

Lecture 12: AVL Tree

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as **$N-1$**
- Time needed to perform insertion and deletion and many other operations can be $O(N)$ in the worst case
- Goal is to keep the height of a binary search tree balanced
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree

AVL Tree

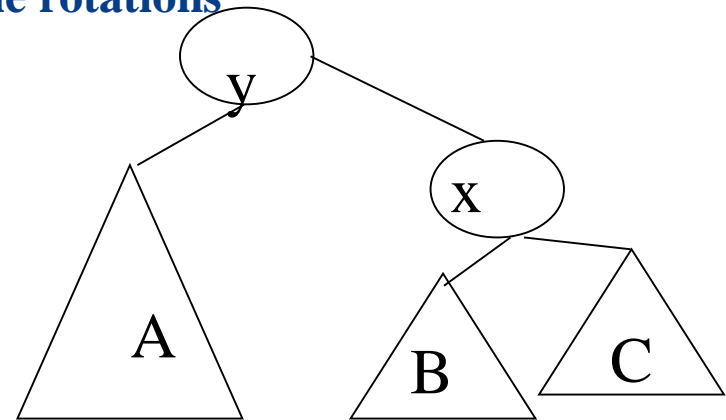
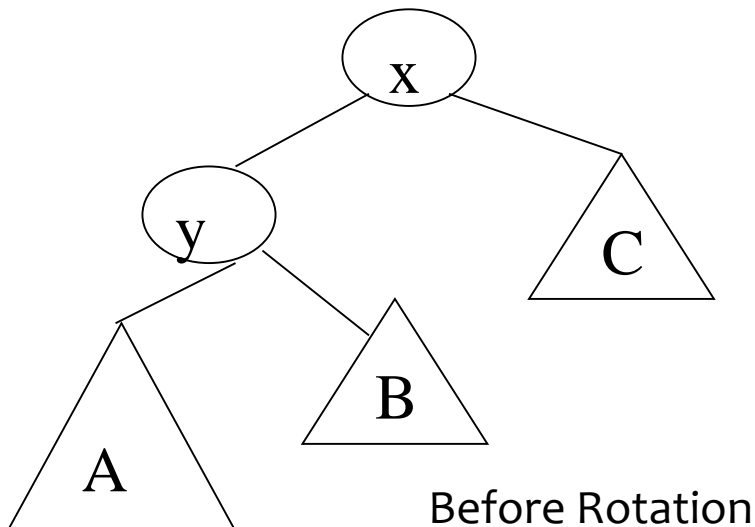
- Named after **Adelson-Velskii** and **Landis**
- The first dynamically balanced trees
- Its Binary search tree with **balance condition** in which the sub-trees of each node can differ by **at most 1** in their height i.e in the range **-1 to 1**
- **balancefactor = height(right-subtree) - height(left-subtree)**
- If balanceFactor is **negative**, the node is "**heavy on the left**" since the height of the left subtree is greater than the height of the right subtree
- With balanceFactor **positive**, the node is "**heavy on the right**"
- A balanced node has **balancefactor = 0**

Balance Factor

- The value of the field is the **difference** between the height of the **right** and **left subtrees** of the node
- **balanceFactor = height(right-subtree) - height(left-subtree)**
- If balanceFactor is **negative**, the node is "**heavy on the left**" since the height of the left subtree is greater than the height of the right subtree
- With balanceFactor **positive**, the node is "**heavy on the right**"
- A balanced node has **balanceFactor = 0**

Rotations

- When the tree structure changes (e.g., **insertion or deletion**), we need to transform the tree to **restore the AVL tree property**
- This is done by using **single rotations** or **double rotations**



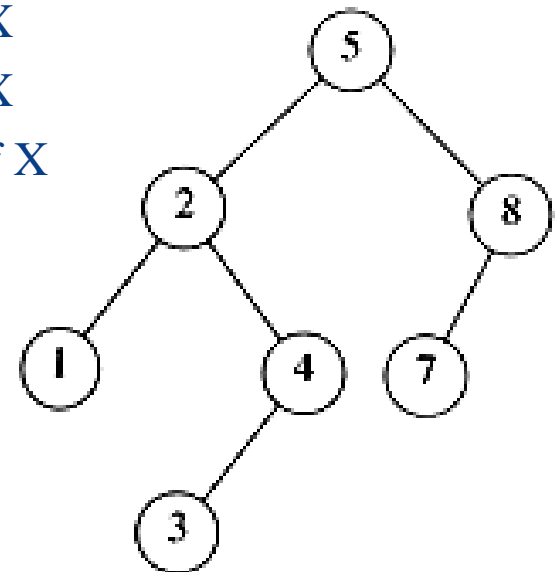
Rotations

- Since an insertion/deletion involves **adding/deleting** a single node, this can only increase/decrease the height of some subtree **by 1**
- If the AVL tree property is violated at a node x , it means that the heights of $\text{left}(x)$ and $\text{right}(x)$ **differ by exactly 2**
- Rotation will be applied to x to restore the AVL tree property

Rebalancing

➤ Suppose the node to be rebalanced is **X**. There are **4 cases** that we might have to fix (two are the mirror images of the other two):

1. An insertion in the **left subtree** of the **left child** of X
2. An insertion in the **right subtree** of the **left child** of X
3. An insertion in the **left subtree** of the **right child** of X
4. An insertion in the **right subtree** of the **right child** of X



