

Vyhledávání

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta a kol., 2017

Datové struktury a algoritmy, B6B36DSA
04/2017, Lekce 10

<https://cw.fel.cvut.cz/wiki/courses/b6b36dsa/start>



Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Vyhledávání (Searching)

Definice:

- Vstup: množina n záznamů obsahujících klíče a hledaný klíč k
- Výstup: Záznamy s klíčem k nebo informace, že žádný takový neexistuje

Implementace:

- Implementace pomocí pole:
 - Sekvenční vyhledávání
 - Binární vyhledávání
- Binární vyhledávací stromy (BVS, nebo BST – Binary Search Trees):
 - Reprezentace uzlů a hran
 - Operace nad BVS
 - Vyvažování BVS

Typické operace při vyhledávání

- Připomeňme si operace související s vyhledáváním:
 - vložení záznamu s klíčem
 - odstranění záznamu s klíčem
 - vyhledání záznamu podle klíče
 - minimum a maximum
 - předchůdce a následník
 - a další ...

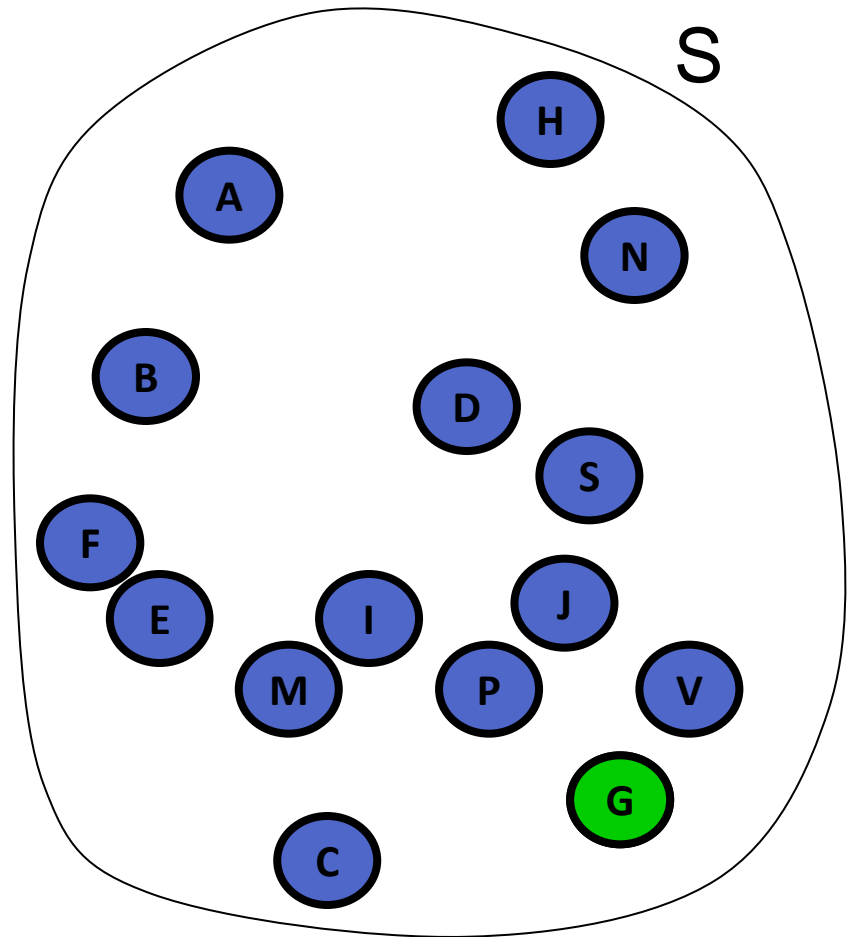
Vyhledávání

Vstup: množina n klíčů a hledaný klíč k

Popis problému: Kde je k ?

G?

- Hledání G v S bylo úspěšné



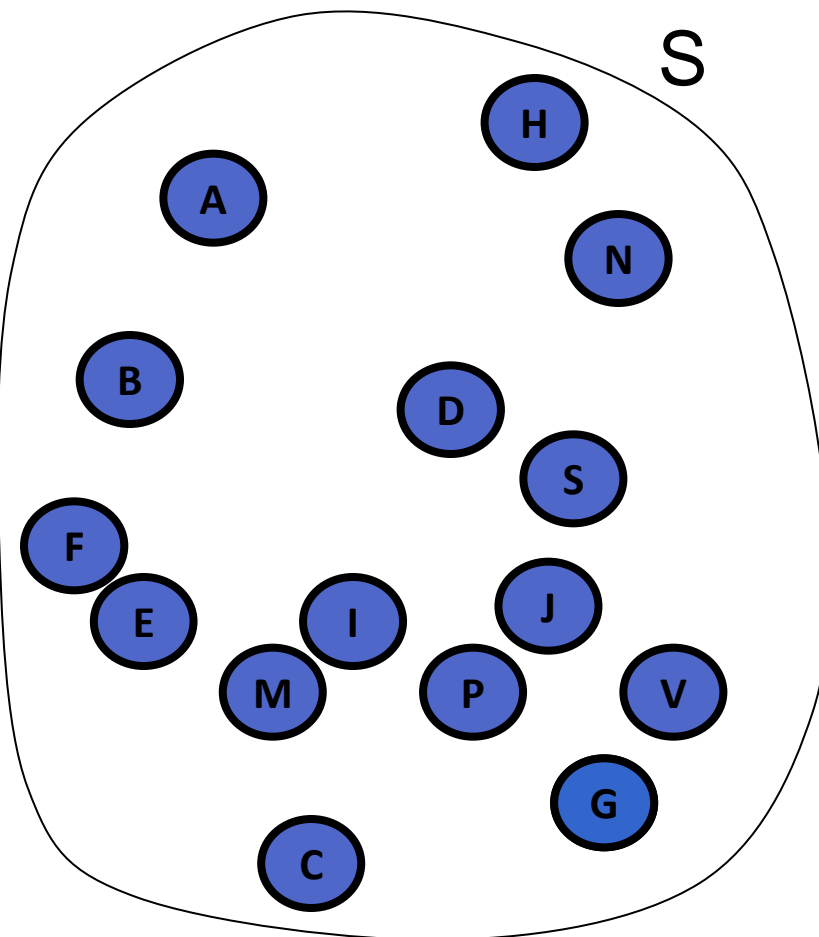
Vyhledávání

Vstup: množina n klíčů a hledaný klíč k

Popis problému: Kde je k ?

L?

- Hledání L v S
nebylo úspěšné



Prostor pro vyhledávání

Prohledávaný prostor S (search space)

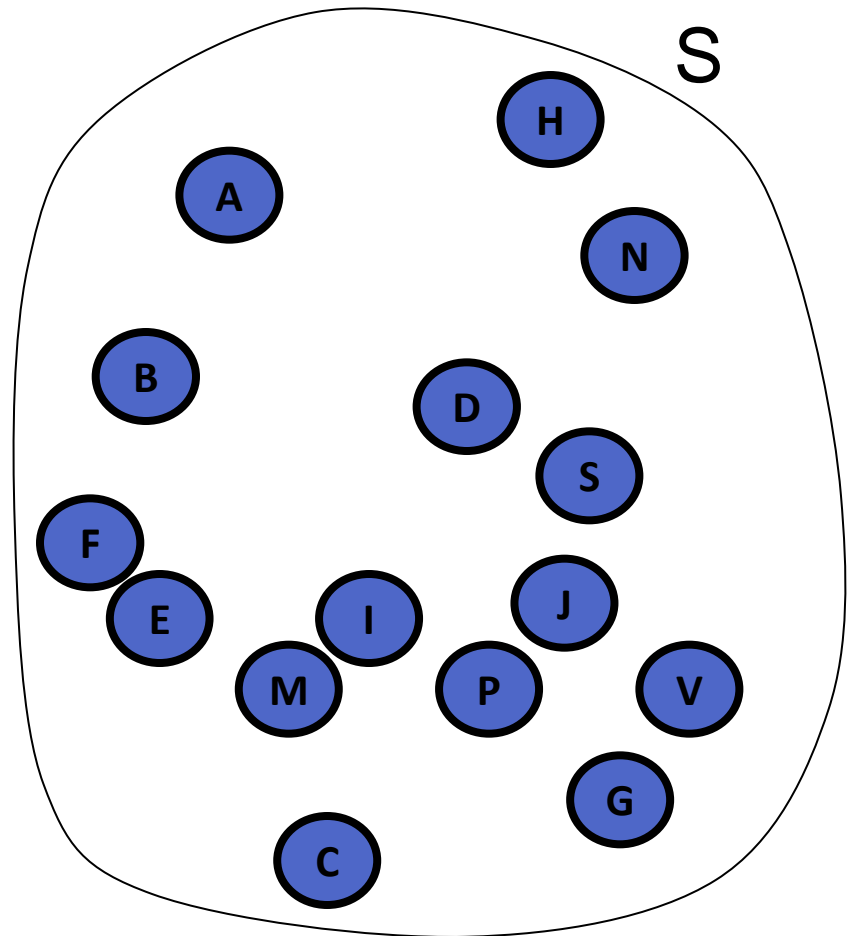
= množina klíčů, ve které hledáme

- přesněji: množina záznamů, ve které hledáme podle klíče
- unikátní klíče
- tabulka, soubor, ...

