AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

23CSE203

DATA STRUCTURES & ALGORITHMS

Unit 2

Trees: Tree Definition and Properties – Tree ADT - Basic tree traversals - Binary tree - Data structure for representing trees – Linked Structure for Binary Tree – Array based implementation. Priority queues: ADT – Implementing Priority Queue using List – **Heaps**. Maps and Dictionaries: Map ADT – List based Implementation – Hash Tables - Dictionary ADT. Skip Lists - Implementation - Complexity.

Course Outcome:

Course Outcome’s	BTL
CO1, CO2, CO3, CO4 and CO5	1,2,3,4

J.UMA, AP-CSE
Amrita School of Computing

HEAP

Dr.J.UMA

AP-CSE

Heap

Binary tree with these two properties -

Structure property

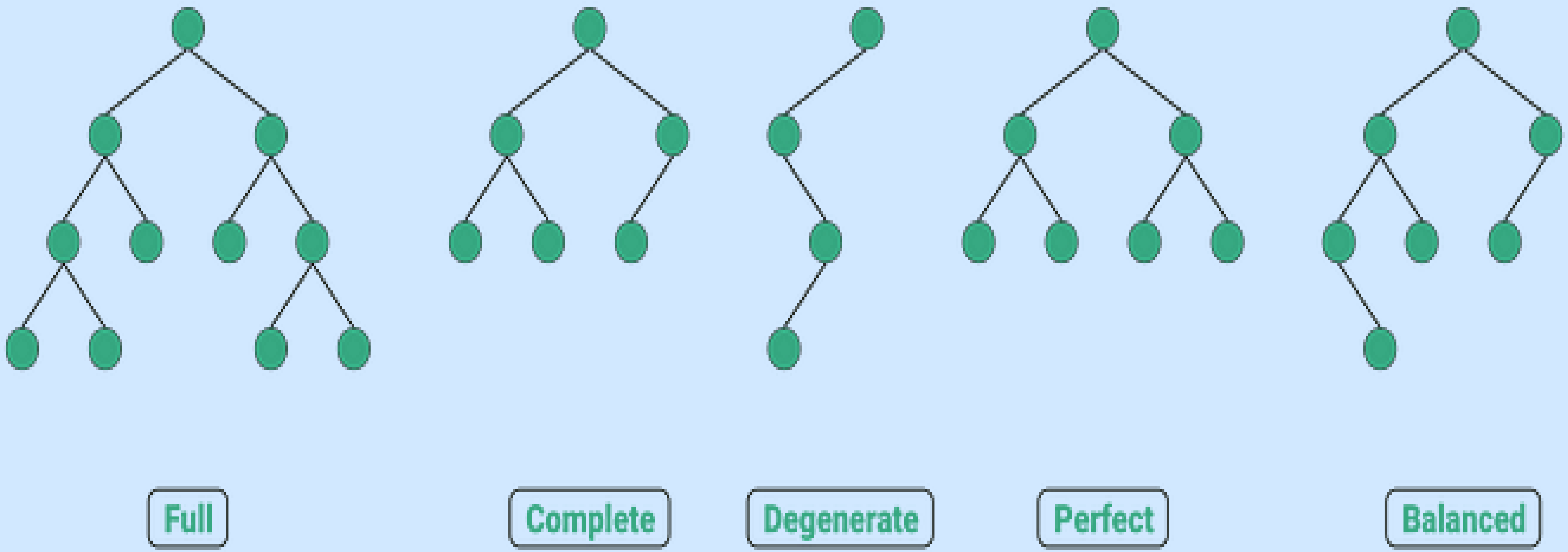
All levels have maximum number of nodes except possibly the last level,
In the last level, all the nodes are to the left

Complete binary tree
Height - $\lceil \log_2(n+1) \rceil$

Heap order property

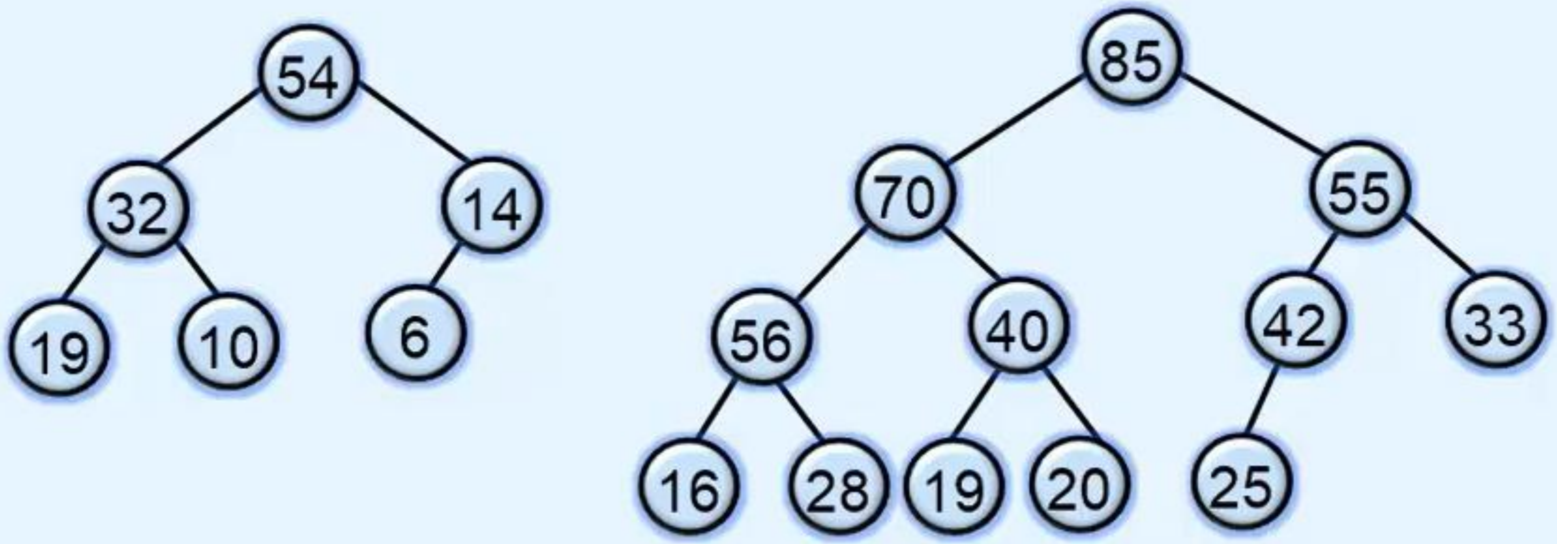
Key in any node N is greater than or equal to the keys in both children of N

Key in node N is greater than or equal to the keys of all its descendants
Root node contains the highest key


Full
Complete
Degenerate
Perfect
Balanced

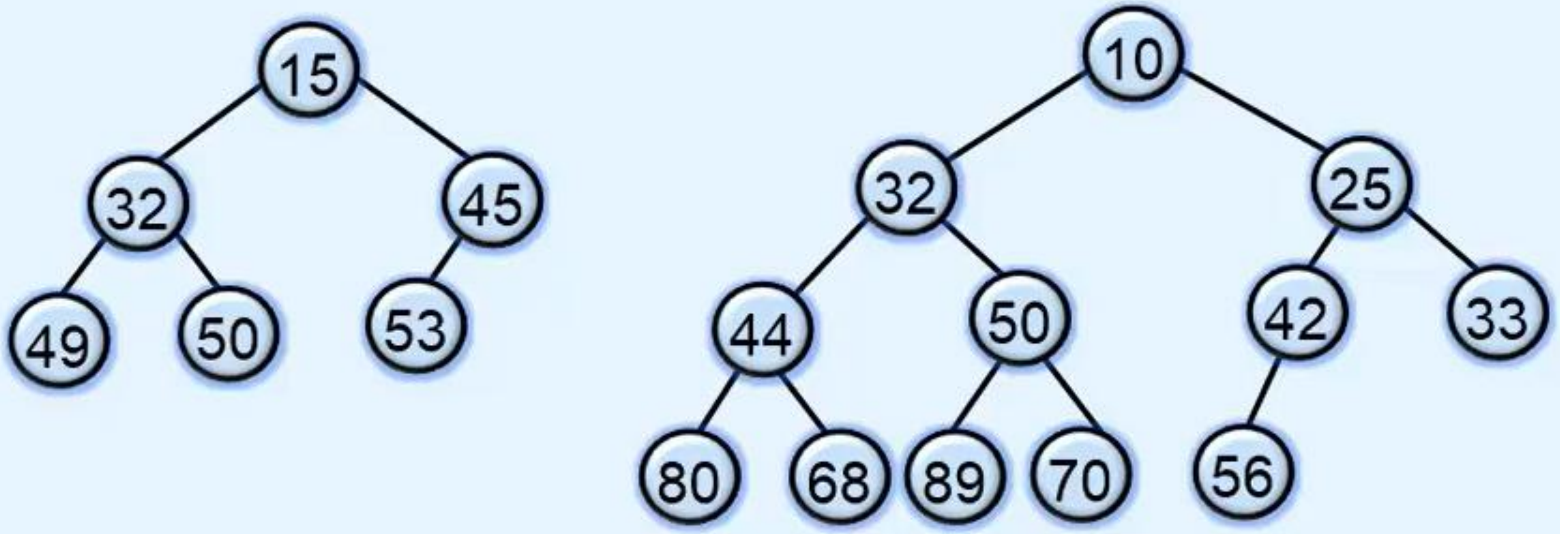
Key in any node N is greater than or equal to the keys in both children of N

Max Heaps



Key in any node N is smaller than or equal to the keys in both its children

Min Heaps



Heap Data Structure Applications

- Heaps have various applications, like:
- Heaps are commonly used to implement priority queues, where elements are retrieved based on their priority (maximum or minimum value).
- Heapsort is a sorting algorithm that uses a heap to sort an array in ascending or descending order.
- Heaps are used in graph algorithms like **Dijkstra's algorithm** and **Prim's algorithm** for finding the shortest paths and minimum spanning trees.

Representation of Heap

root - index 1 of the array

Left child of node N at index i - index 2i $2i > n$ Left child does not exist

Right child of node N at index i - index (2i+1) $2i+1 > n$ Right child does not exist

Parent of node at index i - index floor(i/2)

Heap size : n
Array a is used to implement heap
a[0], a[1], a[2],……………, a[n], a[n+1],a[n+2],…….a[arraySize-1]

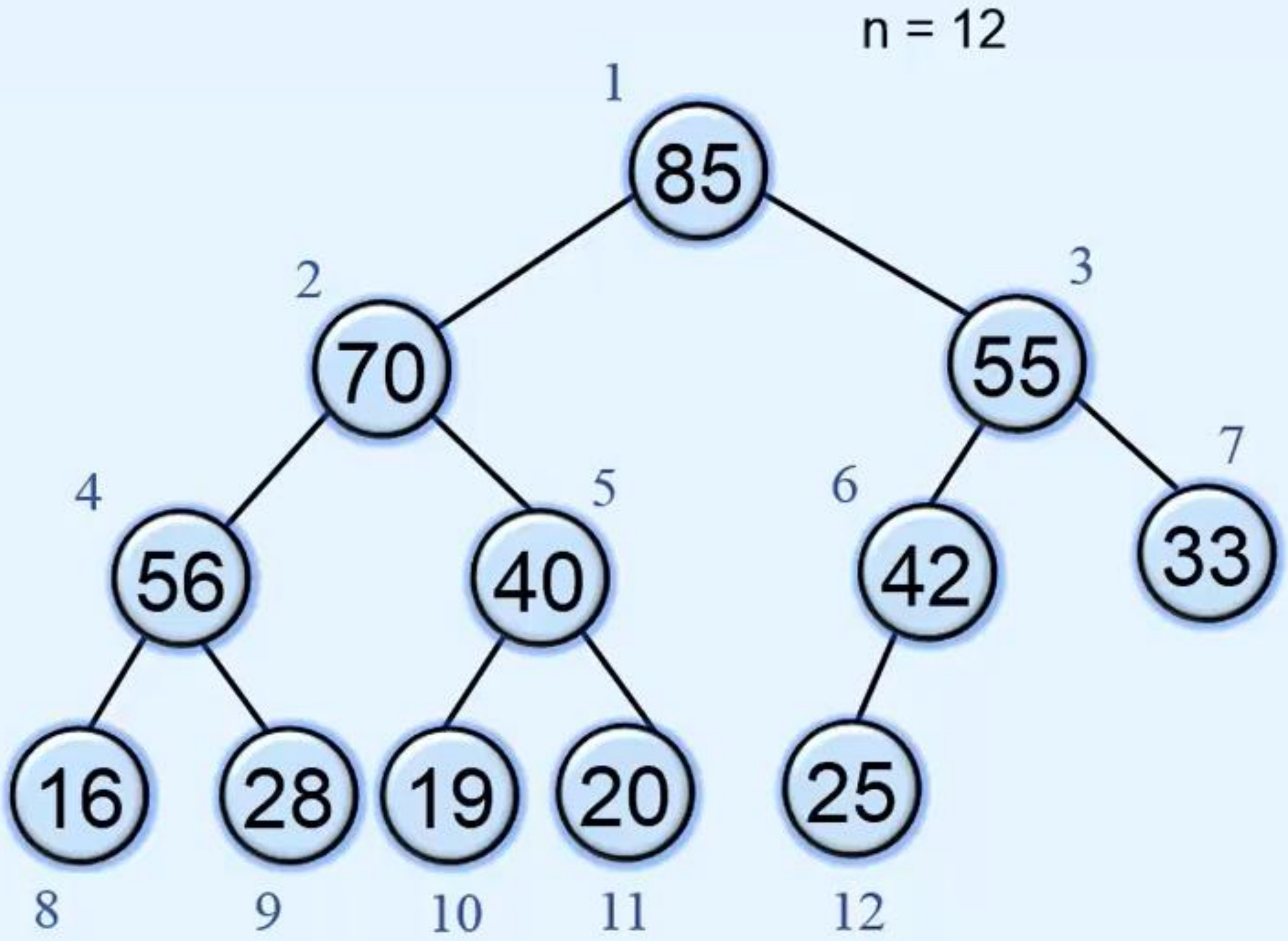
Representation of Heap

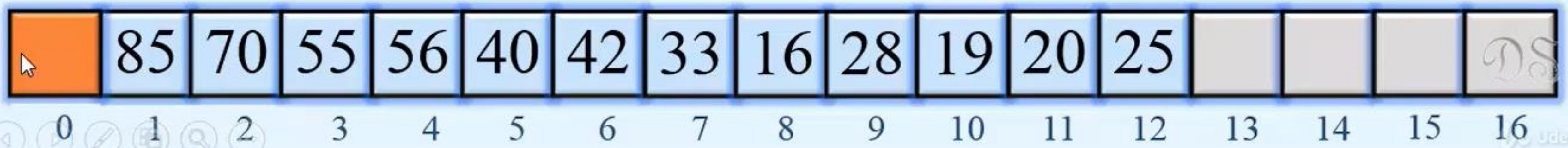
root - index 1 of the array

Left child of node N at index i - index 2i

Right child of node N at index i - index (2i+1)

Parent of node at index i - index floor(i/2)

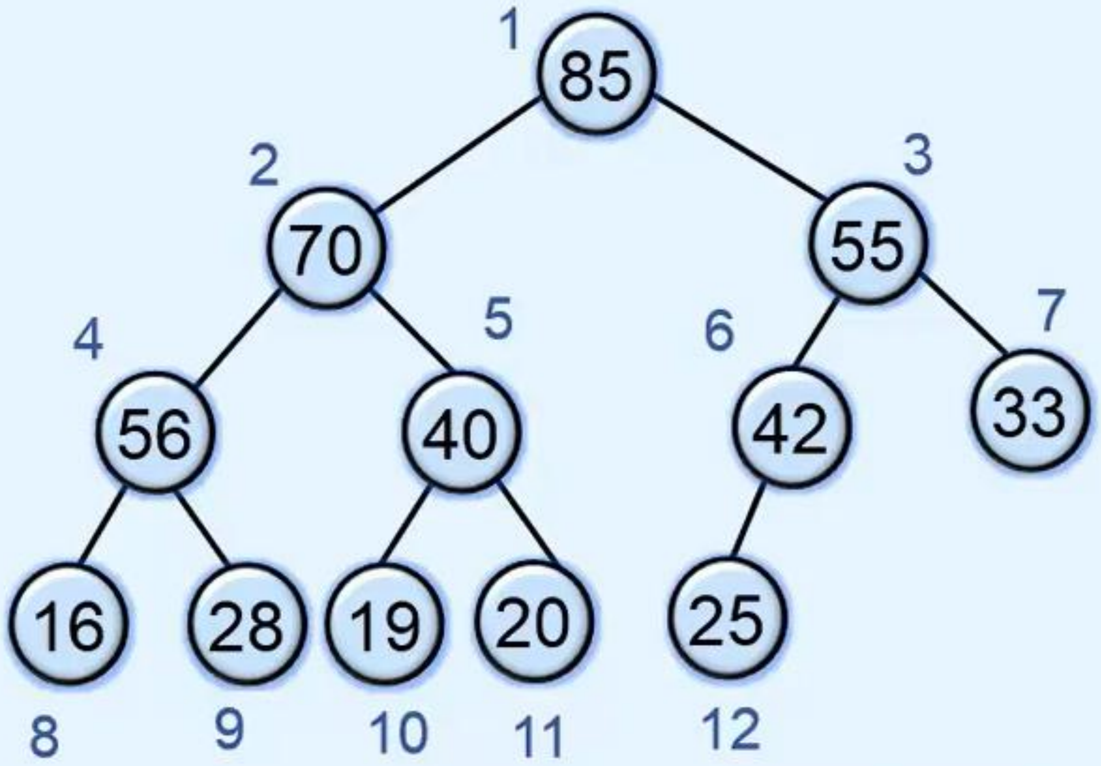




Insertion in heap

Heap size : n $\rightarrow$ n+1

New key is inserted at index (n+1) of the array



n=12

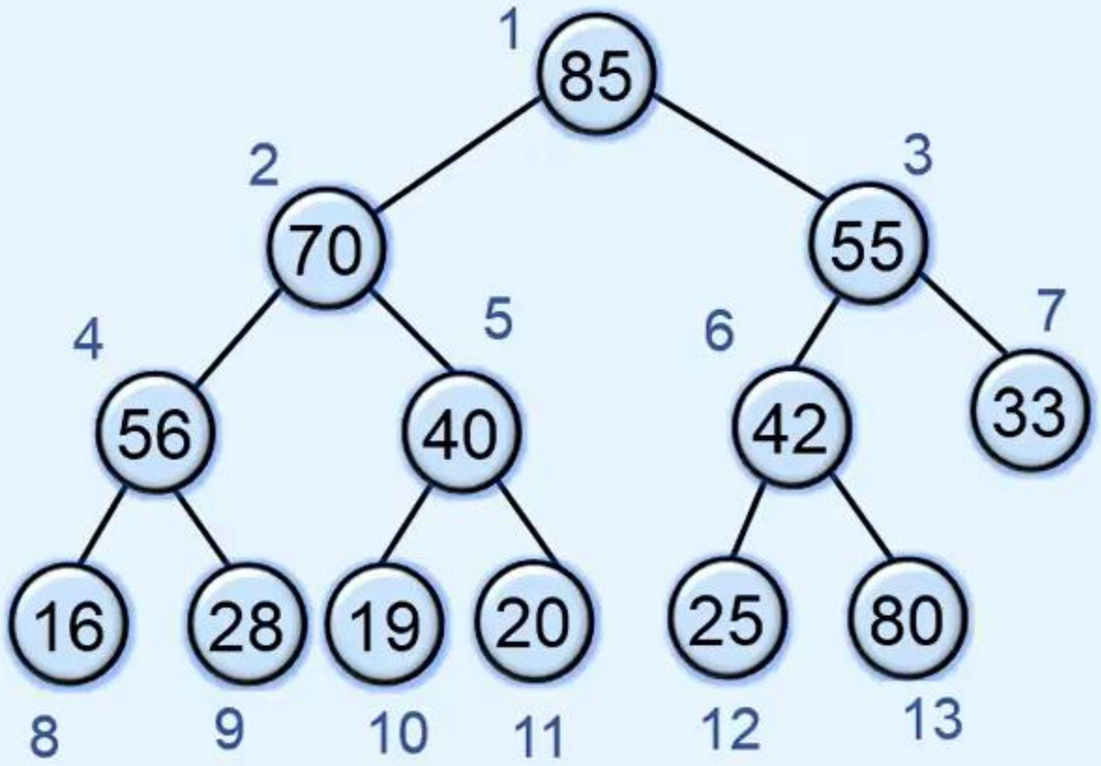
Insert 80

85	70	55	56	40	42	33	16	28	19	20	25				
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Insertion in heap

