Fundamental Algorithms

Lecture #7

Cluj-Napoca

Agenda

- **Augmented trees**
  - **Type 2 – Tree/lists**
- **Balanced trees – why and how (review)**
- **Red-Black Trees (balanced trees type 3)**

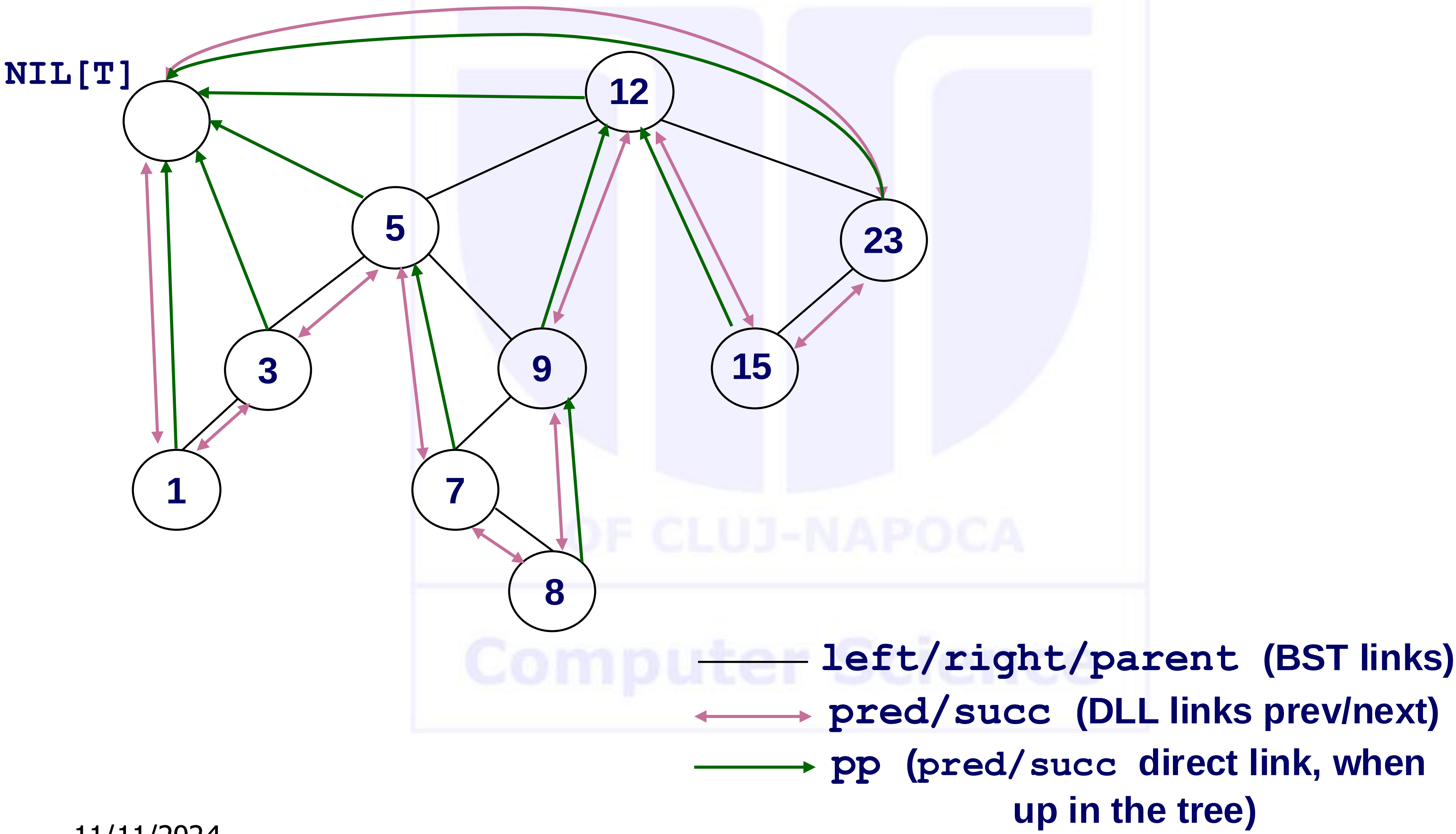
Augmented trees (type 2)

- Requirements:
  - Regular operations are performed as (**same performance also**) in BST (walk (**O(n)**), search, ins, del (**O(h)**))
  - Several other operations are enhanced (i.e. performe faster)
    - Succ
    - Pred
    - Min
    - Max
    - **All required to be performed in O(1)!!!**
  - BUT NONE of the before-defined operations should degrade their performance

Augmented trees – contd.

- Info in a node:
  - Usual info:
    - key
    - left pointer
    - right pointer
    - parent pointer
  - Supplementary info (see picture on the blackboard):
    - succ pointer
    - pred pointer (together ensure walking through the list)
    - pp ensures min/max oper. (in a regular BST, succ/pred calculated **either** based on min/max **or** pp - which is determined at the execution time)

Example


NIL[T]
12
5
23
3
9
15
1
7
8
left/right/parent (BST links)
pred/succ (DLL links prev/next)
pp (pred/succ direct link, when up in the tree)

Augmented trees – contd.

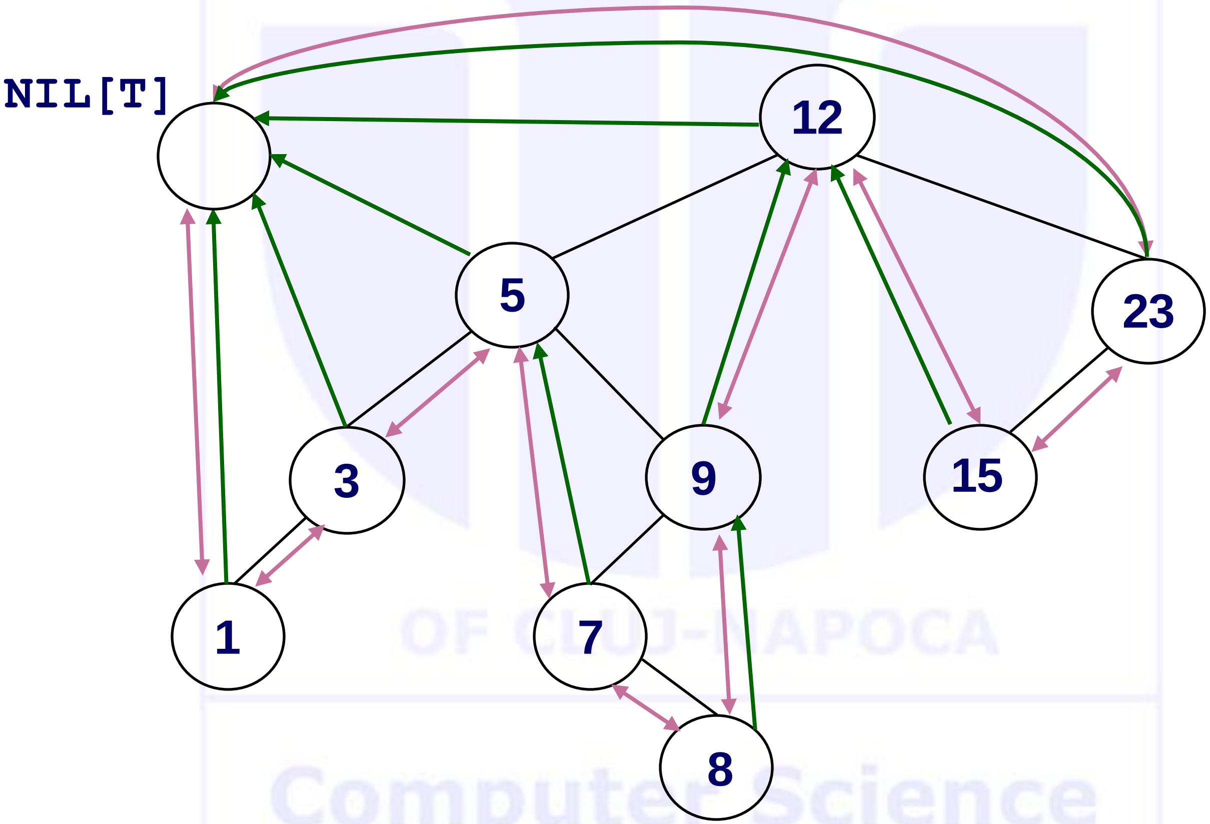
- The structure acts BOTH as a BST and DLL!!
- Regular operations are:
  - done like in any other BST
  - in addition, need to make some updates
- They (the additional updates) refer to:
  - making the appropriate links within the DLL (set/update the *pred* and *succ* pointers)
  - link the double pointer (set/update the *pp* pointer)

Augmented trees – Insert

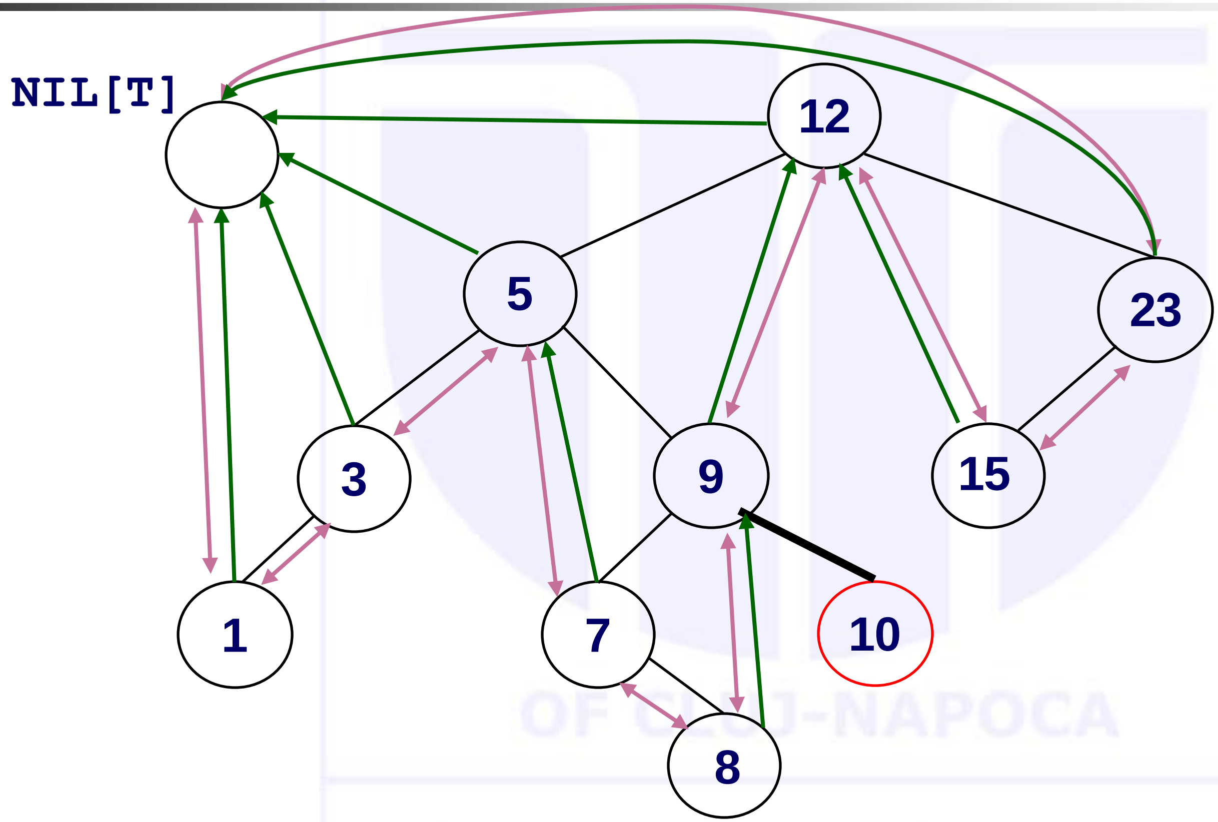
- Regular **insert** operation in a BST (x inserted) **+**

```
if x=right[p[x]]   //node inserted = right child
  then    //case #1
     pp[x]<-succ[p[x]]
     dl_list_ins_after(p[x],x)
  else    //case #2; node inserted = left child
     pp[x]<-pred[p[x]]
     dl_list_ins_after(pp[x],x)
```

AugBST-Insert(T, 10)

