DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Heap: Definition and Implementation

Shylaja S S

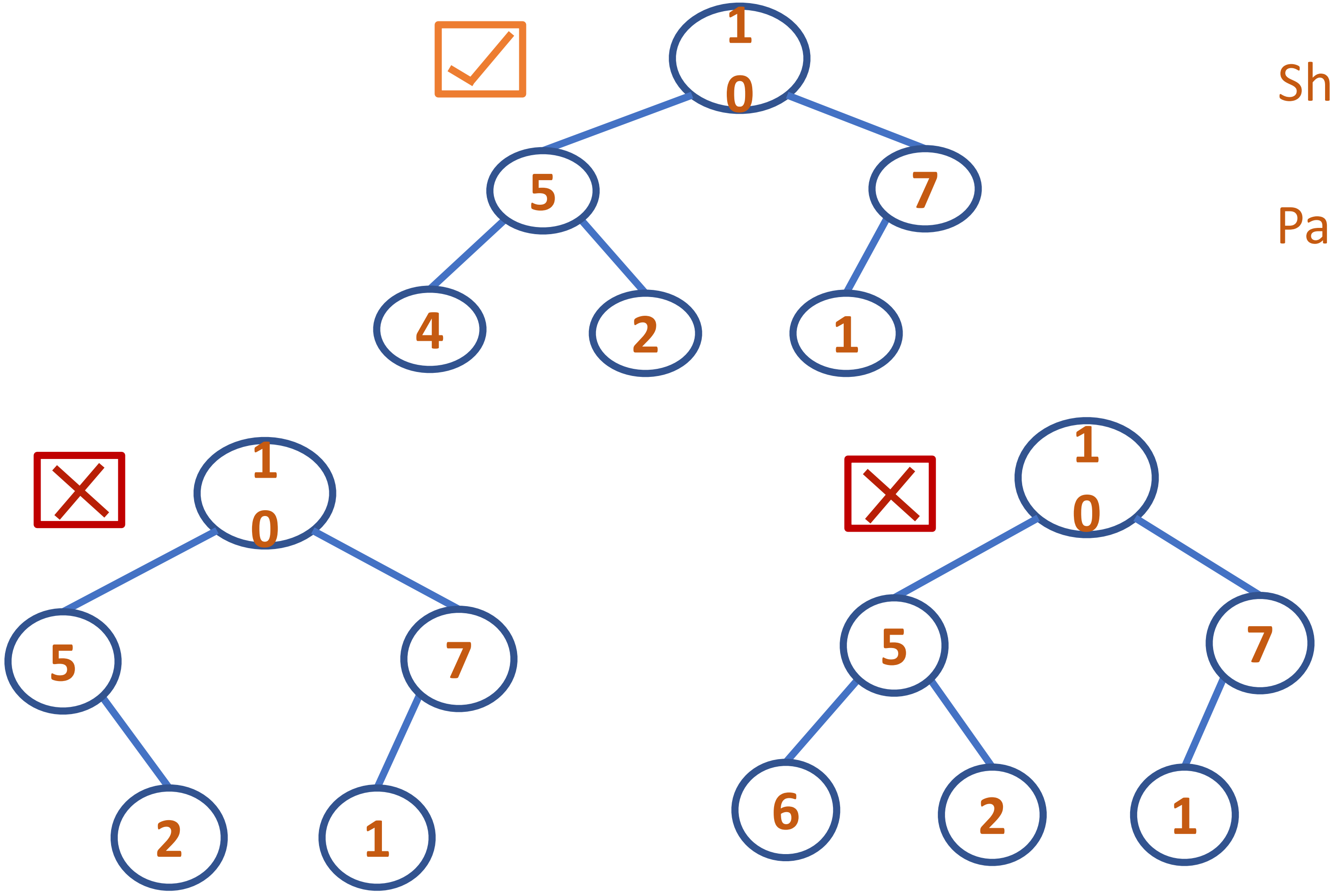
Department of Computer Science & Engineering

Heap Tree

Definition: A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

1. **The tree's shape requirement** - The binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing
2. **The parental dominance requirement** - The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)

Heap Tree



Shape Requirement not satisfied

Parental dominance not satisfied

Only the topmost Binary Tree is a heap. Why?

Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$

2. The root of a heap always contains its largest element

3. A node of a heap considered with all its descendants is also a heap

4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap.

Properties of Heap

...

