What is Heap Data Structure?

A Heap is a special ***Tree-based Data Structure*** *in which the tree is a complete binary tree.*

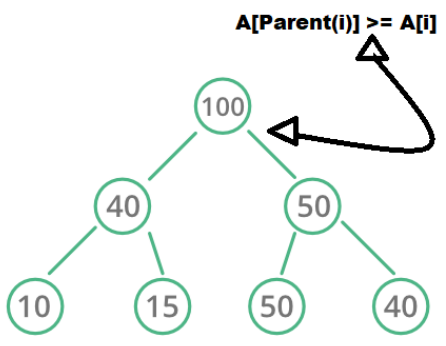
Types of heaps:

Max-Heap:

In this heap, the value of the root node must be the greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.

The total number of comparisons required in the max heap is according to the height of the tree.

The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

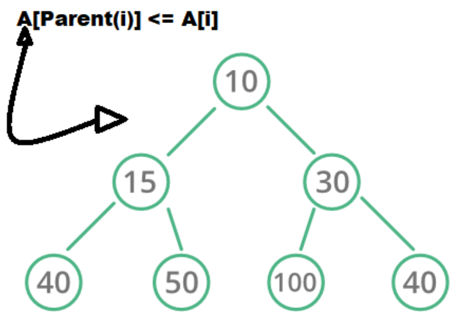


Max Heap

Min-Heap:

In this heap, the value of the root node must be the smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.

The total number of comparisons required in the min heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).



Min Heap

Implementation of Heap Data Structure:-

The following code shows the implementation of a **max-heap**.
Let’s understand the **maxHeapify** function in detail:-
maxHeapify is the function responsible for restoring the property of the Max Heap. It arranges the node **i**, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, **arr[]** representing the complete binary tree. The left and the right child of **i**th node are in indices **2*i+1** and **2*i+2**.
2. We set the index of the current element, **i**, as the ‘MAXIMUM’.
3. If **arr[2 * i + 1] > arr[i]**, i.e., the left child is larger than the current value, it is set as ‘MAXIMUM’.
4. Similarly if **arr[2 * i + 2] > arr[i]**, i.e., the right child is larger than the current value, it is set as ‘MAXIMUM’.
5. Swap the ‘MAXIMUM’ with the current element.
6. Repeat steps **2 to 5** till the property of the heap is restored.

Operations Supported by Heap:

- *Operations supported by **min – heap** and **max – heap** are same.*
- *The difference is just that min-heap contains minimum element at root of the tree and max – heap contains the maximum element at the root of the tree.*

Heapify:

It is the process to rearrange the elements to maintain the property of the heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations on that node.
It takes **O(log N)** to balance the tree.

- For **max-heap,** it **balances** in such a way that the maximum element is the root of that binary tree and

- For **min-heap,** it balances in such a way that the minimum element is the root of that binary tree.

Insertion:

If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the **heapify** operation so that it maintains the property of the heap.
This operation also takes **O(logN)** time.

Process of Insertion: Elements can be inserted to the heap following a similar approach as discussed above for deletion. The idea is to:

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify** this newly inserted element following a bottom-up approach.

Illustration:

```
Suppose the Heap is a Max-Heap as:

      10

    /    \

   5      3

  / \

 2   4
```

The new element to be inserted is 15.

Process:
Step 1: Insert the new element at the end.

```
      10
    /    \
   5      3
  / \    /
 2   4  15
```

Step 2: Heapify the new element following bottom-up approach.

```
-> 15 is more than its parent 3, swap them.
       10
    /    \
   5      15
  / \    /
```

```
 2   4  3

-> 15 is again more than its parent 10, swap them.
         15
       /    \
     5       10
    / \     /
 2   4  3

Therefore, the final heap after insertion is:
         15
       /    \
     5       10
    / \     /
 2   4  3
```
Implementation:

```
// C++ program to insert new element to Heap

#include <iostream>
using namespace std;

#define MAX 1000 // Max size of Heap

// Function to heapify ith node in a Heap
// of size n following a Bottom-up approach
void heapify(int arr[], int n, int i)
{
    // Find parent
    int parent = (i - 1) / 2;

    if (arr[parent] > 0) {
        // For Max-Heap
        // If current node is greater than its parent
        // Swap both of them and call heapify again
        // for the parent
        if (arr[i] > arr[parent]) {
            swap(arr[i], arr[parent]);

            // Recursively heapify the parent node
            heapify(arr, n, parent);
        }
    }
}

// Function to insert a new node to the Heap
```

```cpp
void insertNode(int arr[], int& n, int Key)
{
    // Increase the size of Heap by 1
    n = n + 1;

    // Insert the element at end of Heap
    arr[n - 1] = Key;

    // Heapify the new node following a
    // Bottom-up approach
    heapify(arr, n, n - 1);
}

// A utility function to print array of size n
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    cout << "\n";
}

// Driver Code
int main()
{
    // Array representation of Max-Heap
    // 10
    //    /  \
    // 5     3
    //   / \
    // 2   4
    int arr[MAX] = { 10, 5, 3, 2, 4 };

    int n = 5;

    int key = 15;

    insertNode(arr, n, key);

    printArray(arr, n);
    // Final Heap will be:
    // 15
    //    /   \
    // 5      10
    //   / \   /
    // 2   4 3
    return 0;
}
```

