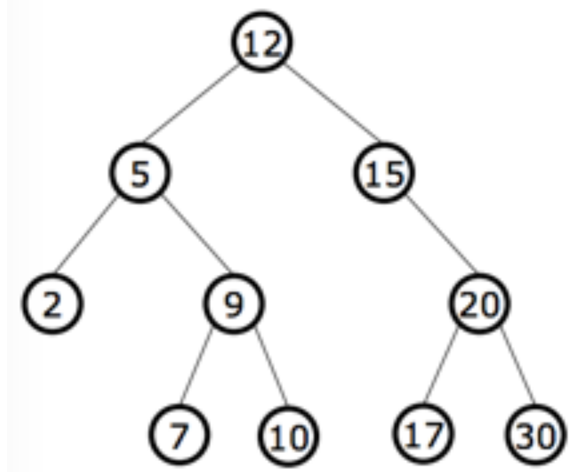
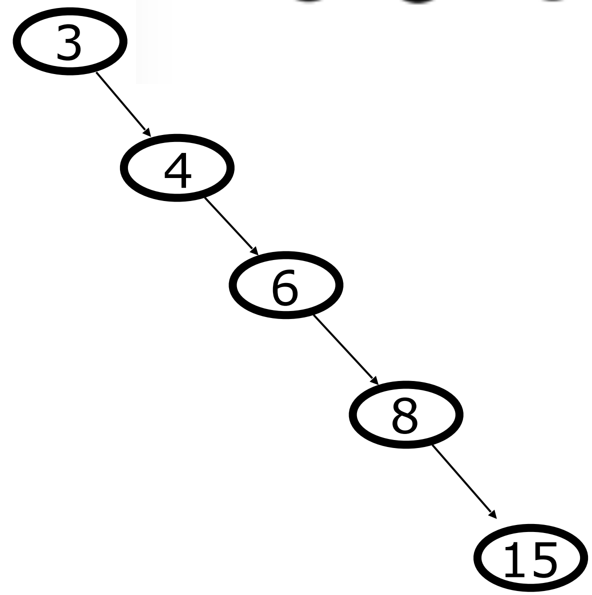
Structuri de date avansate pentru multimi

Arbori binari de cautare echilibrati. Arbori AVL. Arbori Rosu si Negru.

Arbori Binari de Cautare

- Search: *O(h)*
- Insert: *O(h)*
- Delete: *O(h)*
- *h = log n ?*
- Cazul defavorabil?





Performanta ABC

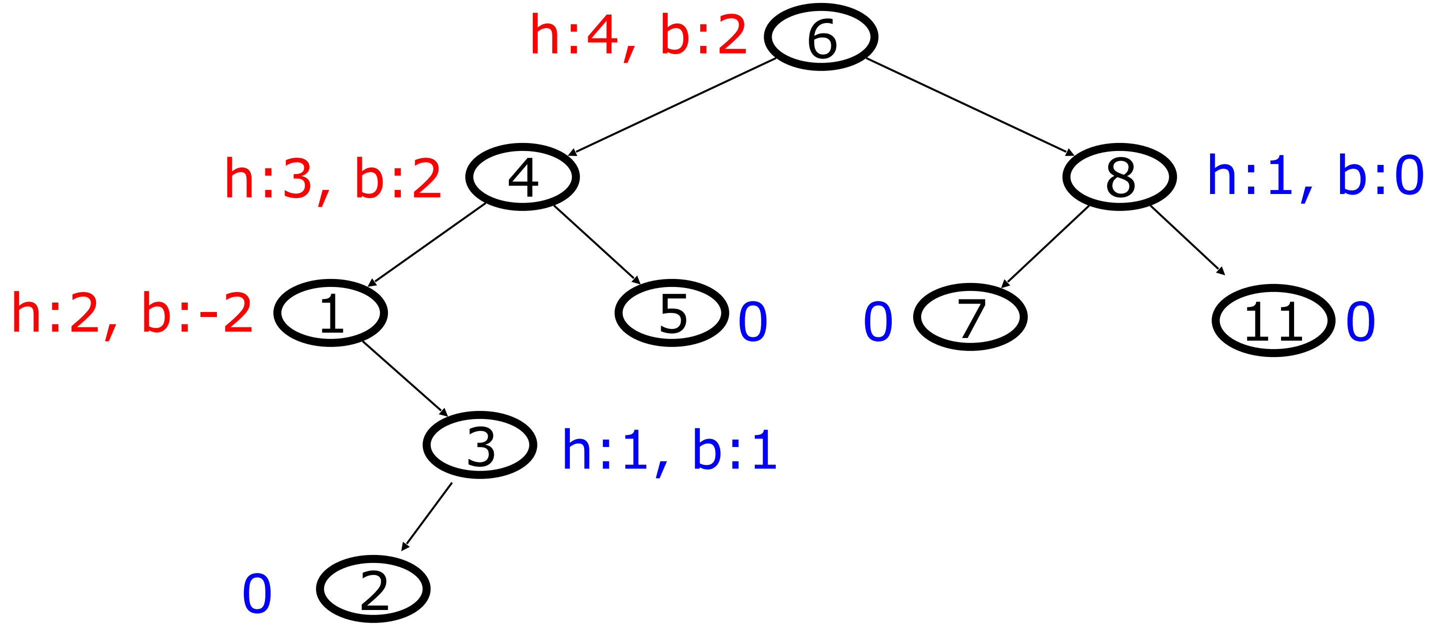
- Putem garanta *h~log n*?
  - constructia initiala
    - chei ordonate -> mediana
    - operatii ulterioare de inserare/stergere nu garanteaza mentinerea conditiei
  - noduri inserate aleator:
    - necesitatea unei conditii de echilibru, care:
      - sa garanteze ca inaltimea este *O(log n)* in orice situatie
      - este usor de mentinut la inserare/stergere

