Data Structures and Algorithms

AVL and Heap Tree

PRASHANT HEMRAJANI

Assistant Professor

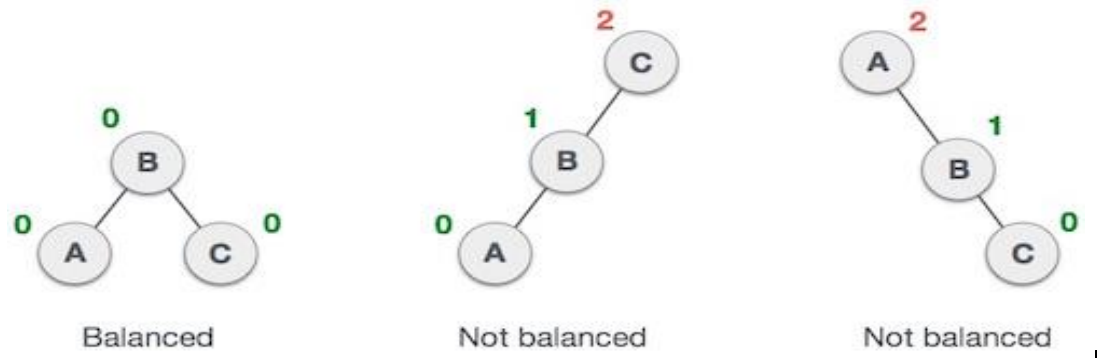
(Computer and Communication Engineering)

AVL Tree

- It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.
- Named after their inventor **Adelson**, **Velski** & **Landis**.
- **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Representation

- Here we see that the first tree is balanced and the next two trees are not balanced –



- In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

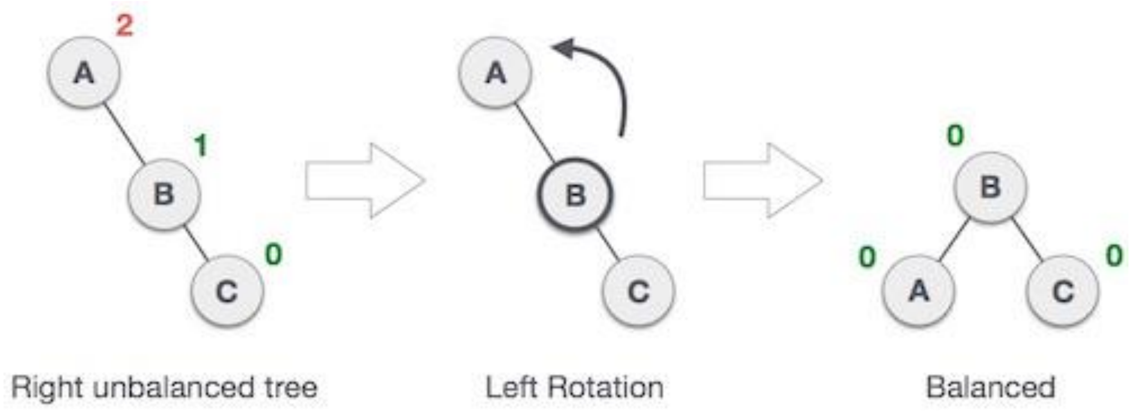
- If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

- To balance itself, an AVL tree may perform the following four kinds of rotations –
  - Left rotation
  - Right rotation
  - Left-Right rotation
  - Right-Left rotation
- The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

