Curs #5

1. Hash Tables
2. Trees
   - 2.1 Binary and multiway
   - 2.2. Representation
   - 2.3. Basic operations

1. Hash Tables

- stochează seturi dinamice de date pentru acces rapid
  ↳ pot varia
- frequent op = search
- DS ce stochează seturi de itemi după Keys
  ~ Operations: ⇒ insert
  ⇒ delete
  ⇒ search
  - goal: O(1) for all

Direct access table

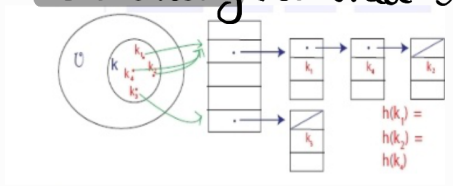
Limitations: Keys ∈ U
large Key range → large storage space
↳ Solution: project universe U of all Keys onto a table of size m

- m – size of the table
- |U| – nb of possible Keys
- n – nb of Keys stored in the HT
- h that compute the Key loc in HT

obs $h(Key_1) = h(Key_2)$ ⇒ collision

H.T. – Collision Resolution

- chaining (linked lists) (closed addressing)
  ↳ Hash functions
  ↳ Universal hash



- Open addressing (in table)
  ↳ linear probing
  ↳ Quadratic "
  ↳ Double hashing

I) Chaining

- h(K) = dispersion value ~ cântarea Trebuie să parcurgă toată lista
- Load factor $\alpha = \frac{n}{m}$ = avg nb of keys/slots
- expected performance of chaining assuming uniform hash $O(1+\alpha)$

WORST CASE: all Keys in the same slot ⇒ $O(n)$

(A) Hash Functions

↳ Division method: $h(K) = K \bmod m$
↳ collision for $K_1 = K_2 \bmod m$ / m divides $|K_1 - K_2|$
↳ good practice m – nr prim care NU e apropiat de o putere a lui 2 sau 10

→ DRAWBACKS: ⇒ prime numbers, if they need to be large
⇒ consecutive keys map to consecutive hash vals.
→ division is slow

↳ **Multiplication method**: $h(K) = m \{KA\}$, $A = \alpha$
$= m(KA - \lfloor KA \rfloor)$

↳ good practice
- $m = 2^p$ for some integers p, so that m fits a sg word
- $h(K) \rightarrow$ easy to calculate

↳ $A = \frac{\sqrt{5}-1}{2}$ val potrivită pentru A.

↳ malicious Keys = all Keys in same loc. $\Rightarrow O(n)$

(B) Universal hash

↳ Randomly → select fct at execution time from a set of functions

THEOREM

↳ if $(n \leq m)$ avg. nb of collisions per Key < 1

if a class of Universal hash functions is considered

II Open adressing $\Rightarrow m \geq n$ MEREU

- seq to try for a key: $(h(K,0), 1, \dots, m-1)$
- ideal: secv. ar trb să fie o permutare pt $\langle 0, 1, \dots, m-1 \rangle$

Linear probing $h(K,i) = (h'(K) + i) \% m$

$h'(K) \Rightarrow$ fct. de Hash

↳ DRAWBACK: clustering = consecutive group of filled slots grows $\Rightarrow$ average search time grows

↗ 2 chei sunt mapate la aceeași adresă

Quadratic probing: $h(K,i) = (h'(K) + c_1 i + c_2 i^2) \% m$

↳ DRAWBACK: Secondary clustering = cheile mapate inițial la aceeași adresă, pot concura pt aceleași locații în iterații consecutive

Double hashing: $h(K,i) = (h'(K) + i\, h''(K)) \bmod m$

obs: $h''(K)$ ar trebui să fie relativ primă pt m

Opening addressing Evaluate

- $\alpha < n/m < 1$, α = load factor

observație
la chaining
$\alpha = \frac{n}{m}$
- pt op. add
α + cel mult n/m și e subunitar

- **Theorem 1**

unsuccessful search time $= \frac{1}{1-\alpha}$

- **Theorem 2**

successful search time $= \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$

Important

II. Trees

= dynamic structures

- target: faster (than LL)
  maintain good running time for other op.

- **Basic operations**

TRAVERSAL	$O(n)$
SEARCH	$O(n)$ regular, $O(h)$ BST
INSERT	$O(h)$
REMOVE	$O(n)$ regular, $O(h)$ BST
Find MIN/MAX/PRED/SUCC	$O(n)$ doar în BST

- **Binary trees representation** → dyamic linked list structures

↳ type of nodes { root; internal = non root, non leaf; leaves = both childrens nil

Multi Way trees

- mai mult de 2 copii
- rep as: • tree with exactly m childrens (array of m potential children)
  - a tree with just one child (linked list)
  - a bin tree → left link = first child
    → right link = brother [next child of parent node → right links = = SLL of brothers]

BST - Binary Search Tree

- any subtree of a BST is a BST

1) • traversals: preorder K, L, R
   inorder L, K, R ⇒ dă crescător
   postorder L, R, K

```
tree_walk(x, order) //x=root; order=in, pre,post
if x<>nil
  then    if order= pre
            then write key[x]          preorder
          tree_walk(left[x],order)
          if order= in
              then write key[x]        inorder
          tree_walk(right[x],order)
          if order= post
              then write key[x]        postorder
```

2) • BST - search recursive ⇒ O(h), h ∈ [lg n, n]

```
BSTreeSearch(x,k)
  if (x=nil or k=key[x])
      then return x
  elsif k< key[x]
      then BSTreeSearch (left[x],k)
  else
      then BSTreeSearch(right[x],k)
```

running time? O(h)

- BST - search ITERATIVE

O(h), DAR smaller machine time for iterative version (overhead w/ stack)

```
BSTreeSearch Iterative(x,k)
  while(x≠ nil and k≠ key[x])
    do
      if k< key[x]
        then x← left[x]
        else x← right[x]
  return x
```

2) BST Insert O(h)

- always as a leaf (nu cont tipul BT)

```
tree_insert(T,z)   //x=root; z=new node, already allocated
y<-nil             //y=x's parent; stays behind x;
x<-root[T]
while x<>nil       //search loop to find the position to insert
  do  y<-x         // y=x at the prev step
  if key[z]<key[x]
      then x<-left[x]
      else x<-right[x]
p[z]<-y            //position found; x=nil; y=new node (z)'s parent
if y=nil           //in case the tree was empty before this insertion
      then root[T]<-z
      else   if key[z]<key[y]
                then left[y]<-z
                else right[y]<-z
```

Iterative

```
treeInsert(T,z)
y← nil
x← root[T]
//search to find pos to insert
while (x≠ nil)
    do y ← x
    if (key[z]<key[x])
        then x← left[x]
        else x← right[x]
p[z]← y
if y=nil
    then root[T]← z
    else if key[z] < key[y]
            then left[y]← z
            else right[y]← z
```

caută poziția de inserat (frunză)

pointăm lui z

niciun nod inserat adăugat acolo încă să îl pun

3) BST – delete

- cases ⇒ 1) as a leaf (e șterge pur și simplu)
  2) 1 child node (copilul îl înlocuiește)
  3) 2 child node
  ↳ înlocuiesc cu max subtree stg / min subtree dpt.

```
tree_delete(T,z)              //z=node to delete; y physically deleted
if left[z]=nil or right[z]=nil
      then y<-z               //Case 1 OR 2; z has at most 1 child => del z
      else y<-tree_successor(z) //find replacement=min(right)
if left[y]<>nil               //we are in Case 2; y is a single child node
      then x<-left[y]         //y has no child to the right; x=y's child
      else  x<-right[y]             //case 2 or 3. Why?
if x<>nil                     //y is not a leaf;
      then p[x]<-p[y]         // y's child redirected to y's parent = x's parent
   //becomes the former single (why?) grandparent
if p[y]=nil                   //means y were the root
      then root[T]<-x         //y's child becomes the new root
      else  if y=left[p[y]] //link y's parent to x which becomes its child
                then left[p[y]]<-x
                else right[p[y]]<-x
return[y]      //outside the procedure: copy y's info into z; dealloc y
```

BST Delete Evaluate O(h)

- find node to delete ⇒ O(h)
- find succ/pred ⇒ O(h)
  ↳ dar dacă găsirea nodului ia O(h) ⇒ leaf ⇒ last
  no succ needed
  ⇒ dacă nu e frunză
  ↳ succ/pred căutat de la acel loc în jos/sus
  ↳ find node + find succ = O(h) total

⇒ OVERALL ⇒ O(h) Delete

Curs #6
Balanced Trees. Augmented Trees

II Augmented Trees

- **Augmented data structures**
  ↳ proprietate / comportament additional pt a ajuta (to speed) realizarea eficientă a taskuri păstrând TOATE proprietățile

