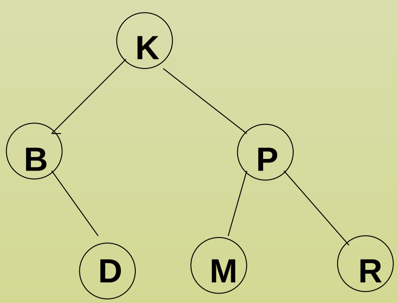
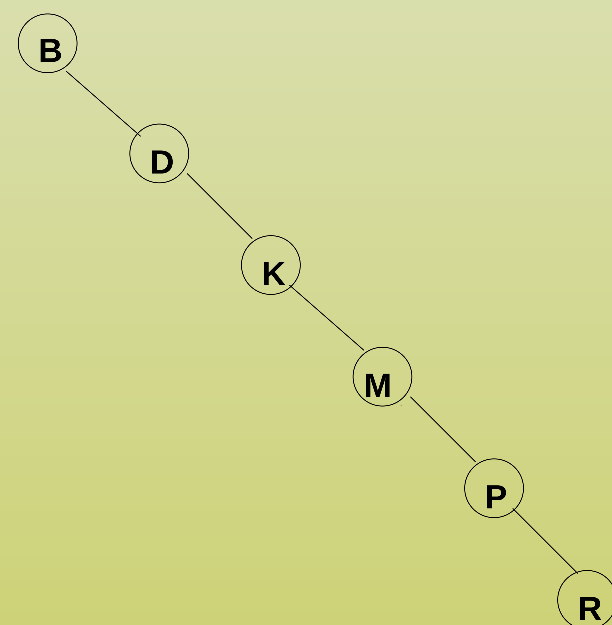
Height-Balanced Trees

Search process is *much* faster using trees ($O(\log_2 n)$) than using linked lists ($O(n)$). But if a tree is unbalanced then this advantage is lost, and search process is $O(n)$.

Well balanced tree



Unbalanced tree



Height-Balanced Trees (cont)

Therefore, it is worth the effort to build a balanced tree or modify an existing tree and make it balanced. A tree is said to be *height-balanced* if all of its nodes have balance factors: 1, 0, or -1.

Balance factor of a tree is defined as the difference between the heights of its left and right subtree.

The height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree. The height of an empty tree is -1.

AVL Tree

An AVL tree is a tree in which the height of left and right subtrees of every node differ by at most one. If a balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced. Basic steps for balancing a tree:

1. Let a node to be inserted travel down the appropriate branch, insert the node in the appropriate point. Besides keep track along the way of the deepest-level node (not the inserted node) on that branch that has a balance factor of 1 or -1 (this node is called the *pivot node*).
2. Starting from the pivot node, recompute all the balance factors along the insertion path (traced in step 1).
3. Determine whether the absolute value of the pivot node's balance factor switched from 1 to 2.
4. If there was such a switch, perform a manipulation on tree pointers centered at the pivot node to bring the tree back into height-balance (using AVL rotations).

Theorem:

An AVL tree of height h has at least $F_{h+3} - 1$ nodes, where F_i is the ith Fibonacci number. The Fibonacci numbers $F_0, F_1, \ldots, F_i$ are defined as follows:

$$F_0 = 0,\ F_1 = 1 \text{ and } F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2.$$

It can be shown that $F_i \approx \phi^i/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2 \approx 1.618$. The height of an AVL tree satisfies $h < 1.44 \log_2(n+1) - 1.328$, where n is the number of nodes in the tree. The worst-case height is at most 44% more than the minimum for binary trees.

Let n_h denote the minimum number of elements in the AVL tree.
Examples:

$n_0 = 1$ – an AVL tree with one element for $h = 0$

$n_1 = 2$ – an AVL tree with two elements for $h = 1$

$n_2 = 2$ – an AVL tree with four elements for $h = 2$, the root r plus $n_0 = 1$ plus $n_1 = 2$

$n_h = n_{h-1} + n_{h-2} + 1$ – an AVL tree with minimum number of elements for h is called *Fibonacci tree*.

$n \geq n_h = F_{h+3} - 1$ – this relation can be proved by induction with respect to the height h of this tree. Also, we know that $F_h = \frac{\Phi^h}{\sqrt{5}}$. Hence $n \geq \frac{\Phi^{h+3}}{\sqrt{5}} - 1$ and solving for h we get:

$h + 3 \leq \frac{\log_2(\sqrt{5}(n+1))}{\log_2 \Phi} \quad h \leq 1.44 \log_2(n+1) - 1.328 \leq 1.44 \log_2(n+1) =$ $\mathrm{O}(\log_2 n)$.

