CS 112: Data Structures

Sesh Venugopal

Heap - Operations

Heap Structure and Function

From the structural point of view, the heap is a special type of binary tree.

From the functional point of view, it can be used either as a priority queue, which is a specialization of the regular FIFO queue we studied earlier, or as a structure for sorting.

Here we will study in detail the role of the heap as a priority queue.
(We will cover the sorting function later.)

Heap as Priority Queue

In the role of a *priority queue,* the heap acts as a data structure in which the items have different priorities of removal: the item with the highest priority is the one that will be removed next.

A FIFO queue may be considered a special case of a priority queue, in which the priority of an item is the time of its arrival in the queue:

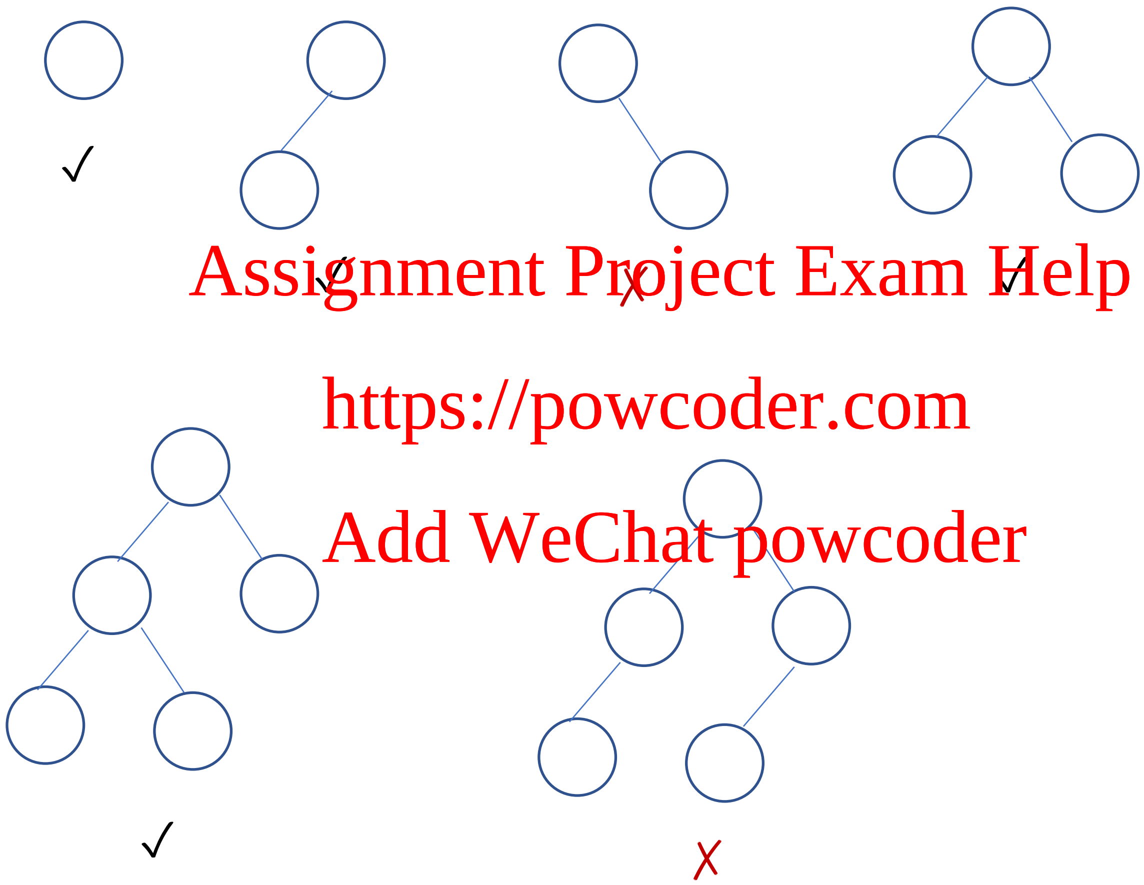
> the earlier the arrival time, the higher the priority, which means the item that arrived earliest is at the front of the queue.