Balancing Operations: Rotations

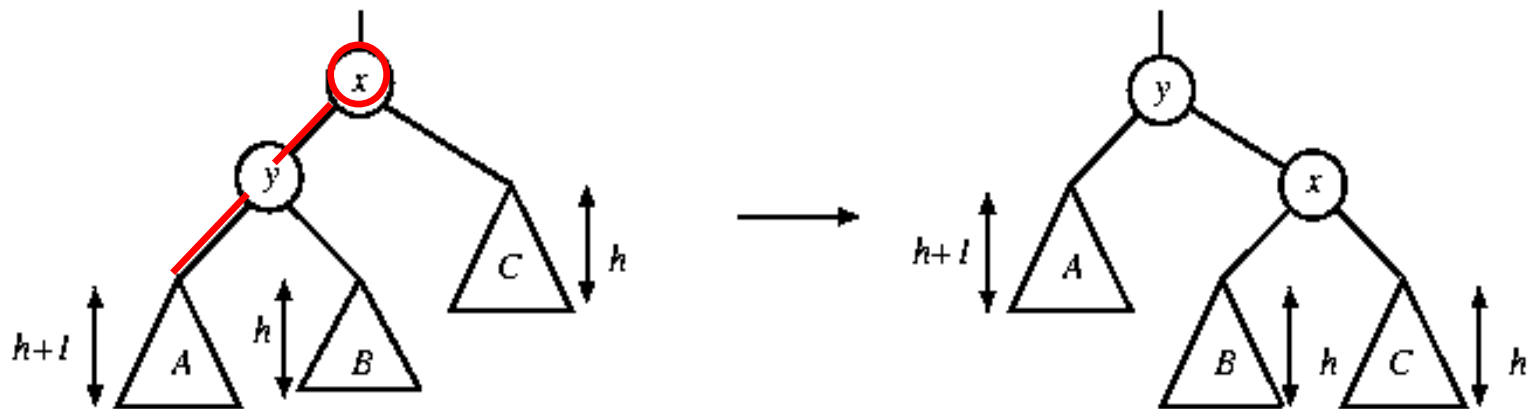
- **Case 1** and **case 4** are symmetric and requires the same operation for balance
 - Cases 1,4 are handled by *single rotation*
- **Case 2** and **case 3** are symmetric and requires the same operation for balance
 - Cases 2,3 are handled by *double rotation*

Single Rotation

- A single rotation switches the roles of the parent and child while maintaining the search order
- Rotate between a node and its child
 - Child becomes parent. Parent becomes right child in case 1, left child in case 4

Single rotation

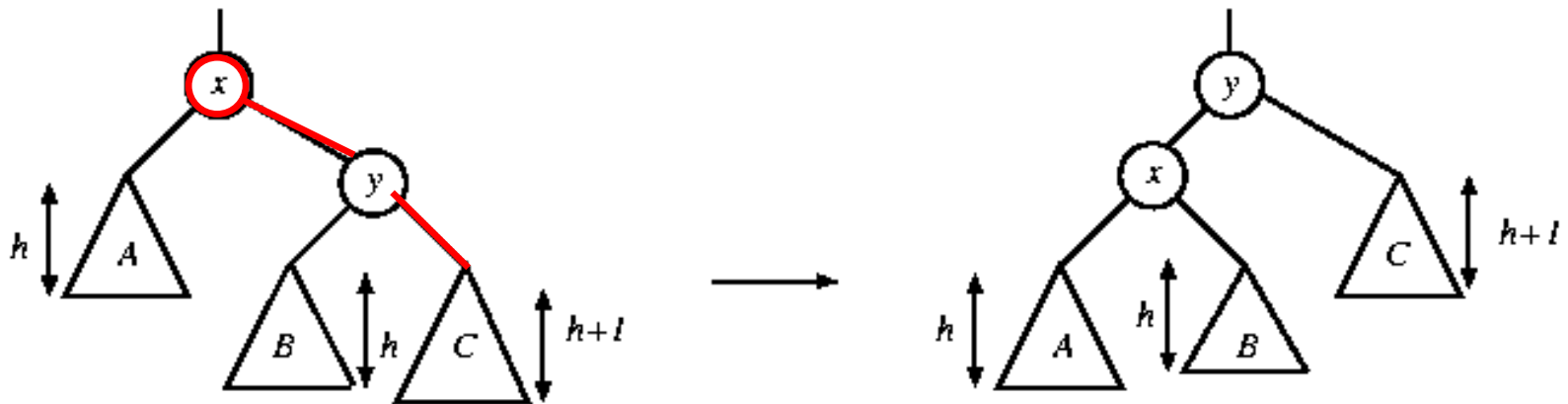
- The new key is inserted in the **subtree A**.
- The AVL-property is **violated at x**
- Height of left(x) is **$h+2$**
- Height of right(x) is **h**



Rotate with left child

Single rotation

- The new key is inserted in the **subtree C**
- The AVL-property is **violated at x**

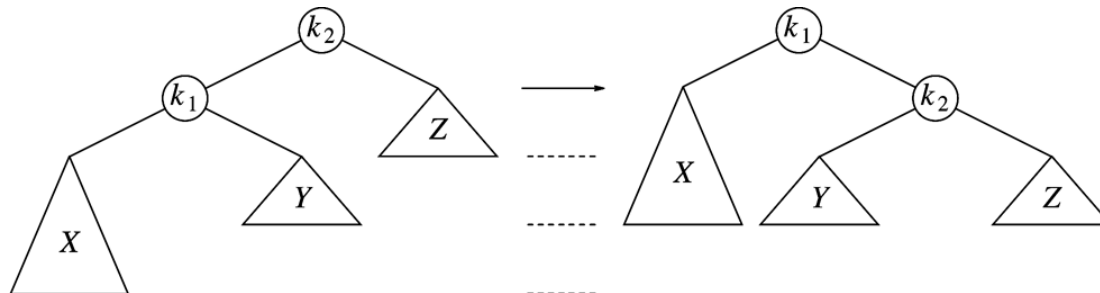


Rotate with right child

Single rotation takes $O(1)$ time.
Insertion takes $O(\log N)$ time.