Heap size : n → n+1

New key is inserted at index (n+1) of the array



n=12

↓

13

Insert 80

85	70	55	56	40	42	33	16	28	19	20	25	80			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Insertion in heap

key k violates heap order property RestoreUp for key k

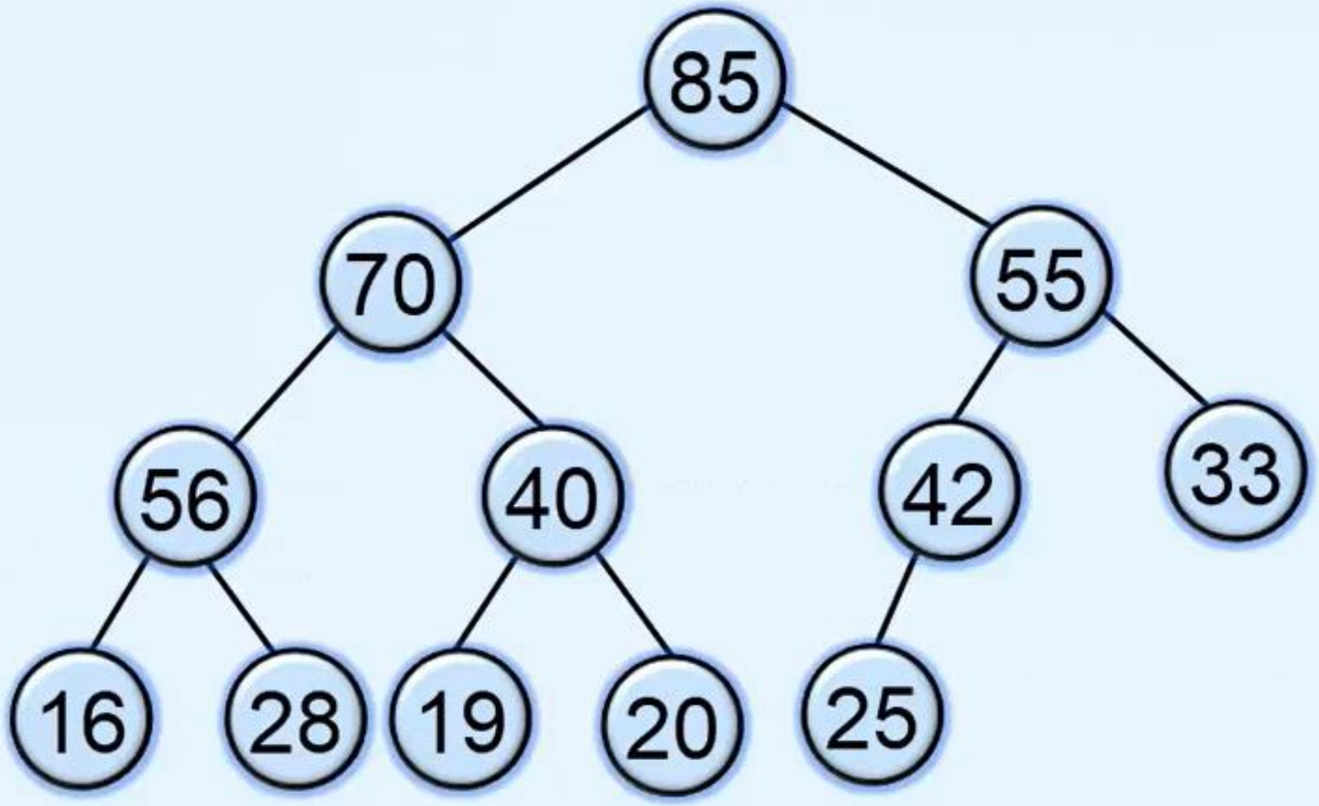
Compare k with the key in parent node

If parent key < k Move the parent key down

Try to insert k in parent's place

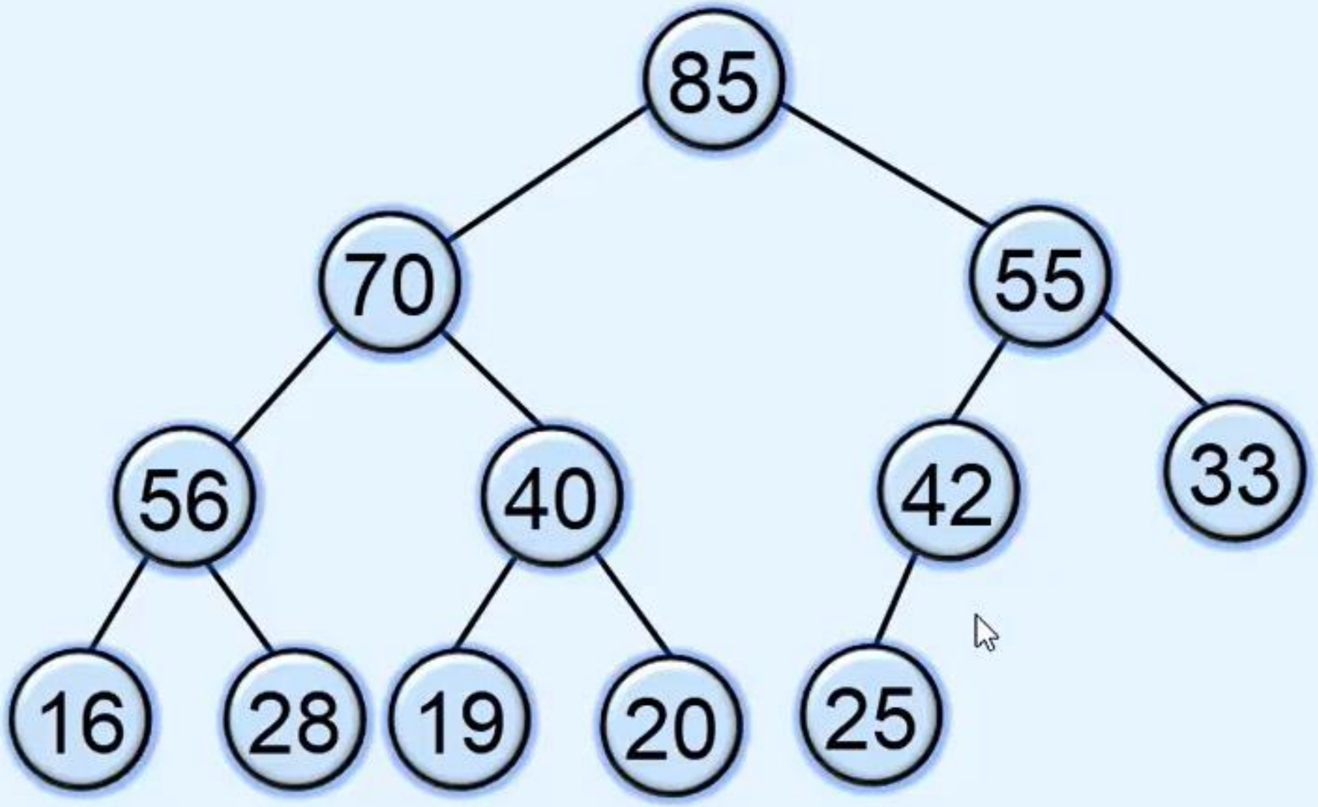
Stop when we get a parent key that is greater than k or we reach the root

Example 1 : Insert 80



n=12

Example 1 : Insert 80

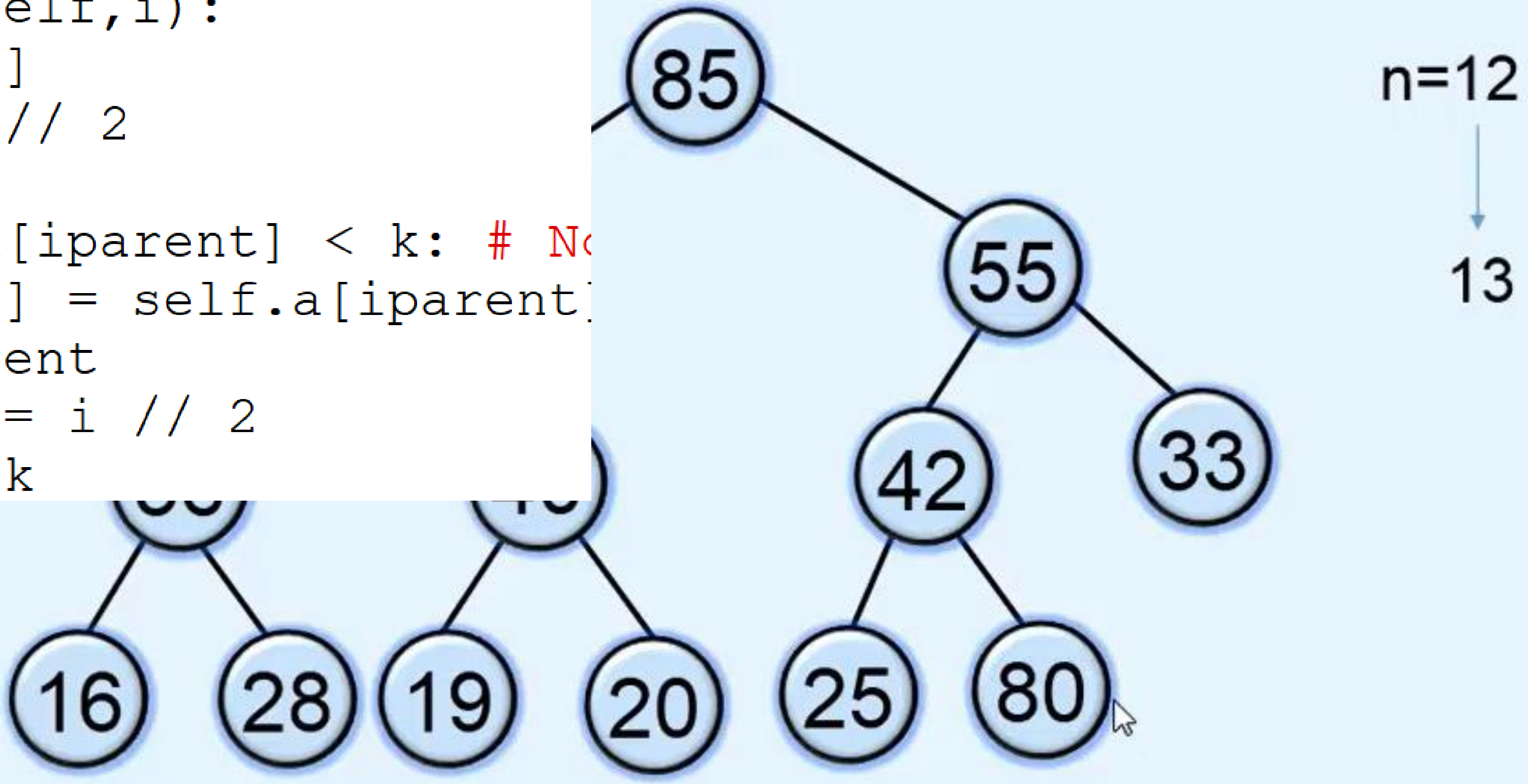


n=12

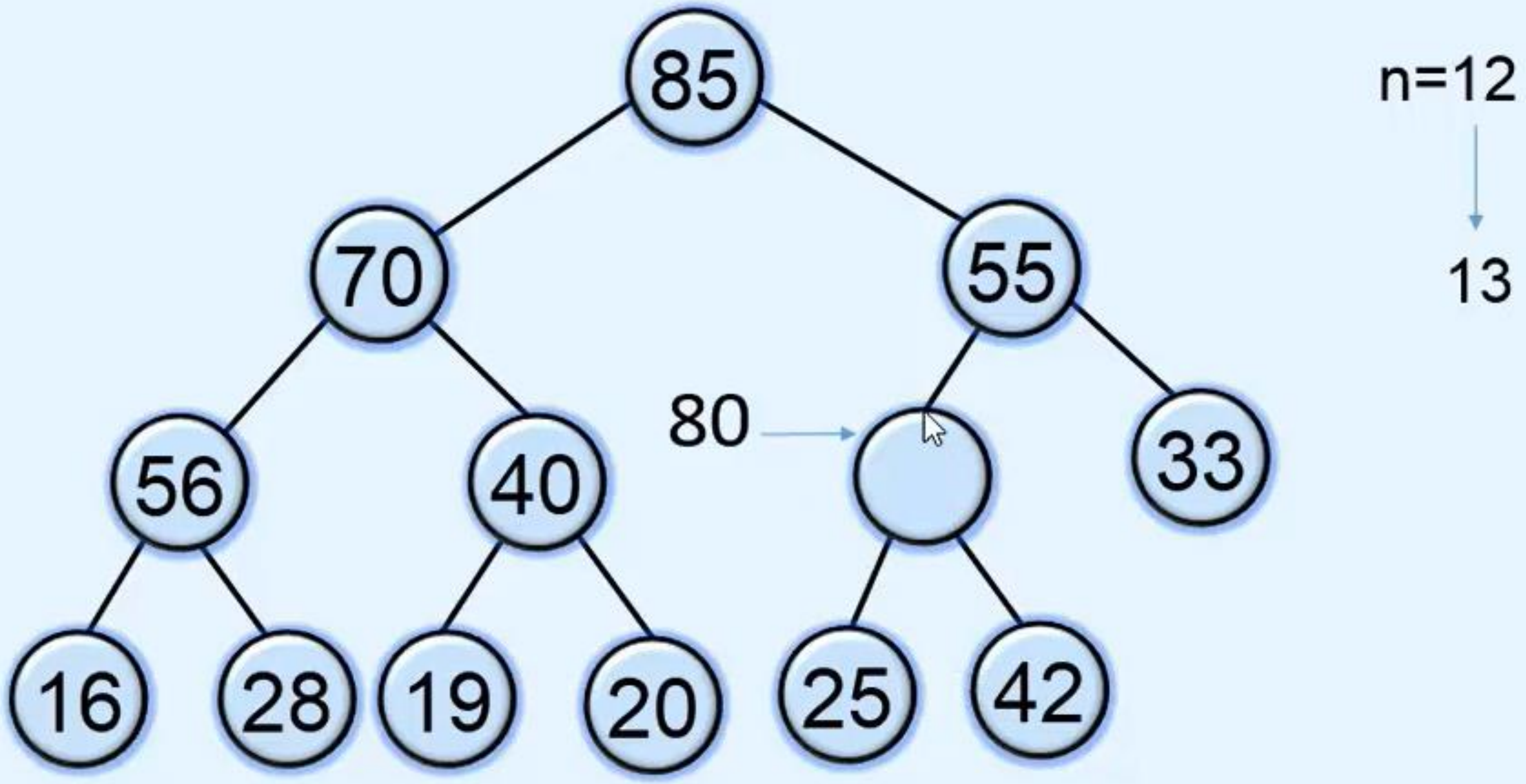
13

```
def insert(self, value):
    self.n+=1
    self.a[self.n] = value
    self.restore_up(self.n)

def restore_up(self,i):
    k = self.a[i]
    iparent = i // 2

    while self.a[iparent] < k: # N
        self.a[i] = self.a[iparent
        i = iparent
        iparent = i // 2
    self.a[i] = k
```

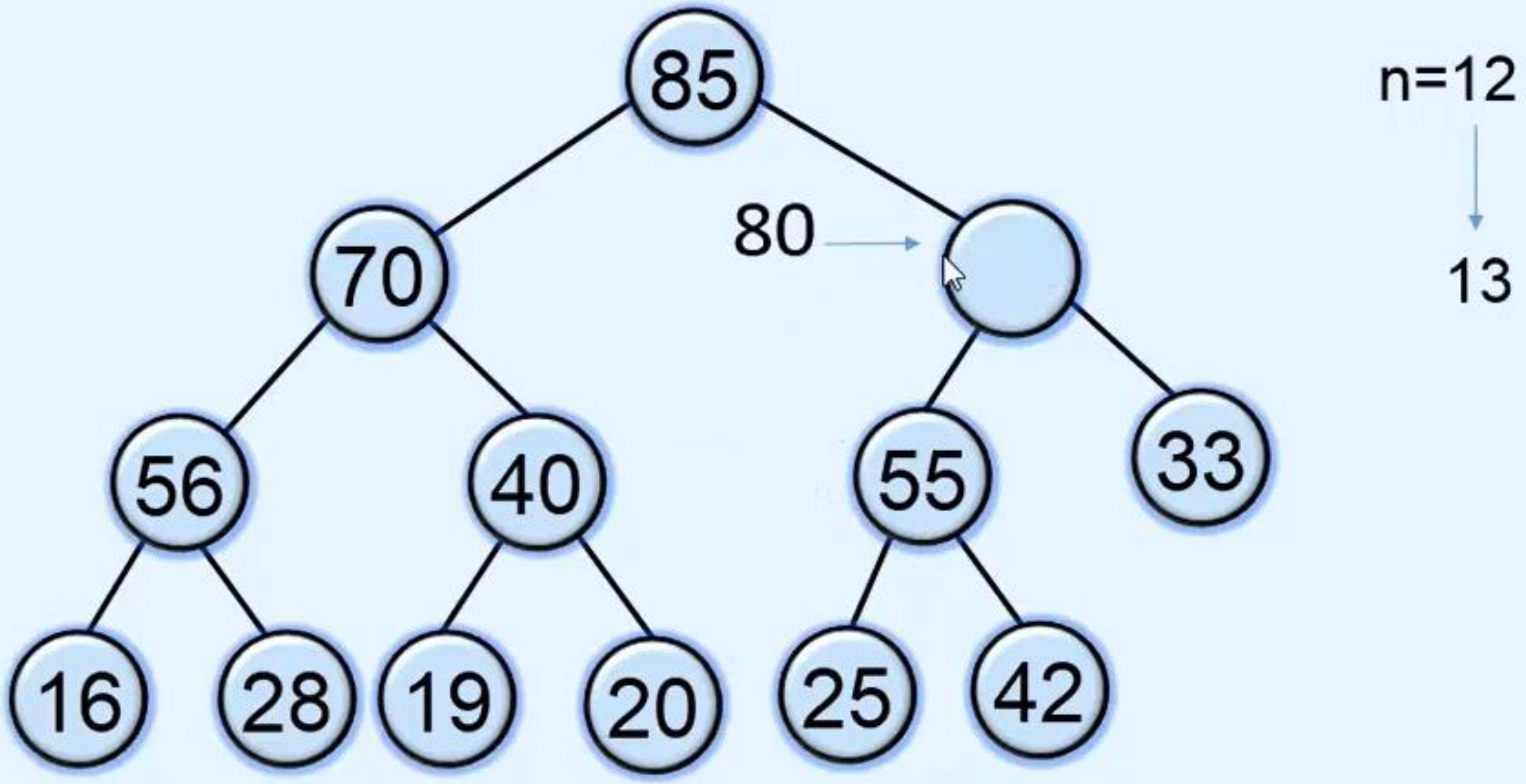
```
def insert(self, value):
    self.n+=1
    self.a[self.n] = value
    self.restore_up(self.n)

def restore_up(self,i):
    k = self.a[i]
    iparent = i // 2

    while self.a[iparent] < k: # No
        self.a[i] = self.a[iparent]
        i = iparent
        iparent = i // 2
    self.a[i] = k
```