A heap has two defining properties:

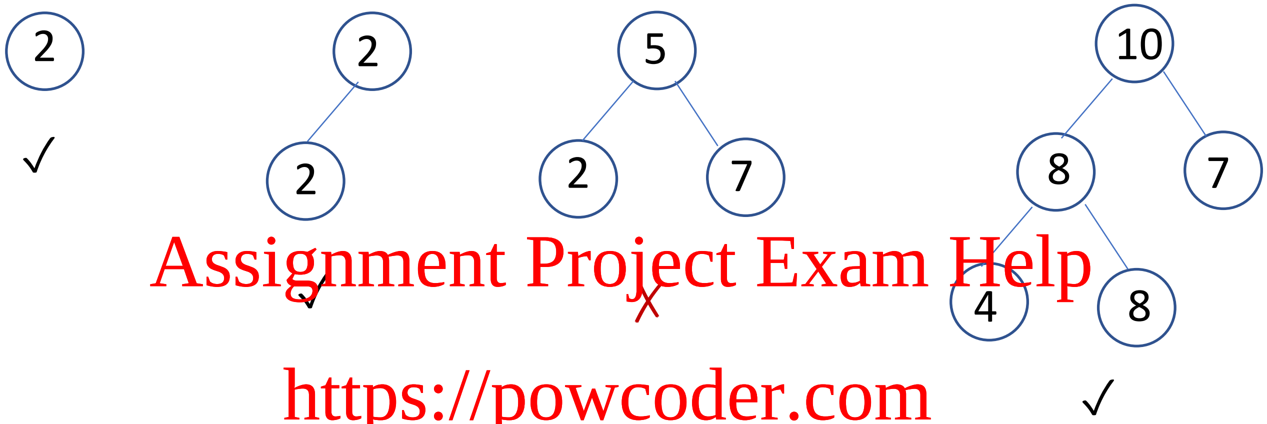
1. Structure: A heap is a complete binary tree, i.e. one in which every level but the last must have the maximum number of nodes possible at that level. The last level may have fewer than the maximum possible nodes, but they should be arranged from left to right without any empty spots.

2. Order: the key at any node *x* is greater than or equal to the keys at its children

Heap Structure



Heap Order



Heap Operations

As a priority queue, a heap must provide the same fundamental operations as a FIFO queue.

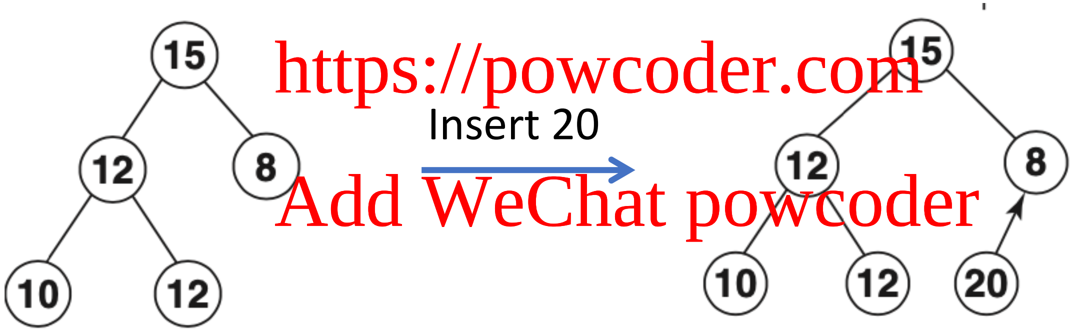
Specifically, it must provide an *insert* operation that inserts a new item in the heap—this new item must enter with a specified priority.

Also, it must provide a *delete* operation that removes the item at the front of the priority queue—this would be the item that has the highest priority of all in the heap, which is the item at the top of the heap.

Heap Insert

Inserting a new key in a heap must ensure that after insertion, both the heap structure and the heap order properties are satisfied.

Structure: First, insert the new key so that the heap structure property is satisfied, meaning that the new tree after insertion is also complete.