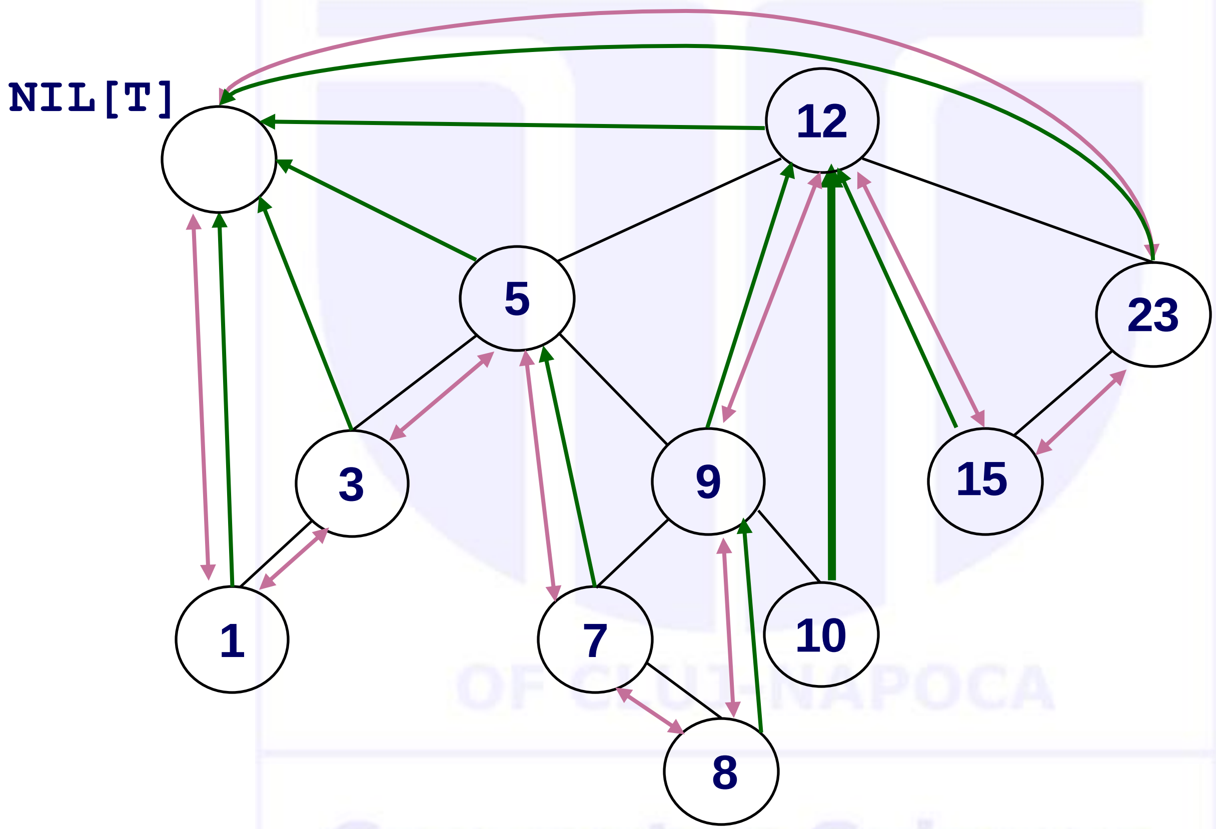


AugBST-Insert(T, 10)



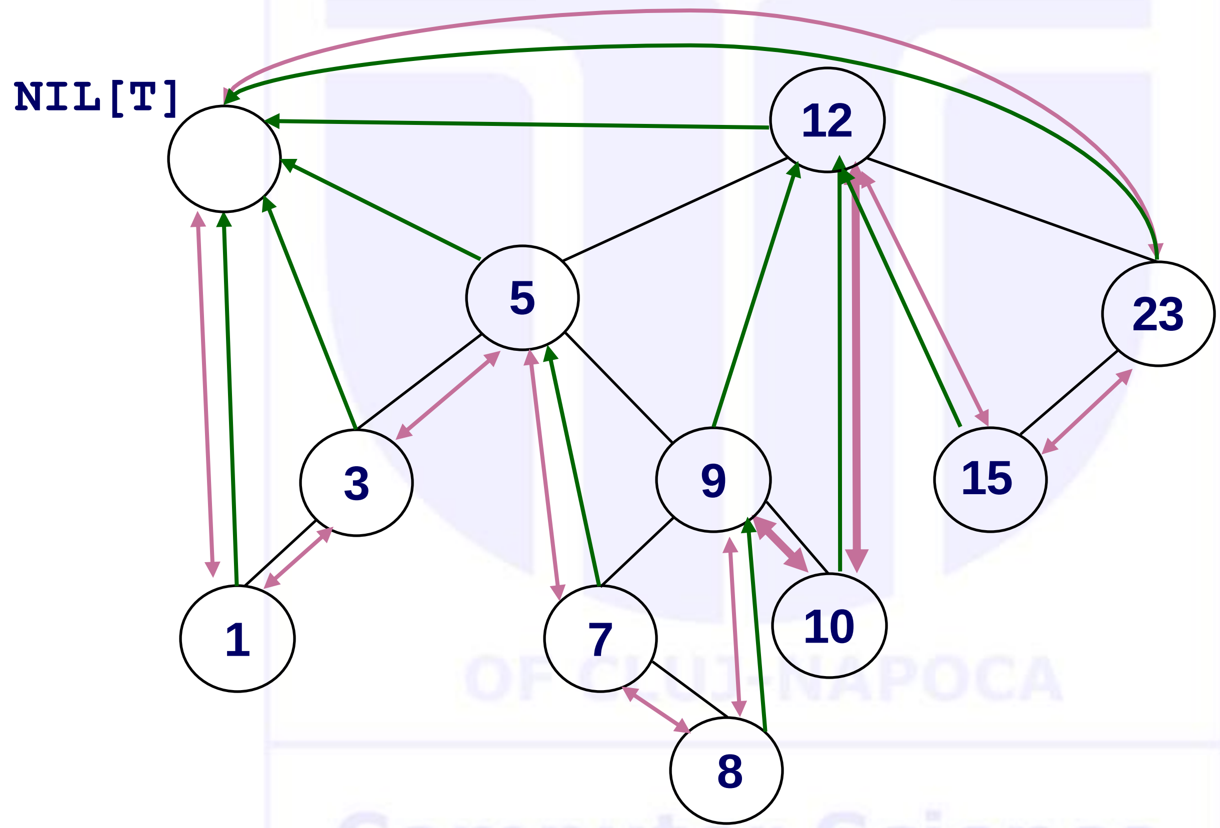
if ***x*=right[p[x]]** //node inserted = right child
then //case #1
pp[x]<-succ[p[x]]
dl_list_ins_after(p[x],x)

AugBST-Insert(T, 10)



if ***x***=right[p[x]] //node inserted = right child
then //case #1
pp[x]<-succ[p[x]]
dl_list_ins_after(p[x],x)

AugBST-Insert(T, 10)



if ***x*=right[p[x]]** //node inserted = right child
then //case #1
pp[x]<-succ[p[x]]
dl_list_ins_after(p[x],x)

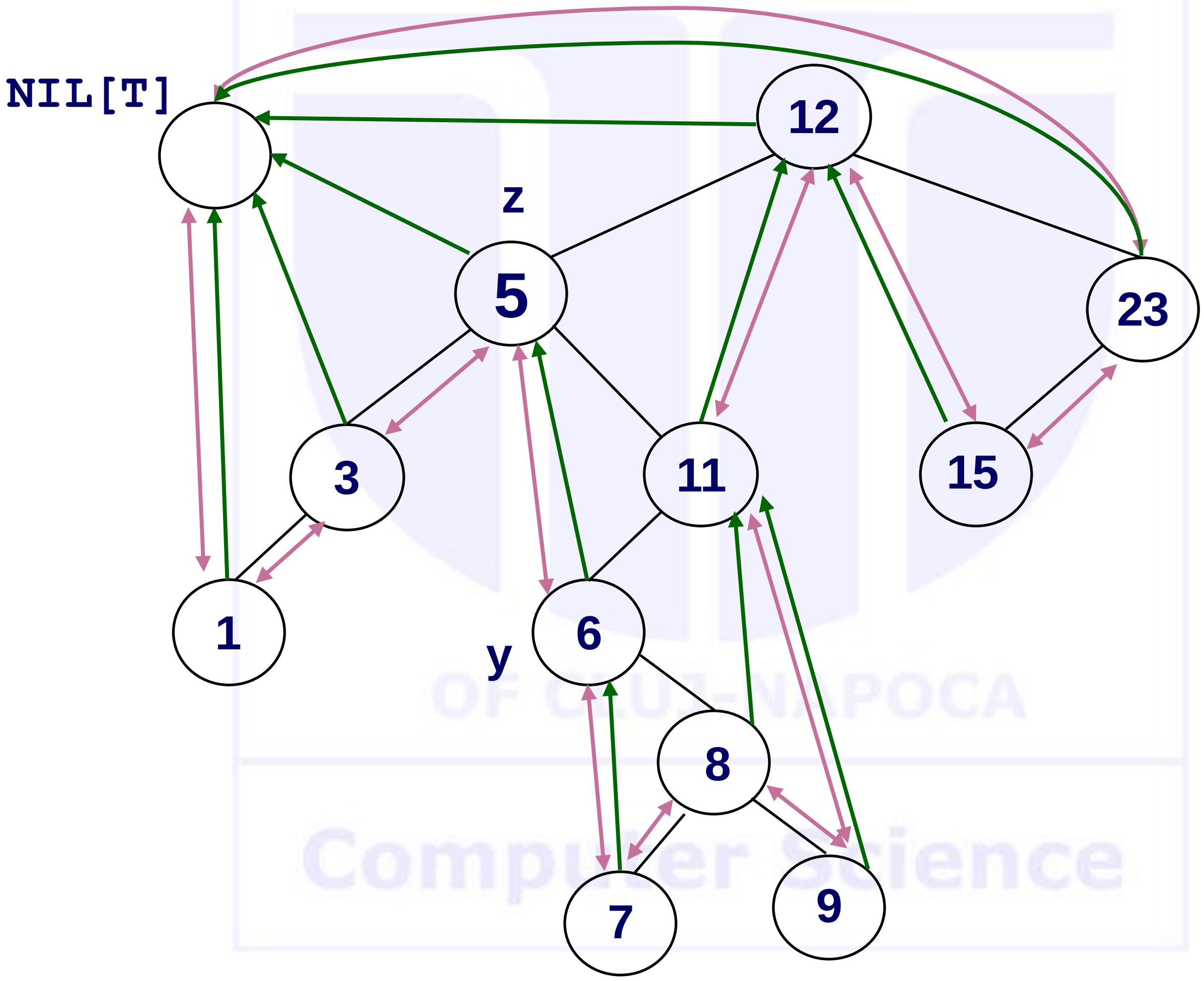
Augmented trees – Delete

(z = node requested to be removed; it's content is replaced by y's content
y=node actually removed = at most 1 child node;
x = its (y) only child/if any, might be nil;
z=y if z has at most one child)

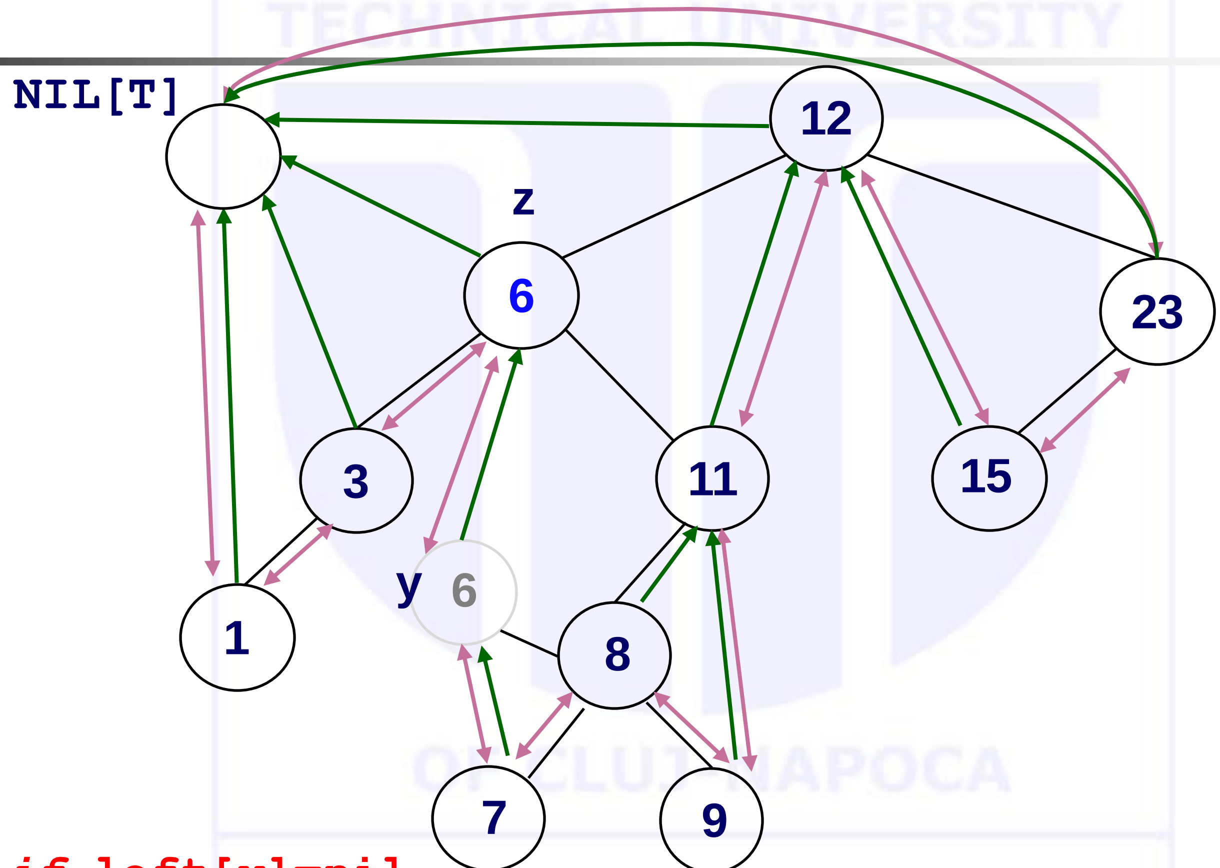
- Apply regular **delete** operation in a BST **+** code below

```
if right[y]=nil
  then                                    //no child to the right
      x<-left[y]                          //x = y's only child
      while x<>nil                        //along x's right branch
            do    pp[x]<-pp[y]            //update pp
                  x<-right[x]
      dl_list_del(y)
  else                                    //symmetric on the left
```

