AVL Trees
AVL Trees

- Consider a situation when data elements are inserted in a BST in sorted order: 1, 2, 3, ...
- BST becomes a degenerate tree.
- Search operation FindKey takes O(n), which is as inefficient as in a list.
AVL Trees

- It is possible that after a number of insert and delete operations a binary tree may become imbalanced and increase in height.
- Can we insert and delete elements from BST so that its height is guaranteed to be $O(\log n)$? \(\rightarrow\) Yes, AVL Tree ensures this.
- Named after its two inventors: Adelson–Velski and Landis.
An Imbalanced Tree

A Balanced Tree
AVL Tree: Definition

- Height-balanced tree: A binary tree is a height-balanced-$p$-tree if for each node in the tree, the difference in height of its two subtrees is at the most $p$.
- AVL tree is a BST that is height-balanced-$1$-tree.
AVL Trees: Examples
AVL Trees

Inserting 1, 2, 3, 4 and 5

BST after insertions

AVL Tree after insertions
**ADT AVL Tree**

**Elements:** The elements are nodes, each node contains the following data type: Type

**Structure:** Same as for the BST; in addition the height difference of the two subtrees of any node is at the most one.

**Domain:** the number of nodes in a AVL is bounded; type AVLTree
ADT AVL Tree

Operations:
1. **Method** FindKey (int tkey, boolean found).
2. **Method** Insert (int k, Type e, boolean inserted).
3. **Method** Remove_Key (int tkey, boolean deleted)
4. **Method** Update(Type e)
ADT AVL Tree

5. Method Traverse (Order ord)
6. Method DeleteSub ( )
7. Method Retrieve (Type e)
8. Method Empty (boolean empty).
9. Method Full (boolean full)
public class Type AVLNode // AVL Tree Node {
private:
    int key
    Type data;
    Balance bal; //Balance is enum +1, 0, -1
    AVLNode<Type> *left, *right;
public AVLNode(int, Type);  // constructors
};
Step 1: A node is first inserted into the tree as in a BST.

There is always a unique path from the root to the new node called the search path.

Step 2: Nodes in the search path are examined to see if there is a pivot node. Three cases arise.

A pivot node is a node closest to the new node on the search path, whose balance is either −1 or +1.
AVL Tree: Insert

- Case 1: There is no pivot node. No adjustment required.
- Case 2: The pivot node exists and the subtree to which the new node is added has smaller height. No adjustment required.
- Case 3: The pivot node exists and the subtree to which the new node is added has the larger height. Adjustment required.
Insert: Case 1

No Pivot node

Insert 40

Insert 55
Insert: Case 2

New node added to the shorter subtrees of the Pivot.

Pivot Node

Insert 5

Insert 45
Insert: Case 3

A VL Tree is no more an AVL Tree after insertion.
Insert: Case 3

- When after an insertion or a deletion an AVL tree becomes imbalanced, adjustments must be made to the tree to change it back into an AVL tree.
- These adjustments are called rotations.
- Rotations are either single or double rotations.
- For Case 3 there are 4 sub-cases (2 + 2)
Insert: Case 3 (Sub-Case 1)

Remainder of the tree

A

B

Pivot

T3

h

T2

h

T1

New Node

Single Rotation

Remainder of the tree

A

B

T1

T2

T3

New Node
Insert: Case 3 (Sub-Case 2)

Pivot

Single Rotation

New Node
Insert: Case 3 (Sub-Case 3)

One of these is a new node

Double Rotation
Insert: Case 3 (Sub-Case 4)

One of these is a new node
Insertion Example

unbalanced...

...balanced
Step 1: Delete the node as in BSTs. Leaf or node with one child, will always be deleted.

Step 2: For each node on the path from the root to deleted node, check if the node has become imbalanced; if yes perform rotation operations otherwise update balance factors and exit. ➔ Three cases can arise for each node p, in the path.
Step 2 (contd.):  **Case 1**: Node $p$ has balance factor 0. No rotation needed.

**Case 2**: Node $p$ has balance factor of $+1$ or $-1$ and a node was deleted from the taller sub-trees. No rotation needed.

**Case 3**: Node $p$ has balance factor of $+1$ or $-1$ and a node was deleted from the shorter sub-trees. Rotation needed. Eight sub-cases. ($4 + 4$)
Delete: Case 1

Remainder of the tree

Node to be deleted.

Remainder of the tree

h-1

h

0

p

h-1

h

+1

p
Delete: Case 2

Remainder of the tree

p

-1

h

h

Node to be deleted.

Remainder of the tree

p

0

h

h
Delete: Case 3 (Sub–Case 1)

Remainder of the tree

\[ \begin{align*}
T_1 & \quad h-1 \\
T_2 & \quad h-1 \\
T_3 & \quad h
\end{align*} \]

Deleted Node

Single Rotation

Remainder of the tree

\[ \begin{align*}
C & \quad +1 \\
T_2 & \quad +1 \\
T_1 & \quad \text{Deleted Node}
\end{align*} \]
Delete: Case 3 (Sub-Case 2)

Single Rotation

Remainder of the tree

\[ p \]
\[ B \]
\[ C \]
\[ T_1 \]
\[ T_2 \]
\[ T_3 \]

-1
0
+1

-1
+1

Remainder of the tree

\[ C \]
\[ B \]
\[ T_3 \]
\[ T_1 \]
\[ T_2 \]
Delete: Case 3 (Sub-Case 3)

Double Rotation

Remainder of the tree

p

B

+1

-1

h-1

T1

A

h-1

-1

C

T2

T3

h-2

Deleted Node

h-1

Remainder of the tree

C

0

B

0

A

+1

h-1

h-1

T1

T2

h-2

T3

T4

h-1
Delete: Case 3 (Sub–Case 4)

Double Rotation

Remainder of the tree

Deleted Node

T1
h-1

C
h-1

A
h-1

B

p

+1

T4

Remainder of the tree

C

-1

B

-1

0

A

h-1

T1

h-1

h-2

T2

h-1

T3

h-1

T4
Delete: Case 3 (Other Sub-Cases)

- Sub-Case 5: mirror image of Sub-Case 1.
- Sub-Case 6: mirror image of Sub-Case 2.
- Sub-Case 7: mirror image of Sub-Case 3.
- Sub-Case 8: mirror image of Sub-Case 4.
Deletion: Example

Delete p
Deletion: Example

Sub-Case 1
Single Rotation

Delete p
Deletion: Example

Sub Case 8
Double Rotation
Deletion: Example

After Double Rotation