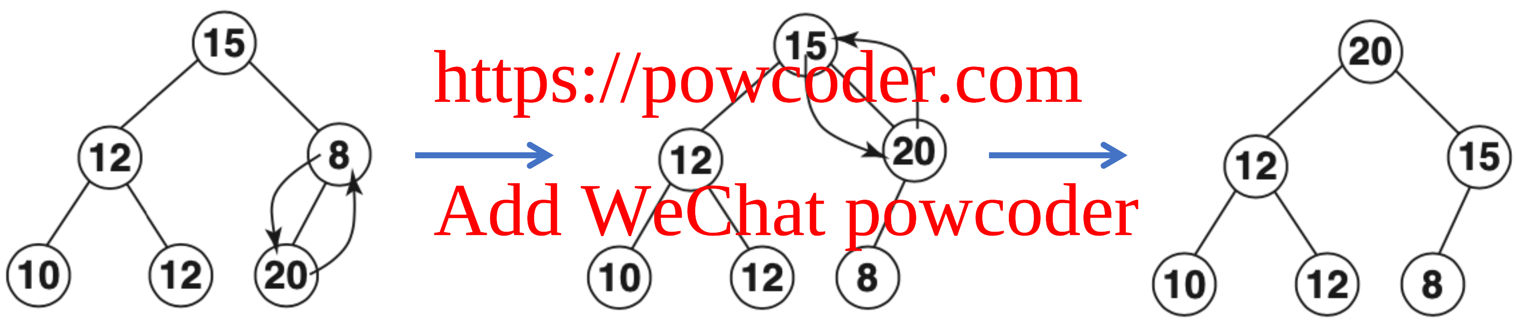
20 must be inserted as the left child of 8.

This is the only position for a new node that will ensure that the new heap is also a complete binary tree – the last level nodes are arranged left to right without gaps

Heap Insert

Inserting a new key in a heap must ensure that after insertion, both the heap structure and the heap order properties are satisfied.

Order: Second, make sure that the heap order property is satisfied by *sifting up* the newly inserted key.



1. 20 is compared with 8
2. Since it is larger, it is swapped with 8
1 comparison + 1 swap

1. 20 is compared with 15
2. Since it is larger, it is swapped with 15
1 comparison + 1 swap

Total: 2 comparisons + 2 swaps

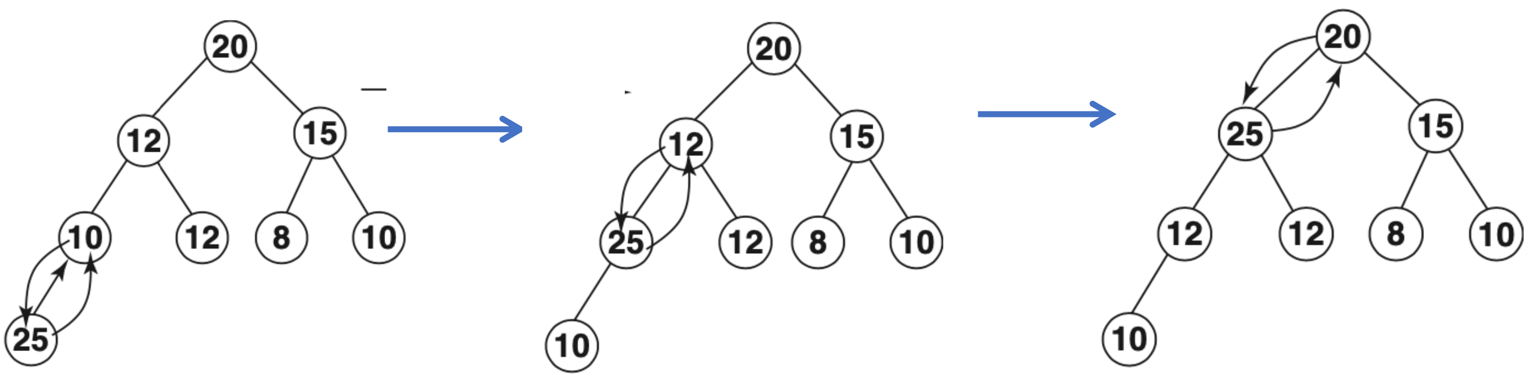
Heap Insert

Sift Up

Sifting up consists of comparing the new key with its parent, and swapping them if the new key is greater than its parent