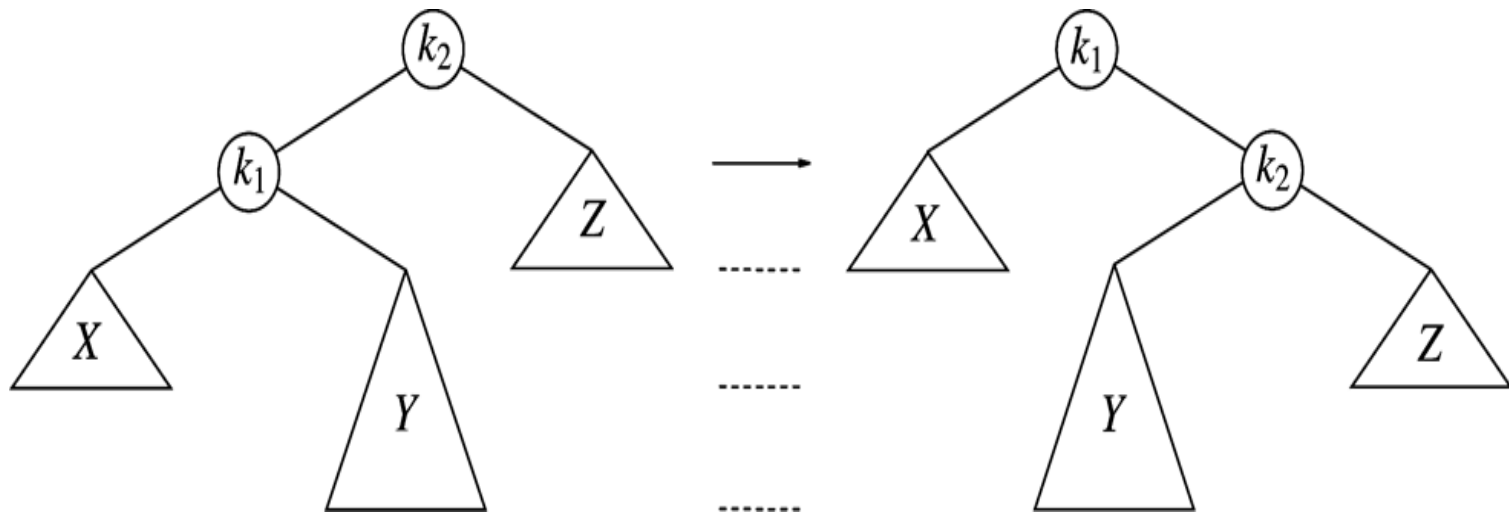
Single Rotation

```
1      /**
2      * Rotate binary tree node with left child.
3      * For AVL trees, this is a single rotation for case 1.
4      * Update heights, then set new root.
5      */
6      void rotateWithLeftChild( AvlNode * & k2 )
7      {
8          AvlNode *k1 = k2->left;
9          k2->left = k1->right;
10         k1->right = k2;
11         k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12         k1->height = max( height( k1->left ), k2->height ) + 1;
13         k2 = k1;
14     }
```



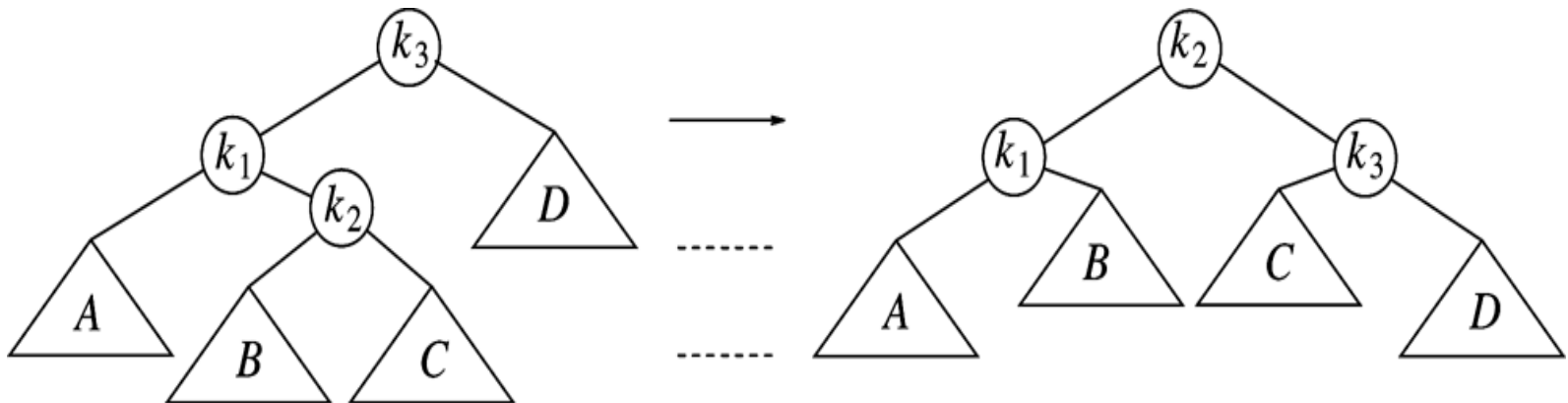
Single Rotation Will Not Work for the Other Case