Universum U pro vyhledávací prostor

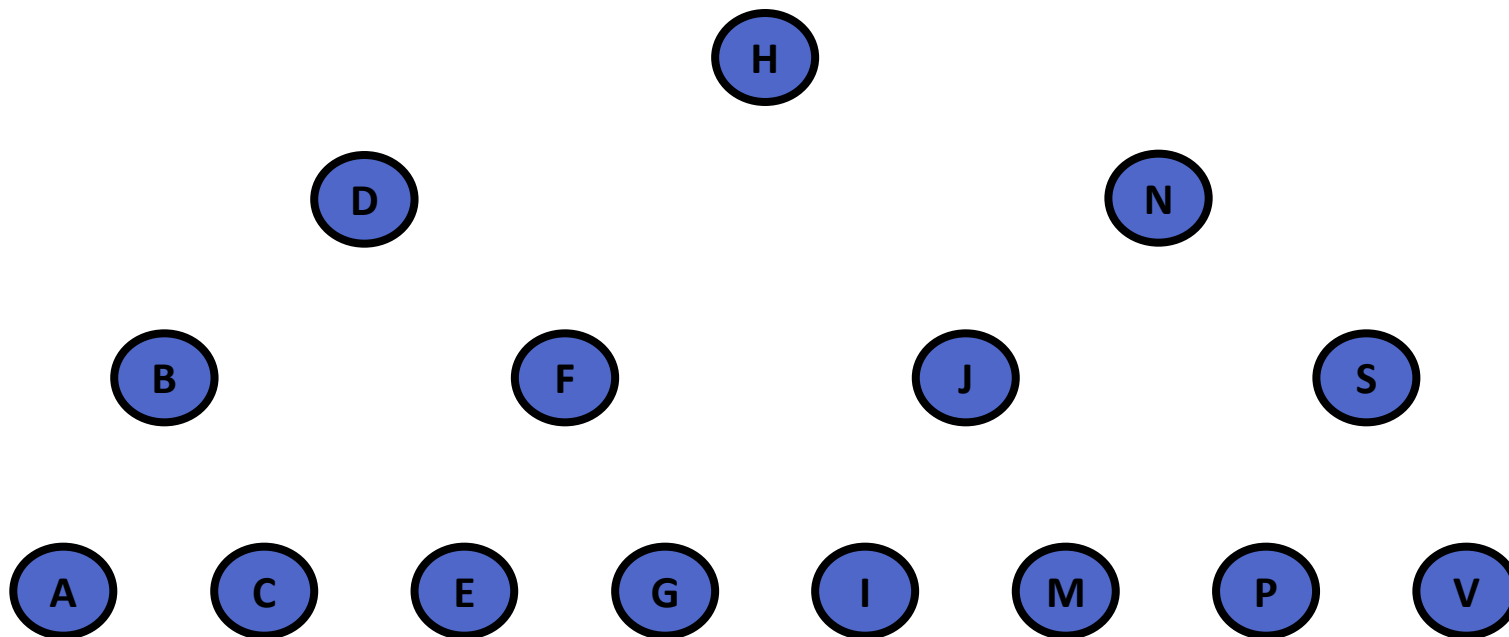
= množina všech možných klíčů

$$S \subset U$$



Problémy vyhledávání

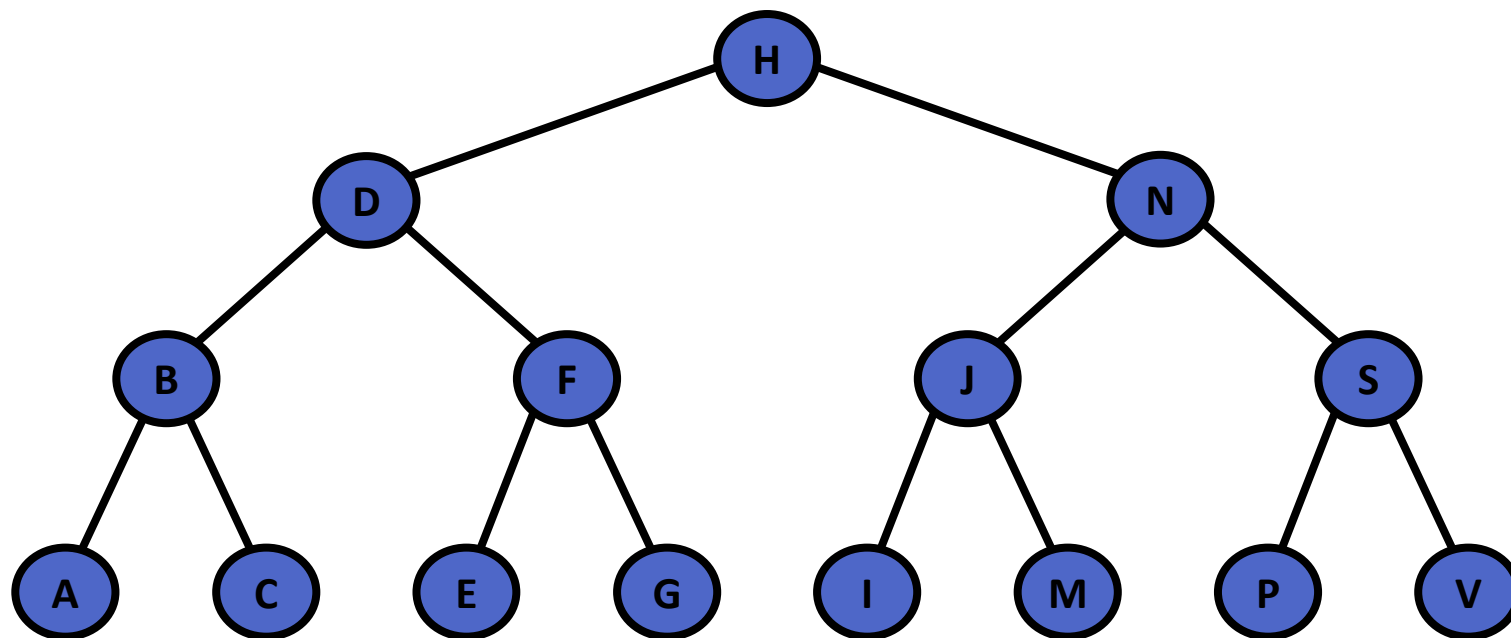
- Rychlost



Problémy vyhledávání

Vstup: množina n klíčů a hledaný klíč k

Popis problému: Kde je k ?

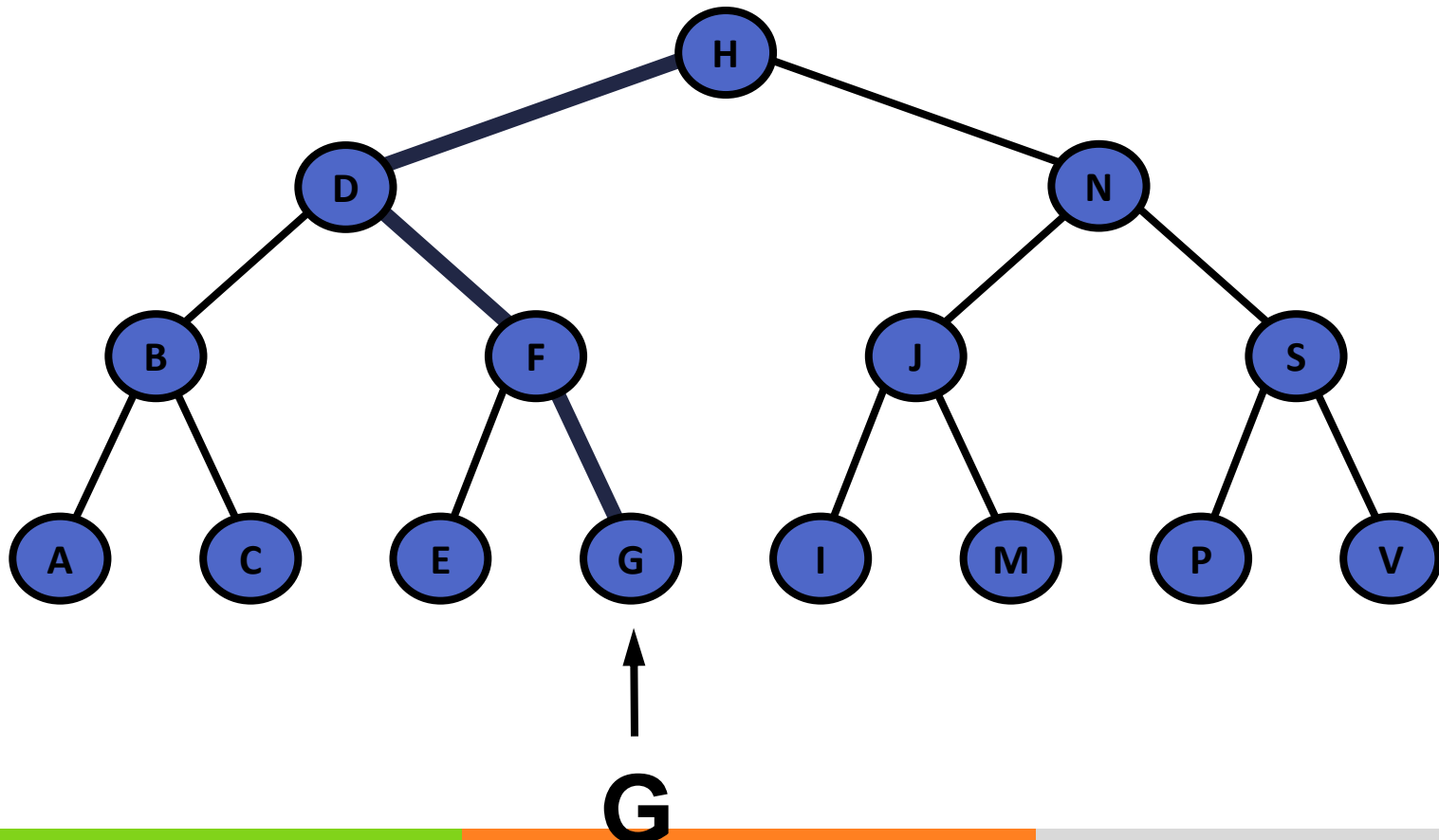


G?