BST-Delete(T,5)

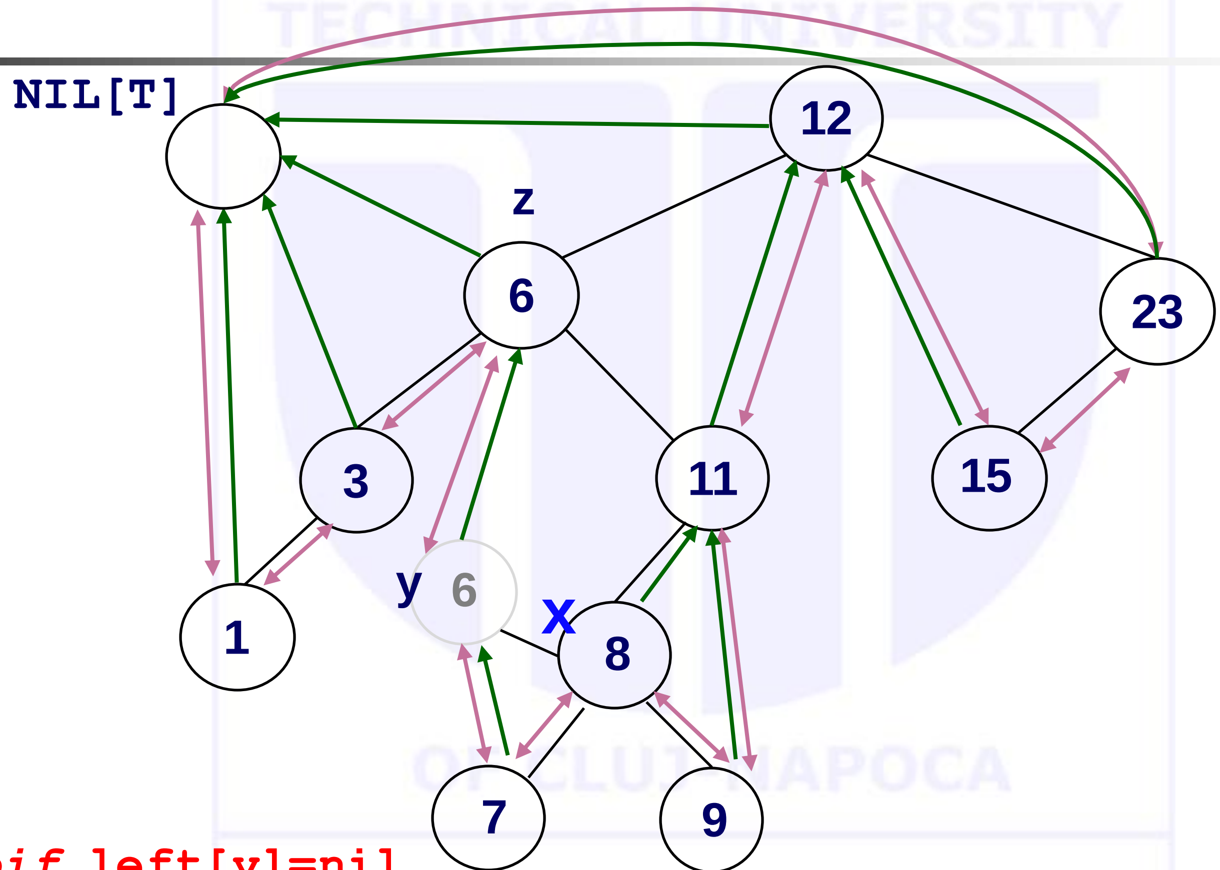


BST-Delete(T,5)



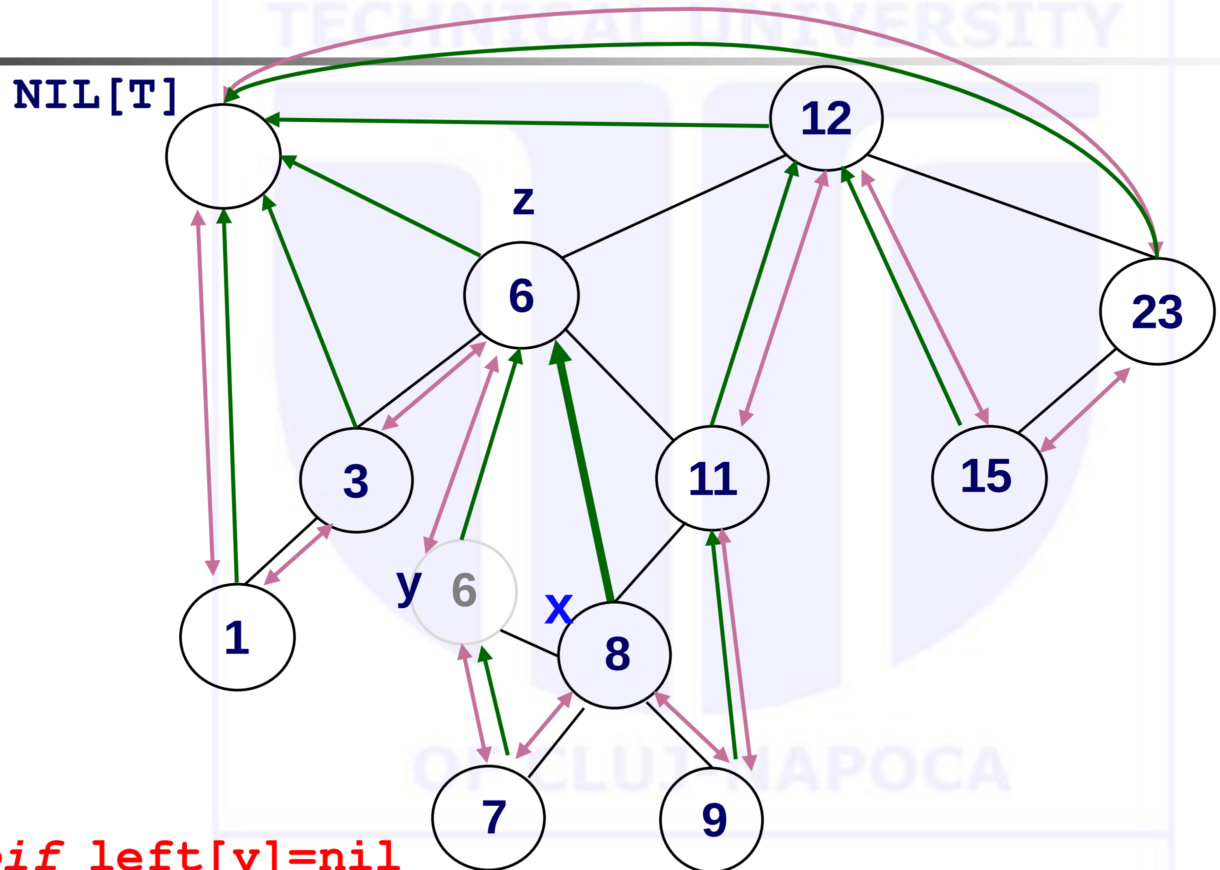

```
else if left[y]=nil
    then                              //no child to the left
        x<-right[y]                   //x = y's only child
        while   x <> nil              //along x's left branch
            do    pp[x]<-pp[y]        //update pp
                  x<-left[x]
        dl_list_del(y)
```

BST-Delete(T,5)



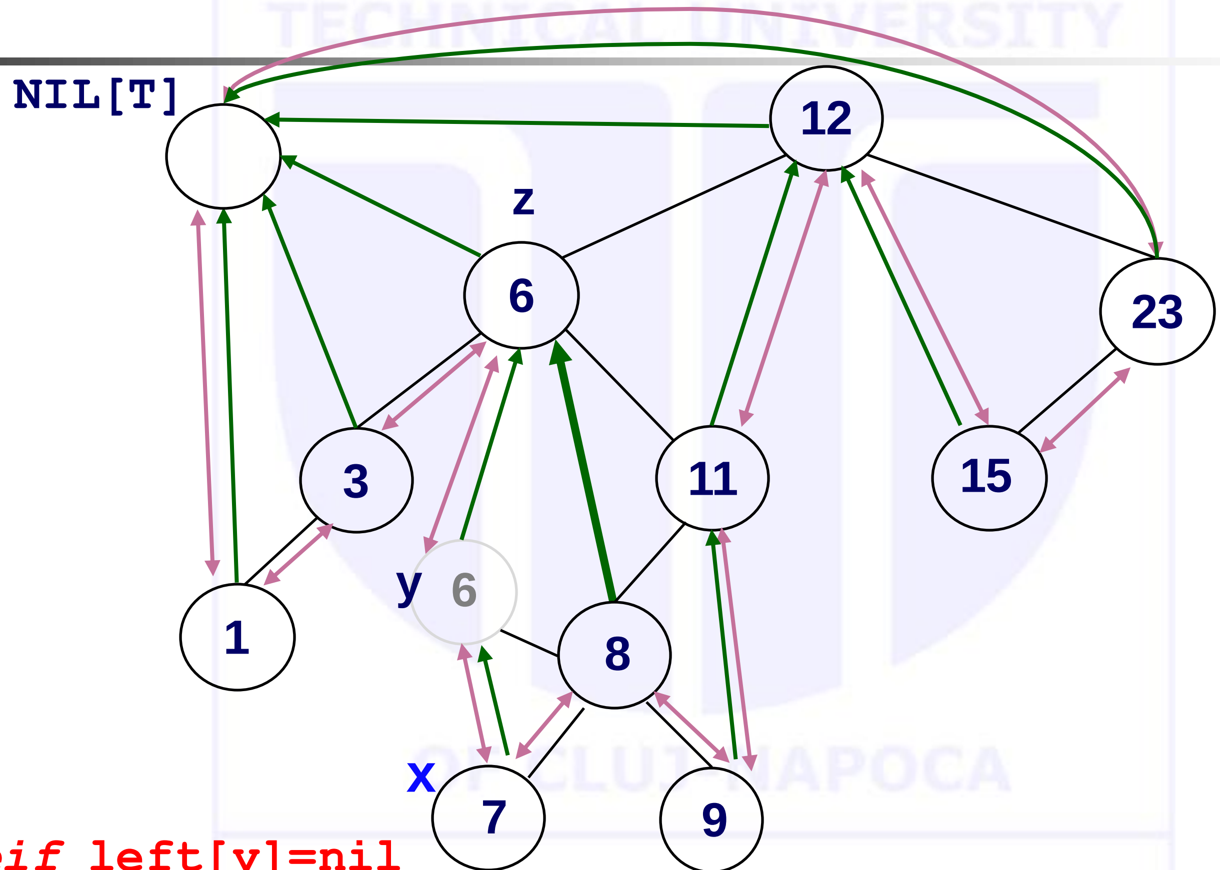

```
elseif left[y]=nil
    then
        x<-right[y]                //no child to the left
                                   //x = y's only child
        while    x <> nil          //along x's left branch
            do    pp[x]<-pp[y]          //update pp
                  x<-left[x]
        dl_list_del(y)
```

BST-Delete(T,5)



