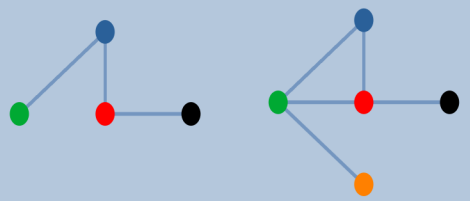
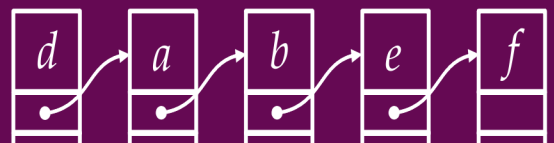
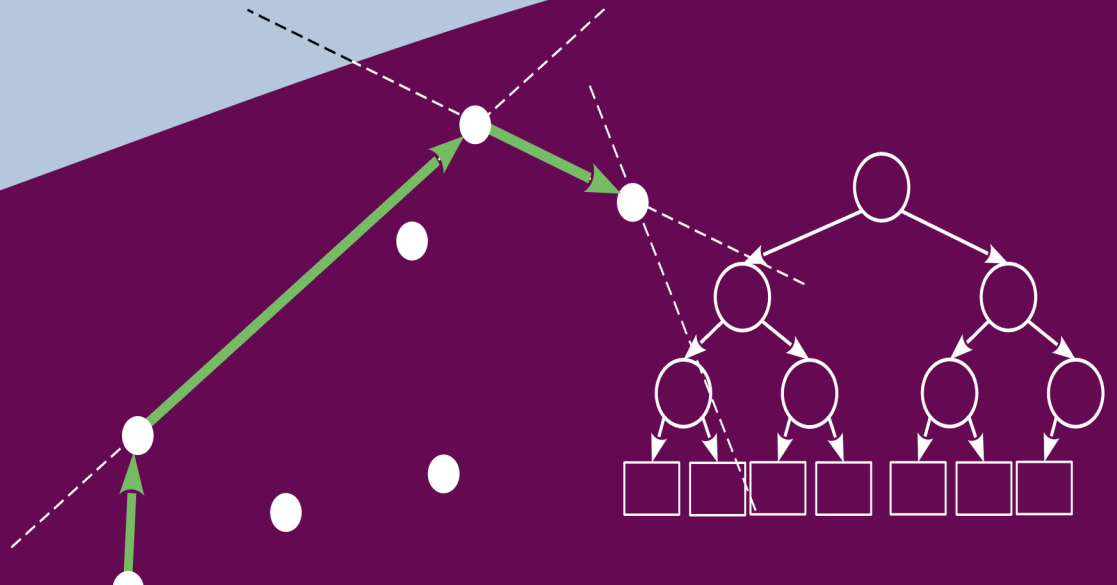
Advanced algorithms and data structures

Week 1: Advanced data structures

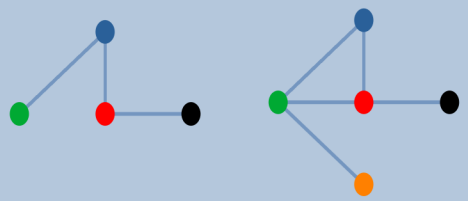


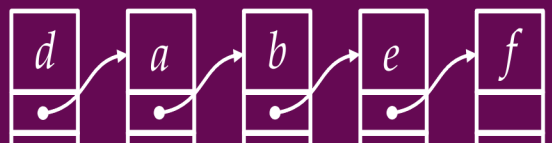


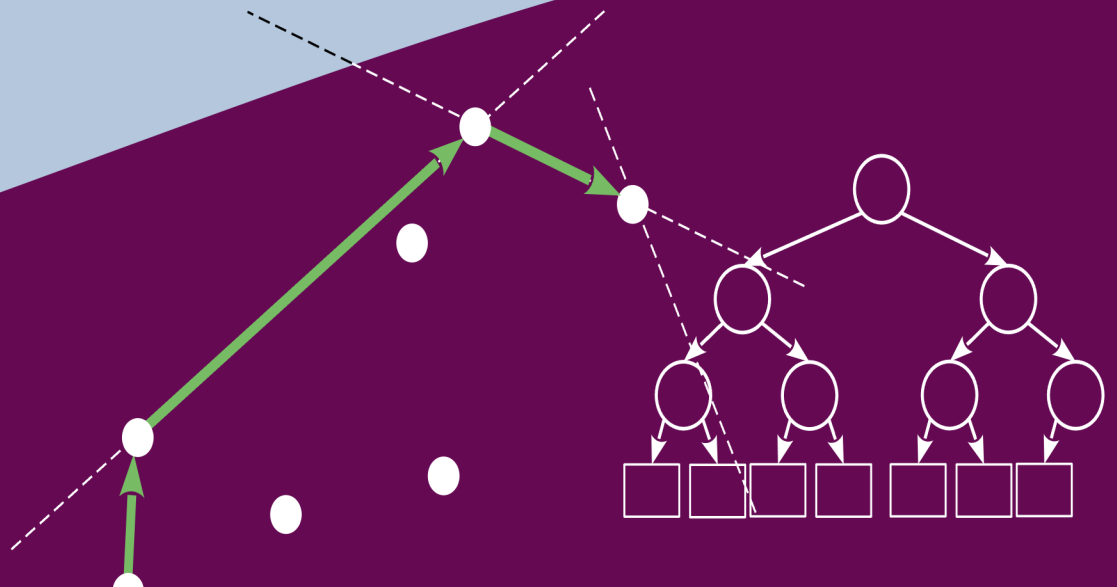


Napredni algoritmi i strukture podataka

Tjedan 1: Napredne strukture podataka







Creative Commons

- you are free to:
  - share — reproduce, distribute and communicate the work to the public
  - rework the work
- under the following conditions:
  - Attribution: You must acknowledge and attribute the authorship of the work in a way specified by the author or licensor (but not in a way that suggests that you or your use of their work has their direct endorsement).
  - non-commercial: You may not use this work for commercial purposes.
  - share under the same conditions: if you modify, transform, or create using this work, you may distribute the adaptation only under a license that is the same or similar to this one.

Creative Commons

- slobodno smijete:
  - dijeliti — umnožavati, distribuirati i javnosti priopćavati djelo
  - prerađivati djelo
- pod sljedećim uvjetima:
  - imenovanje: morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
  - nekomercijalno: ovo djelo ne smijete koristiti u komercijalne svrhe.
  - dijeli pod istim uvjetima: ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.

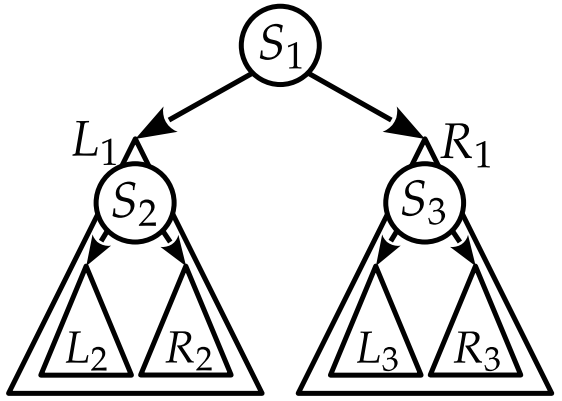
U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.
Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.
Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava. *Tekst licence preuzet je s http://creativecommons.org/*

Binary trees (1)

- Trees are directed acyclic graphs
- Binary trees are specific - each node can have a maximum of two children
  - A binary tree can be described by an ordered triple

$$= (, ,)$$

    - where L is the left subtree, S is the root of the binary tree B, and R is the right subtree
  - Recursive definition

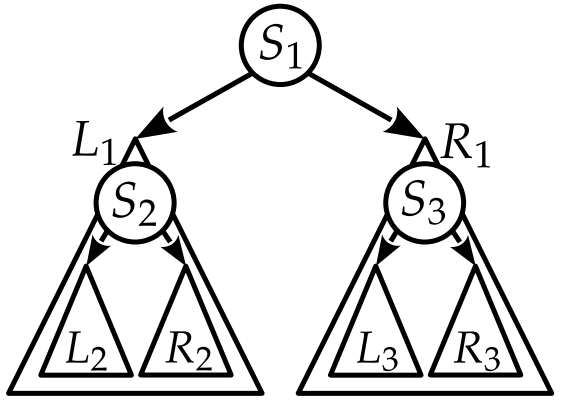


Binarna stabla (1)

- Stabla su usmjereni aciklički grafovi
- Binarna stabla su specifična – svaki čvor može imati najviše dva djeteta
  - Binarno stablo se može opisati uređenom trojkom

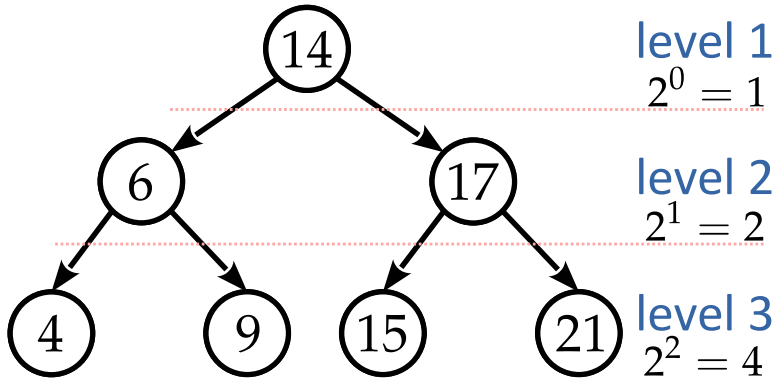
$$B = (L, S, R)$$

    - gdje je L lijevo podstablo, S je korijen binarnog stabla B, a R je desno podstablo
  - Rekurzivna definicija



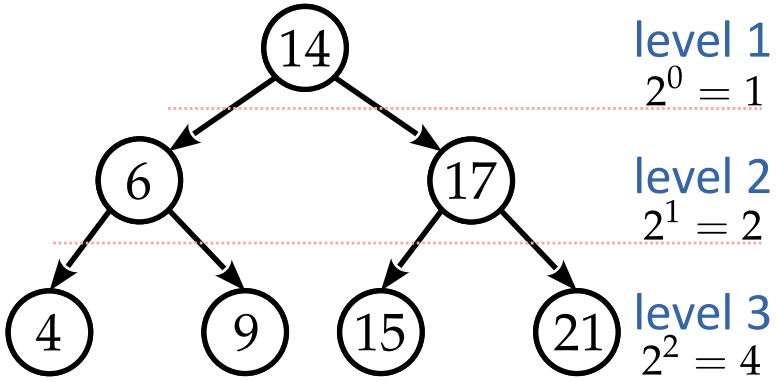
Binary trees (2)

- To describe the relationship between the nodes of a binary tree, we use genealogical terms
  - Parent, child, twins
  - Expressions such as: great-grandfather (parent's parent), uncle (twin of parents) can be used.
- **A perfect binary tree**(*perfect*)
  - A binary tree in which all levels are completely filled with nodes



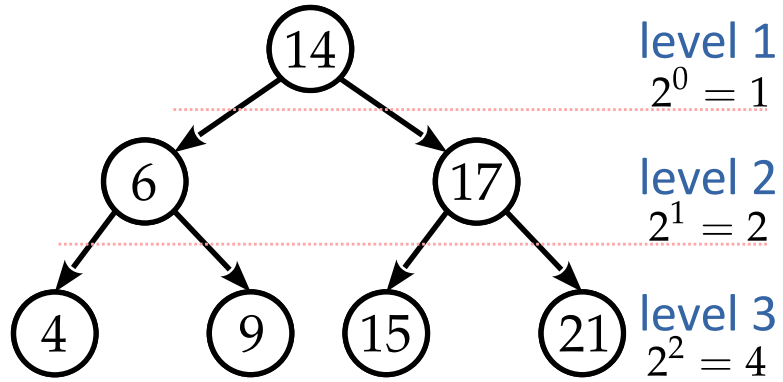
Binarna stabla (2)

- Za opis odnosa između čvorova binarnog stabla koristimo rodoslovne izraze
  - Roditelj, dijete, blizanci
  - Mogu se koristiti izrazi kao : pradjed (roditelj roditelja), ujak (blizanac roditelja)
- **Savršeno binarno stablo** (*perfect*)
  - Binarno stablo kojem su sve razine do kraja popunjene čvorovima



Binary trees (3)

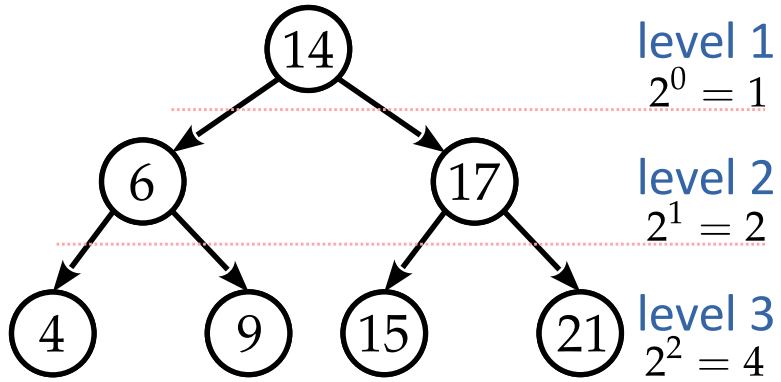
- Properties of a perfect binary tree
  - Number of nodes = $2^{h}-1$
  - Number of sheets = 2^{h-1}
  - Number of internal nodes = $2^{h-1}-1$
  - Height $h = \log_2(n+1) = \log_2 n$



- **Complete binary tree**(*complete*) – A binary tree that has all but the lowest levels completely filled with nodes. In the lowest level, the sheets are filled from the left.
  - Number of nodes ≤ $2^{h}-1$
  - Number of sheets ≤ 2^{h-1}
  - Number of internal nodes ≤ $2^{h-1}-1$

Binarna stabla (3)

- Svojstva savršenog binarnog stabla
  - Broj čvorova $n = 2^h - 1$
  - Broj listova $l = 2^{h-1}$
  - Broj unutarnjih čvorova $i = 2^{h-1} - 1$
  - Visina $h = log_2(n+1) = log_2 2^h$



- **Kompletno binarno stablo** (*complete*) – Binarno stablo koje ima sve razine, osim najdonje, potpuno popunjene čvorovima. U najdonjoj razini listovi se popunjavaju s lijeve strane.
  - Broj čvorova $n \leq 2^h - 1$
  - Broj listova $l \leq 2^{h-1}$
  - Broj unutarnjih čvorova $i \leq 2^{h-1} - 1$

Binary trees (4)

- The height of the complete binary tree - directly related to the complexity of the search

$$h = \lceil \log_2(n + 1) \rceil$$

  - It is also valid

$$n = 2^h - 1 \leq 2^h - 1$$

- **Full binary tree**(*full*) – A binary tree whose internal nodes have exactly two children.
- Organization**sorted**binary tree $= (L, k, R)$
  - No duplicates $\forall k_L \in L, k_R \in R: key(k_L) < key(k) < key(k_R)$
  - With duplicates $\forall k_L \in L, k_R \in R: key(k_L) \leq key(k) < key(k_R)$

Binarna stabla (4)