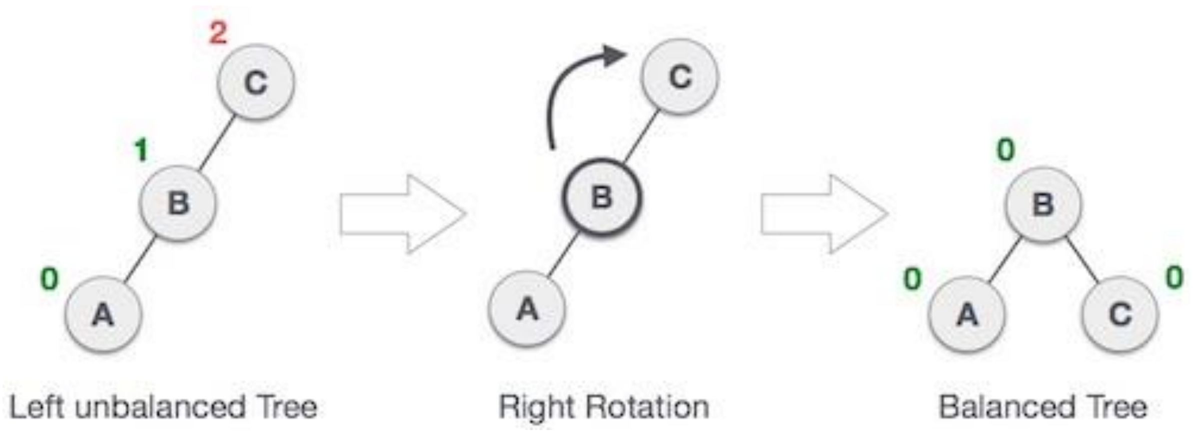
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

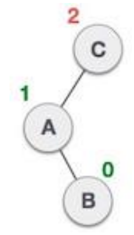
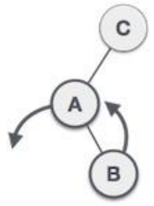
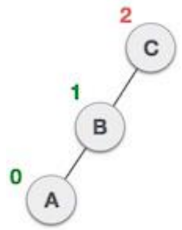
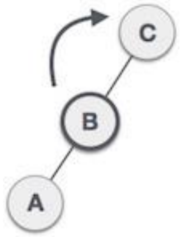
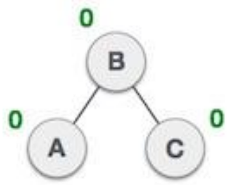
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

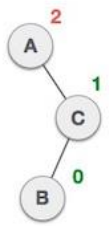
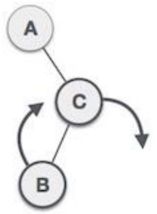
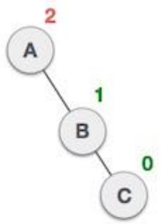
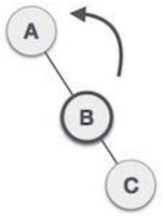
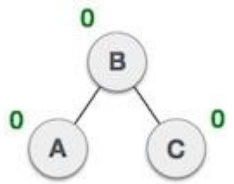
Left-Right Rotation

- Double rotations are slightly complex version of already explained versions of rotations.
- To understand them better, we should take note of each action performed while rotation.
- Let's first check how to perform Left-Right rotation.
- A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

- The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Practice Work

Implement AVL tree and Perform Rotations:

- 45, 36, 63, 27, 39, 54, 72, 70, 71
- 45, 36, 63, 27, 39, 54, 72, 18, 9
- 45, 36, 63, 27, 39, 54, 72, 89, 91
- 45, 36, 63, 54, 72, 89
- 45, 36, 63, 27, 39, 37
- 63, 9, 19, 27, 18, 108, 99, 81
- 16, 27, 9, 11, 36, 54, 81, 63, 72

Heap Tree

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly.
- If **α** has child node **β** then –