Order Statistic (OS) Trees

↳ BALANCED BST
↳ tin sub control înălțimea

- augmentare? stochează la nr. nodului info adiționată (nr de noduri în arbore ce are ca rădăcină un nod dat)

dim[x] = dim[left[x]] + dim[right[x]];

- operațiile pt arbori basic?
  → search, insert, delete, traversal, update

↳ se adaugă Selection & Ranking

I Selection

↳ ret. a i-a cea mai mică cheie din arbore

- Se dă: rank i
- se cere: pointer la i-a cea mai mică cheie
  augmentare: dim environment

$$\begin{cases} dim[x] = dim[left[x]] + dim[right[x]] + 1 \\ dim[nil] = 0 \end{cases}$$

OS Select O(h)

```
r ← dim[left[x]] + 1
if i = r then
        return x ; //găsit
else if (i < r) then
        return OS_Select(left[x], i)
else return OS_Select(right[x], i-r)
```

OS_Select(x, i)

```
r<-dim[left[x]]+1//number of nodes on the left + root
if   i=r                    //found it
  then    return x
  else    if i<r            //ith smallest is on the left
            then
          return OS_Select(left[x],i)
            else            //ith smallest is on the right
          return OS_Select(right[x],i-r)
```

11/15/22

II Ranking

se dă: cheie C a cheie existentă din arbore

se cere: rank-ul în arbore C poziția în inorder walk

Rank = nb of smaller than the checked keys in the tree; **Approach**: count them all
↳ all before = all to left

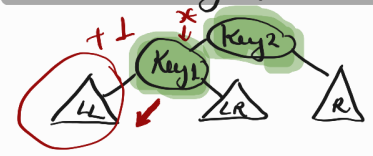
Case #1 nodul dat e un copil drept;

$rank(Key2) = \dim RL + 1 + \dim L + 1$

Count nb of nodes in any subtree până la rădăcina din stg încep. de la nodul dat, până la rădăcină

Case #2 nodul este un copil stâng

$rank(Key1) = \dim LL + 1$



Count nb of nodes in any subarbore către rădăcina din stg începând de la nodul curent la rădăcină

OS Rank O(h)

```
r ← dim[left[x]] + 1
y ← x;
while y ≠ root[T]
do
   if y = right[p[y]]
   then
        r ← r + dim[left[p[y]]] + 1   //case 1
                                      //else case 2 do nothing
   y ← p[y]
return r;
```

OS_Rank(T, x)

```
r<-dim[left[x]]+1
y<-x
while y<>root[T]
do
  if y=right[p[y]]
  then                    //case #1
     r<-r+ dim[left[p[y]]]+1
                          //case #2 (do nothing)
  y<-p[y]
return r
```

11/15/22

Evaluation - Augmented Trees by dimension

Rank & Select

WORST CASE $O(h) = O(\lg n)$

dem 1) pt arb. echilibrați $h = \lg n$

dem 2) OS trees are Red Black trees

Augmented trees - Type II

cerințe: operațiile normale se execută ca în BST' { Walk O(n)
Search, Insert, Delete O(h)

- alte operații: Succ, Pred, Min, Max
  TOATE în O(1)

Informații: – key, left, right, parent pointer –

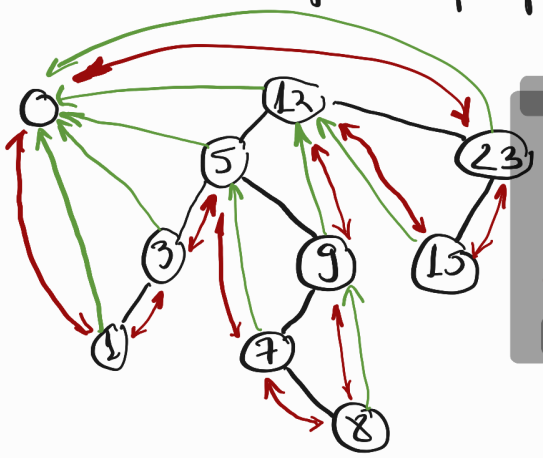
Supplyinfo: succ pointer

Pred pointer → walk through the list

pp ce dă min/max

↳ Se comportă simultan ca BST și DLL

- regular operations (done like in any other BST)
  - ↳ additional updates:
    - set/update pred/succ pointers
      - ⇒ legăturile pt DLL
    - legăturile pt pointerul dublu (pp pointer)



BST links

(pp) pred/succ direct link when up in tree

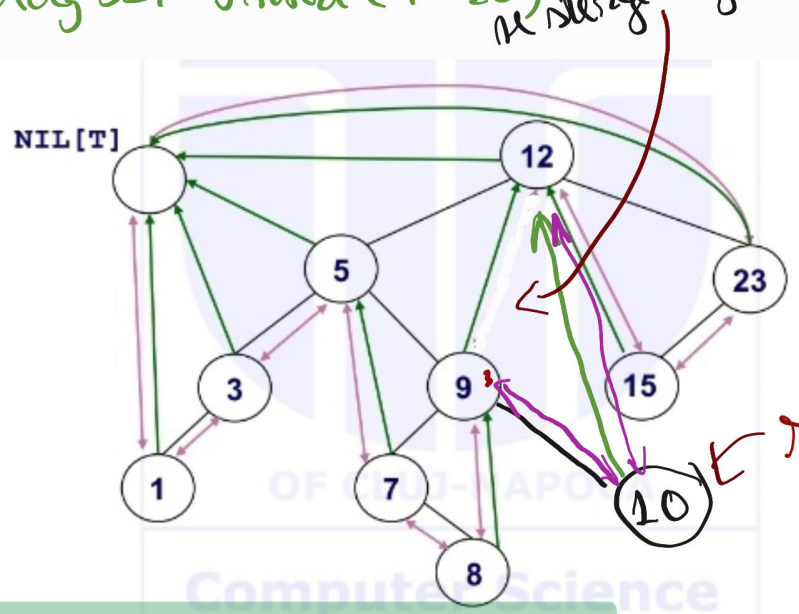
DLL links prev/next

Don't forget!

Augmented Trees – Insert

```
if x=right[p[x]] //inserted node = right child
 then //C1
    pp[x]<- succ[p[x]]
dll_insert_after(p[x],x)
else //C2   inserted node = left child
    pp[x]<- pred[p[x]]
dll_insert_after(pp[x],x)
```

Aug BST – Insert (T, 10)



← right child

```
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)
```

Augmented trees DELETE

z = nodul ce trebuie șters (content replaced by y)

y = nodul ce e de fapt șters (max 1 copil)

x = singurul (sau niciunul) copil al lui y

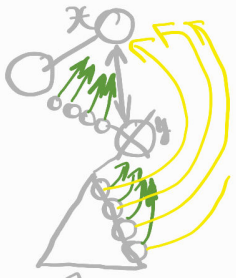
z = y dacă are cel mult un copil

DELETE ⇒ Regular DELETE for BST (+)

```
if right[y] = nil
   then
      x ← left[y]
      while (x ≠ nil)
            do pp[x] ← pp[y]
               x ← right[y]
      dll_del(y);
elsif left[y] = nil;
      x ← right[y]
        while (x ≠ nil)
              do pp[x] ← pp[y]
                 x ← left[y]
      dll_del(y);
```

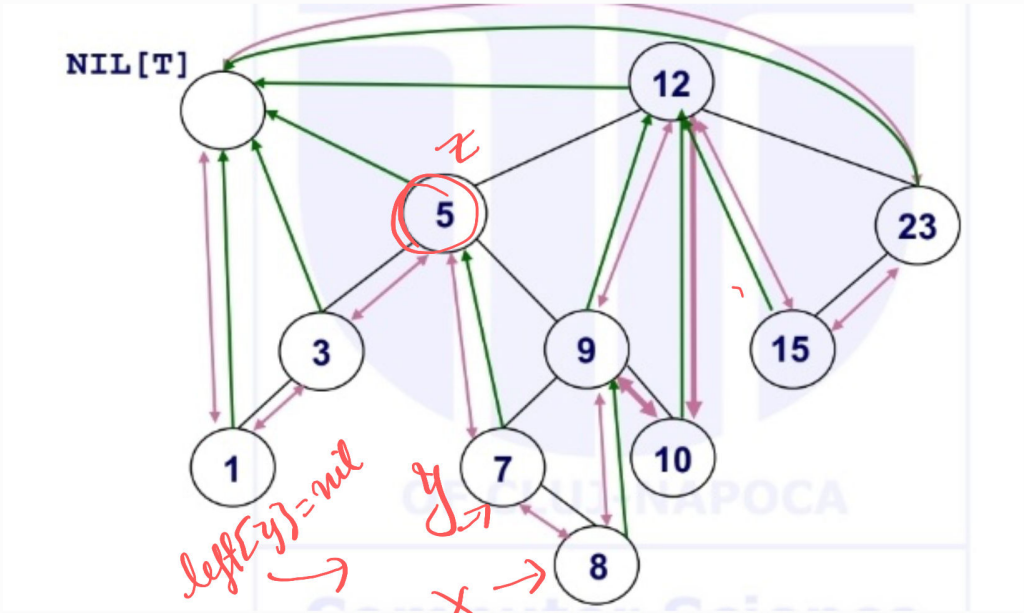
right child

left child



obs: pp-urile pointează spre y ... când îl șterg trb să updatez pp-urile pt tot ce este pe dreapta ca să pointeze ppul spre x