- For case 2
- After single rotation, k_1 still **not balanced**
- **Double rotations** needed for **case 2** and **case 3**



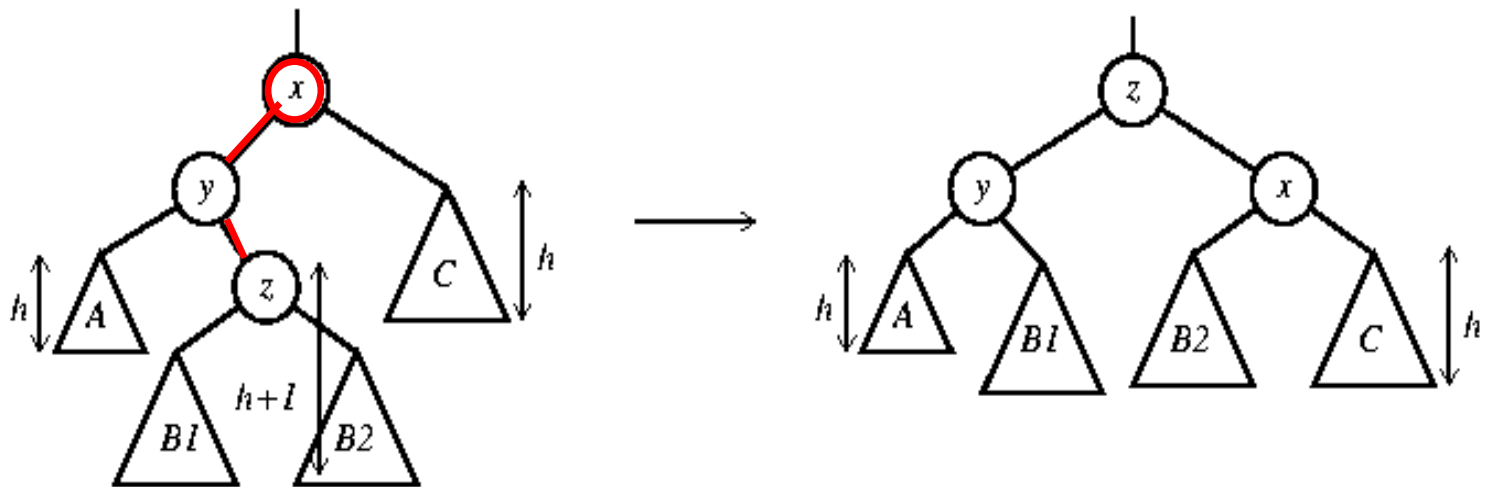
Double Rotation

- Left-right double rotation to fix case 2
- First rotate between k_1 and k_2
- Then rotate between k_2 and k_3
- Case 3 is similar



Double rotation

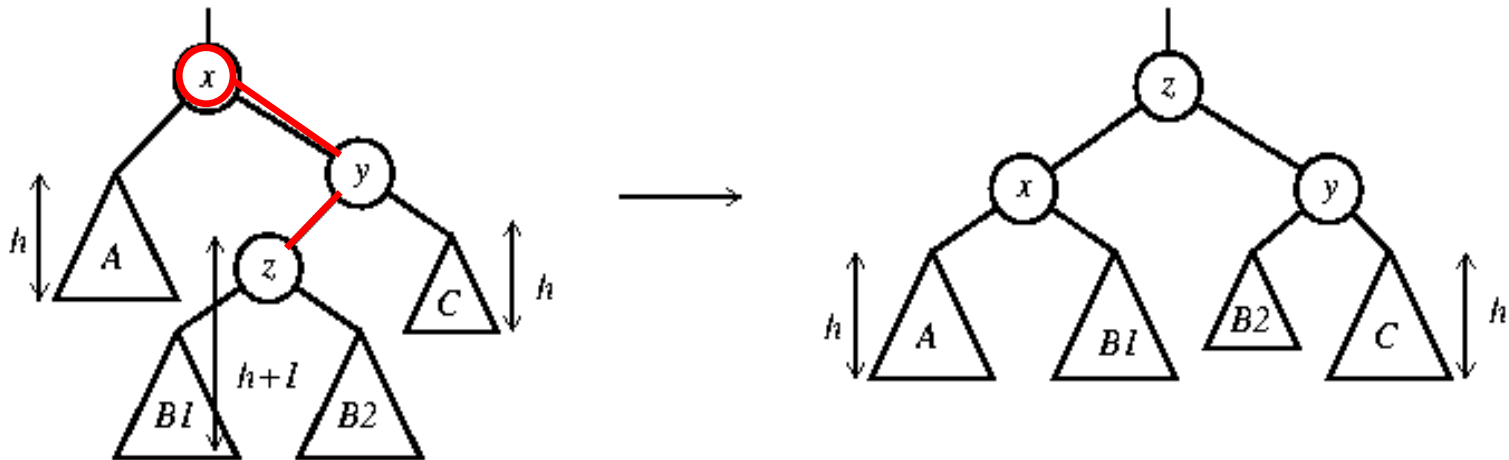
- The new key is inserted in the subtree B1 or B2
- The AVL-property is violated at x
- x - y - z forms a zig-zag shape



Double rotate with left child

Double rotation

- The new key is inserted in the subtree B1 or B2
- The AVL-property is violated at x



Double rotate with right child

Node declaration for AVL trees

```
template <class Comparable>
class AVLTree;
```

```
template <class Comparable>
class AVLNode
{
    Comparable element;
    AVLNode *left;
    AVLNode *right;
    int height;
```

```
    AVLNode( const Comparable & theElement, AVLNode *lt,
              AVLNode *rt, int h = 0 )
        : element( theElement ), left( lt ), right( rt ),
          height( h ) { }
    friend class AVLTree<Comparable>;
};
```