- Visina kompletnog binarnog stabla – direktno vezano uz kompleksnost pretraživanja

$$h = \lceil log_2(n+1) \rceil$$

  - Vrijedi i

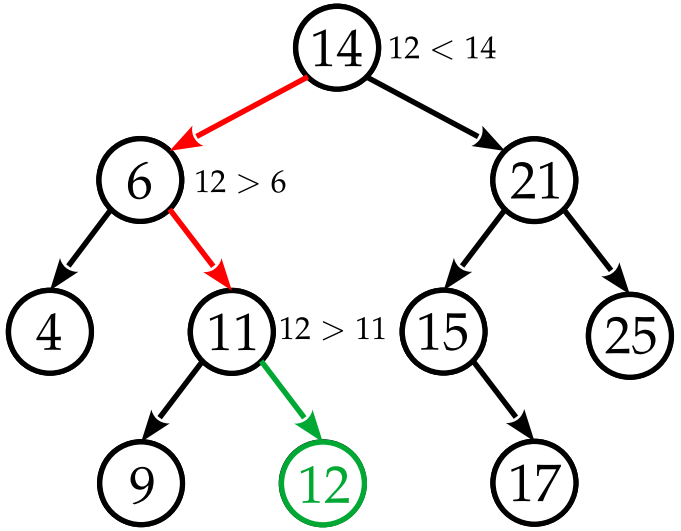
$$n = i + l \leq 2^h - 1$$

- **Puno binarno stablo** (*full*) – Binarno stablo čiji unutarnji čvorovi imaju točno dva djeteta.
- Organizacija **sortiranog** binarnog stabla $B = (L, S, R)$
  - Bez duplikata $v\big(S(L)\big) < v(S) < v\big(S(R)\big)$
  - Sa duplikatima

$$v\big(S(L)\big) \leq v(S) < v\big(S(R)\big)$$
$$v\big(S(L)\big) < v(S) \leq v\big(S(R)\big)$$

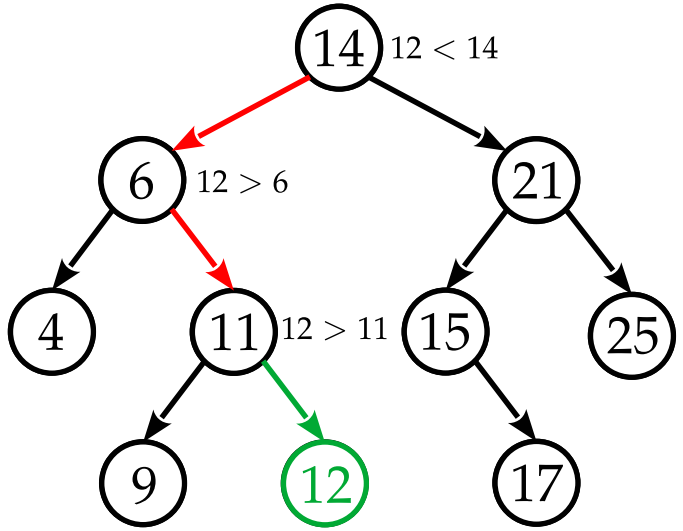
Binary tree operations (1)

- **Let's repeat**adding values to a binary tree
  - Adding is preceded by a search
  - Upon encountering a free place, we add a new node according to the organization of the binary tree:
    - Left if smaller
    - To the right if it's bigger



Operacije nad binarnim stablom (1)

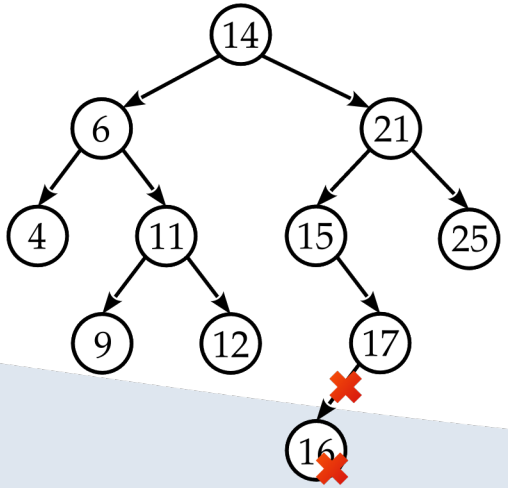
- **Ponovimo** dodavanje vrijednosti u binarno stablo
  - Dodavanju prethodi pretraživanje
  - Nailaskom na slobodno mjesto, dodajemo novi čvor prema organizaciji binarnog stabla:
    - Lijevo ako je manji
    - Desno ako je veći

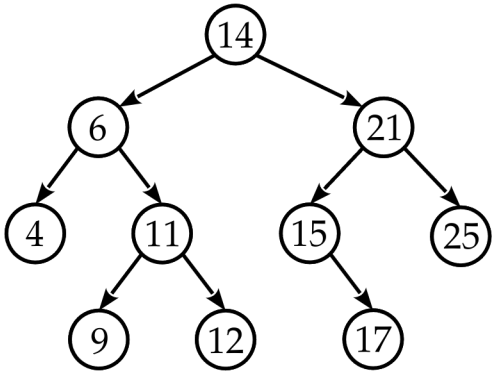


Deleting Nodes (1)

- First we find the node we are deleting
- Three cases:
  1. The node to be deleted is a leaf
  2. The node being deleted has one child
  3. The node being deleted has both children

1. The node to be deleted is a leaf – we just delete the node

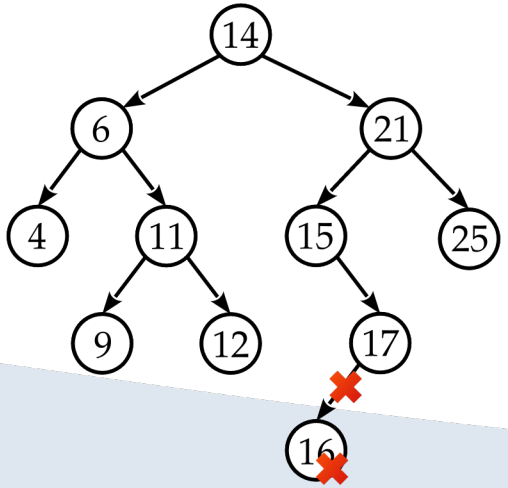


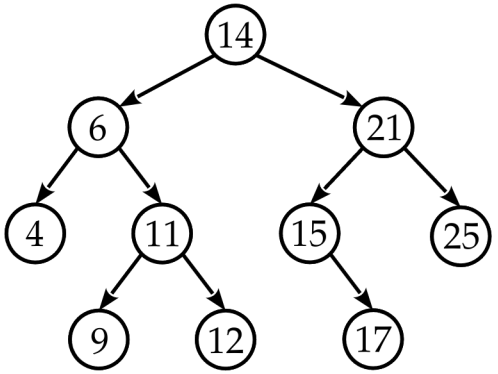


Brisanje čvorova (1)

- Prvo pronađemo čvor koji brišemo
- Tri slučaja:
  1. Čvor koji se briše je list
  2. Čvor koji se briše ima jedno dijete
  3. Čvor koji se briše ima oba djeteta

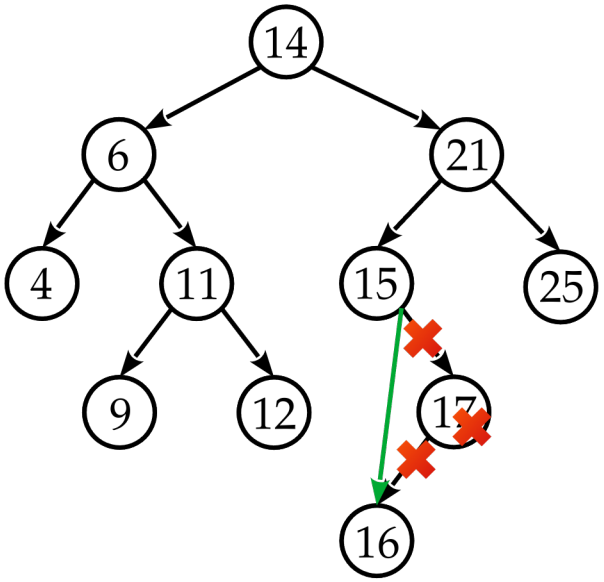
1. Čvor koji se briše je list – samo obrišemo čvor

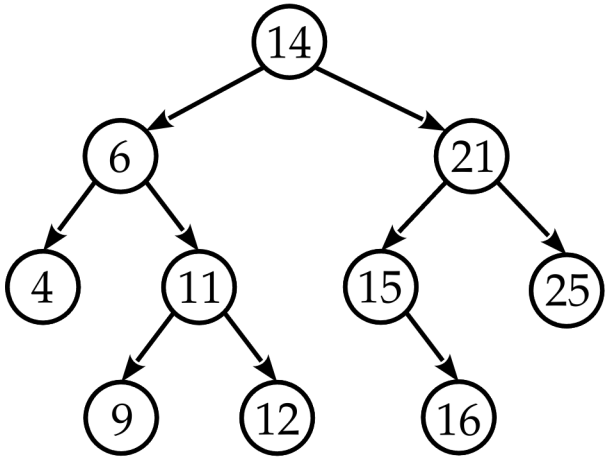




Deleting nodes (2)

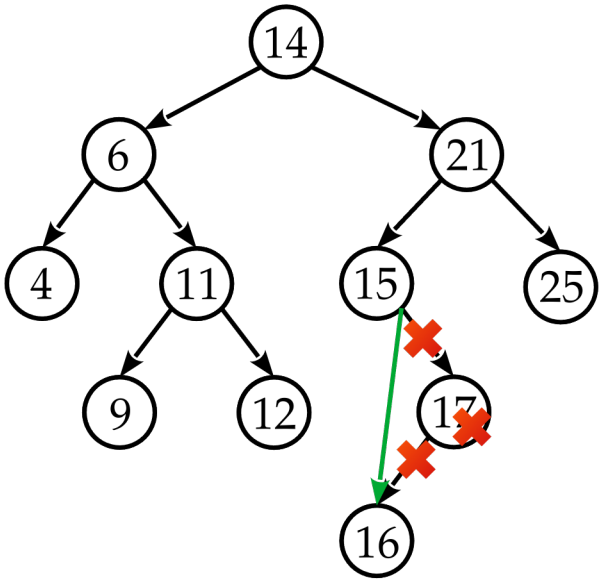
2. The node has one child – That child becomes the new child of the parent of the node being deleted

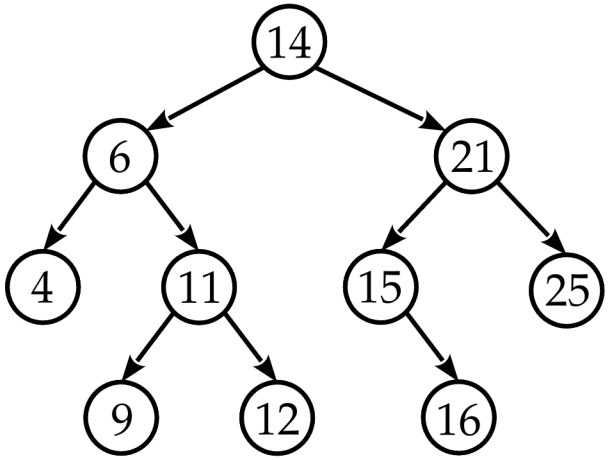




Brisanje čvorova (2)

2. Čvor ima jedno dijete – To dijete postaje novo dijete roditelja čvora koji se briše





Deleting nodes (3)

3. The node has two children - options
    - Delete by merging – significantly changes the structure
    - Delete by copy – the structure changes minimally

- Deletion by union
    - Find the node being deleted and its parent (if the root node is not being deleted)
    - Let's determine on which side the node that is being deleted is in relation to its parent
        - If the root node is in question, then it does not matter which subtree we take as the union subtree (*merging subtree*)
        - Otherwise we have two options:
            - If the node to be deleted is in**the left**subtree of the parent, then the union tree is its right subtree
            - If the node to be deleted is in**with the right**subtree of the parent, then the union tree is its left subtree

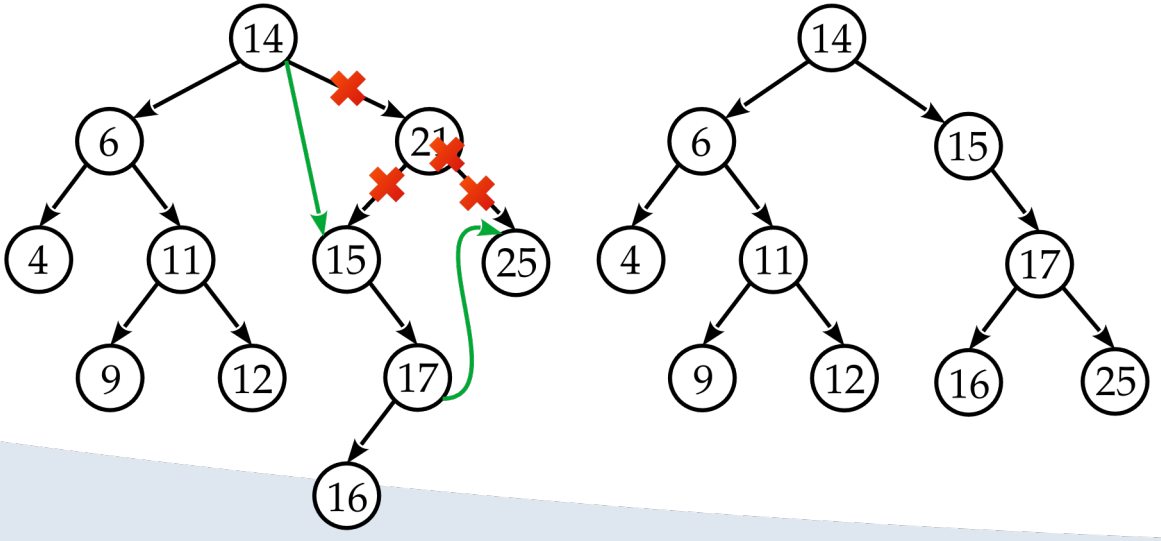
Brisanje čvorova (3)

3. Čvor ima dva djeteta – opcije
   - Brisanje sjedinjenjem (delete by merging) – znatno mijenja strukturu
   - Brisanje kopiranjem (delete by copy) – struktura se minimalno mijenja

- Brisanje sjedinjenjem
  - Naći čvor koji se briše i njegovog roditelja (ako se ne briše korijenski čvor)
  - Utvrdimo na kojoj strani je čvor koji se briše u odnosu na svojeg roditelja
    - Ako je korijenski čvor u pitanju, onda je svejedno koje podstablo uzimamo za podstablo sjedinjenja (*merging subtree*)
    - Inače imamo dvije opcije:
      - Ako je čvor koji se briše u **lijevom** podstablu roditelja, tada je postablo sjedinjenja njegovo desno podstablo
      - Ako je čvor koji se briše u **desnom** podstablu roditelja, tada je postablo sjedinjenja njegovo lijevo podstablo

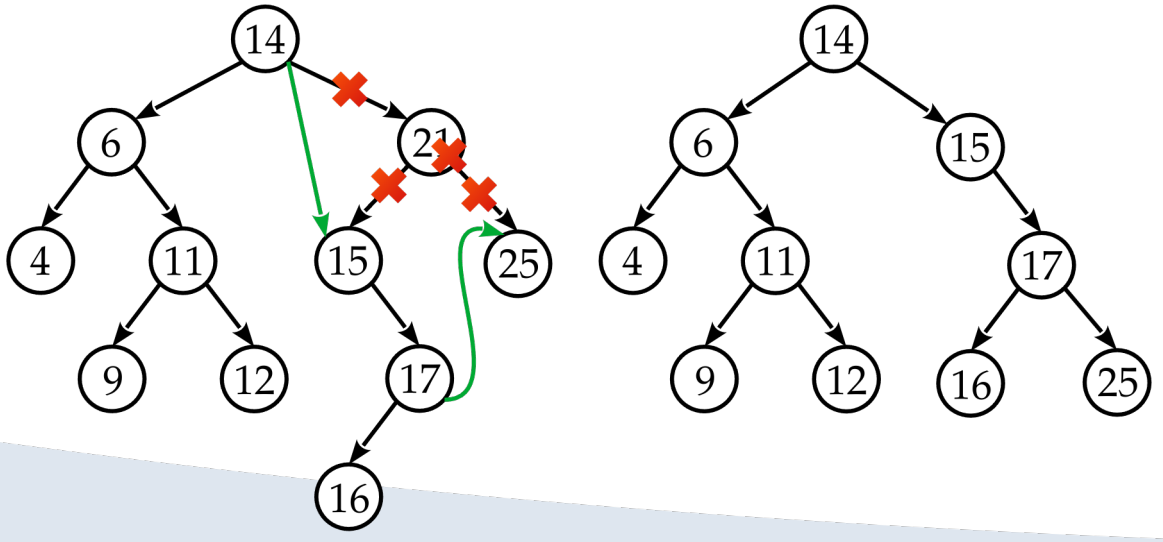
Deleting nodes (4)