Delete (T, 5)



left[y] = nil

x ← right[y] = 8

x ≠ NIL ⇒ pp[x] ← pp[y]

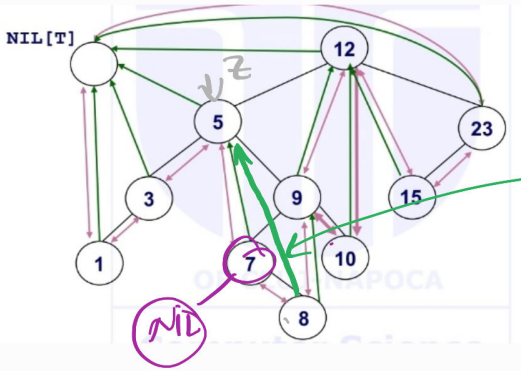
⇒ pp[8] ← 5

x ← left[x] = NIL // stop

dll-del(y) // 7

(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)

Del (T, 5)



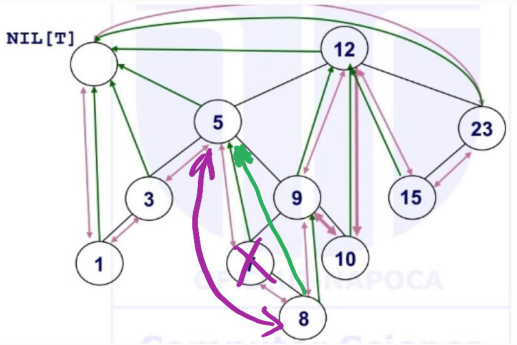
z = 5

y = 7

x = left[7] = 8

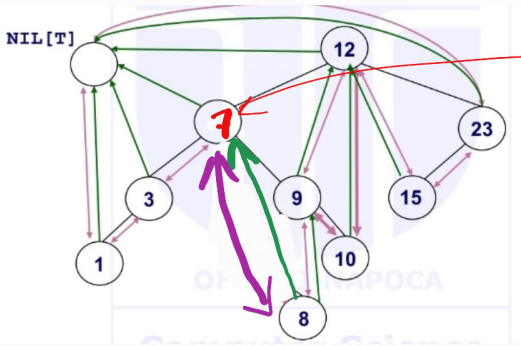
pp[8] ← pp[7] = 5

x ← left[8] = NIL



dll_del(y=7)

↳ refacе și legătura!

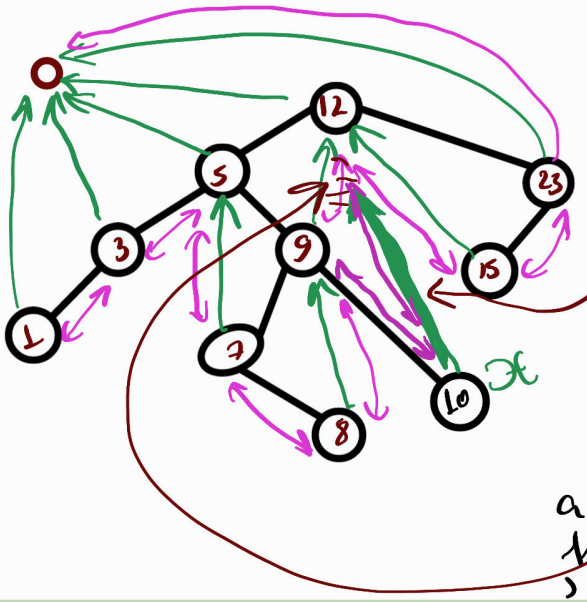


îl înlocuiește val lui 7 cu cea a lui y

```
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)
```

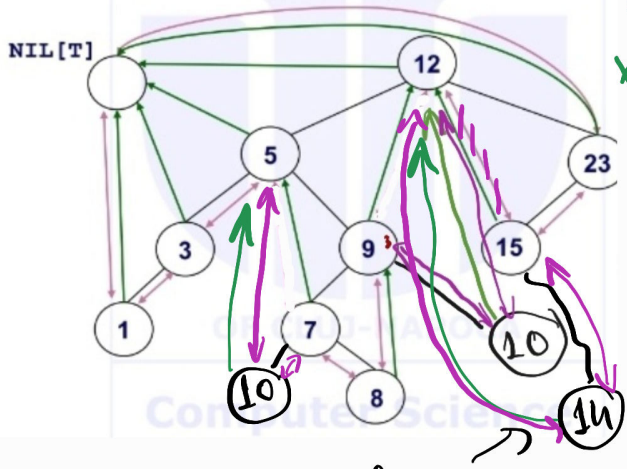
Alte exemple pt insert

Ins(T,10)



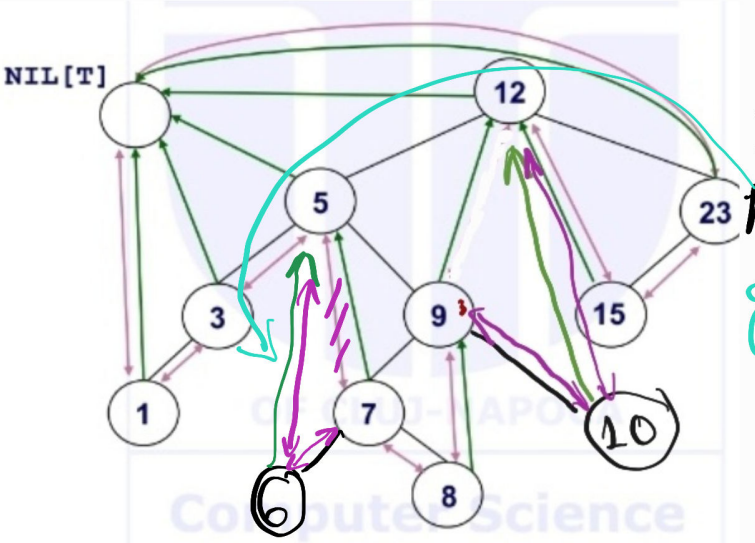
x=10; p[x]=9
x=right(9)
Case #1
pp[x] = succ[p[x]]
= succ(9) = 10
dll insert after(p[x],x)
⇓
refacc leg de la p[x] si succ[p[x]]
a.î. să succ(p(x)) = x
și succ(x) = old succ[p[x]]

Se va refacc leg. a.î. pred[p[x]] ↔ x și x ↔ p[x], și se STERGE leg pp[p[x]] ↔ p[x]



left!

Insert(T, 14)
x = left(p(x))
⇒ pp[14] = pred(15)
11
12
Si
dll insert(pp[x], x)
⇓
pred(px) ↔ x
și
x ↔ p[x]
și
Sterg pp[p(x)] - p[x]



Insert(T, 6)

x = left(7)
pp[x] = pred[p(x)]

dll_insert (pp[x], x)

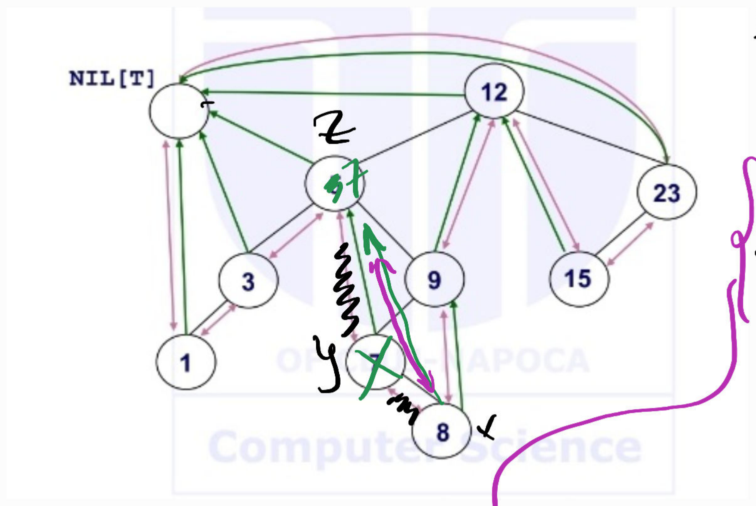
Exemplu delete

```
elseif left[y]=nil
  then
    x<-right[y]
    while  x <> nil
      do  pp[x]<-pp[y]
          x<-left[x]
    dl_list_del(y)
```

```
if right[y]=nil
  then
    x<-left[y]
    while x<>nil
      do  pp[x]<-pp[y]
          x<-right[x]
    dl_list_del(y)
  else                    //symmet
```