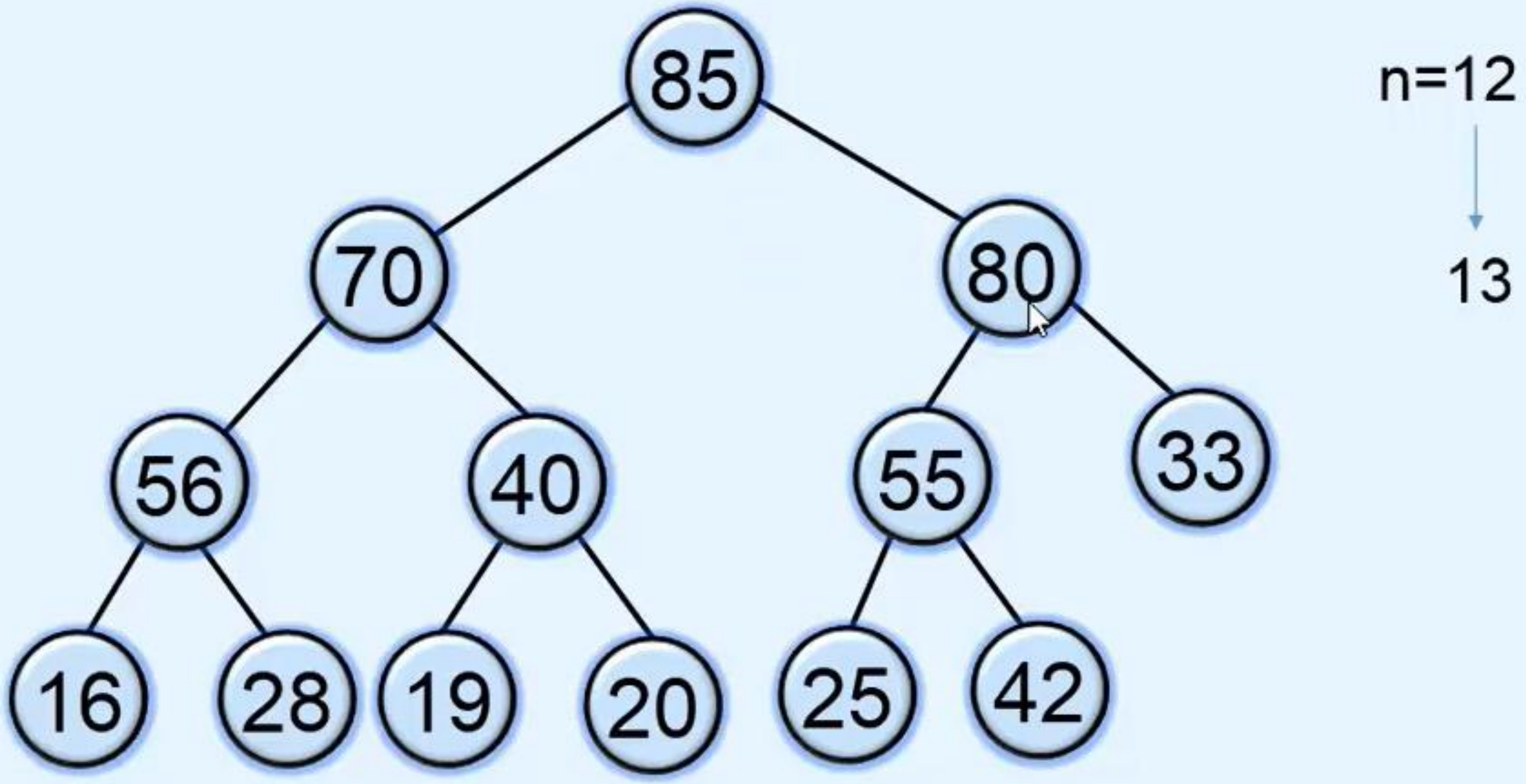
Example 1 : Insert 80



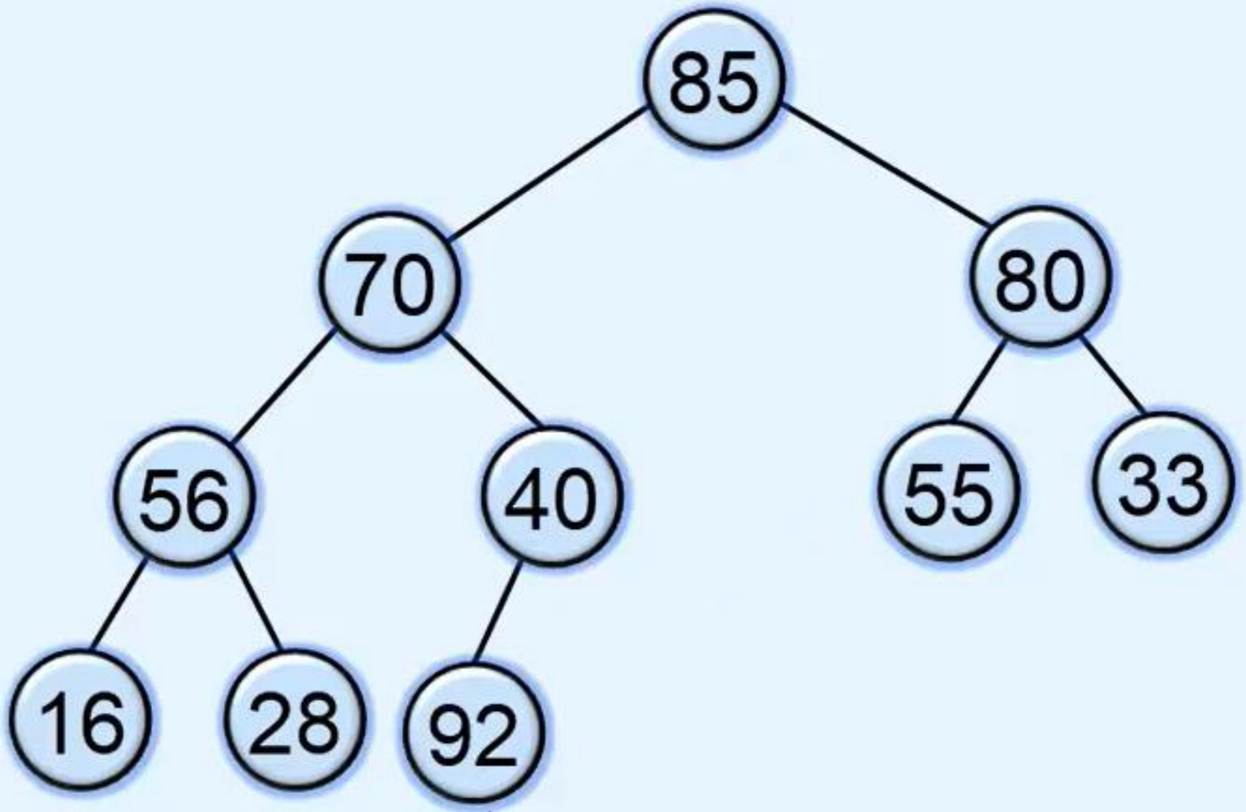
Example 1 : Insert 80



Example 1 : Insert 80



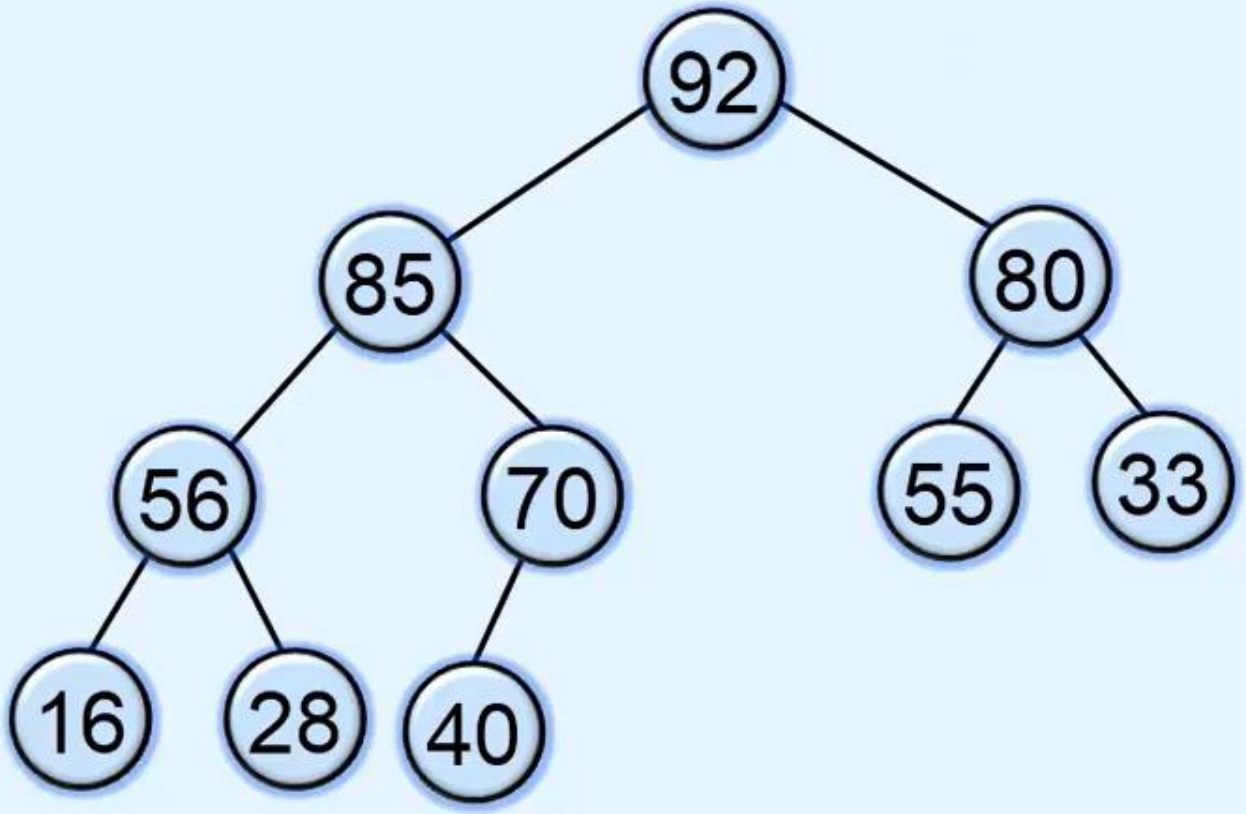
Example 2 : Insert 92



n=9

10

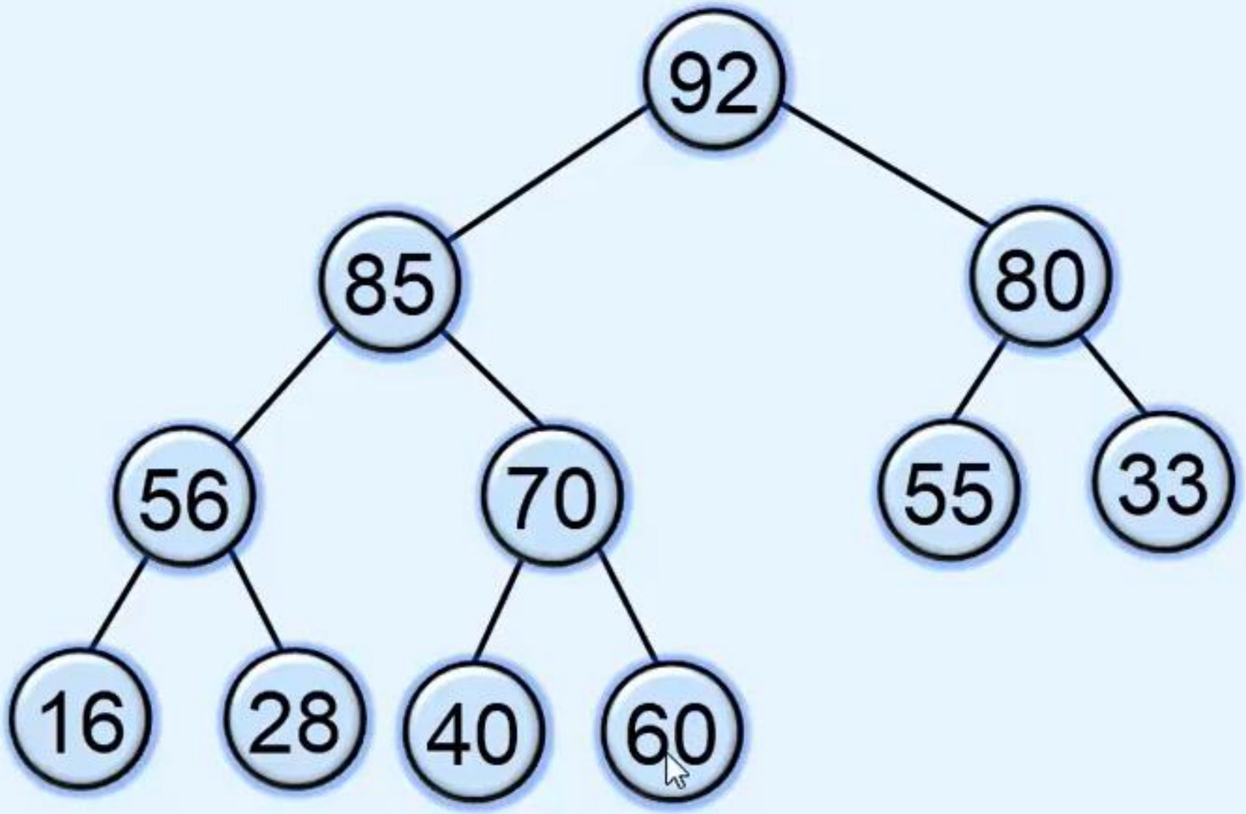
Example 2 : Insert 92



n=9

10

Example 3 : Insert 60



n=10

11

Insertion in Heap

Move from leaf to root node O(h) O(log n)

Worst case Key has to be placed in the root node Insertion of 92

Best case No need to move the key up Insertion of 60

Deletion in heap

Heap of size n

key in the root is stored in some variable

key in last leaf node is copied to the root node → Key at index n is copied to index 1

Size of heap is decreased to n-1

restoreDown for key in root node

Key k violates heap order property

RestoreDown for key k

Compare k with both its left and right child

If both children are smaller than k — Nothing to be done

If one child is greater than k — This greater child moved up

If both children are greater than k — Larger of the two children is moved up

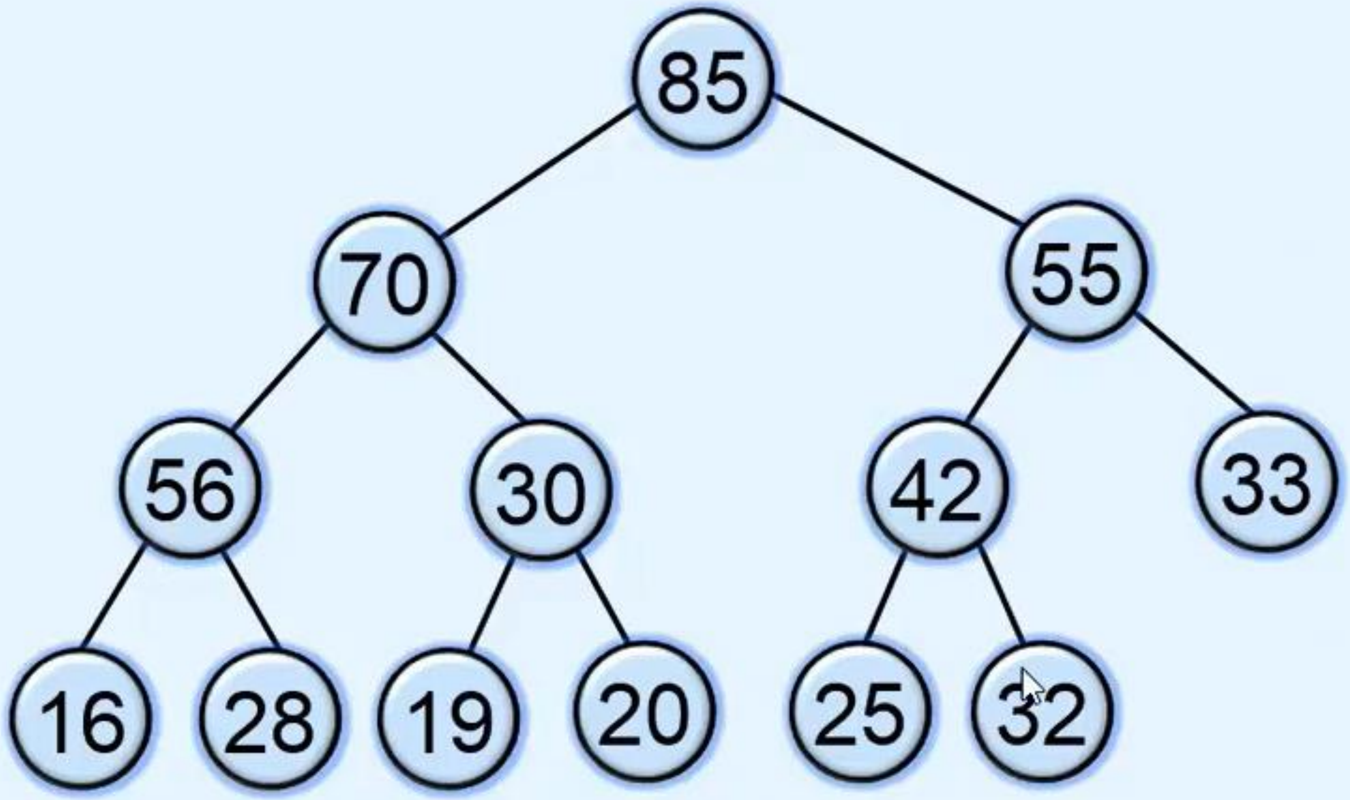
Try to insert key k in place of child that is moved up