Arbori AVL (Adelson-Velski-Landis)

- pentru orice nod *n*:

$$|balance(n)| \leq 1, \text{unde}$$

balance(n)=height(n.left) - height(n.right)



AVL - conditia de echilibru

- sa garanteze ca inaltimea este *O(log n)*
  - *n(h)* - numarul minim de noduri pt. AVL de inaltime h
  - *n(1) = 1, n(2)=2*
  - *n*>2*,* AVL contine cel putin:
    - *radacina*
    - *sub-arbore AVL de inaltime h-1*
    - *sub-arbore AVL de inaltime h-2*
  - Adica: $n(h) = 1 + n(h-1) + n(h-2)$
  - Intrucat $n(h-1) > n(h-2) \Rightarrow n(h) > 2n(h-2) > 4n(h-4) > \ldots > 2^i n(h-2i)$
  - Rezolvand: $n(h) > 2^{h/2-1}$
  - Aplicam log: $h < 2\log n(h) + 2$

AVL - conditia de echilibru

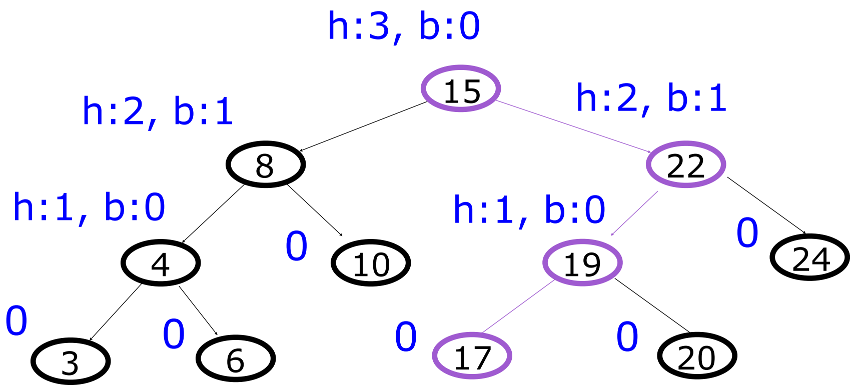
- usor de intretinut
  - traditional, se mentine factorul de echilibru (balance factor) la fiecare nod: +1, 0 sau -1
  - Proprietatile algoritmului de echilibrare
    - dupa *Insert*:
      - modificarea informatiei de echilibru se produce la mai multe noduri inspre radacina, DAR
      - de indata ce s-a executat o rotatie simpla/ dubla arborele se re-echilibreaza
    - dupa *Delete:*
      - este posibil sa fie nevoie de rotatii pe toate nodurile de pe calea de cautare