Delete(T,5); Regular Delete ⇒ y = 7 (min subarb dr)

Augmented tree

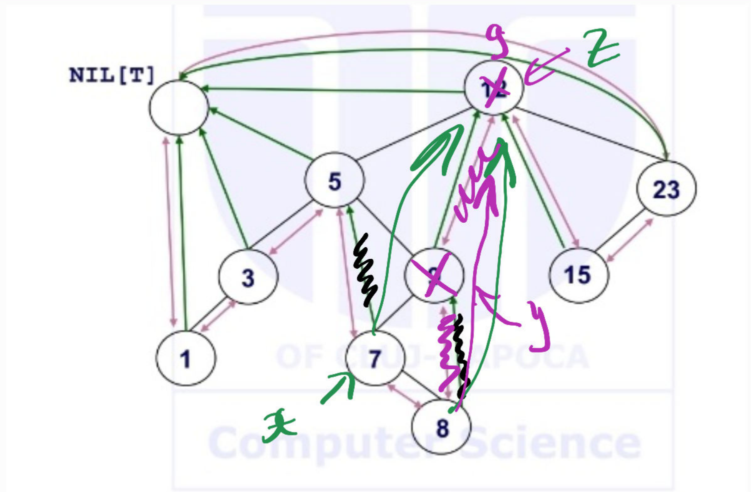


left(y) = NIL (Case 2)

=> { x = 8
pp[x] ← pp[y]
x ← NIL ↳ Stop!
dll_list_del(y)

→ sterge legătura între pred(y), y și succ(y) + face leg între Z și right[y]

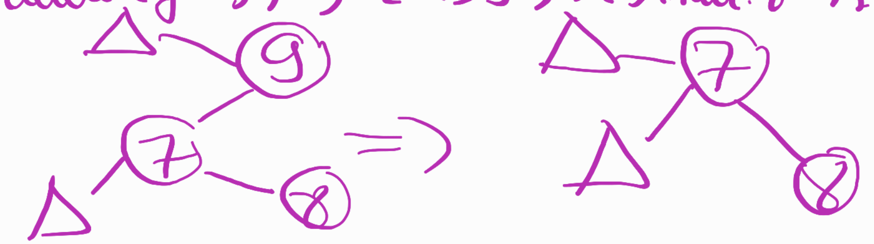
→ toți copiii din stg o să aibă pp[x] → pp[y] (Z)



Delete(12) => Z = 12 ; (aleg max sub Stg)
=> y = 9 => x = left(y) = 7

{ pp[x] ← pp[y] = 12
x ← left[x] => x ← 8 => pp[8] ← 12

dll_delete(y = 9) => 12↔9, 9↔ inloc. 8 → 12



In a nutshell

P0) caut cu Delete4BST nodul cu care pot înlocui nodul ce doresc să-l șterg (max S. stg / min S dr) (in exemplu iau max S stg)

P1) verific dacă are copii (va avea maxim un copil)
↳ dacă îl am pe dreapta, TREBUIE să refac legătura pred/succ pt fiecare nod din stg pt că el pointa la nodul y .... cum îl ștergem pe y „schimbarea de direcție" se realizează fix la nodul Z
↳ la copiii din dreapta nu mă bag... și pointerii în cel mai „rău" caz la p[y] ce nu suferă modificări

P2) e necesar să fac legăturile DLL între frunza de pe stg (în cazul în care merg pe stg ca la P2) cu poziția lui Z (val. va fi înlocuită cu val y)
+ trebuie să mai șterg legătura între y↔Z și y ↔ ultima frunză din stg

Augmented trees

MAX

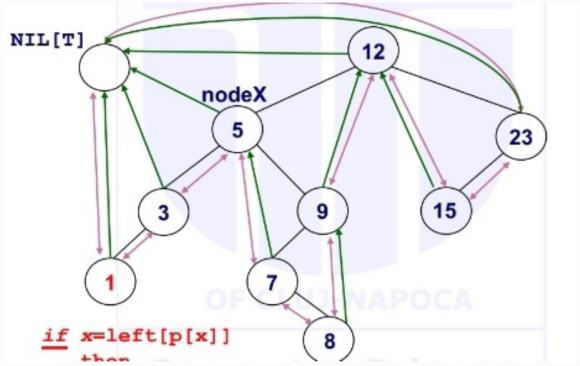
```
if x = left[p[x]]
  then
      return pred[p[x]]
  else
      return pred[pp[x]]
```

MIN

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

obs

'/on the rightmost branch, **HAS TO BE** pp[x]=nil and
pred[nil[T]] = last node in inorder = last node in the list



Min(1) = ?
succ[p[x]] = succ(3) = 1

Min(7) = 1
left(9) = 7 (5)
=> ret succ[pp[7]] = 1

Max(23) = ?
x = 23 ≠ left(12)
=> pred[p[23]] =
= pred(12) = 23

Max(1) = ?
x = 1 => left(3) = 1 =>
=> ret pred(pp[x]) =
pred[nil] = 23 (avem santinela)

Max(9) = ?; x = 9 ≠ left[5]
=> pred[pp[9]] =
= pred[12] = 23
(santinela)

Max(3) = ? x = 3 = left[5]
=> ret pred[p[3]] = 23
(santinela)

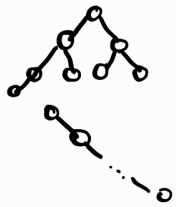
OBS

for insert in empty tree

```
Tree_ins(T,x)
  if(x = root[T])
      then
          root[T] ← x
          p[x] ← nil[T];
          pp[x] ← nil[T];
          dll_insert_after(pp[x],x);
  else .....
      // regular case pag 4
```

I Basic trees

- Walk
  - ⇒ pre/in/post order $O(n)$ daca compl. în afara ap. rec este $O(1)$
- Search
  - BT $O(n)$
  - BST $O(h)$ → best $h = \lg n$; → worst $h = n$

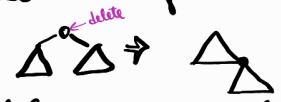


  - balanced BST $O(\lg n)$ merge

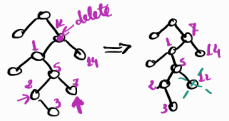
BST DELETE

- cazuri

0) Leaf ⇒ just remove it!
1) 1 child node ⇒ înlocuiesc copilul cu părintele
2) 2 childrens node
   - 2.1) change the tree



   - 2.2) înlocuiesc nodul cu succesorul/predecesorul și îl șterg




```
tree_delete(T,z)          //z=node to delete; y physically deleted
if left[z]=nil or right[z]=nil
      then y<-z           //Case 1 OR 2; z has at most 1 child => del z
      else y<-tree_successor(z)  //find replacement=min(right
if left[y]<>nil           //we are in Case 2; y is a single child node
      then x<-left[y]     //y has no child to the right; x=y's child
      else  x<-right[y]         //case 2 or 3. Why?
if x<>nil                 //y is not a leaf;
      then p[x]<-p[y]     // y's child redirected to y's parent = x's parent
   //becomes the former single (why?) grandparent
if p[y]=nil               //means y were the root
      then root[T]<-x     //y's child becomes the new root
      else  if y=left[p[y]]  //link y's parent to x which becomes its chil
                then left[p[y]]<-x
                else right[p[y]]<-x
return[y]       //outside the procedure: copy y's info into z; dealloc y
```

11/15/22

```
tree_delete(T,z)   // ret y ⇒ nodul șters
if (left[z]=nil or right[z]=nil]   //0/1 ⇒ 0 copii
      then y←z
      else y←tree_successor(z)     // sau predecesor
if (left[y]≠NIL)                   // 1/2 copii Case 2/3
      then x←left[y]
      else x←right[y]              // still case 2/3
if x≠NIL                           // y nu e frunză
   then p[x]←p[y]
if p[y]=nil
      then root[T]←x
      else if y=left[p[y]]         // legăturile părintelui
                                   // devin legăturile
                                   // fiului
           then left[p[y]]←x;
           else right[p[y]]←x;
return[y];
```