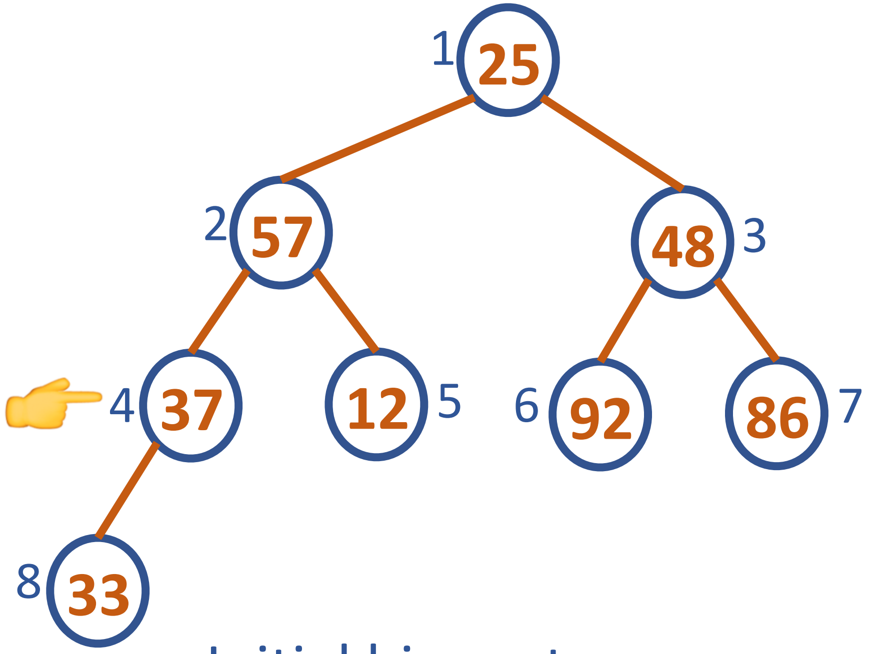
In such a representation,

a) The parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions

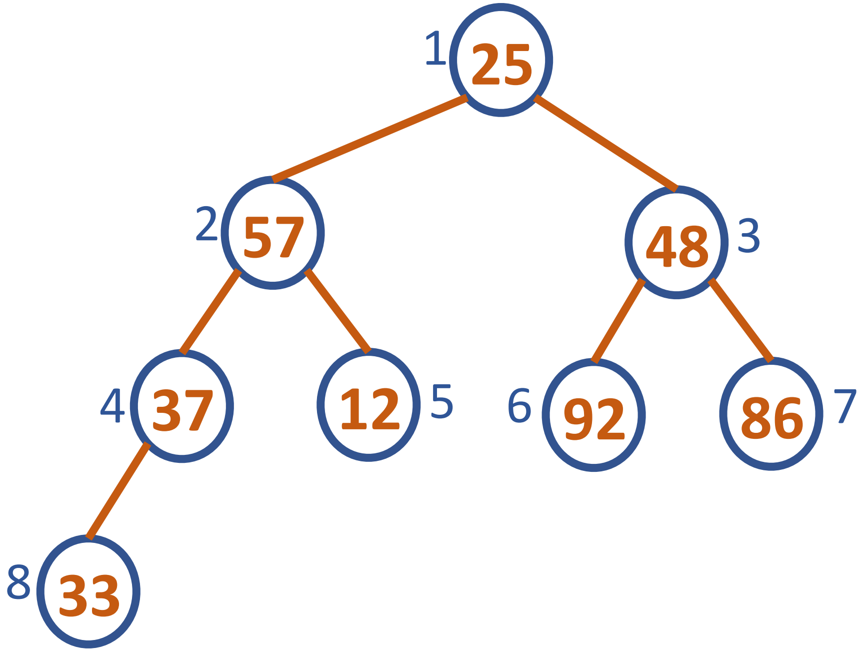
b) The children of a key in the array's parental position i ($1 <= i <= \lfloor n/2 \rfloor$) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i ($2 <= i <= n$) will be in position $\lfloor n/2 \rfloor$

