09

Heap Data Structure

DATA STRUCTURE LAB

SESSION - 09

HEAP

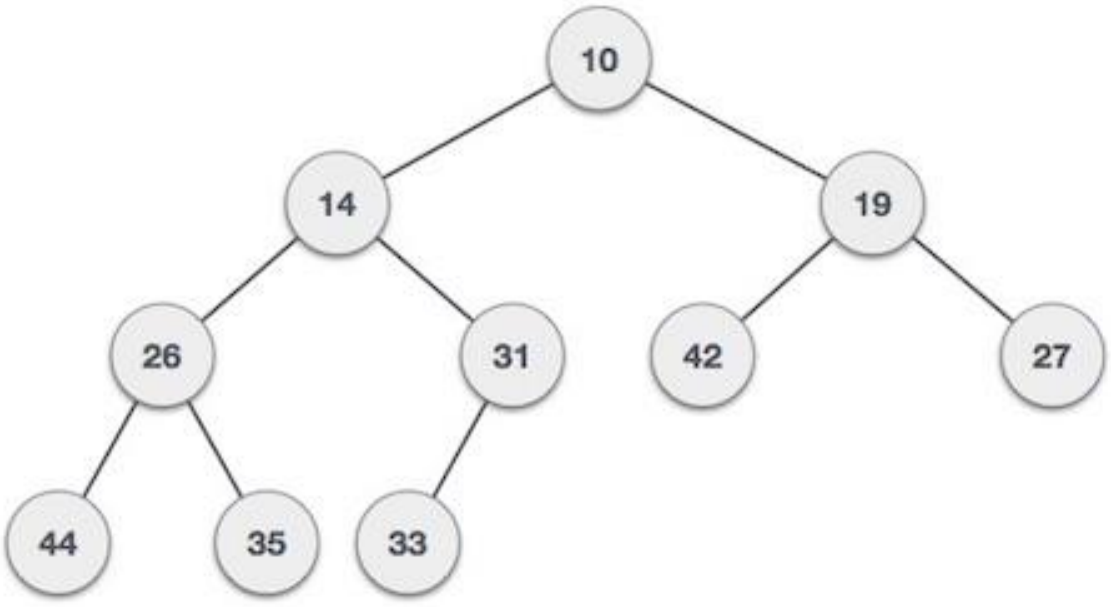
A special case of complete binary where root-node element is compared with its child and arranged accordingly. Must be a complete binary tree.

→ Root >= left – right
[max heap]

→ Root <= left – right
[min heap]

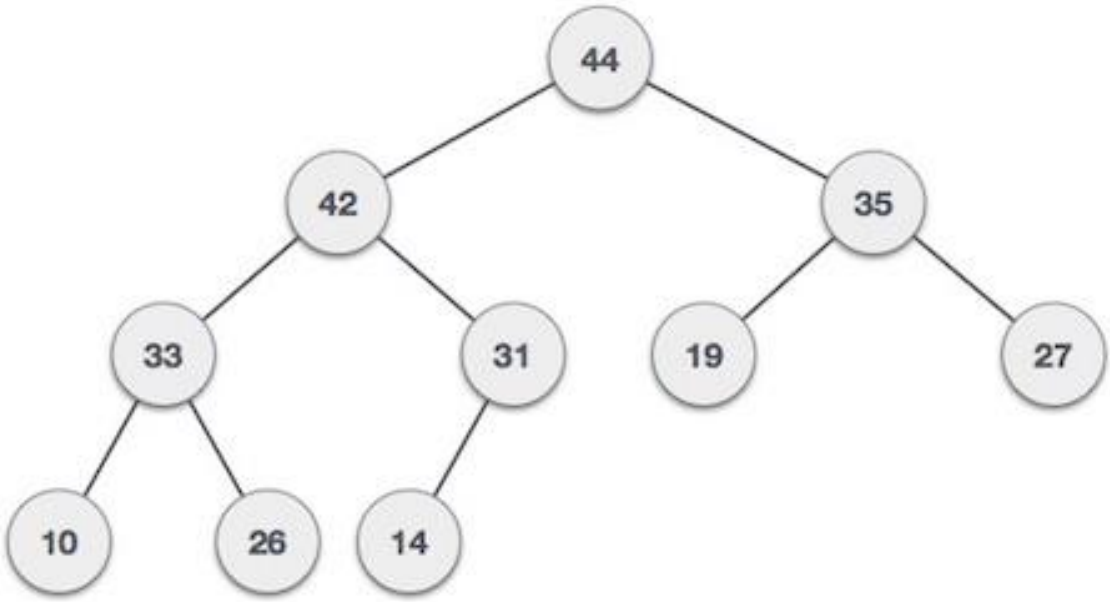
Min Heap:

Value of the root node is less than or equal to either of its children.