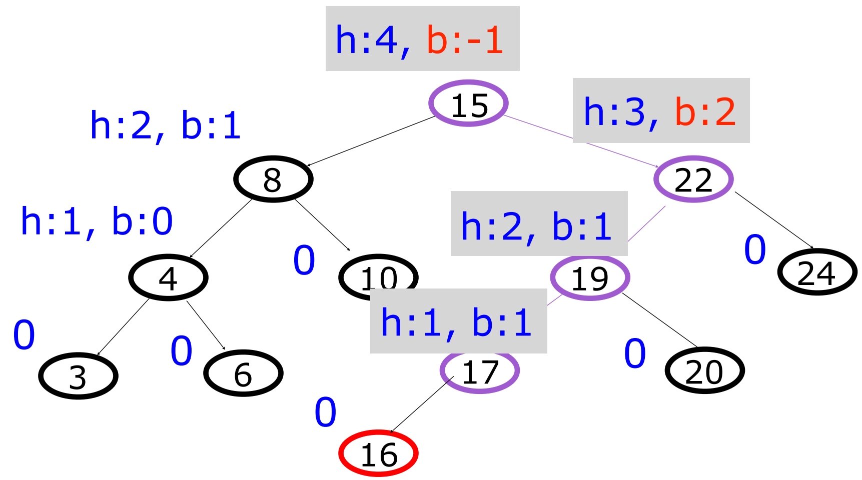
AVL - Operatii

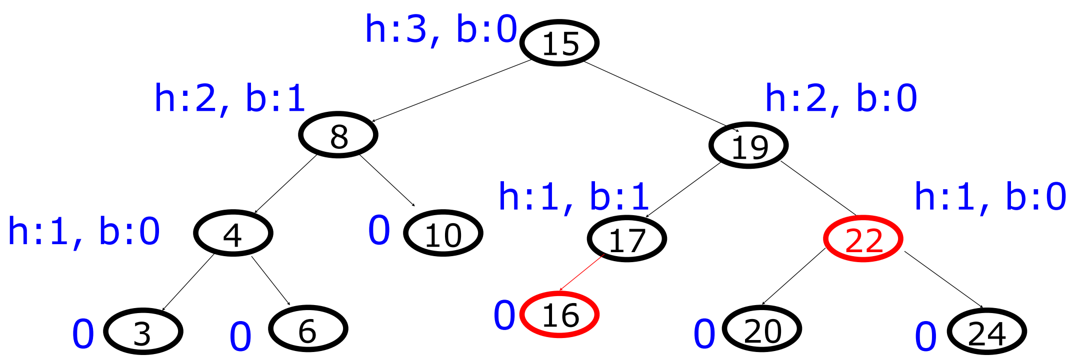
- Search: la fel ca si in ABC
- Insert:
  - inserare ca si frunza (ca si in ABC)
  - verificare echilibru
  - echilibrare (4 cazuri diferite)
    - se rezolva cu rotatii simple/duble
    - exista un cel mai adanc nod care este dezechilibrat
      - daca acesta se reechilibreaza, totul deasupra lui este echilibrat
  - Delete:
    - se elimina nodul ca si in ABC
    - se …

AVL - Exemplu inserare - rotatie simpla

Insert(16)