Problémy vyhledávání

Vstup: množina n klíčů a hledaný klíč k

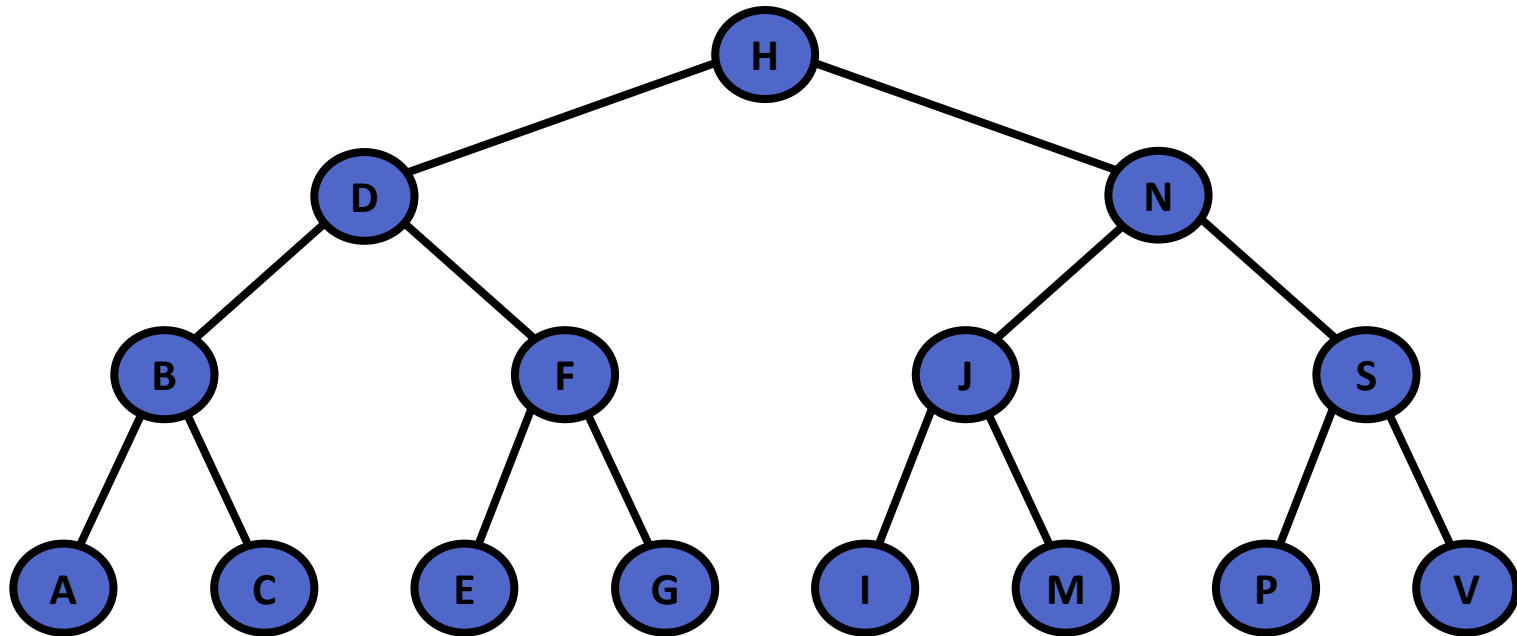
Popis problému: Kde je k ?



Problémy vyhledávání

Vstup: množina n klíčů a hledaný klíč k

Popis problému: Kde je k ?

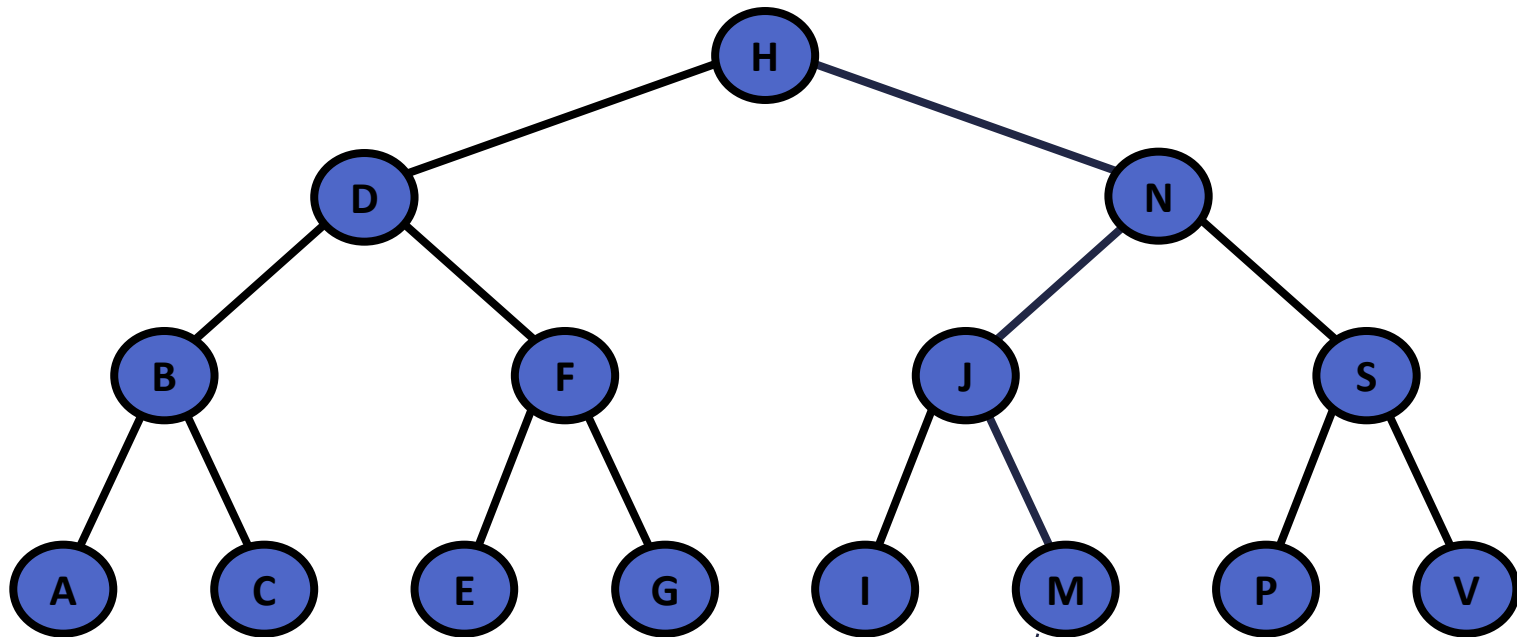


L?

Problémy vyhledávání

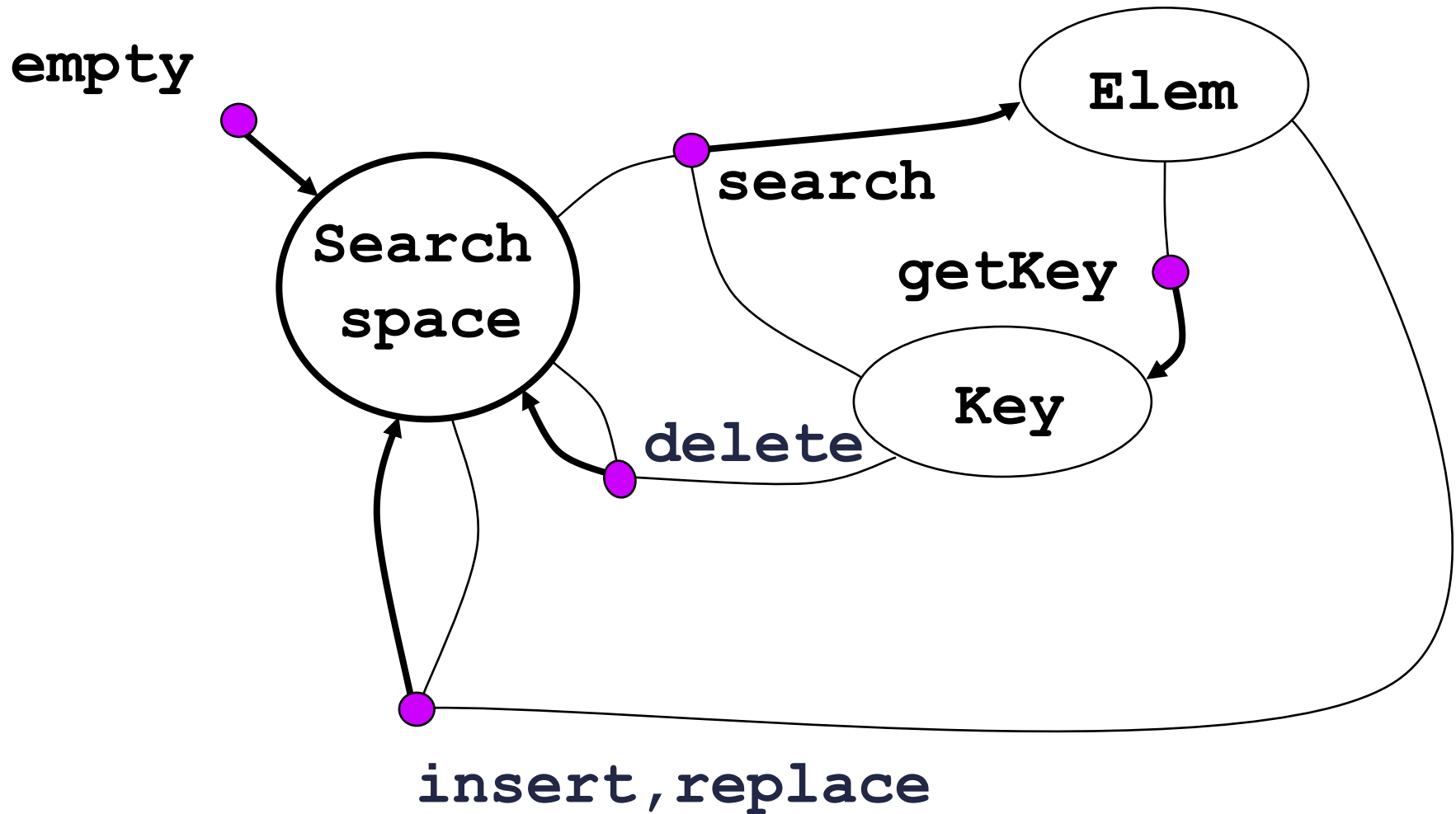
Vstup: množina n klíčů a hledaný klíč k

Popis problému: Kde je k ?



