Fundamental Algorithms

Lecture #7&8

Cluj-Napoca

November 20, 2019

Agenda Lecture 7

- **MT**
- **Augmented Trees (type 2) – check Lecture 6 for notes**

Agenda Lecture 8

- Red-Black Trees
  - Insert
  - Delete
- Balanced Trees
  - Comparative analysis and conclusions
- Disjoint Sets
  - Representation
  - Basic Operations
  - Implementation

Red-Black trees

- Balanced trees
- Both insert/delete operatiuons take O(lgn), with constant time for rebalancing

Def: 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 any node to a leaf has the **same number of black nodes**

Red-Black trees

Th: A RB tree with n **internal** nodes (without leaves = nil nodes) has its **height at most 2lg(n+1)**

Proof: notation: 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) $2^0-1=1$

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

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

OR bh(x)-1 (if x is black)

nb of internal nodes of x= nb of internal nodes of children(x) +1 (itself)=>at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1= 2^{bh(x)}-1$ qed (end of step1)

Red-Black trees

Step 2:

We know P(bh) is true, ie

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

$$\Rightarrow n \geq 2^{bh(x)}-1 \quad \text{(from (1))}$$

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

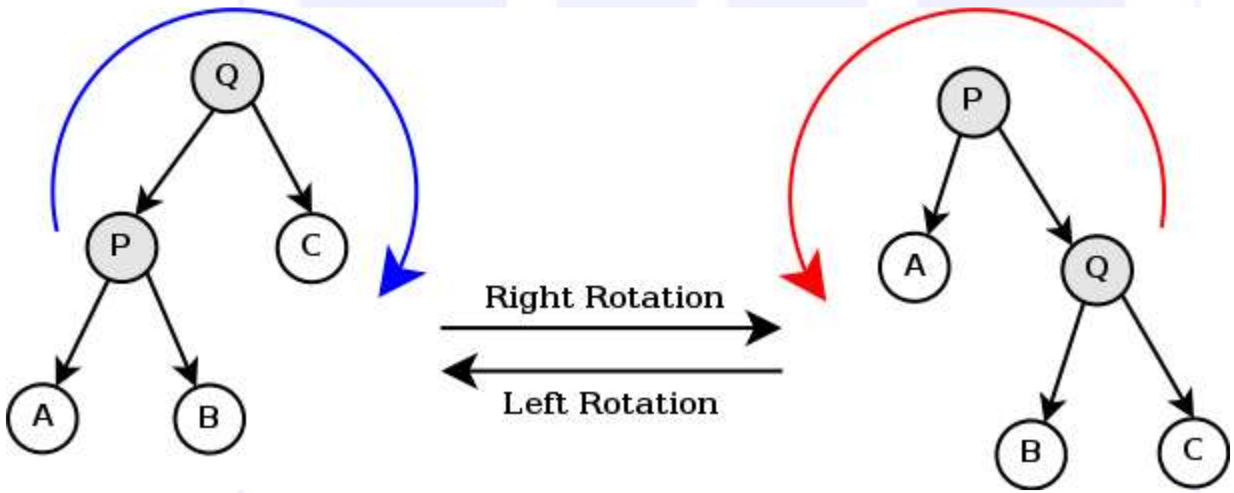
$$n \geq 2^{h/2}-1 \Leftrightarrow n+1 \geq 2^{h/2} \Leftrightarrow h/2 \leq \lg(n+1) \Leftrightarrow h \leq 2\lg(n+1)$$

(qed, end of Th proof)

Red-Black trees - rotations

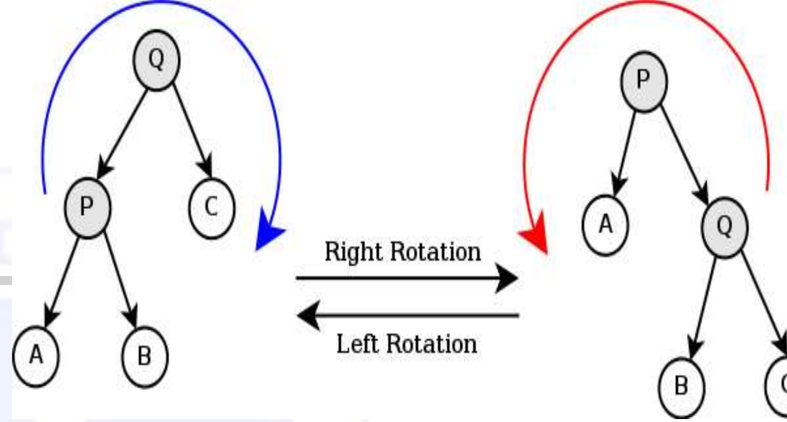
Similar to single rotations right/left from AVL

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 (WHY?)
- 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 always true; in case the parent of the inserted node is RED colored => red (parent)-red (inserted)
  - Cases to analyze and remove inconsistencies