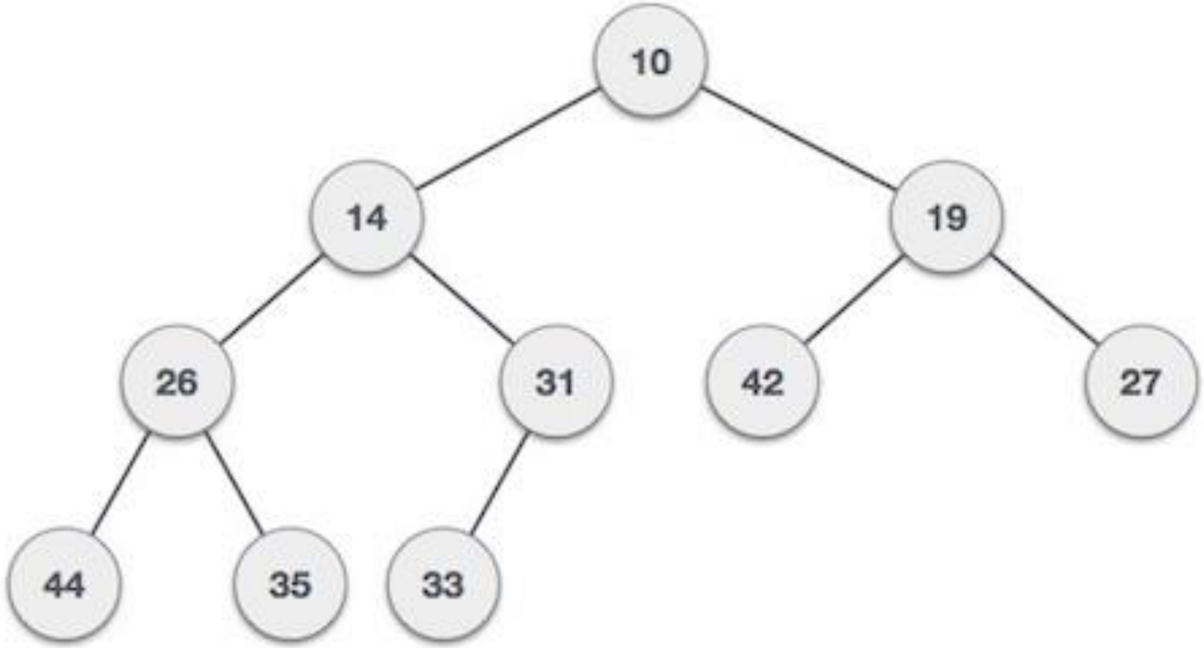
  $$\mathbf{key(\alpha) \geq key(\beta)}$$

- As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types.

Min Heap

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

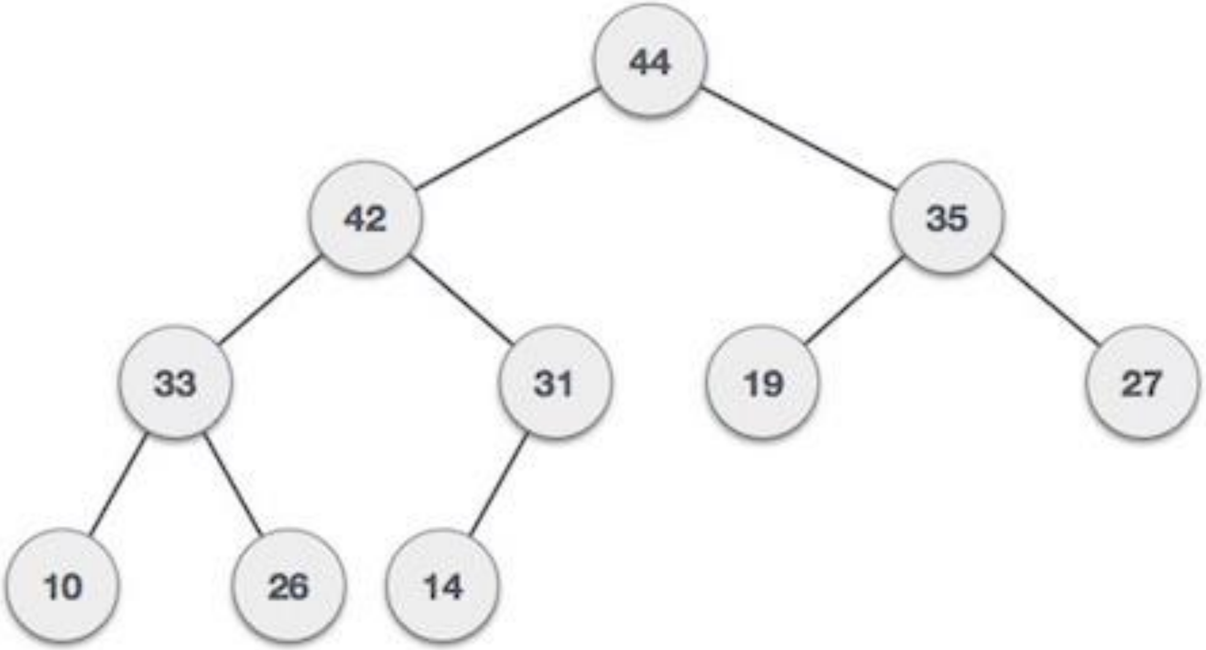
For Input → 35 33 42 10 14 19 27 44 26 31