Height

```
template class <Comparable>
int AvlTree<Comparable>::height( AvlNode<Comparable> *t) const
{
    return t == NULL ? -1 : t->height;
}
```

Double Rotation

```
/**
```

- * Double rotate binary tree node: first left child.
- * with its right child; then node k3 with new left child.
- * For AVL trees, this is a double rotation for case 2.
- * Update heights, then set new root.

```
*/
```

```
template <class Comparable>
```

```
void AvlTree<Comparable>::doubleWithLeftChild( AvlNode<Comparable> * & k3 ) const  
{  
    rotateWithRightChild( k3->left );  
    rotateWithLeftChild( k3 );  
}
```

```
/* Internal method to insert into a subtree.  
 * x is the item to insert.  
 * t is the node that roots the tree.  
 */
```

```
template <class Comparable>  
void AvlTree<Comparable>::insert( const Comparable & x, AvlNode<Comparable> * & t ) const  
{  
    if( t == NULL )  
        t = new AvlNode<Comparable>( x, NULL, NULL );  
    else if( x < t->element )  
    {  
        insert( x, t->left );  
        if( height( t->left ) - height( t->right ) == 2 )  
            if( x < t->left->element )  
                rotateWithLeftChild( t );  
            else  
                doubleWithLeftChild( t );  
    }  
    else if( t->element < x )  
    {  
        insert( x, t->right );  
        if( height( t->right ) - height( t->left ) == 2 )  
            if( t->right->element < x )  
                rotateWithRightChild( t );  
            else  
                doubleWithRightChild( t );  
    }  
    else  
        ; // Duplicate; do nothing  
    t->height = max( height( t->left ), height( t->right ) ) + 1;  
}
```

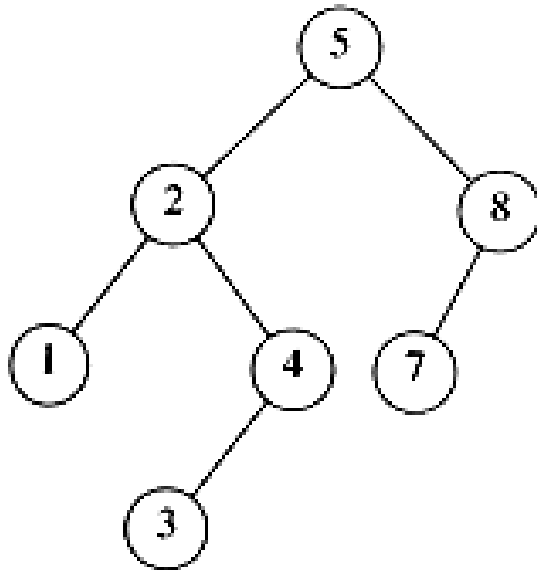
Insertion

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of $\text{left}(x)$ and $\text{right}(x)$ differ by at most 1
- If yes, proceed to $\text{parent}(x)$. If not, restructure by doing **either a single rotation or a double rotation**
- For insertion, once we perform a rotation at a node x , we won't need to perform any rotation at any ancestor of x

Insertion

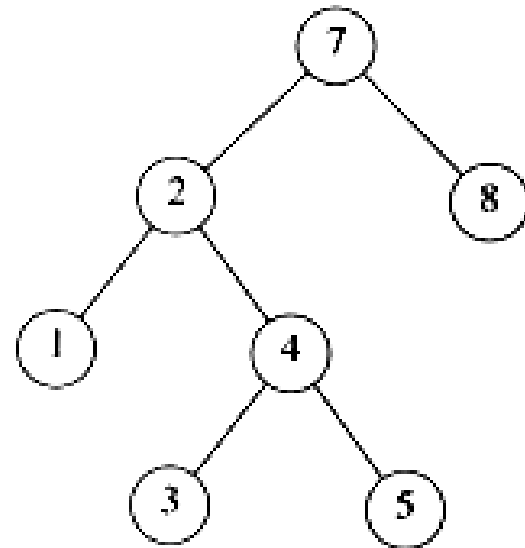
- Let x be the node at which $\text{left}(x)$ and $\text{right}(x)$ differ by more than 1
- Assume that the height of x is $h+3$
- There are 4 cases
 - Height of $\text{left}(x)$ is $h+2$ (**i.e. height of $\text{right}(x)$ is h**)
 - * Height of $\text{left}(\text{left}(x))$ is $h+1 \Rightarrow$ single rotate with left child
 - * Height of $\text{right}(\text{left}(x))$ is $h+1 \Rightarrow$ double rotate with left child
 - Height of $\text{right}(x)$ is $h+2$ (**i.e. height of $\text{left}(x)$ is h**)
 - * Height of $\text{right}(\text{right}(x))$ is $h+1 \Rightarrow$ single rotate with right child
 - * Height of $\text{left}(\text{right}(x))$ is $h+1 \Rightarrow$ double rotate with right child

AVL tree?