- Deleting node A by union
  - We connect the twin node to the root node of the union subtree at
    - Knot**follower**if the subtree of the union was the right subtree of node A
    - Knot**predecessor**if the subtree of the union was the left subtree of node A
  - We connect the root of the union subtree with the parent of the deleted node
    - Except in the case when the root node is deleted



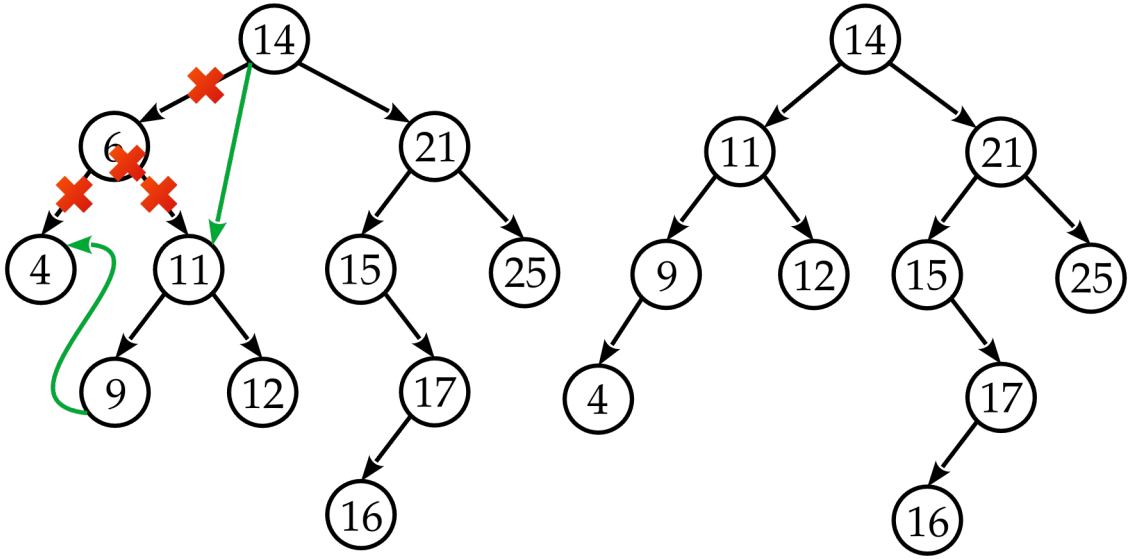
Brisanje čvorova (4)

- Brisanje čvora A sjedinjenjem
  - Čvor blizanac korijenskom čvoru podstabla sjedinjenja spajamo na
    - Čvor **sljedbenik** ako je podstablo sjedinjenja bilo desno podstablo čvora A
    - Čvor **prethodnik** ako je podstablo sjedinjenja bilo lijevo podstablo čvora A
  - Korijen podstabla sjedinjenja spajamo s roditeljem čvora koji se obrisao
    - Osim u slučaju kada se briše korijenski čvor



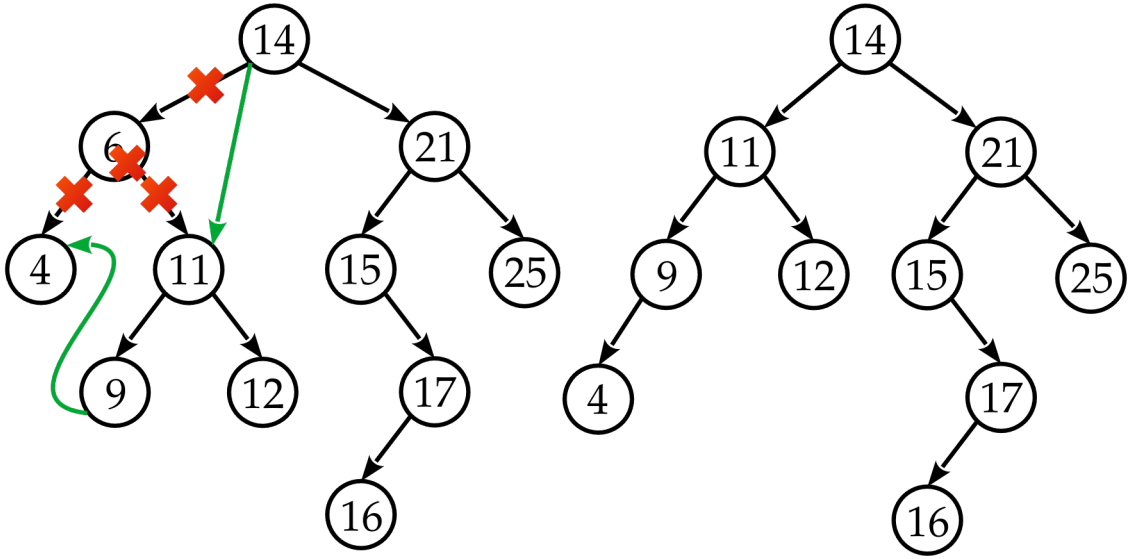
Deleting nodes (5)

- Deletion by union



Brisanje čvorova (5)

- Brisanje sjedinjenjem



Deleting nodes (6)

- Delete by copying
  - It boils down to deleting a node without children or with one child
  - A replacement node is found and deleted - the binary tree structure is minimally changed

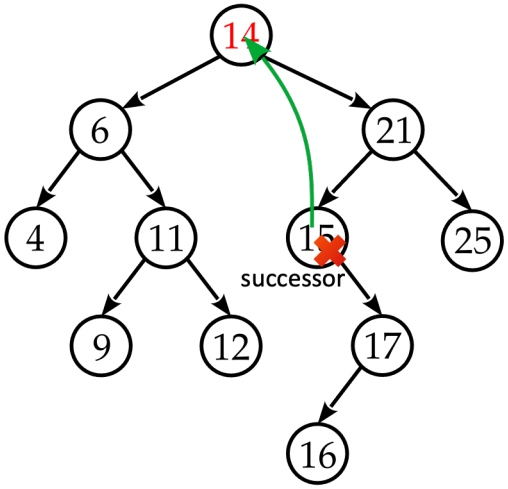
1. We find the node A that we are deleting
2. We find the node X that contains a direct predecessor or successor – a replacement node
   - The predecessor is the rightmost node in the left subtree of A
   - A follower is the leftmost node in the right subtree of A
3. We copy the value of the replacement node X to the node A that is being deleted
4. The replacement node X is removed

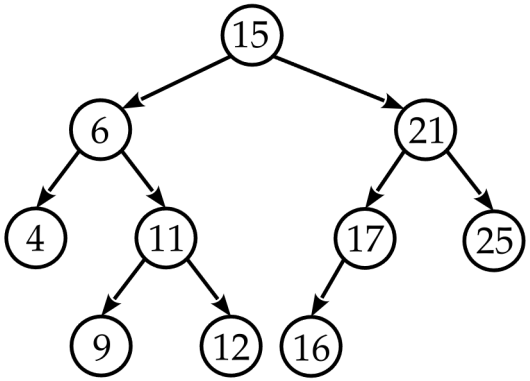
Brisanje čvorova (6)

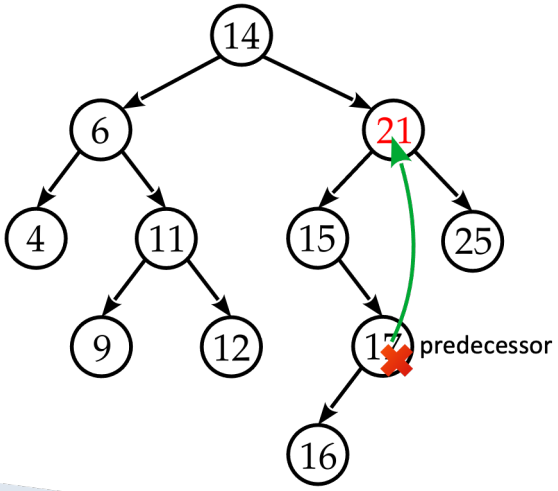
- Brisanje kopiranjem
  - Svodi se na brisanje čvora bez djece ili s jednim djetetom
  - Pronađe se zamjenski čvor koji se obriše - minimalno se mijenja struktura binarnog stabla

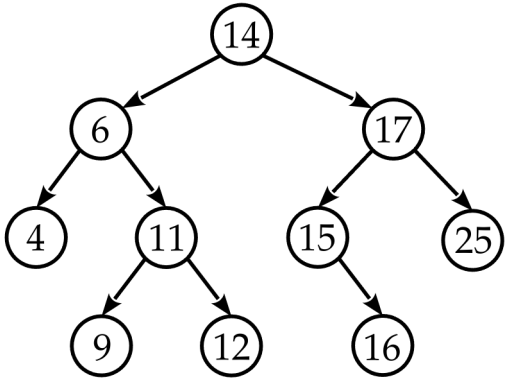
1. Pronađemo čvor A koji brišemo
2. Pronađemo čvor X koji sadrži direktnog prethodnika ili sljedbenika – zamjenski čvor
   - Prethodnik je najdesniji čvor u lijevom podstablu A
   - Sljedbenik je najljeviji čvor u desnom podstablu A
3. Prekopiramo vrijednost zamjenskog čvora X u čvor A koji se briše
4. Zamjenski čvor X se ukloni

Deleting nodes (7)

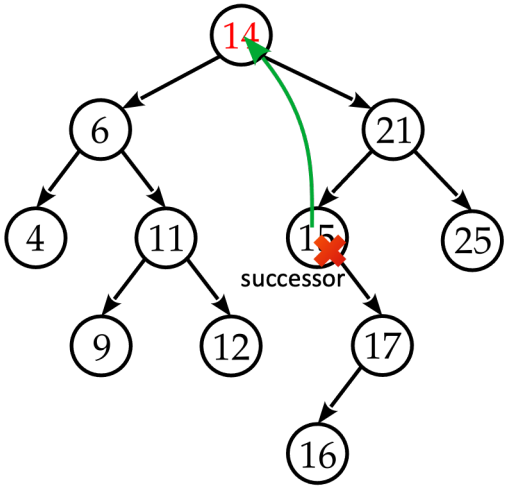


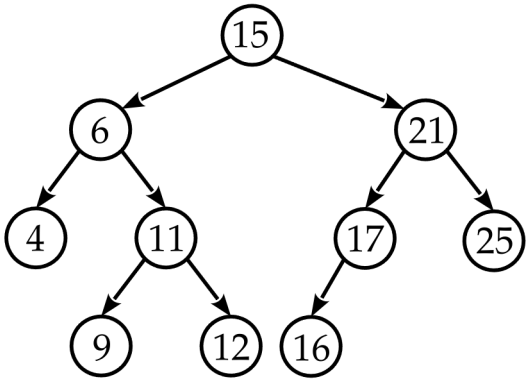


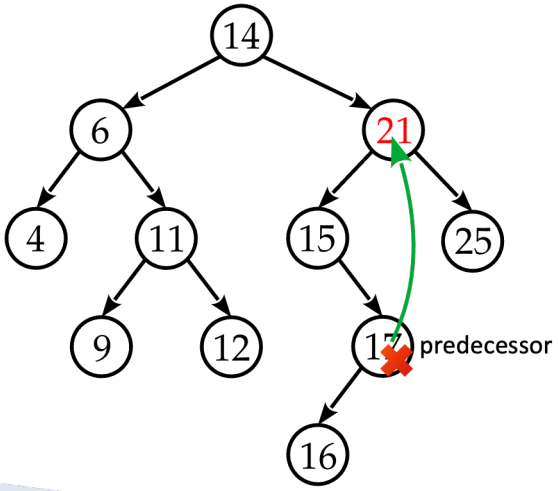


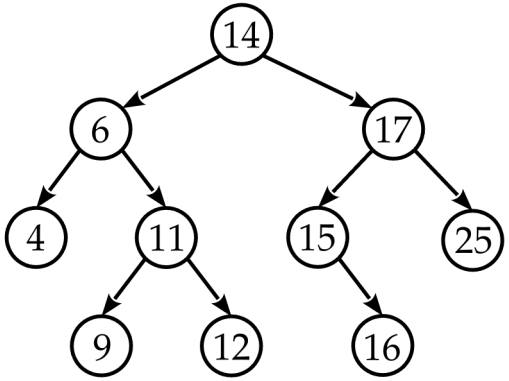


Brisanje čvorova (7)



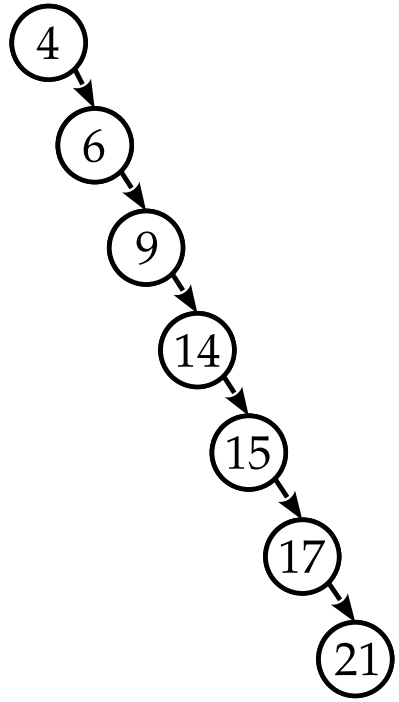






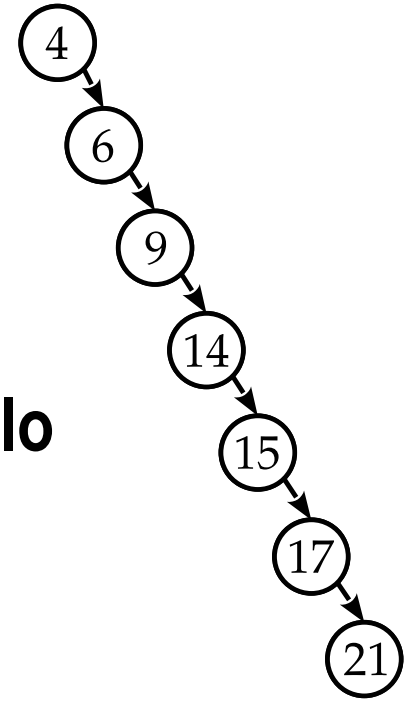
Balanced Binary Tree (1)

- Why is a balanced binary tree interesting to us?
- We want to achieve a search complexity of
  (!)
- The other extreme -**degenerate (oblique) binary tree** (*degenerates*) – example on the right
  - The complexity is ( )



Uravnoteženo binarno stablo (1)

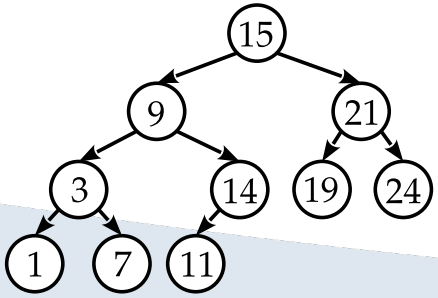
- Zašto nam je uravnoteženo binarno stablo zanimljivo?
- Želimo postići složenost pretraživanja od $O(log_2 n)$
- Druga krajnost – **degenerirano (koso) binarno stablo** (*degenerate*) – primjer desno
  - Složenost je $O(n)$

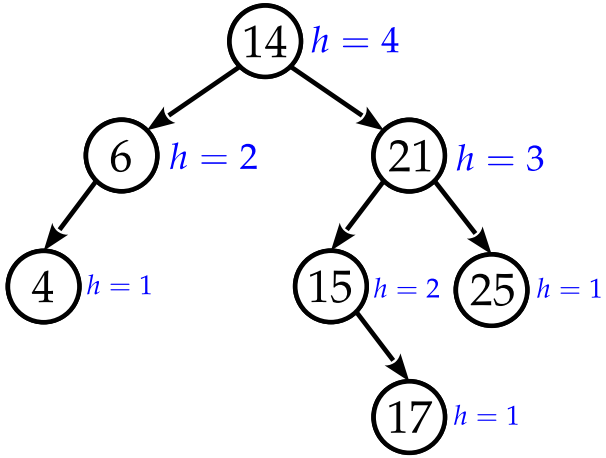


Balanced Binary Tree (2)