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



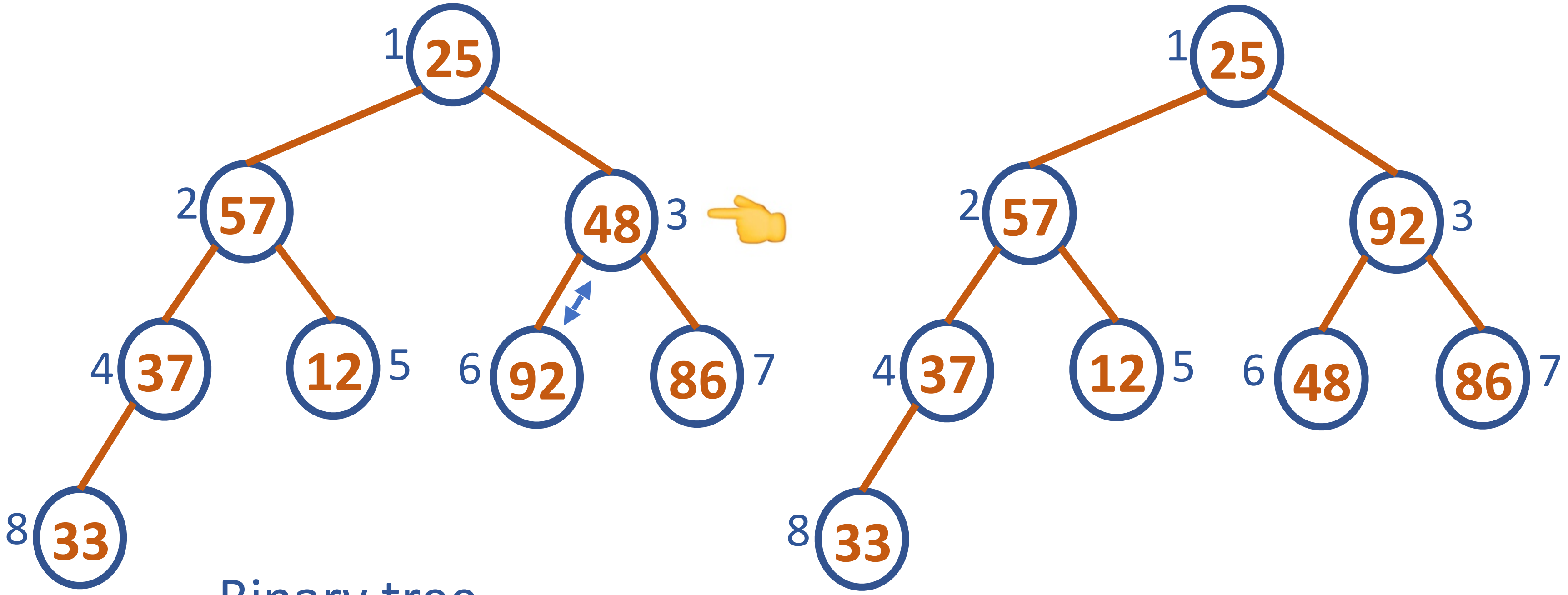
Initial binary tree



At k = 4, v = 37
Compare 37 with its only child 33
37 > 33, it’s a heap at k=4

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



Binary tree
after one iteration at k=4

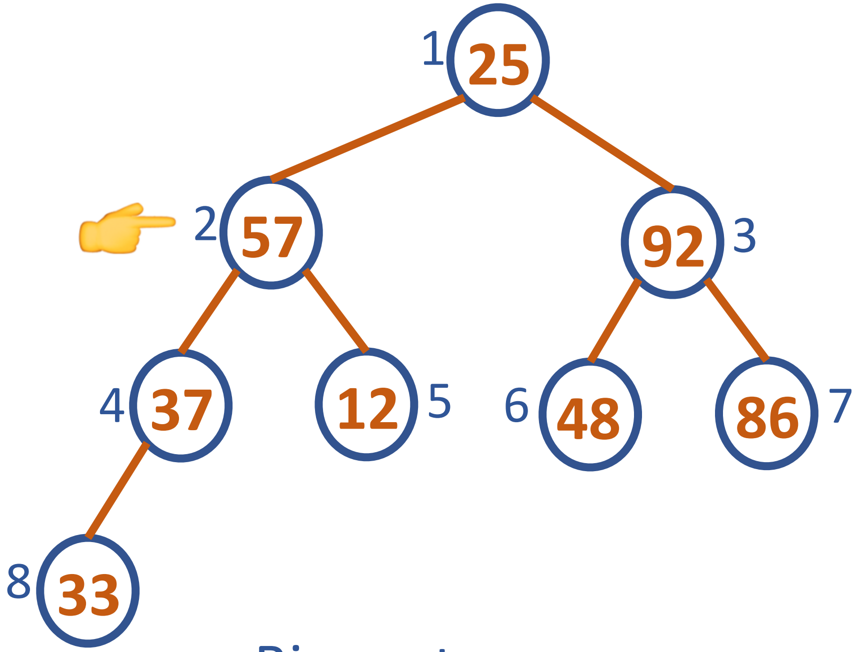
At k = 3, v = 48

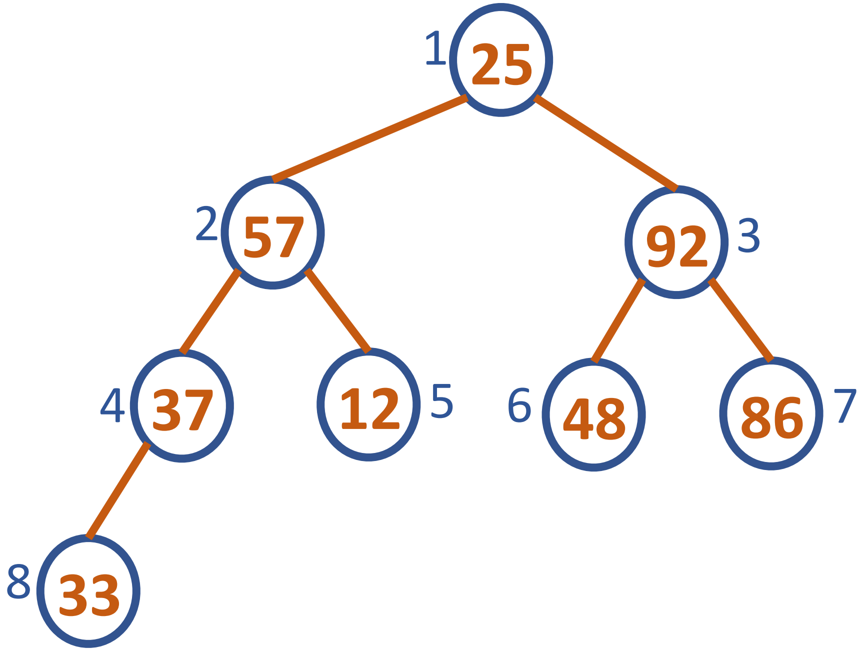
Largest child: 92

Compare 48 with its largest child

48 < 92, Heapify

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8





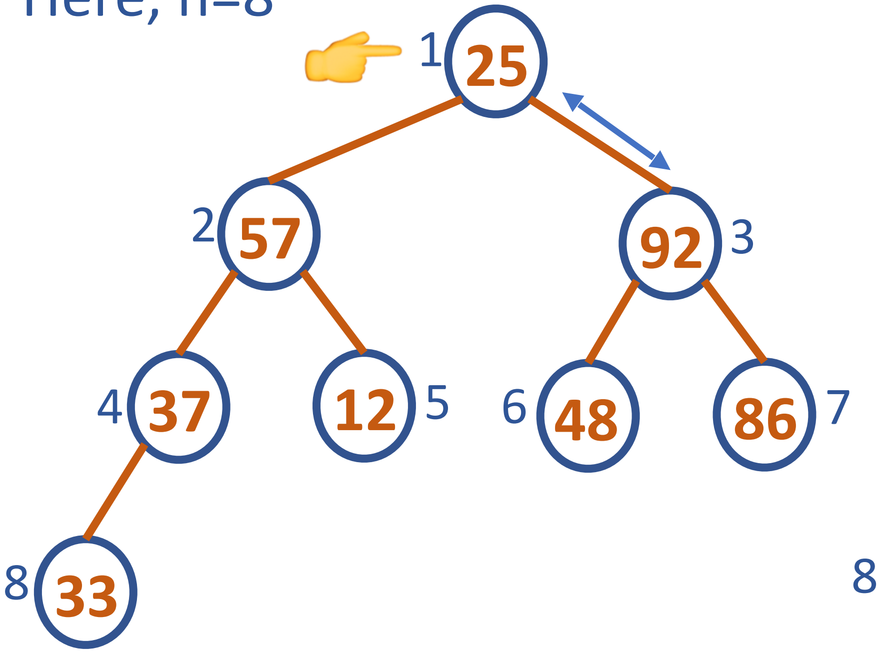
Binary tree
after two iterations at k=4, k=3