RB-insert- Case#1

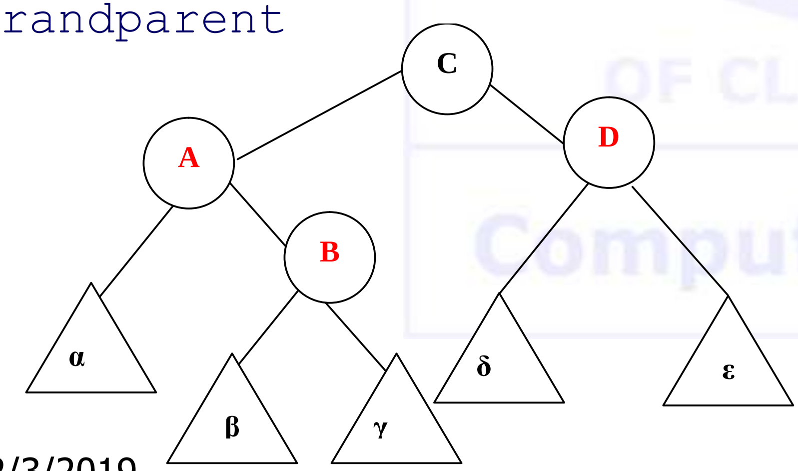
- **B inserted node (pointed by x)**
- Parent (A)=RED, uncle (D)=Red, grandparent (C)=BLACK
- α, β, γ, δ, ε are RB trees (β, γ empty at first)
- Swap colors between grandparent (C) and parent (A)/uncle (D)

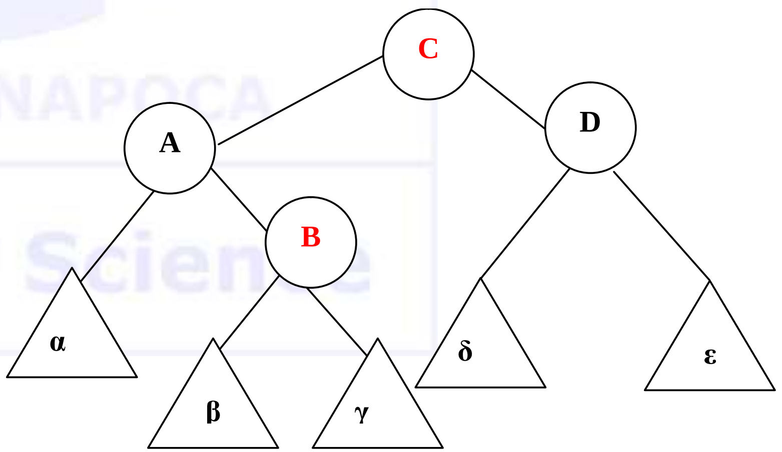
```
parent<-black
```
//no more P3 conflict A-B

```
uncle<-black
```
//to preserve P4

```
grandparent<-red
```
//to preserve P4

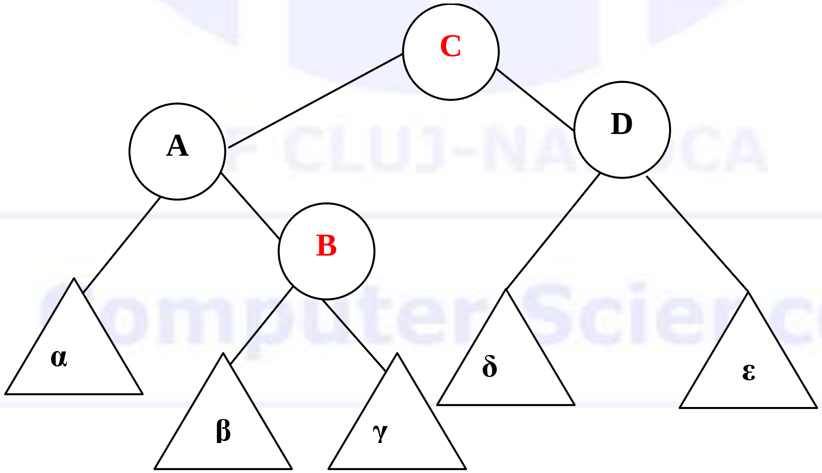
```
x<-grandparent
```





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 = all way up to the root) O(h) to rebalance (2lg(n+1)/2=lgn)