Prove that $n_h = F_{h+3} - 1$ by induction.

h	n_h	F_h	$F_h - 1$
0	1	0	—
1	2	1	0
2	4	1	0
3	7	2	1
4	12	3	2
5	20	5	4
6	33	8	7
7	54	13	12
8	88	21	20

From the table you can notice that $n_0 = F_3 - 1$, $n_1 = F_4 - 1$. It is our base step.

Inductive step.

Assume that $n_h = F_{h+3} - 1$ for $h \leq k$ so

$n_{k-1} = F_{k+2} - 1$ for $k - 1$

and

$n_k = F_{k+3} - 1$ for k

We will prove that $n_{k+1} = F_{k+4} - 1$ for $k + 1$.

Proof: By definition of the Fibonacci tree with height $k + 1$,

$n_{k+1} = n_k + n_{k-1} + 1$ for $k + 1$

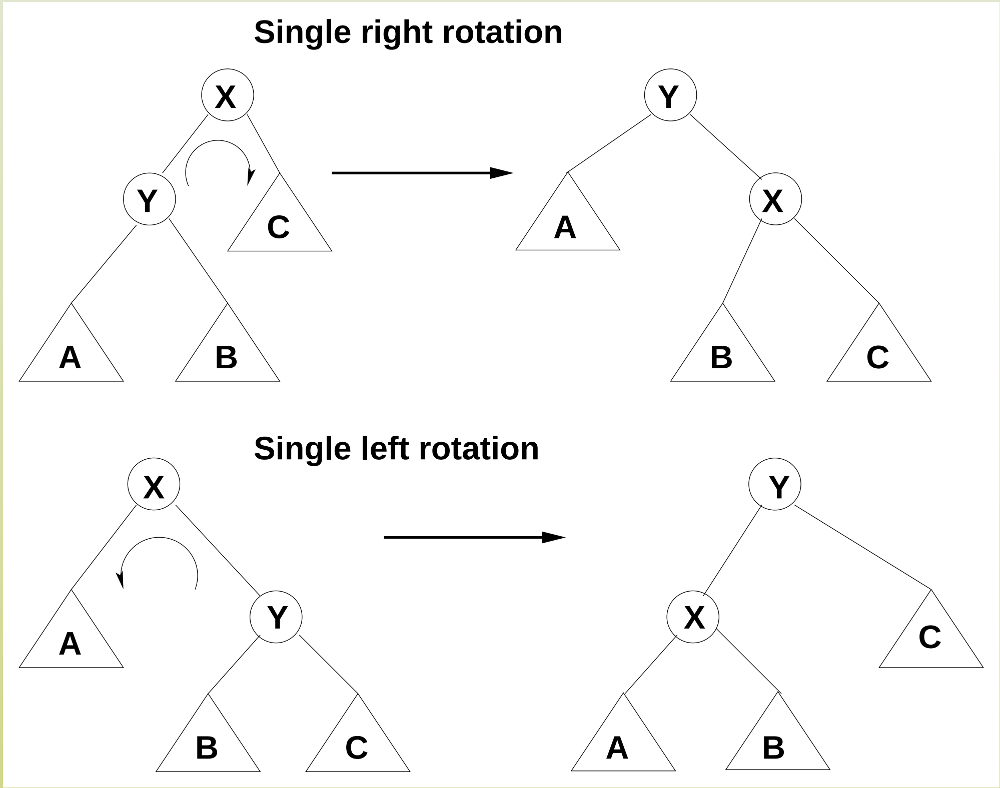
From the inductive step $n_{k+1} = F_{k+3} - 1 + F_{k+2} - 1 + 1 = F_{k+4} - 1$