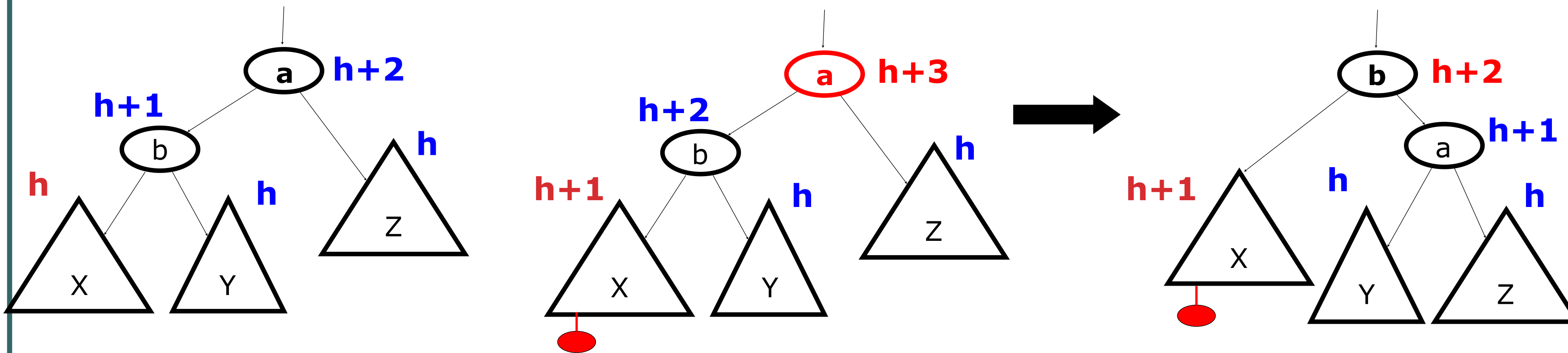




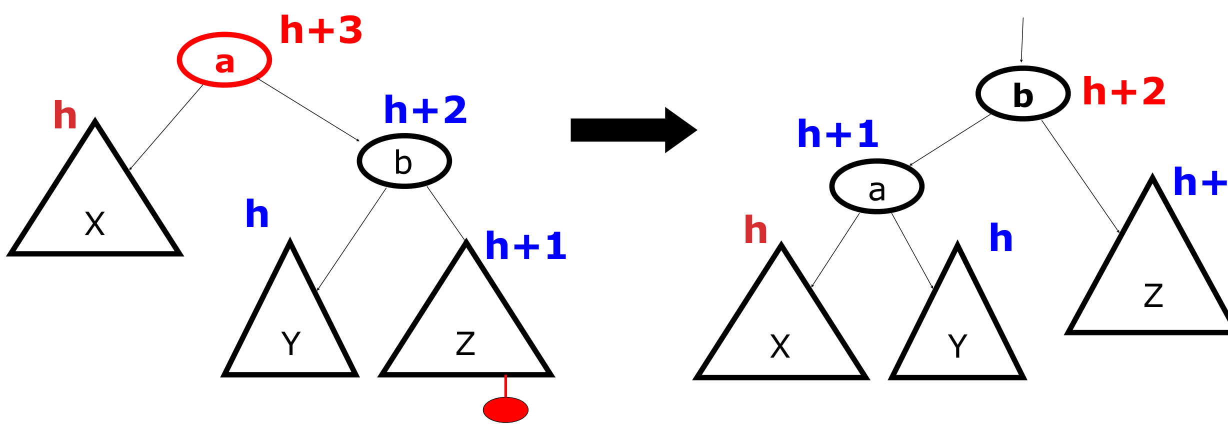


AVL - Rotatii simple

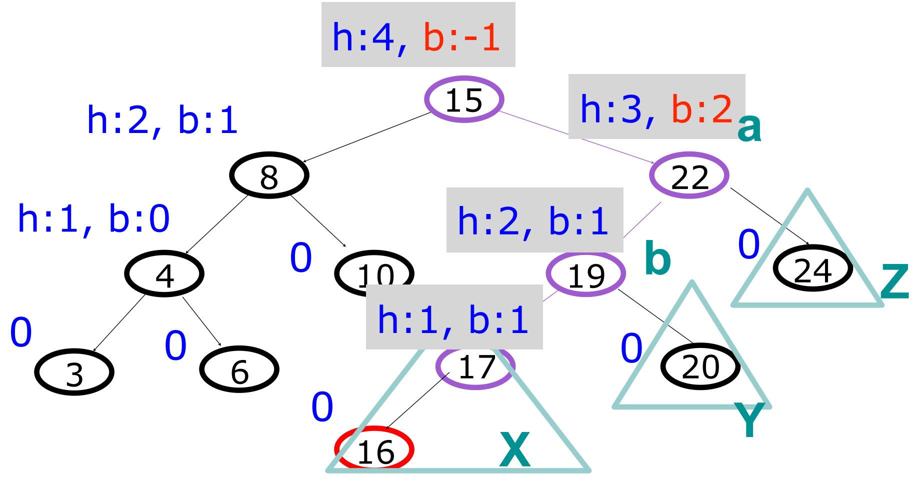
Dreapta

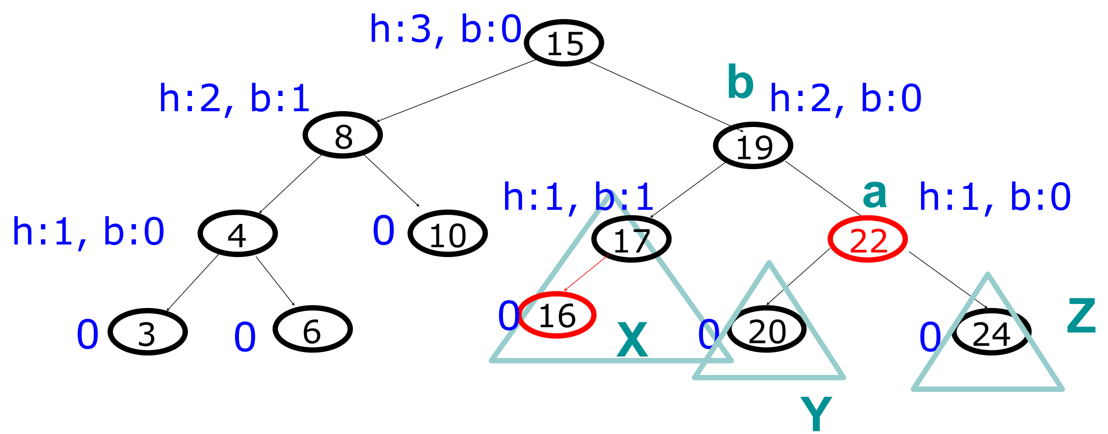


Stanga



AVL - Rotatie dreapta

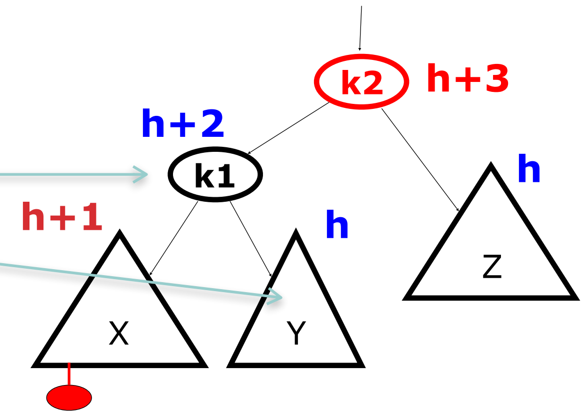


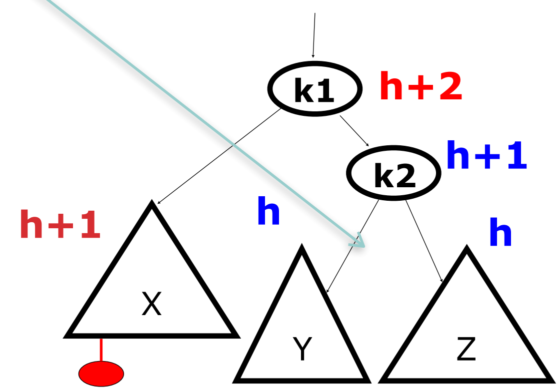


AVL - Rotatie dreapta (exemplu cod)

```
void snglRotRight(AVLNodeT **k2)
{
  AVLNodeT *k1 ;
  k1 = (*k2)->left ;
  (*k2)->left  = k1->right ;
  k1->right = *k2 ;
  (*k2)->height = max(
      height((*k2)->left ),
      height((*k2)->right)) + 1;
  k1->height = max(
     height( k1-> left ),
     (*k2)->height ) + 1 ;
  *k2 = k1; // assign new  root
}
/* snglRotLeft is symmetric */
```

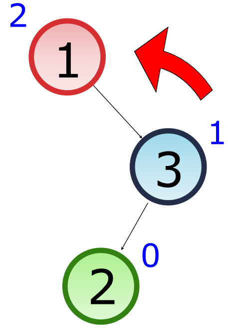
etc….