Heap deletion

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

Illustration:

```
Suppose the Heap is a Max-Heap as:
      10

     /    \

    5      3

   / \

  2   4


The element to be deleted is root, i.e. 10.

Process:
The last element is 4.

Step 1: Replace the last element with root, and delete it.
      4
     /    \
    5      3
   /
  2

Step 2: Heapify root.
Final Heap:
      5
     /    \
    4      3
   /
  2
```

```
#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is
// an index of arr[] and n is the size of heap
```

```
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Function to delete the root from Heap
void deleteRoot(int arr[], int& n)
{
    // Get the last element
    int lastElement = arr[n - 1];

    // Replace root with last element
    arr[0] = lastElement;

    // Decrease size of heap by 1
    n = n - 1;

    // heapify the root node
    heapify(arr, n, 0);
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver Code
```

```
int main()
{
    // Array representation of Max-Heap
    //     10
    //    /  \
    //   5    3
    //  / \
    // 2   4
    int arr[] = { 10, 5, 3, 2, 4 };

    int n = sizeof(arr) / sizeof(arr[0]);

    deleteRoot(arr, n);

    printArray(arr, n);

    return 0;
}
```

Heap sort

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

- *Build a heap from the given input array.*
- *Repeat the following steps until the heap contains only one element:*
    - *Swap the root element of the heap (which is the largest element) with the last element of the heap.*
    - *Remove the last element of the heap (which is now in the correct position).*
    - *Heapify the remaining elements of the heap.*
- *The sorted array is obtained by reversing the order of the elements in the input array.*

Huffman Coding | Greedy Algo

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Message: BCCABBDDAECCBBAEDDCC

Char	count	code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000

Methods

1. Arrange the symbols in increasing order of their count or frequency, horizontally

2. Merge two smallest one and make one node and make a tree

3. repeat the steps 1 and 2

3. start traversing the tree from the root node to that particular leaf node

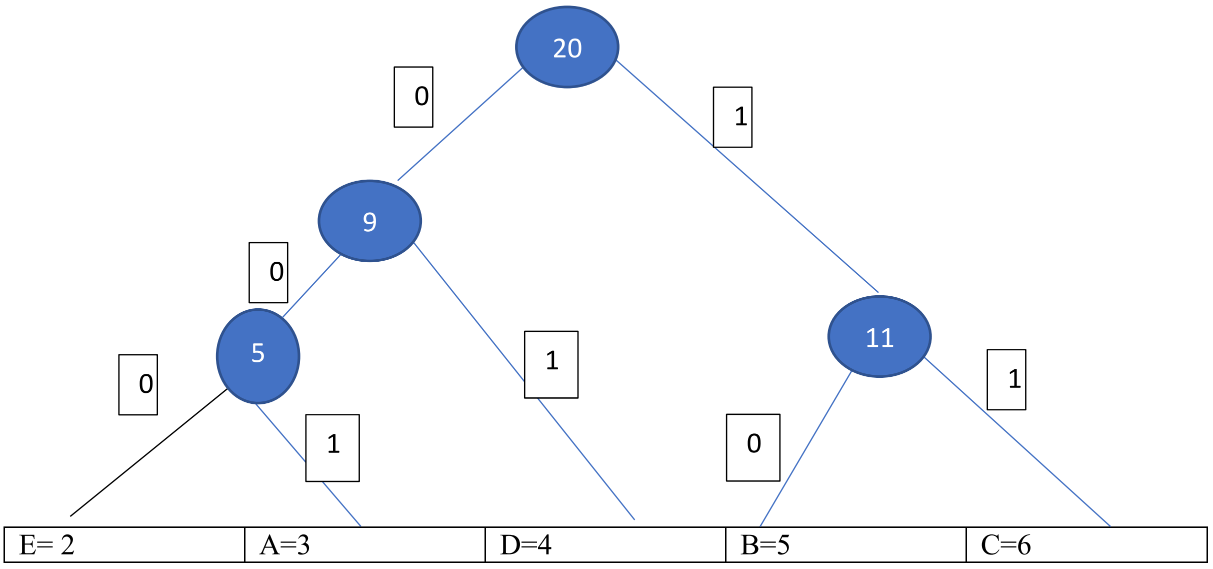


Fig. Optimal merge pattern tree

If the message is given in the form of the code, we can decode the message by traversing the tree from the root node to the leaf node

For example:

0011111011011111001

Traverse the tree from the leaf node:

001=A

11=C

11=C

01=D

10=B……

AVL trees

*An **AVL tree** defined as a self-balancing **Binary Search Tree** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*

Machine Scheduling

job	*J1*	*J2*	*J3*	*J4*	*J5*	
profit	*20*	*15*	*10*	*5*	*1*	
deadline	*2*	*2*	*1*	*3*	*3*	

Each job takes a single fix unit time

Step 1 first, sort the jobs in decreasing order of their profits

Step 2 Then find the highest deadline among all deadline

Step 3 Next, we need to assign time slots to individual job ids

Step 4 first, check if the maximum possible time slot for the job, i.e., its deadline, is assigned to another job or not. If it is not filled yet, assign the slot to the current job id.