- For a binary tree = (, ,) the definition of a balanced tree is
  $$\forall \quad |h(\ ) - h(\ )| \leq 1$$
  - The difference in the height of the left and right subtrees **of each node** can be at most 1

- **A perfectly balanced binary tree** (*perfectly balanced*) – balanced and complete
  - All levels except the last one are completely filled with nodes

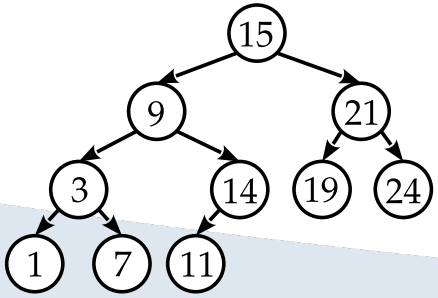


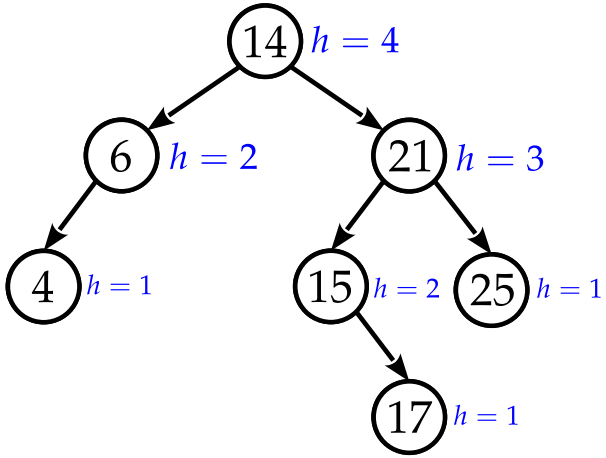


Uravnoteženo binarno stablo (2)

- Za binarno stablo $B = (L, S, R)$ definicija uravnoteženog stabla je
$$\forall S \quad |h(L) - h(R)| \leq 1$$
  - Razlika visina lijevog i desnog podstabla **svakog čvora** smije biti najviše 1

- **Savršeno uravnoteženo binarno stablo** (*perfectly balanced*) – uravnoteženo i kompletno
  - Sve razine osim zadnje su posve popunjene čvorovima



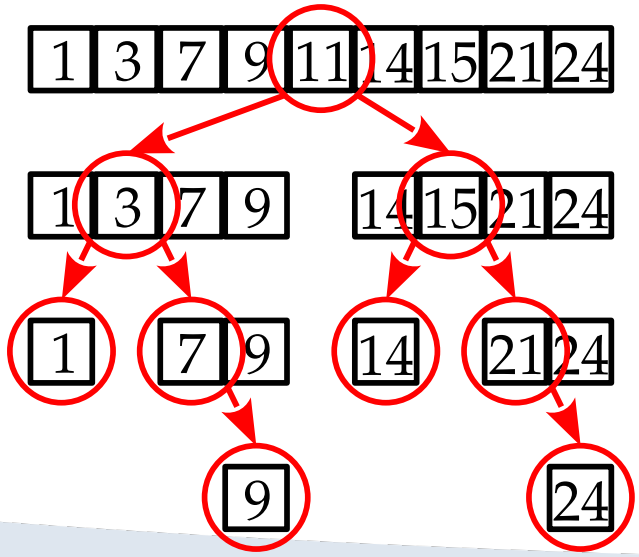


Creating a binary tree (1)
(from a sorted array of values)

- We have a sorted field available (*array*) values
  "= 1,3,7,9,11,14,15,21,24 ⟩

1. We find the positional mean value *c* in the field
2. Let's create a root value node *c* of the current binary tree
3. For the subfield to the left of *c* the left subtree is recursively created
4. For the subfield to the right of *c* the right subtree is recursively created
5. We repeat until we can create a left or right subtree

```
function CreateBalancedTree(V_s)
    n ← |S|
    if n > 0 then
        i ← (n ÷ 2) + (n % 2)
        root ← create node having value V_s[i − 1]
        leftChild(root) ←CreateBalancedTree(V_s[0, i − 1])
        rightChild(root) ←CreateBalancedTree(V_s[i, n])
        return root
    else
        return nil
```



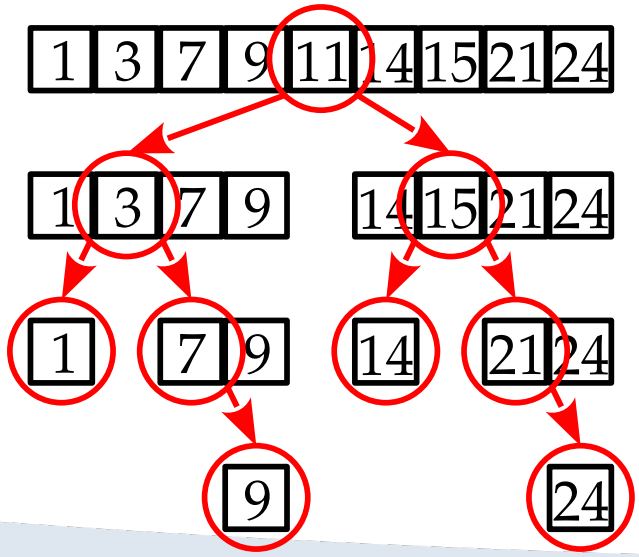
Stvaranje binarnog stabla (1)
(iz sortiranog polja vrijednosti)

- Na raspolaganju imamo sortirano polje (*array*) vrijednosti

$$V_S = \langle 1,3,7,9,11,14,15,21,24 \rangle$$

1. Pronađemo pozicijski srednju vrijednost v u polju
2. Stvorimo korijenski čvor vrijednosti v trenutnog binarnog stabla
3. Za podpolje lijevo od v se rekurzivno stvara lijevo podstablo
4. Za podpolje desno od v se rekurzivno stvara desno podstablo
5. Ponavljamo dok možemo stvoriti lijevo ili desno podstablo

```
function CreateBalancedTree(V_s)
    n ← |S|
    if n > 0 then
        i ← (n ÷ 2) + (n % 2)
        root ← create node having value V_s[i − 1]
        leftChild(root) ←CreateBalancedTree(V_s[0, i − 1])
        rightChild(root) ←CreateBalancedTree(V_s[i, n])
        return root
    else
        return nil
```



Creating a binary tree (2)
(from a sorted array of values)

- This algorithm has complexity $O(n! + n)$
  - Field sorting + traversal of all field elements
- The algorithm can only be used in special situations
  - When we have a field of values and we want to create a balanced binary tree from it
  - It is used relatively often, concrete examples will be given later in the lectures
- The binary tree created in this way is balanced

Stvaranje binarnog stabla (2)
(iz sortiranog polja vrijednosti)

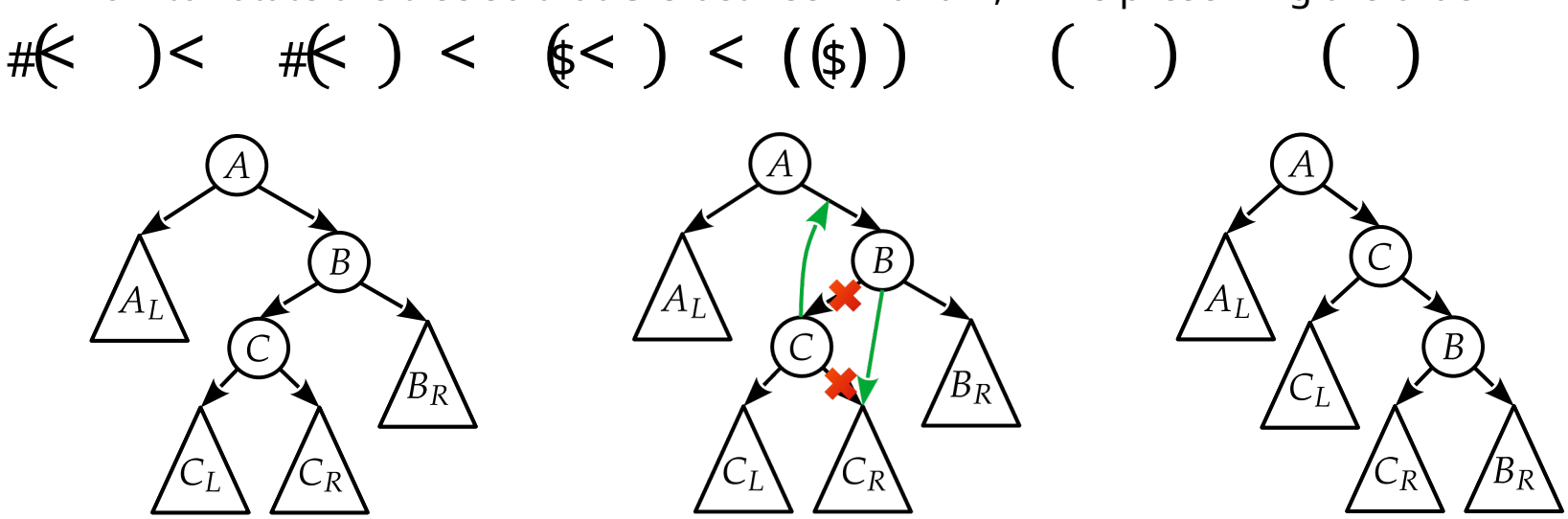
- Ovaj algoritam ima kompleksnost $O(n\ log_2 n + n)$
  - Sortiranje polja + prolaz po svim elementima polja

- Algoritam se može upotrijebiti samo u specijalnim situacijama
  - Kada imamo polje vrijednosti i od njega želimo stvoriti uravnoteženo binarno stablo
  - Koristi se relativno često, konkretni primjeri će biti kasnije u predavanjima

- Ovako stvoreno binarno stablo je uravnoteženo

Rotations in the tree (1) - right

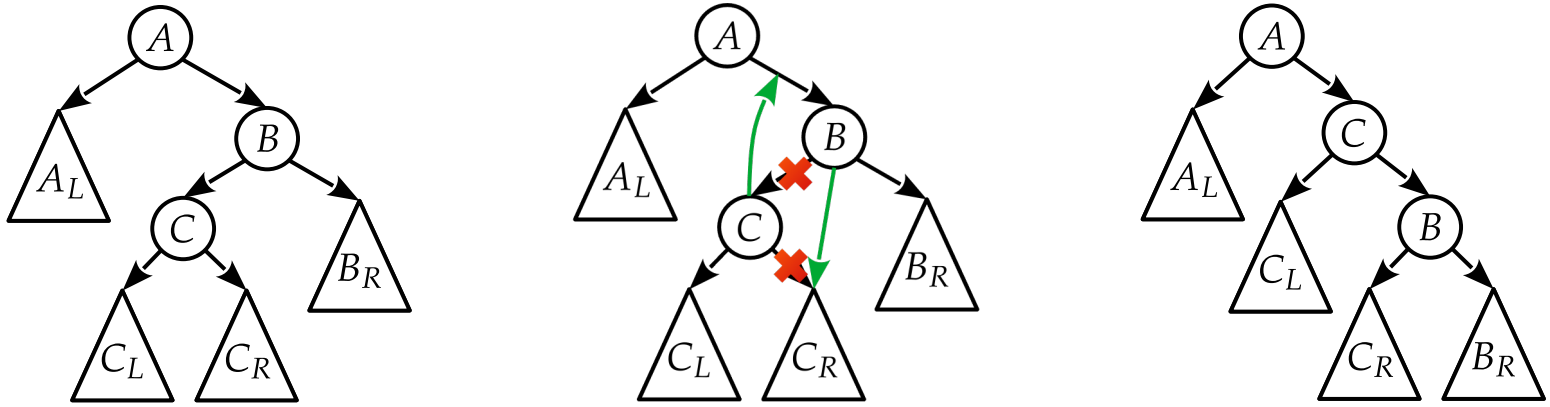
- To balance trees we need two operations (rotations) on binary trees (pulley analogy)
- Right rotation of C about B
  - How to rotate the tree so that C is between A and B, while preserving the order



Rotacije u stablu (1) - desna

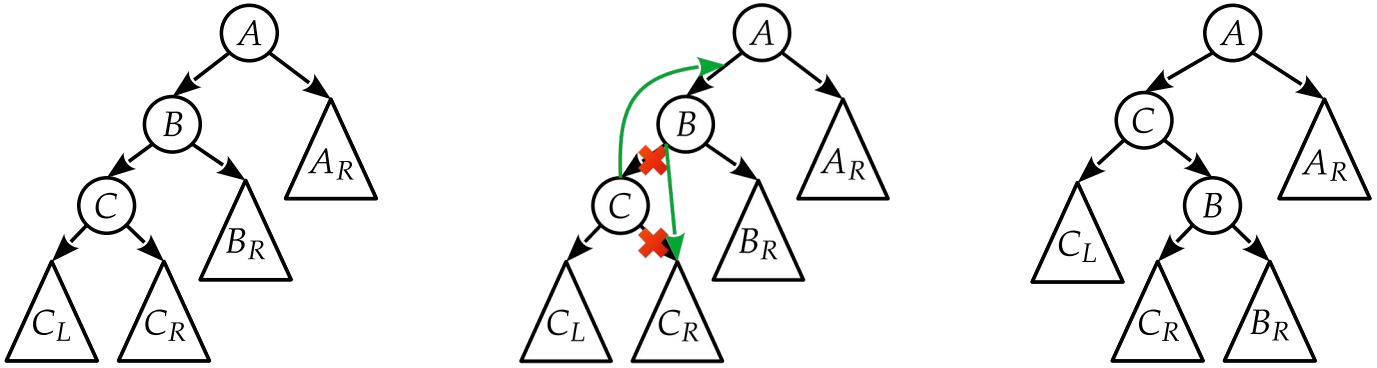
- Za uravnotežavanje stabala trebamo dvije operacije (rotacije) nad binarnim stablima (analogija koloture)
- Desna rotacija C oko B
  - Kako rotirati stablo da C bude između A i B, a da se sačuva poredak

$$v(A_L) < v(A) < v(C_L) < v(C) < v(C_R) < v(B) < v(B_R)$$

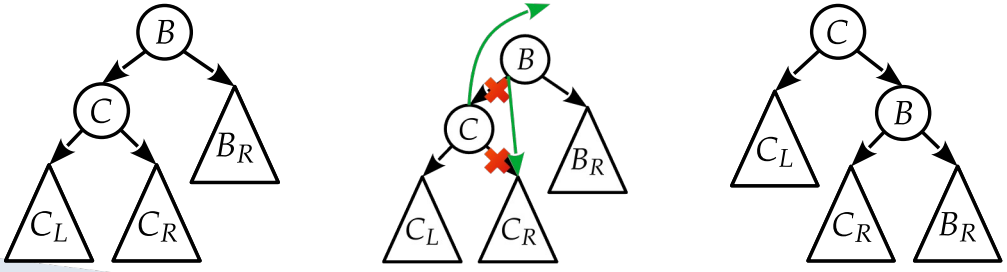


Rotations in the tree (2) - right

- Another right rotation of C around B

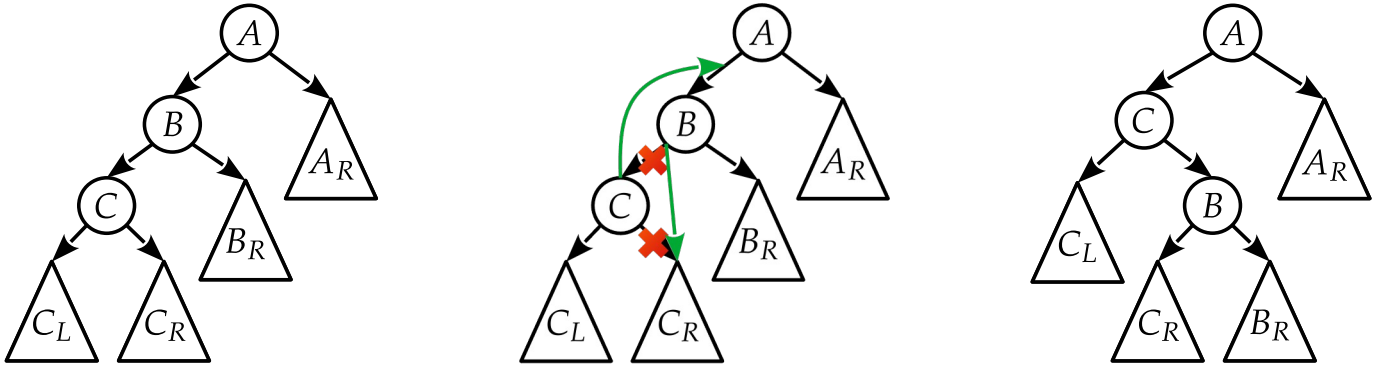


  - The right child of C becomes the left child of B
  - B becomes the right child of C
  - C becomes a child of the former parent of node B (if it exists)