L nebylo nalezeno

Signatura operací pro vyhledávání

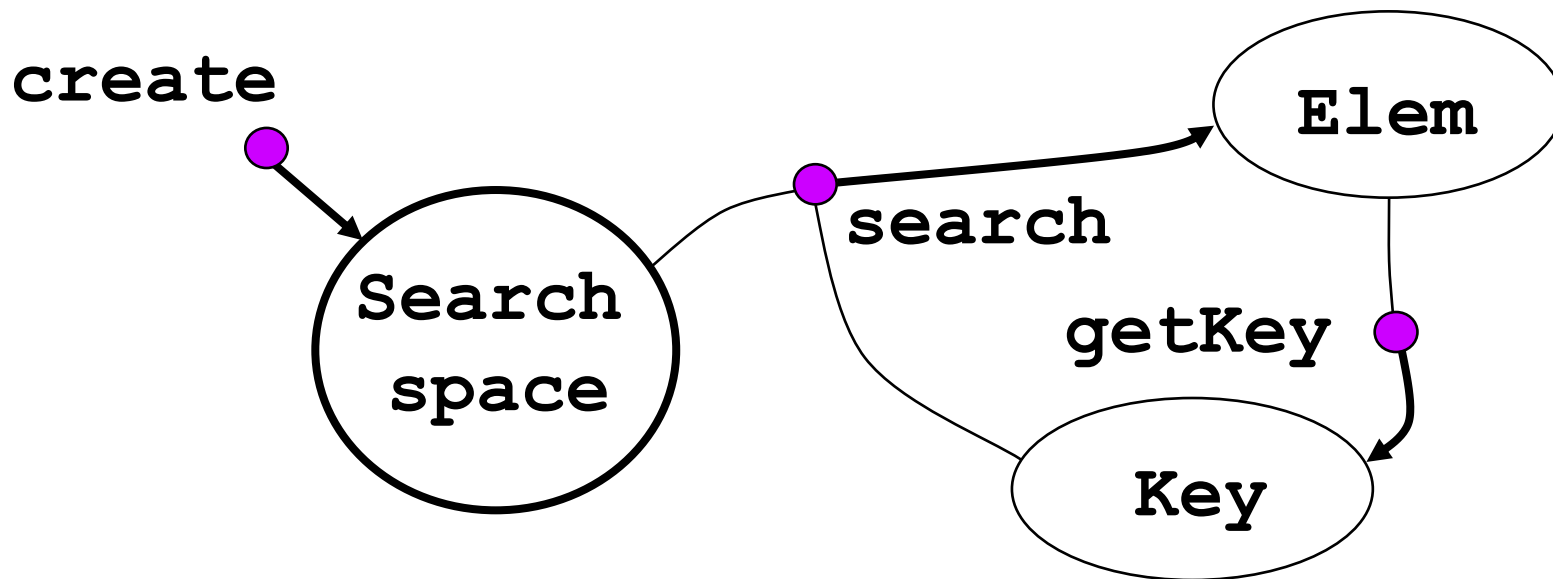


Taxonomie pro vyhledávání

Prohledávaný prostor

- **Statický**
 - prohledávaný prostor je fixní
 - > jednodušší implementace
 - > změna znamená novou verzi
 - > příklad: telefonní seznam, tištěný adresář
- **Dynamický**
 - prohledávaný prostor se může měnit v čase
 - > složitější implementace
 - > změny se provádějí pomocí operací:
insert, delete, replace
 - > příklad: tabulka symbolů v kompilátoru

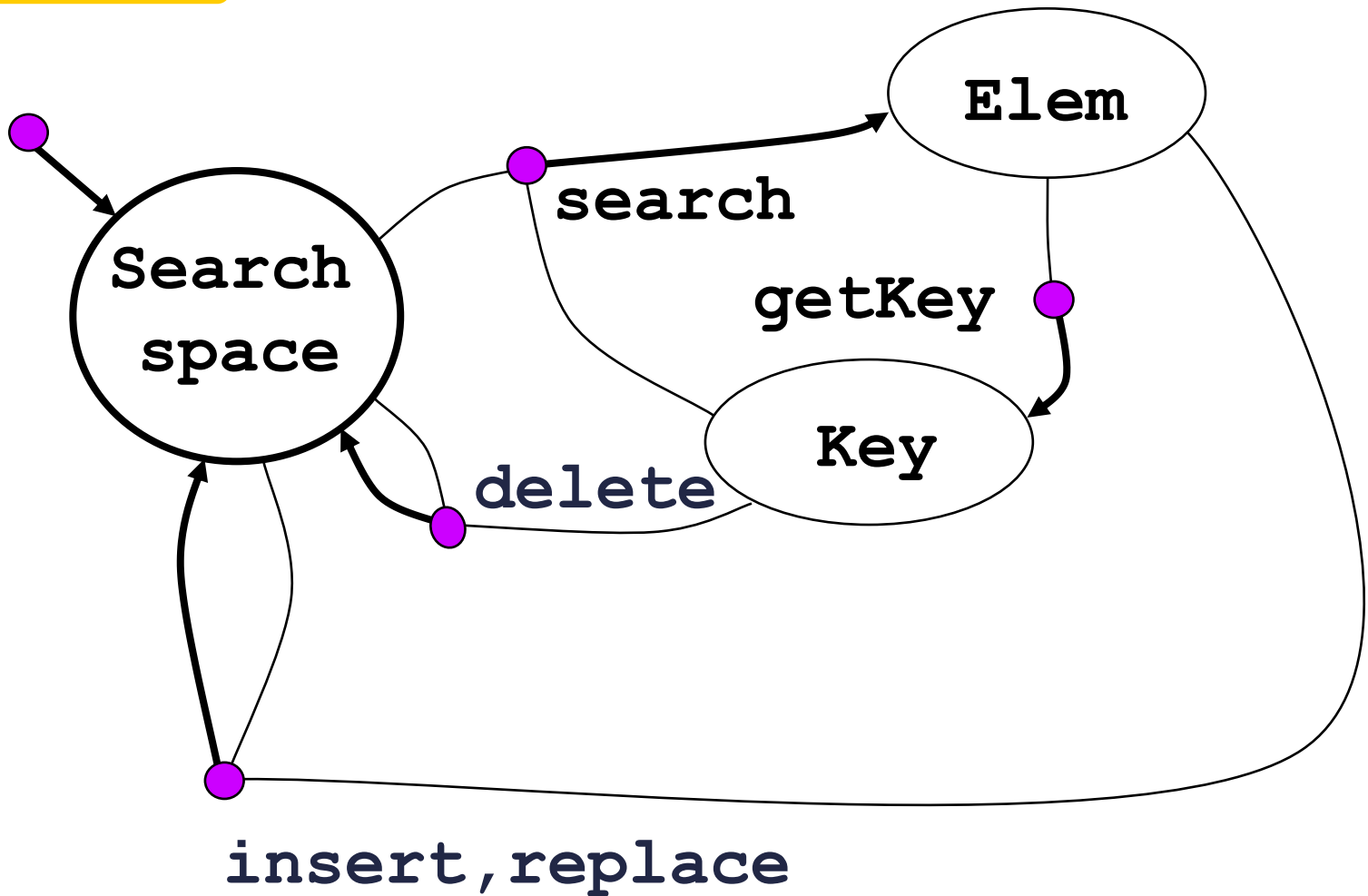
Statický prohledávaný prostor



Dynamický prohledávaný prostor

Dynamic search space

empty



Vyhledávací operace

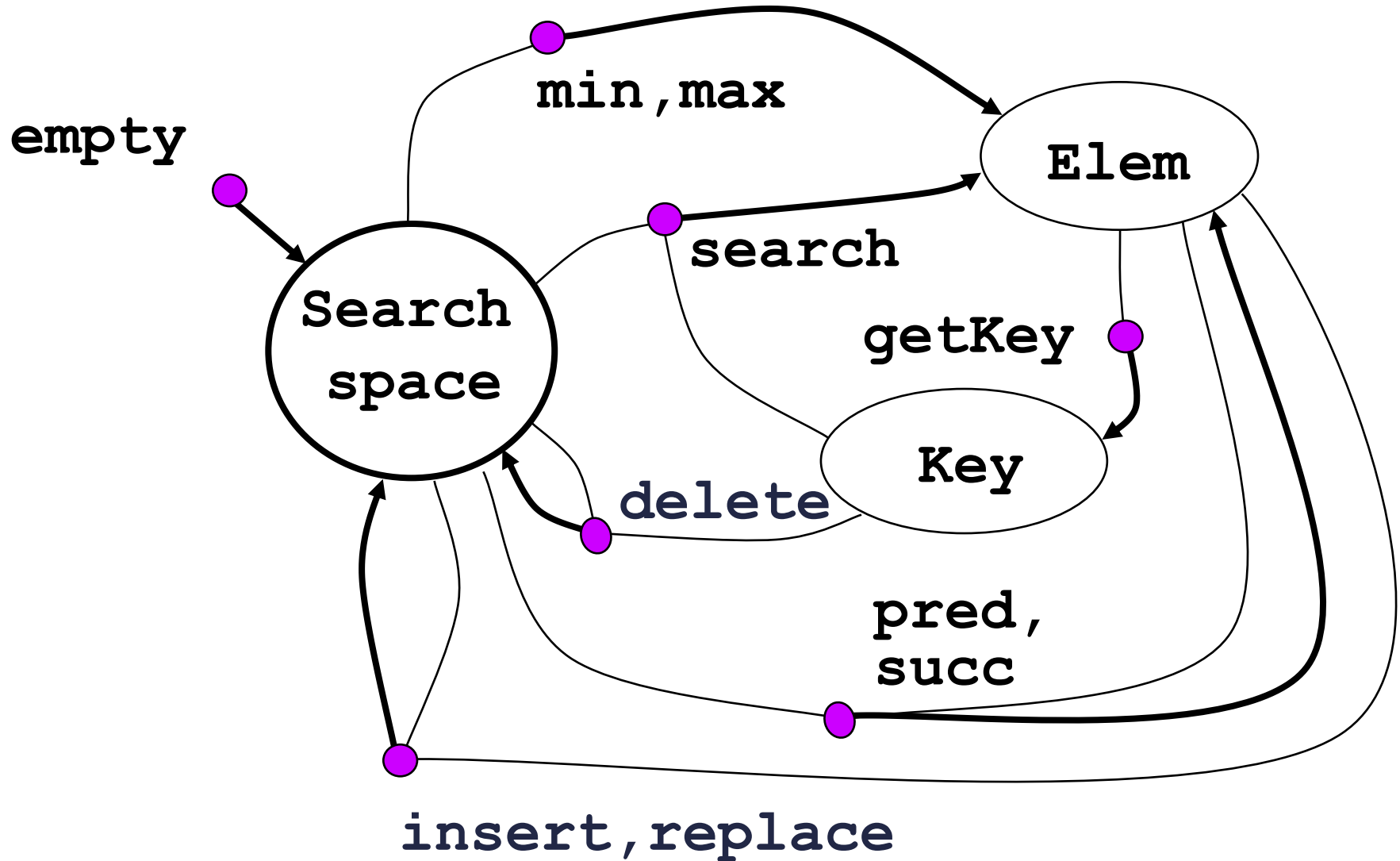
Symboly:

- k ... klíč (key)
- e ... element s klíčem k (element with key k)
- s ... prohledávaná sada (data set)

Operace (informativní seznam):

- Selektory:
 - $\text{search}(k, s)$
 - $\text{min}(s), \text{max}(s)$
 - $\text{pred}(e, s), \text{succ}(e, s)$
 } extenze
 - Modifikátory:
 - $\text{insert}(e, s), \text{delete}(k, s), \text{replace}(e, s)$
- klíč
nahrazovaného
elementu je
součástí
nového
elementu e

Rozšířené vyhledávání



Jiná klasifikace

Adresní vyhledávání

- Adresní vyhledávání - založeno na digitálních vlastnostech klíčů
 - výpočet pozice z klíče $pos = f(k)$
 - přímý přístup (direct access), hašování
 - pole, tabulka, ...
 - přímý přístup => rychlé ... $O(1)$