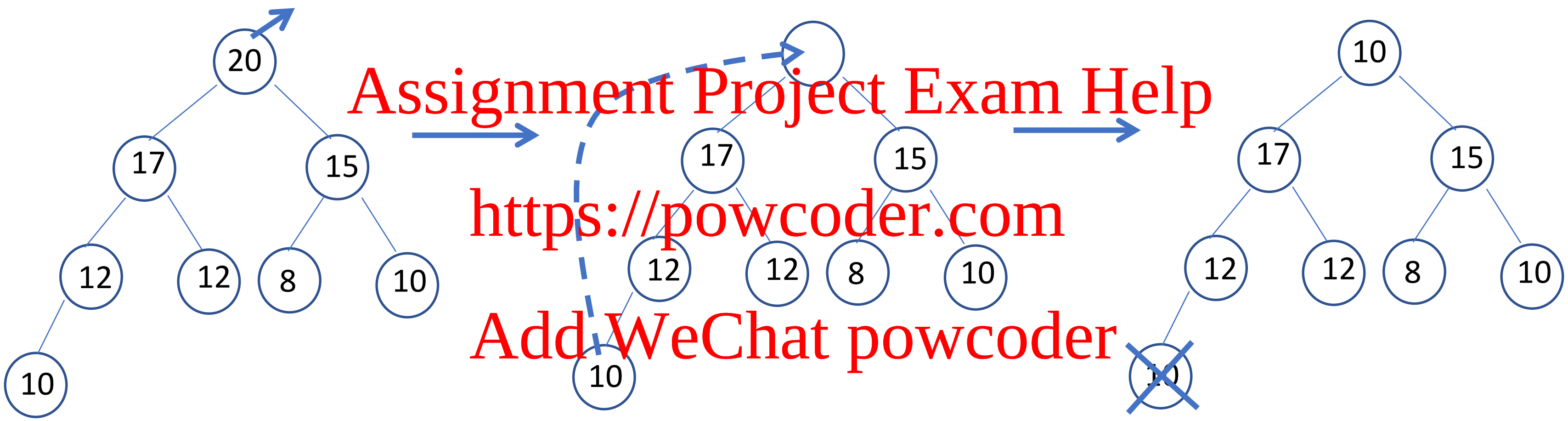
In the best case, a comparison is made but no swap is done – here 10 is inserted but it it is not greater than its parent, 15.
So 1 comparison, no swap

In the worst case, comparisons and swaps are done all the way up to the root:



Heap Delete

The item at the top of the heap is the one with the maximum key. Deletion removes this item from the heap. This leaves a vacant spot at the root, and the heap has to be *restored* so there is no vacant spot.



20 is deleted from the top

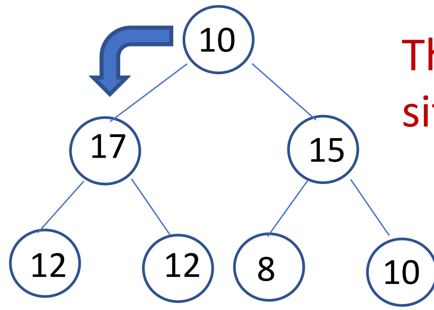
The vacant spot is filled in with the item at the "last" node

