AVL

Definition
An AVL tree is a binary search tree which satisfies:
the heights of the two child sub trees of any node differ by at most one

Remark:
Representation stores the balance factor or the height of the node

Operations over AVL

- search, insert and delete
    all take O(log n) time in average and worst cases
    where n is the number of nodes in the tree prior to the operation.

Consider the next representation:

```
AVLTreeNode  =     record
                       info: TComparable
                       left: ^ AVLTreeNode
                       right: ^ AVLTreeNode
                       h: Integer
                   end
```

Search

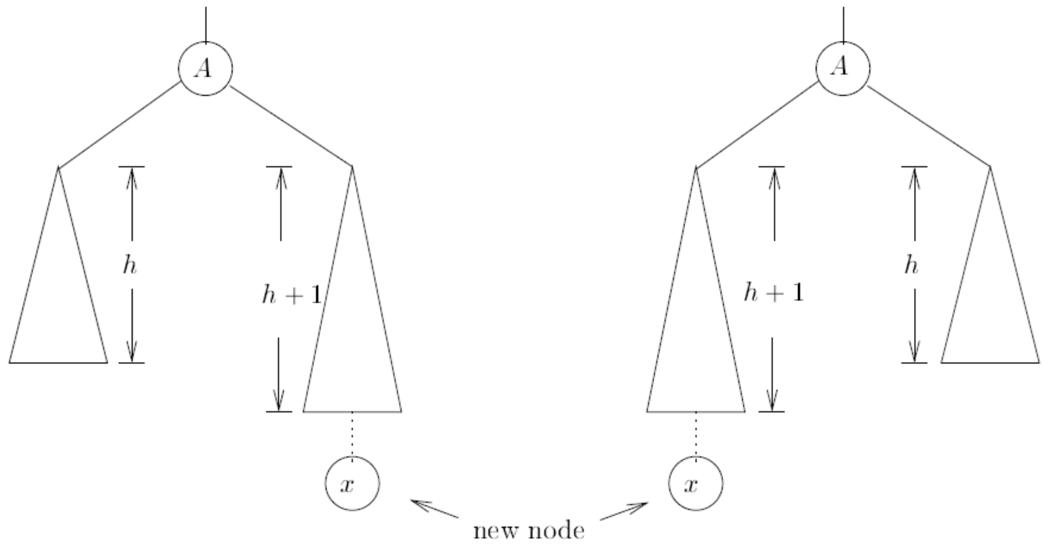
- BST search

Insert

Insertion:

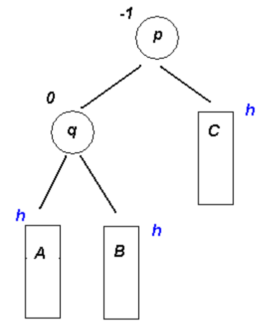
- require the tree to be rebalanced
    - insert an element like in BST case
    - rebalance the tree (if it is the case)
        consider all the ancestors (to the root)
        rebalance → one or more tree rotations.

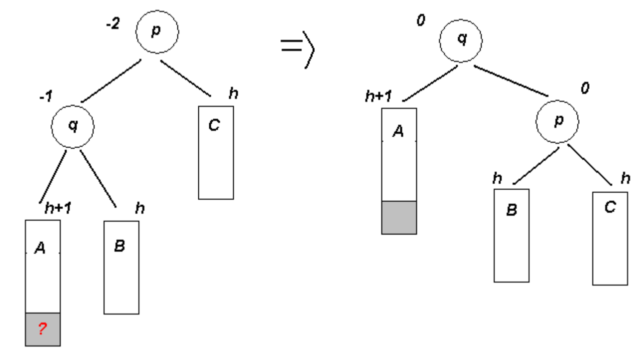
When to rebalance :