Step 5 Otherwise, search foe any empty time slot less than the deadline of the current job. If such a slot is found, assign it to the current job id and move to the next job id.

Solution of the above problem:

0------------------------1------------------------2-----------------------3

J2	*J1*	*J4*

job	*J1*	*J2*	*J3*	*J4*	*J5*	*J6*	*J7*
profit	*35*	*30*	*25*	*20*	*15*	*12*	*5*

deadline	*3*	*4*	*4*	*2*	*3*	*1*	*2*

0-----------------1-------------------2------------------3---------------4

Slot 0   slot 1   slot 2   slot 3

20 j4	*25 j3*	*35 j1*	*J2 30*

Job considered	*slot*	*solution*	*profit*
-	*-*	*-*	*-*
J1	*[2,3]*	*J1*	*20*
J2	*[0,1][1,2]*	*J1,j2*	*20+15*
J3(x)	*[0,1][1,2]*	*J1,j2*	*20+15*
J4	*[0,1][1,2][2,3]*	*J1,j2,j4*	*20+15+5=40*

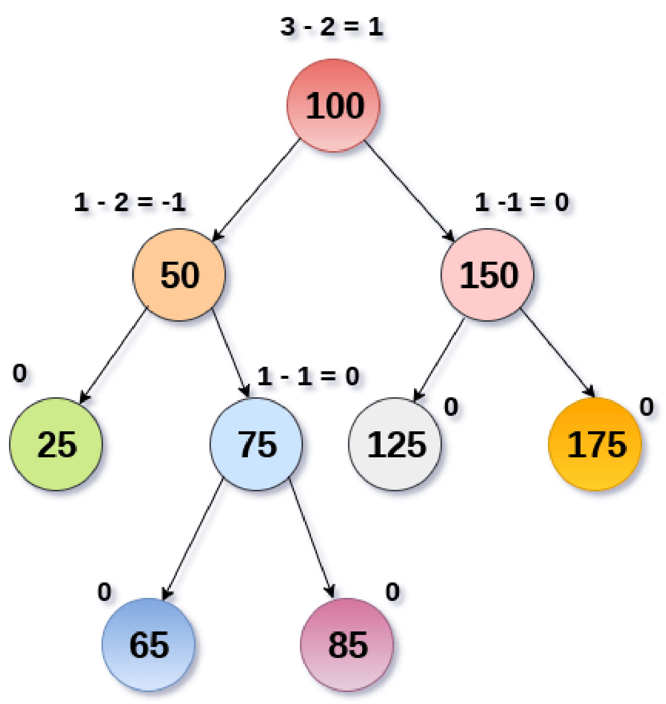
AVL Trees:

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

The tree is said to be balanced if the balance factor of each node is between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(subtree)) - height (right(subtree))



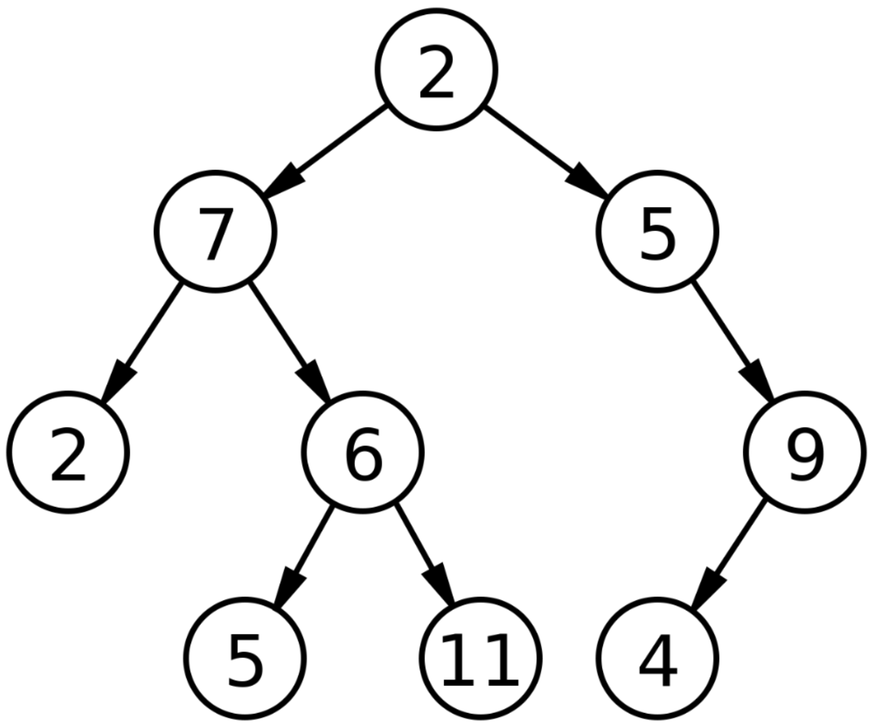
AVL Tree

Calculation of balance factor

- If we know the heights of the left and right child of a node then we can easily calculate the balance factor of the node.