Evaluate BST Delete

find node to delete $O(h)$

find succ/pred $O(h)$

but..
- dacă căutarea în $O(h) \Rightarrow$ nodul e leaf $\Rightarrow$ case 1 $\Rightarrow$ no succ/pred needed
- dacă nodul ce e șters nu e frunză, succesorul caut de acolo în jos (nu de la root) $\Rightarrow$ find node + find succ = $O(h)$ în total

Delete ... $O(h)$

Find Min / Max in BST — O(h)

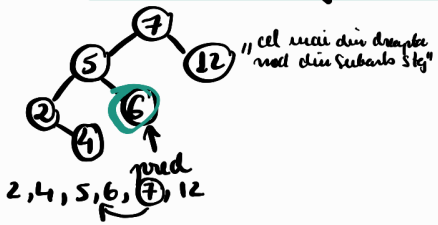
```
find_tree_max(x)
while (right[x] ≠ NIL)
do x ← right[x];
return x
```

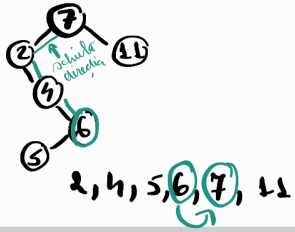
```
find_tree_min(x)
while (left[x] ≠ NIL)
do x ← left[x];
return x
```

Q: What if left[x] = nil => ret root = x

Find pred / succ — O(h)

pred = max in the left subtree SAU primul nod (de jos in sus) urcat „ se schimbă direcția la stânga






```
find_tree_predecesor(x)
if left[x] ≠ NIL          // regular case
  then return find_tree_max(left[x])
y ← p[x]
while (y ≠ NIL and x == left[y])
    do x ← y
       y ← parent[y]
return y;
```

O(h)

Find succesor => O(h)

succ = min in the right subtree SAU primul nod pt care se schimbă direcția la dreapta

```
find_tree_succesor(x)
if (right[x] ≠ nil)
      then return find_tree_min(right[x]);
y ← p[x];
while y ≠ nil and x = right(y)
   do x ← y
      y ← p[y]
return y;
```

O(h)

BST - Evaluate

Theorem: All operations in a BST (except trav.) takes O(h)

↳ mai rapizi decât listele

↳ WORST CASE: h = n

↳ eficientizare? BALANCED trees

BALANCED trees

⇒ augmented BST to keep height under control
⇒ indiferent de tipul de balansare, înălțimea e proporțională cu $\lg n$ ($c \cdot \lg n$, $c \geq 1$ dar ct. MIC)

Perfectly balanced trees
PBT
– nodes related –

- OBS: orice subarbore al unui PBT este TOT PBT
- $b = n_R - n_L \in \{-1, 0, 1\}$
- $h = \lg n$
- Insert: $O(n)$ → inserat ca un BST ($O(h) = O(\lg n)$) dar e nevoie de n rotații pt rebalansare → $O(n)$
- Delete: $O(n)$ → șterge ca un BST $O(h) = O(\lg n)$ dar sunt necesare n rotații ⇒ $O(n)$

Concluzie? Best h property, but difficult to mentain

Balanced trees ~ AVL
– height related –

- AVL = BST + balance (height related)
- any subtree of an AVL tree is an AVL tree
- $b = h_R - h_L \in \{-1, 0, 1\}$
- PBTs are AVLs
- cel mai rebalansat AVL? Fibonacci trees ($b = -1$ pt orice nod)

AVL – evaluate

- Insert: $O(h) = O(\lg n)$
- Delete: $O(h + \lg n)$ ⇒ sunt necesare max $\lg n$ rot

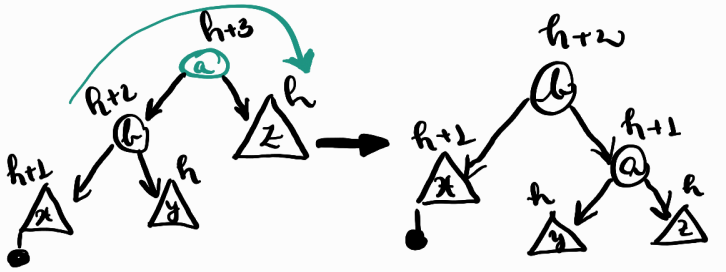
$$h \leq 1.45 \log n$$

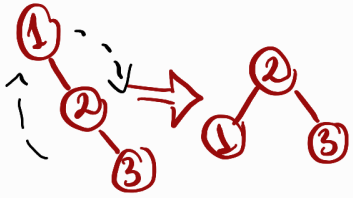
- easy to mentain for insertion
- deletion might make may changes in the structure

AVL – ROTATIONS

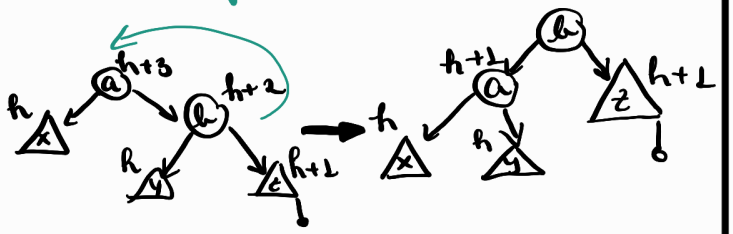
I. Single rotation

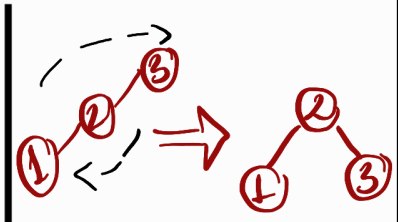
I.1 Left-Left – / Right rotation "\"





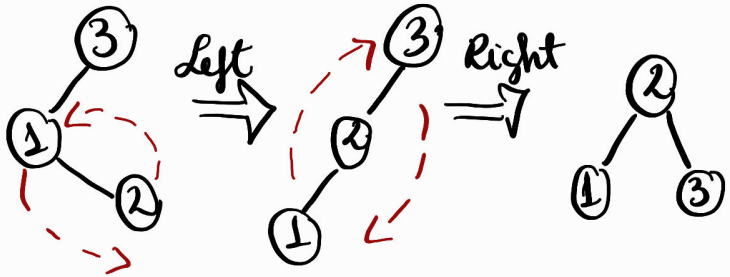
I.2) Left rotation "/"



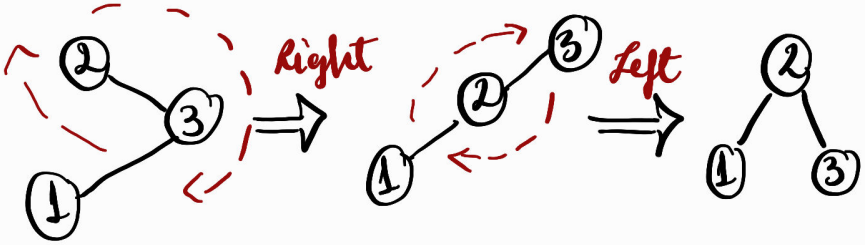


II Rotații duble

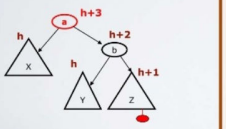
II.1) Left Right Rotation "<"



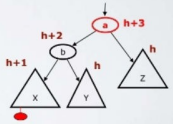
II 2) Right Left Rotation ">"



Rotatie stanga: cand un nod este inserat la ***dreapta*** copilului **dreapta (b)** a celui mai apropiat stramos cu *bf = -2 (dupa inserare)* **(a)**



Rotatie dreapta: cand un nod este inserat in sub-arborele **stang** al copilului **stanga (b)** al celui mai apropiat stramos *cu bf = +2 (dupa inserare)* **(a)**

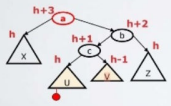


19

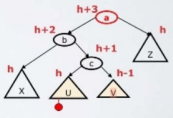
AVL - CUM FOLOSIM ROTATIILE DUBLE?