YES

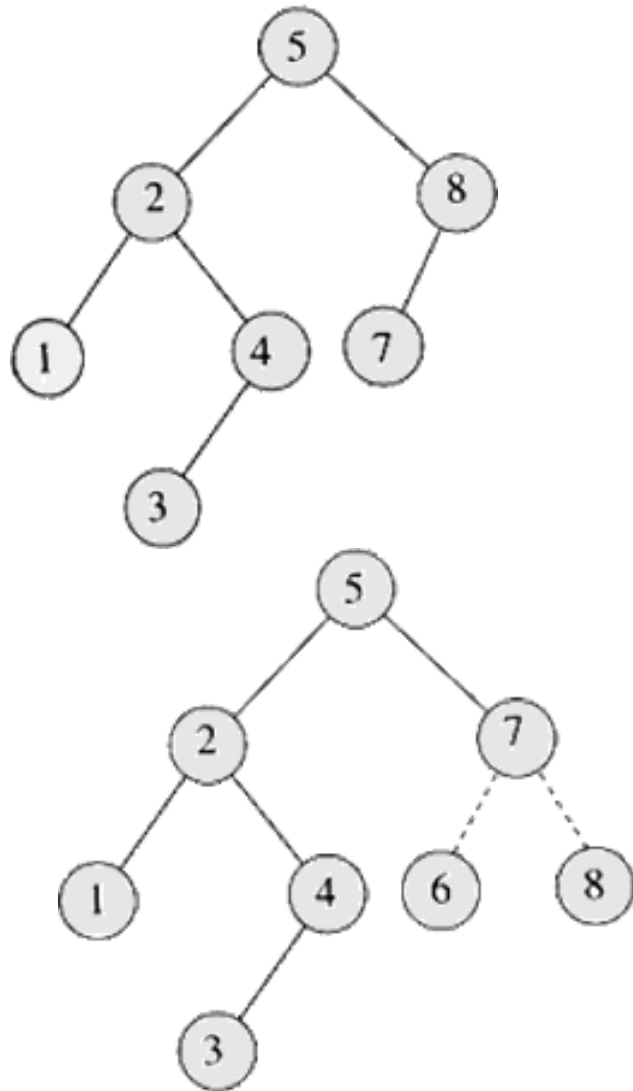
Each left sub-tree has height 1 greater than each right sub-tree