Use recursion algorithm — **Post-Order Traversal** to compute the balance factors.

Example : Consider the following tree :



Using Post Order Traversal Algorithm -:

1. For root 2 we need heights of 7 and 5.
    a. For 7 need heights of 2 and 6.
        i. Height of 2 is 1 (or 0). So balance factor of 2 is 0 (leaf node).
        ii. For 6 need heights of 5 and 11. Both have height 1 and balance factor 0. So height of 6 is 2 and balance factor of 0.
        iii. So height of 7 is max(height(2),height(6))+1 = 3 and B.F = 1–2 = -1.
    b. For 5 need height of 9
        i. For 9 need height of 4.
        ii. Height of 4 is 1 and B.F = 0.
        iii. So height of 9 is 2 and B.F = height(4)-height(null) = 1–0 = 1.
        iv. Height of 5 = max(height(null),height(9))+1 = 3 and B.F = 0–2=-2.
    c. So height of 2 = max(height(7),height(5))+1=4 and B.F = 3–3 = 0.

So we have calculated Balance Factor of every node using heights of their children.

Note : Assume that height of leaf-node = 1 and B.F = 0.

Insertion in AVL tree

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST operations that can be performed to balance a BST without violating the BST property.

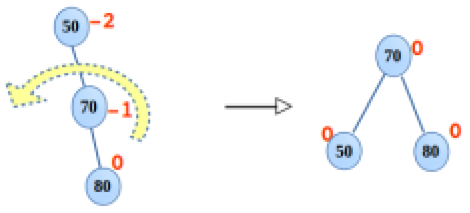
AVL tree insertion

Steps to follow for insertion:

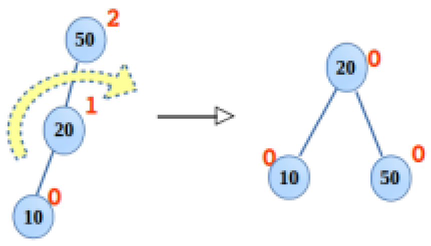
Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
    - y is the left child of z and x is the left child of y (Left Left Case)
    - y is the left child of z and x is the right child of y (Left Right Case)
    - y is the right child of z and x is the right child of y (Right Right Case)
    - y is the right child of z and x is the left child of y (Right Left Case)

- Shown below is RR rotation (anticlockwise rotation)

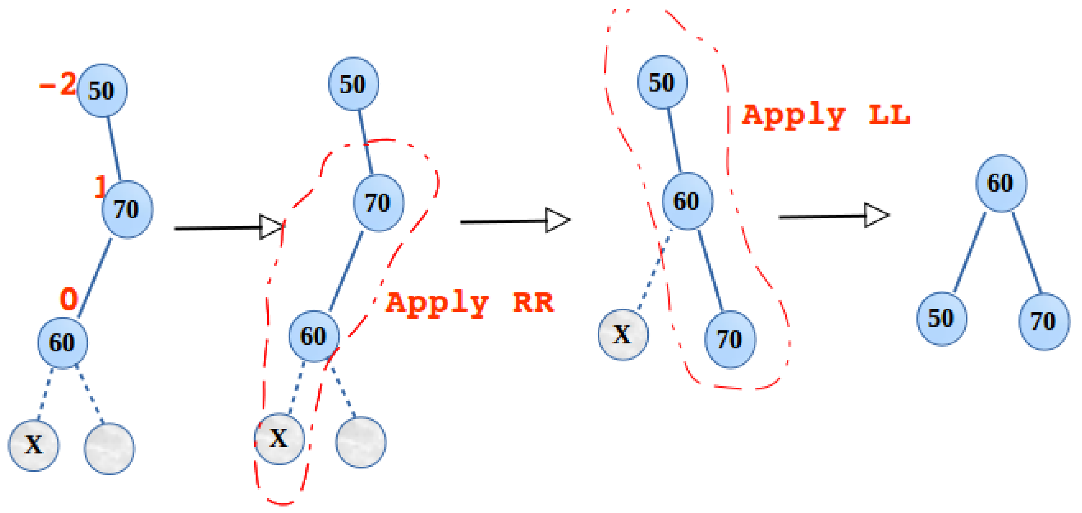


- Shown below is LL rotation (clockwise rotation)



Shown below is the case of LR rotation, here two rotations are performed. First RR and then, LL as follows,

- Right rotation is applied at 70, after restructuring, 60 takes the place of 70 and 70 as the right child of 60.
- Now left rotation is required at the root 50, 60 becomes the root. 50 and 70 become the left and right child respectively.