UNIVERSITATEA TEHNICA

Dreapta-stanga: cand un nod este inserat in sub-arborele **stang** al copilului **drept (b)** al celui mai apropiat stramos cu *bf = -2 (dupa inserare)* **(a)**



Stanga-dreapta: cand un nod este inserat in sub-arborele **drept** al copilului **stang (b)** al celui mai apropiat stramos cu *bf = +2 (dupa inserare)* **(a)**

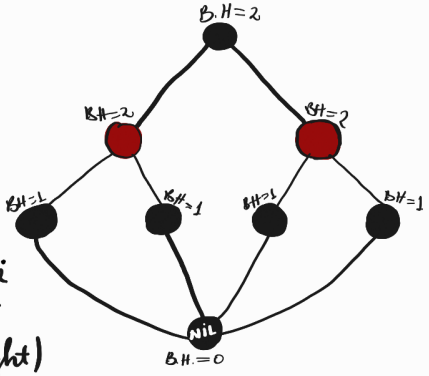


Curs 7 – FA

I Red-Black Trees. II Disjoint Sets

I. Red Black Trees

- Balanced trees
- Insert / Delete $\Rightarrow O(\lg n)$
  - ↳ constant time for rebalancing



DEF:

P_0: root $\Rightarrow$ black

P_1: nodurile sunt fie negre fie roșii

P_2: toate nodocinile (nil) sunt negre

P_3: ambii copii ai unui nod roșu sunt negri

P_4: orice drum de la orice nod până la frunză are același număr de frunze negre (black height)

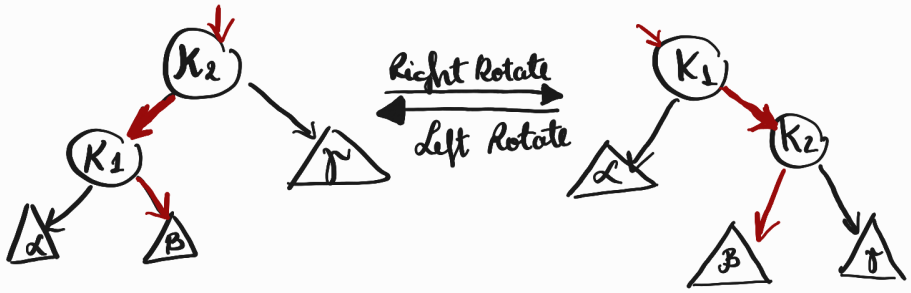
obs: subarborele

Teoremă: RB tree cu n noduri are înălțime max $2\lg(n+1)$

obs:
- un arbore are cel puțin $2^{bh(x)}-1$ nod.
- $bh(x) \geq h/2$

- **Red Black trees rotations**

→ similar cu single rot. pt AVL




```
Left-Rotate(T, x)
//x root of rotation (points on k1)
y<-right[x]          //y saves k2
right[x]<-left[y]    //right of k1 goes on β
if left[y]<>nil      //if β exists = is not nil
      then p[left[y]]<-x // β's parent becomes k1
p[y]<-p[x]           //k2's parent what was k1's parent
if p[y]=nil //k1 used to be the root of the tree
      then root[T]<-y
      else if x=left[p[x]]      // the parent of k1 becomes the parent of k2
            then  left[p[x]]<-y
            else  right[p[x]]<-y
left[y]<-x           //k1 goes the left child of k2
p[x]<-y              //k2 becomes the parent of k1
```

Red Black Insert

similar cu BST → ca frunză

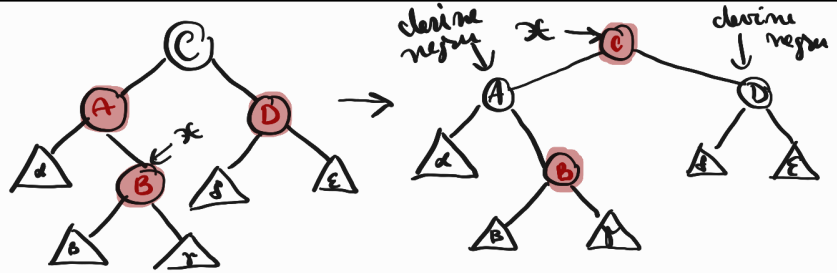
- Culoare? ROȘU De ce? Strică P_4 (Black height)
- verificăm proprietățile
- Rebalansare: RB-INSERT-FIXUP
- P_3 → ambii copii ai unui nod roșu sunt negri
  - ↳ adev. pt copii (nils)
  - ↳ fals pt noduri inserate al căror părinte e roșu

- **RB Insert Fixup – Case #1**

A = părinte (red); D = unghi (R); C = unchi (B)

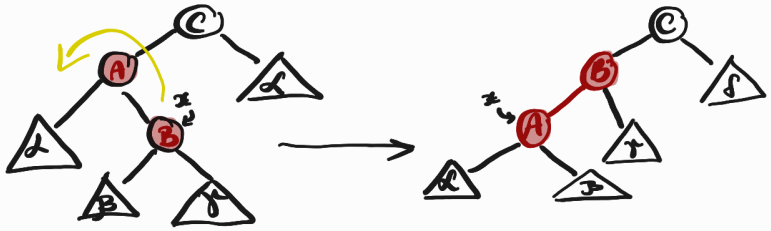
B – nod inserat, pointat de x

```
parent ← black // eliminăm P3
unele ← black  // menținem P4
grandparent ← red  // menținem P4
x ← grand p
```



RB Insert Fixup Case #2

parent(A)=red, Uncle(x's root) = BLACK, grand parent(C)=black



left rotate(x)
x ← left(x)

OBS: cazul #2 ⇒ O(1)
DAR P3 încă nu e respectată

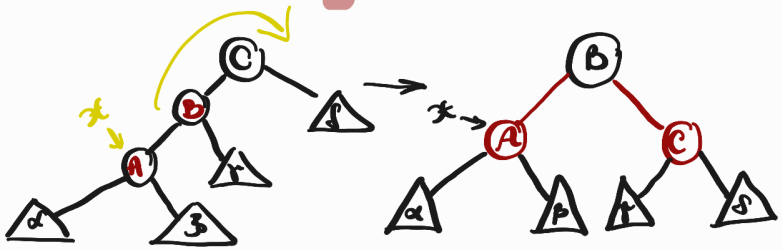
RB Insert Fixup - Case #3

- Insert A

Parent (B)=red; uncle (y's root)=black, grandparent(C)=black

- α, β, γ δ → RB trees

parent ← black
grandparent ← red
right_rotate (p[p[x]])



RB Insert Evaluate

- fiecare cat poate să coste O(1)
- caz #1 se poate repeta până la rădăcină
- caz #2 e urmat de C#3 (nu e încă prop P3)
- caz #3 rezolvă problema

RB insert Rebalancing evaluate

- caz #1 se repetă până la root O(h)
- caz #2 + caz #3 ⇒ rezolvă pb O(1)
- caz #3 ⇒ rezolvă pb O(1)

Insert O(lg n) + rebalancing

— Running Time —

WORST : #1 repeats	O(lg n)
Best : #3 ⇒ 1 rotation	O(1)
Others : #3 + #2 ⇒ 2 rotations	O(1)

O(lg n) overall worst time

↳ caz 1 repeats, at most 2 rotations

RB DELETE

- as regular BST + properties check to rebalance
  ↳ P4 (black height) - ia issue

```
rb_delete (T, z)
    tree_delete (T, z)
    if color [y] = black
        then RB-DELETE-FIXUP (T, x);
```

- z ⇒ nodul ce va fi șters
- y ⇒ nodul ce e de fapt șters
- x = singurul (sau nil) copil al lui y (după ce y e șters, x îl înlocuiește)
- w = fratele lui y (bf. del) ~ fratele lui x (aft. del)

RB-DELETE - FIXUP

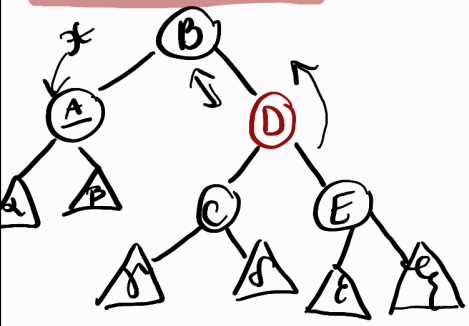
↳ verifică P4 (black height)

```
if color[x] = red
    then color[x] ← black   // pb fixed... DONE
else color[x] ← double_black  // P1 issue (red or black only)
```

↳ double black is another color

When a black node is deleted and replaced by a black child, the child is marked as *double black*. The main task now becomes to convert this double black to single black.

CASE #1



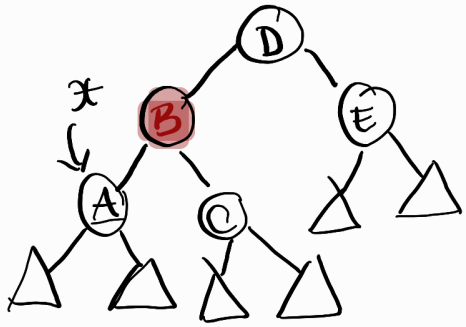
A - double black node

α, β, γ, δ, ε, ζ - RB trees

B - negru (părinte), O - roșu (frate A)

Sol: B ↔ D color interchange + left rotate + care 2/3/4

```
parent[x] ← red
brother[x] ← black
left rotate(p[x])
```



CASE #2

părinte R/B, frate B, copii fraterni ai lui B

Solution?