Max Heap:

Value of the root node is greater than or equal to either of its children.



Max Heap Construction:

Step 1 – Create a new node at the end of heap.

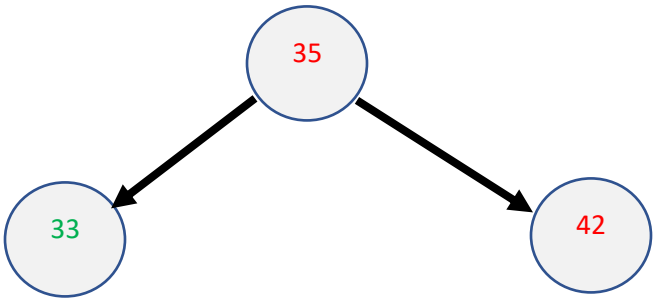
Step 2 – Compare the value of this child node with its parent.

Step 3 – If value of parent is less than child, then swap them.

Step 4 – Repeat step 2 & 3 until Heap property holds.

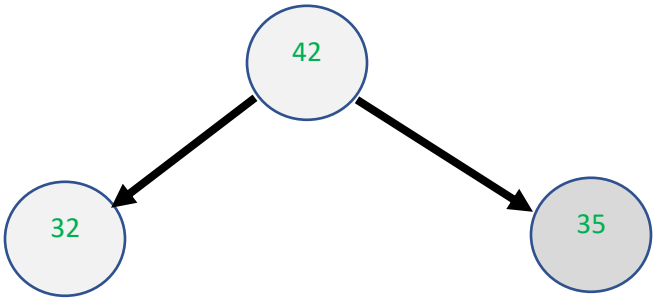
Input → 35 33 42 10 14 19 27 44

Input → 35 33 42 10 14 19 27 44

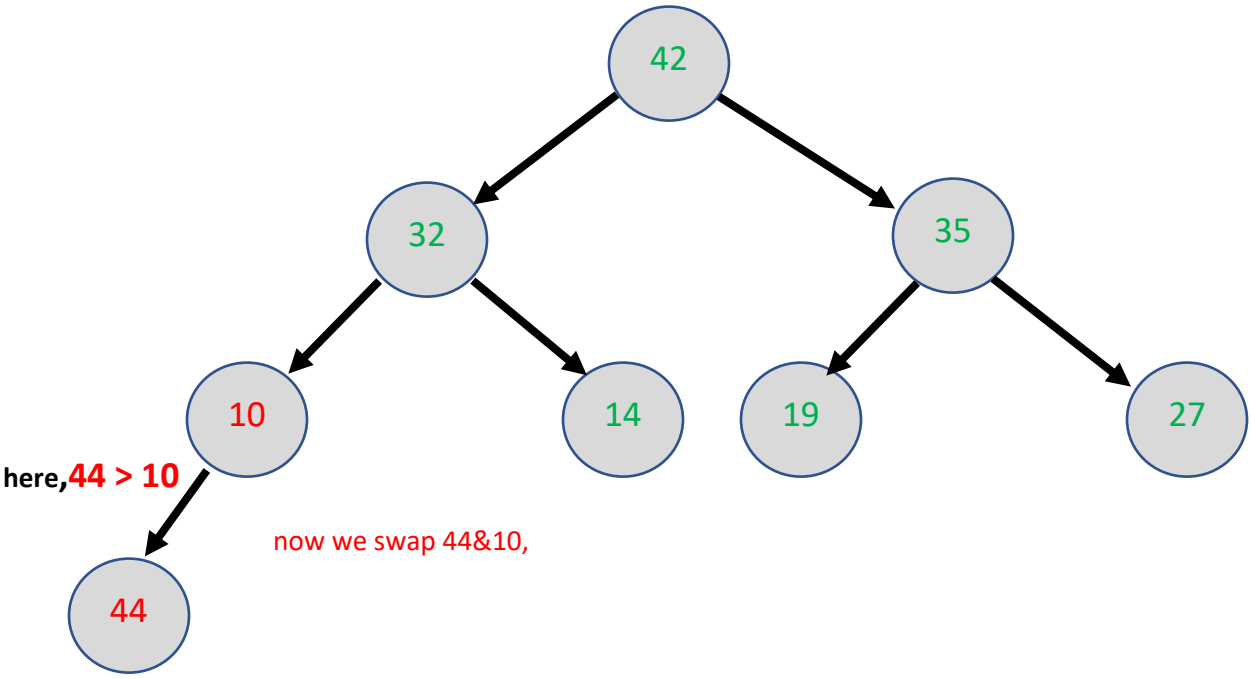


Here, "**42 > 35**"

Now we swap 42&35.