The last node is deleted

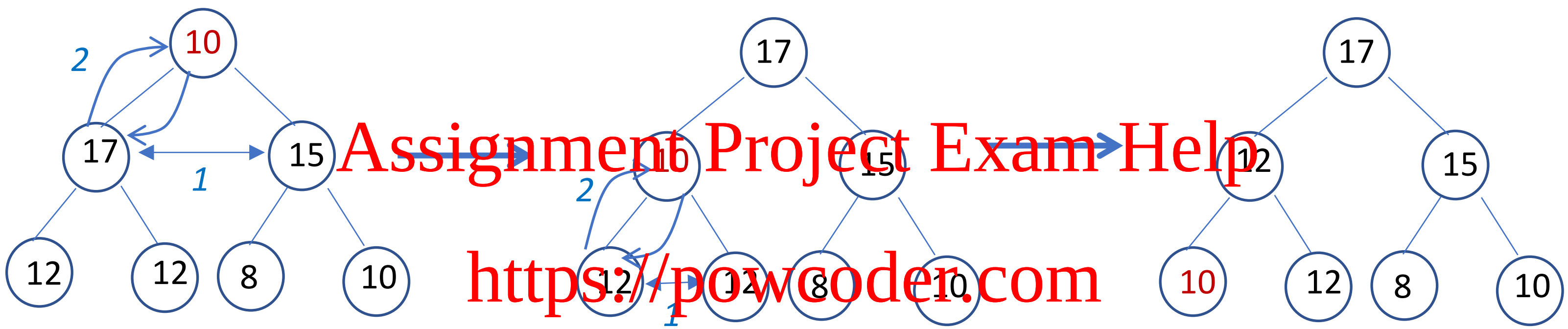
The sequence so far adjusts the heap structure so that there are no vacant spots, and there is one node less. But the heap order has to be restored as well, and this is done by sifting down the item at the top of the heap

Heap Delete

Sift Down



The item at the top of the heap is sifted down to restore the heap order



1. The children of 10 are compared to find the larger, which is 17
2 .17 is compared with 10 and since it is larger, it is swapped with 10
2 comparisons + 1 swap

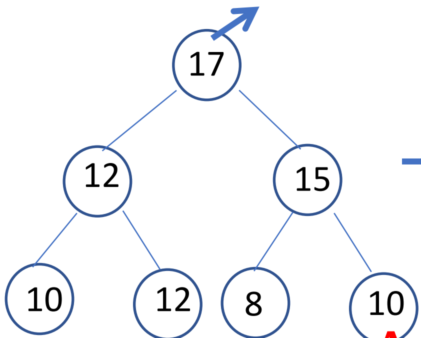
1. The children of 10 are compared to find the larger: it's a tie, so either of the 12's can be picked
2 .12 is compared with 10 and since it is larger, it is swapped with 10
2 comparisons + 1 swap

The heap is fully restored!

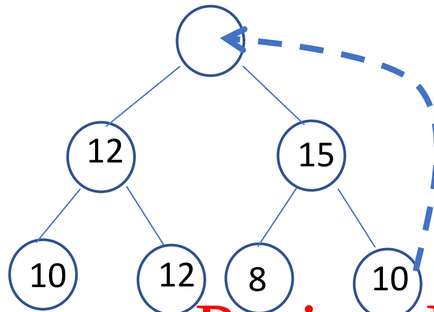
Total: 4 comparisons + 2 swaps

In the example, in the second iteration of sift down, the tie between the 12's was broken in favor of the left child 12. But we could equally well have broken the tie in favor of the right child 12, in which case, in the final heap, 10 would appear as the right child of the parent 12, instead of the left child

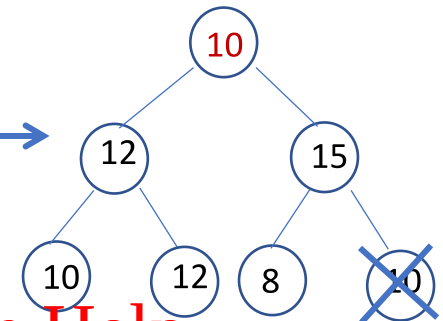
Heap Delete – Example 2



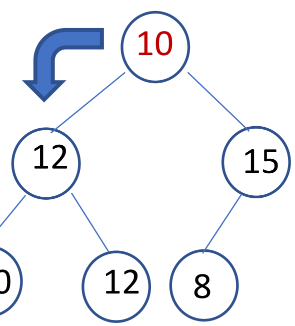
17 is deleted from the top



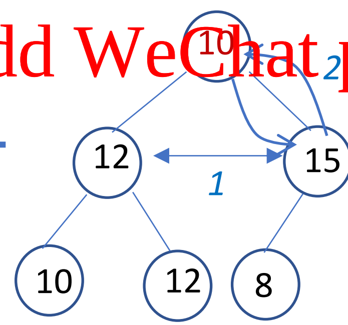
10 from last node is copied over into the top vacant spot