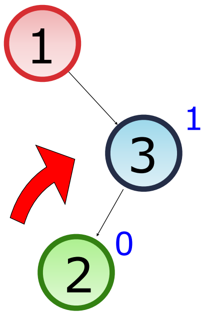


AVL - Proprietati rotatii

- Nodurile din sub-arborele nodului rotat nu sunt afectate!
- O rotatie ia O(1)
- Inainte si dupa arborele isi pastreaza ordonarea de ABC
- Nota: codul pentru rotatie stanga este simetric



(1)



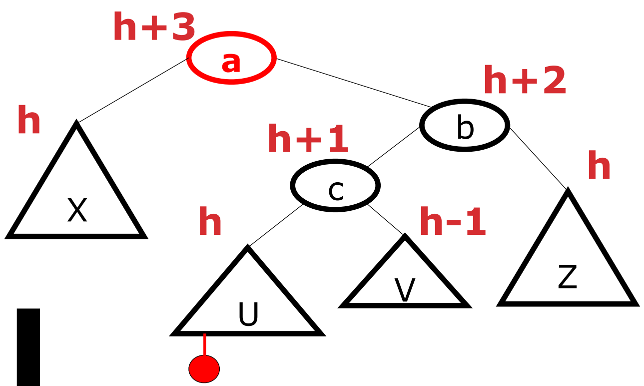
(2)

Rotatiile simple nu sunt suficiente!!!

What if….(2), apoi (1)?

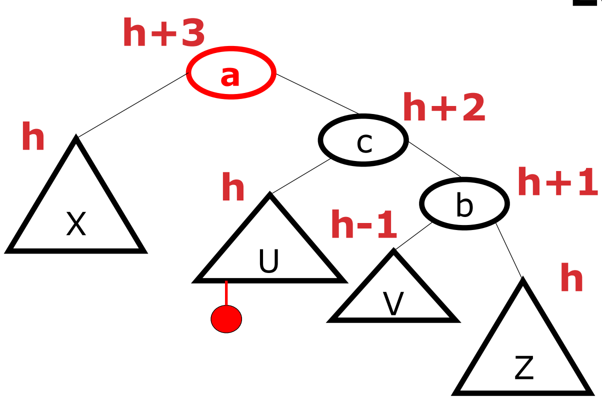
- rotatie intre copil si nepot problematici
- …apoi intre nod si noul copil

AVL - Rotatie dubla: dreapta-stanga

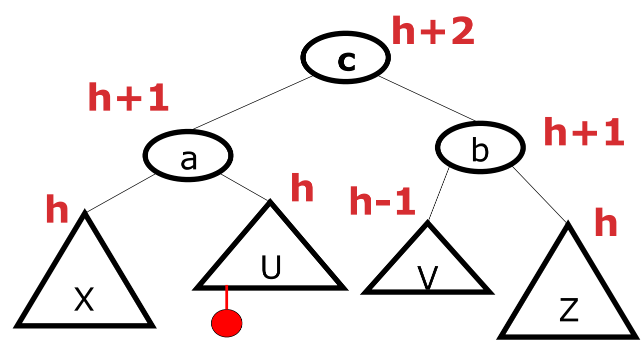



```
void  dblRotLeft( AVLPtr k3 )
{
   // rotate between k1 and k2
   snglRotRight((*k3)->left);
   // rotate between k3 and k2
   snglRotLeft(k3);
}
```

RightRot (b)



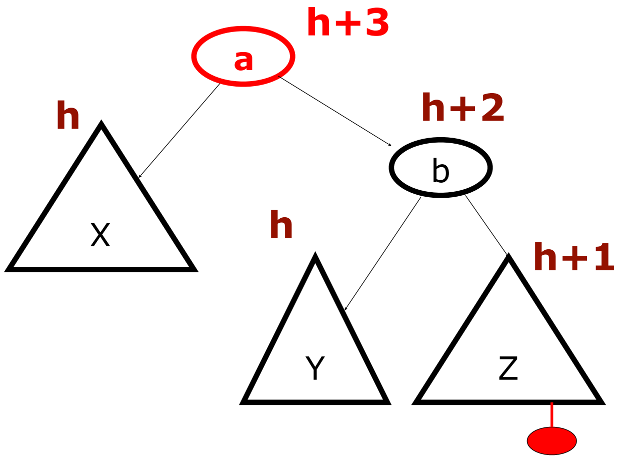
LeftRot (a)