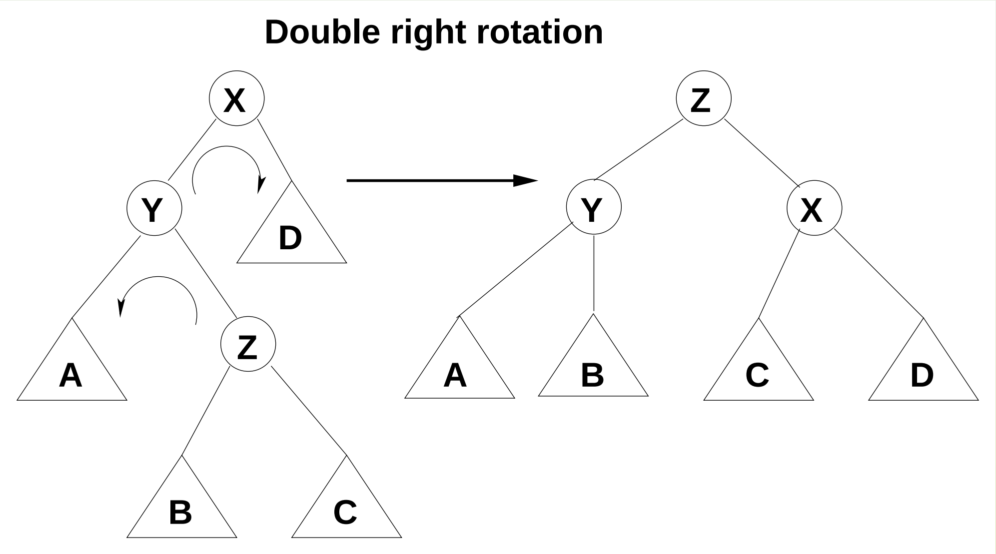
AVL Rotations

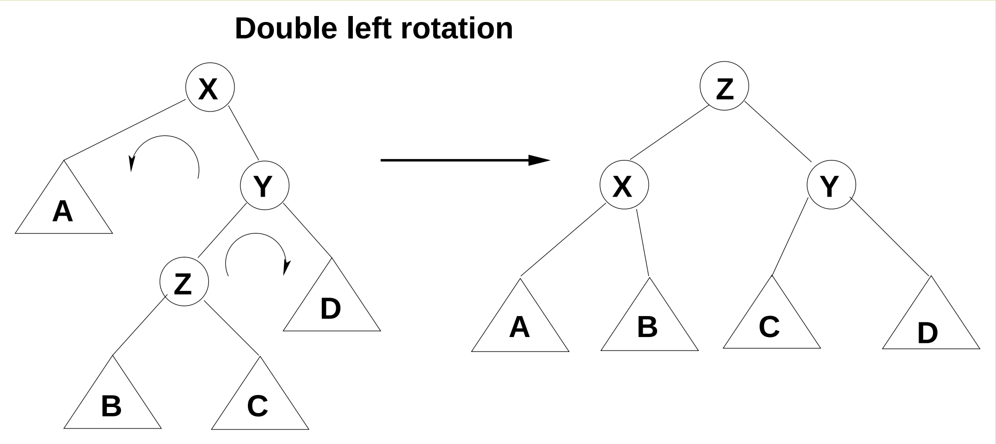
Assume that the node to be rebalanced is X. A violation might occur in any of four cases:

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

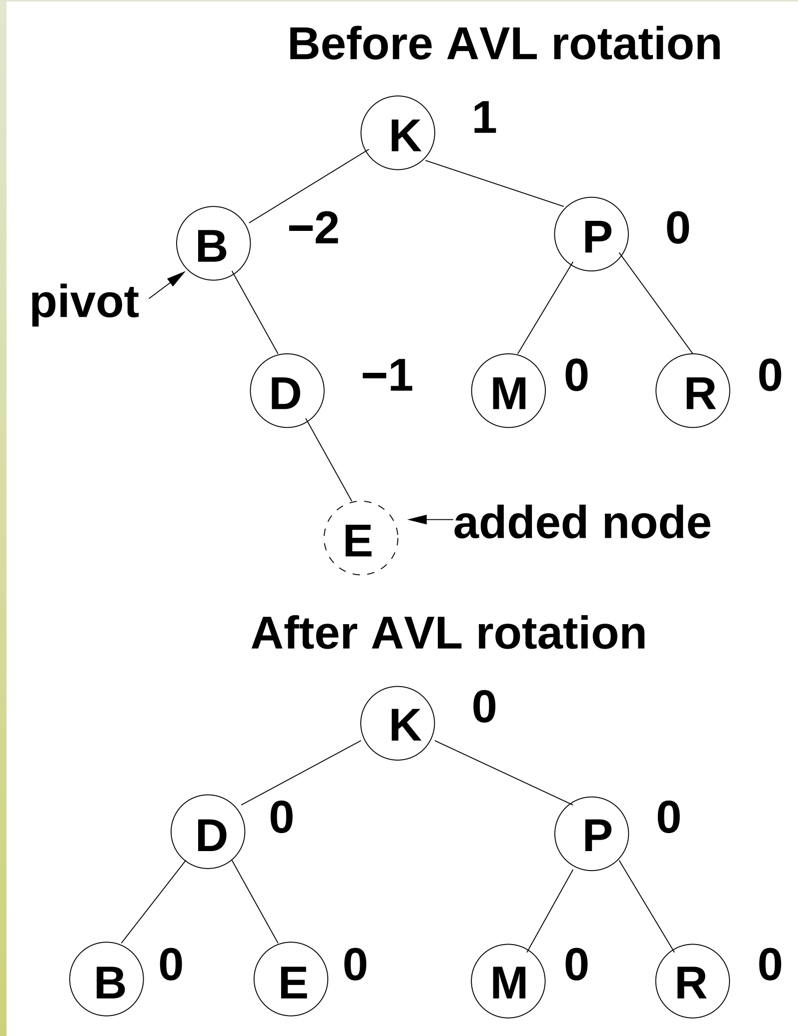
Cases 1 and 4 are mirror-image symmetries with respect to X, as are cases 2 and 3. The cases 1 and 4 are fixed by a *single rotation* of the tree. The cases 2 and 3 are fixed by (slightly more complex) *double rotation*. These rotations are sufficient to maintain the binary search tree balanced.


Single right rotation
X
Y
C
A
B
Y
A
X
B
C
Single left rotation
X
A
Y
B
C
Y
X
C
A
B