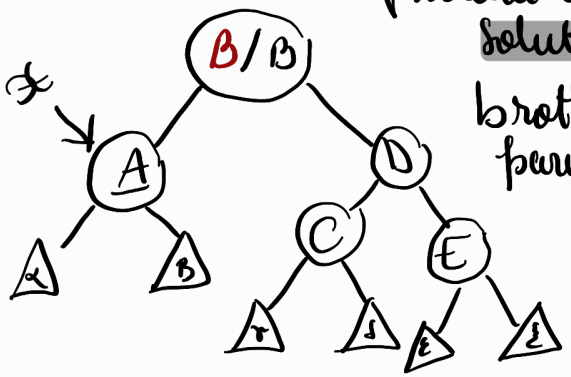
brother ← red (D)

parent ← B/black sau double black

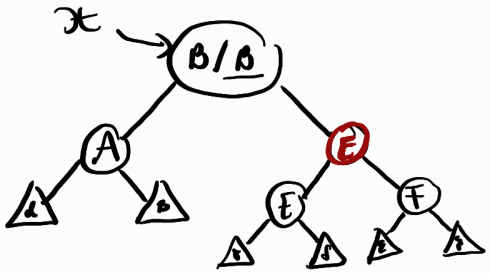
↓

pasez problema „mai sus" în arbore cu un nivel mai sus

cazul 2 se repetă de max $\lg n \Rightarrow$ problem solved




```
brother[x] ← red
if p[x] = black then
        p[x] ← double_black
        x ← p[x]
elsif p[x] = red then
        p[x] ← black
```



CASE #3

x = double Black

p[x] = red / black

bro[x] = black

left child [bro[x]] = red

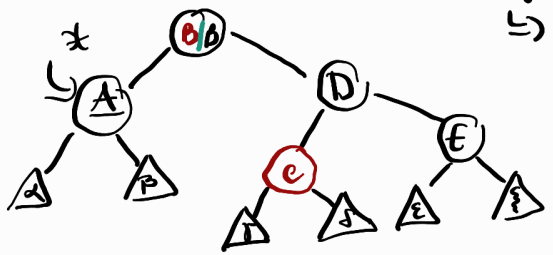
↳ right child [bro[x]] = black

Solution?

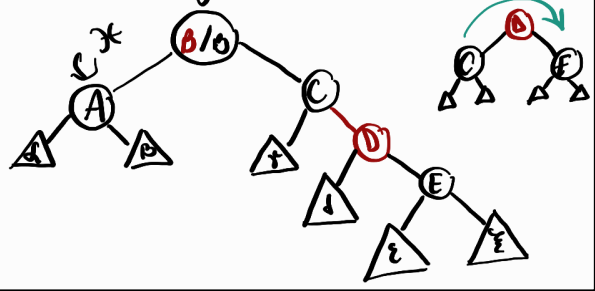
left child [bro[x]] = black

bro[x] ← red

right rotation (bro[x])

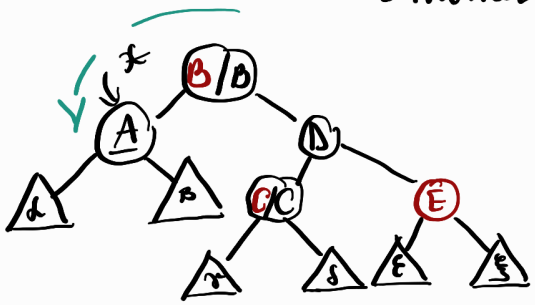



```
brother[x] ← red
left[brother[x]] ← black
right_rotate(brother[x])
```



CASE #4

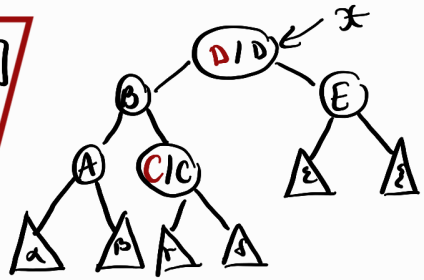
- parent: Black / red
- brother: black → left child: black/red
  → right child: red



Solution?

brother ← parent color
parent ← black
left not brother
color right [bro[x]] ← black

```
brother[x] ← color[parent[x]]
parent[x] ← black
left_rotate(p[x])
color[right[w]] ← black
x ← root[T]
```



RB-del → Rebalancing Evaluate

- Case #1 + oricare din C2,3,4 — O(1)
  - #1 + #2 ⇒ pb solved ⇒ O(1)
  - #1 + #3 + #4 ⇒ pb solved ⇒ O(1)
  - #1 + #4 ⇒ pb solved ⇒ O(1)
- Case #2 (no rotation, only coloring)
  ↳ repeats 1 levels up in the tree

WORST CASE ⇒ O(lg n)
BEST CASE ⇒ O(1)

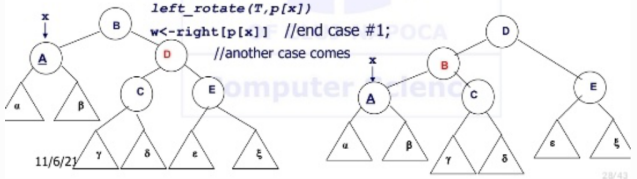
- Case #3 → rotație urmată de C#4 ⇒ O(1)
- Case #4 — rotație, solve the pb ⇒ O(1)

DELETE O(lg n) + rebalancing

- WORST : #2 repeats (recoloring only) ⇒ O(lg n)
- Best : #4 ⇒ 1 rotation ⇒ O(1)
- others : #1 + #2 / #1 + #3 + #4 } ⇒ 2/3 rotations ⇒ O(1)

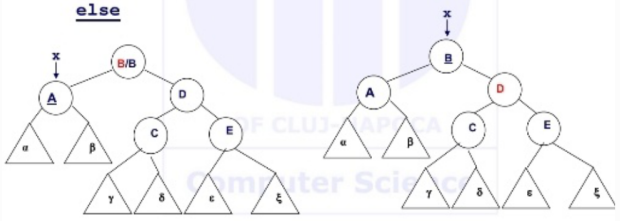
```
RB-DELETE-FIXUP (T,x)
while x<>root[T] and color[x]=black
do
   if x=left[p[x]]                    //cases on the left
   then                               //else case symmetric on the right; not discussed
        w<-right[p[x]]                //w=x's brother
        if color[w]=red
        then   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
```

Case #1



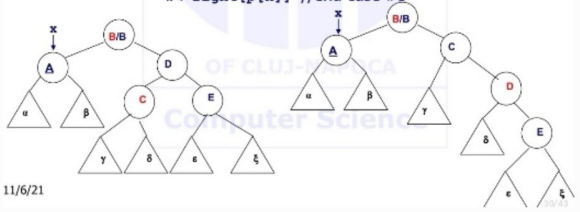

```
if color[left[w]]=black and color[right[w]]=black
   then                               //case #2
        color[w]<-red
        x<-p[x]
   else
```

Case #2



```
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
```

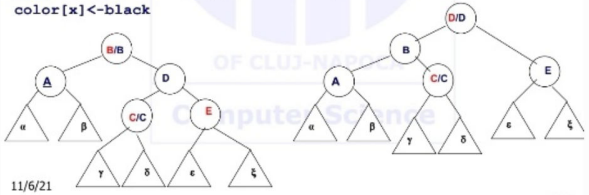
case #3



11/6/21

```
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
```

#case 4



11/6/21

Conclusions on balanced Searched trees

TREE TYPE	HEIGHT	Ins	DEL
BST	$[\lg n, n]$	$O(h)$	$O(h)$
RBT	$[\lg n, 2 \cdot \lg n]$	2 rot	3 rot
AVL	$[\lg n, 1.45 \lg n]$	1 rot	$\lg n$ rot
PBT	$\lg n$	n rot	n rot

II Disjoint Sets

o ce sunt? colecții de DS dinamice, $S \in \{S_1, \dots, S_k\}$
↳ (?) n elemente în toate cele k seturi ($n \geq k$)
↳ fiecare element S_i se identif. de către elem. reprezentativ sau reprezentativ

o OPERAȚII

- MAKE-SET(x) generează un nou set cu un singur element
  → inițial sunt n seturi ↗ fiecare element rep. propriul său element reprezentativ

- UNION(x,y)
  ↳ unește 2 mulțimi disjuncte rep. de x și y
  ↳ construiește un nou set $S_x \cup S_y$ și distruge seturile S_x, S_y
  ↳ reprezentantul e dat de unul dintre cei 2 rep. (x sau y)

- FIND-SET(x)
  ↳ returnează pointer la elem reprezentativ al setului ce conține elementul x

- Implementare
  ⇒ linked list
  ↳ elem reprezentativ ⇒ head
  ↳ • elem din listă / key
  • pointer la urm elem
  • pointer la elem. reprezentativ

Op:

- MAKE SET (x) → constr. listă cu 1 elem O(1)
- FIND SET (x) → ret. rep. O(1)
- UNION(x, y) → pune lista lui x la finalul listei lui y
  - rep = former y's representative
  - all x's elements have to update rep. pointer

WORST CASE O(m²) for all op

- n make SETS
- union ⇒ n times (to get a single set)
  1+2+.....+n-1 → O(n²)
- n ~ m (m > n, dar n e liniar în m)

On AVERAGE O(m) → UNION call x m calls
⇒ O(m²)

How to Increase efficiency ?