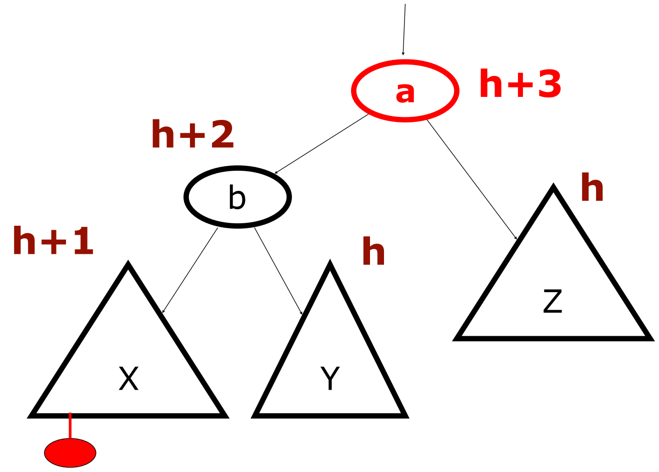


AVL - Cum folosim rotatiile simple?

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

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

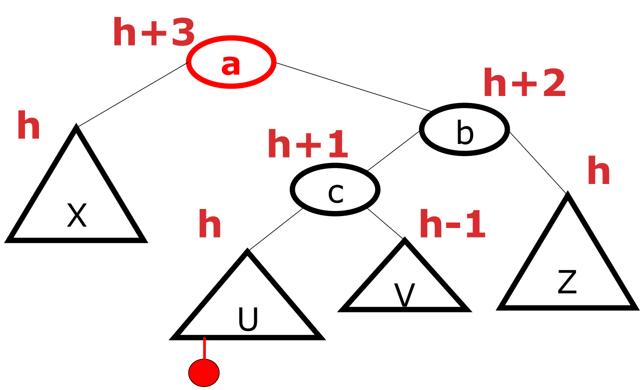


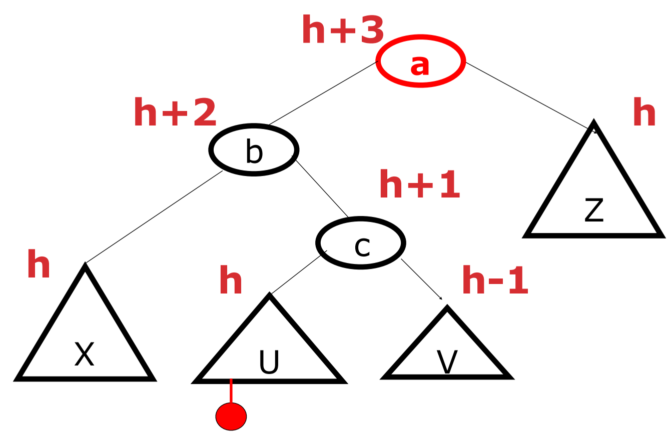


AVL - Cum folosim rotatiile duble?

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

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





AVL - Insert - Complexitate

- Cazul defavorabil: *O(log n)*
  - Rotatie: *O(1)*
  - Lungimea caii catre radacina: *O(log n)* (de ce?)
  - Cel mult 2 rotatii la o inserare (de ce?)
- Complexitate *Search*?
- Complexitate *BuildTree*?

AVL - Stergere

- Eliminarea nodului se face folosind strategia ABC de inlocuire cu succesor/predecesor
- Dezechilibrul se repara prin rotatii
- Se identifica parintele nodului sters in realitate (poate fi predecesorul/succesorul)
  - daca s-a sters un copil stanga: parent.bf--
  - daca s-a sters un copil dreapta: parent.bf++

AVL - Stergere - re-echilibrare

- Fie **a** *cel mai adanc nod care trebuie re-echilibrat*
- Daca nodul *sters* se afla in <u>sub-arborele drept al lui a</u>, fie **b** *radacina sub-arborelui stang al lui a.* Atunci:
  - daca ***b.bf = 0*** *sau* ***b.bf = +1*** *dupa stergere: rotatie dreapta*
  - daca ***b.bf = -1*** *dupa stergere: rotatie stanga-dreapta*
- Daca nodul *sters* se afla in <u>sub-arborele stang al lui a</u>, fie **b** *radacina sub-arborelui dreapta al lui a*. Atunci:
  - daca ***b.bf = 0*** sau ***b.bf = -1*** dupa stergere: *rotatie stanga*
  - daca ***b.bf = +1*** dupa stergere: *rotatie dreapta-stanga*