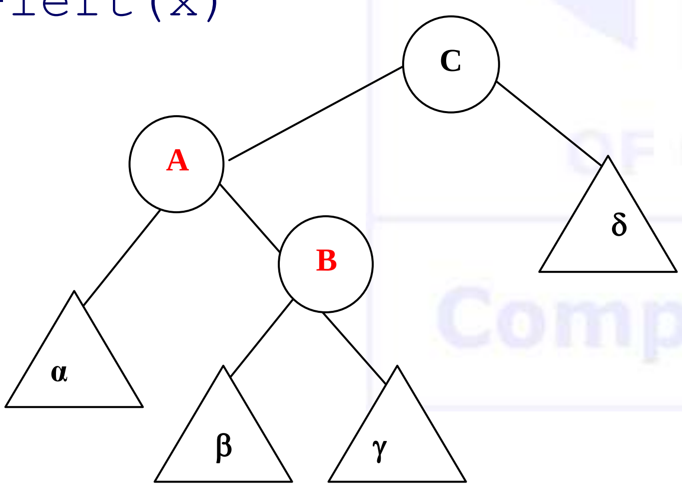


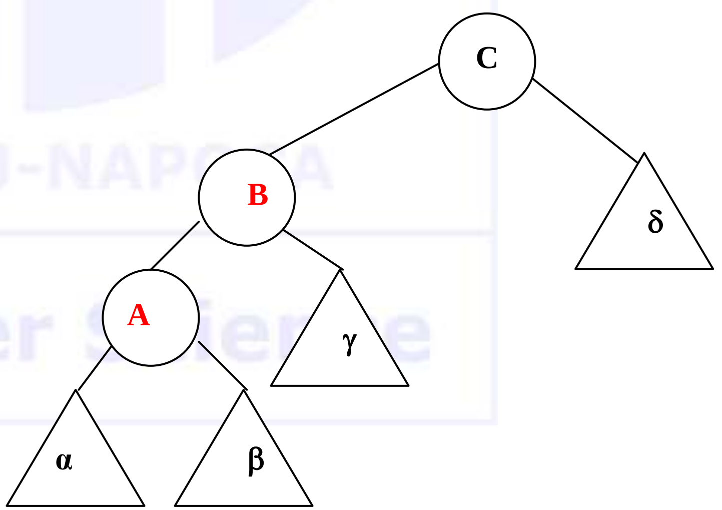
RB-insert- Case#2

- **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

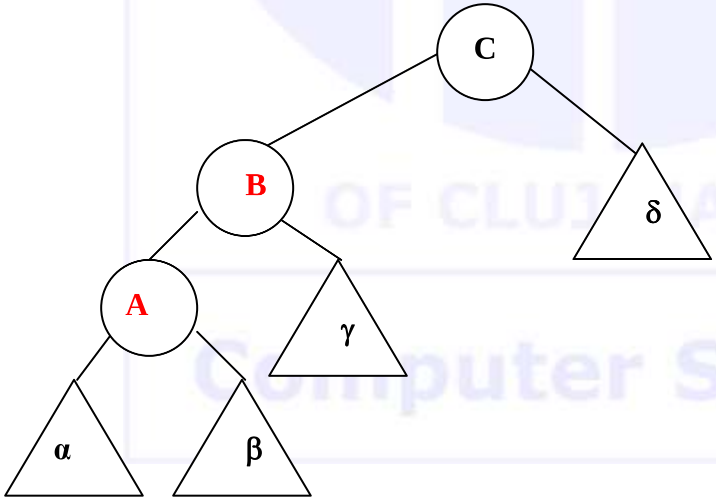
```
x<-left(x)
```





RB-insert- Case#2-eval

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



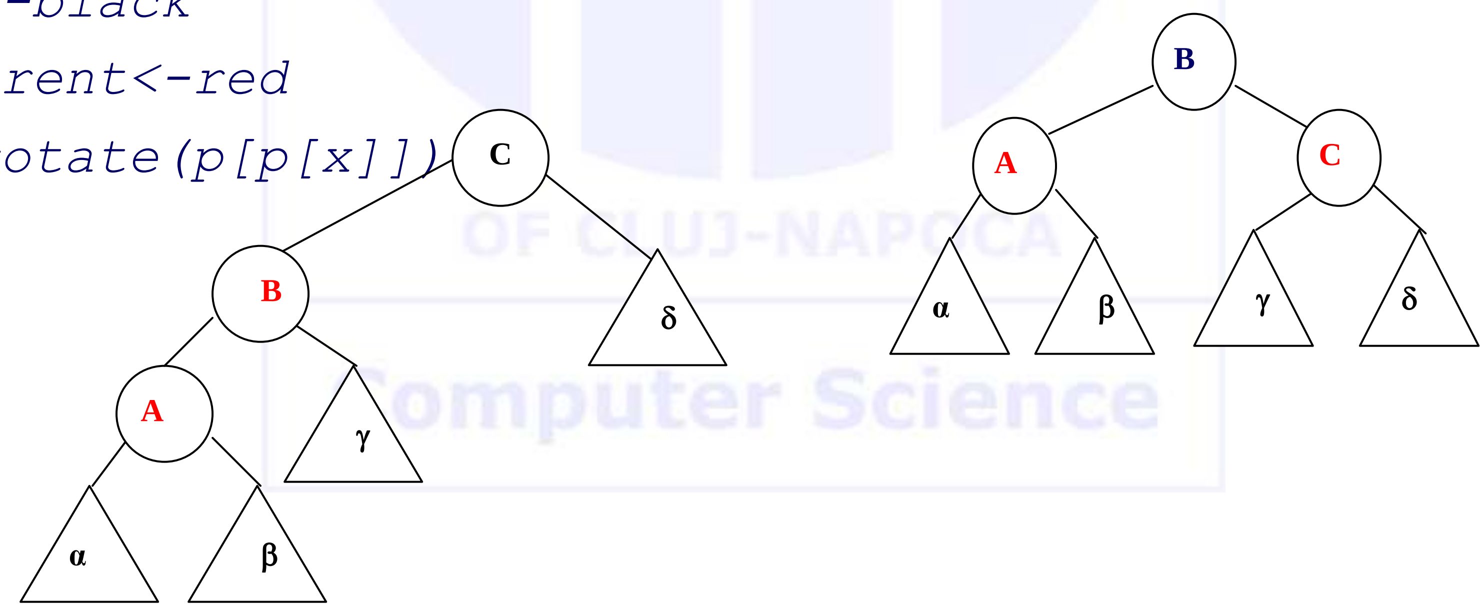
RB-insert- Case#3

- **Inserted A** /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 case: #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)
```
//else NO issue at all

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