Double right rotation
X
Y
D
A
Z
B
C
Z
Y
X
A
B
C
D


Double left rotation
X
A
Y
Z
D
B
C
Z
X
Y
A
B
C
D

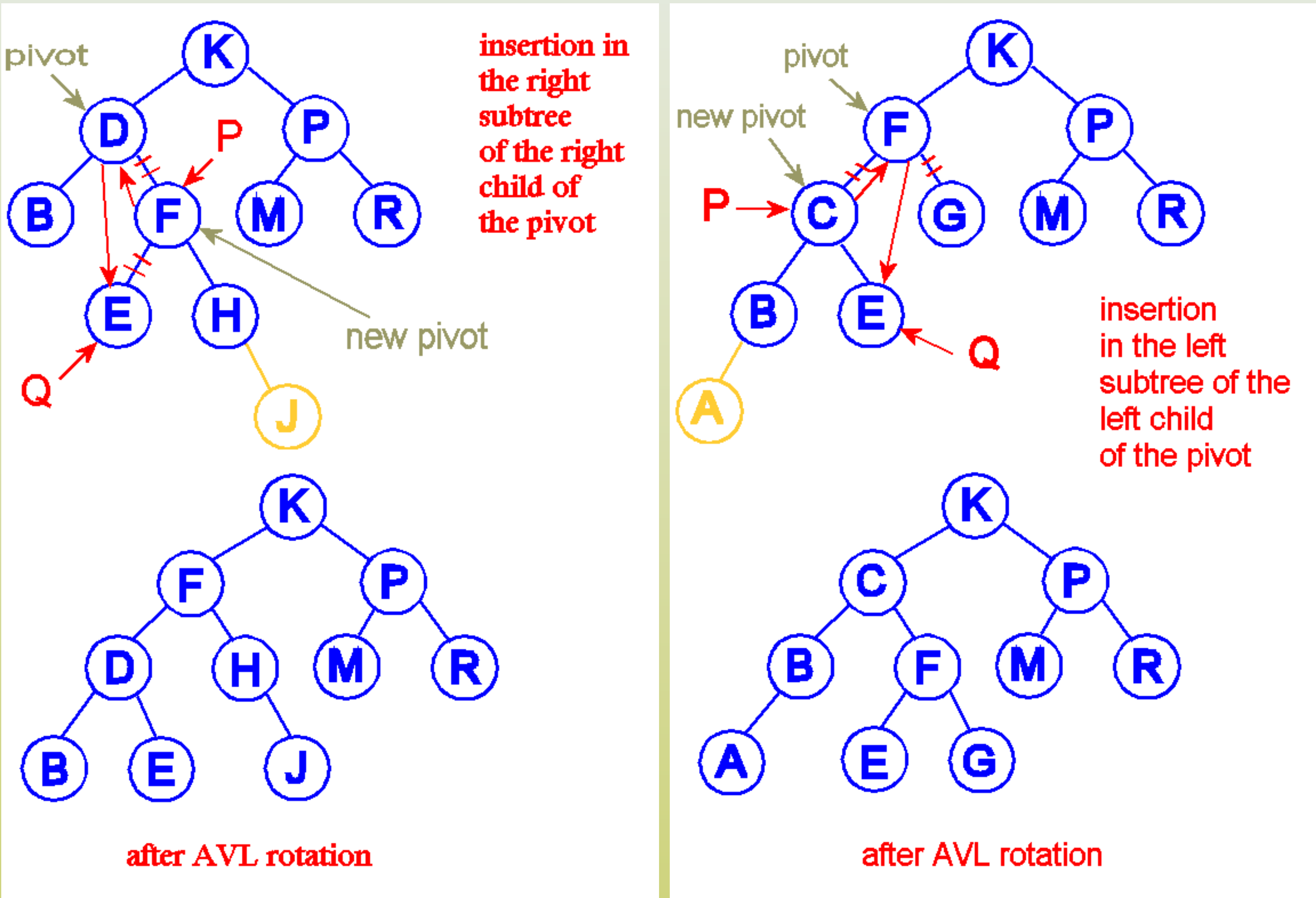
Example

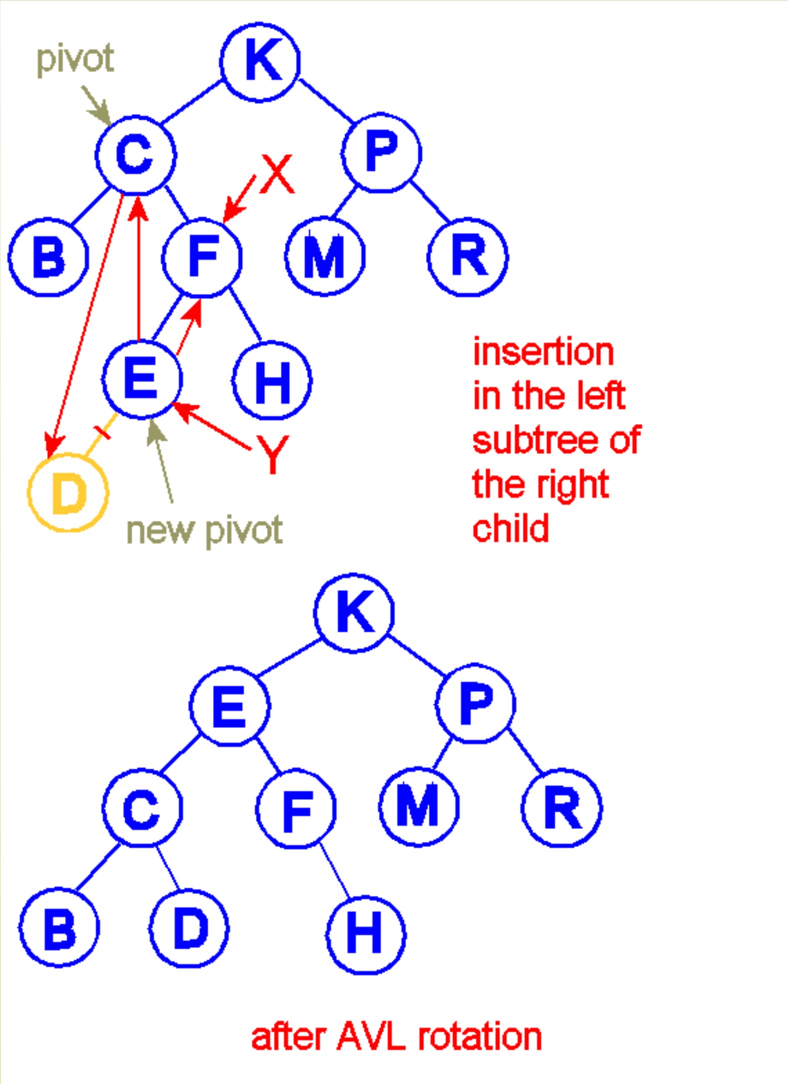

Before AVL rotation
K
1
B
−2
P
0
pivot
D
−1
M
0
R
0
E
added node
After AVL rotation
K
0
D
0
P
0
B
0
E
0
M
0
R
0