Asociativní vyhledávání

- Asociativní vyhledávání - založeno na porovnání elementů
 - elementy jsou umístěny podle vztahu k ostatním
 - sekvenční hledání, binární hledání, vyhledávací stromy
 - vyžaduje hledání => pomalejší ... $O(\log n)$ až $O(n)$

Ještě jiné klasifikace

Interní nebo externí

- Interní nebo externí
 - Interní – v paměti
 - Externí – na disku (příp. pásce)

Dimenze klíčů

- Dimenze klíčů
 - Jedno-dimenzionální - k
 - Multi-dimenzionální - $[x,y,z]$

Měření kvality (Quality measures)

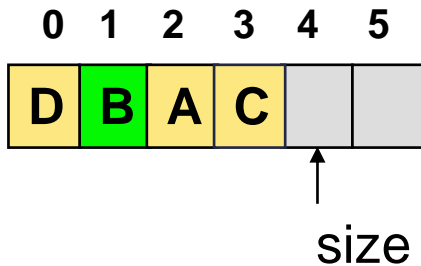
Prostor pro data

- $P(n)$ = paměťová složitost (memory complexity)

Časová náročnost

- $Q(n)$ = časová složitost **search**, **query**
- $I(n)$ = časová složitost **insert**
- $D(n)$ = časová složitost **delete**

Sekvenční vyhledávání v neseřazeném poli



- **Neseřazené pole**
- sekvenční hledání
- insert
- delete
- min, max

$P(n) = O(n)$

$Q(n) = O(n)$

$I(n) = O(1)$

$D(n) = O(n)$

in $O(n)$



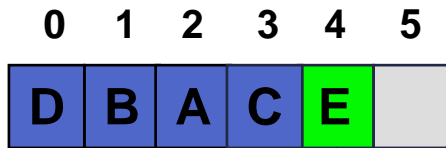
```

nodeT seqSearch( key k, nodeT a[] ) {
    int i = 0;
    while( (i < a.size) && (a[i].key != k) )
        i++;
    if( i < a.size ) return a[i];
    else return NODE_NOT_FOUND;
}

```

Java-like pseudo code

Vylepšené vyhledávání v neseřazeném poli



↑
size

search("E", a)

Neseřazené pole se záložkou (sentinel)

Sekvenční hledání je stále $Q(n) = O(n)$



```

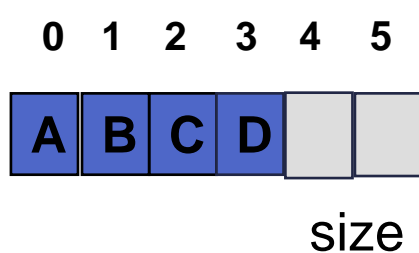
nodeT seqSearchWithSentinel( key k, nodeT a[] ) {
    int i = 0;
    a[a.size] = createArrayElement(k); // add sentinel
    while( a[i].key != k ) // save one test per step
        i++;
    if( i < a.size ) return a[i];
    else return NODE_NOT_FOUND;
}

```



Java-like pseudo code

Binární hledání v seřazeném poli



Seřazené pole

Binární hledání

insert

delete

min, max

$P(n) = O(n)$

$Q(n) = O(\log(n))$

$I(n) = O(n)$

$D(n) = O(n)$

$O(1)$



```

nodeT binarySearch( key k, nodeT sortedArray[] ) {
    int pos = bs( k, sortedArray, 0, sortedArray.size - 1 );

    if( pos >= 0 ) return sortedArray[pos];
    else
        return NODE_NOT_FOUND;
        // bs can return -(pos+1), i.e.
        // position to insert the node with key k
}
  
```

Java-like pseudo code

Binární hledání – rekurze/iterace

// Recursive version

```
int bs( key k, nodeT a[], int first, int last ) {
    if( first > last ) return -(first + 1); // not found
    int mid = ( first + last ) / 2;
    if( k < a[mid].key ) return bs( k, a, first, mid - 1);
    if( k > a[mid].key ) return bs( k, a, mid + 1, last );
    return mid; // found!
}
```

Java-like pseudo code

// Iterative version

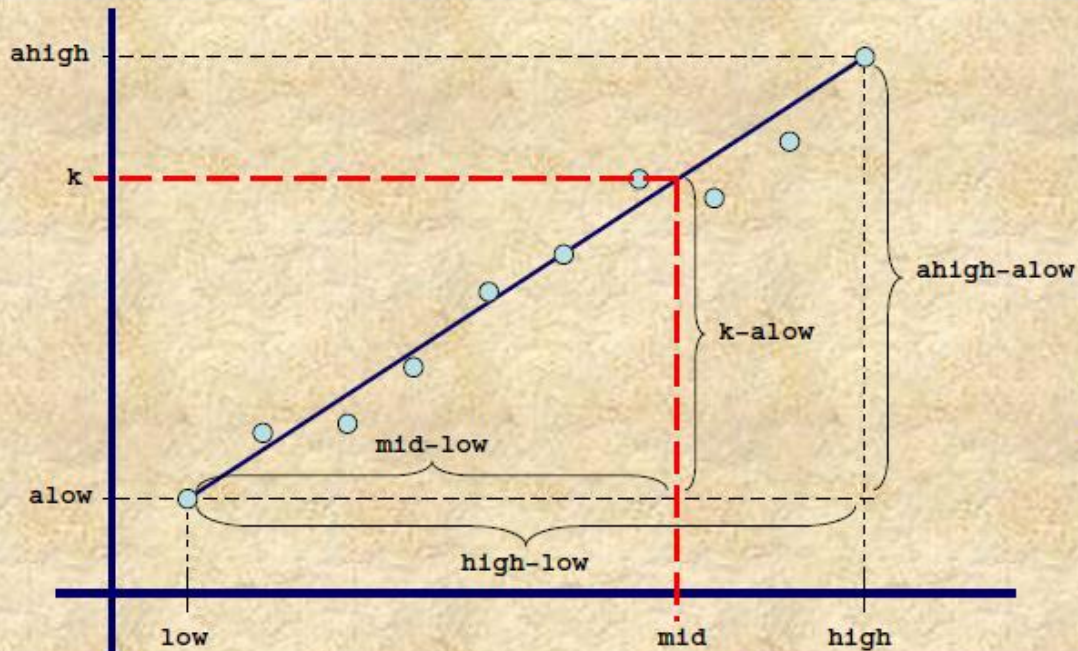
```
int bs(key k, nodeT a[], int first, int last ) {
    while (first <= last) {
        int mid = (first + last) / 2; // mid point
        if (k > a[mid].key) first = mid + 1;
        else if (key < a[mid].key) last = mid - 1;
        else return mid; // found
    } return -(first + 1); // failed to find key
}
```

Java-like pseudo code

Interpolační vyhledávání

Jak hledáme ve slovníku nebo v tlf seznamu? Určitě ne pŮlením ...

Předpokládejme **rovnoměrné rozložení (číselných) klíčů**, umístění klíče odhadujeme podle jeho hodnoty interpolací:



$$\begin{aligned} (k - \text{low}) / (\text{ahigh} - \text{low}) &= \\ &= (\text{mid} - \text{low}) / (\text{high} - \text{low}) \end{aligned}$$

$$\text{mid} = \text{low} + (k - \text{low}) * (\text{high} - \text{low}) / (\text{ahigh} - \text{low})$$

Porovnání binární/interpolační hledání

Binární hledání - $O(\log n)$

```
mid = ( first + last ) / 2;
```

Interpolační hledání - $O(\log \log n)$

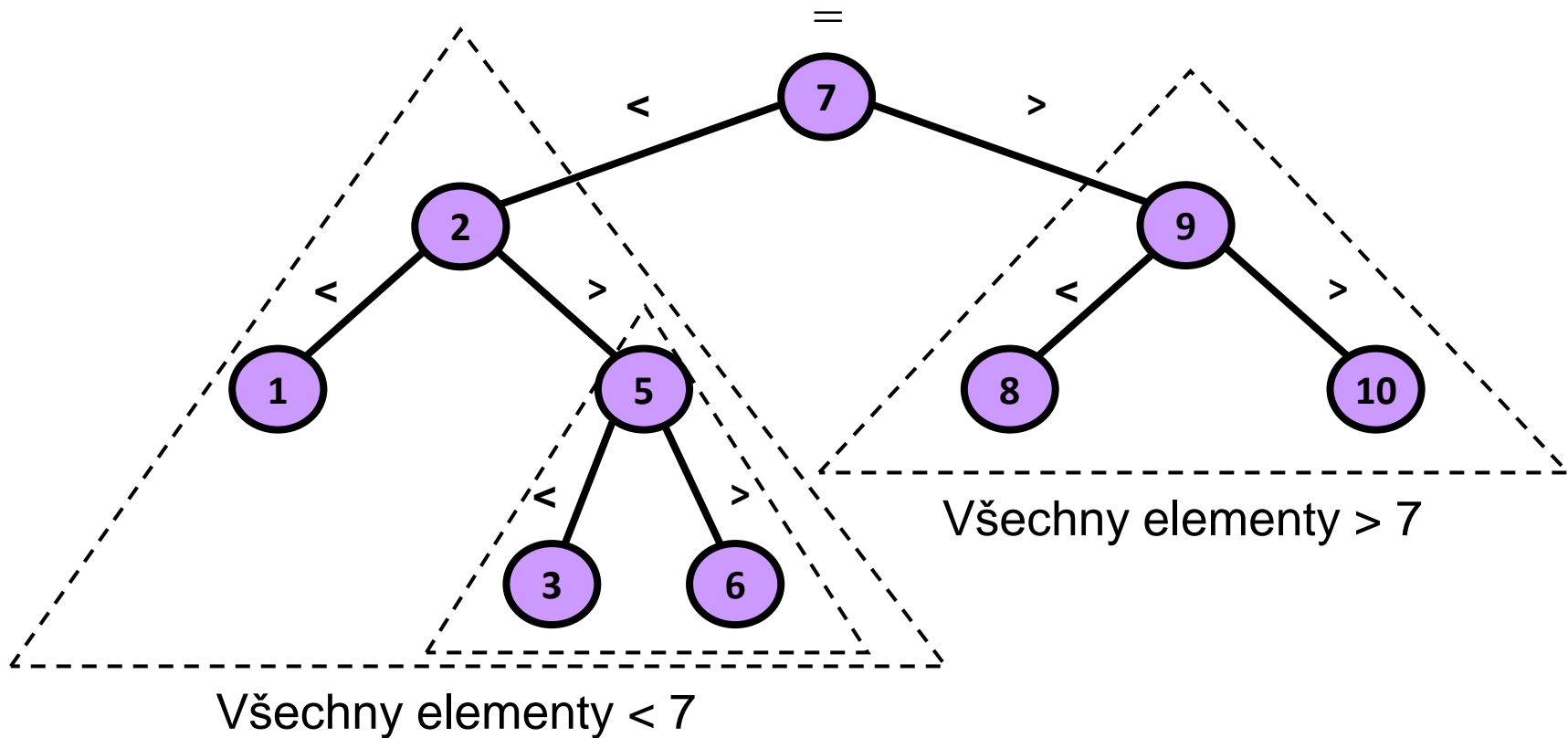
$$\text{pos} = \text{first} + \frac{(\text{last} - \text{first})}{\text{a}[\text{last}] - \text{a}[\text{first}]} (\text{x} - \text{a}[\text{first}])$$

```
Elem searchInterpol( Elem[] a, int low, int high, Key k) {
    if (low > high) return null;
    int mid = low + (k-a[low].key)*(high-low)/(a[high].key-a[low].key);
    if (a[mid].key == k) return a[mid];
    if (a[mid].key < k)
        return searchInterpol( Elem[] a, mid+1, high, k );
    else return searchInterpol( Elem[] a, low, mid-1, k );
}
```

