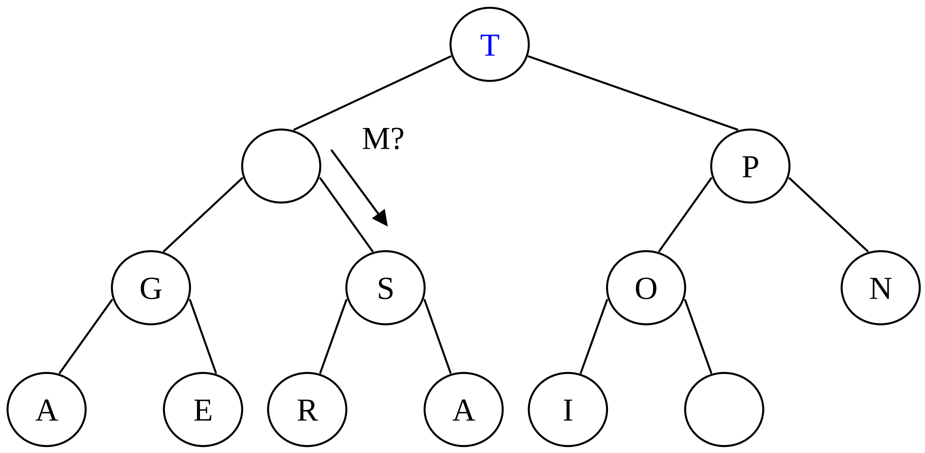
Finally you end up with:

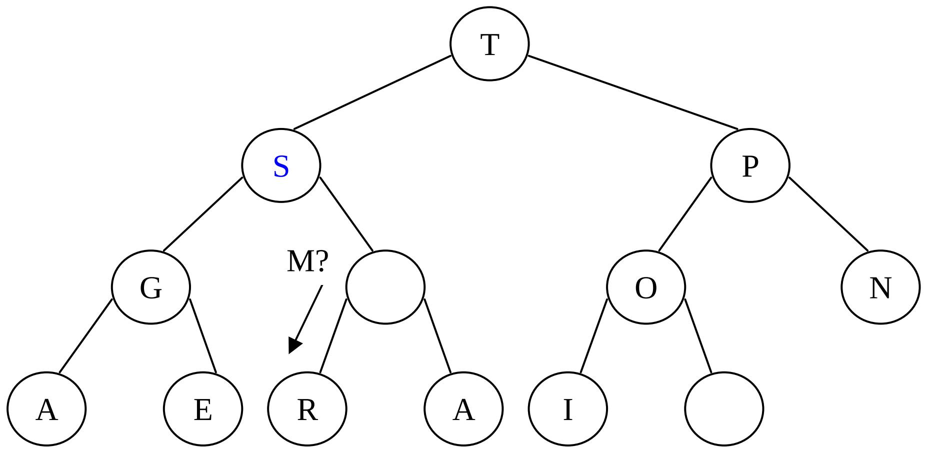


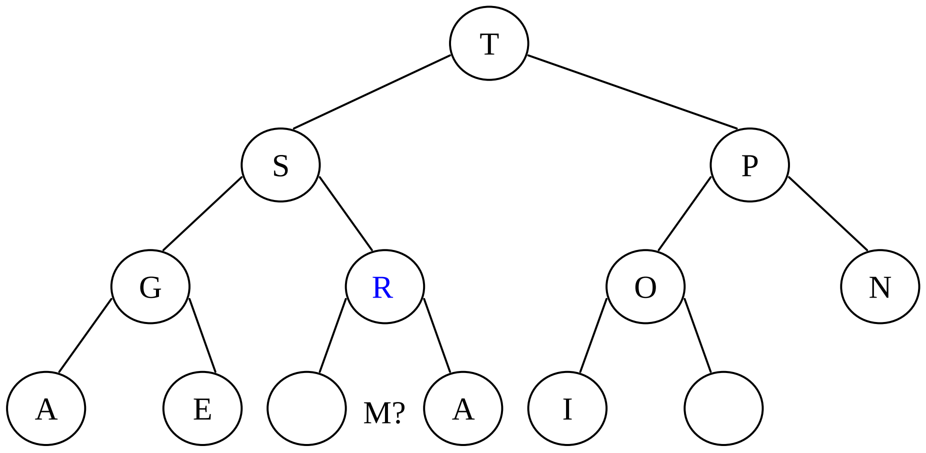
Removing largest value X from top of heap

X is removed, and the last value M in heap is then moved to top of heap and **sifted down**.