AVL - Stergere - re-echilibrare

- Spre deosebire de inserare, ***o rotatie s-ar putea sa nu fie suficienta pentru a restabili echilibrul in arbore!***
- Trebuie cautat urmatorul nod unde echilibrul nu este satisfacut (fie acest nod **a**)
- Daca ***a.bf > 0***, fie **b** *copilul stanga* al lui **a**
  - daca ***b.bf > 0***: *rotatie dreapta*
  - daca ***b.bf < 0***: *rotatie stanga-dreapta*
  - daca ***b.bf = 0***: *orice rotatie*
- Daca ***a.bf < 0 …***
  - ***…***

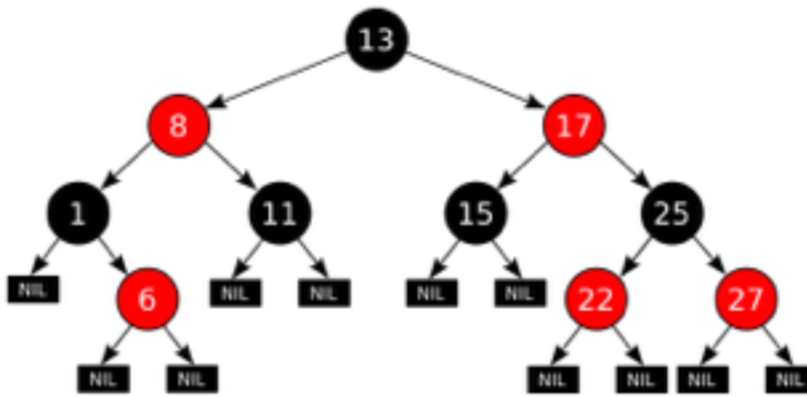
AVL - Stergere - re-echilibrare

- Daca ***a.bf < 0,*** fie **b** *copilul stang* al lui **a:**
  - daca ***b.bf < 0***: *rotatie stanga*
  - daca ***b.bf > 0***: *rotatie dreapta-stanga*
  - daca ***b.bf = 0***: *orice rotatie*

- *Delete* - Complexitate
  - search: *O(log n)*
  - reechilibrare prin rotatii *de la nodul sters pana la radacina: O(log n)*

Arbori Rosu si Negru

- Orice nod: **rosu** sau **negru**
- **Radacina** este **neagra**
- Nodurile **NIL** sunt **negre**
- Daca un nod este **rosu**, ambii copii tb sa fie **negri**
- Orice cale de la un nod dat la un NIL contine acelasi numar de noduri **negre** (*black depth)*



Calea de la radacina la cea mai indepartata frunza nu este mai lunga decat dublul lungimii drumului de la radacina la cea mai apropiata frunza

Aproximativ echilibrat pe inaltime!

Structuri de date pt ADT dictionar

	insert	search	delete
vector nesortat	O(1)	O(n)	O(n)*
lista simplu inlantuita nesortata	O(1)	O(n)	O(n)
vector sortat	O(n)	O(log n)	O(n)
lista simplu inlantuita sortata	O(n)	O(n)	O(n)
Arbore binar de cautare echilibrat	O(log n)	O(log n)	O(log n)
Tabela de dispersie	O(1)	O(1)	O(1)**

Obs: Cazul **mediu statistic!** La **tabele de dispersie, cazul defavorabil - O(n)**

Wait...

- ABC echilibrati - O(log n) in cazul defavorabil
- Tabele de dispersie - O(1) in cazul mediu

- Constant e mai bine!

- **De ce ne pasa atunci de arbori echilibrati?**

Dificultati la tabele de dispersie

- Dispersia e dificil de realizat
  - proprietatea de dispersie uniforma - coliziuni minime
  - rapid de calculat
- Alte operatii se executa extrem de incet pe tabele de dispersie:
  - *findMin*, *findMax*, *Predecessor*, *Successor*: *O(log n) -> O(n)*
  - afisare ordonata (*printSorted*): *O(n) -> O(n log n)*

ABC echilibrati vs tabele de dispersie

- Morala:
  - daca se utilizeaza frecvent operatii bazate pe ordinea sortata a elementelor -> se prefera un ABC echilibrat
  - daca accentul este pe cautari rapide: tabela de dispersie e potrivita
  - poate conta si cantitatea de chei pe care vrem sa le stergem…

Bibliografie