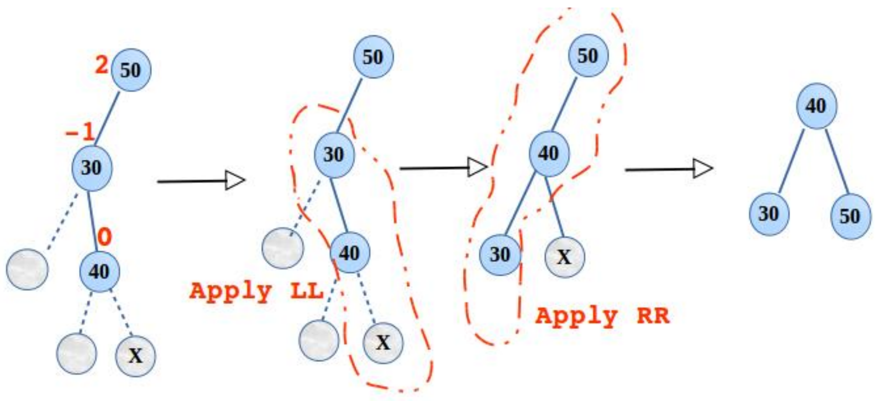
Trick:

Make median node as root node and arrange it as BST

4. Right left double rotation(RL)

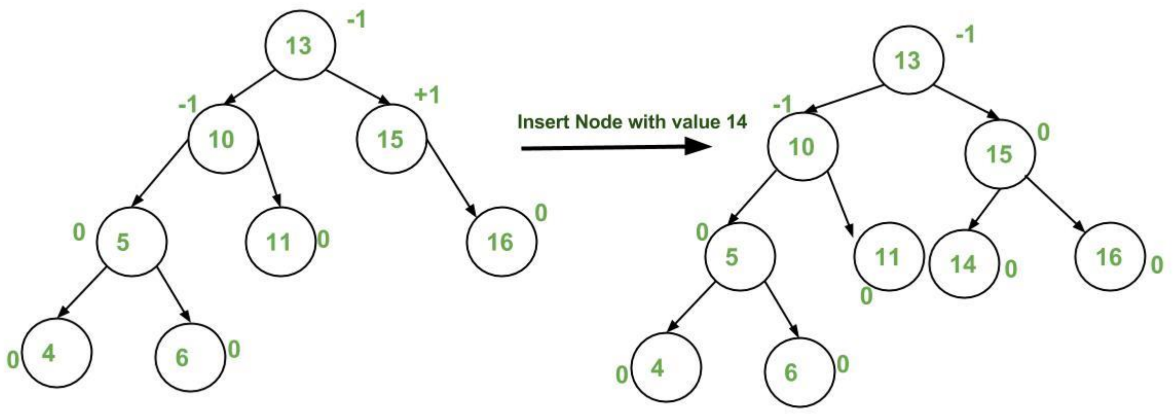
Shown below is the case of RL rotation, here two rotations are performed. First LL and then, RR as follows,

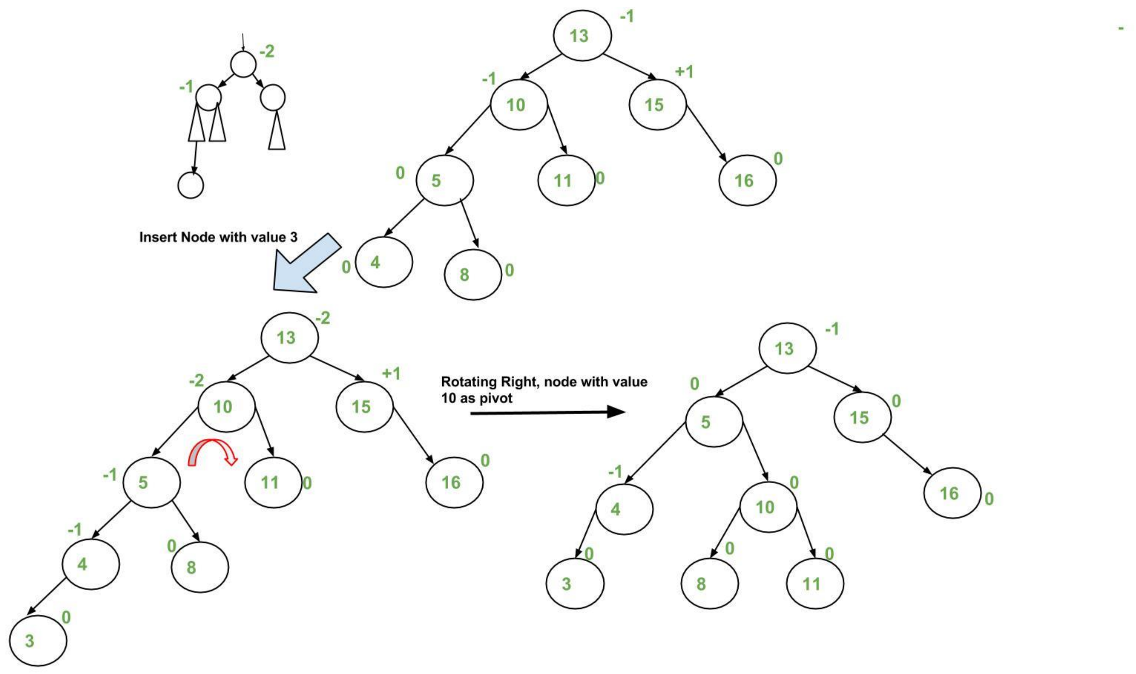
- Left rotation is applied at 30, after restructuring 40 takes the place of 30 and 30 as the left child of 40.
- Now right rotation is required at the root 50, 40 becomes root. 30 and 50 becomes the left and right child respectively.