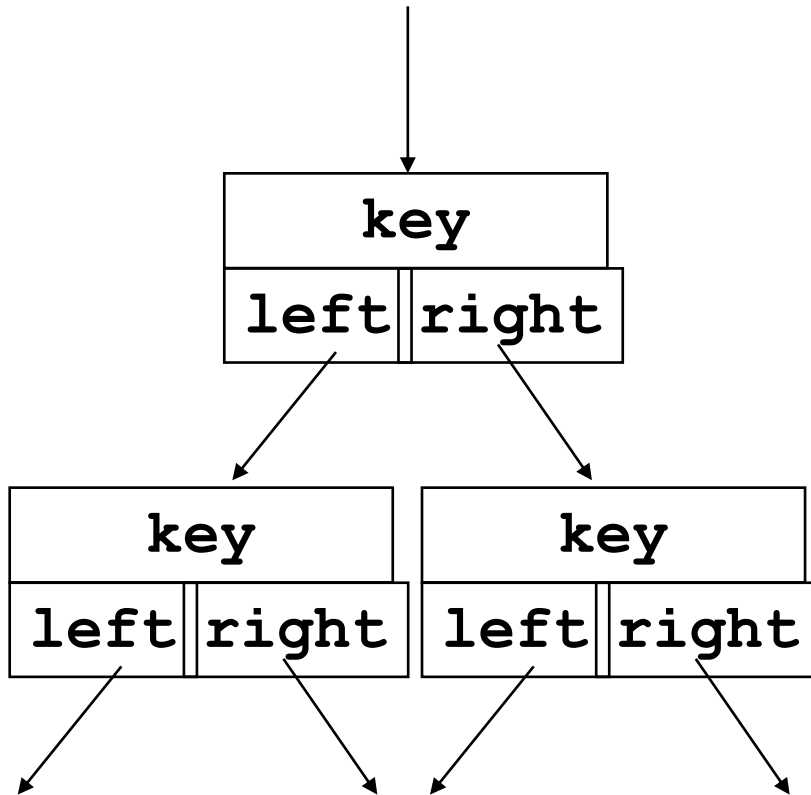
Kořenový binární vyhledávací strom (BVS)

- *Kořenový binární strom*
 - uzel má 0, (1), 2 následníky (nemusí být pravidelný)
- Binární vyhledávací strom (BVS)
 - uspořádaný kořenový binární strom
 - u je kořen
 - pro všechny uzly u_L z levého podstromu platí:
$$\text{klíč}(u_L) < \text{klíč}(u)$$
 - pro všechny uzly u_R z pravého podstromu platí:
$$\text{klíč}(u_R) > \text{klíč}(u)$$

Binární vyhledávací strom (BVS)



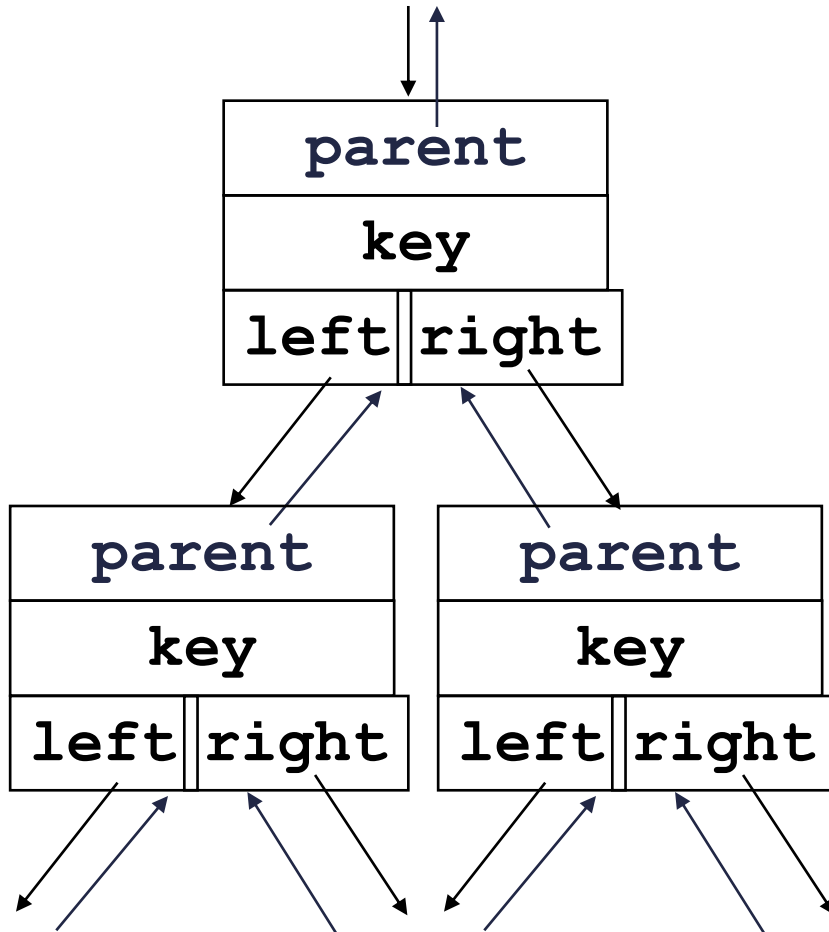
Reprezentace uzlů stromu



Výhodné pro:

- search
- min, max

Reprezentace uzlů stromu II.



Vhodné pro:

- search
- min, max
- pred, succ

Reprezentace uzlů stromu

```

public class Node {
    public Node left;
    public Node right;
    public int key;
    public Node(int k) {
        key = k;
        left = null;
        right = null;
        data = ...;
    }
}

public class Tree {
    public Node root;
    public Tree() {
        root = null;
    }
}

```

```

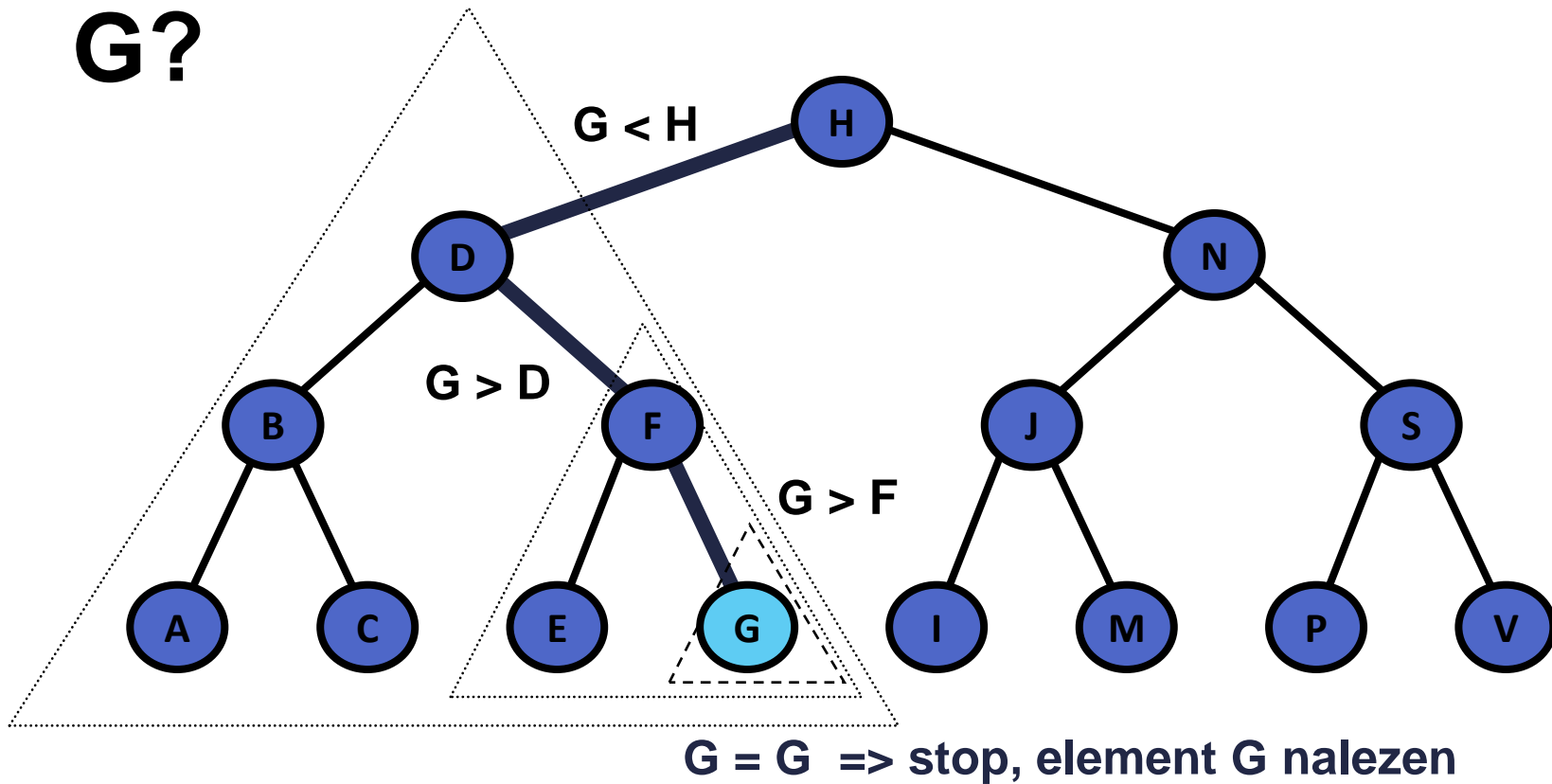
public class Node {
    public Node parent;
    public Node left;
    public Node right;
    public int key;
    public Node(int k) {
        key = k;
        parent = null;
        left = null;
        right = null;
        data = ...;
    }
}

public class Tree {
    ...
}

```

Prohledávání BVS

G?