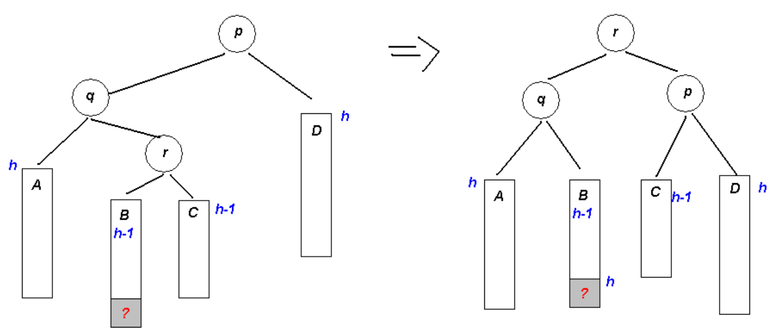


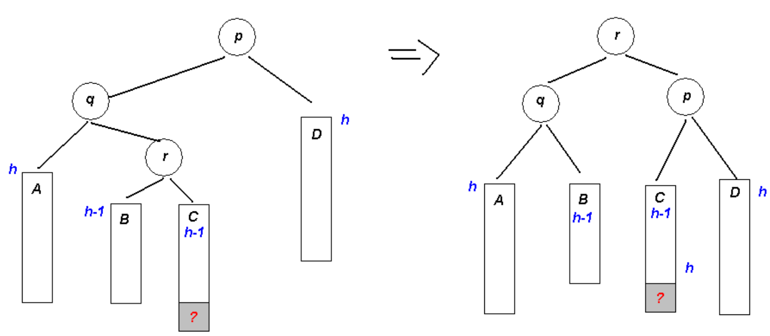
Insert cases - examples

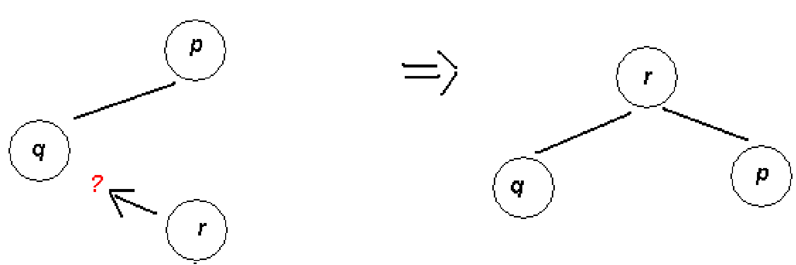
*Assume: (*next) initial situation & new node on the left subtree