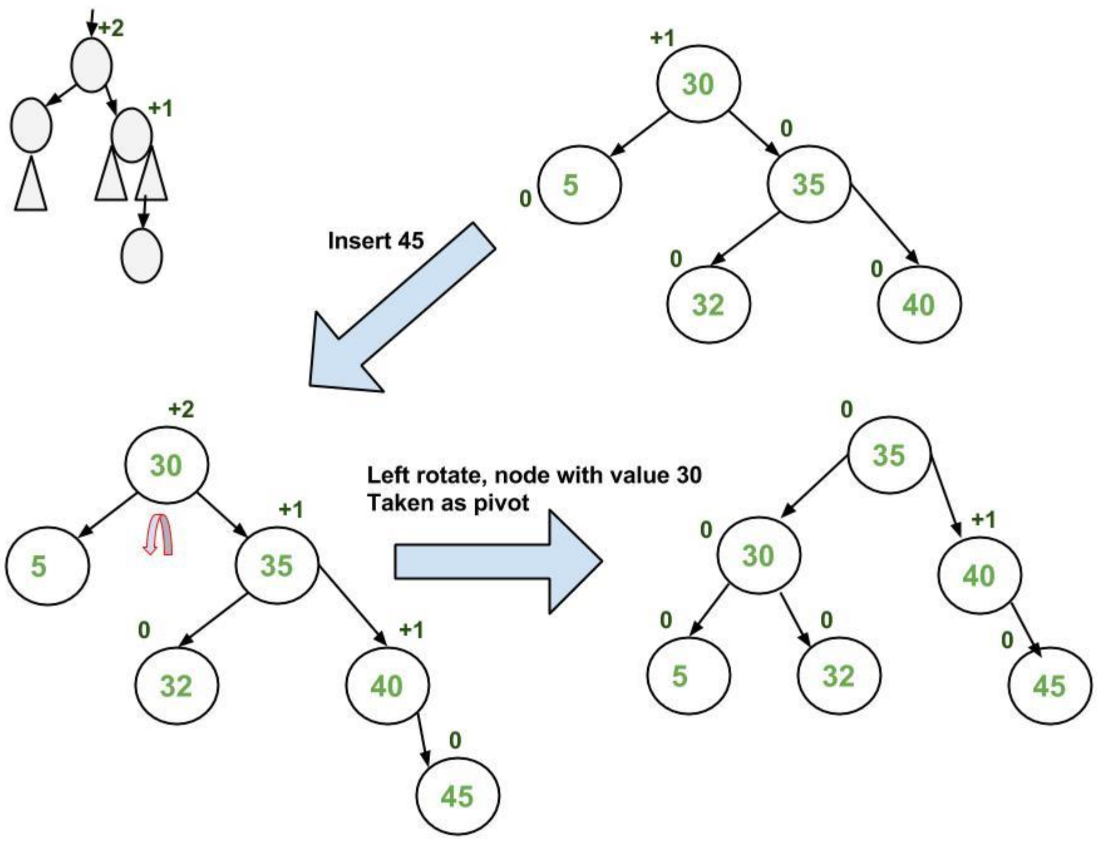


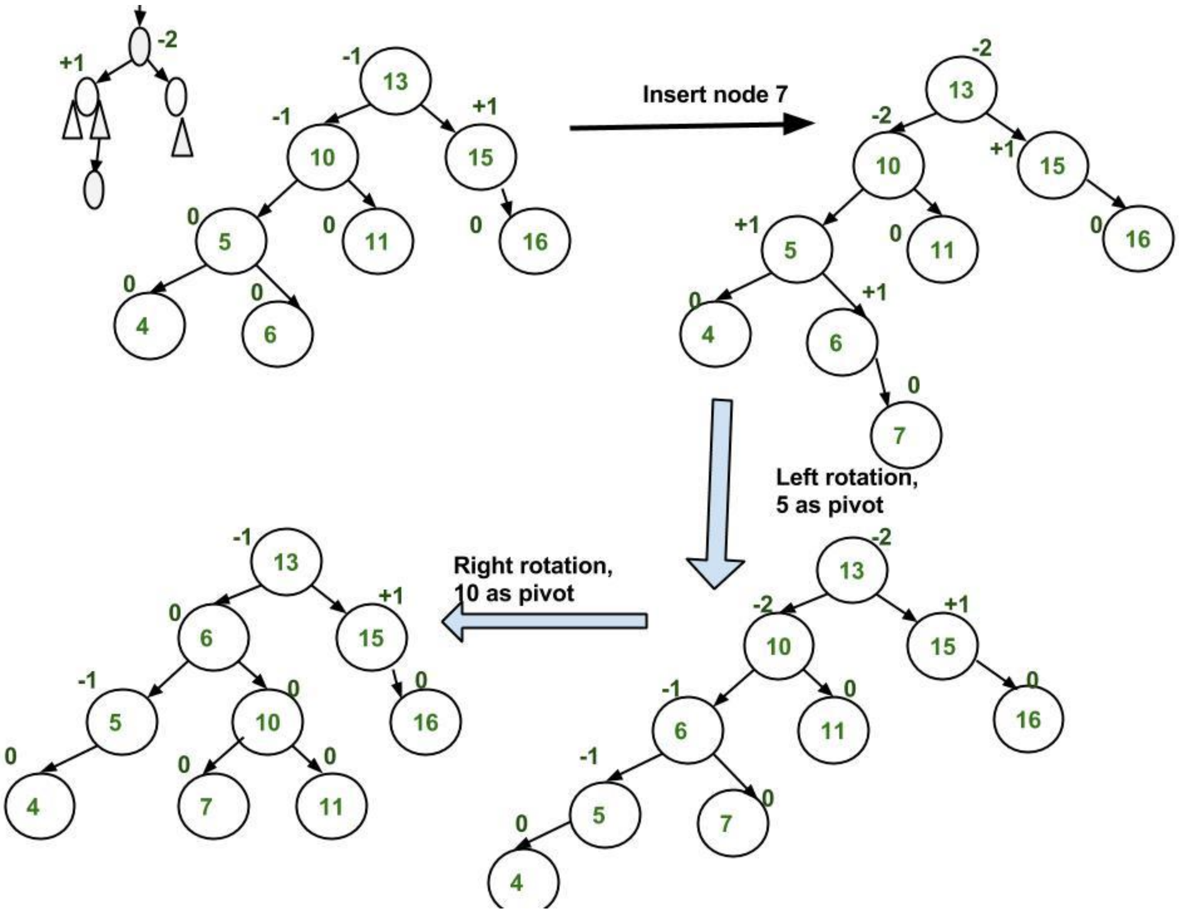
Approach:
The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.
Follow the steps mentioned below to implement the idea:

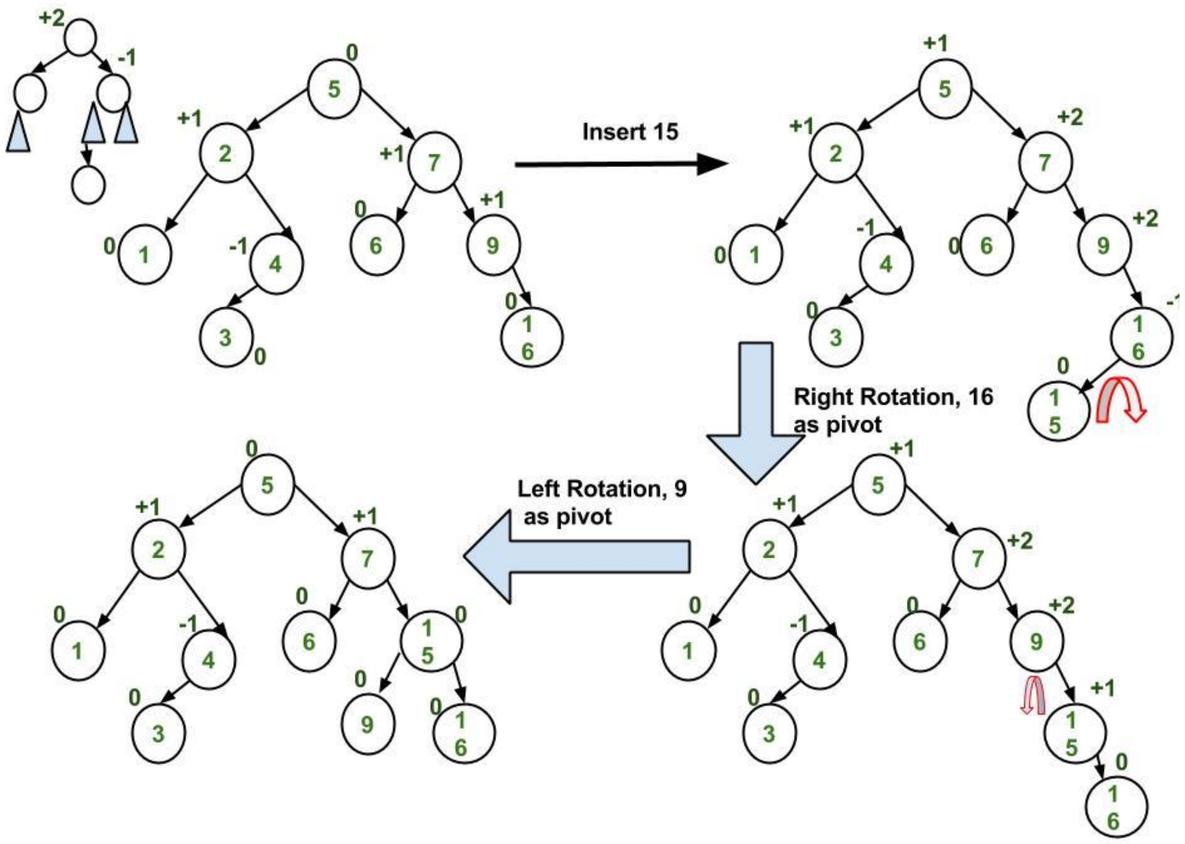
- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.
- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.