At k = 2, v = 57
Largest child: 37
Compare 57 with its largest child
57 > 37, it’s a heap at k=2

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

Here, n=8



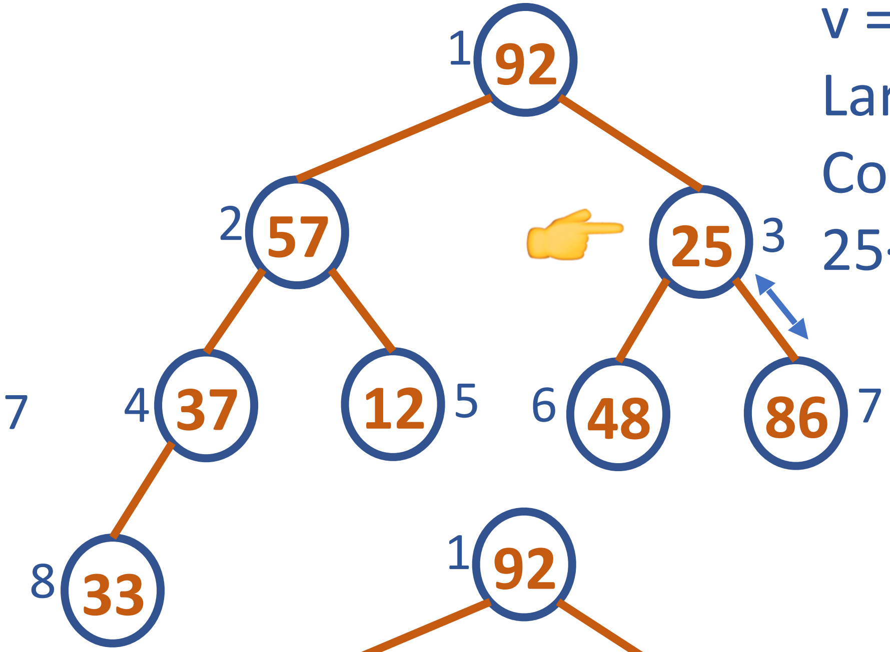
Binary tree after three iterations at k=4, k=3, k=2