Stop when both children are smaller than k or we reach a leaf node

Example 1 : Delete root

maxValue

85

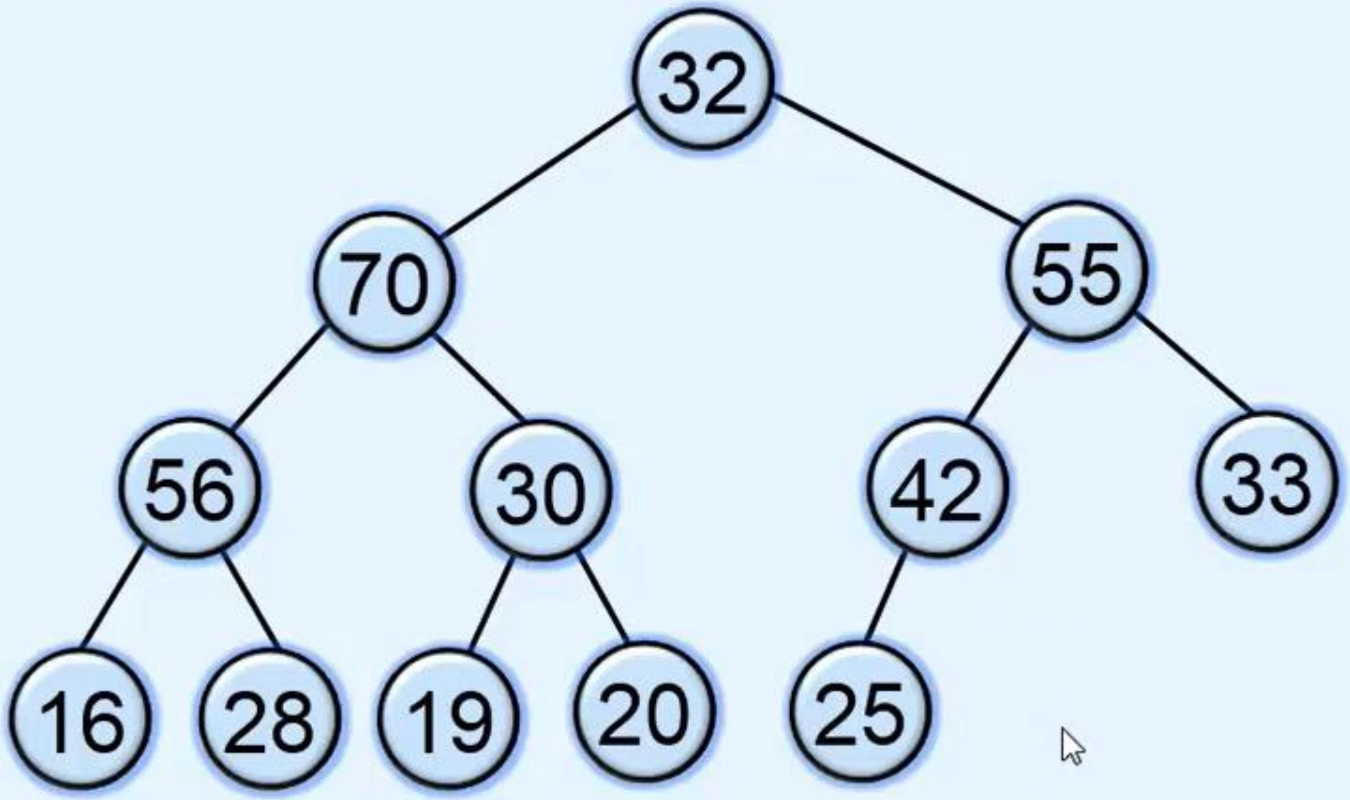


n=13

Example 1 : Delete root

maxValue

85



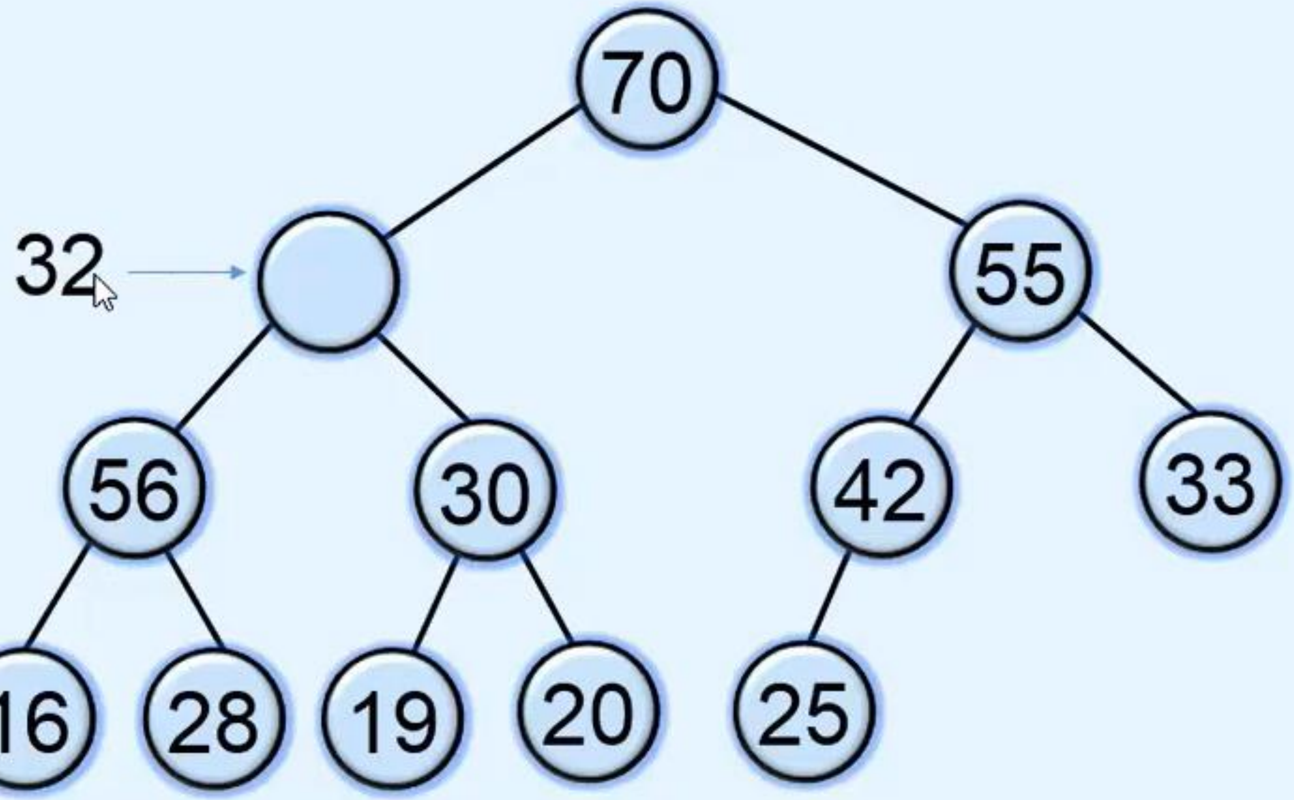
n=13

12

Example 1 : Delete root

maxValue

85



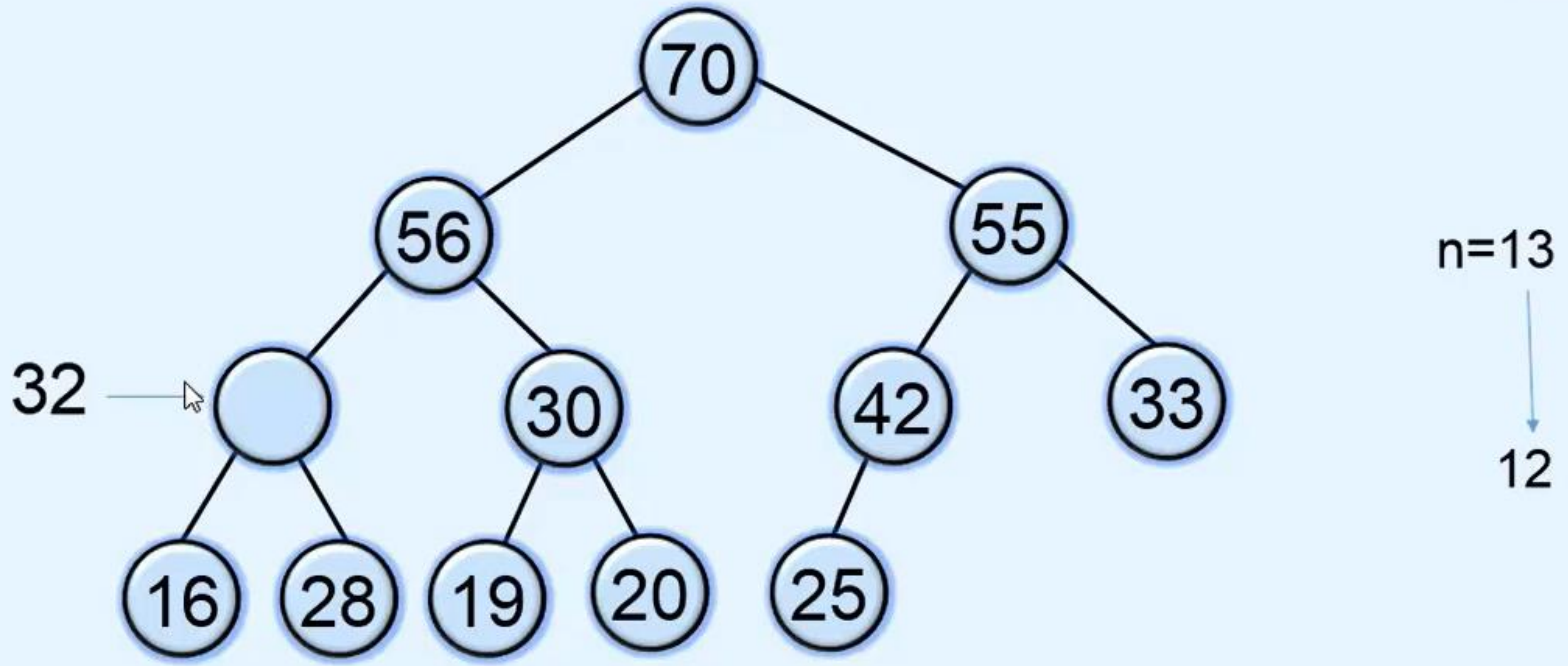
n=13

12

Example 1 : Delete root

maxValue

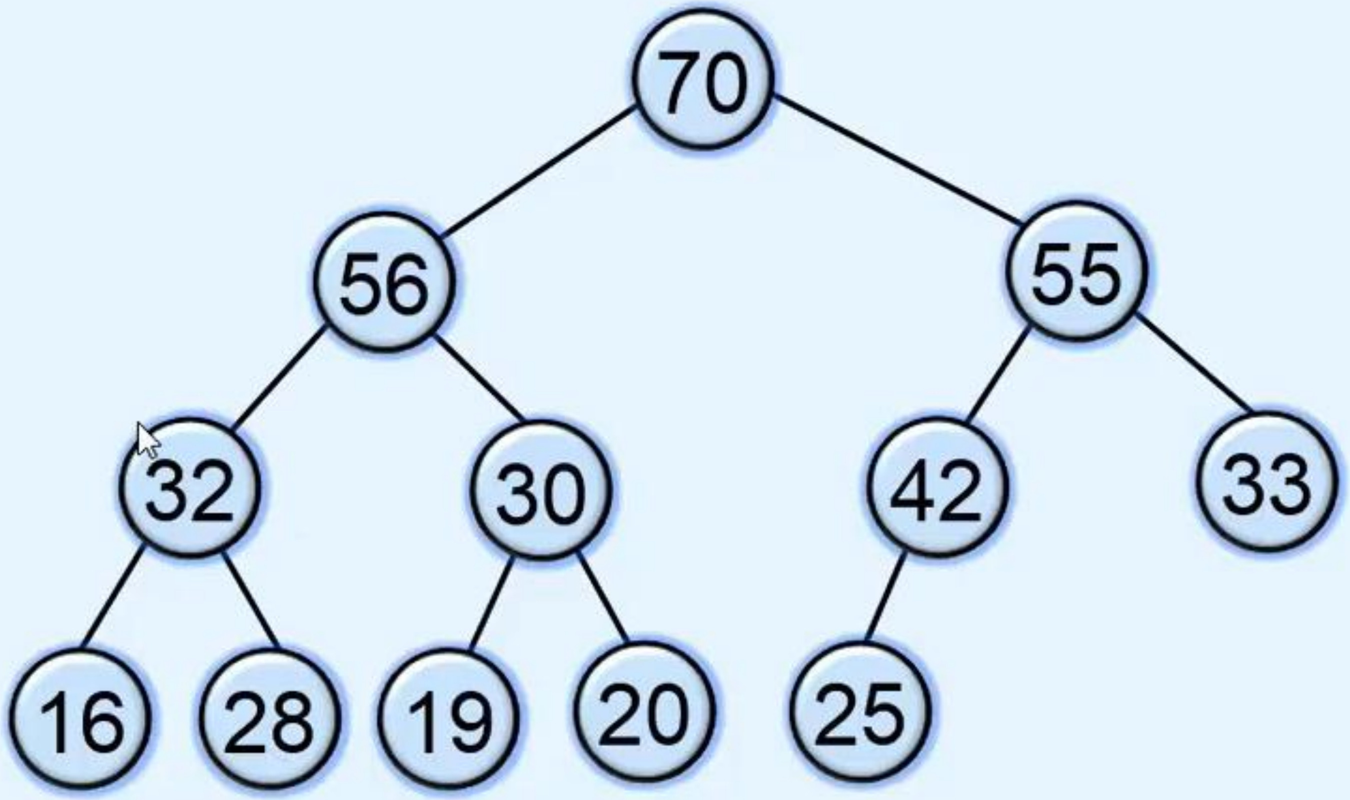
85



Example 1 : Delete root

maxValue

85



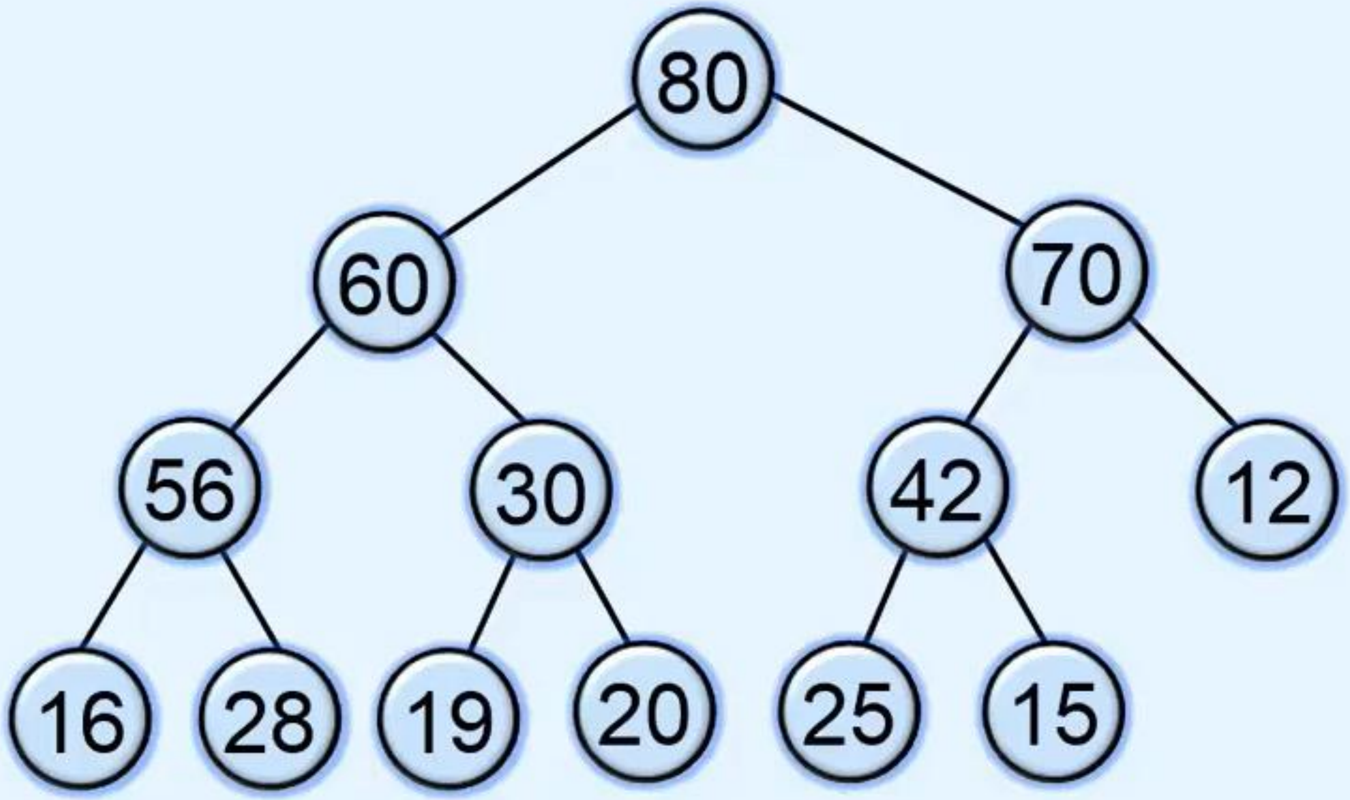
n=13

12

Example 2 : Delete root

maxValue

80

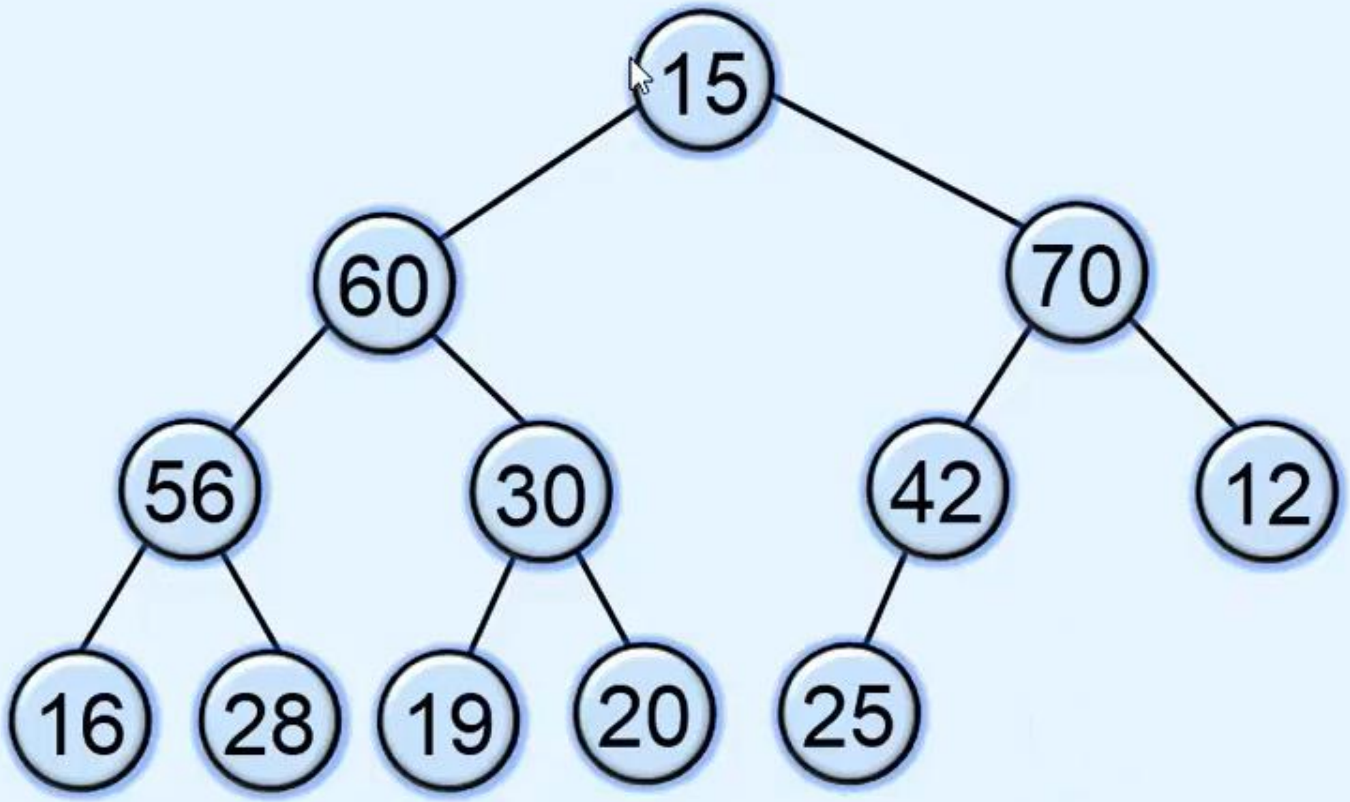


n=13

Example 2 : Delete root

maxValue

80



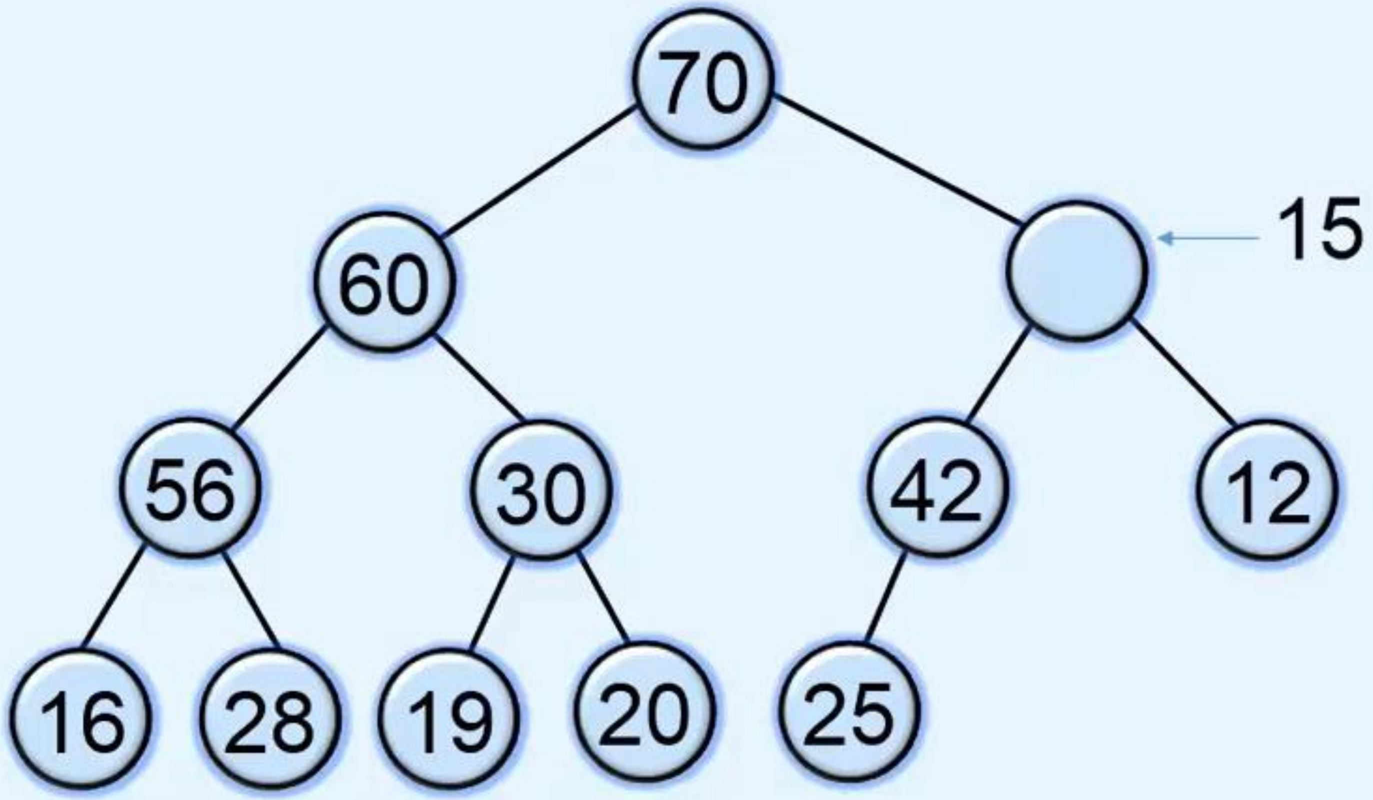
n=13

12

Example 2 : Delete root

maxValue

80



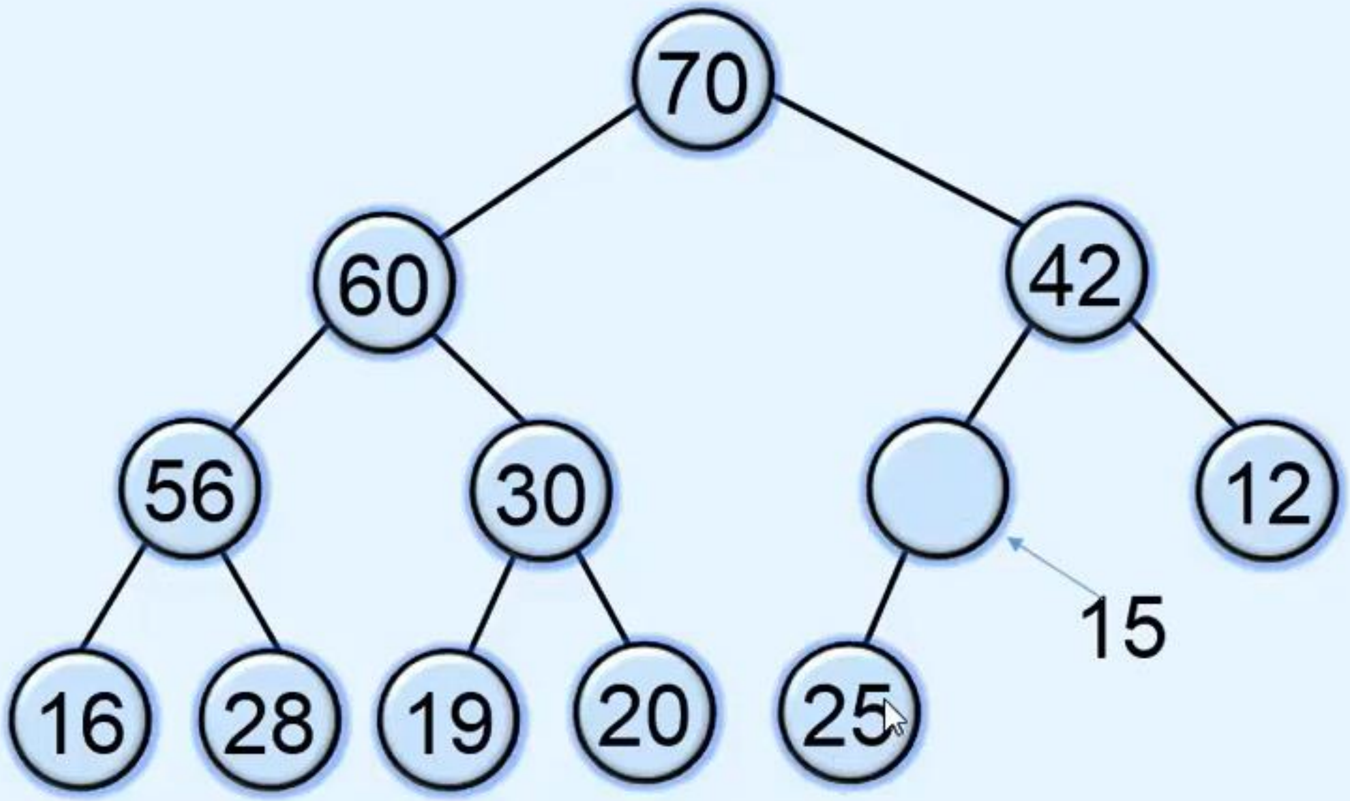
n=13

12

Example 2 : Delete root

maxValue