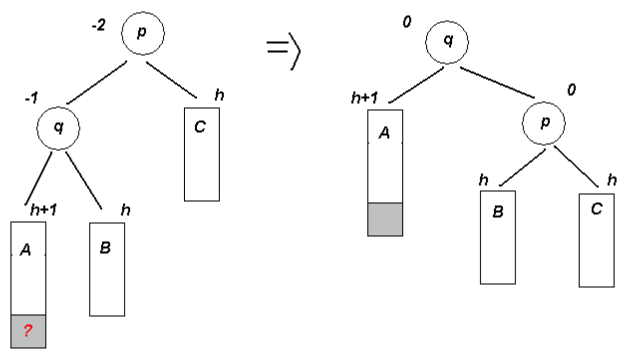








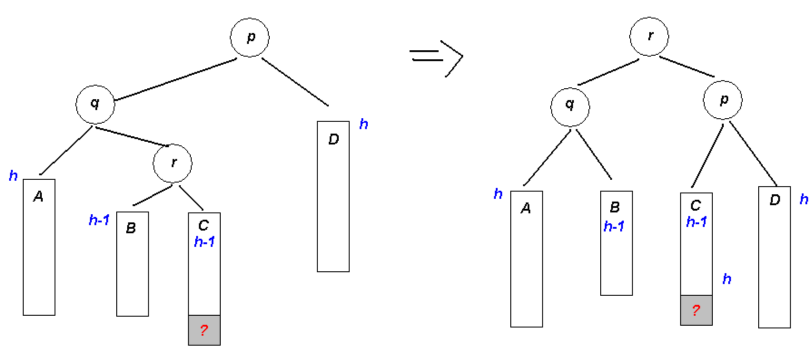
Rotations



/* ***Representation without link to parent*** */
/* Update heights, then return new root */

```
Function RotateRight ( p )
    q := p^.left
    p^.left := q^.right;
    q^.right := p;
    p^.h := Max( Height( p^.left ), Height( p^.right ) ) + 1;
    q^.h := Max( Height( q^.left ), Height ( q^.right ) ) + 1;
   RotateRight := q  /* New root */
end_RotateRight

Functin RotateLeft ( p )
    q := p^.right;
    p^.right = q^.left
    q^.left = p
    p^.h = Max( Height( p^.left ), Height( p^.right ) ) + 1
    q^.h = Max( Height( q^.left ), Height( q^.right ) ) + 1
   RotateLeft :=q
end_RotateLeft
```




```
Function DblRotateLeftRight ( p)
    p^.left := RotateLeft (p^.left )
    DblRotateLeftRight := RotateRight ( p );
end_DblRotateLeftRight
```

```
Function insert_rec(p , el)
// ElementType el, AvlTreeNode p
// return the new p
  if( p = NIL)
        p := new AvlTreeNode
        p^.info := el
        p^.h := 0;
        p^.left := NIL
        p^.right := NIL
else
        if( el < p^.info)  then
                p^.left := insert_rec(p^.left , el )
                if(Height(p^. right) - Height( p^. left ) = -2 )
                        if( el < p^.left^.info)
                                p := RotateRight ( p )
                        else
                                p := DblRotateLeftRight ( p )
                        endif
                endif
        else            // el >= [p].info
                p^.right = insert_rec(p^.right , el )
                if( Height( p^.right ) - Height( p^.left ) = 2 )
                        if( el > p^.right^.info ) then
                                p := RotateLeft ( p )
                        else
                                p := DblRotateRightLeft( p );
                        endif
                endif
        endif
        p^.h := Max( Height( p^.left ), Height( p^.right ) ) + 1;
endif
insert_rec := p
End_insert_rec

Subalg. insert(T , el)
        p := getRoot(T)
        np :=insert_rec(p, el)
        setRoot(T, np)
end_insert
```

Delete

- find the node x where k is stored
- delete the contents of node x ~similar with BST

    Deleting a node in an AVL tree can be reduced to deleting a leaf
    (next) alg. delete_rec – delete leaves