At k = 1, v = 25

Largest child: 92

Compare 25 with its largest child

25 < 92, Heapify

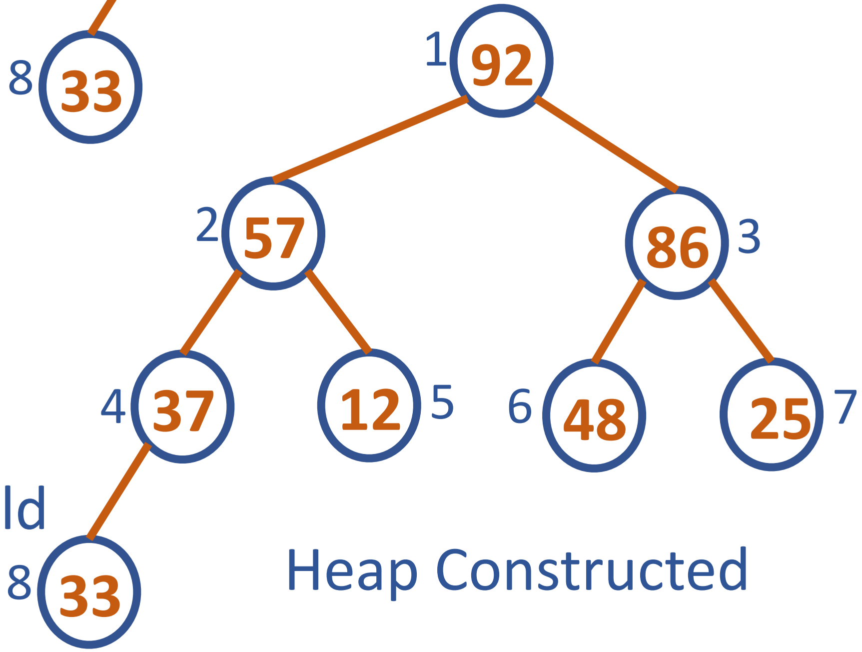


v = 25, Now at k = 3,

Largest child: 86

Compare 25 with its largest child

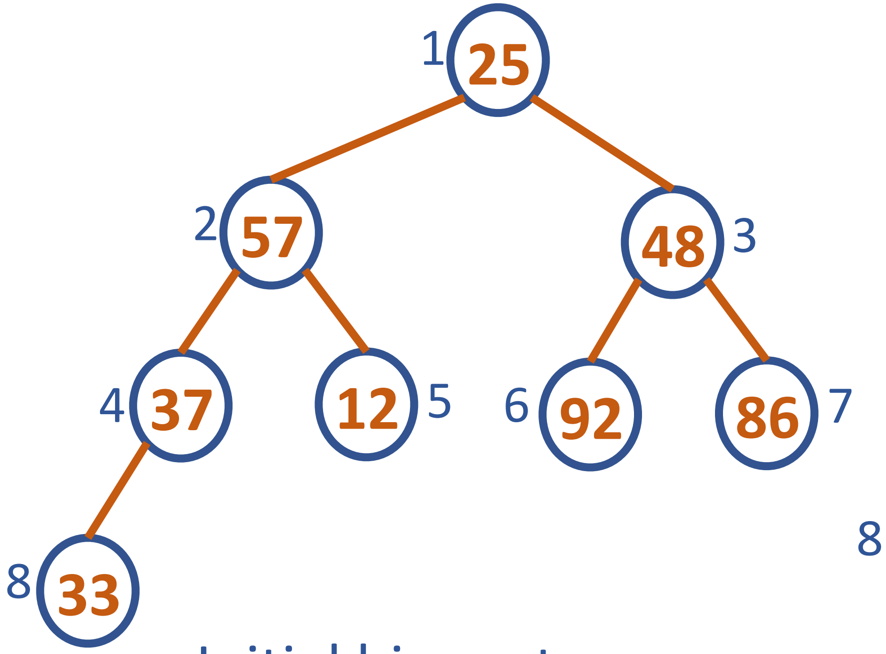
25<86, Heapify



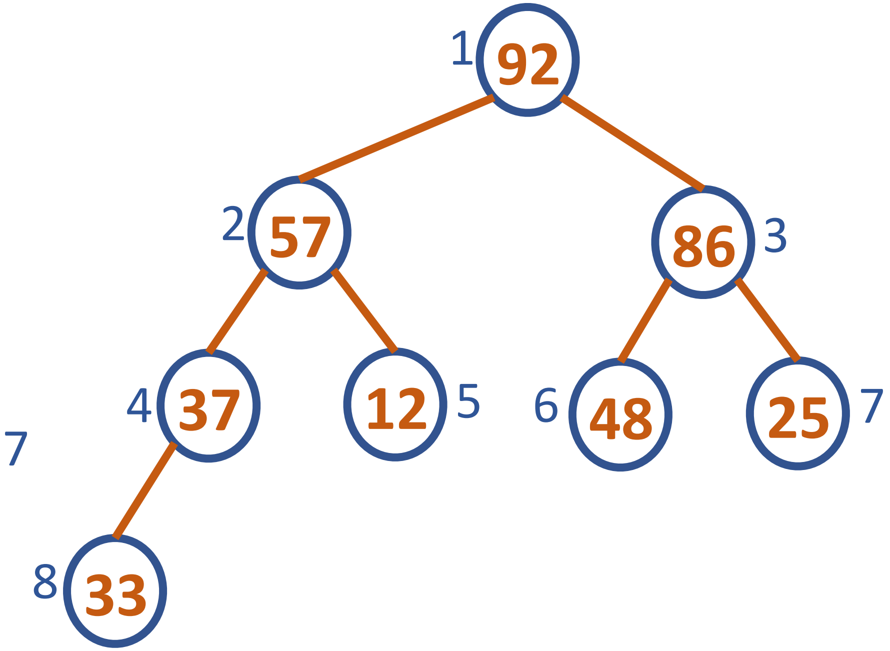
Heap Constructed

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



Initial binary tree



Bottom Up Heap Constructed

Heap Construction – Bottom Up

```
ALGORITHM HeapBottomUp(H[1…n])
//Constructs a heap from the elements of a given array by bottom-up algorithm
//Input: An array H[1…n] of orderable items
//Output: A heap H[1…n]
for i ←  ⌊n/2⌋ downto 1 {
    k ← i
    v ← H[k]
    heap ← false
    while not heap and 2*k ≤ n {
        j ← 2*k
        if j < n                    //if there are two children
            if H[j] < H[j+1]
                j ← j+1                     //find position of largest child
        if v ≥ H[j]         //if key of parent node ≥ key of largest child
            heap ← true        //it's a heap
        else {                      //heapify
                H[k] ← H[j]
                k ← j
        }       //end of else
    }   //end of while
    H[k] ← v
}   //end of for
```