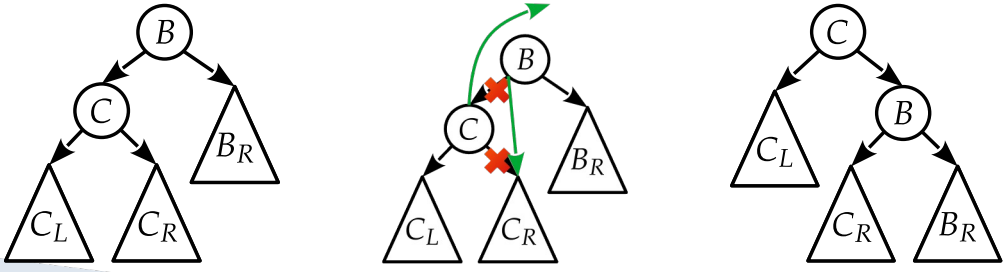


Rotacije u stablu (2) - desna

- Još jedna desna rotacija C oko B

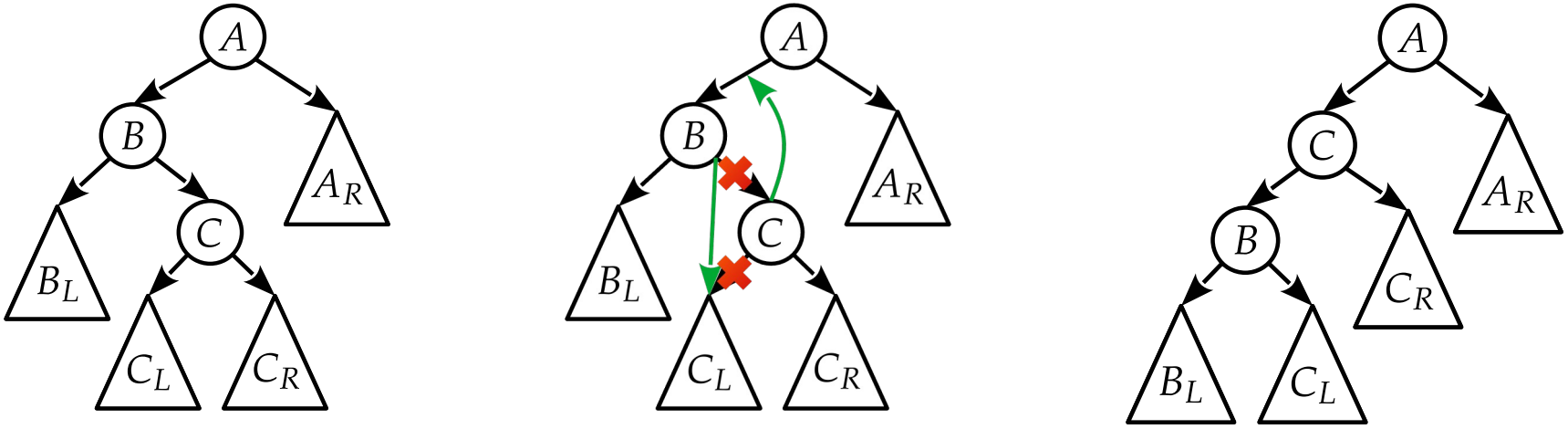


  - Desno dijete od C postaje lijevo dijete od B
  - B postaje desno dijete od C
  - C postaje dijete od bivšeg roditelja čvora B (ako postoji)



Rotations in the tree (3) - left

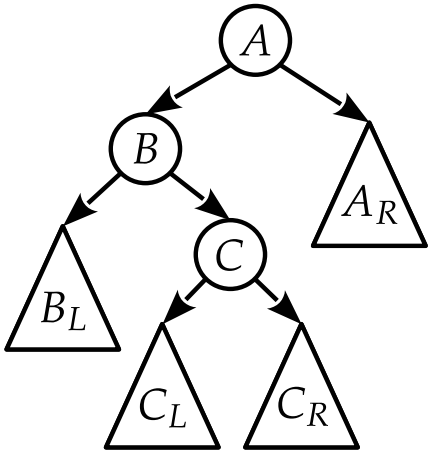
- Left rotation of C around B
  - How to rotate the tree so that C is between A and B, while preserving the order

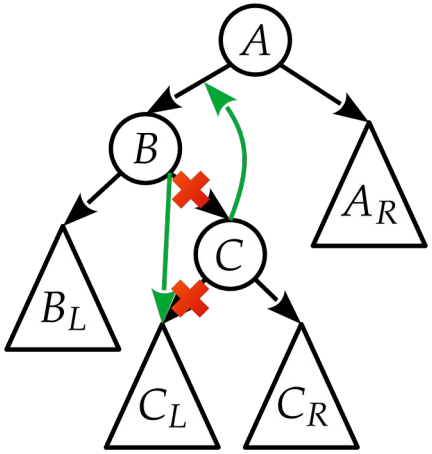


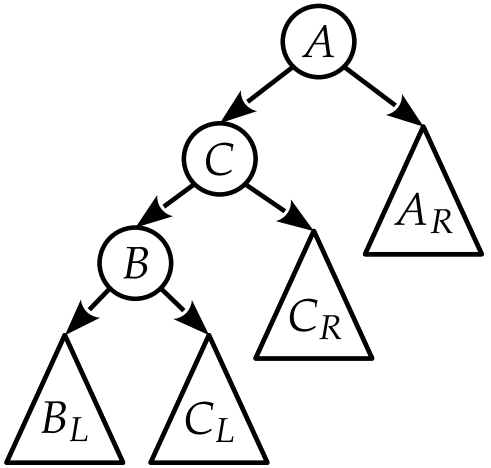
Rotacije u stablu (3) - lijeva

- Lijeva rotacija C oko B
  - Kako rotirati stablo da C bude između A i B, a da se sačuva poredak

$$v(B_L) < v(B) < v(C_L) < v(C) < v(C_R) < v(A) < v(A_R)$$

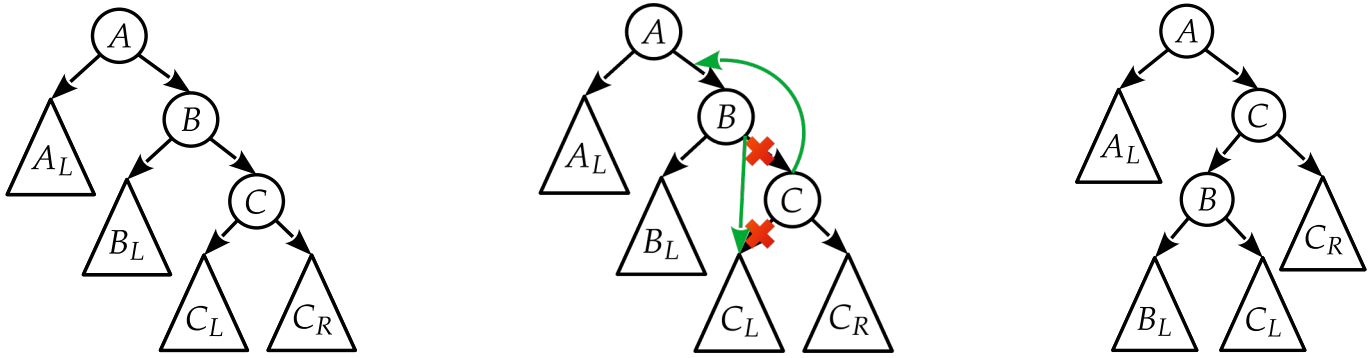




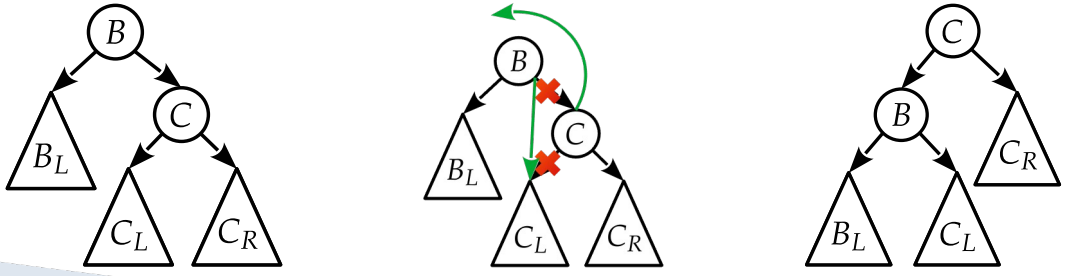


Rotations in the tree (4) - left

- Another left rotation of C around B

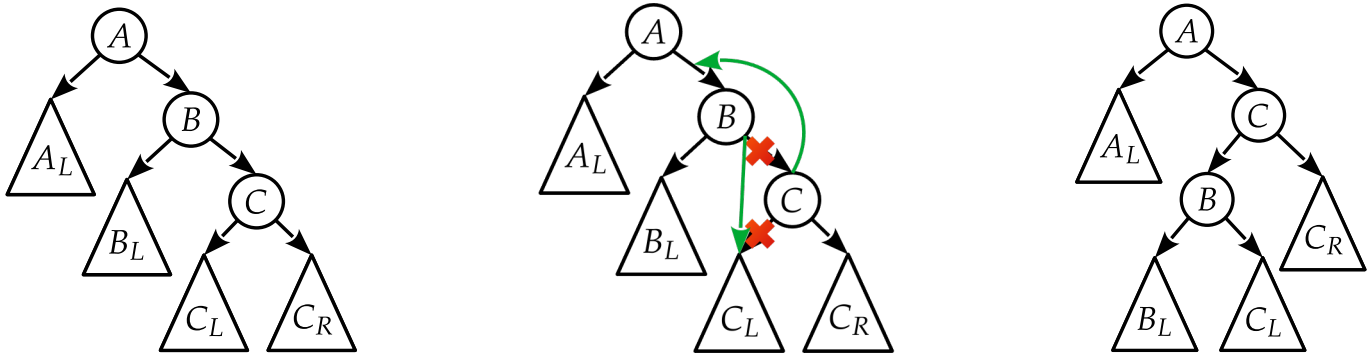


  - The left child of C becomes the right child of B
  - B becomes the left child of C
  - C becomes a child of the former parent of node B (if it exists)

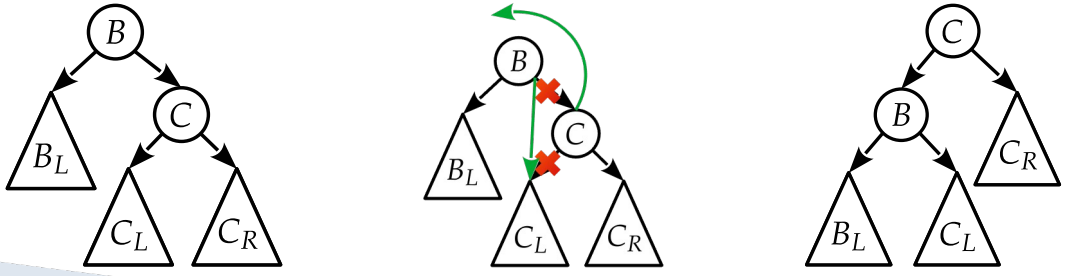


Rotacije u stablu (4) - lijeva

- Još jedna lijeva rotacija C oko B



  - Lijevo dijete od C postaje desno dijete od B
  - B postaje lijevo dijete od C
  - C postaje dijete od bivšeg roditelja čvora B (ako postoji)



Day-Stout-Warren algorithm (DSW)

- Two phases of the algorithm

1. Making the spine (oblique tree)

2. Recursively breaking the spine back into a complete tree

Day-Stout-Warren algoritam (DSW)

- Dvije faze algoritma

1. Izrada kralježnice (koso stablo)

2. Rekurzivno lomljenje kralježnice nazad u kompletno stablo

DSW – spine manufacturing

```
procedure RightBackbone(root)
    B ← root
    A ← nil
    while B ≠ nil do
        C ← leftChild(B)
        if C ≠ nil then
            RightRotate(A, B)
            if A = nil then
                root ← C
            B ← C
        else
          ▷ Descending right to the first node that has the left child
            A ← B
            B ← rightChild(B)
```

- The first step is to create a backbone from a binary tree - eg right spine
- We rotate the left children of the nodes to the right
- We repeat right rotations until there are no left children

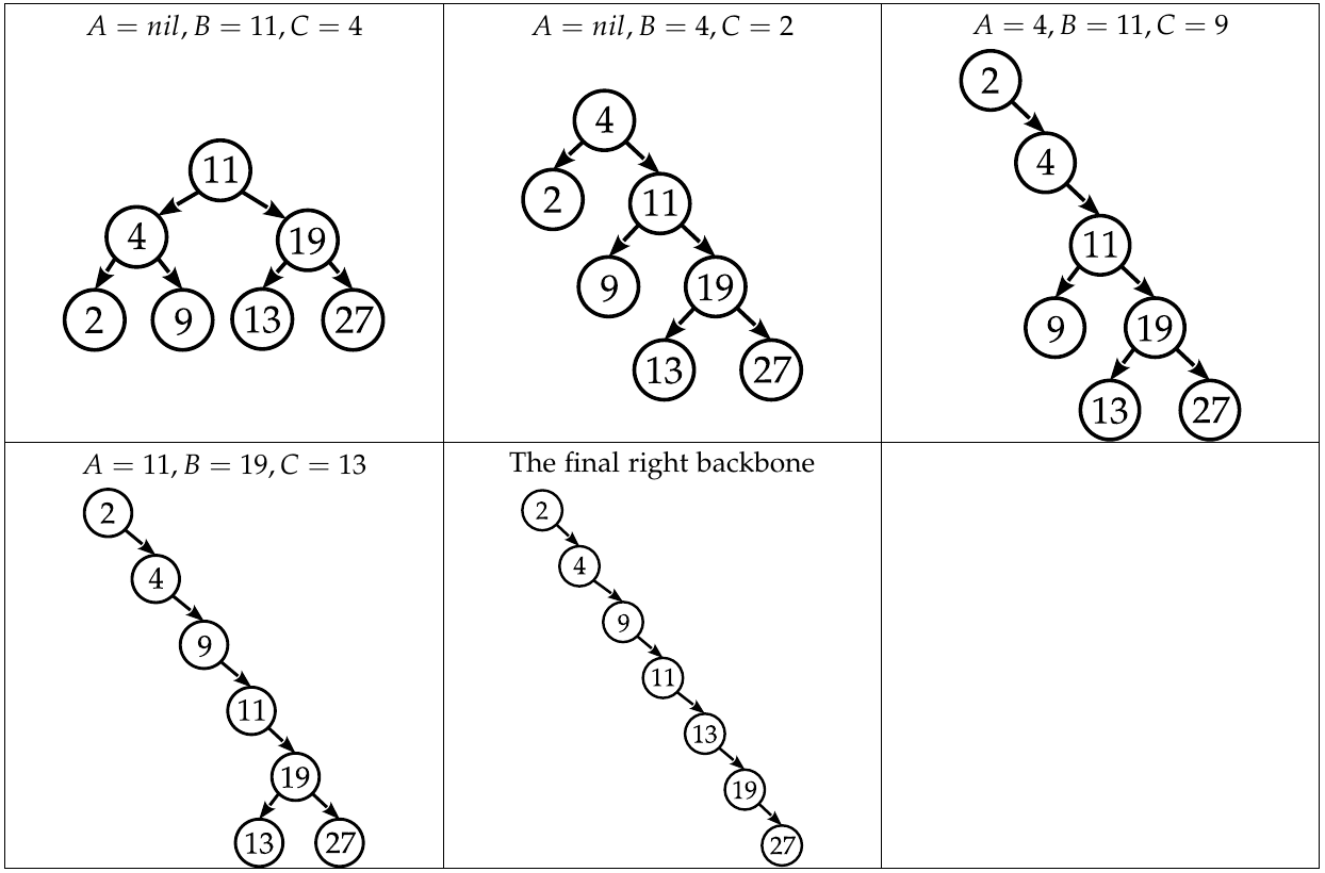
DSW – izrada kralježnice

```
procedure RightBackbone(root)
    B ← root
    A ← nil
    while B ≠ nil do
        C ← leftChild(B)
        if C ≠ nil then
            RightRotate(A, B)
            if A = nil then
                root ← C
            B ← C
        else
          ▷ Descending right to the first node that has the left child
            A ← B
            B ← rightChild(B)
```