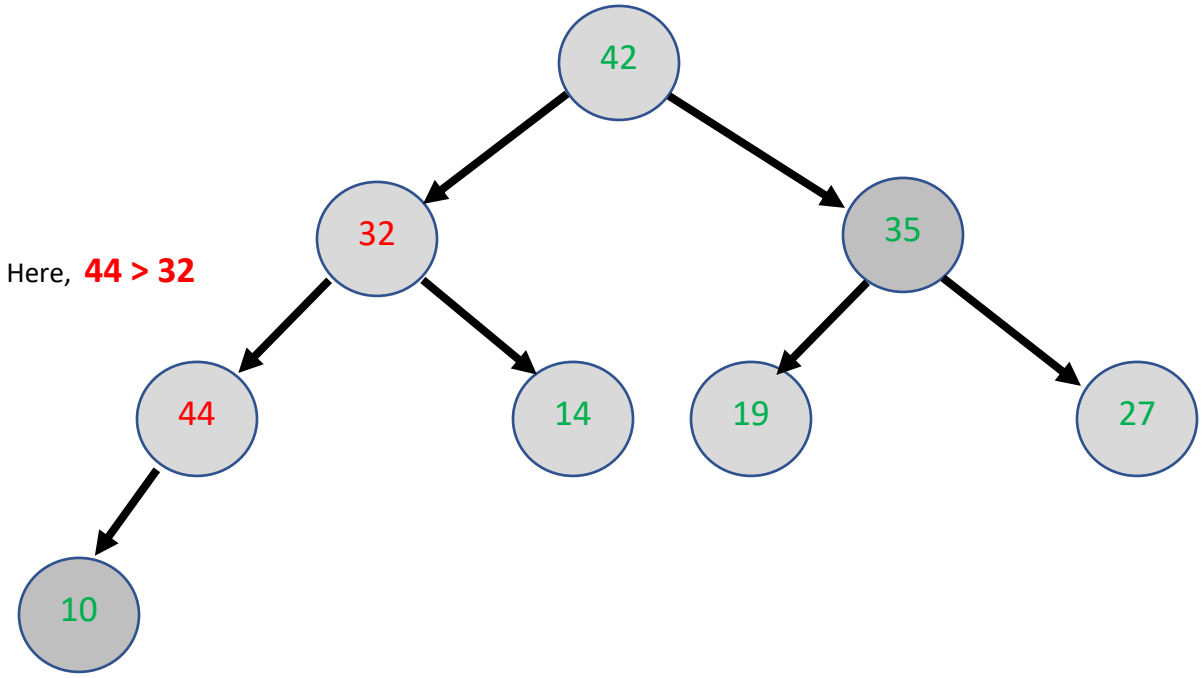


Input → 35 33 42 10 14 19 27 44



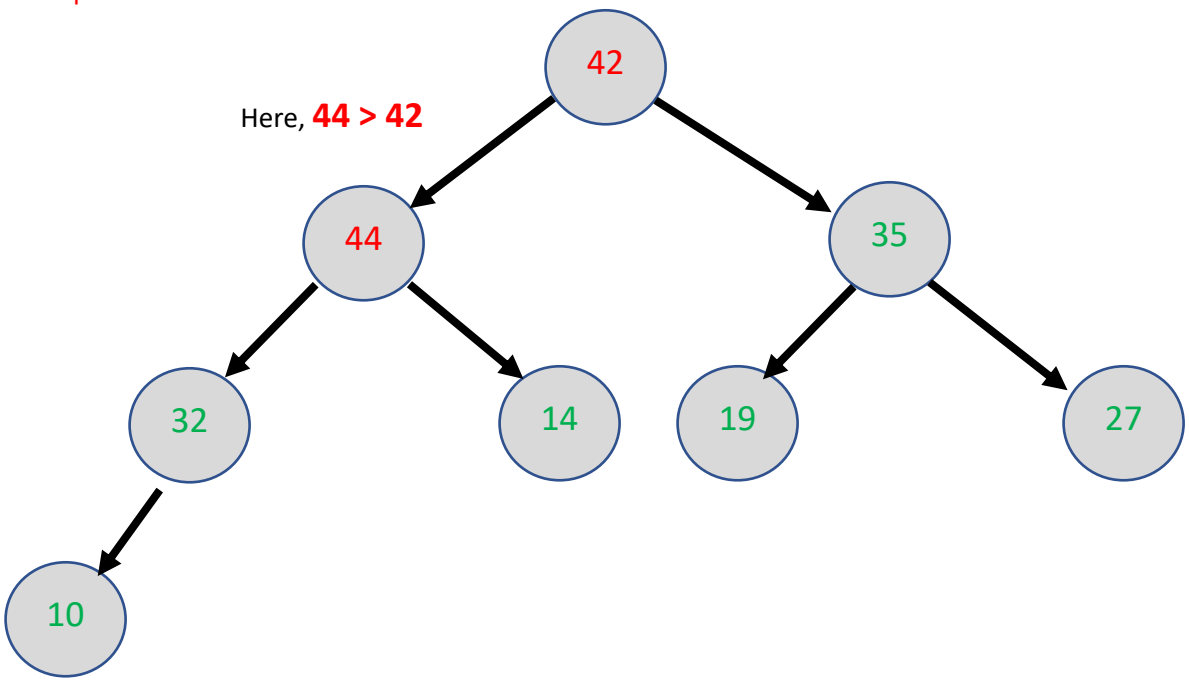
here,44 > 10

now we swap 44&10,



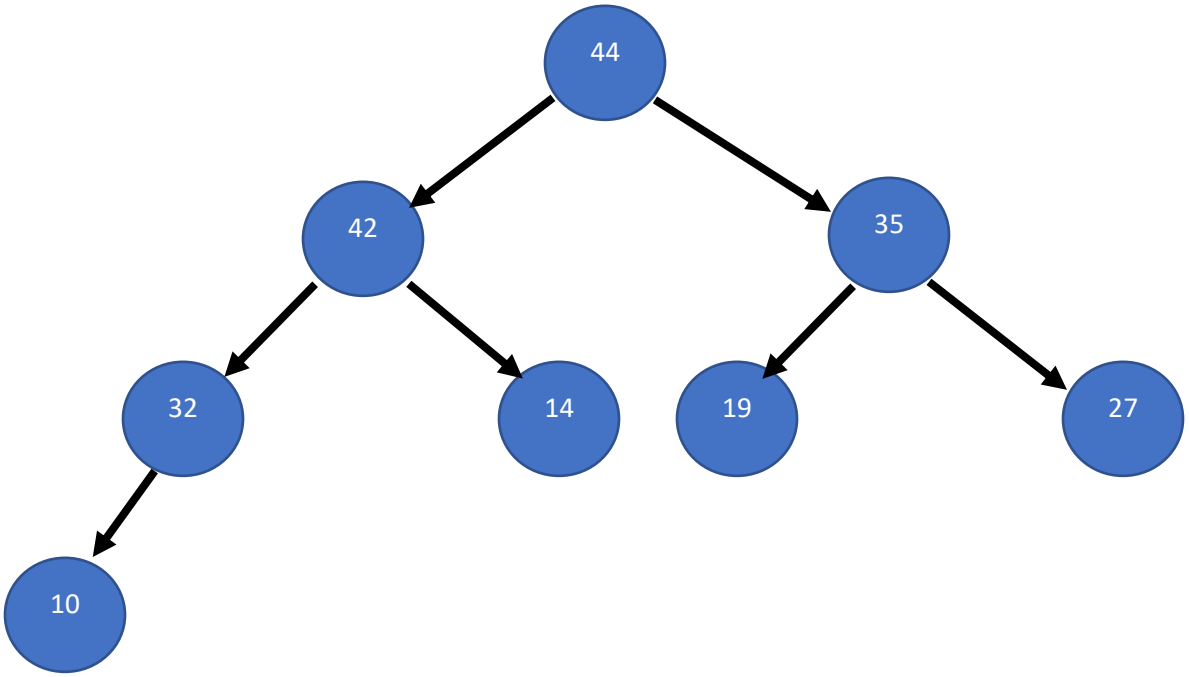
Here, **44 > 32**

Now we swap 44&32



Here, **44 > 42**

Now we swap 44&42,


44
42
35
32
14
19
27
10

*****

Time Complexity:

Time complexity of an algorithm signifies the total time required by the program to run to completion. Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm.

Calculate T(n) : no of operations

1. int x = 0; 2 operations (declaration & assign)

   int sum = 0; 2 operations (declaration & assign)

   T(n) = 2+2

   = 4

2. int i, sum, n; 3 operations (3 declaration)
   sum = 0, n=5; 2 operations (assign)
   for(i = 0; i < n; i++) n operation [looping n times]
   sum + =i; 2 operations [inside each loop]

   T(n) = 3+2+(n*2)

   = 5+2n

The time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm. Using these notation we can represent the 3 cases.

- ***O Notation*** (The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.)

- ***Ω Notation*** (The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.)
- ***θ Notation*** (The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.)

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components –

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept –

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

```
{
   int A,B,C;

    C=A+B+10;
}
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

****