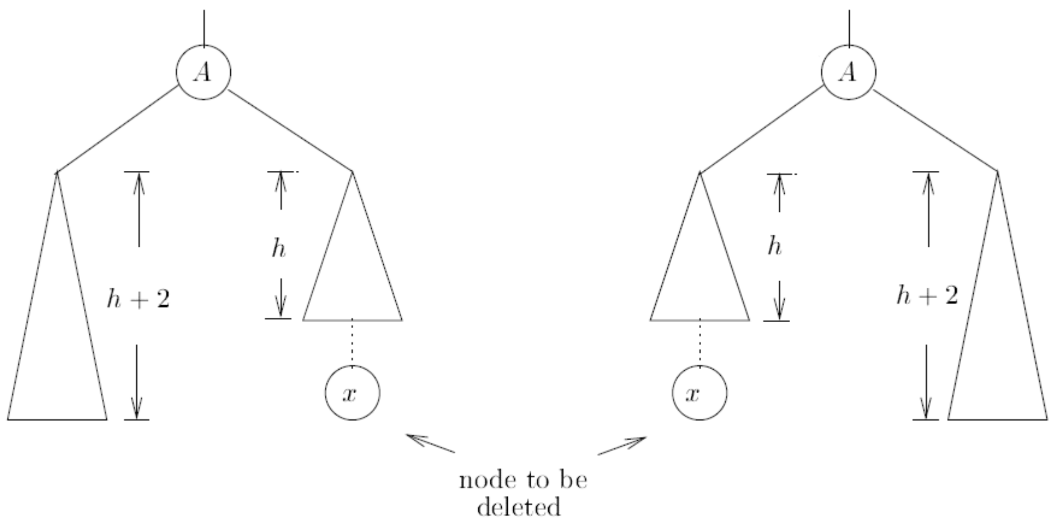
rebalance

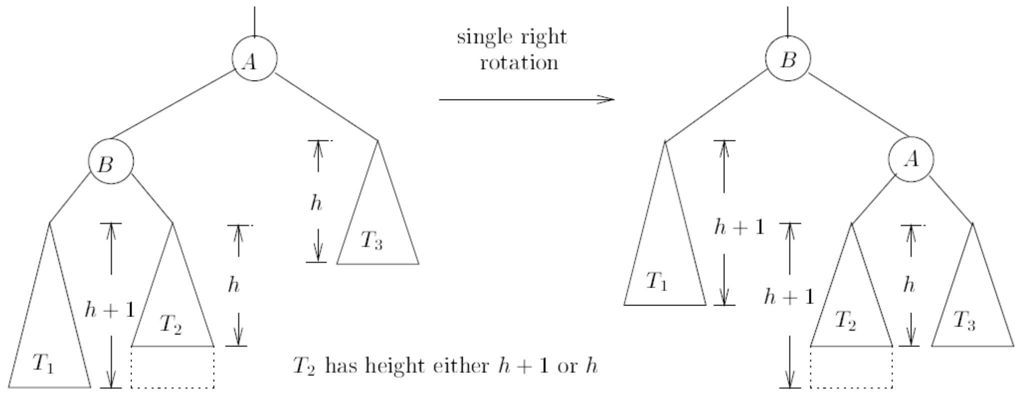
    go from the deleted leaf towards the root

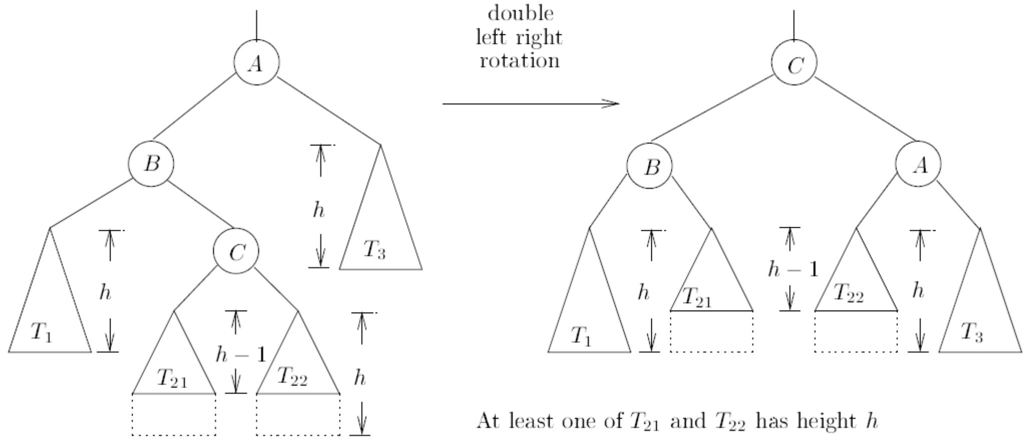
        -update the balance factor

        -rebalance with rotations if necessary.

Rebalance cases








```
Function delete_rec (p , el)
if p = NIL then return NIL endif
if el = p^.info then
        if p^.left=NIL and p^.right=NIL then
                delete p; p:=NIL
        else    q:=p
                if p^.left <> NIL       then    p:=p^.left
                                        else    p:=p^.right
                endif
                delete q
        endif
else
        if( el > p^.info) then
                p^.right = delete_rec ( p^.right , el );
                if( Height( p^.right ) - Height( p^.left ) = -2 )  then
                        if(Height( p^.left ^.left) = Height( p^.right ) +1) then
                                p = RotateRight ( p )
                        else // Height( [[p].left ].left = Height( [p].right )
                                p = DblRotateLeftRight(p)
                        endif
                endif
        else // el( el < [p].info)
                        //…
        endif
endif
if p<> NIL then p^.h = Max( Height( p^.left ), Height( p^.right ) ) + 1 endif
```

```
delete_rec := p
End_ delete_rec
```