Prohledávání BVS – rekurzivně/iterativně

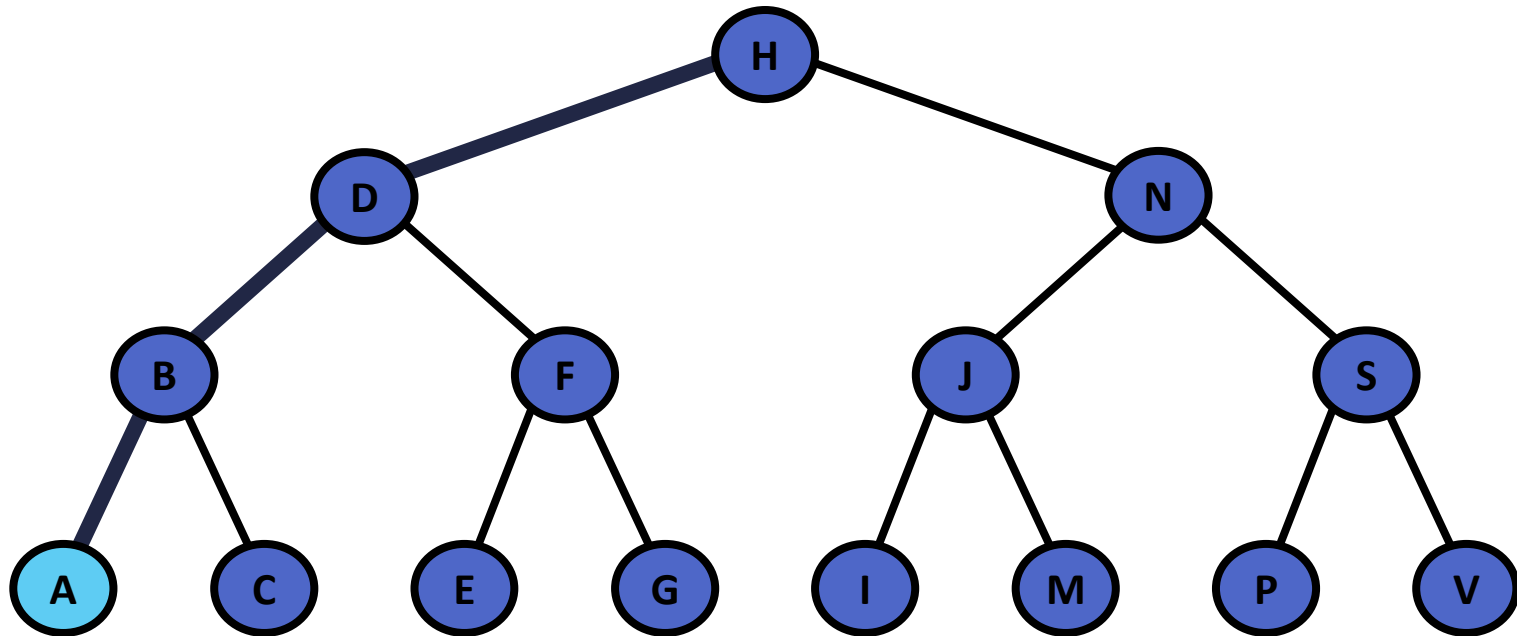
```
//Recursive version
Node treeSearch( Node x, key k )
{
    if(( x == null ) or ( k == x.key ))
        return x;
    if( k < x.key )
        return treeSearch( x.left, k );
    else
        return treeSearch( x.right, k );
}
```

Java-like pseudo code

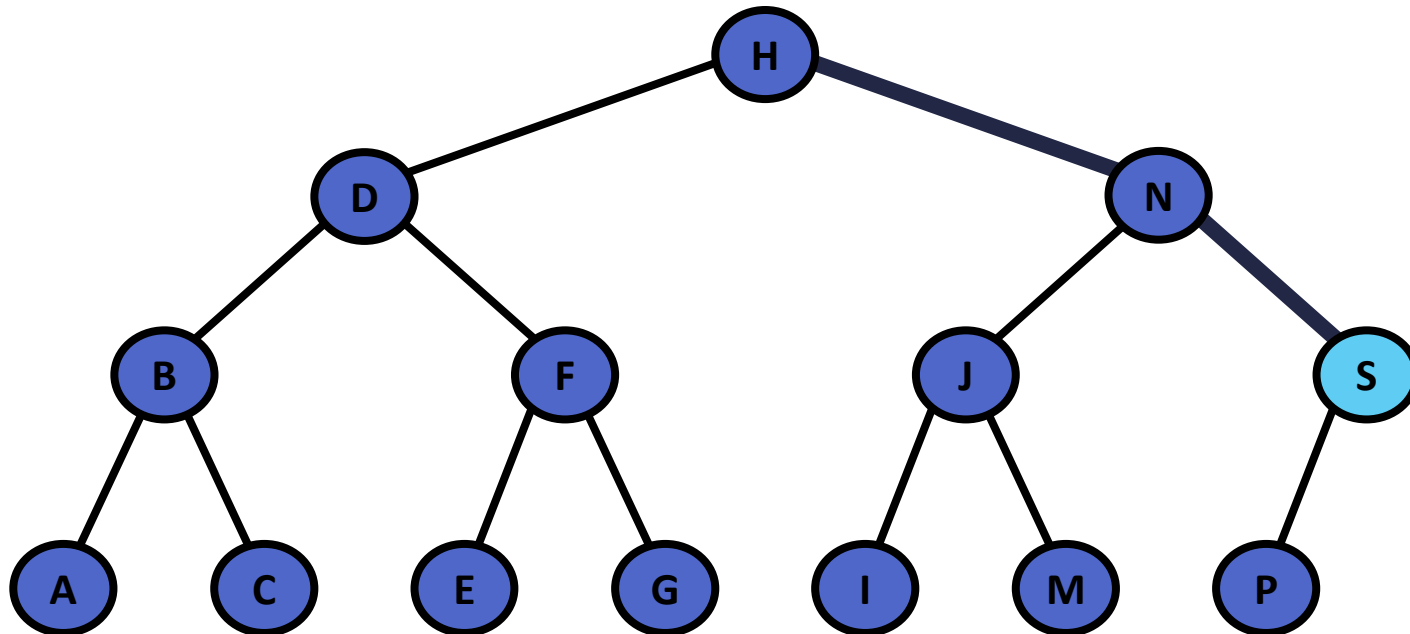
```
//Iterative version
Node treeSearch( Node x, key k )
{
    while(( x != null ) and (k != x.key ))
    {
        if( k < x.key ) x = x.left;
        else           x = x.right;
    }
    return x;}
}
```

Java-like pseudo code

Minimum pro BVS



Maximum pro BVS



Minimum a mazimum pro BVS – iterativně

```
Node treeMinimum( Node x )
{
    if( x == null ) return null;
    while( x.left != null )
    {
        x = x.left;
    }
    return x;
}
```

Java-like pseudo code

```
Node treeMaximum( Node x )
{
    if( x == null ) return null;
    while( x.right != null )
    {
        x = x.right;
    }
    return x;
}
```

Java-like pseudo code

Následník v BVS

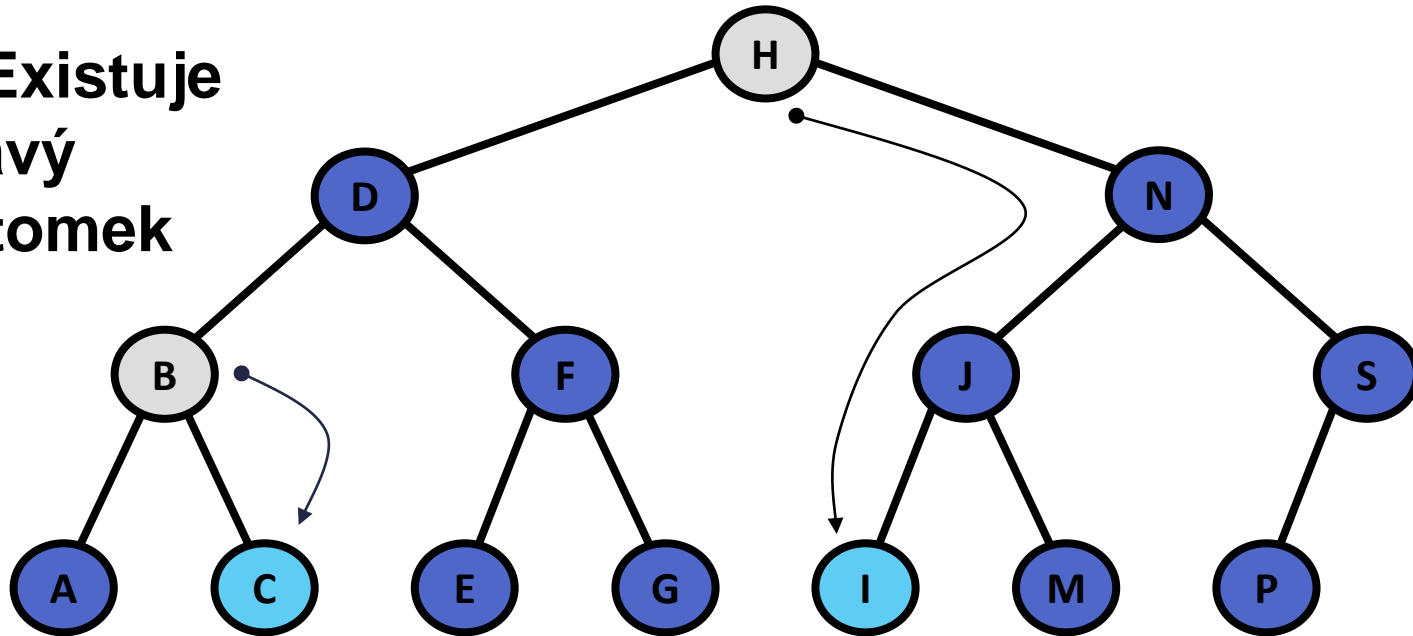
podle uspořádání (in-order tree walk)