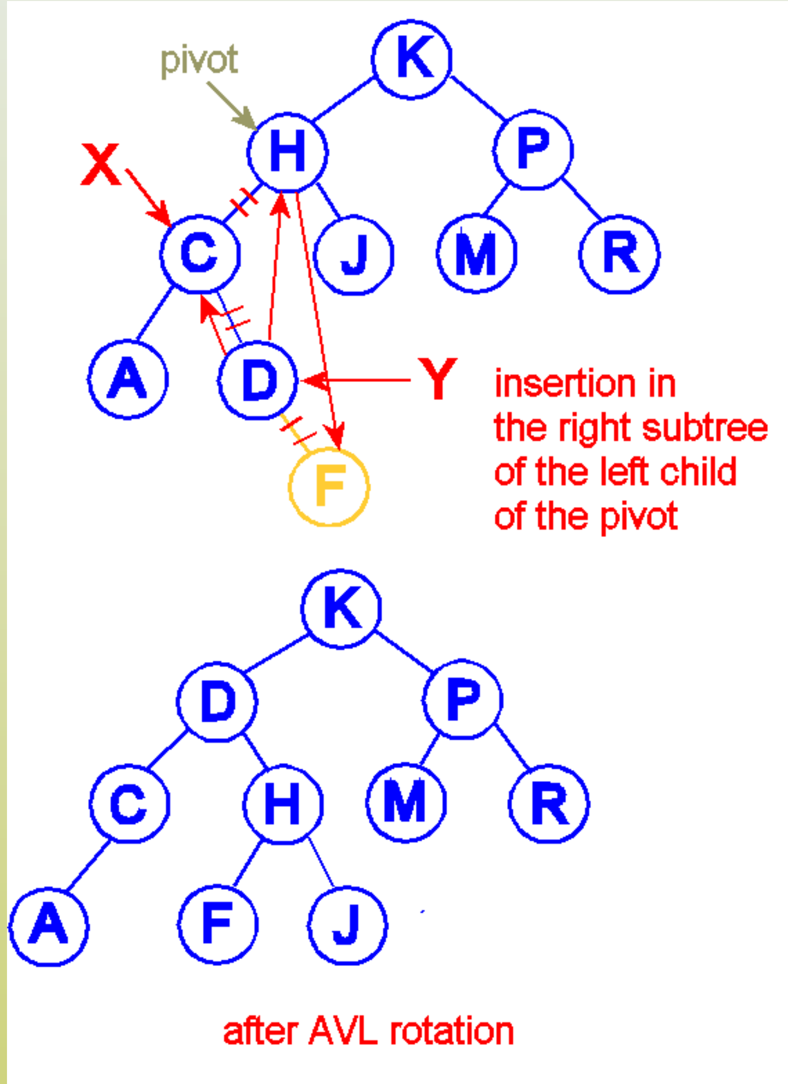
C++ Codes for Rotations

```
//single right rotation
BinaryNode* singleRight(BinaryNode* X)
{
   BinaryNode Y = X->left;
   X->left = Y->right;
   Y->right = X;
   return Y;
}
//single left rotation
BinaryNode* singleLeft(BinaryNode* X)
{
   BinaryNode Y = X->right;
   X->right = Y->left;
   Y->left = X;
   return Y;
}
```

```
// double right rotation
BinaryNode* doubleRight(BinaryNode* X)
{
   X->left = singleLeft(X->left);
   return singleRight(X);
}
// double left rotation
BinaryNode* doubleLeft(BinaryNode* X)
{
   X->right = singleRight(X->right);
   return singleLeft(X);
}
```


pivot
K
D
P
B
F
M
R
E
H
J
Q
new pivot
insertion in the right subtree of the right child of the pivot
K
F
P
D
H
M
R
B
E
J
after AVL rotation
pivot
new pivot
K
F
P
C
G
M
R
B
E
A
Q
insertion in the left subtree of the left child of the pivot
K
C
P
B
F
M
R
A
E
G
after AVL rotation


pivot
K
C
P
X
B
F
M
R
E
H
insertion in the left subtree of the right child
D
Y
new pivot
K
E
P
C
F
M
R
B
D
H
after AVL rotation


pivot
K
X
H
P
C
J
M
R
A
D
Y
insertion in the right subtree of the left child of the pivot
F
K
D
P
C
H
M
R
A
F
J
after AVL rotation