80



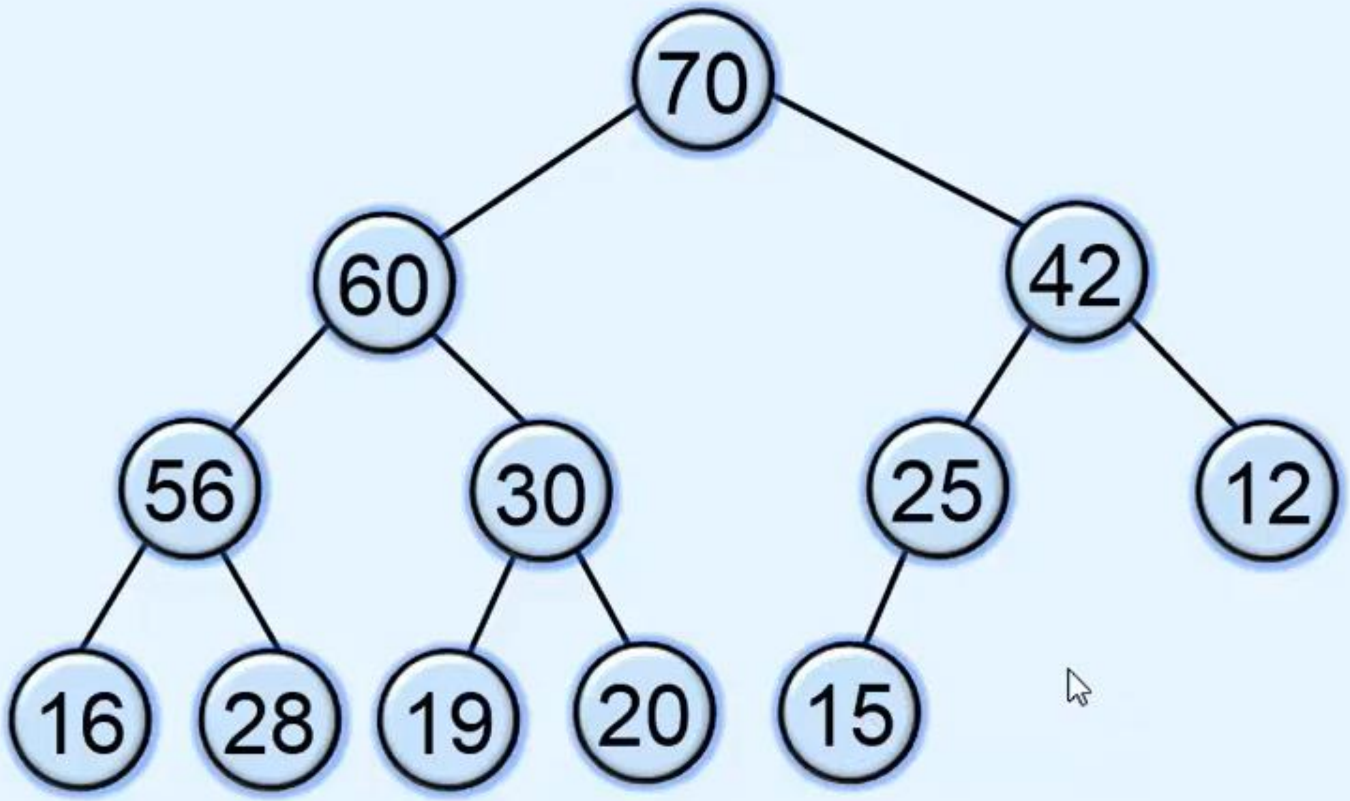
n=13

12

Example 2 : Delete root

maxValue

80



n=13

12

Deletion of root node in Heap

Move from root to a leaf node O(h) O(log n)

Building a heap from an array (Heapify)

- Top Down Approach

  Use restoreUp Procedure

- Bottom Up Approach

  Use restoreDown Procedure

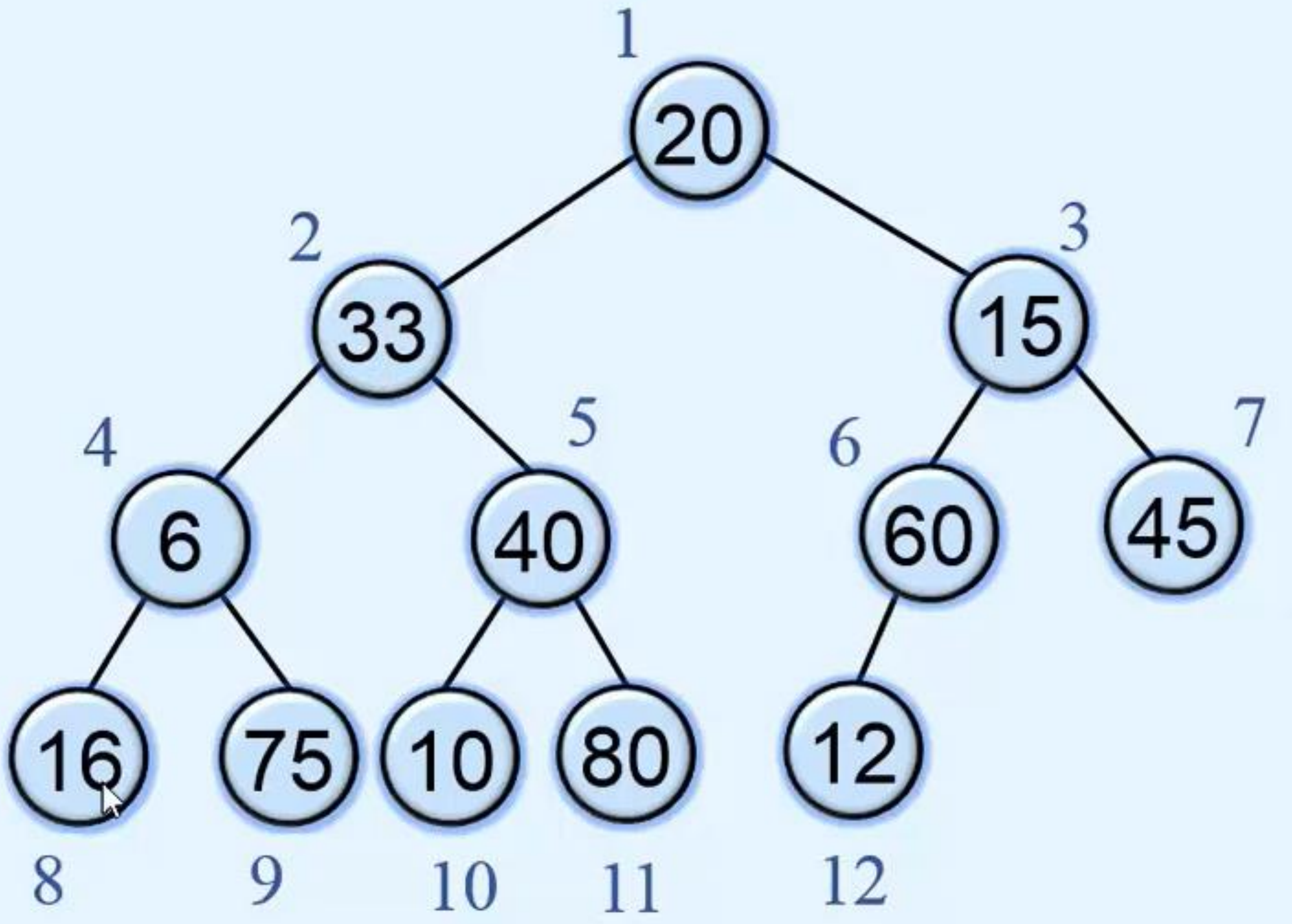
Heapify: Converts an arbitrary binary tree into a heap.

Top Down Approach

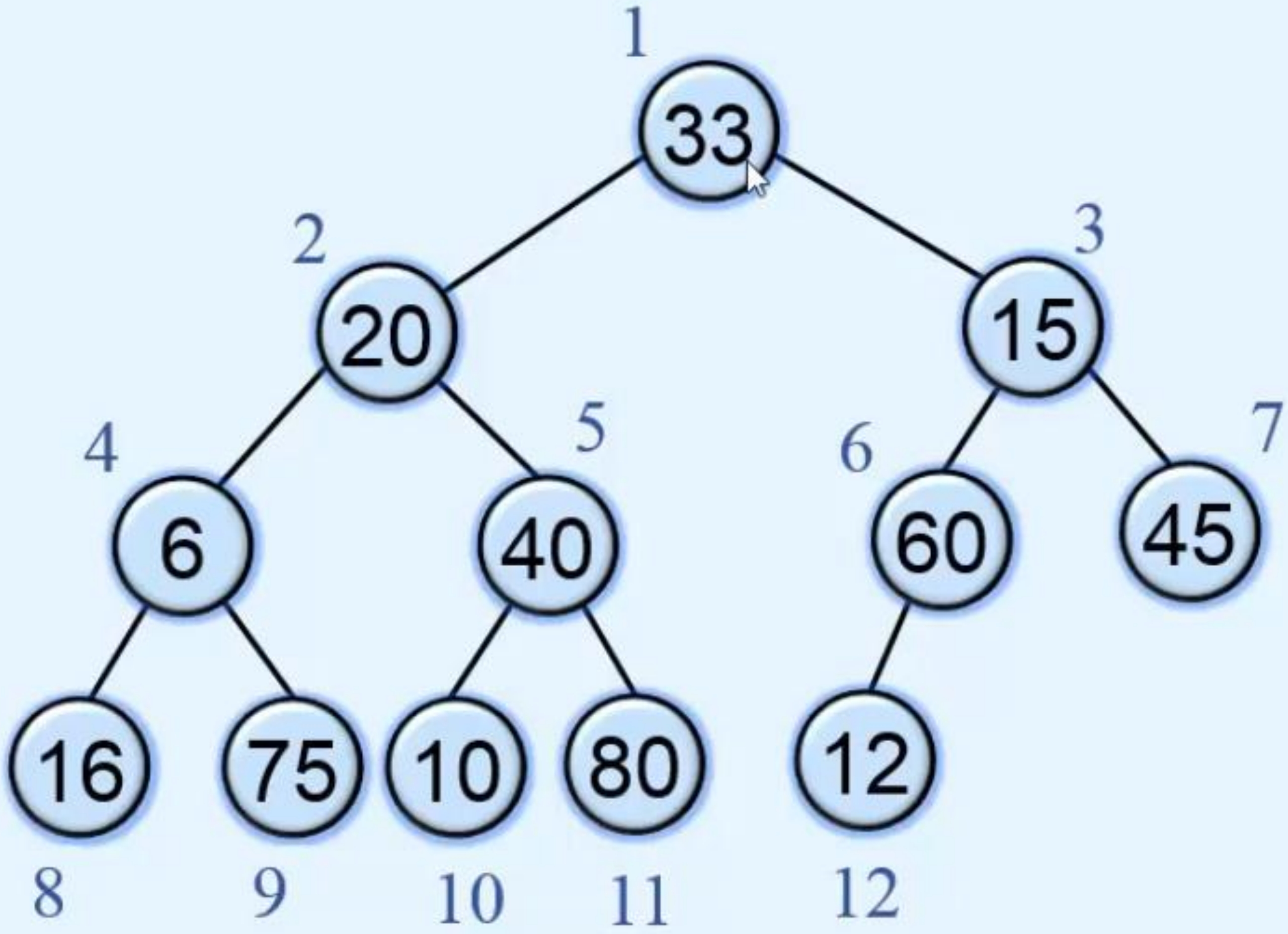
Consider that the array represents a complete binary tree

Call restoreUp for all elements from a[2] to a[n]

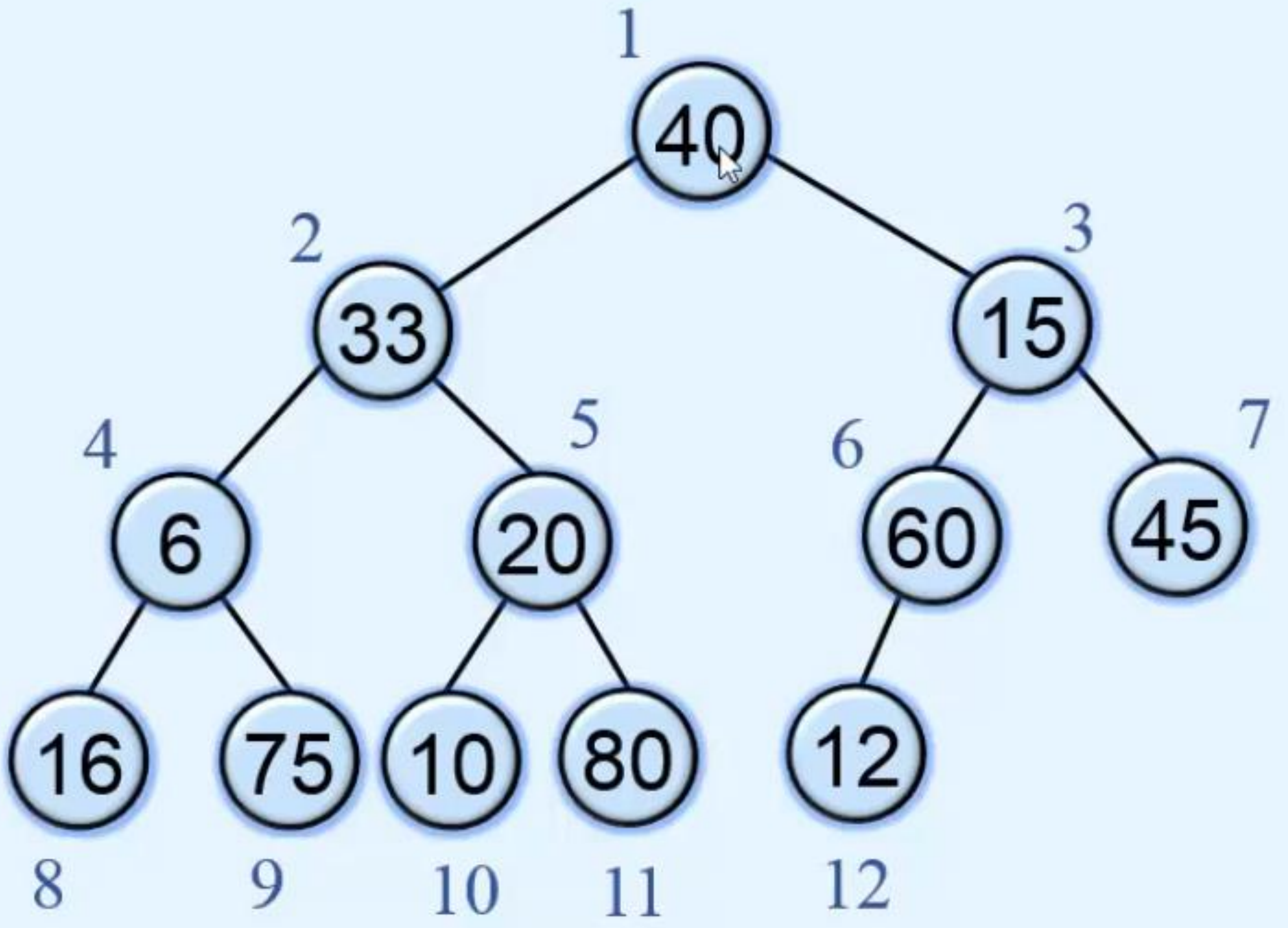
20	33	15	6	40	60	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



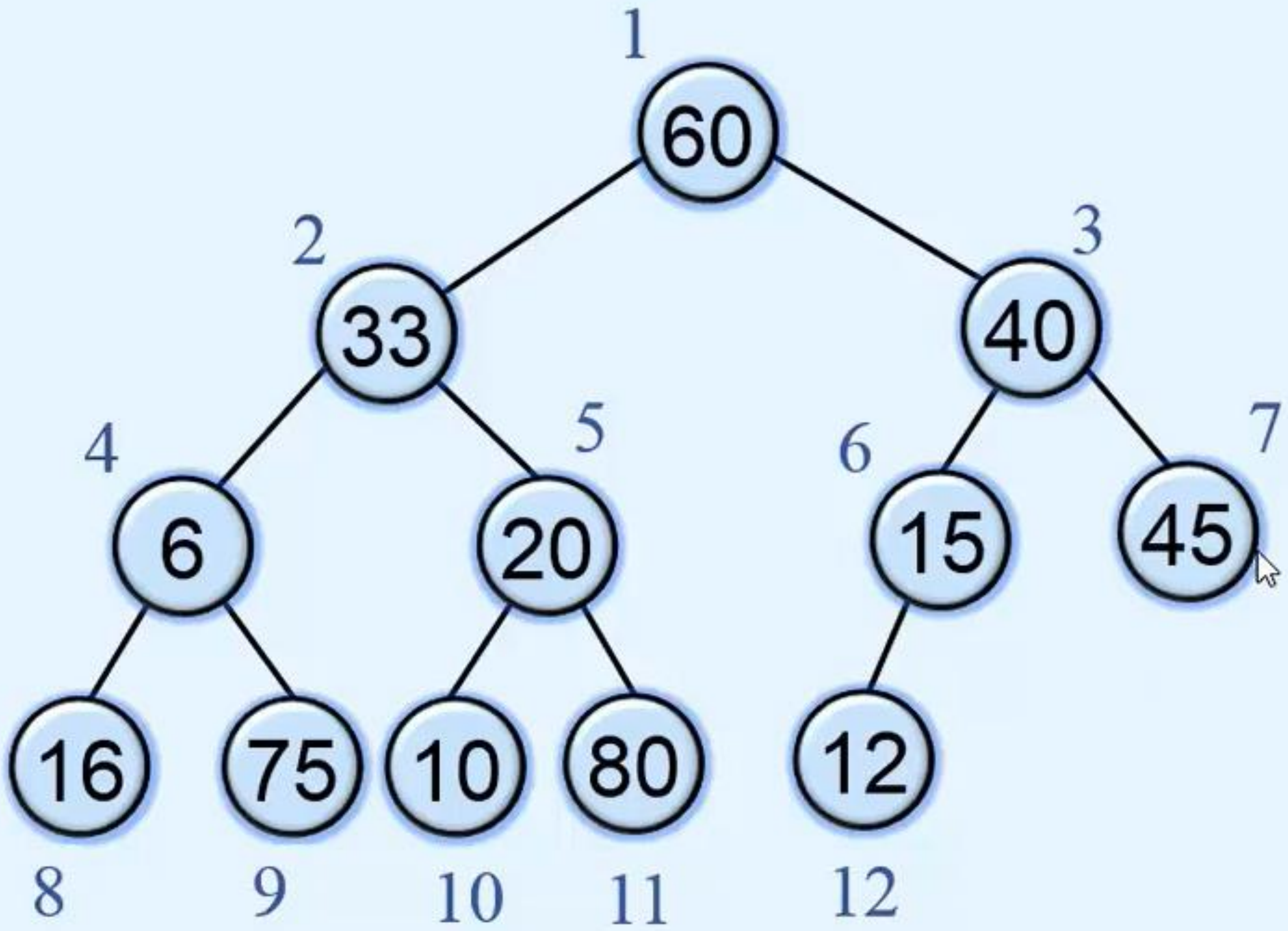
33	20	15	6	40	60	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