- Prvi korak je stvaranje kralježnice od binarnog stabla – npr. desna kralježnica
- Lijevu djecu čvorova rotiramo desno
- Ponavljamo desne rotacije dok nema lijeve djece

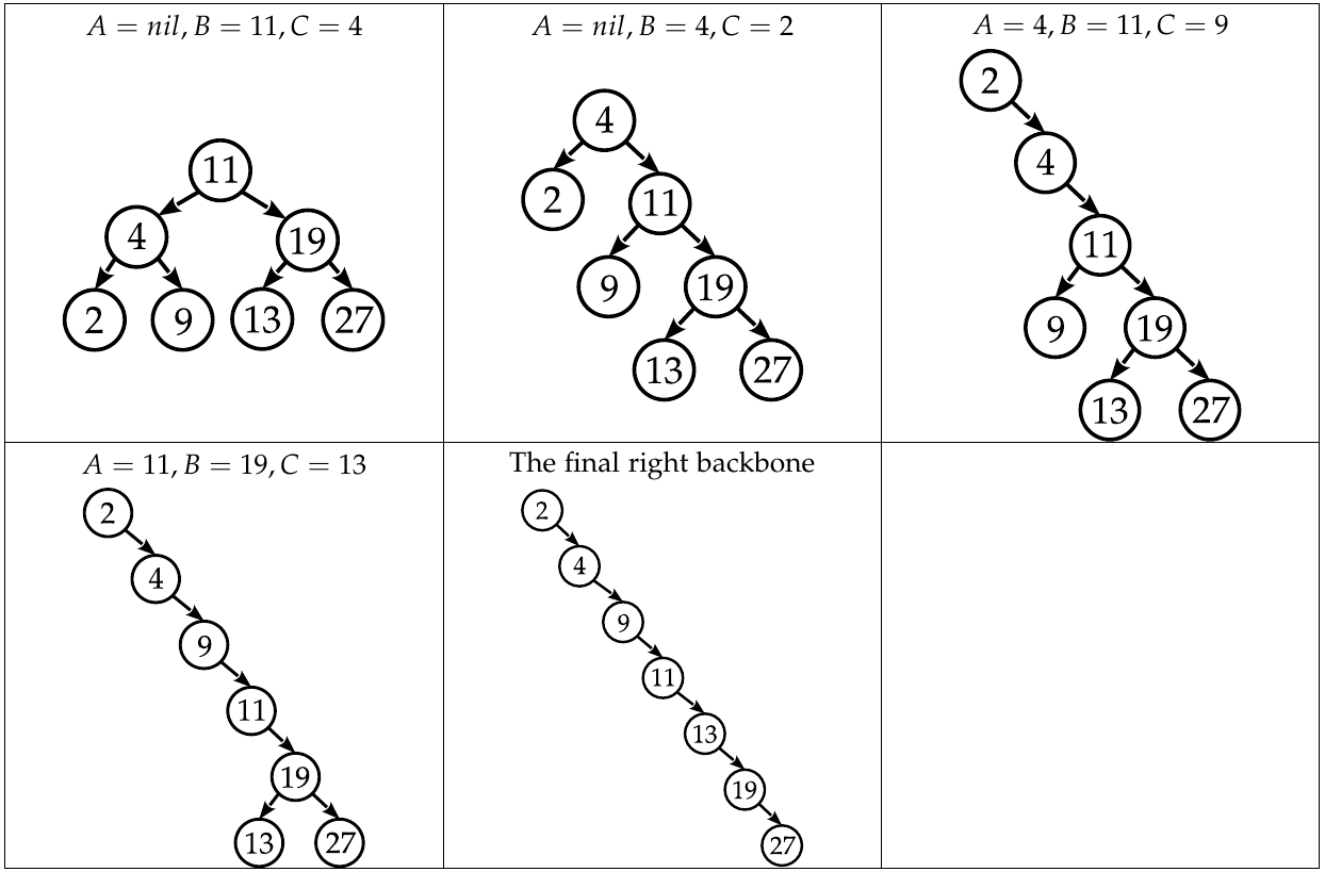
DSW – spine manufacturing

- Example



DSW – izrada kralježnice

- Primjer



DSW - breaking

- Strategically positioned left rotations for a perfectly balanced binary tree

```
procedure DSW(tree, n)
    h ← ⌈log2(n + 1)⌉
    i ← 2^(h−1) − 1
    perform n − i rotations of every second node from the root
    while i > 1 do
        i ← ⌊i/2⌋
        perform i rotations of every second node from the root
```

  - h – height of the binary tree for n nodes
  - i – number of internal nodes
  - For the right spine, we do left rotations to bring the nodes back into the binary tree structure

DSW - lomljenje

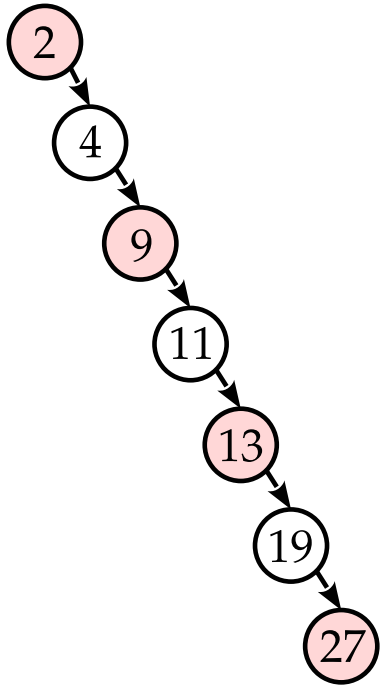
- Strateški pozicionirane lijeve rotacije za savršeno uravnoteženo binarno stablo

```
procedure DSW(tree, n)
    h ← ⌈log2(n + 1)⌉
    i ← 2^(h−1) − 1
    perform n − i rotations of every second node from the root
    while i > 1 do
        i ← ⌊i/2⌋
        perform i rotations of every second node from the root
```

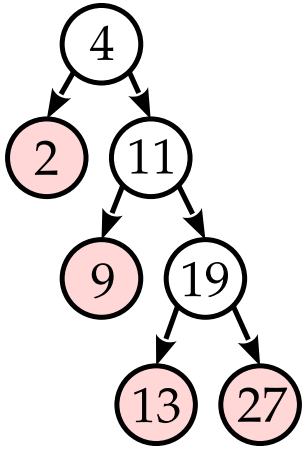
  - h – visina binarnog stabla za n čvorova
  - i – broj unutarnjih čvorova
  - Za desnu kralježnicu radimo lijeve rotacije kako bismo čvorove vratili u strukturu binarnog stabla

DSW – breaking, example

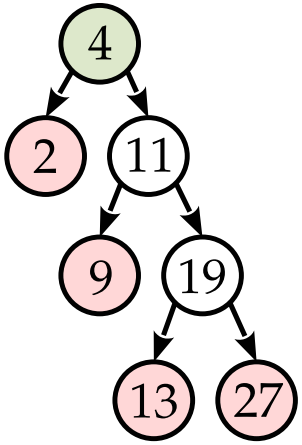
= 7
h = 3
= 3



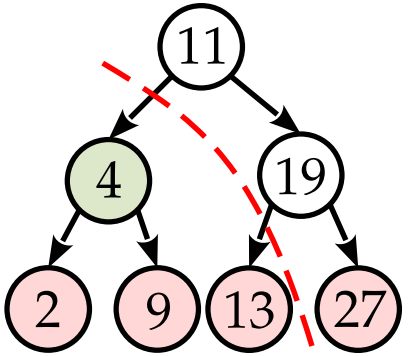
– = 4
left rotations
for leaves



⌊/2 ⫞ 3/2⫟ = 1⌋
of left rotations for
internal nodes

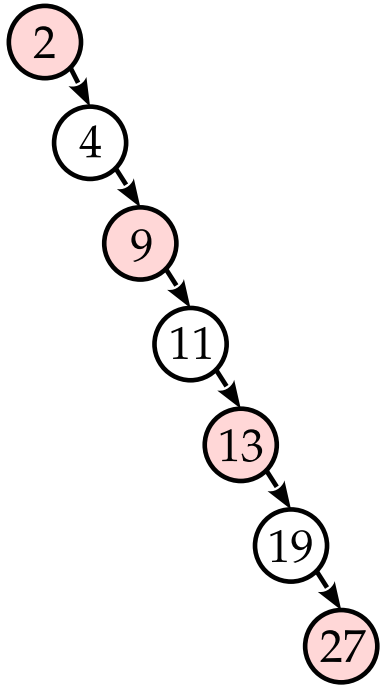


it is obtained
perfectly binary
tree

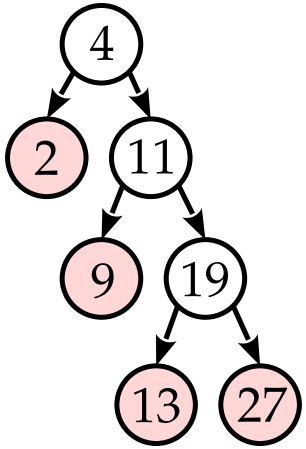


DSW – lomljenje, primjer

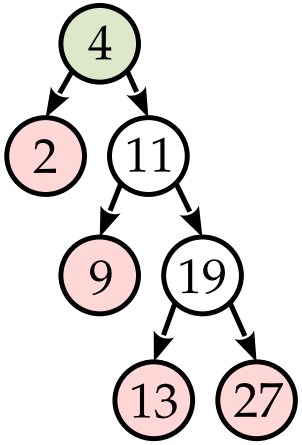
$$n = 7$$
$$h = 3$$
$$i = 3$$



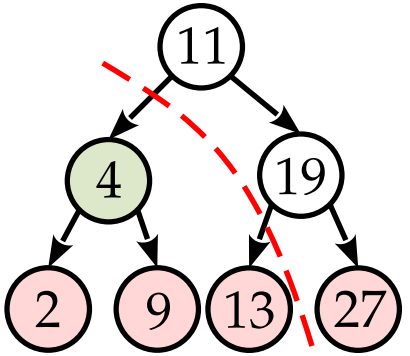
$n - i = 4$
lijevih rotacija
za listove



$\lfloor i/2 \rfloor = \lfloor 3/2 \rfloor = 1$
lijevih rotacija za
interne čvorove



dobije se
savršeno binarno
stablo



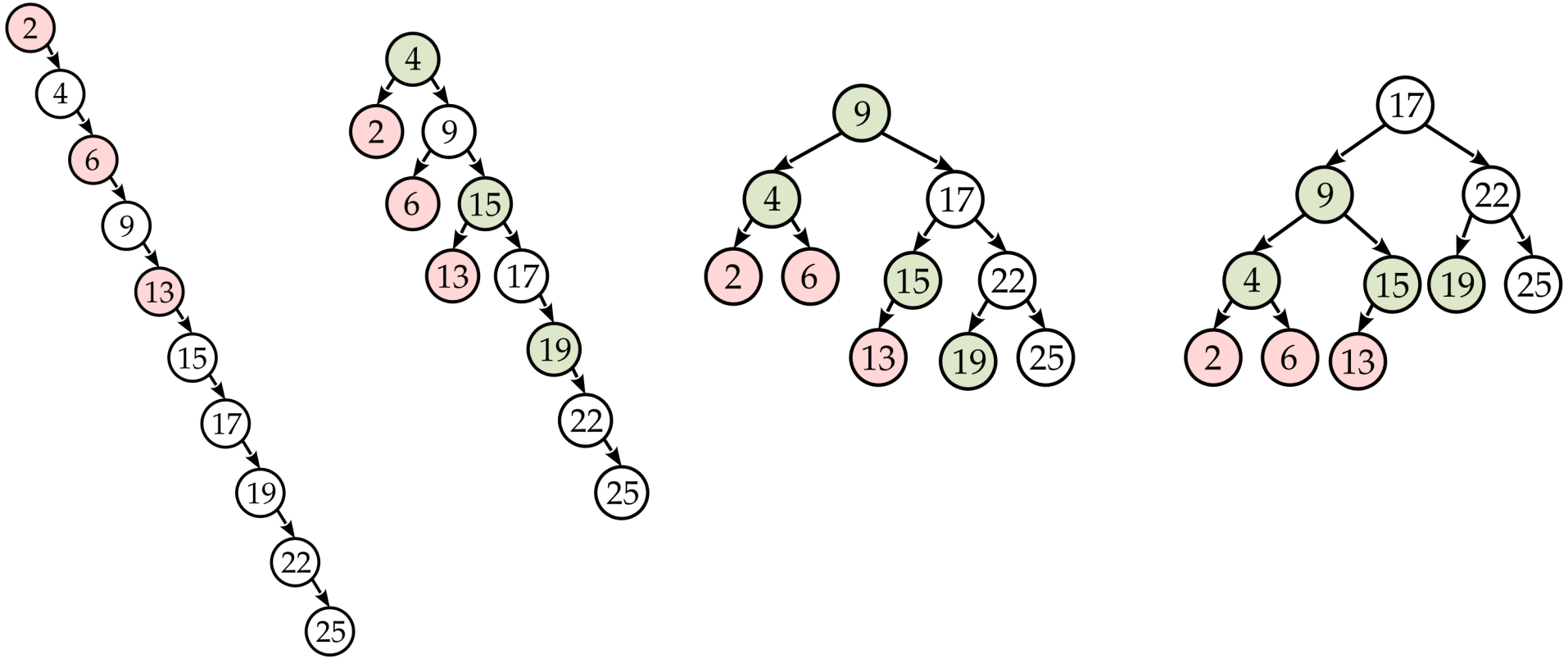
DSW – breaking, example

- Example
  - = 10
  - It is$h$ = 4levels
  - The total number of internal nodes is = 7
  - Nodes in the lowest level is10 − 7 = 3 –first we do 3 left rotations of every other node of the right spine, starting from the root node
  - Then we work$\lfloor 7/2 \rfloor = 3$left rotations of every other node of the rest of the right spine, starting from the root node, which gives us the lowest level of internal nodes
  - Then we work$\lfloor 3/2 \rfloor = 1$left rotations of every other node of the rest of the right spine, starting from the root node, which gives us the lowest level of internal nodes