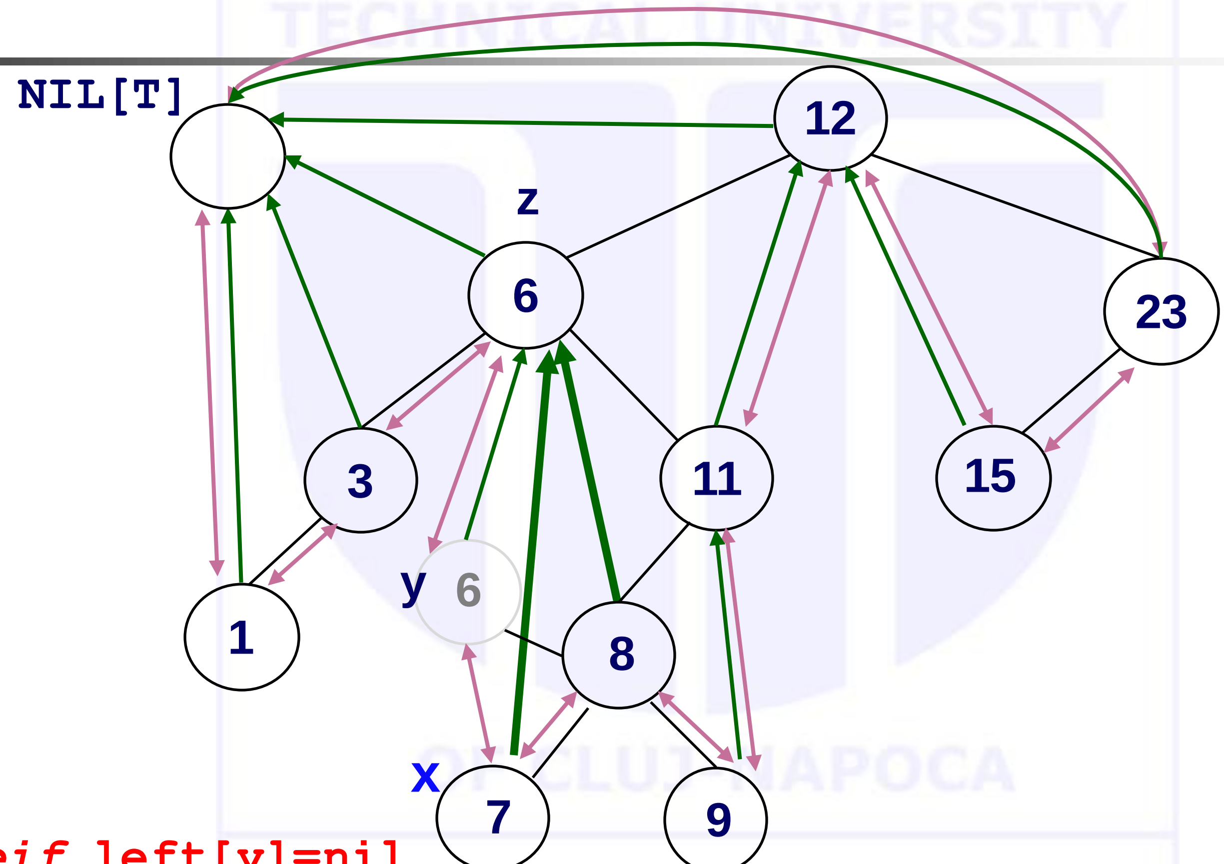

```
elseif left[y]=nil
    then                                   //no child to the left
        x<-right[y]                        //x = y's only child
        while   x <> nil                   //along x's left branch
            do     pp[x]<-pp[y]            //update pp
                   x<-left[x]
        dl_list_del(y)
```

BST-Delete(T,5)



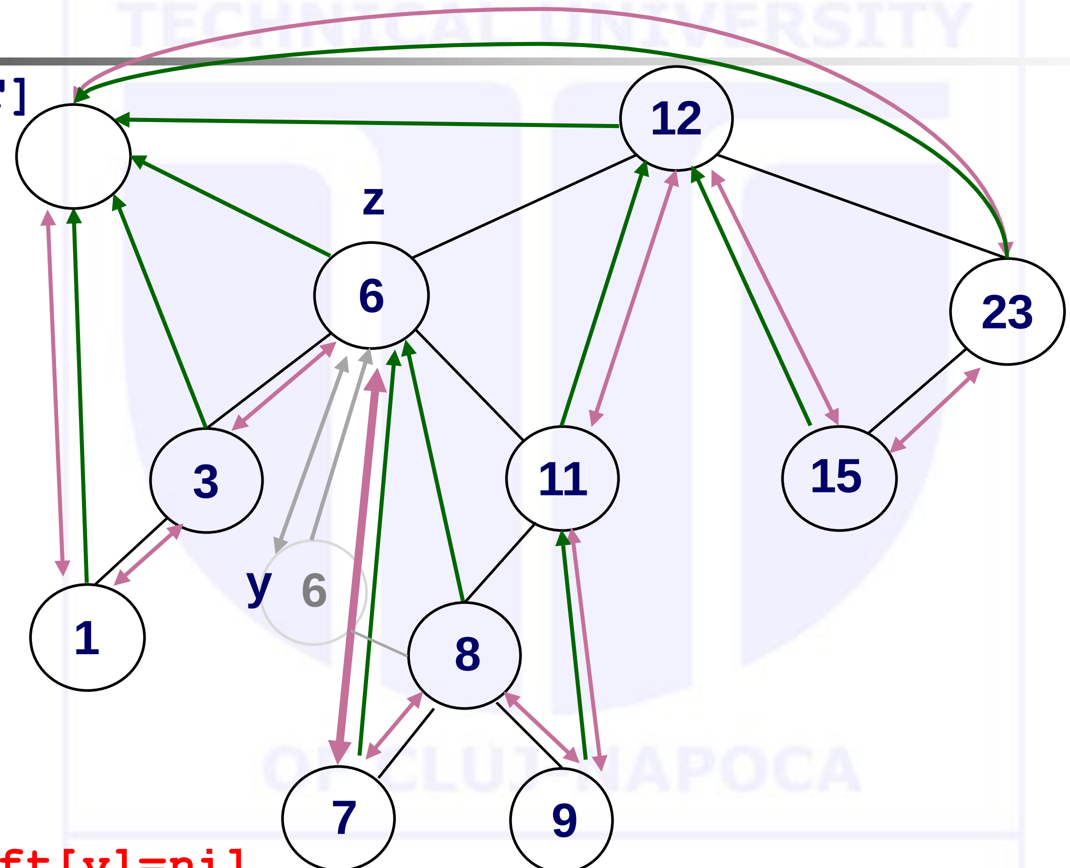

```
elseif left[y]=nil
    then                                //no child to the left
        x<-right[y]                     //x = y's only child
        while   x <> nil                //along x's left branch
            do   pp[x]<-pp[y]           //update pp
                 x<-left[x]
        dl_list_del(y)
```

BST-Delete(T,5)




```
elseif left[y]=nil
    then                          //no child to the left
        x<-right[y]               //x = y's only child
        while  x <> nil           //along x's left branch
            do   pp[x]<-pp[y]     //update pp
                 x<-left[x]
        dl_list_del(y)
```

BST-Delete(T,5)



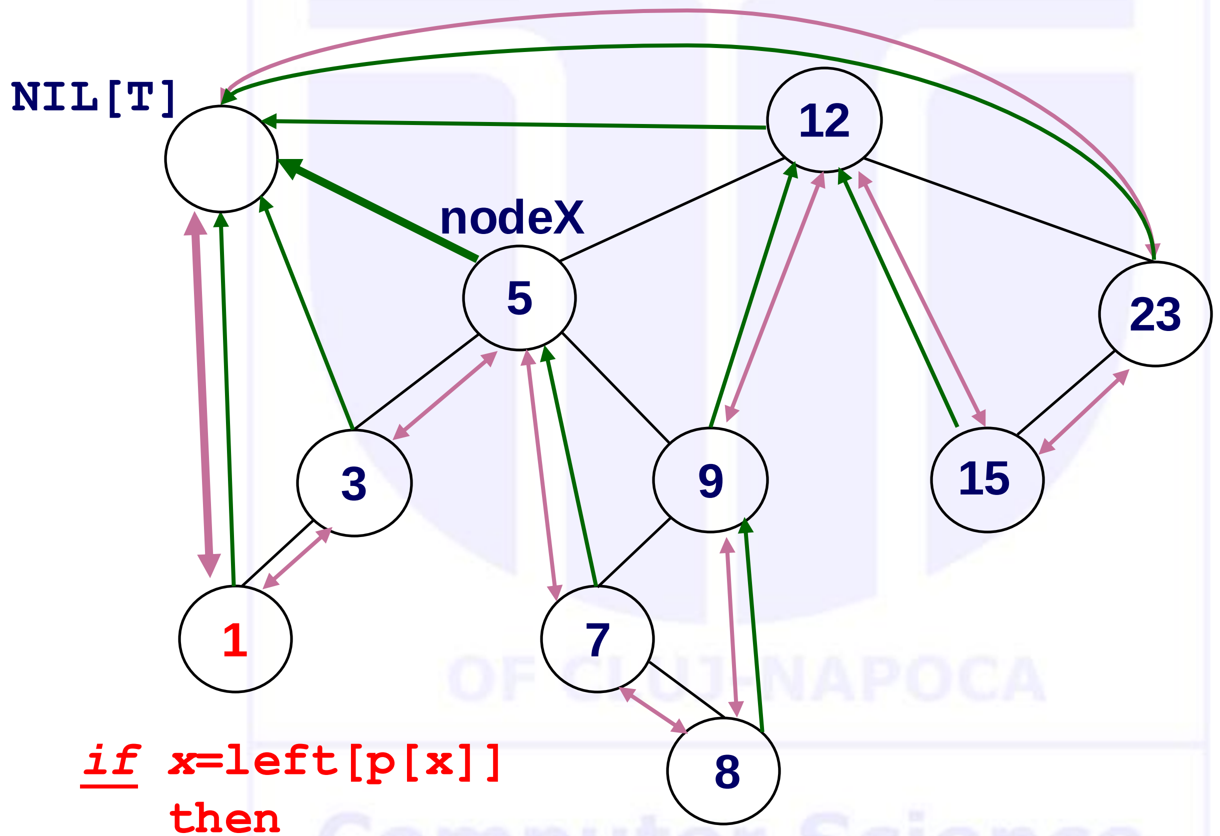

```
elseif left[y]=nil
    then                                //no child to the left
        x<-right[y]                     //x = y's only child
        while   x <> nil                //along x's left branch
            do    pp[x]<-pp[y]          //update pp
                  x<-left[x]
        dl_list_del(y)
```

Augmented trees – Min

- **min** (based on *succ* and *pp* as opposed to regular BST, where *succ* is calculated based on *min* **or** determined *pp*)

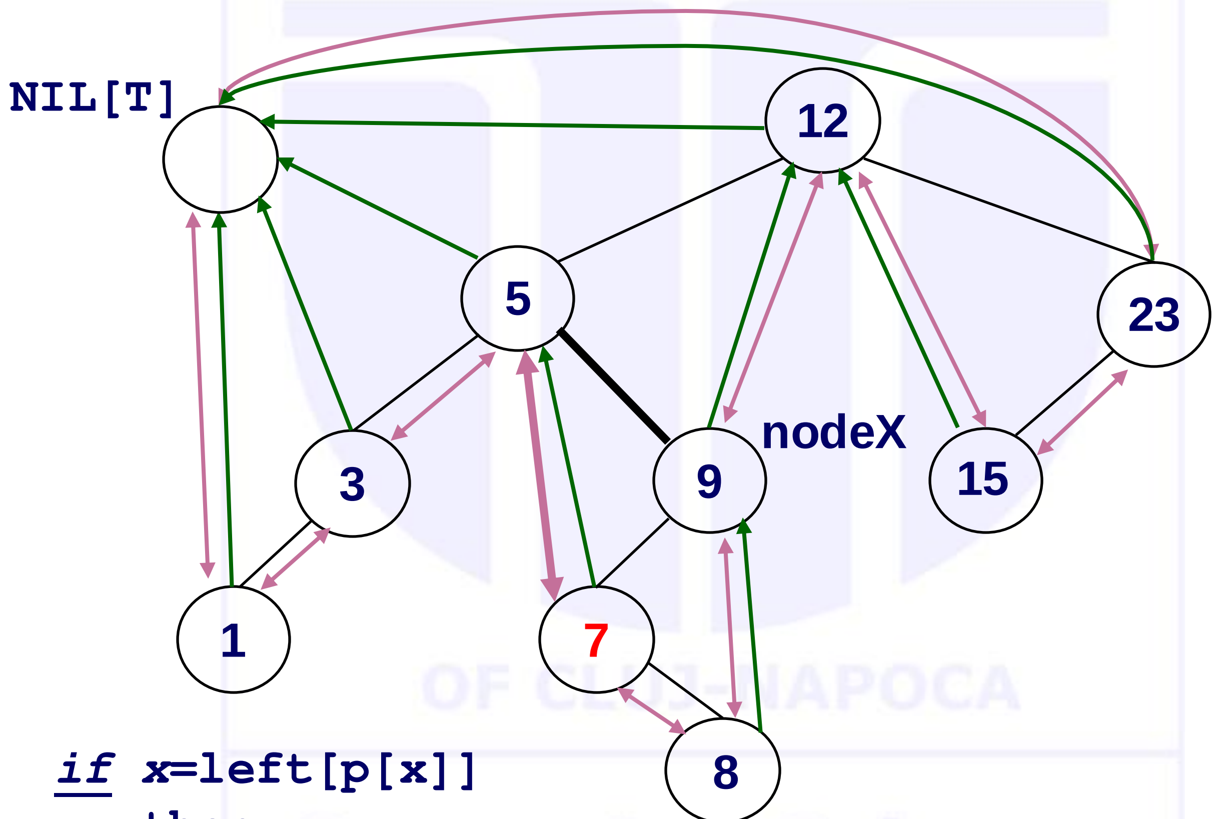
```
if x=left[p[x]]
  then
     return succ[pp[x]]
//on the leftmost branch, HAS TO BE  pp[x]=nil!!!
  else
     return succ[p[x]]
```

BST-Min(nodeX)




```
if x=left[p[x]]
   then
        return succ[pp[x]]
else
        return succ[p[x]]
```

BST-Min(nodeX)




```
if x=left[p[x]]
   then
        return succ[pp[x]]]
else
        return succ[p[x]]
```

Augmented trees – Max

- **max** (based on *pred* and pp as opposed to regular BST where *pred* is calculated based on *max* **or** determined *pp*)

```
if x=left[p[x]]
  then
     return pred[p[x]]
  else
     return pred[pp[x]]
```

//on the rightmost branch, **HAS TO BE** pp[x]=nil and pred[nil[T]] = last node in inorder = last node in the list

Augmented trees – contd.