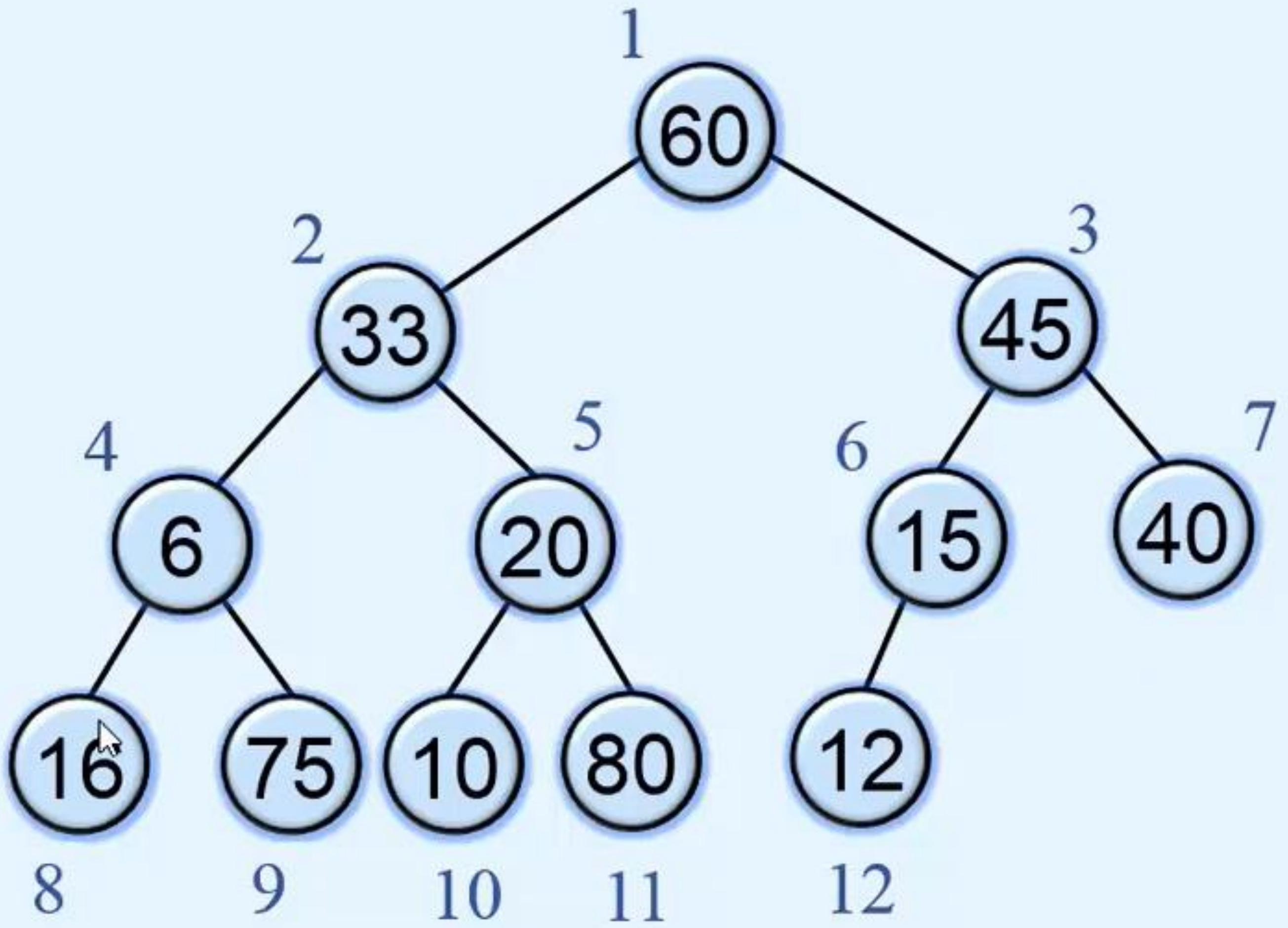
40	33	15	6	20	60	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



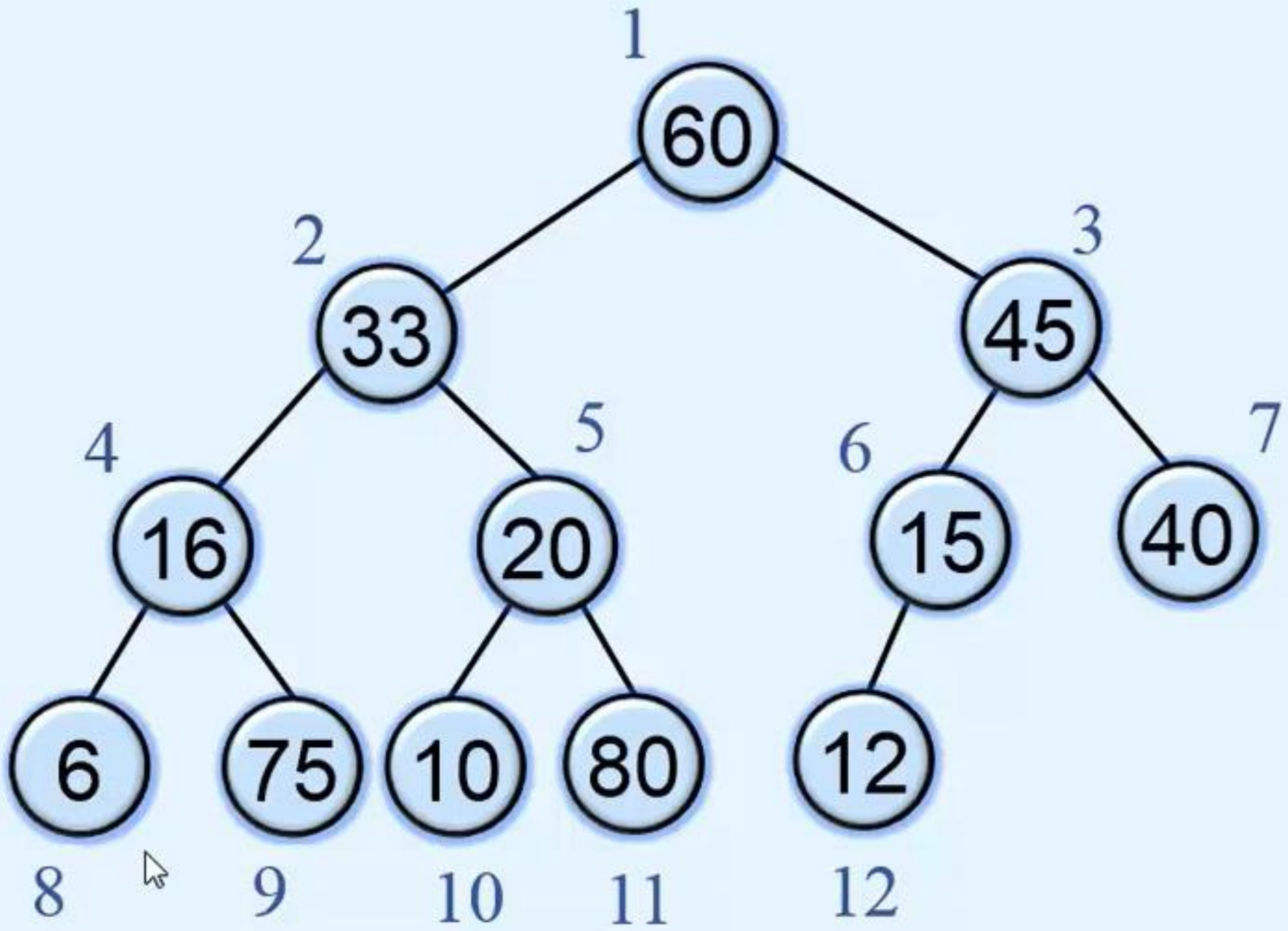
60	33	40	6	20	15	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



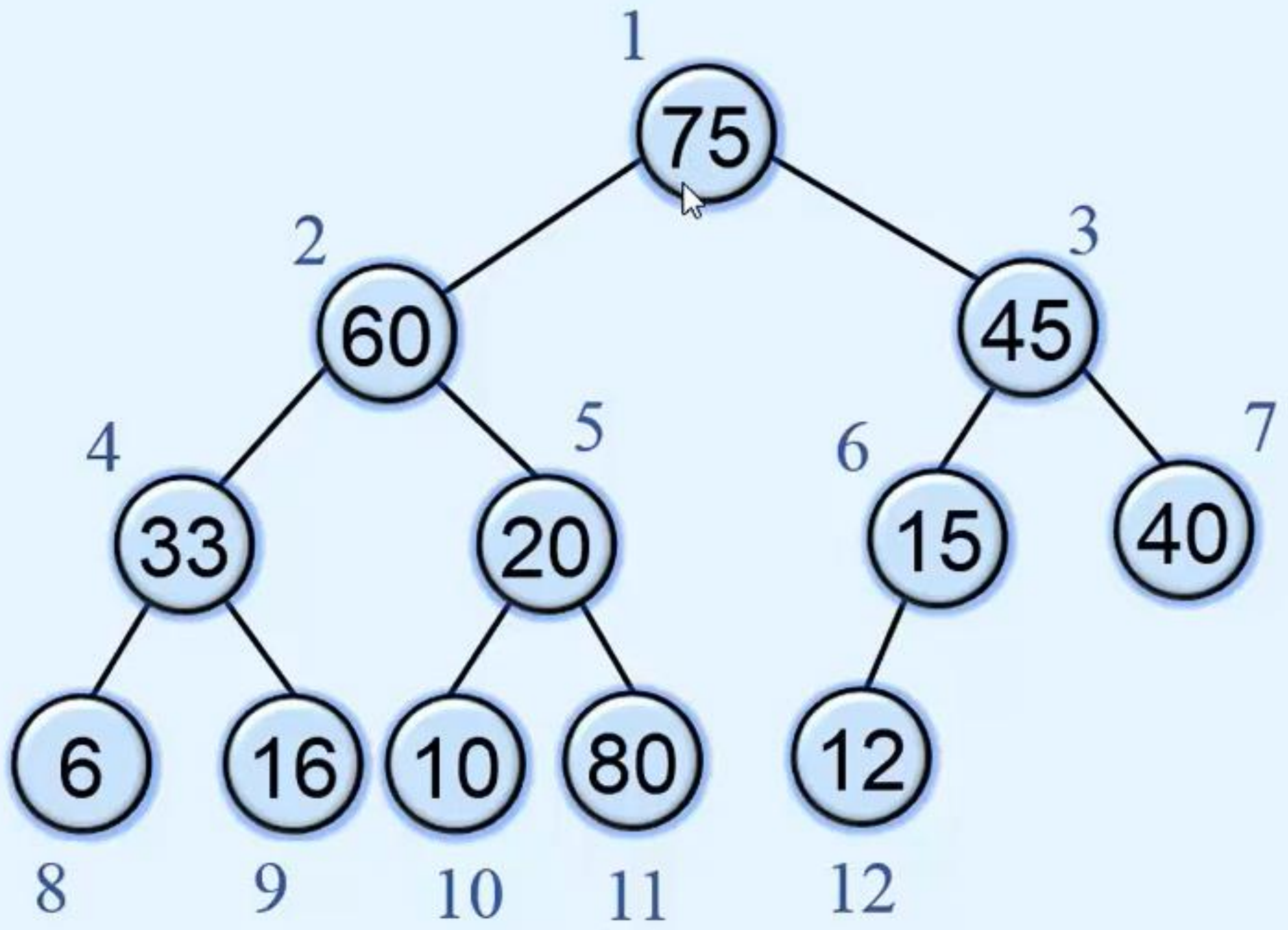
60	33	45	6	20	15	40	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



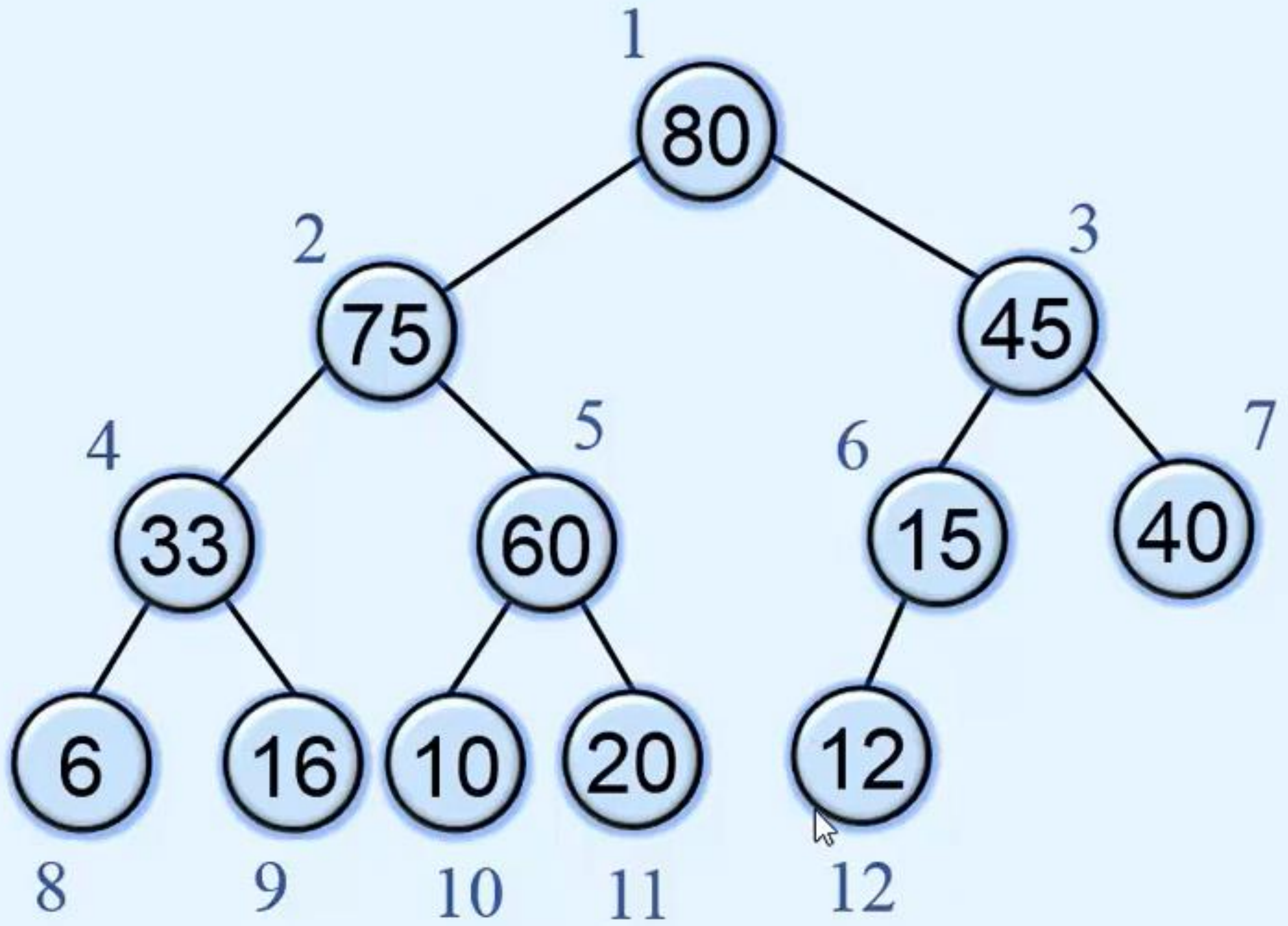
60	33	45	16	20	15	40	6	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