DSW – lomljenje, primjer

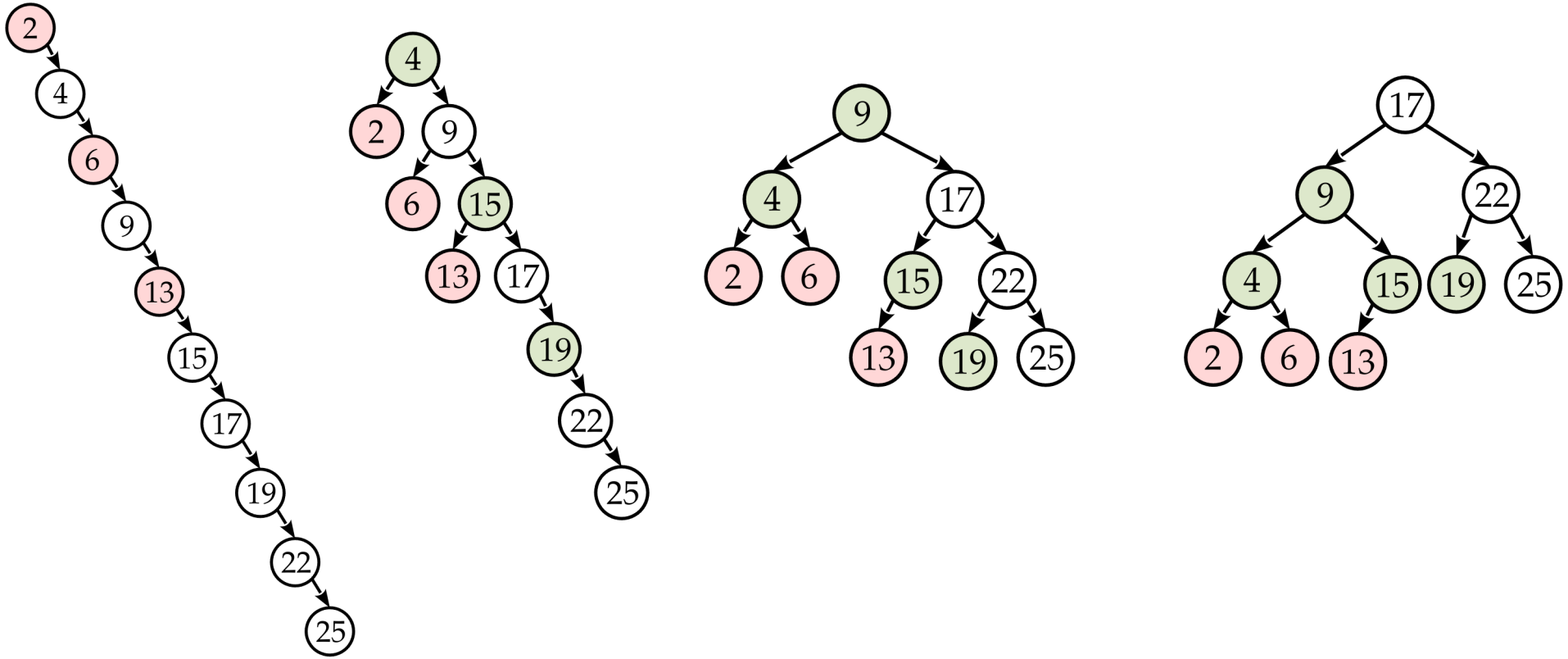
- Primjer
  - $n = 10$
  - To je $h = 4$ razine
  - Ukupni broj internih čvorova je $i = 7$
  - Čvorova u najdonjoj razini je $10 - 7 = 3$ – prvo radimo 3 lijeve rotacije svakog drugog čvora desne kralježnice, počevši od korijenskog čvora
  - Zatim radimo $\lfloor 7/2 \rfloor = 3$ lijeve rotacije svakog drugog čvora ostatka desne kralježnice, počevši od korijenskog čvora, što nam daje najdonju razinu internih čvorova
  - Zatim radimo $\lfloor 3/2 \rfloor = 1$ lijeve rotacije svakog drugog čvora ostatka desne kralježnice, počevši od korijenskog čvora, što nam daje najdonju razinu internih čvorova

DSW – breaking, example



- This is global binary tree balancing
  - We first destroy the structure of the tree to balance it
  - The complexity of the DSW algorithm is ( )

DSW – lomljenje, primjer



- Ovo je globalno uravnoteživanje binarnog stabla
  - Strukturu stabla prvo uništimo da bismo ga uravnotežili
  - Složenost DSW algoritma je $O(n)$

Adelson-Velski-Landis binary tree (AVL)

- Previous examples are **offline** balancing

- **Online** balancing – adding new values
  - Check if the binary tree is balanced?
  - If not, balance that tree with minimal intervention in its structure

- This is called local balancing

Adelson-Velski-Landis binarno stablo (AVL)

- Prethodni primjeri su **offline** uravnotežavanje

- **Online** uravnotežavanje – dodavanje novih vrijednosti
  - Provjeriti da li je binarno stablo uravnoteženo?
  - Ako nije, uravnotežiti to stablo s minimalnim zahvatom u njegovu strukturu

- To se zove lokalno uravnoteživanje

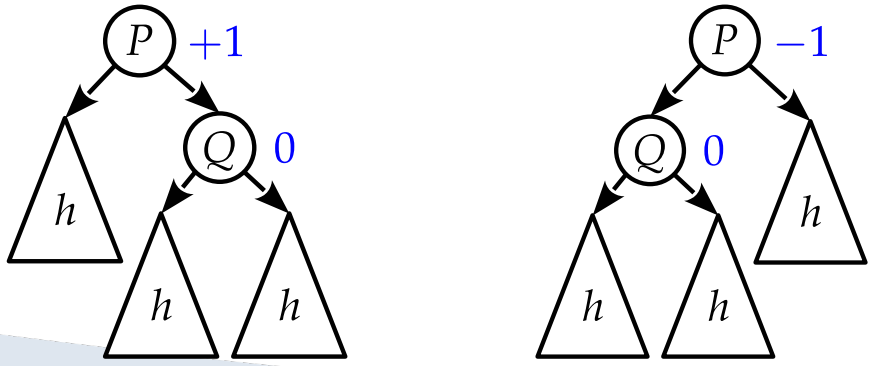
AVL (2)

- By adding a new leaf, we can move along the path to the root node, and check the balance in each node

- For a binary tree = (, ,)we define the balance factor (*balance factor*) as

$$= h( ) - h( )$$

- For all nodes that have −1 ≤ ( ) ≤ 1we consider their subtree to be balanced



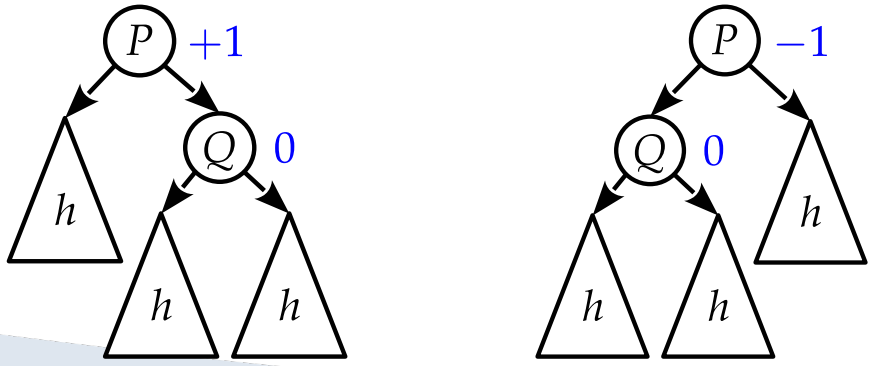
AVL (2)

- Dodavanjem novog lista možemo se kretati po putanji do korijenskog čvora, te provjeravati uravnoteženost u svakom čvoru

- Za binarno stablo $B = (L, S, R)$ definiramo faktor ravnoteže (*balance factor*) kao

$$BF(S) = h(R) - h(L)$$

- Za sve čvorove koji imaju $-1 \leq BF(S) \leq 1$ smatramo da je njihovo podstablo uravnoteženo



AVL (3)

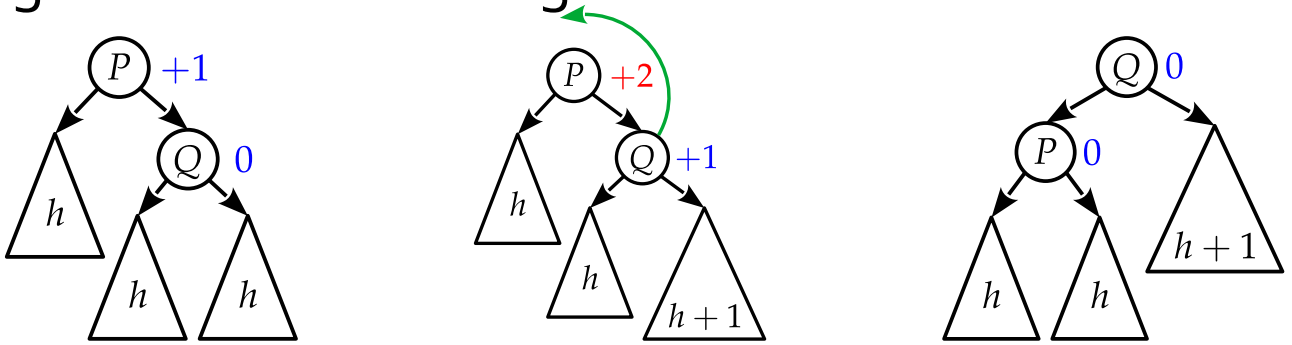
- After adding, we update the balance factors on the vertical path. If there is a node S with ( ) = −2or ( ) = 2,local balancing is required
- Two cases – a node and its child (depending on the sign of the node):
  - **Level up**case, identical signs BF:
    - The balance factor of the node is +2 and the right child has a balance factor of 0 or +1 – the right flattened case
    - The balance factor of a node is -2 and the left child has a balance factor of 0 or -1 – the left aligned case
  - **Broken**case, strictly opposite signs BF:
    - The balance factor of the node is +2 and the right child has a balance factor of -1 – the right broken case
    - The balance factor of the node is -2 and the left child has a balance factor of +1 – the left broken case

AVL (3)

- Nakon dodavanja ažuriramo faktore ravnoteže na vertikalnoj putanji. Ako postoji čvor S sa $BF(S) = -2$ ili $BF(S) = 2$, potrebno je lokalno uravnotežavanje
- Dva slučaja – čvor i njegovo dijete (ovisno o predznaku čvora):
  - **Izravnati** slučaj, identični predznaci BF:
    - Faktor ravnoteže čvora je +2 i desno dijete ima faktor ravnoteže 0 ili +1 – desni izravnati slučaj
    - Faktor ravnoteže čvora je -2 i lijevo dijete ima faktor ravnoteže 0 ili -1 – lijevi izravnati slučaj
  - **Izlomljeni** slučaj, strogo suprotni predznaci BF:
    - Faktor ravnoteže čvora je +2 i desno dijete ima faktor ravnoteže -1 – desni izlomljeni slučaj
    - Faktor ravnoteže čvora je -2 i lijevo dijete ima faktor ravnoteže +1 – lijevi izlomljeni slučaj

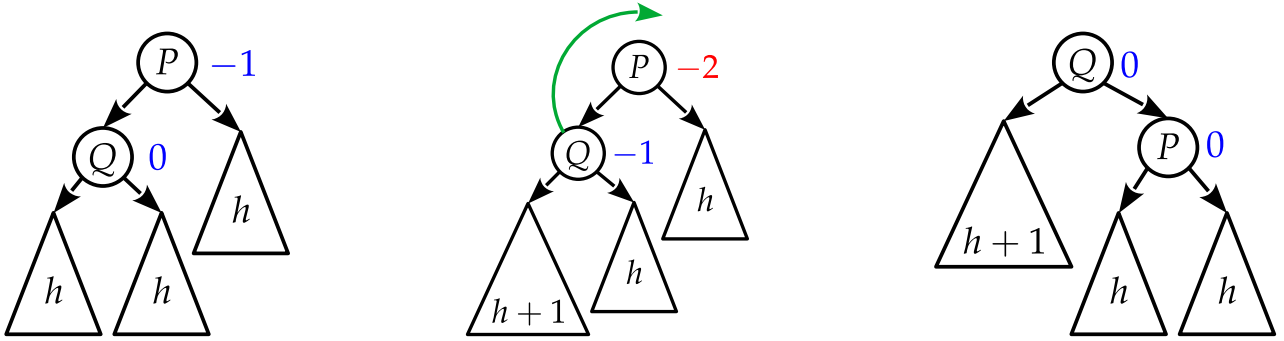
AVL (4)

- Right aligned case: +2 and right child 0 or +1



  - We add the new value to the right subtree of node Q
  - A left rotation of Q around P is performed

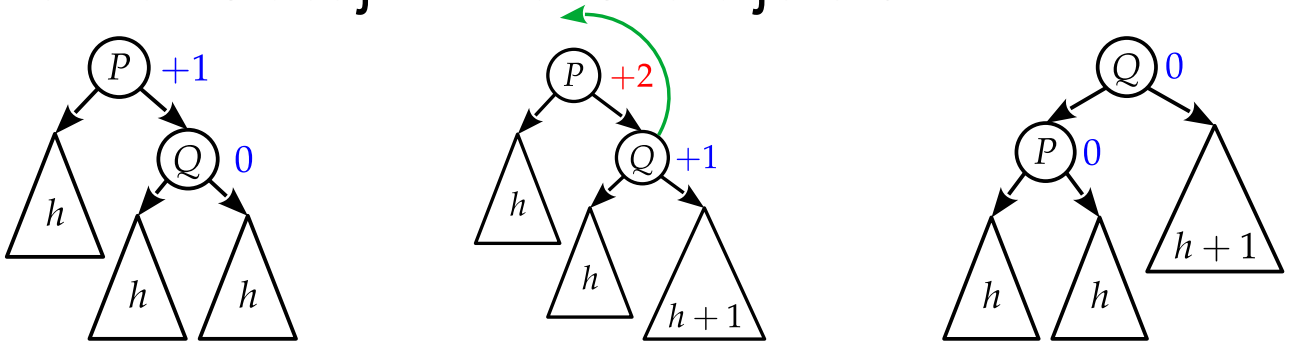
- Left aligned case: -2 and left child 0 or -1



  - A right rotation of Q around P is performed

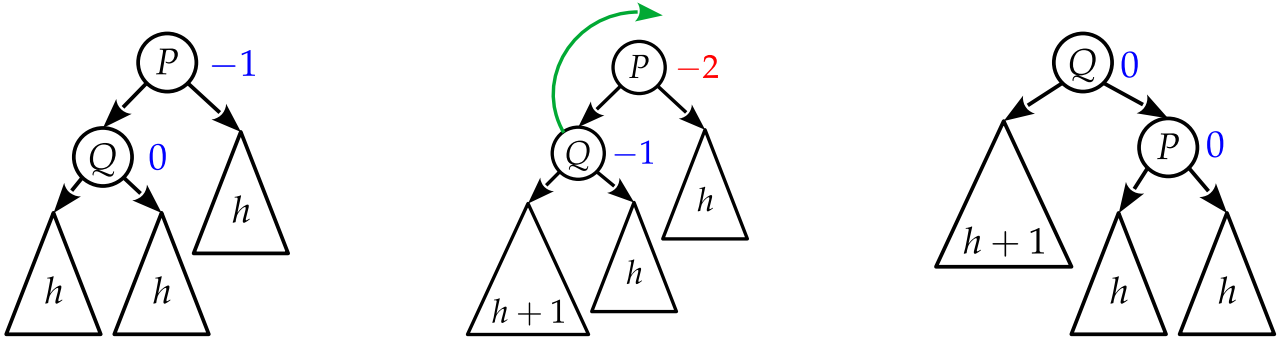
AVL (4)

- Desni izravnati slučaj: +2 i desno dijete 0 ili +1



  - Novu vrijednost dodajemo u desno podstablo čvora Q
  - Radi se lijeva rotacija Q oko P