y=node actually removed (y≡z in case z has at most 1 child); y's info 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 new brother (after delete operation takes place; it's y's brother before the deletion)

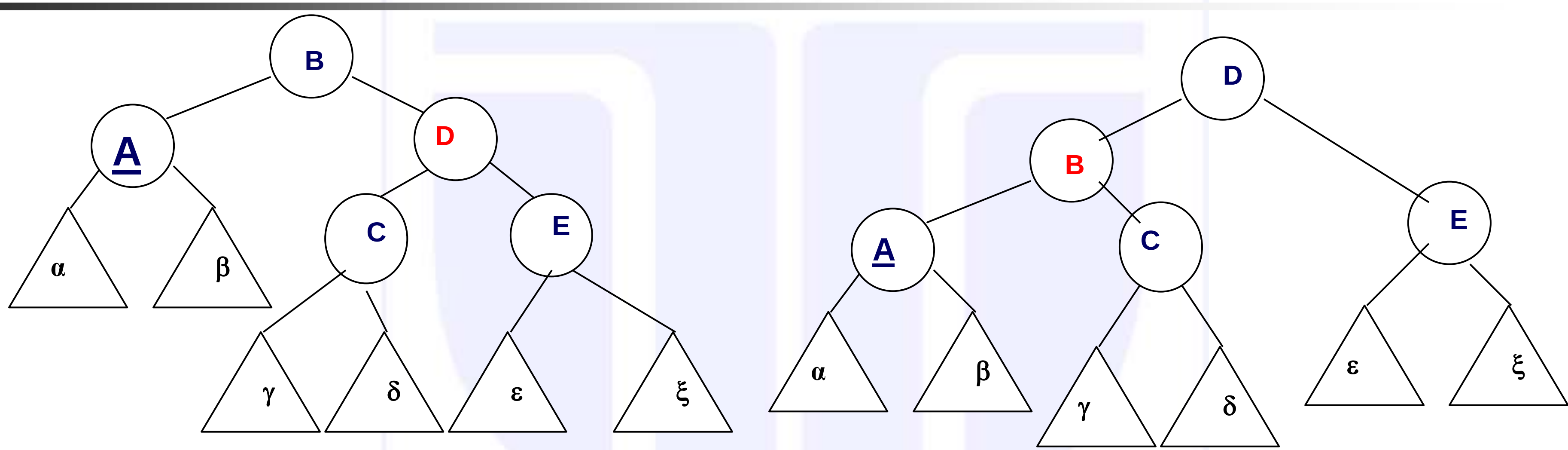
RB-delete

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
                    //does not exist something like this
                    //brings one additional black
                    //to keep the black height property (P4)
                    //yet, a P1 property issue!
```