75	60	45	33	20	15	40	6	16	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



80	75	45	33	60	15	40	6	16	10	20	12
1	2	3	4	5	6	7	8	9	10	11	12



DS

Bottom up Approach

Consider that the array represents a complete binary tree

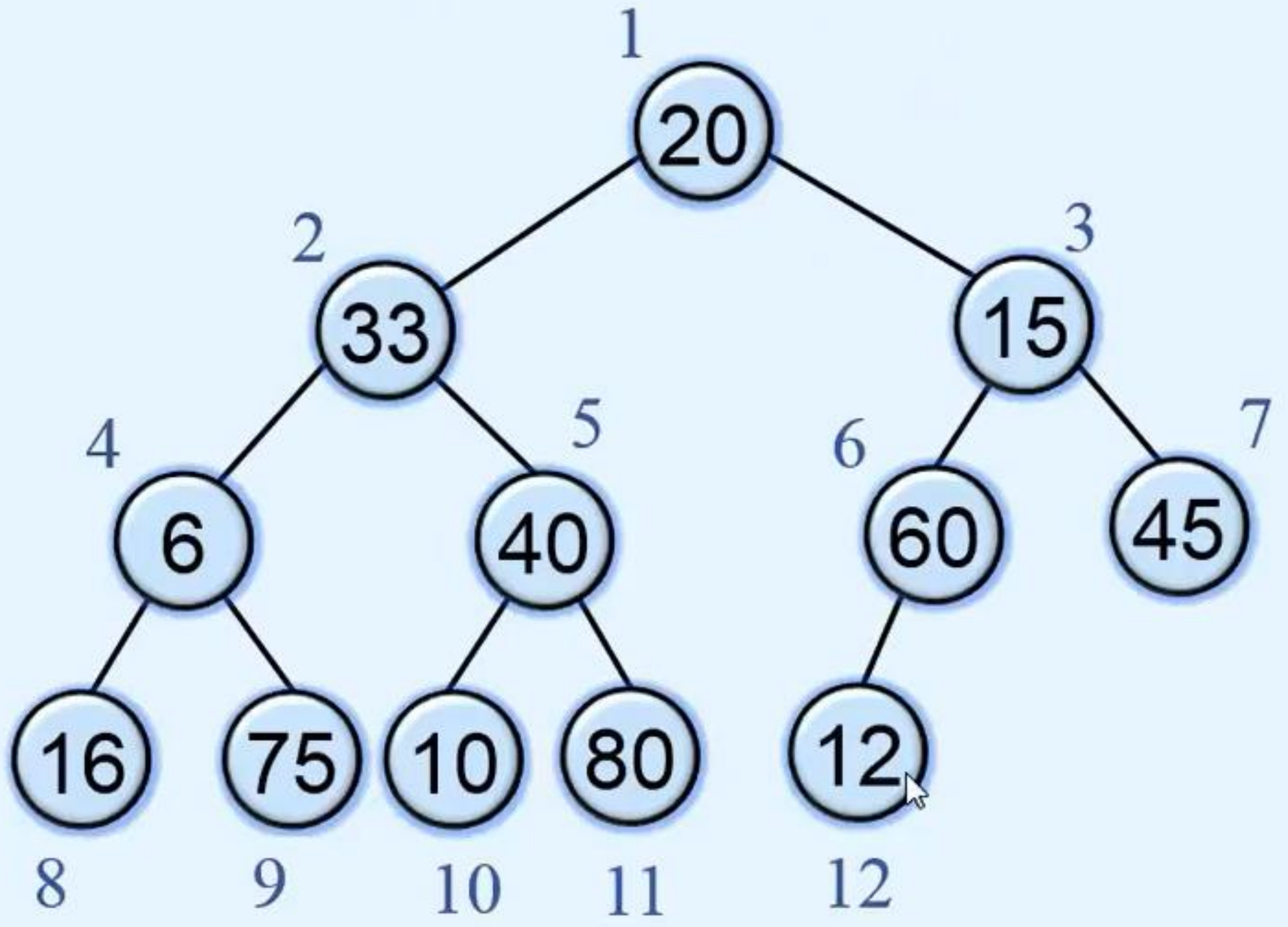
Start from first non leaf node

Call restoreDown for each node of the tree till root node

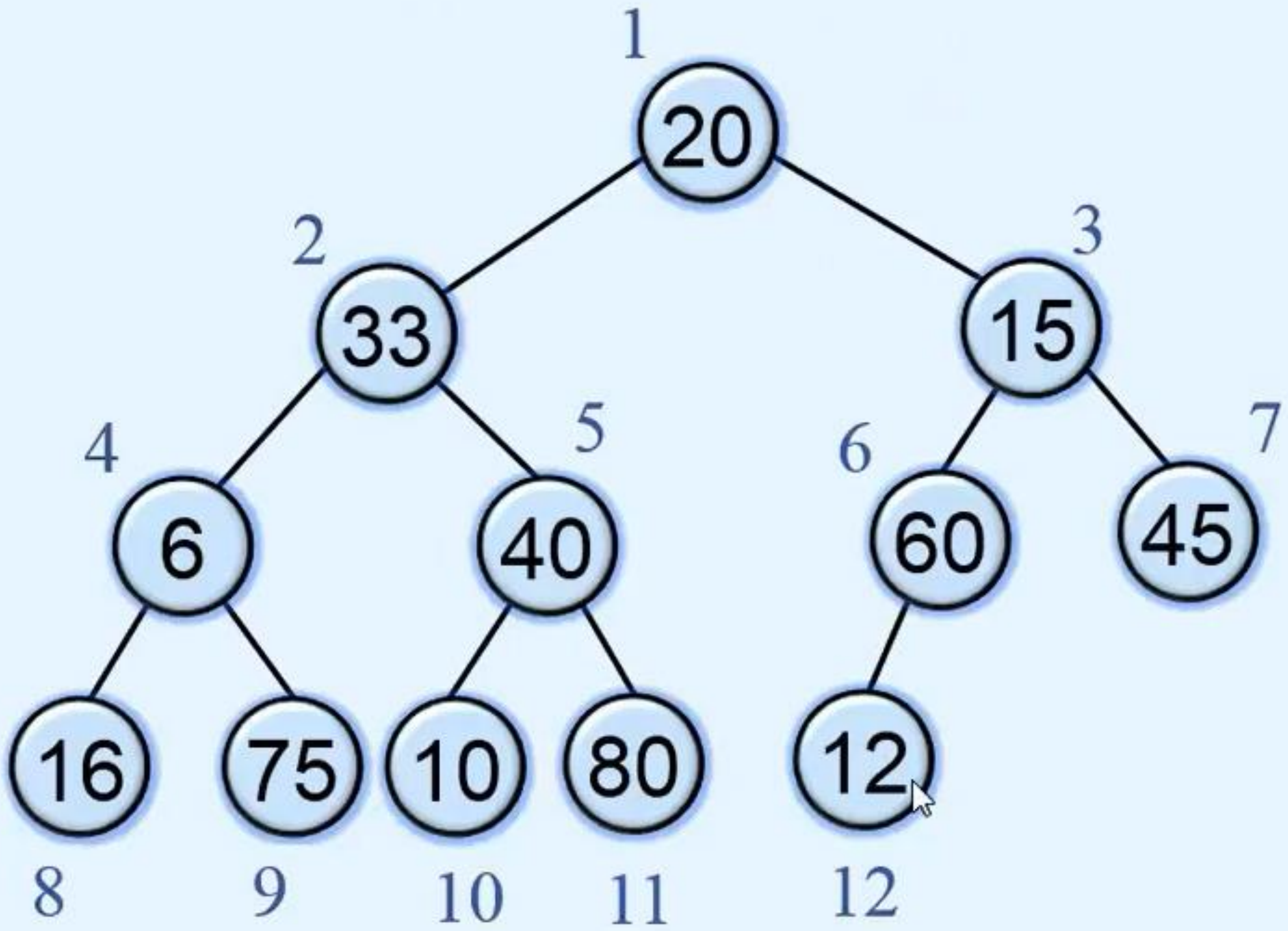
First non leaf node – index floor(n/2) = x

Call restoreDown for a[x], a[x-1], a[x-2].........a[2], a[1]

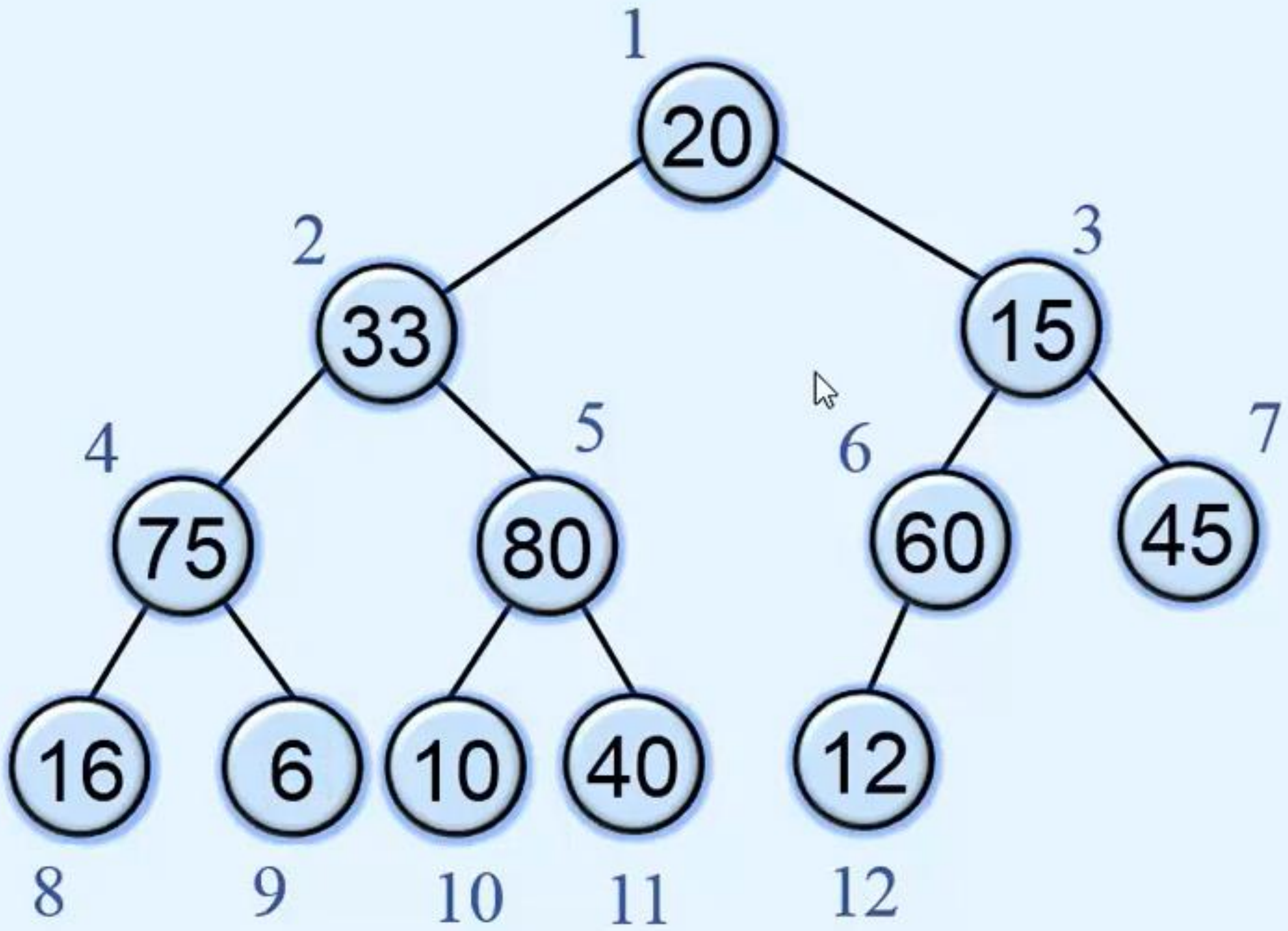
20	33	15	6	40	60	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



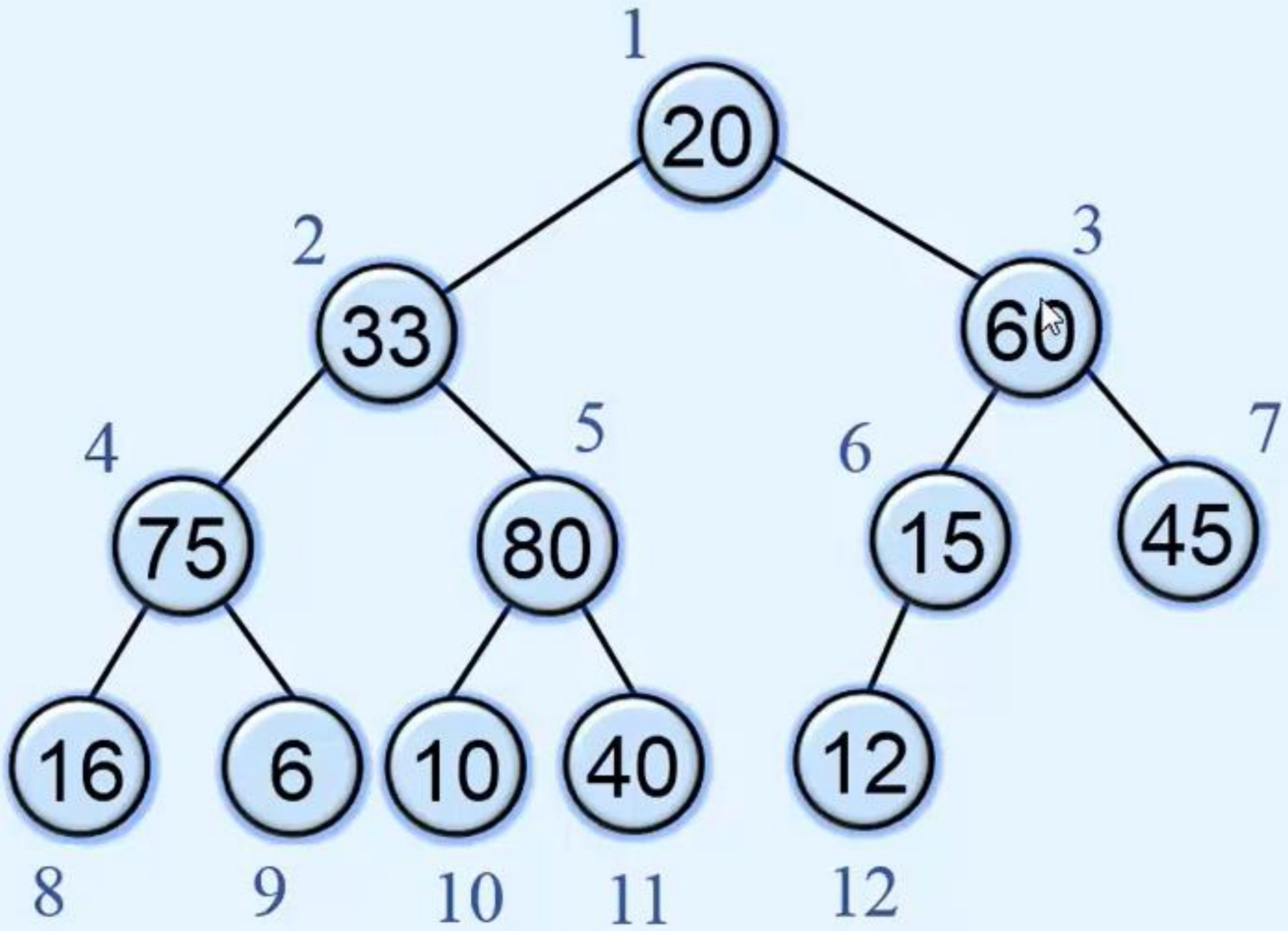
20	33	15	6	40	60	45	16	75	10	80	12
1	2	3	4	5	6	7	8	9	10	11	12



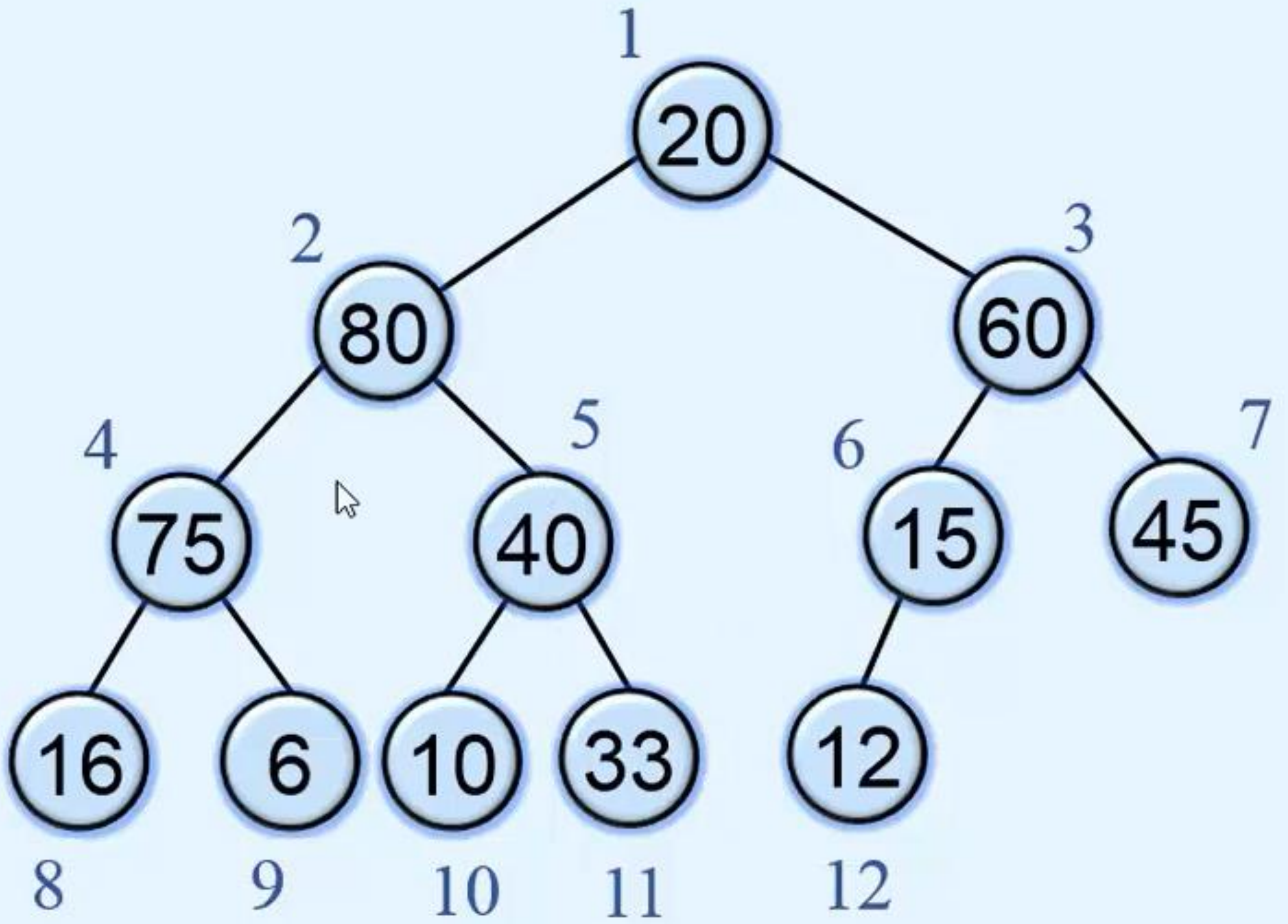
20	33	15	75	80	60	45	16	6	10	40	12
1	2	3	4	5	6	7	8	9	10	11	12



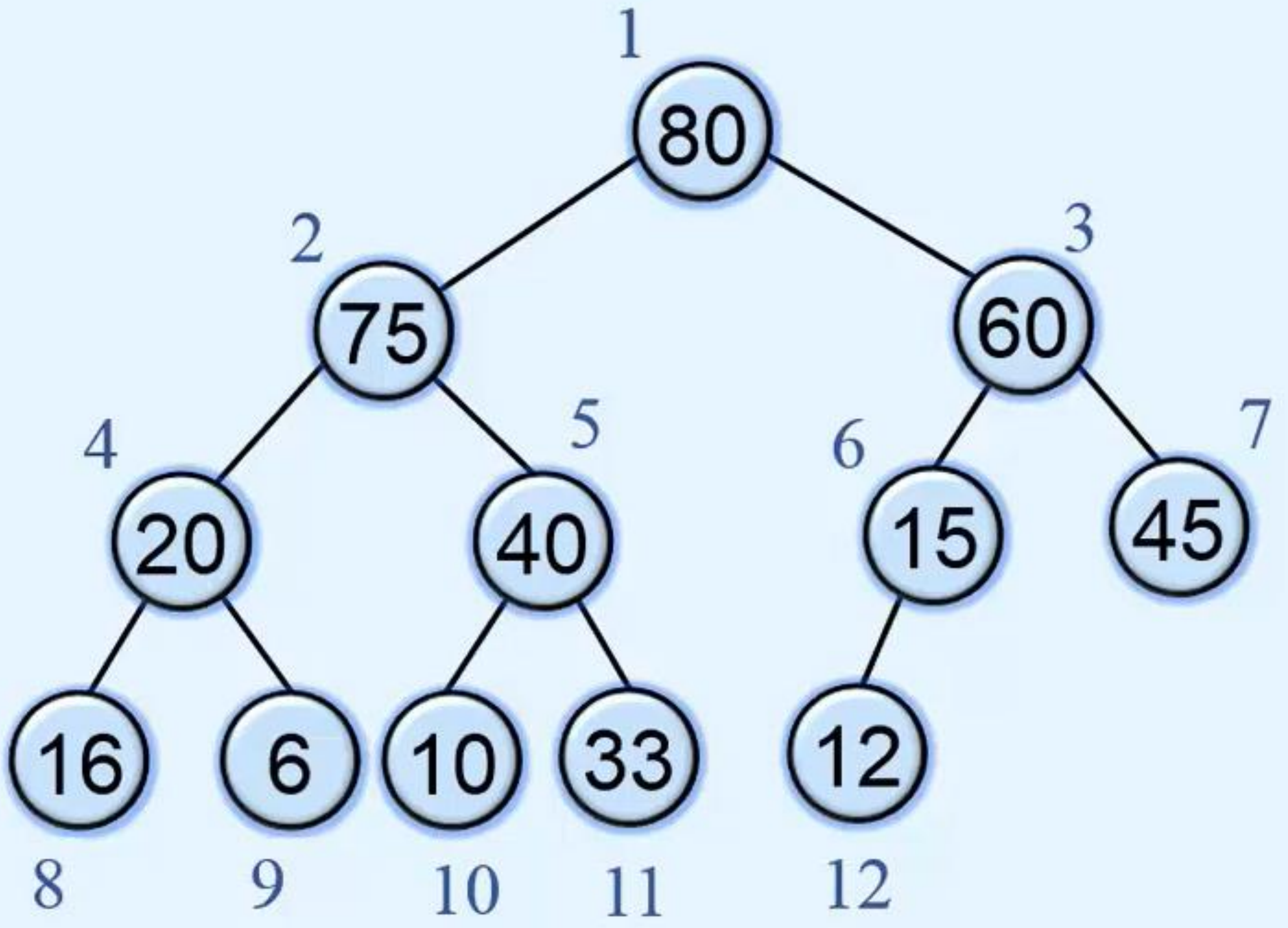
20	33	60	75	80	15	45	16	6	10	40	12
1	2	3	4	5	6	7	8	9	10	11	12



20	80	60	75	40	15	45	16	6	10	33	12
1	2	3	4	5	6	7	8	9	10	11	12



80	75	60	20	40	15	45	16	6	10	33	12
1	2	3	4	5	6	7	8	9	10	11	12



Applications of Heap

Used in problems where largest(or smallest) value has to be found

- Selection Algorithm

  Finding kth largest element

  Heap is built and then root is deleted k times

- Implementation of Priority Queue

  Queue - Insertion is O(1) and deletion is O(n)

  Sorted List - Insertion is O(n) and deletion is O(1)

  Heap - Insertion and deletion is O(log n)