- Particular (initial) cases discussion on the blackboard!
- First **insert** (in the empty tree)

```
Tree_ins(T,x)
if x = root[T]
   then        //the node just inserted is the root = tree was empty before
        root[T]<-x
        p[x]<-nil[T]
        pp[x]<-nil[T]
        dl_list_ins_after(pp[x],x)
   else        //the regular case described earlier
        …
```

Homework: updates for delete!

Red-Black trees

- Balanced trees
- Both insert/delete operations take *O(lgn)*, with at most O(*lgn*) for rebalancing

<u>Def</u>: A RBT is a BST with the following properties:

P_0: the root is black

P_1: each *node* is colored either **black** or **red**

P_2: each *leaf* (*NIL*) is **black**

P_3: both *children of a **red** node* are **black**

P_4: every *path* from <u>any</u> node to a leaf has the *same number of **black** nodes ("black height")*

RB Tree – Example
Is it a RBT? Why not?

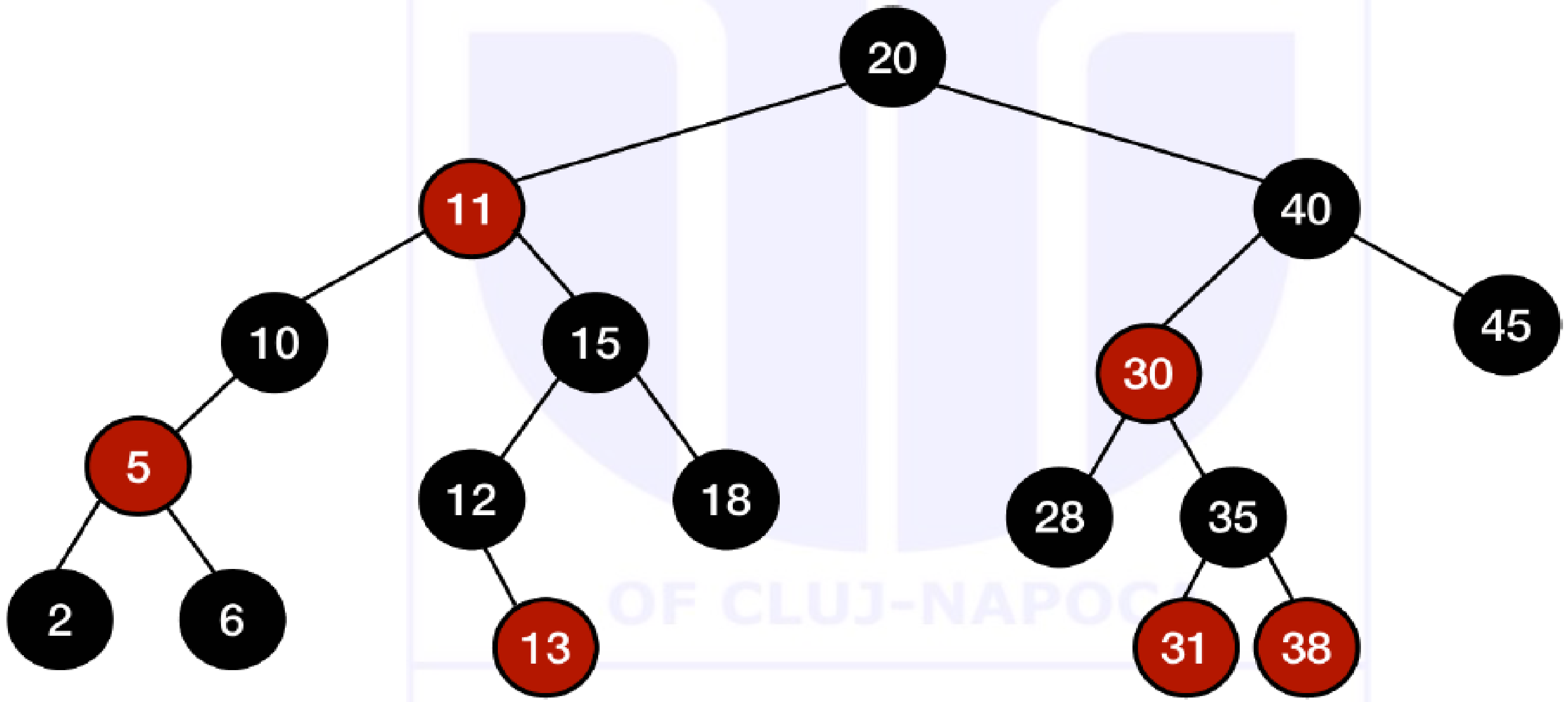


Image taken from: ***https://www.codesdope.com/course/data-structures-red-black-trees/***

Red-Black trees

Th: The height of a RB tree with n internal nodes is at most $2\lg(n+1)$

Proof: Let's denote by bh(x)=the black height (without x) of node x

Step 1: Define the statement P(bh) as follow:

P(bh): $\forall x \in RBT$, the tree rooted by x has at least $2^{bh(x)}-1$ nodes

Induction:

P(0) $2^0-1=0$

Assume P(bh) true =>P(bh+1) true?

x has 2 children; each child has the black height: if x is red: bh(x)

if x is black: bh(x)-1

nb of internal nodes of x= nb of internal nodes of children(x) +1

(itself) => at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1= 2^{bh(x)}-1$ *q.e.d.* (end of ***Step 1***)

Red-Black trees

Step 2:

We know P(bh) is true, i.e.

P(bh): $\forall x \in RBT$, the tree rooted by x has at least $2^{bh(x)}-1$ nodes (1)

By P_3 of RBT def (use contradiction to prove) $bh(x) \geq h/2$ (2)

//since after each red node comes a black one

$=> n \geq 2^{bh(x)}-1$ (from (1))

$\geq 2^{h/2}-1$ (from (2))

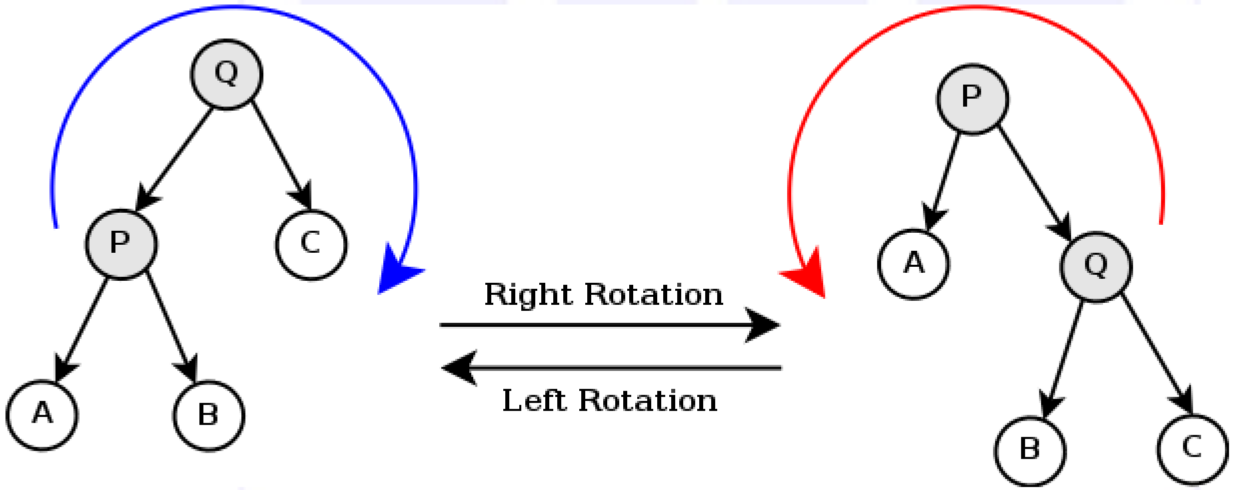
$n \geq 2^{h/2}-1 \Leftrightarrow n+1 \geq 2^{h/2} \Leftrightarrow h/2 \leq \lg(n+1) \Leftrightarrow h \leq 2\lg(n+1)$

q.e.d. (end of ***Th proof***)

Red-Black trees - rotations

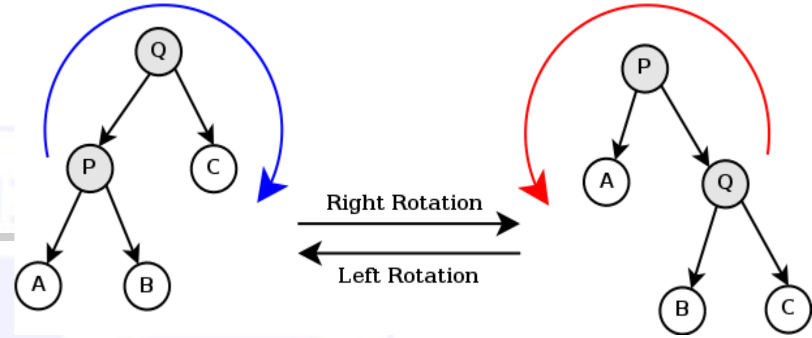
Similar to single rotations for AVL

They are symmetric



Picture from wiki

Red-Black trees - rotations




```
left_rotate(T,x)
//x root of rotation (points on P)
y<-right[x]                 //y saves Q
right[x]<-left[y]           //right of P goes on B
if left[y]<>nil             //if B exists = is not nil
      then p[left[y]]<-x    //B's parent becomes P
p[y]<-p[x]                  //Q's parent what was P's parent
if p[y]=nil          //P used to be the root of the tree
      then root[T]<-y
      else if x=left[p[x]] // the parent of P becomes the parent of Q
                 then left[p[x]]<-y
                 else right[p[x]]<-y
left[y]<-x           //P goes the left child of Q
p[x]<-y              //Q becomes the parent of P
```

RB-insert

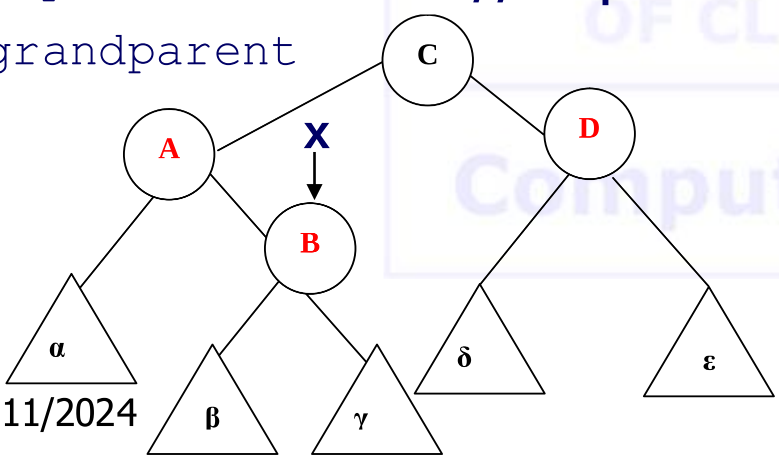
- **Insert** like in ANY other BST
  - As a LEAF, as for any other BST
- **Assign** it a color
  - RED
- Check the properties
- Re-balance if needed (**RB-INSERT-FIXUP** – check the textbook for the complete code)
- **P$_3$**: both children of a red node are black
- True for the children (NILs) of the inserted node
- Not true for the inserted node, in case its parent is RED colored
- Cases to analyze and remove inconsistencies

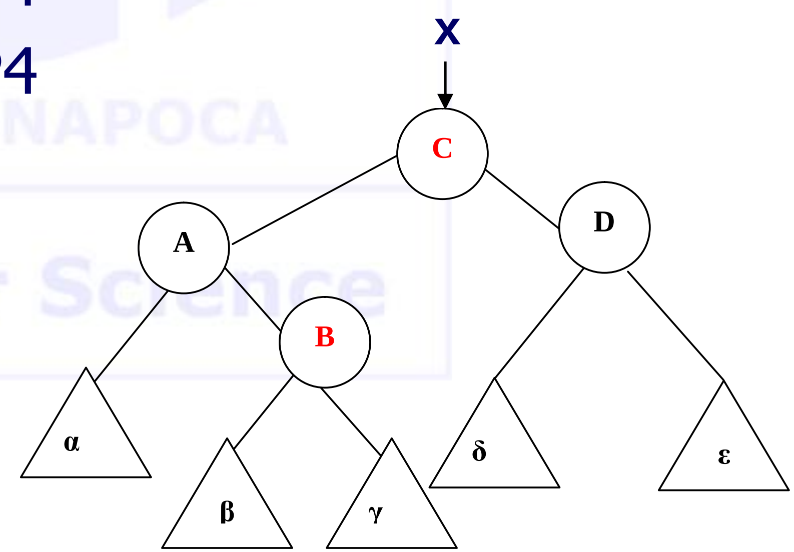
RB-insert- Case#1

- **B inserted node (pointed by x)**
- Parent (A)=RED, uncle (D)=RED, grandparent (C)=BLACK
- α, β, γ, δ, ε are RB trees (β, γ empty at first)