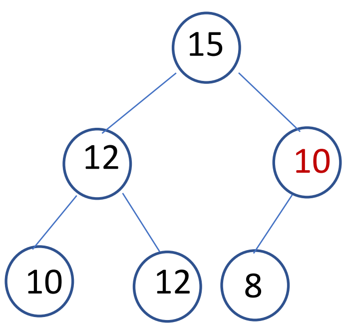
Last node 10 is deleted



10 is sifted down from the top



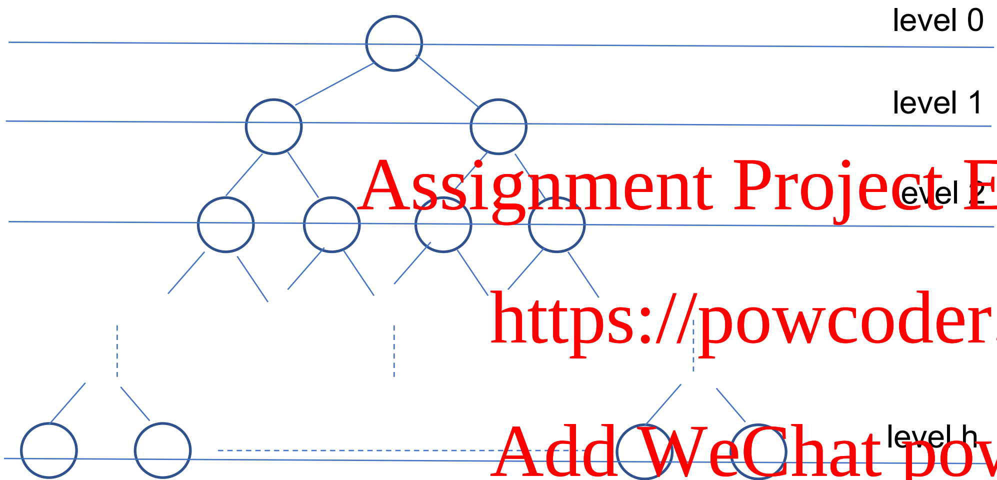
1. 12 and 15 are compared, and 15 being larger is picked
2. 15 is larger than 10, so it is swapped with 10
2 comparisons +1 swap



1. 10 has only one child, 8
2. 8 is not larger than 10, so no swap
1 comparison + no swap

Worst case Big O running time

Heap is a complete tree, so all levels except last must be full.
Let's assume that the last level is full as well – it won't make a difference for the big O result



Assume the height of the heap is *h*

The number of nodes/items, n, in this heap is computed by adding up the number of nodes at each level, and we can then write h in terms of n:

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

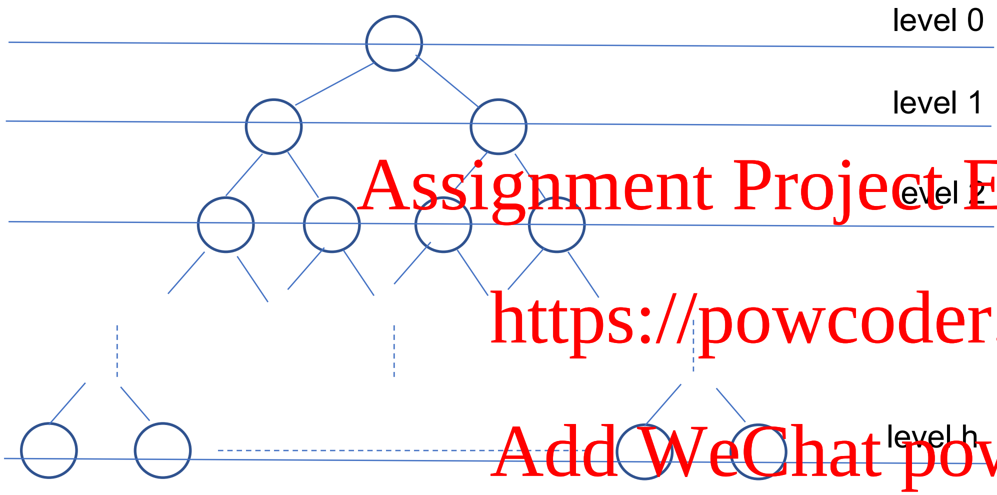
$$\Rightarrow \; n + 1 = 2^{h+1}$$

$$\Rightarrow \; h + 1 = \log_2(n+1)$$

$$\Rightarrow \; h = \log_2(n+1) - 1$$

Worst case Big O running time - insert

The actual insertion of a new node is $O(1)$ time.
The restoration of the heap order using sift up is where the real work is done.



Sifting up takes one comparison per level between the new key and its parent.

In the worst case, the new key may be sifted all the way up to the root, or h levels,

for a total of h comparisons.

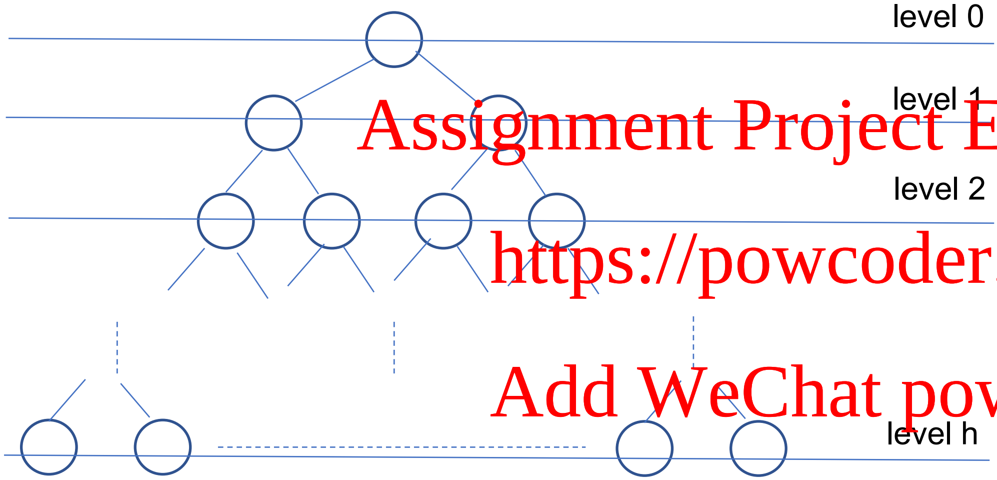
Since $h = \log_2(n+1) - 1$, the total number of comparisons is
$\log_2(n+1) - 1 \equiv O(\log n)$

The total time for insert is $O(1) + O(\log n) = O(\log n)$

(We are not counting swaps. Since a swap is only done when a comparison is done, the number of comparisons is a proxy for swaps as well, and it does not change the big O)

Worst case Big O running time - delete

Copying the item in the last node to the top node, and deleting the last node will take $O(1)$ time.
The restoration of the heap order using sift down is where the real work is done.



Sifting down each level requires 2 comparisons: one between the children, and one between the larger child and the parent

In the worst case, the top item may be sifted all the way up to the bottom, or h levels, for a total of $2*h$ comparisons.

Since $h = \log_2(n+1) - 1$, the total number of comparisons is

$2*(\log_2(n+1) - 1) \equiv O(\log n)$

The total time for delete is $O(1) + O(\log n) = O(\log n)$

(We are not counting swaps. Since a swap is only done when a comparison is done, the number of comparisons is a proxy for swaps as well, and it does not change the big O)