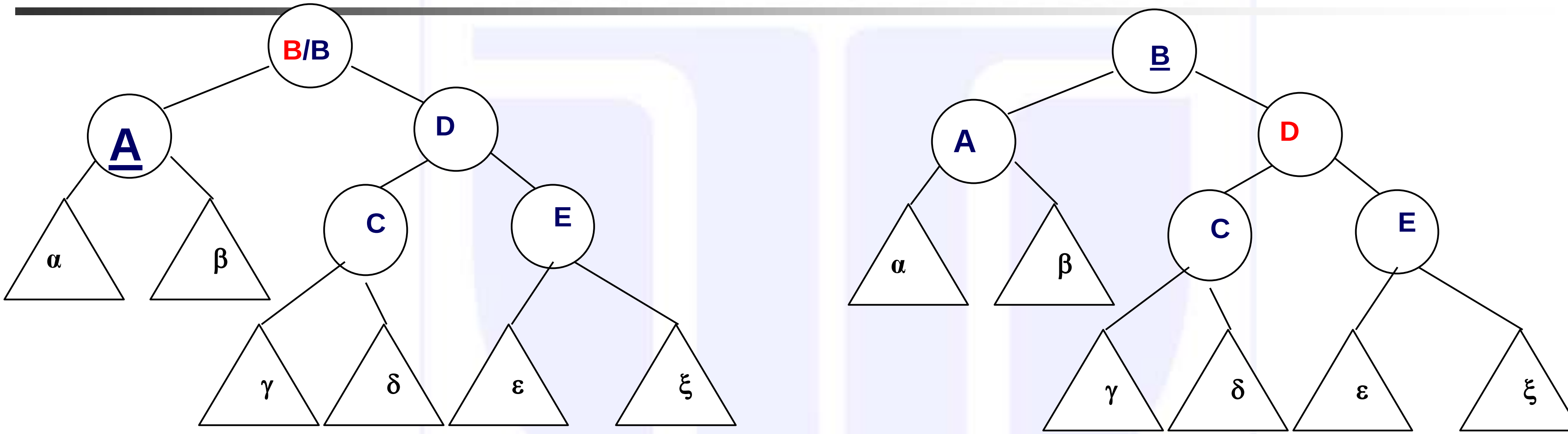
RB-delete- Case#1



- 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 follows

```
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 (stand alone case #2), brother (D)=Black, node C = Black

`brother[x]<-red`

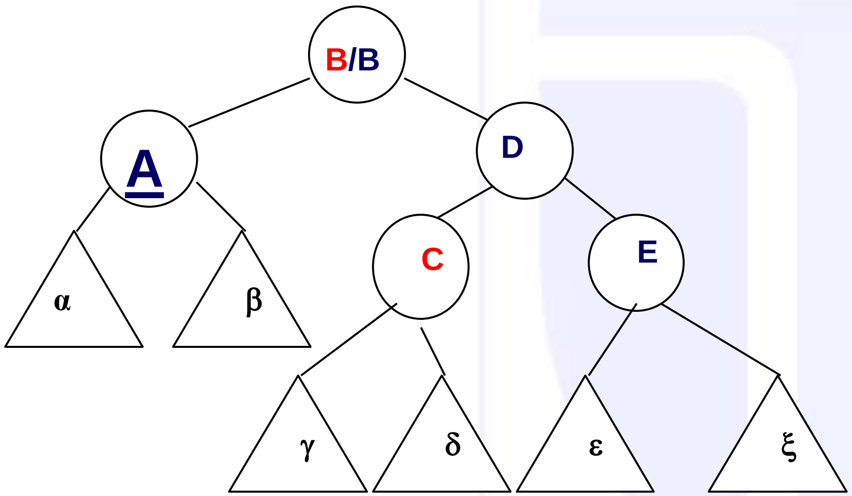
`if p[x]=red`

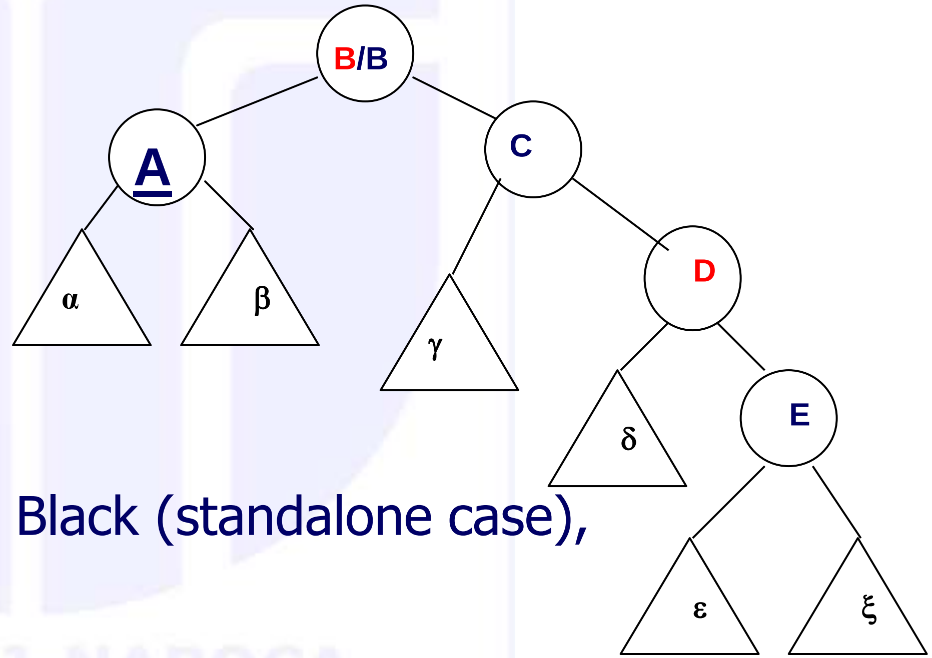
  `then p[x]<-black` //when case#2 comes after case#1; problem solved

  `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

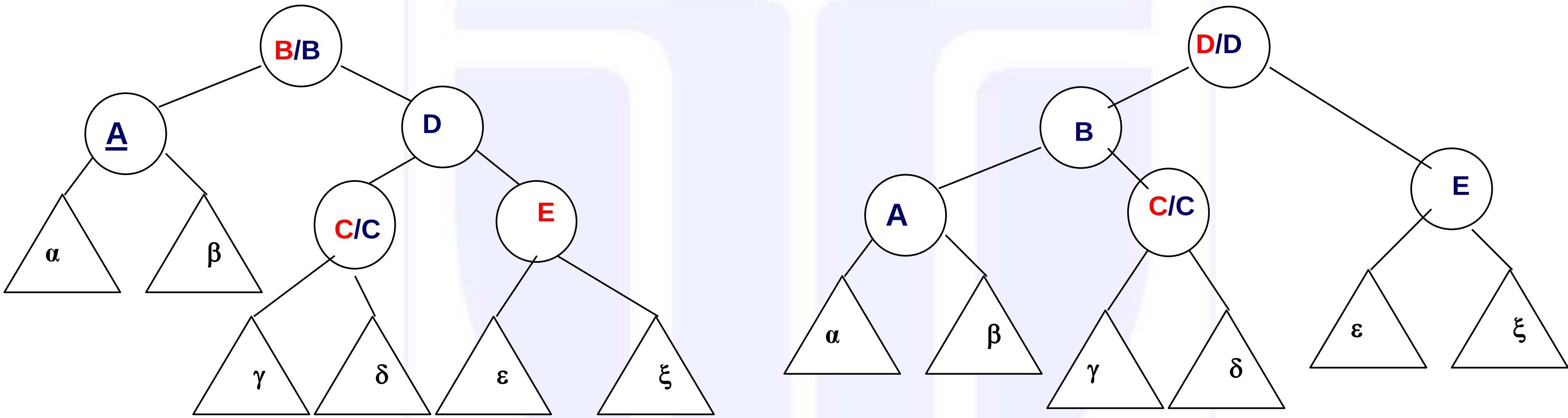




- Parent (B)=Red (if after case #1) or Black (standalone case), brother (D)=Black, node C = Red
- α, β, γ, δ, ε, ξ are RB trees
- A=child of the deleted node (double Black <u>A</u> (pointed by x))
- C<->D color interchange +right rotate => **case 4**

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

RB-delete- Case#4



- Parent=Red (if after case #1) or Black,

brother=Black, Node C is either Red or Black; node E = Red

- α, β, γ, δ, ε, ξ are RB trees
- A=child of the deleted node (double Black **A** (pointed by x))