=> **Swap colors** between grandparent (C) and parent/uncle

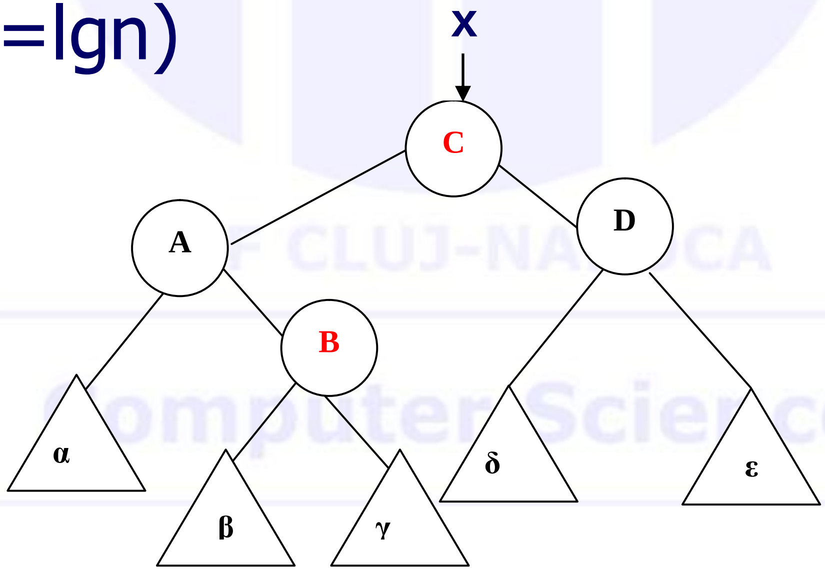
```
parent<-black          //no more P3 conflict A-B
uncle<-black           //to preserve P4
grandparent<-red       //to preserve P4
x<-grandparent
```





RB-insert- Case#1-eval

- P_3 may still be invalid, for the new **x** (i.e. C)
- Problem transferred 2 levels up in the tree (now β, γ not empty any longer)
- It takes (in the worst case) O(h) to rebalance (2lg(n+1)/2=lgn)



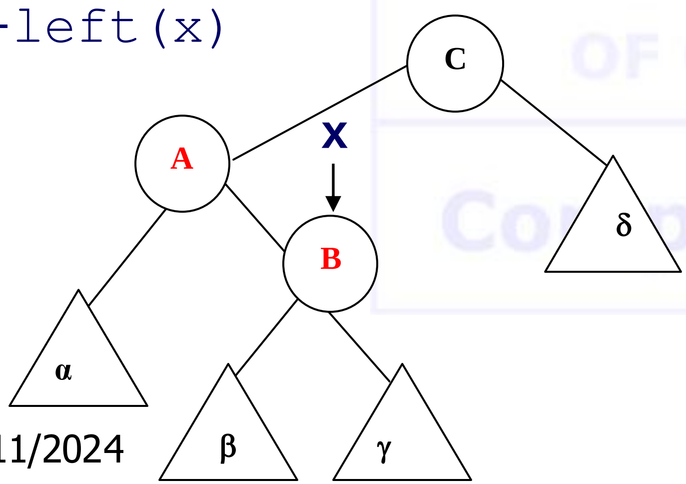
RB-insert- Case#2

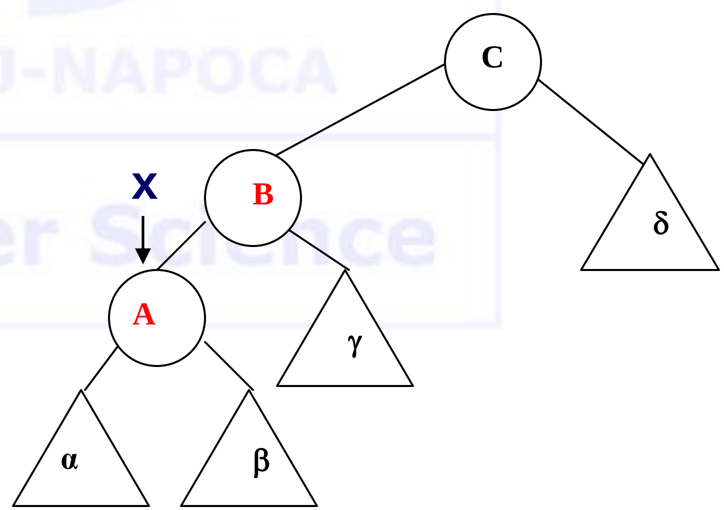
- **B inserted node (pointed by x)**
- Parent(A)=RED, uncle (δ's root)=BLACK (here is the difference compared to case #1), grandparent (C)=BLACK
- α, β, γ, δ are RB trees; δ's root is BLACK

=>

left_rotate(p(x)) //no more P3 conflict B-parent conflict

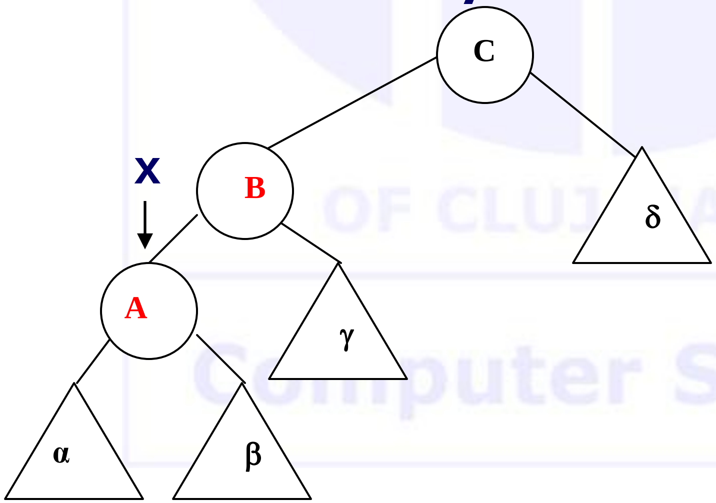
x<-left(x)





RB-insert- Case#2-eval

- Case #2 takes just O(1) to apply, but
- P_3 is still invalid, for the new **x** (i.e. A-B conflict)
- => it is followed by case #3

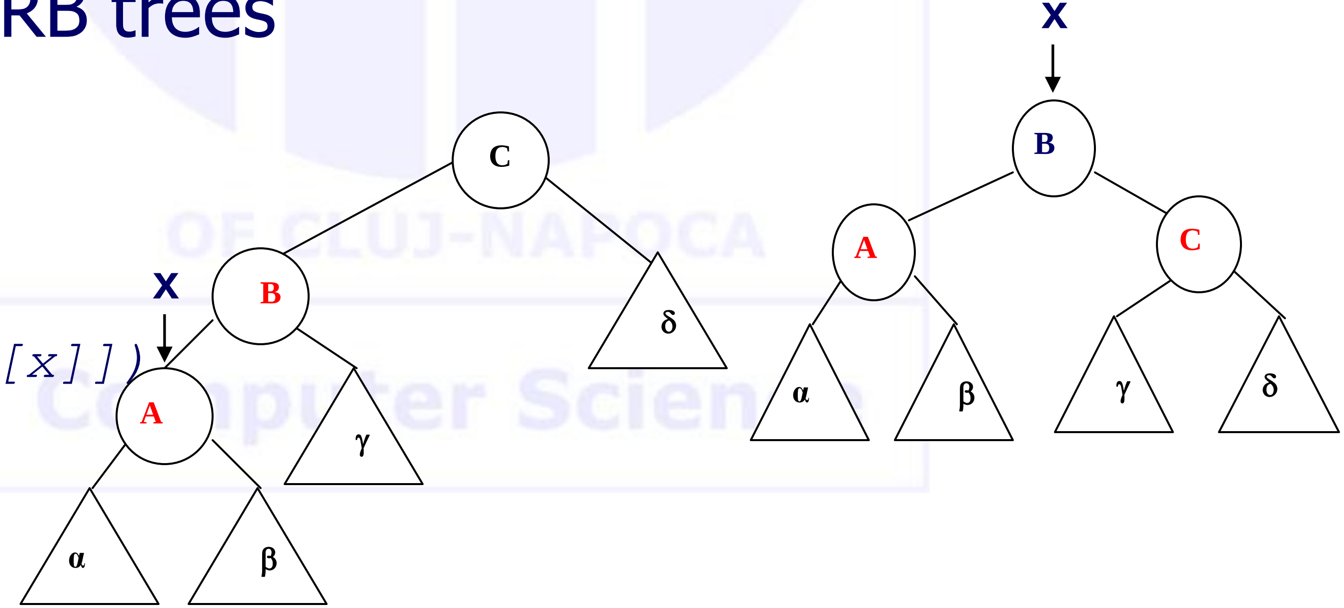


RB-insert- Case#3

- Either **Inserted A**, or coming from #2 (node pointed by **x**)
- Parent (B)=RED, uncle (γ's root)= BLACK, grandparent (C)=BLACK
- α, β, γ, δ are RB trees

=>

```
parent<-black
grandparent<-red
right_rotate(p[p[x]])
```



RB-insert- Case#3-eval

- Problem solved
- Each individual case takes O(1)
- Case #1 may repeat (up in the tree)
- Case #2 is followed by #3
- Case #3 solves the problem

RB-insert – Rebalancing eval

- Case #1 repeats up to the root O(h)
- Case #2+#3 => problem fixed O(1)
- Case #3 => problem fixed O(1)
- Insert O(lgn) + rebalancing
  - Worst case: #1 repeats O(lgn)
  - Best case: #3 => 1 rotation O(1)
  - Other cases: #2+3 =>2 rotations O(1)
- O(lgn) overall worst time (case 1 repeats), at most 2 rotations (case 2)

RB-delete

- Del as in regular BST + properties check to rebalance, if needed (RB-DELETE-FIXUP – check the textbook for the code)
- P4 (black height) is an issue

```
rb_delete(T,z)
   tree_delete(T,z)
   if color[y]=black
      then rb_del_fix(T,x)
```

z=node *to be removed* (see picture on the blackboard)

y=node *actually removed* (y≡z in case z has at most 1 child); info in y is placed in z's node

x=*y's only child before the delete process* takes place (could be nil, in case y has no children). After y is deleted, x becomes the child of y's parent (thus, x's parent could have now both children, one being x)

w=*x's brother (after delete operation takes place; it's y's brother before the deletion)*

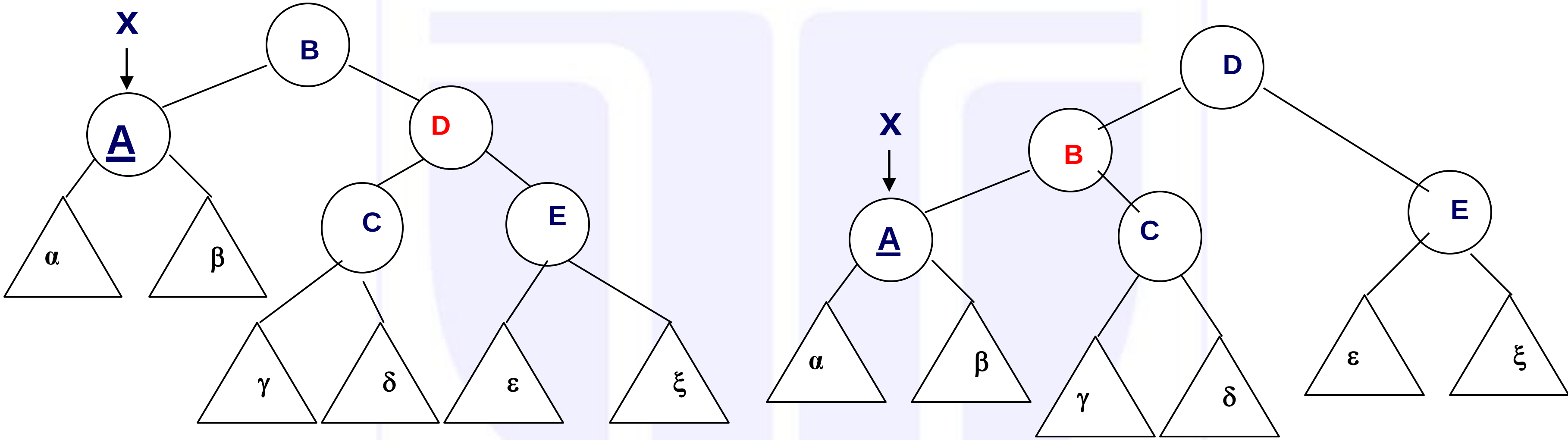
RB-delete

On x's branch check P4 property

x is y's (the removed node) only child

```
if color[x]=red
     then color[x]<-black
                //problem fixed; DONE!
                //x brings its former father color
     else color[x]<-double_black
                //P1 property issue!
```