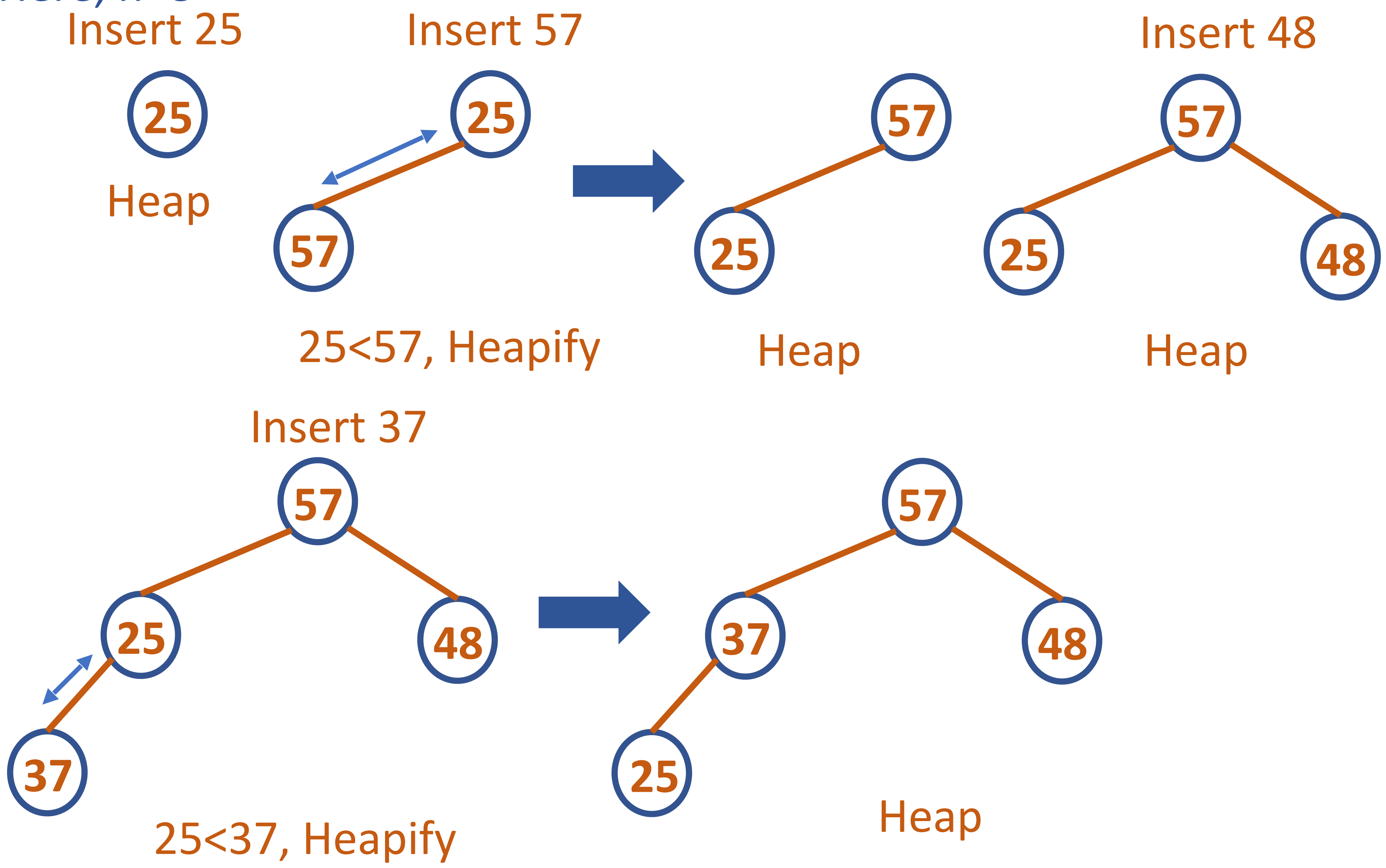
Efficiency

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{level\ i\ keys} 2(h-i)$$
$$= \sum_{i=0}^{h-1} 2(h-i)2^i$$
$$= 2(n - \log_2(n+1))$$

Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

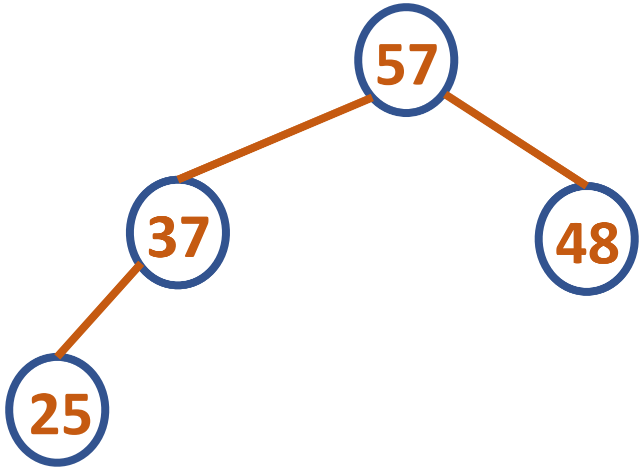
Here, n=8



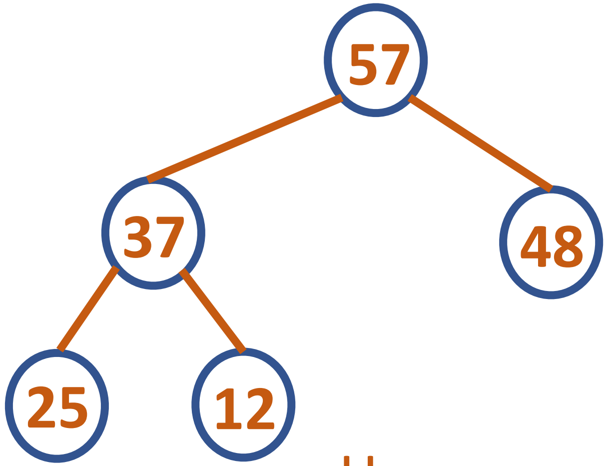
Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