- B<->D color interchange +left rotate => problem solved

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

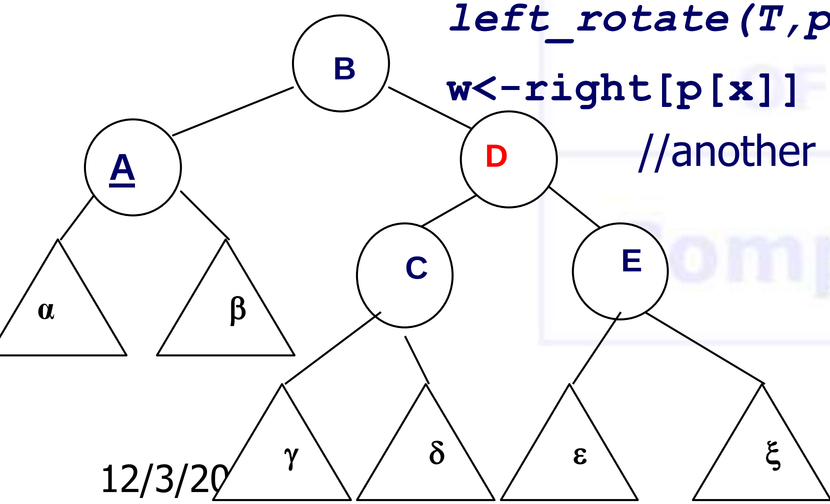
RB-del – Rebalancing eval

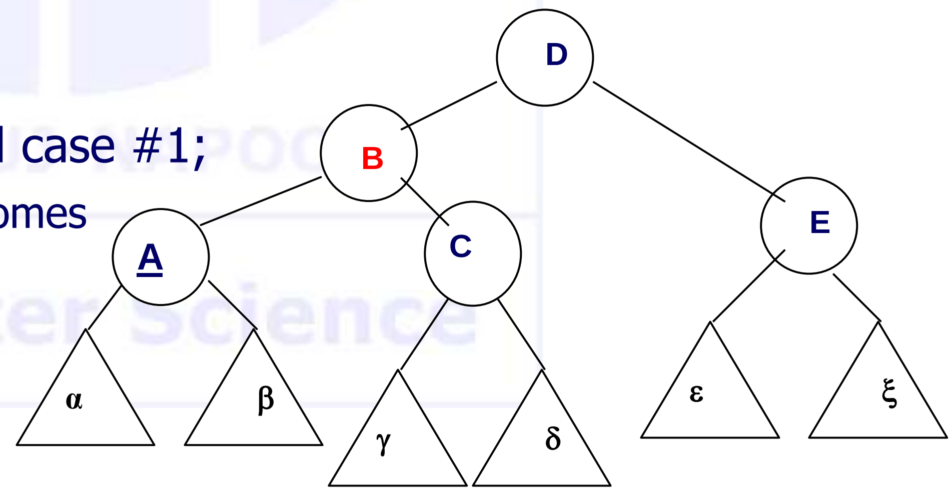
- Case #1 rotation followed by any other case
  - 1+2 => problem solved O(1)
  - 1+3+4=> problem solved O(1)
  - 1+4=> problem solved O(1)
- Case #2 **no rotation,** only recoloring - repeats 1 level up in the tree
  - Worst case O(lgn)
  - Best case O(1)
- Case #3 rotation followed by case #4 O(1)
-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

RB-del - procedure

rb_del_fix(T,x)

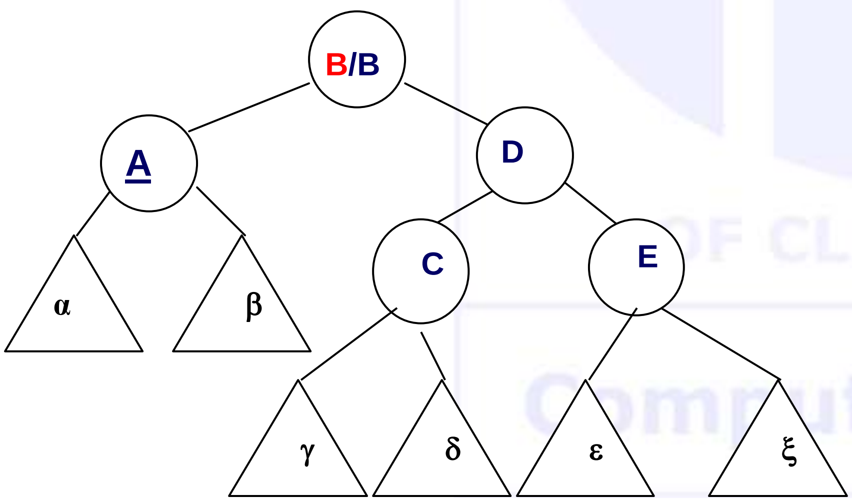
```
while x<>root[T] and color[x]=black
do
    if x=left[p[x]]                  //cases on the left
    then                             //else case symmetric on the right; not discussed
        w<-right[p[x]]               //w=x's brother
        if color[w]=red
        then    color[w]<-black     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