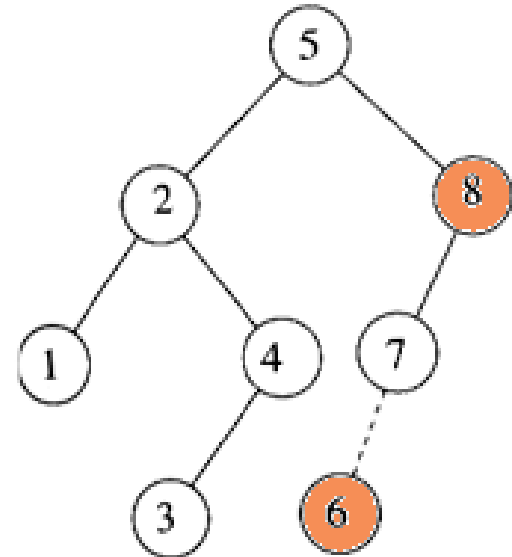
NO

Left sub-tree has height 3, but right sub-tree has height 1

Insertion

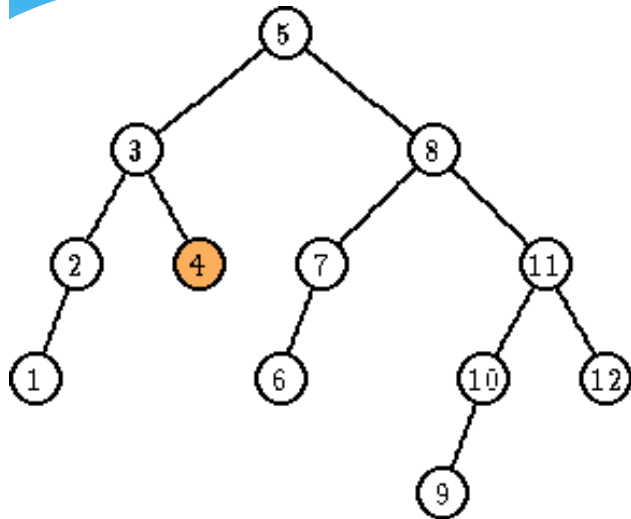


→
Insert 6

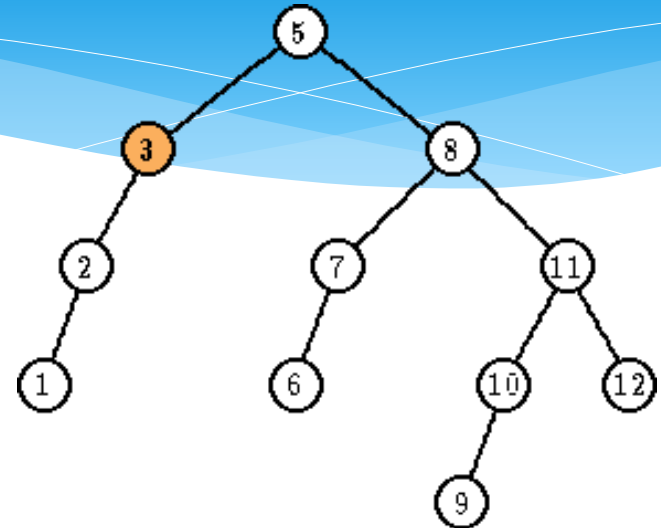


↙
Imbalance at 8
Perform rotation with 7

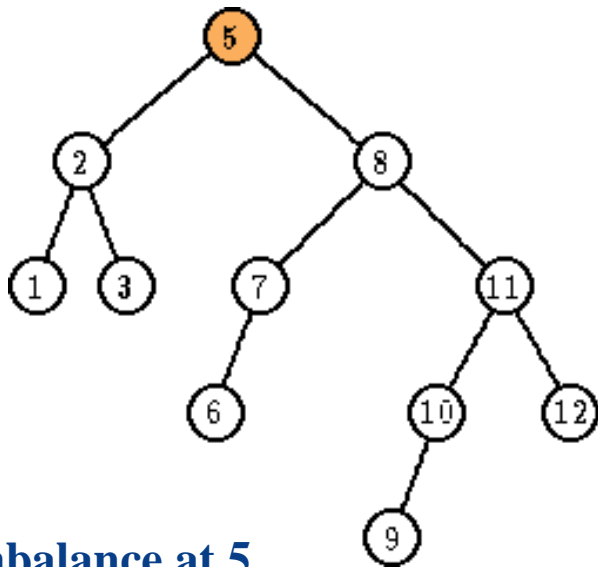
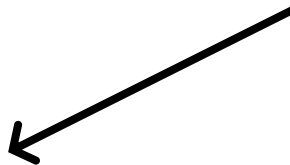
Deletion



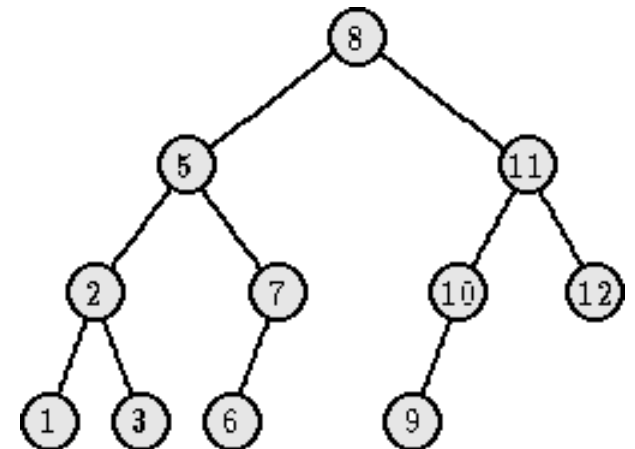
Delete 4



Imbalance at 3
Perform rotation with 2



Imbalance at 5
Perform rotation with 8

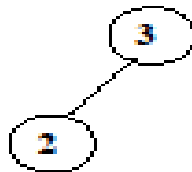


Insert 3,2,1,4,5,6,7, 16,15,14

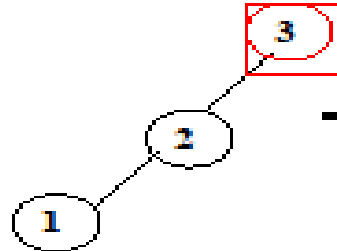
Insert 3



Insert 2

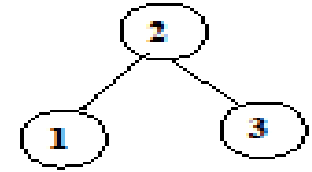


Insert 1 (non-AVL)

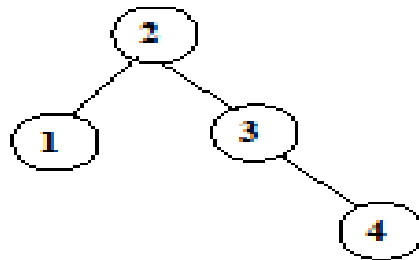


Single rotation

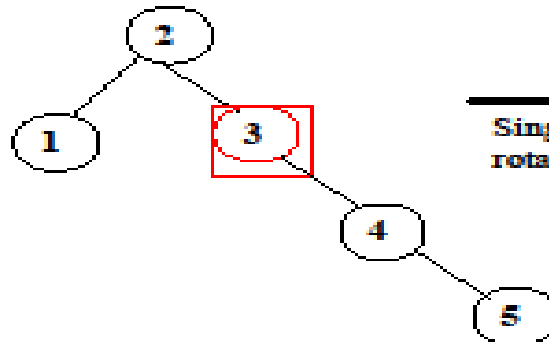
AVL



Insert 4

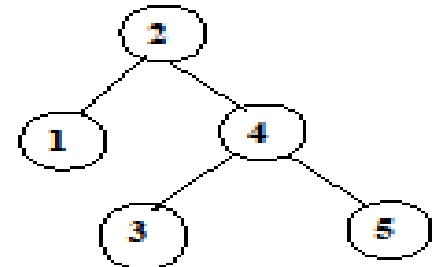


Insert 5 (non-AVL)

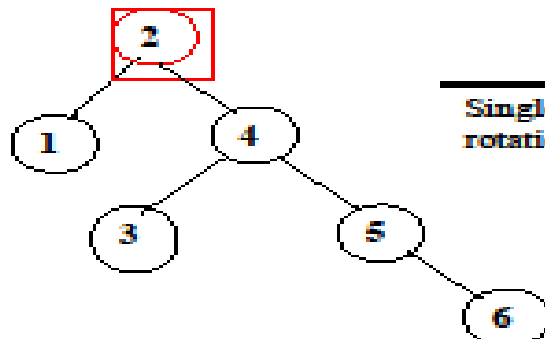


Single rotation

AVL

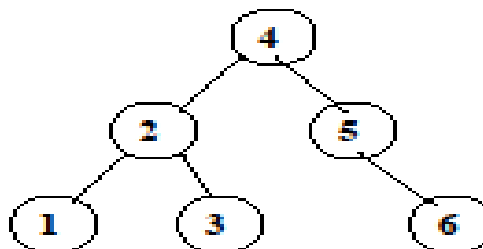


Insert 6 (non-AVL)