Here, n=8

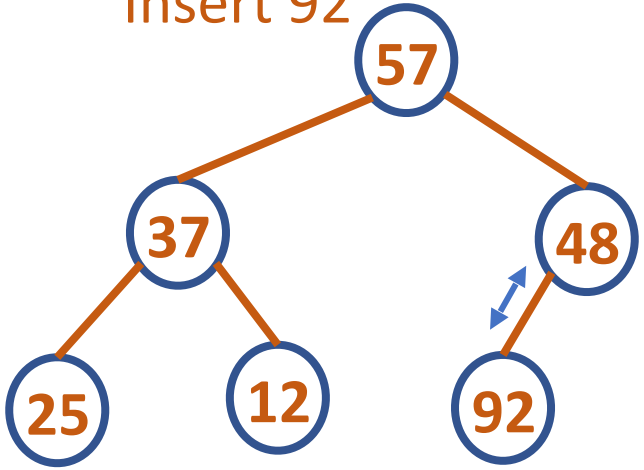


Insert 12

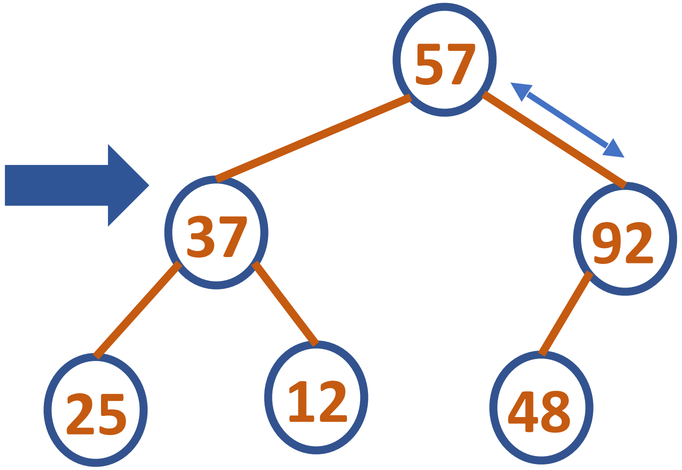


Heap

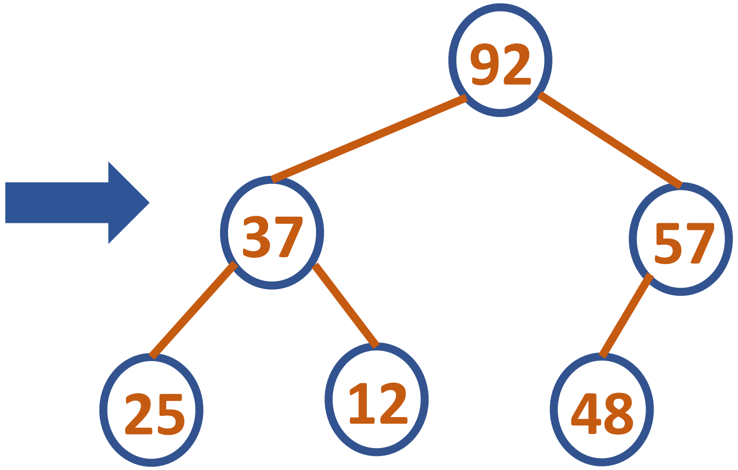
Insert 92



48<92, Heapify

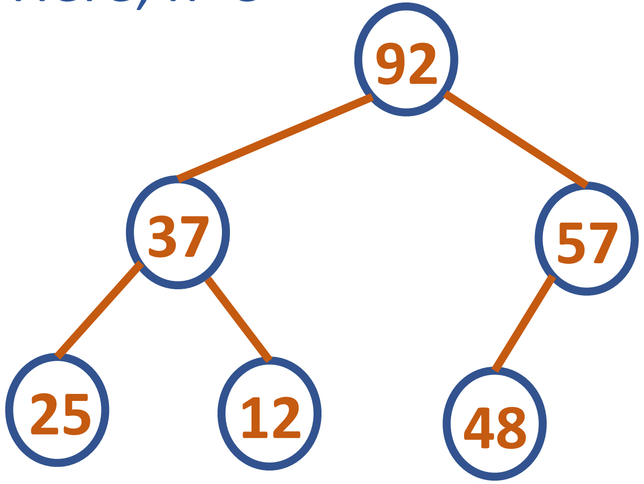


57<92, Heapify

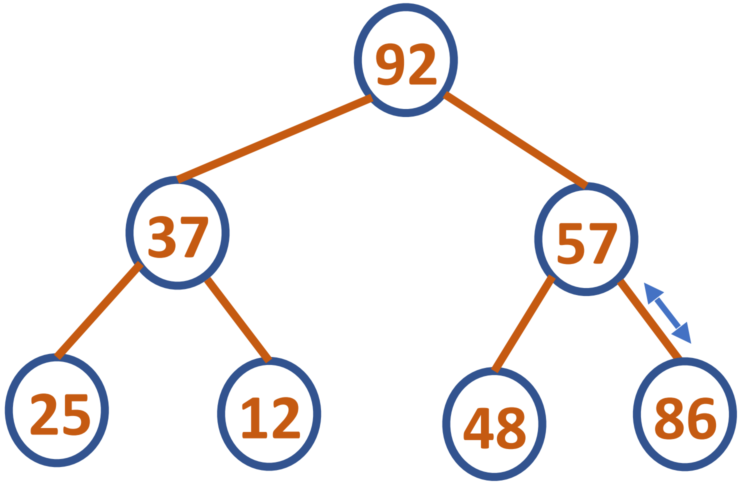


Heap

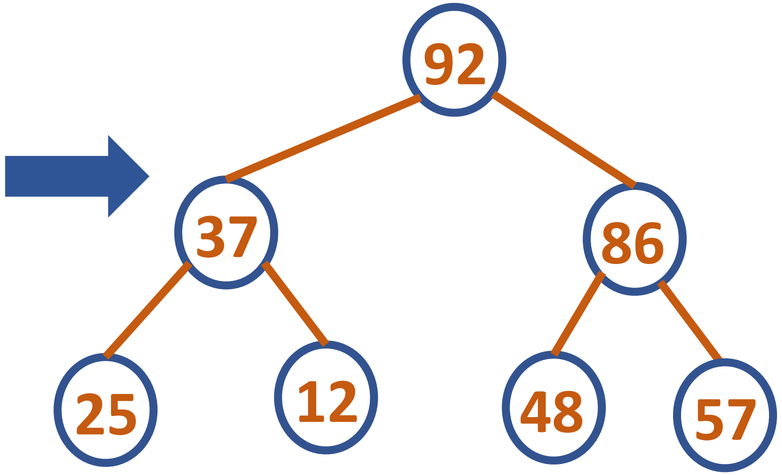
Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