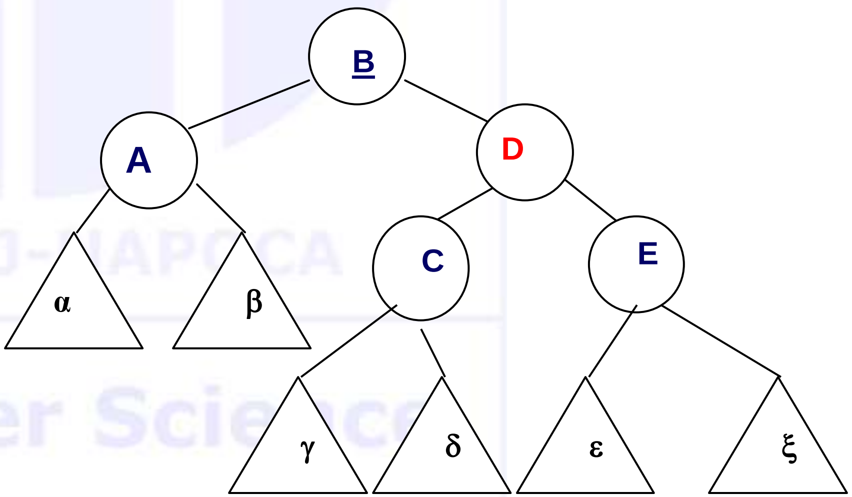




RB-del - procedure

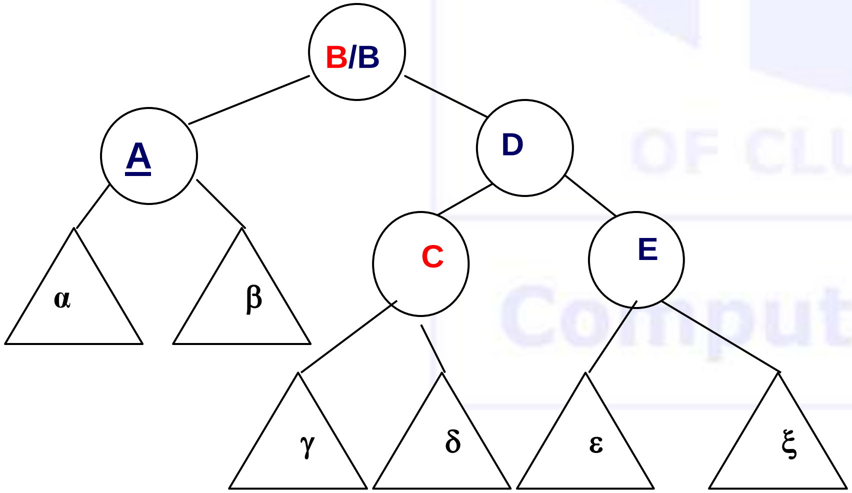
```
if color[left[w]]=black and color[right[w]]=black
    then                                         //case #2
            color[w]<-red
            x<-p[x]
    else
```