RB-delete- Case#1

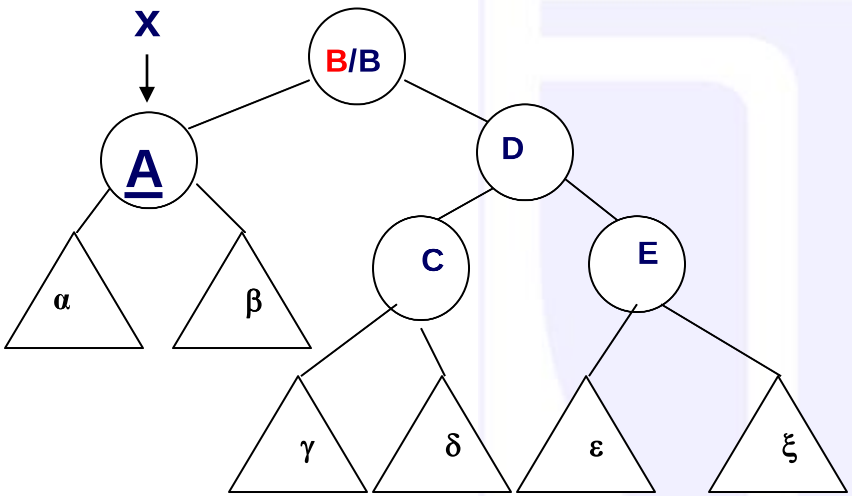


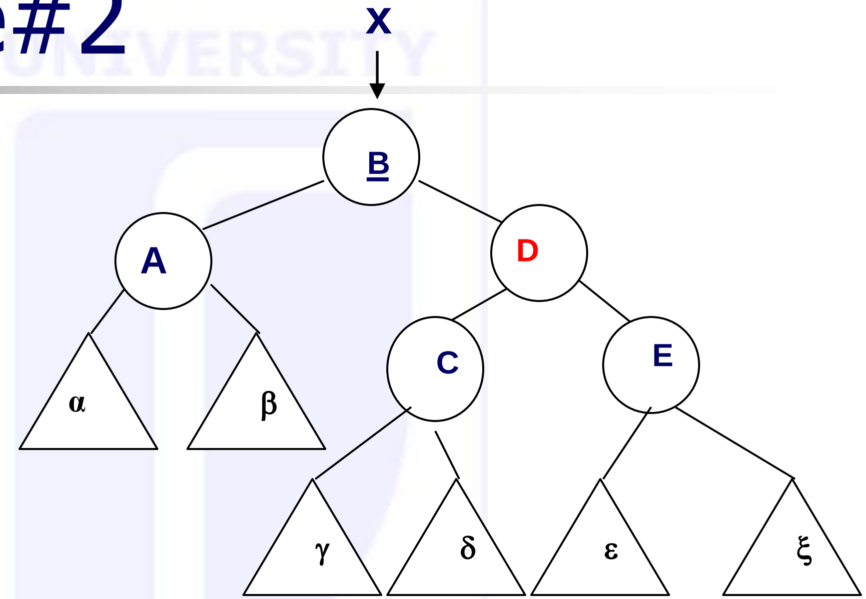
- Issue at node **A** (pointed by x) which is **double black**!
- A= was the only child of the deleted node
- **Parent (B) =Black, brother (D) =Red**
- α, β, γ, δ, ε, ξ are RB trees

=> B<->D color interchange +left rotate=> case 2 or 3 or 4

```
parent[x]<-red
brother[x]<-black    //colors interchanged
left_rotate(p[x])
```

RB-delete- Case#2

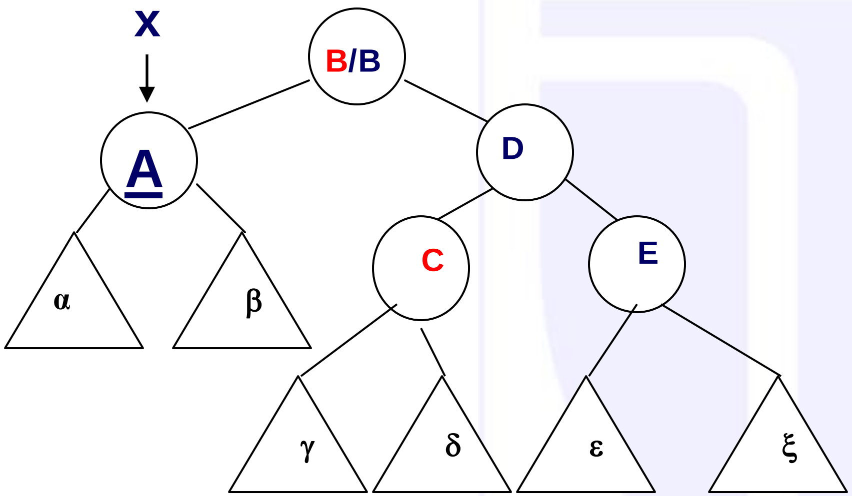


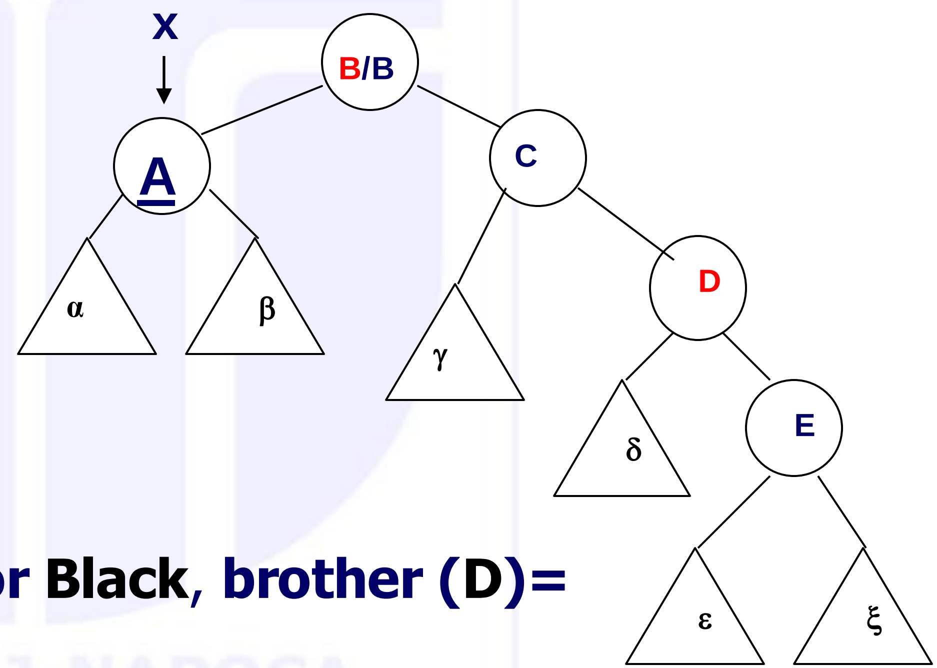


Parent (B)=Red (if after case #1) or Black, brother (D)=Black, node C is Black

```
brother[x]<-red
if p[x]=red
   then p[x]<-black          //when case#2 comes after case#1; problem solved
   else p[x]<-double_black
          x<-p[x]   //the same problem as at the beginning of case #2, just 1level above; case 2 repeats; in lgn problem solved
```

RB-delete- Case#3



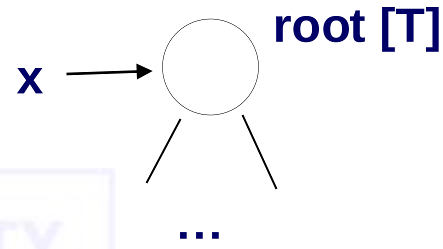


- **Parent (B)=Red (if after case #1) or Black, brother (D)= Black, node C is Red**
- α, β, γ, δ, ε, ξ are RB trees

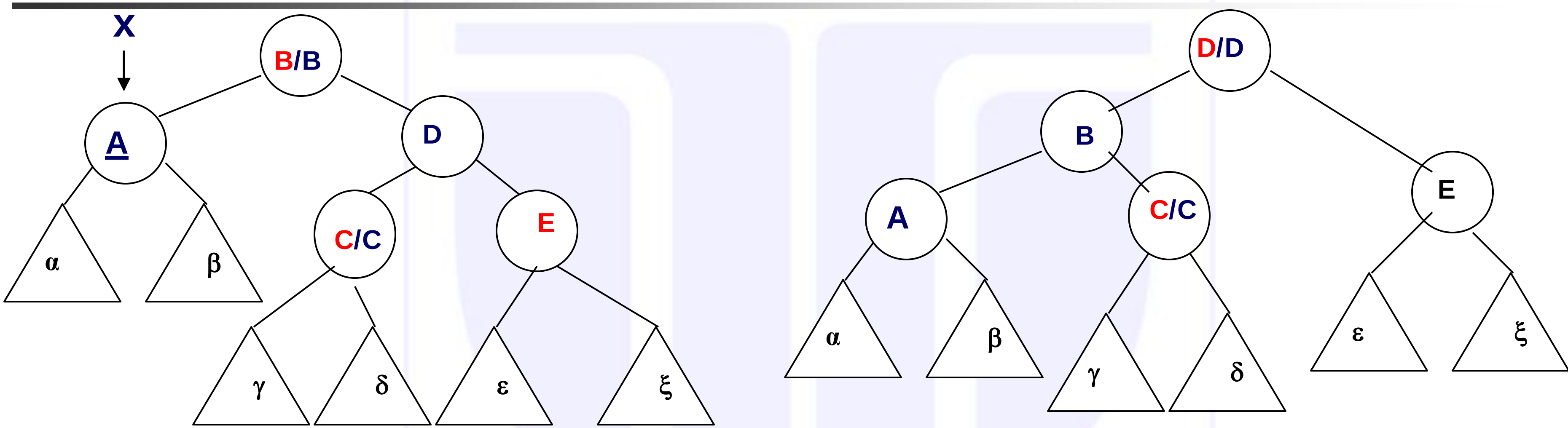
(A=child of the deleted node, double_black, pointed by x)

=> C<->D color interchange +right rotate => **case 4**

```
brother[x]<-red
left[brother[x]]<-black
right_rotate(brother[x])
```



RB-delete- Case#4



- **Parent (B)=Red (if after case #1) or Black, brother (D)= Black, node C is Red or Black, node E is Red**
- α, β, γ, δ, ε, ξ are RB trees

(A=child of the deleted node, double_black, pointed by x)

=> B<->D color interchange, E <– black + left rotate => problem solved

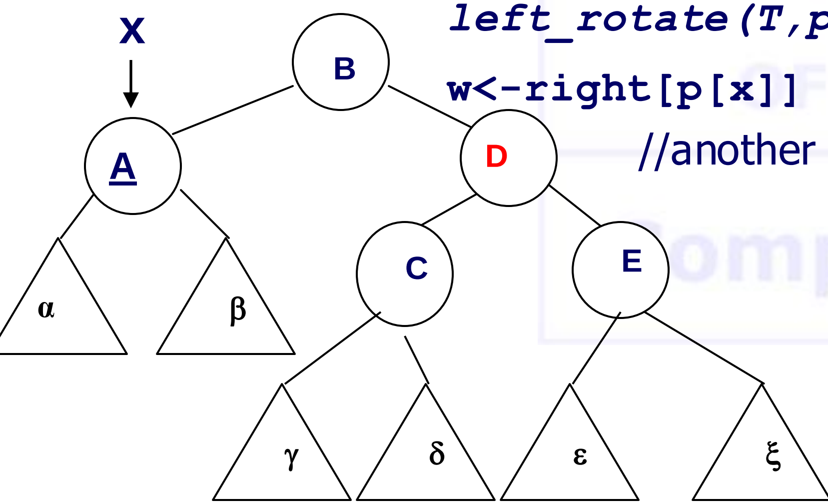
```
brother[x]<-color[parent[x]]
parent[x]<-black
right[brother[x]]<-black
left_rotate(p[x])      //1 more black node on x's branch
```