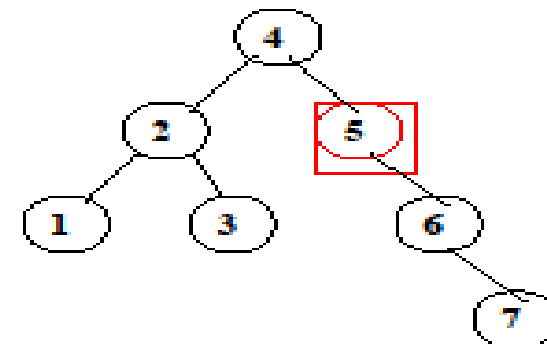


Single rotation

AVL



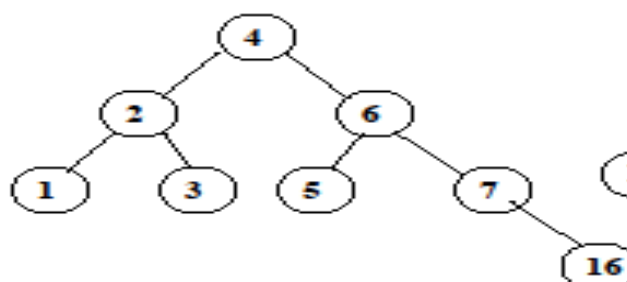
Insert 7 (non-AVL)



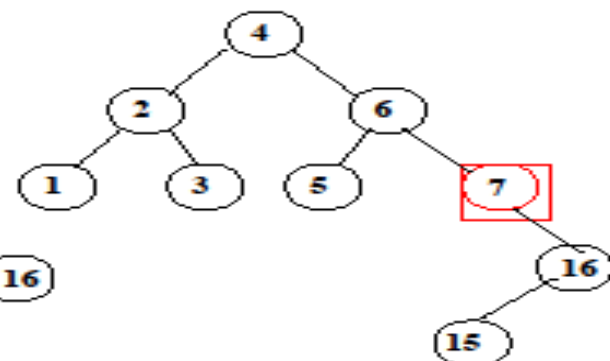
AVL



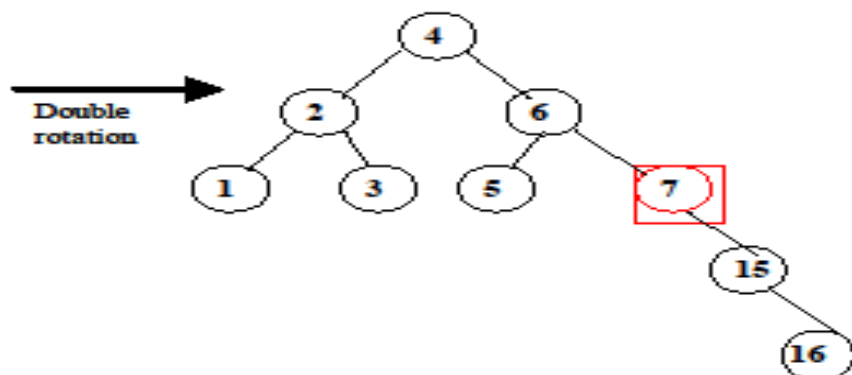
Insert 16



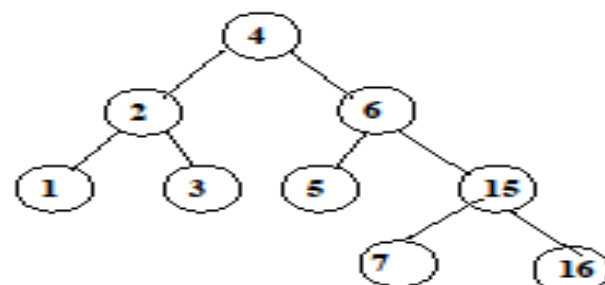
Insert 15 (non-AVL)



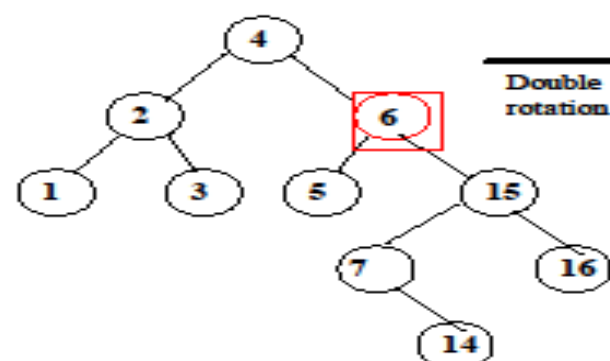
Step 1: Rotate child and grandchild



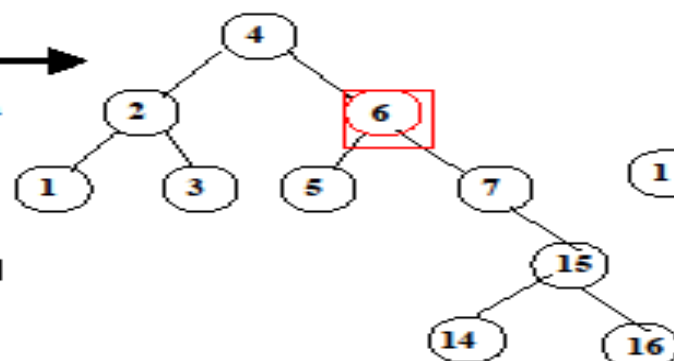
Step 2: Rotate node and new child (AVL)



Insert 14 (non-AVL)



Step 1: Rotate child and grandchild



Step 2: Rotate node and new child (AVL)