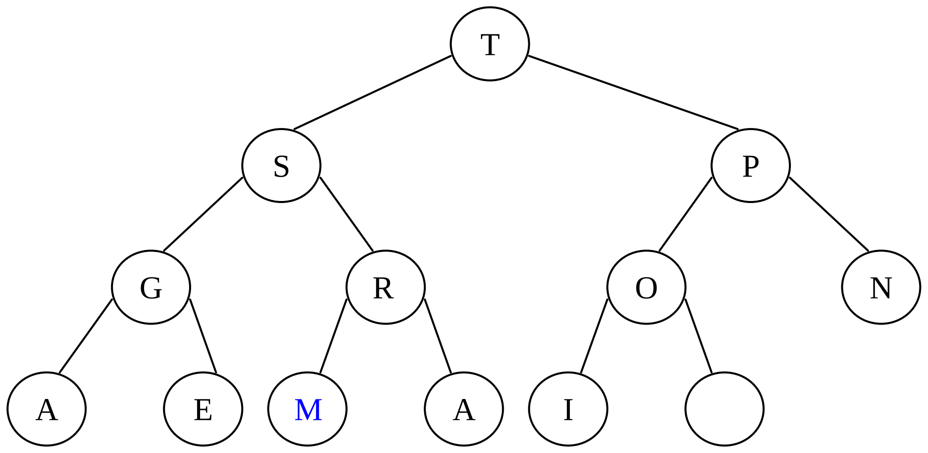








Finally M finds the right heap node to stay in.



Heap Operations in Pseudocode

```
Key          // a type, usually int, describes type of values in heap
int N        // number of elements in array or heap
Key h[ ]     // heap array of size N containing items of type Key.
```

The heap array *h[]* and *N* will be encapsulated inside the heap object.

```
insert(Key x)
    h[++N] = x
    siftUp( N)
```

```
// siftUp from position k. The key or node value at position k
// may be greater that that of its parent at k/2
// k is a position in the heap array h
siftUp( int k)
    v = h[k]
    h[0] = ∞
    while( v > h[k/2] )
        h[k] = h[k/2]
        k = k/2
    h[k] =  v
```

```
// Key of node at position k may be less than that of its
// children and may need to be moved down some levels
// k is a position in the heap array h
siftDown(int k)
    v = h[k]
    while( k ≤ N/2 )  // while node at pos k has a left child node
        j = 2k
        if( j < N ∧ h[j] < h[j+1])  ++j
        if( v ≥ h[j] ) break
        h[k] = h[j];  k = j
    h[k] =  v
```

```
Key remove( )
    v = h[1]
    h[1] = h[N--]
    siftDown( 1)
    return v
```

Complexity of siftUp, siftDown, insert and remove

The number of computational steps in these operations is proportional to number of iterations of the while loop of siftUp() or siftDown() which is at most equals the height *h* of the tree.

For a full binary tree with height h (h+1 levels) and N nodes we have:

$$N = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1$$

i.e. $N+1 = 2^{h+1}$, this implies $h+1 = \log_2(N+1)$
or

$$h = \log_2(N+1) - 1 \approx \log_2 N$$

Therefore

$$\boxed{\text{complexity} = O(h) = O(\log_2 N)}$$

Comment

In the above pseudocode we moved around the node key values or the contents of the heap. We also compared these key values against one another, e.g. h[j+1] > h[j], or in other words the heap key value also gave its priority. But in general we won't be comparing the heap node keys directly.

In a later application (TownHeap and MST algorithm) the heap keys will represent indices into an array or graph vertices which will also be used as indices into another array, and the priority of node k will be given by **`town[h[k]]->dist`** or for the graph application **`dist[ h[k]]`**.

k	-	heap position
h[k]	-	index in array town[] or array dist[] for graphs
town[h[k]]->dist	-	priority of heap node at position k in heap array