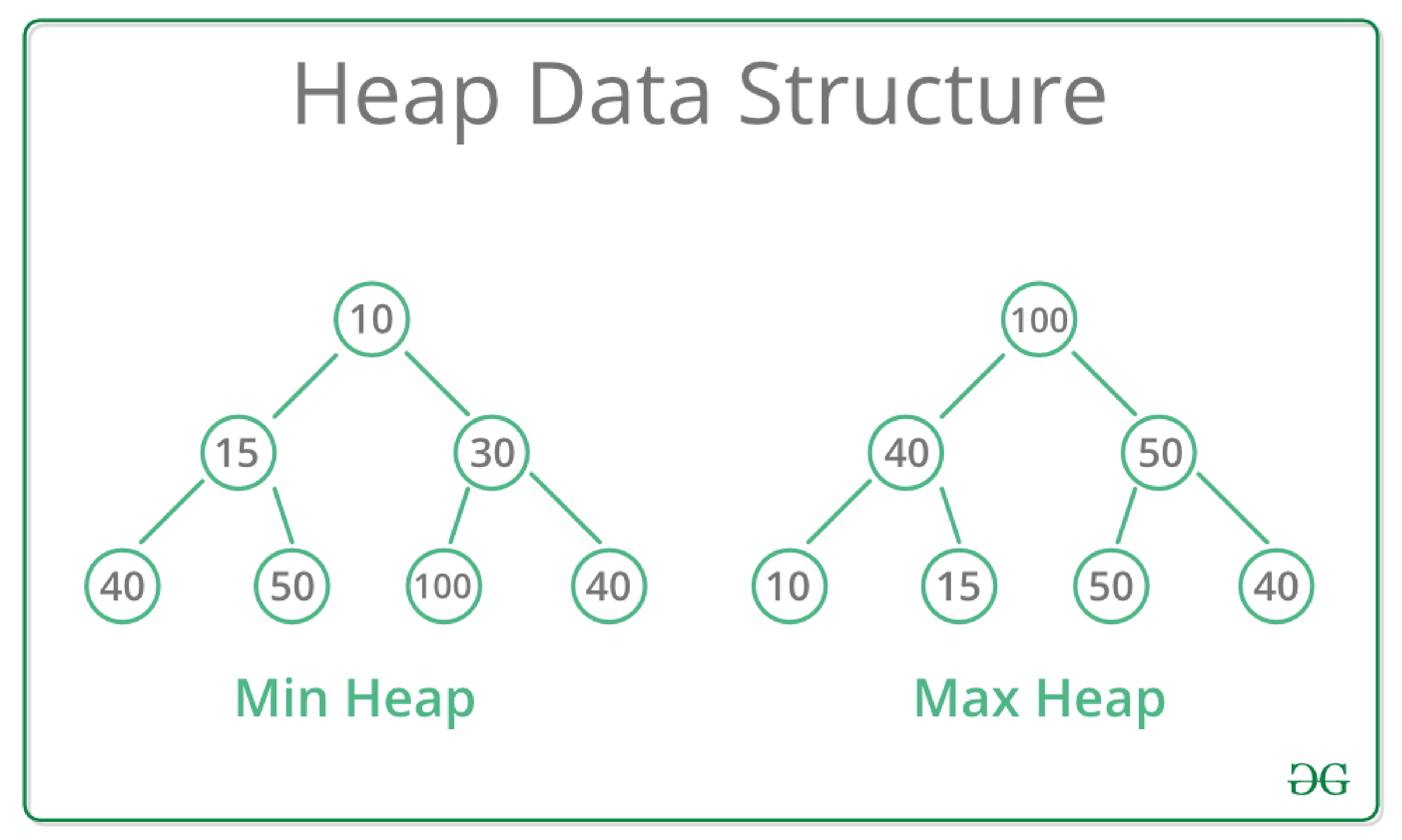
- Heap Sort

OPERATION	TIME COMPLEXITY		SPACE COMPLEXITY
Insertion	Best Case:	O(1)	O(1)
	Worst Case:	O(logN)	
	Average Case:	O(logN)	
Deletion	Best Case:	O(1)	O(1)
	Worst Case:	O(logN)	
	Average Case:	O(logN)	
Searching	Best Case:	O(1)	O(1)
	Worst Case:	O(N)	
	Average Case:	O(N)	
Max Value	In MaxHeap:	O(1)	O(1)
	In MinHeap:	O(N)	
Min Value	In MinHeap:	O(1)	O(1)
	In MaxHeap:	O(N)	
Sorting	All Cases:	O(NlogN)	O(1)
Creating a Heap	By Inserting all elements:	O(NlogN)	O(N)
	Using Heapify	O(N)	O(1)

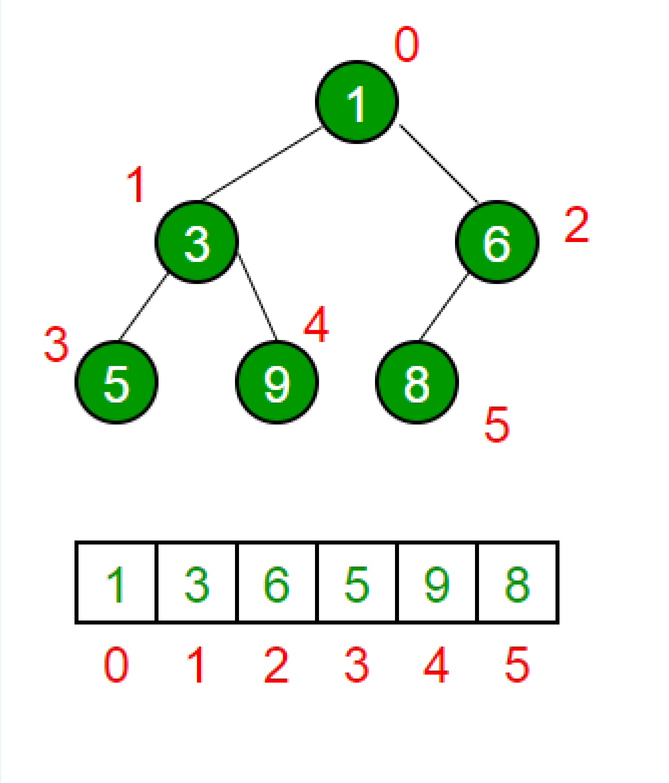
Other variants of Heap

1. Binary Heap
2. Min Heap
3. Max Heap
4. Binomial Heap
5. Fibonacci Heap
6. D-ary Heap
7. Pairing Heap
8. Leftist Heap
9. Skew Heap
10. B-Heap

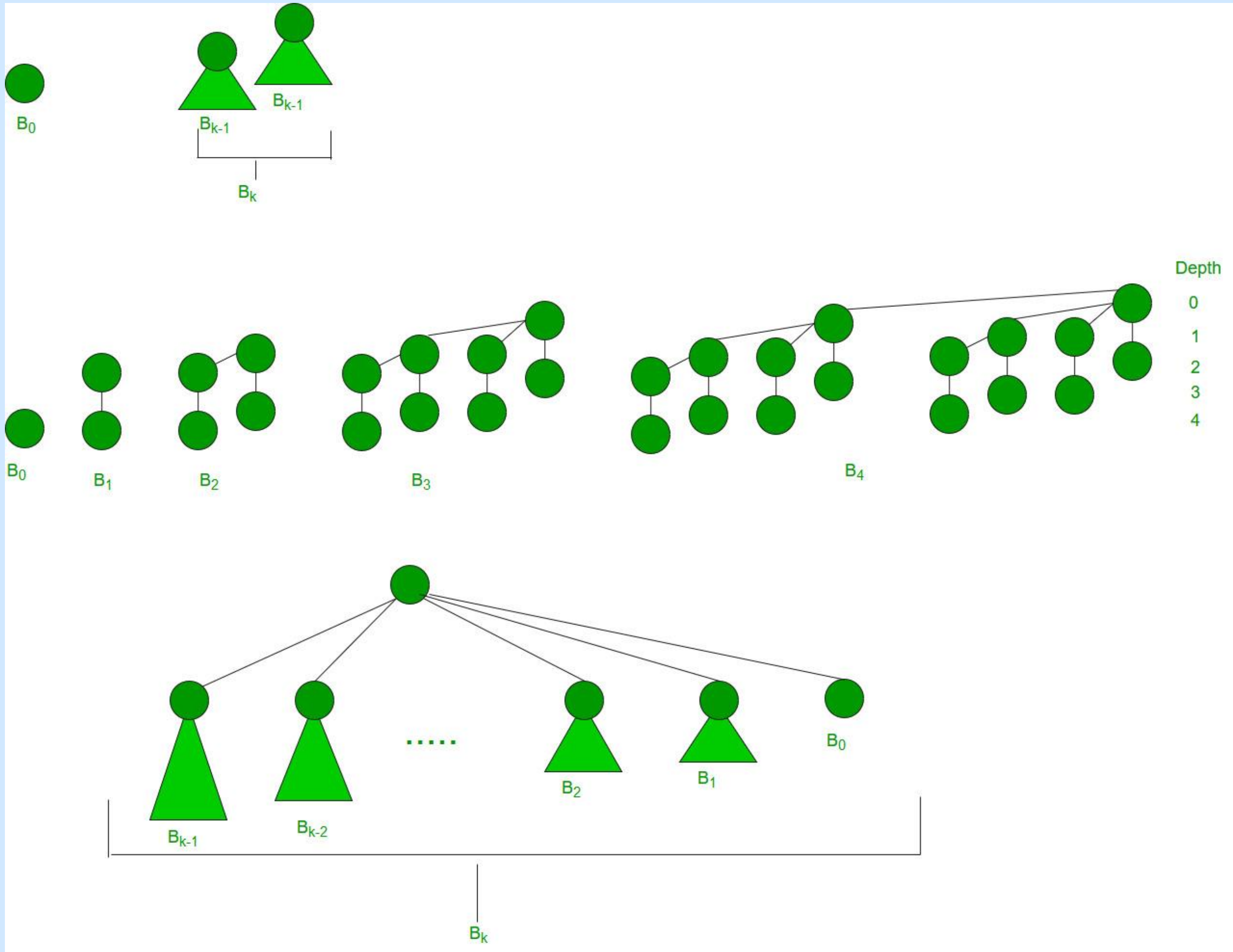
Max Heap   Min Heap



Binary Heap

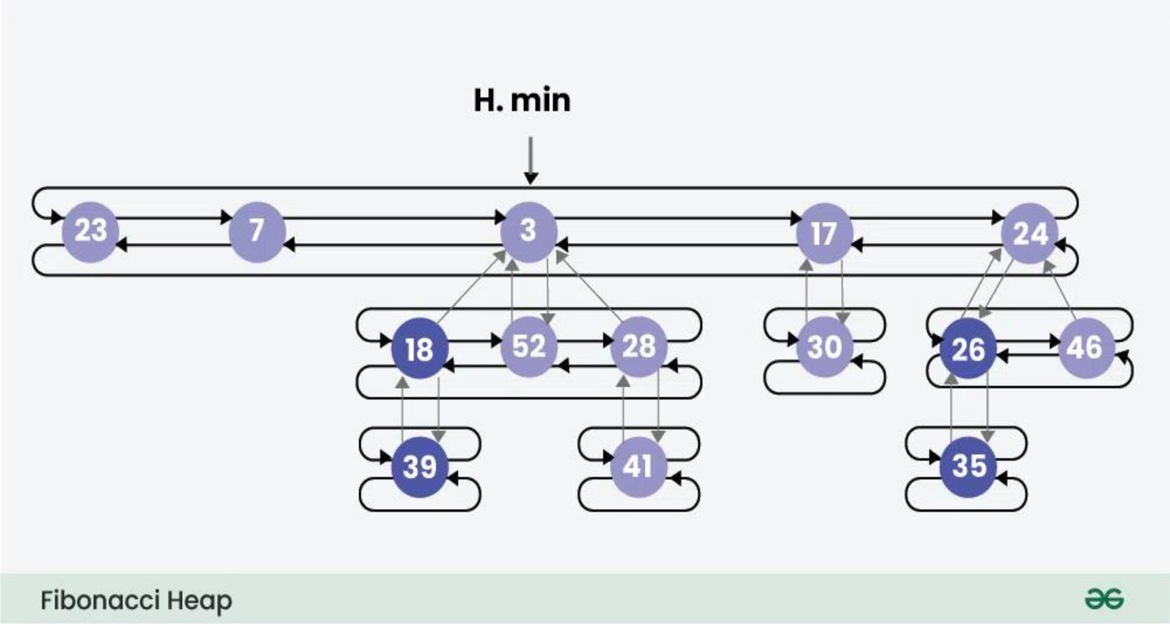


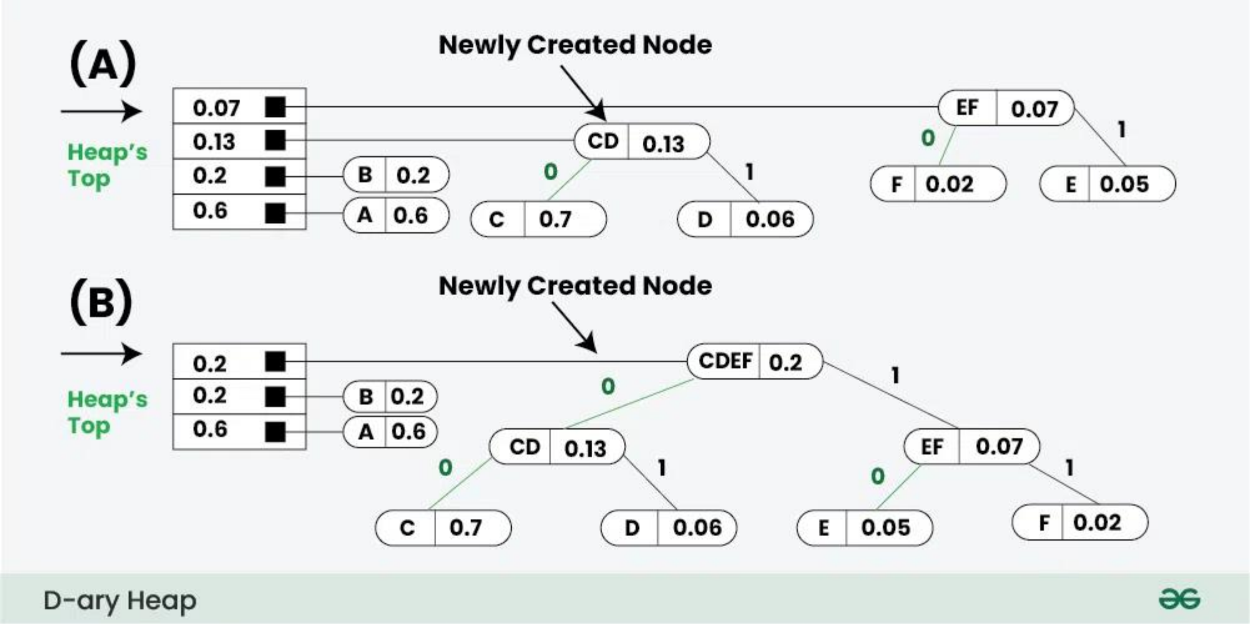
Binomial Heap



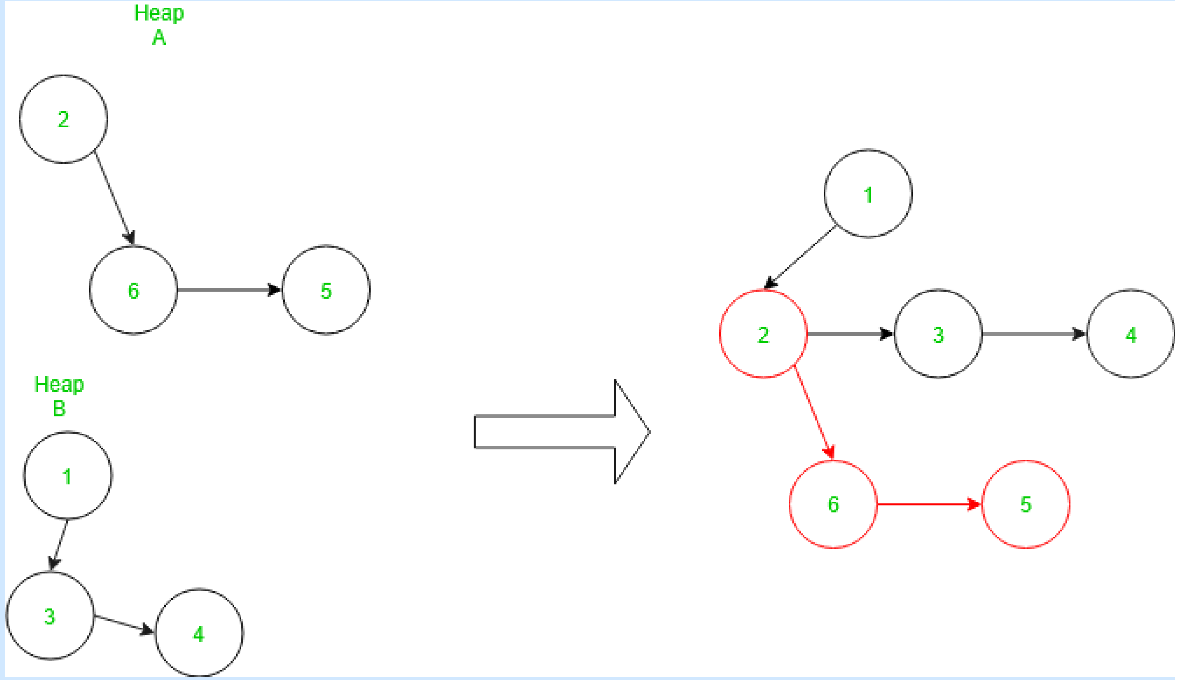
Fibonacci Heap

D-ary Heap

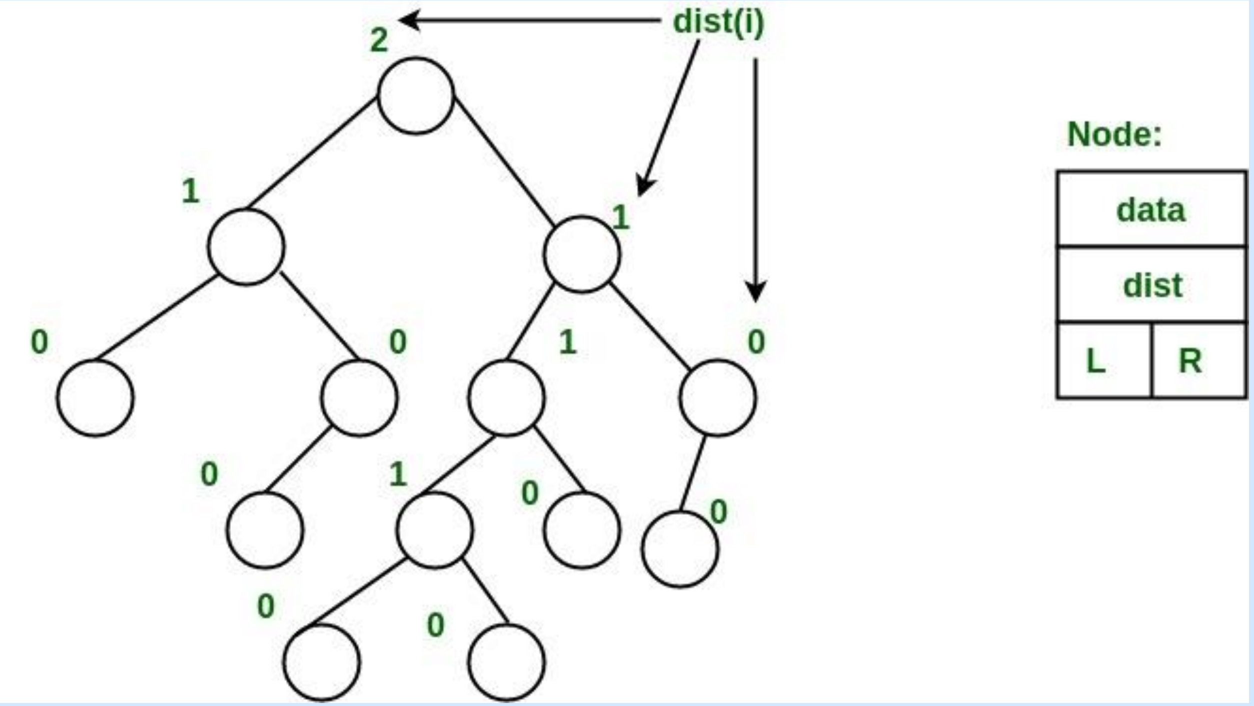




Pairing Heap



Leftist Heap



Skew Heap

B- Heap