Insert 86



57<86, Heapify

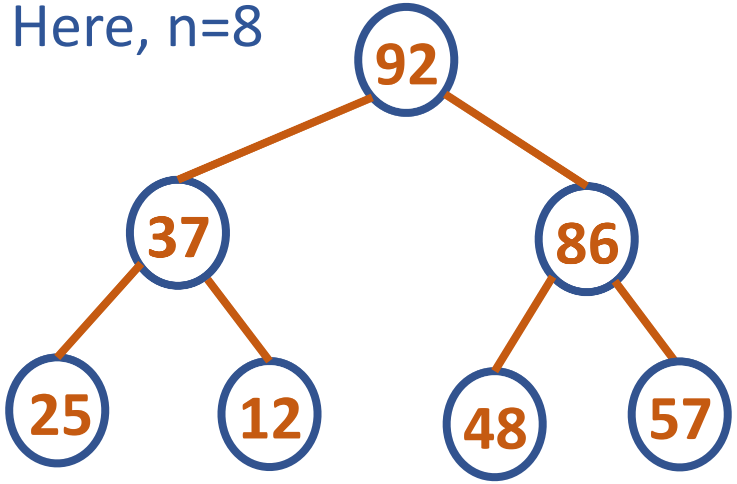


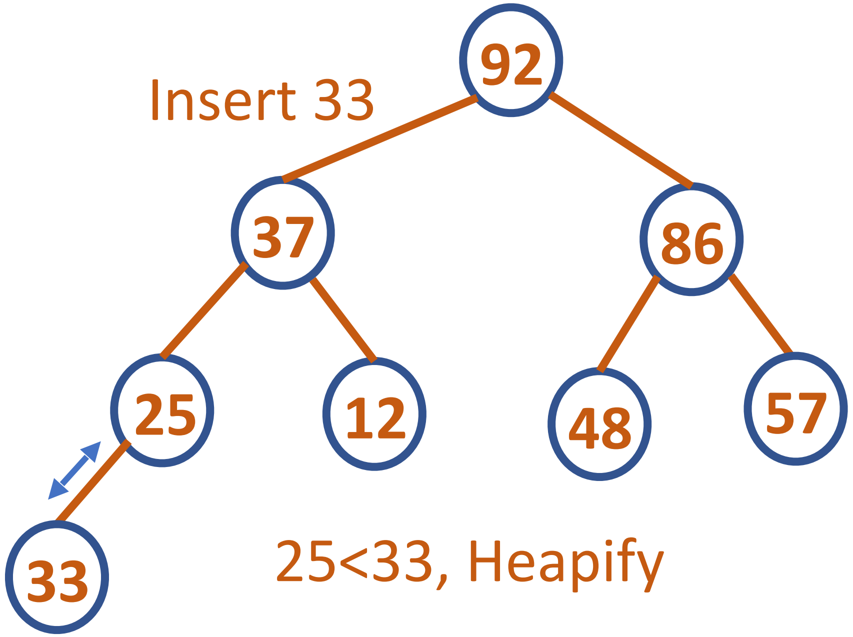
Heap

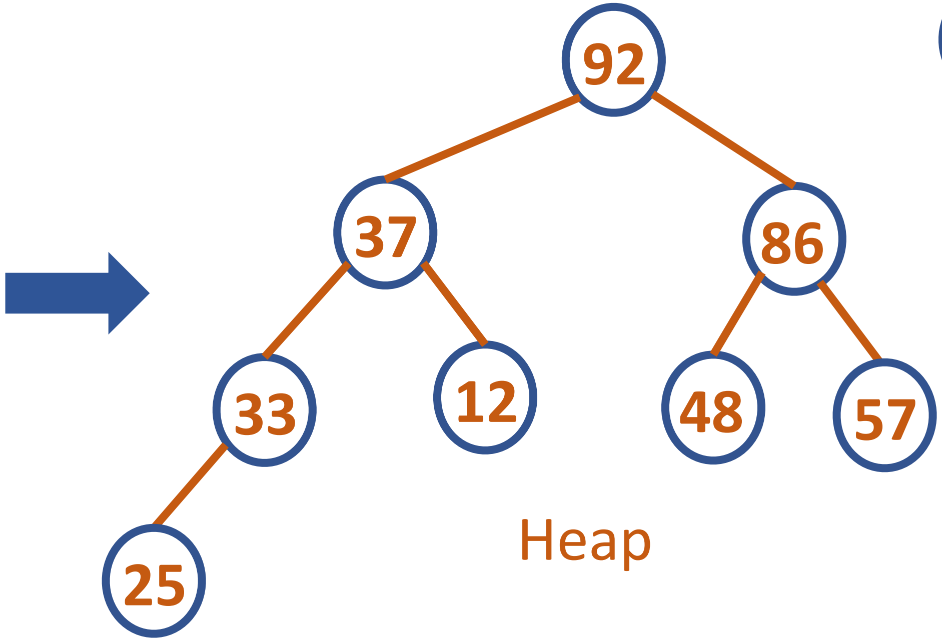
Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

Here, n=8







Heap Construction – Top Down

1. First, attach a new node with key *K* in it after the last leaf of the existing heap
2. Then sift *K* up to its appropriate place in the new heap as follows
3. Compare *K* with its parent's key: if the latter is greater than or equal to *K,* stop (the structure is a heap);
4. otherwise, swap these two keys and compare *K* with its new parent
5. This swapping continues until *K* is not greater than its last parent or it reaches the root
6. In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get *K* 's value

Heap Construction – Top Down

- Efficiency of insertion is O(log n)

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

+91 9449867804