Insert Node with value 14


Insert Node with value 3
Rotating Right, node with value 10 as pivot

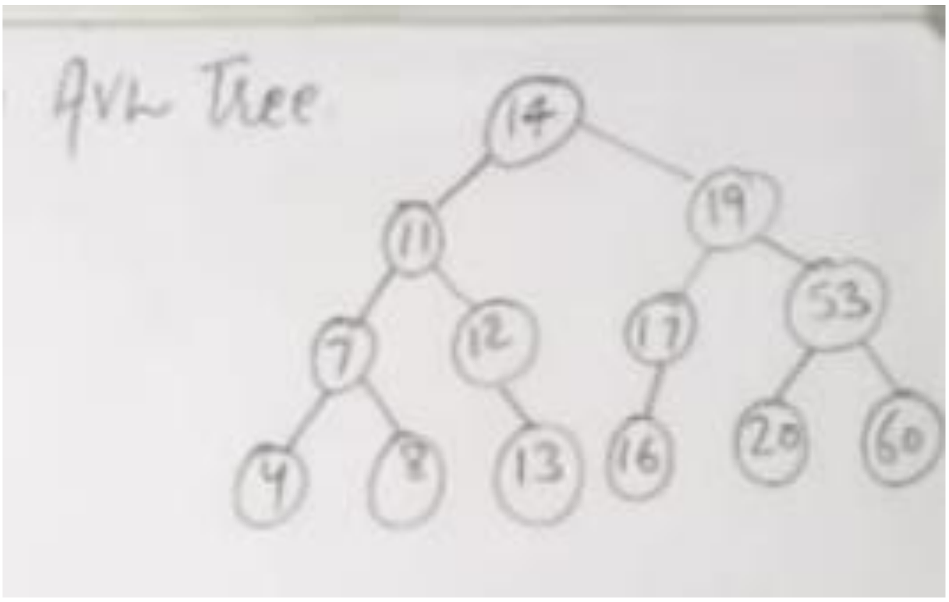

Insert 45
Left rotate, node with value 30
Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot



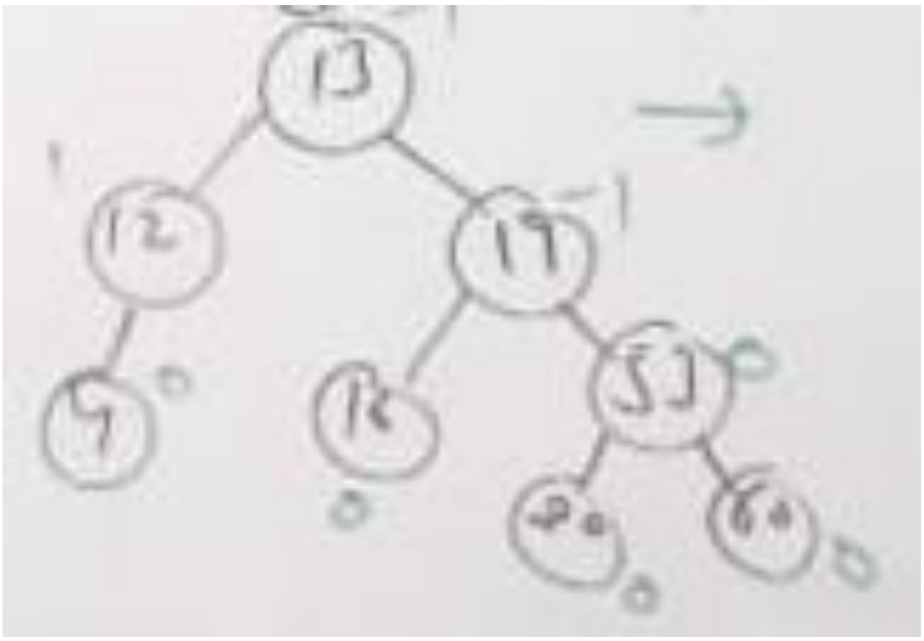
Delete operation:
Same as BST only difference is to balance the tree after every deletion.

Practice
Construct AVL tree by inserting the following data
14,17,11,7,53,4,13,12,8,60,19,16,20
Perform the operation and match with the following figure



Delete 8,7,11,14,7 from the above AVL tree (perform the operation and match with the following result)

Ans:



Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary

4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.