Max Heap

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

For Input → 35 33 42 10 14 19 27 44 26 31



Max Heap Construction Algorithm

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

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Min Heap Construction Algorithm

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

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is greater than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Max Heap Deletion Algorithm

Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Min Heap Deletion Algorithm

Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is greater than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Insert Heap Algorithm 2

```
InsertHeap (HEAP, N, VAL)
Step 1: [Add the new value and set its POS]
            SET N:= N+1, POS := N
Step 2: SET HEAP [N] := VAL
Step 3: [Find appropriate location of VAL]
            Repeat Step 4 and 5 while POS<0
Step 4: SET PAR = POS/2
Step 5: IF HEAP[POS] < HEAP[PAR], then Exit
             Else
                        SWAP HEAP[POS], HEAP [PAR]
                        POS = PAR
            [End of IF]
            [End of Step 3 Loop]
Step 6: Exit
```

Delete Heap Algorithm 2

DeleteHeap (HEAP, N)

```
Step 1:  [Remove the last node from the heap]
         SET LAST := HEAP[N], SET N:= N-1
Step 2:  [Initialization]
         SET PTR := 0, LEFT =1, RIGHT = 2
Step 3:  SET HEAP [PTR] := LAST
Step 4:  Repeat Steps 5 to 7 while LEFT <=N
Step 5:  IF HEAP [PTR] > HEAP [LEFT] AND HEAP [PTR] > HEAP [RIGHT], Then Exit
         [End of IF]
Step 6:  IF HEAP [RIGHT] < HEAP [LEFT], then
                 SWAP HEAP [PTR], HEAP [LEFT]
                 SET PTR := LEFT
         ELSE
                 SWAP HEAP [PTR], HEAP [RIGHT]
                 SET PTR := RIGHT
         [End of IF]
Step 7:  SET LEFT := 2 * PTR and RIGHT = LEFT +1
         [End of Step 4 Loop]
Step 8:  Exit
```

Practice Work

Implement Max and Min Heap Tree:

- 45, 36, 63, 27, 39, 54, 72, 70, 71
- 63, 9, 19, 27, 18, 108, 99, 81
- 16, 27, 9, 11, 36, 54, 81, 63, 72
- 98, 2, 48, 12, 56, 32, 4, 67, 23, 87, 55, 46

Applications of Heaps

- Heaps are preferred for applications that include:
  1. **Heap Sort:** It is one of the best sorting methods that has no quadratic worst case scenarios.
  2. **Selection Algorithms:** These algorithms are used to find the minimum and maximum values in linear or sub-linear time.
  3. **Graph Algorithms:** Heaps can be used as internal traversal data structures. This guarantees that runtime is reduced by an order of polynomial. Heaps are therefore used for implementing Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

Any Queries
????