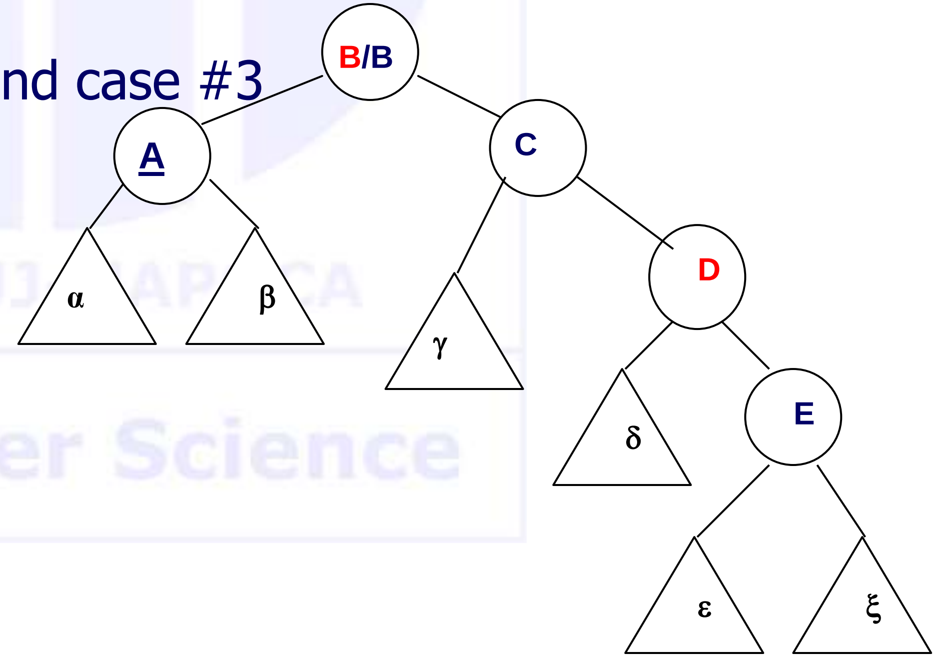




RB-del – procedure - cont

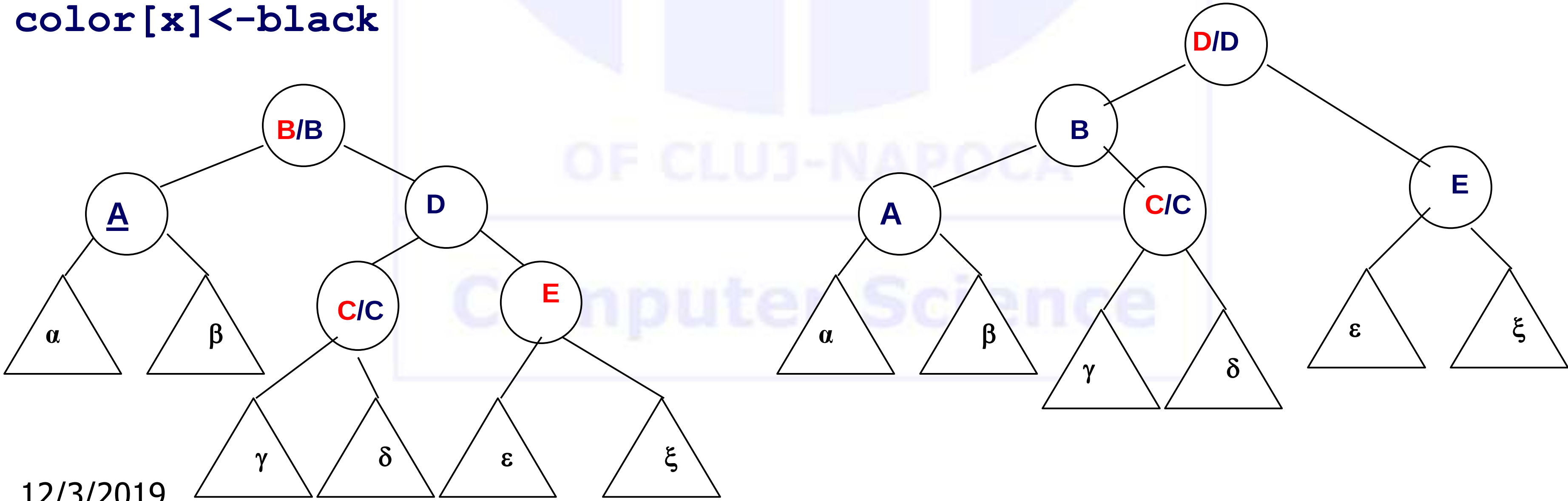
```
else              //color[left[w]]≠ black or color[right[w]] ≠black
    if color[right[w]]=black
    then                  //case #3
          color[left[w]]<-black
          color[w]<-red
          right_rotate(T,w)
          w<-right[p[x]]  //end case #3
```





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

Disjoint Sets

- Collection of dynamic DS $S=\{S_1, \ldots, S_k\}$
- $\exists$ n elements (objects) in all k sets ($n \geq k$)
- each set S_i is identified by its *representative* element, $x \in S_i$;
- Basic operations:
  - **Build-Set** (x)

    Generates a new set, with a single element => n sets initially, each object has its own set, and it is its own representative el.
  - **Unify** (x, y)

    joins 2 disjoint sets, represented by x $\in$and y; builds $S_x \cup S_y$ (and destroys S_x and S_y); the representative becomes any of the 2 representatives;
  - **Find-Set** (x)

    Returns a pointer to the representative element of the set containing element x.

Disjoint Sets – contd.

n = nb. of objects in the whole S

m = total nb. of operations (Build-Set, Unify, Find-Set)

m>=n (as we have n Build-Set operations)

Utility/Applications:

- speeds up execution when we need to find/group items with similar features
- graphs (connected components; MST)
- many other

Disjoint Sets - implementation

- LL
- A set = a **linked list**
- representative= the first element (head) of the list
- An object in such a list contains
  - The element from the set;
  - The pointer to the next element in the list (LL)
  - Pointer to the representative (ex: blackboard)
- Build-Set (builds a list with a single element) O(1)
- Find-Set (returns the representative) O(1)
- Unify (x, y) – adds x's list at the end of y's list;
  - representative = former y's representative
  - all x's elements have to update representative pointer (ex: blackboard)

Disjoint Sets – implementation – contd.

- Worst case: $O(m^2)$ for all operations
- n Build-Set (1 for each element)
- Unify
  - n times (to get to a single set)
  - $1 + 2 + 3 + \ldots +2 = O(n^2)$ (show on the blackboard)
- n~m (actually m>n, yet n is linear in m)
- On average, O(m) for a call of Unify, m calls, $O(m^2)$

Disjoint Sets – implementation increase efficiency

- Update pointers for the shorter lists
- Keep as knowledge their length (similar to Order Statistic Trees)
- **Theorem**: For n objects in LL with weighted unify, for m Build-Set, Find-Set and Unify takes O(m + nlgn)
- **Proof**: (check the textbook – identify an **informal** justification)

Forest of Disjoint Sets

- Set = **tree** with root; keep parent pointer
- 1 node = 1 element (=1 obj) from the set
- 1 tree = a set
- The root = *representative* el.
- Basic Implem. ~ to lists (no improvement)
  - Build-Set (x) build the tree with root only
  - Find-Set (x) goes up and return the representative
  - Unify (x, y) Ex: (blackboard)

Forest of Disjoint Sets – Heuristics
(to increase performance)

- Unify based on **rank**
  - Similar to weighted unify on lists
  - The tree with less nodes will point to the tree with many nodes
  - Info kept at root level = rank = max height of the tree
  - rank ≅ lg (dim) (is an approximation, not an exact value; a guarantee that value is never exceeded)
- Tree shrink
  - Within the **Find-Set**, each node on the *search path* will update the parent node to the *representative* (instead of parent), and leave the **rank unchanged**!
  - **Shrink does NOT change rank!** Why? Ex: blackboard

rank ≅ lg (dim) It is an approximation, ONLY

Forest of Disjoint Sets – Heuristics

- Rank[x]

  = max height of the subtree rooted by x

  = nb. of edges on the longest path from x to a leaf

  rank[leaf] = 0
- Find-Set leave ranks unchanged

Forest of Disjoint Sets - Implementation

Build-Set (x)

```
p[x] <- x
rank[x]<-0
```

Reunion (x, y)

```
Unify (Find-Set(x), Find-Set(y))
```

Unify (x, y)

```
if rank [x] > rank [y]
  then p[y] <-x
  else p[x] <-y
if rank [x] = rank [y]
  then rank [y]= rank [y]+1
```