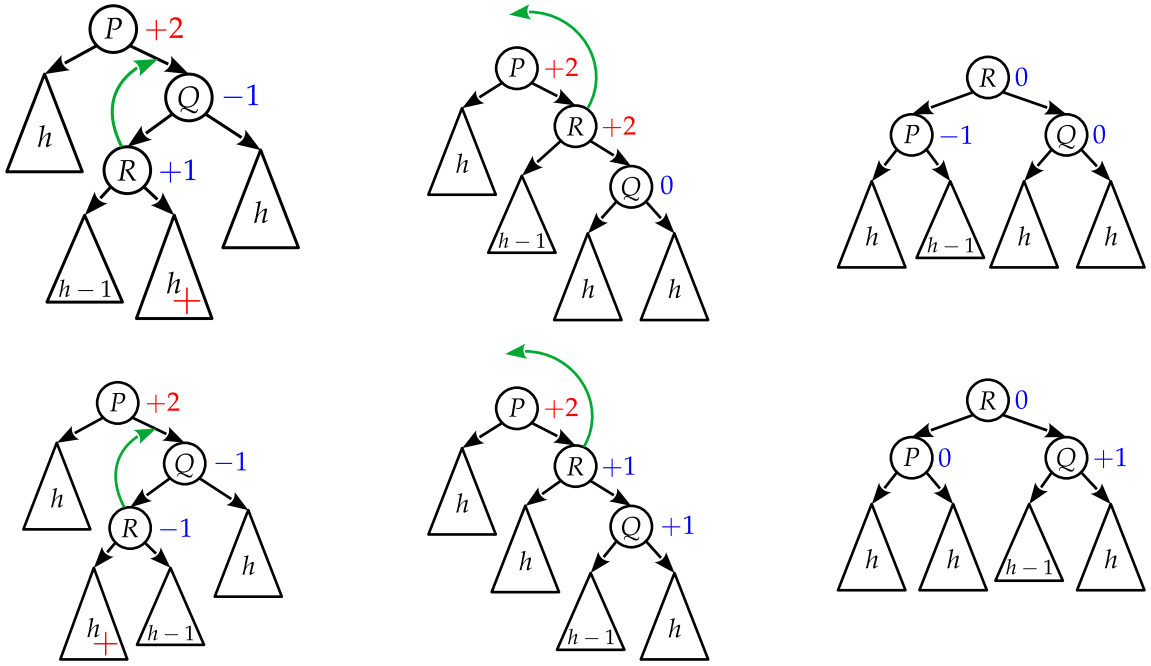
- Lijevi izravnati slučaj: -2 i lijevo dijete 0 ili -1



  - Radi se desna rotacija Q oko P

AVL (5)

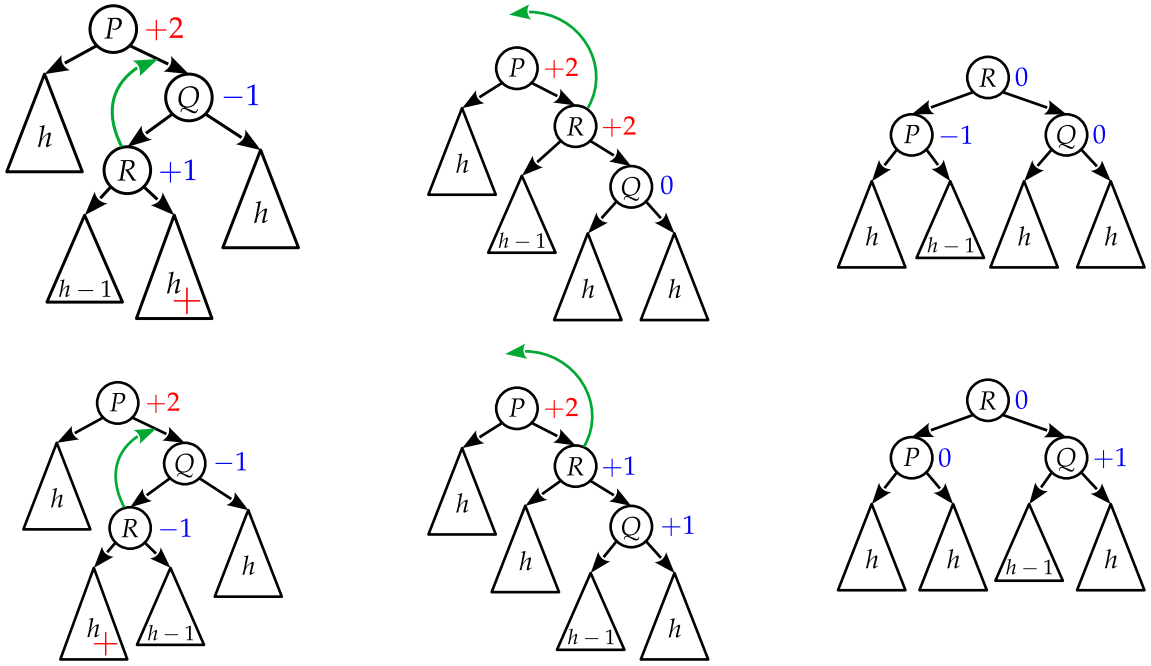
- Right fractured case: +2 and right child -1



- First the right rotation of R about Q
- Then a left rotation of R around P

AVL (5)

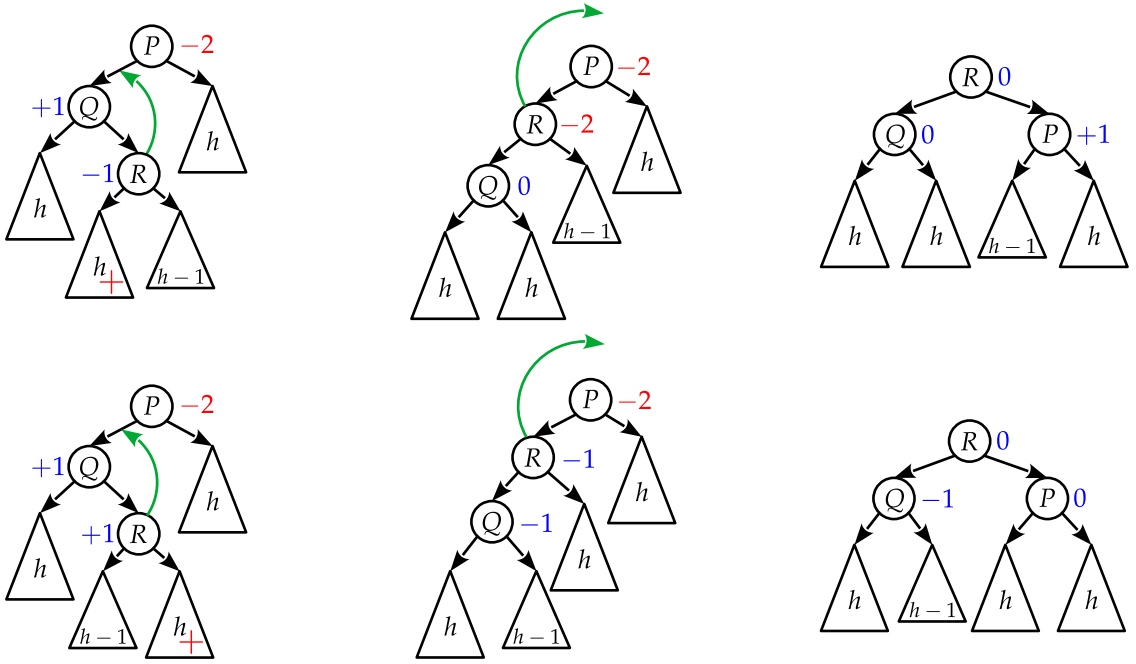
- Desni izlomljeni slučaj: +2 i desno dijete -1



- Prvo desna rotacija R oko Q
- Zatim lijeva rotacija R oko P

AVL (6)

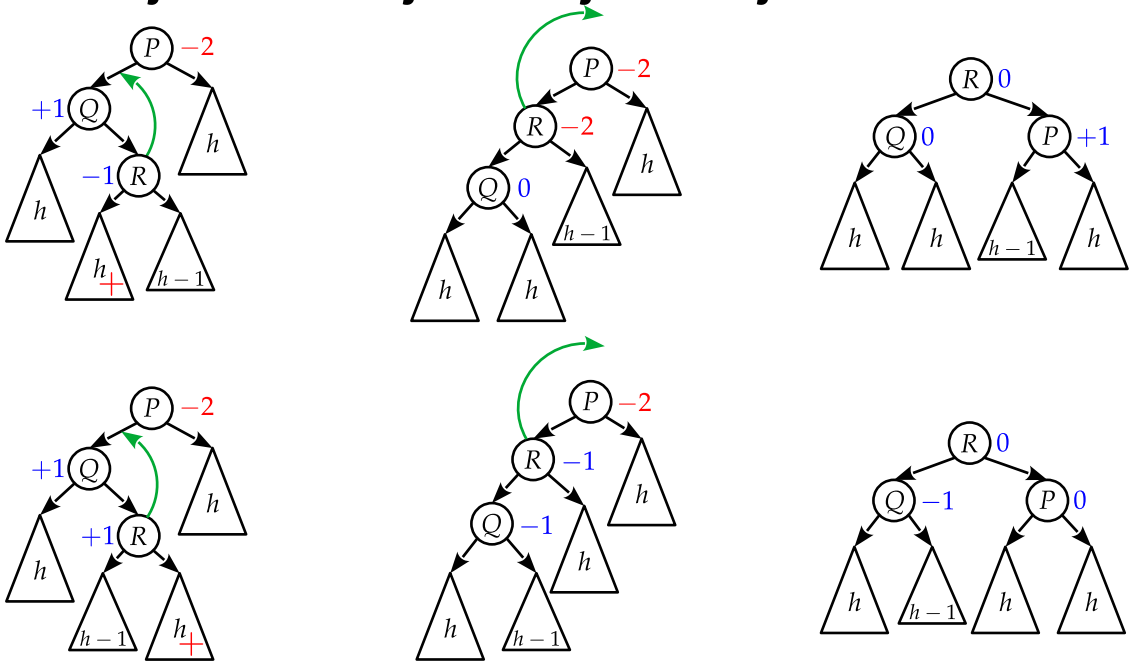
- Left broken case: -2 and left child +1



- First a left rotation of R about Q
- Then a right rotation of R around P

AVL (6)

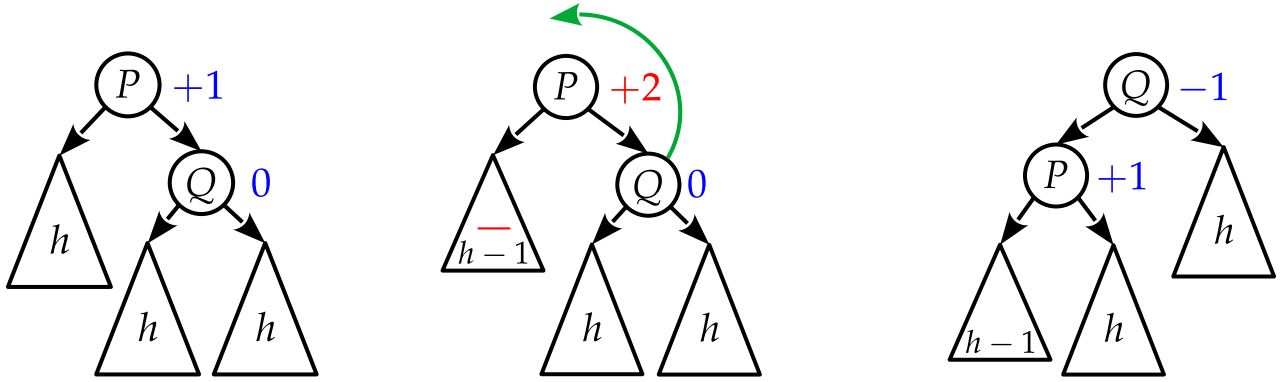
- Lijevi izlomljeni slučaj: -2 i lijevo dijete +1



- Prvo lijeva rotacija R oko Q
- Zatim desna rotacija R oko P

AVL (7)

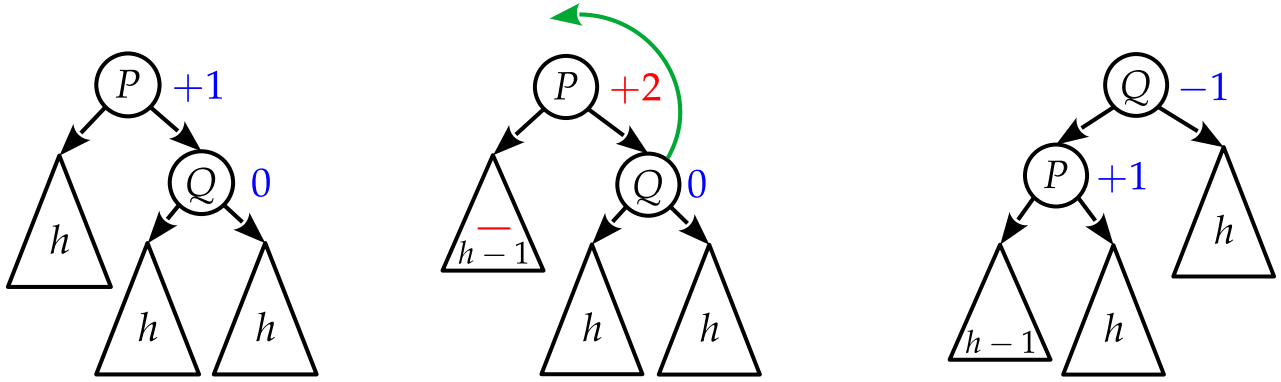
- Deleting values - always delete by copying
  - With such deletion, the height of one of the subtrees can be reduced



  - We can see that deleting a node in the left subtree of node P caused an imbalance
  - It is the right flat case

AVL (7)

- Brisanje vrijednosti – uvijek brisanje kopiranjem
  - Kod takvog brisanja jednom od podstabala se može smanjiti visina



  - Vidimo da je brisanje čvora u lijevom podstablu čvora P uzrokovalo disbalans
  - To je desni izravnati slučaj

AVL (8)

```
function AVLDetectRotate(n)
    if balanceFactor(n) is +2 then
        n1 ← rightChild(n)
        if balanceFactor(n1) is 0 or +1 then
            left rotate n1 around n
        if balanceFactor(n1) is −1 then
            n2 ← leftChild(n1)
            right rotate n2 around n1
            left rotate n2 around n
    else
        n1 ← leftChild(n)
        if balanceFactor(n1) is 0 or −1 then
            right rotate n1 around n
        if balanceFactor(n1) is +1 then
            n2 ← rightChild(n1)
            left rotate n2 around n1
            right rotate n2 around n
procedure AVLBalance(n)
    p ← parent(n)
    if balanceFactor(n) is −2 or +2 then
        AVLDetectRotate(n)
    if p is not nil then
        AVLBalance(p)
```

- The complexity of the search is the same as for a classic binary tree (!)
- When writing or deleting values, we return along the vertical path back to the root node, which gives complexity (2 !)
- The theoretical height of the AVL tree is !(+ 1) ≤ *h* ≤ . !(+ 2) − 0.328
  - Proof in Drozdek

AVL (8)

```
function AVLDetectRotate(n)
    if balanceFactor(n) is +2 then
        n1 ← rightChild(n)
        if balanceFactor(n1) is 0 or +1 then
            left rotate n1 around n
        if balanceFactor(n1) is −1 then
            n2 ← leftChild(n1)
            right rotate n2 around n1
            left rotate n2 around n
    else
        n1 ← leftChild(n)
        if balanceFactor(n1) is 0 or −1 then
            right rotate n1 around n
        if balanceFactor(n1) is +1 then
            n2 ← rightChild(n1)
            left rotate n2 around n1
            right rotate n2 around n
procedure AVLBalance(n)
    p ← parent(n)
    if balanceFactor(n) is −2 or +2 then
        AVLDetectRotate(n)
    if p is not nil then
        AVLBalance(p)
```

- Složenost pretraživanja je kao i kod klasičnog binarnog stabla $O(log_2 n)$
- Kod upisa ili brisanja vrijednosti, vraćamo se po vertikalnoj putanji natrag to korijenskog čvora što daje kompleksnost $O(2log_2 n)$
- Teoretska visina AVL stabla je

$$log_2(n+1) \leq h \leq \mathbf{1.44}\, log_2(n+2) - 0.328$$

  - Dokaz u Drozdeku

Questions?

Pitanja ?