- update pointerii pt lista cu dim. mai mică
  ↳ menținez dimensiunea (ca la OSTrees)

THEOREM pt n obiecte în LL cu „weighted" union ?? pt m MAKE-SETS, FIND-SET, UNION are complexitatea O(m + n lg n)

FOREST OF Disjoint SETS

- Set = tree cu rădăcină, păstrez pointer la părinte
  1 node = 1 el.
  1 tree = 1 set
- the root = representative element
  op echivalente:
  - MAKE SET(x) - build tree doar cu 1 nod (rădăcina)
  - FIND SET(x) - goes up and ret the representative
  - UNION(x, y)

EURISTICI

- by Rank ⇒ rank[x] =
  ⇒ înălțime maximă a subarborelui cu rădăcina x
  ⇒ nr de laturi pe cel mai lung drum de la x la o frunză : rank[leaf] = 0
- Find - Set leave ranks unchanged
- O(m · Ack(n)) → O(m)

Implementări

```
MAKE-SET(x)
p[x] <- x
rank[x]<-0
UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))
LINK(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
```

11/6/21

```
FIND-SET(x)
 if x!=p[x]
  then p[x] <- FIND-SET(p[x])
  return p[x]
```

De la curs

Ce aduce echilibru în RBT?
black height + P3 (roșu -> childrens black)

Recap Inserare => as BST + node=RED + fixups

Obs: e nev de max o rot dublă pt reechilibrare
și posibil $\frac{\log n}{2}$ recolorări

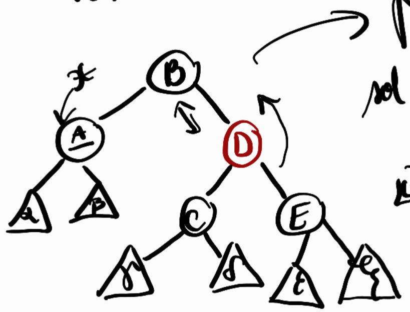
Stergere : inue: stergem nod negru
↳ artificiu => double black
dacă după stergere un nod are
focusul asupra lui => doubleblack

y = nod ce se sterge efectiv fizic
w = fost frate y / actual frate y
x = sg. copil y / null
z = nodul ce se dorește șters

if(x = red).... done ... color[x] <- black
else.. double black.

Case #1

fratele lui A = red



sol => rot stg brother doubleblack
+ schimbare de culoare
↳ fratele lui A => black
#1, #3, #4

Case #2

culoarea părintelui... și a copiilor fratelui
R/B ↳ ambii black
⇓
B/B coboară roșul în copil

⇓ rezolvat.

Case #3

seamănă cu cel de sus Case#2.
copil de frate = Red

Case #4

Recap
1) culoarea fratelui
C1) ↳ red => sch. culoare + rot stg
C2) copii black, black → transmit mai sus culoarea
C3) BR
C4) RR

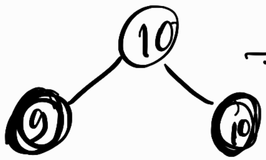
sterg y -> black... rămâne x.. black => un nod mai puțin negru .... x -> double black

Delete $O(\log n)$ + worst => log n repeat #2
+ best => 1 rot #4
+ average => #1, 2 / #4 b, 4 => 2/3 rot

PBT -> self balanced.
=> când ajungem toate cheile de la început .... nu crește h mai mult de log n

RBT	2 rot	3 rot.
AVL	1 rot	lg n rot.

TEMĂ pe arbore



=) insert(14)
insert(36)
insert(39)
delete(15)
delete(30)

Disjoint Sets

căutare => Hash Table
căutare + ord elem => Arbore echilibrați
↳ op bazate pe succ, pred/min/max

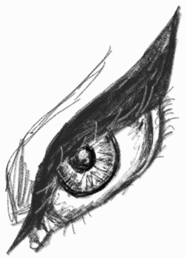
Dacă scopul nu e de a căuta elem; ... ci de partiționare:
Disjoint sets
util la -> Det comp. conexe:

n = nr. op make set = nr. ele din structură
m = nr. total (make set, union, find)

Implementare: vect) L.L. ; first = rep.
↳ fiecare nod are leg. din la rep
=> FIND ... O(1)
0 op. = O(m) <- UNION: $O(m^2)$ -> concatenare -> update rep
MAKESET: O(1)
FIND SET: O(1)
îmbunătățire
) numim nod => lista mai scurtă
↳ toate comp. size
↳ O(log n) pe union
↳ union by RANK (= lungime)

Theorem: op in DS sets => O(m + lg n)

Forest of Disjoint Sets
• a array

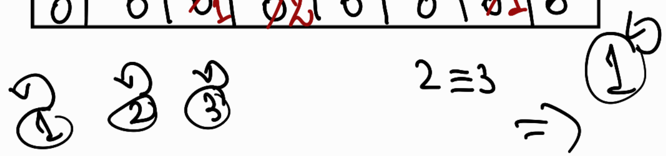


union by rank -> scurtcircuit a f.
să pointeze la rep

find set ... parte a
orig. compresie: the TDK...

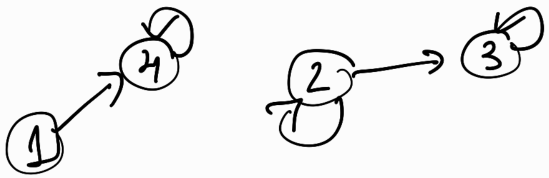
Example union by rank + path compression

	1	2	3	4	5	6	7	8
π	~~1~~ 4	2	~~3~~ 4	4	5	~~6~~ 7	7	8
r	0	0	~~0~~ 1	~~0~~ 2	0	0	~~0~~ 1	0

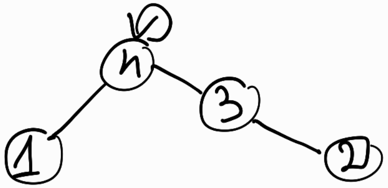


2 ≡ 3
=>

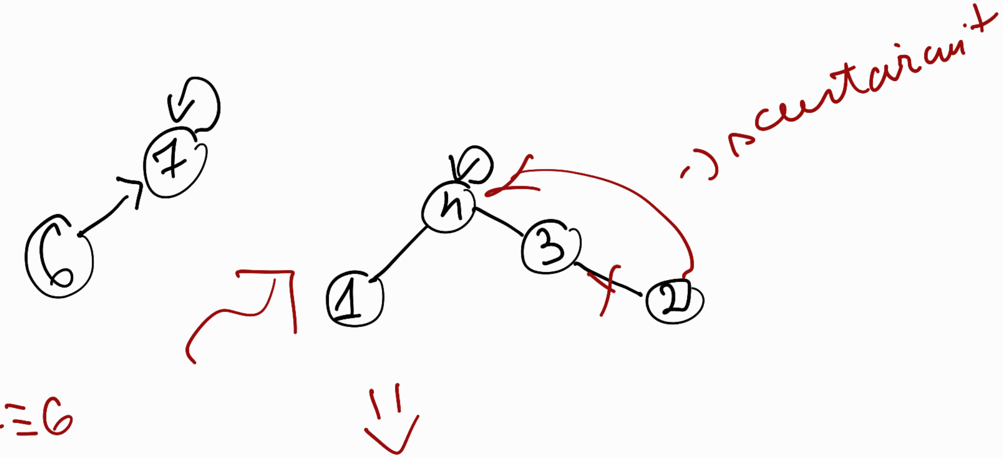
1=4 =>

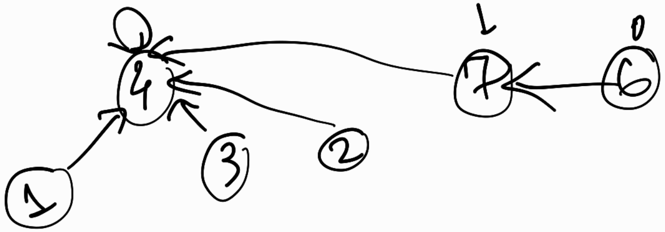


3=4 =>



2=6





ACK
↳ coroana

Obs -> fd crește înlet => considerăm ct