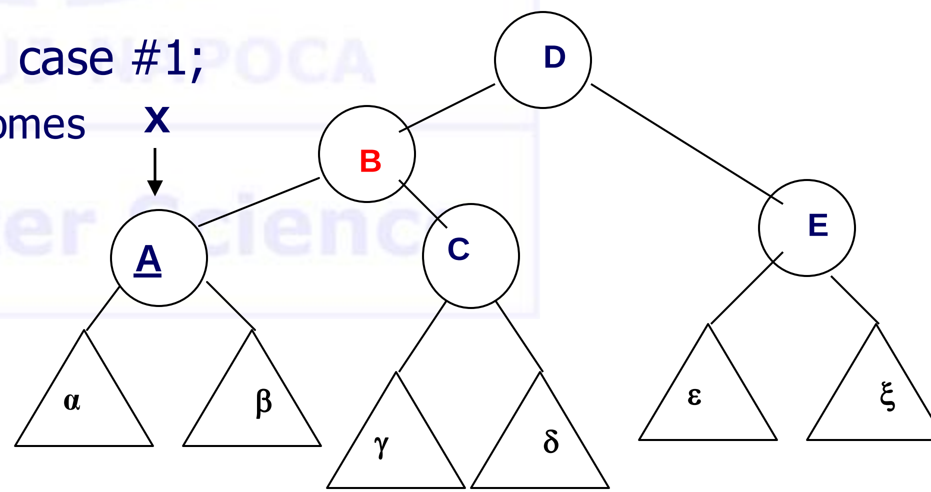
RB-del – Rebalancing eval

- Case #1 rotation followed by any other case
  - 1+2 => problem solved O(1)
  - 1+3+4=> problem solved O(1)
  - 1+4=> problem solved O(1)
- Case #2 (**no rotation,** only recoloring) repeats 1 level up in the tree
  - Worst case O(lgn)
  - Best case O(1)
- Case #3 rotation followed by case #4 O(1)
- Case #4 rotation; solves the problem O(1)
- Delete O(lgn) + rebalancing
  - Worst case: #2 repeats (recoloring only) O(lgn)
  - Best case: #4=> 1 rotation O(1)
  - Other cases: #1+2 or 1+ 3+4=>2 or 3 rotations O(1)
- O(lgn) overall, at most 3 rotations

RB-del - procedure

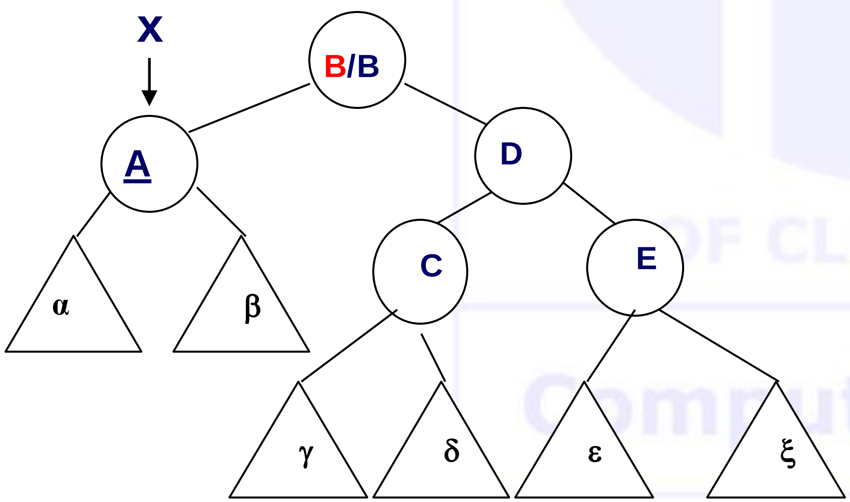
```
rb_del_fix(T,x)
while x<>root[T] and color[x]=black
do
    if x=left[p[x]]                         //cases on the left
    then                                    //else case symmetric on the right; not discussed
        w<-right[p[x]]                      //w=x's brother!
        if color[w]=red
        then    color[w]<-black                    //case #1 APPLY ; coloring+rotation
                color[p[x]]<-red
                left_rotate(T,p[x])
                w<-right[p[x]]  //end case #1;
                           //another case comes
```

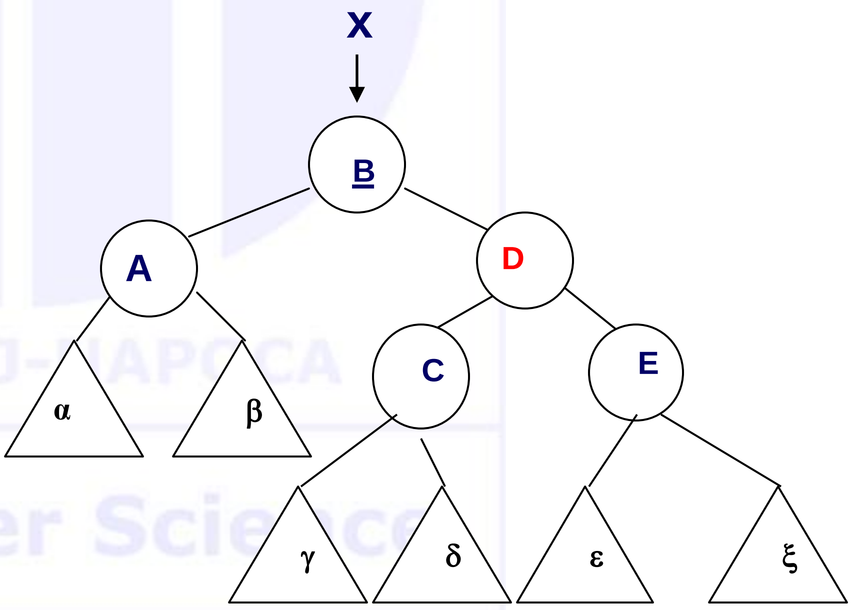




RB-del - procedure

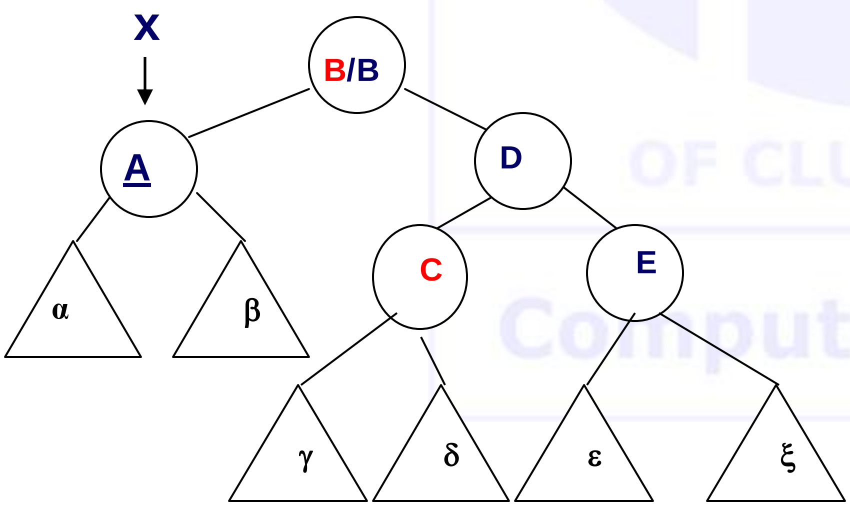
```
if color[left[w]]=black and color[right[w]]=black
    then                                //case #2
            color[w]<-red
            x<-p[x]
    else
```

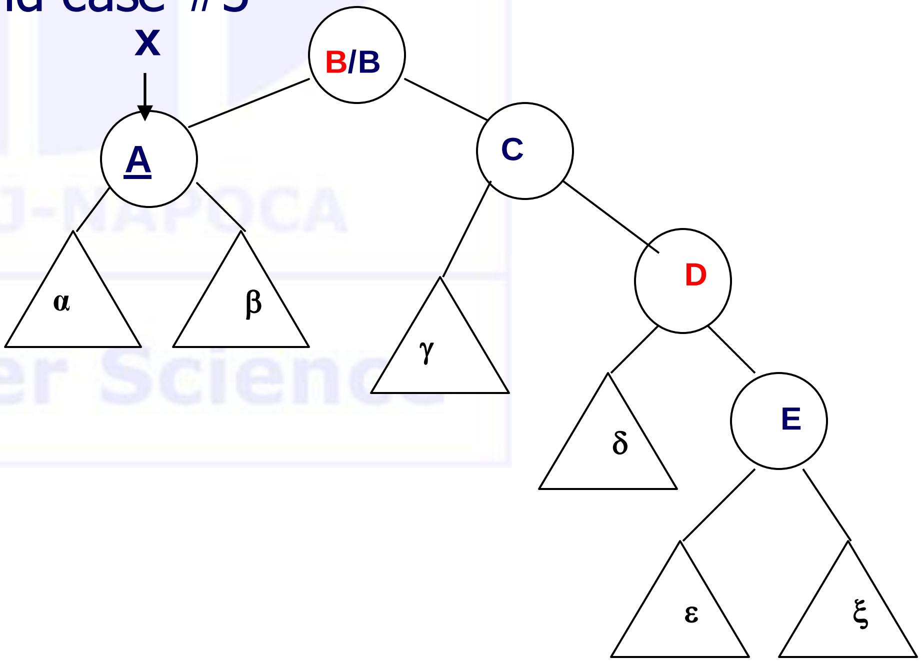




RB-del – procedure - cont

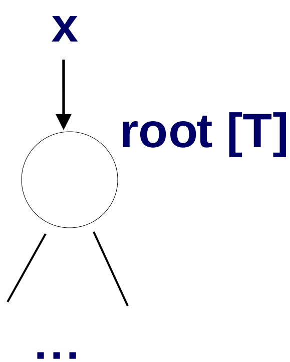
```
else                //color[left[w]]≠ black or color[right[w]] ≠black
    if color[right[w]]=black  //E is black
    then                  //case #3
        color[left[w]]<-black
        color[w]<-red
        right_rotate(T,w)
        w<-right[p[x]]  //end case #3
```

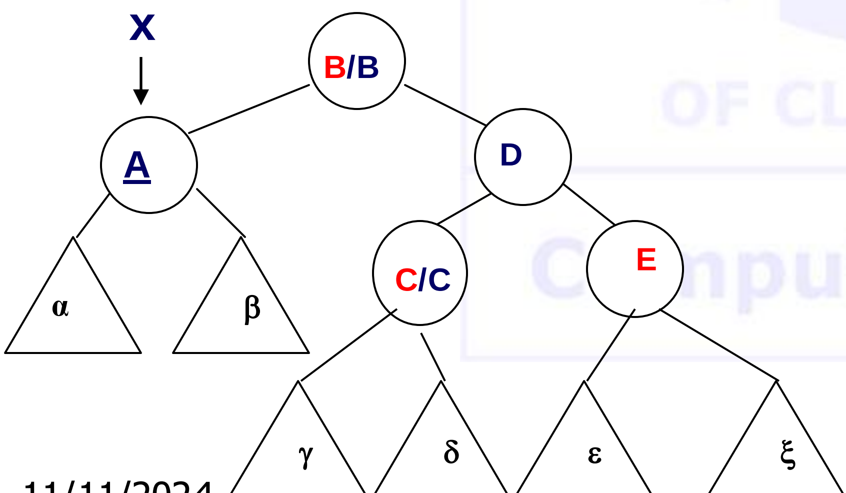


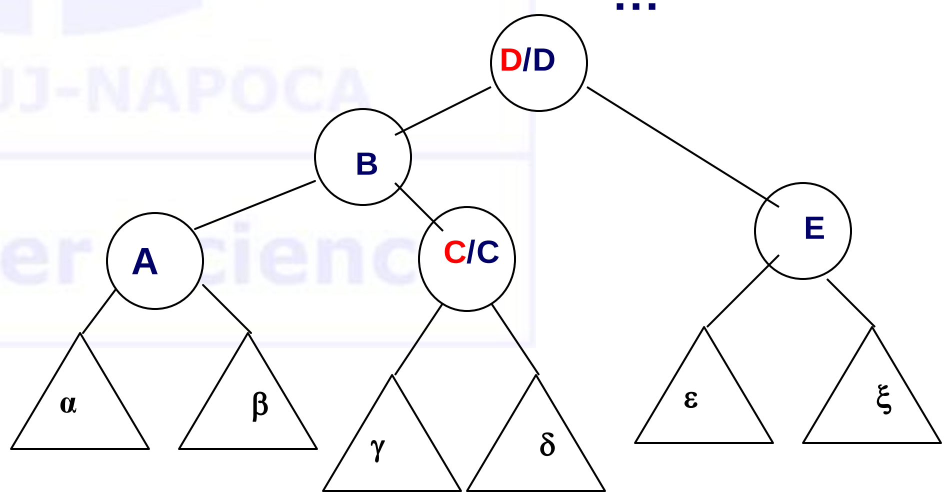


RB-del – procedure - cont

```
        color[w]<-color[p[x]]          //case #4
        color[p[x]]<-black
        color[right[[w]]<-black
        left_rotate(T,p[x])
        x<-root[T]
else    …        //x=right[p[x], all 4 cases symmetric to the right
color[x]<-black
```







Conclusions on balanced search trees

Tree	Height	Ins	Del
BST	[lgn, n]	O(h)	O(h)
RBT	[lgn, 2lgn]	2 rot	3 rot
AVL	[lgn, 1.45lgn]	1 rot	lgn rot
PBT	lgn	n rot	n rot

For RBT, at most lgn/2 color updates needed