- Jsou dvě možnosti:
 1. **Existuje pravý potomek**
 2. **Neexistuje pravý potomek**

Následník v BVS

podle uspořádání (in-order tree walk)

1. Existuje
pravý
potomek



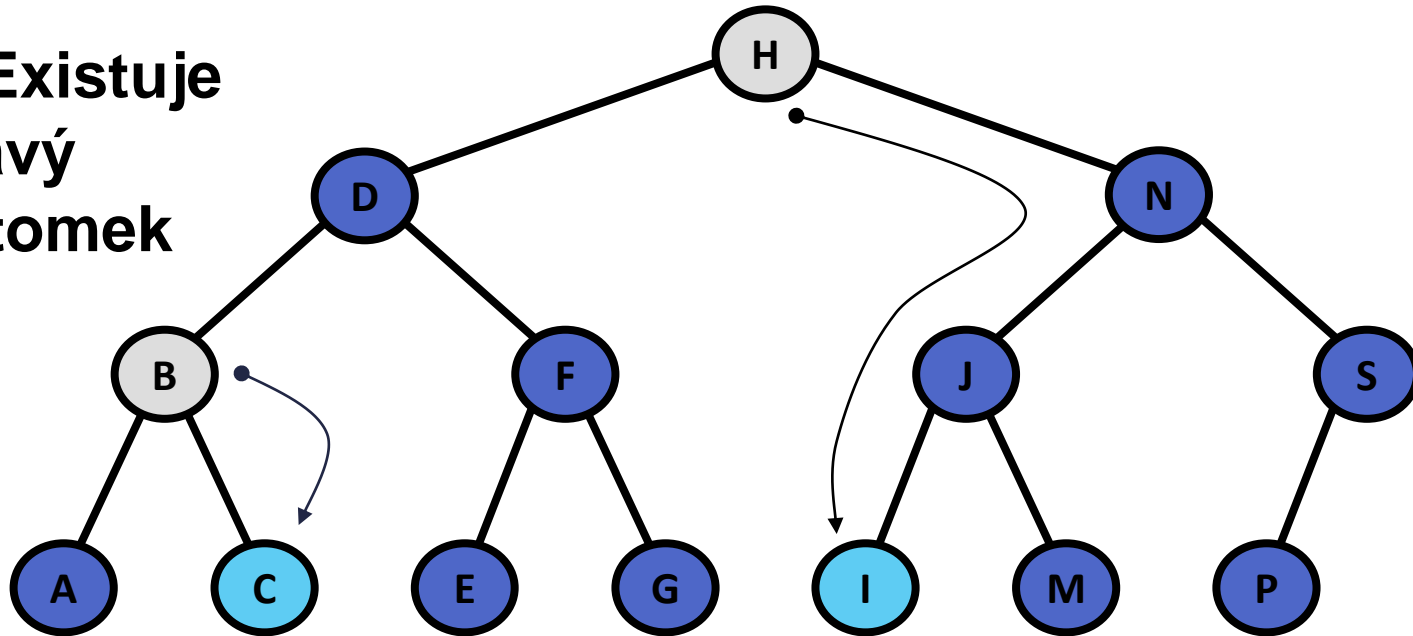
$\text{succ}(B) \rightarrow C$
 $\text{succ}(H) \rightarrow I$

Jak?

Následník v BVS

podle uspořádání (in-order tree walk)

1. Existuje
pravý
potomek



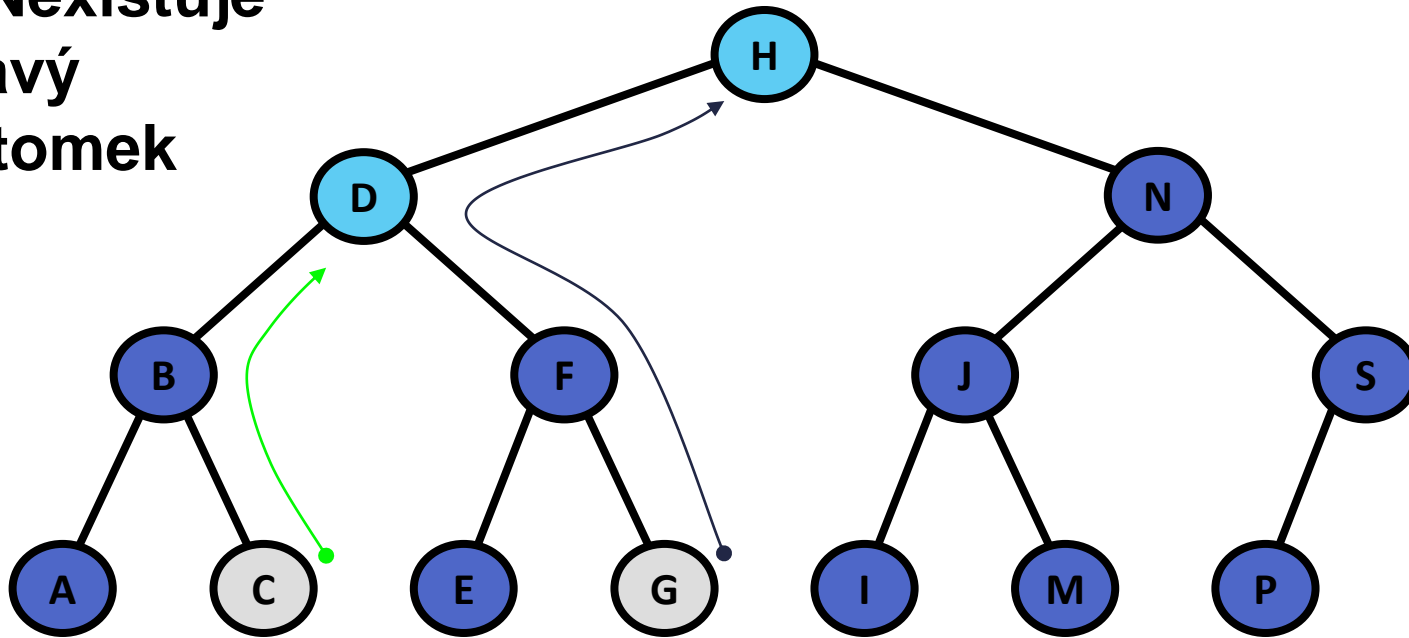
$\text{succ}(B) \rightarrow C$
 $\text{succ}(H) \rightarrow I$

Najdi *minimum* v pravé větvi
 = $\text{min}(x.\text{right})$

Následník v BVS

podle uspořádání (in-order tree walk)

1. Nexistuje
pravý
potomek

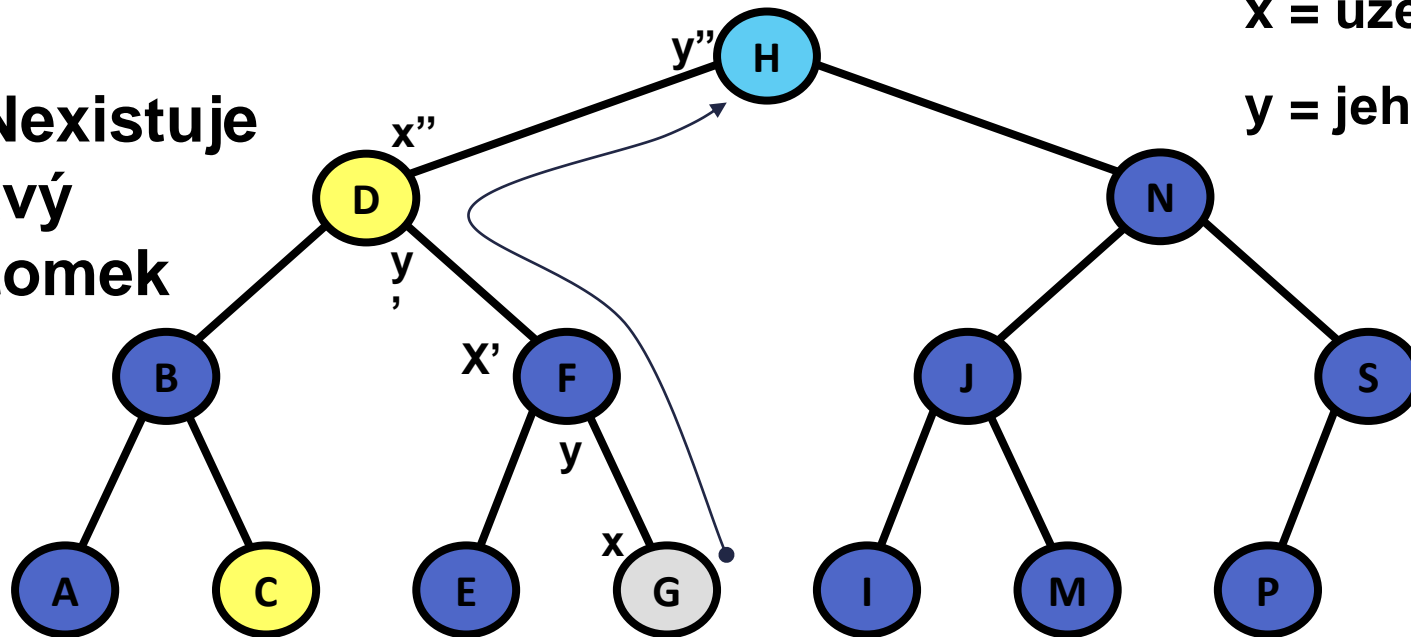


succ(C) \rightarrow D
succ(G) \rightarrow H) Jak?

Následník v BVS

podle uspořádání (in-order tree walk)

1. Nexistuje
pravý
potomek



x = uzel na cestě

y = jeho rodič

$\text{succ}(G) \rightarrow H$

Najdi *minimálního* rodiče vpravo
(minimální rodič na cestě do kořene)

Následník v BVS

podle uspořádání (in-order tree walk)

x = uzel na cestě, y = jeho rodič

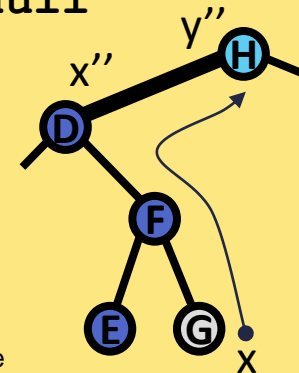
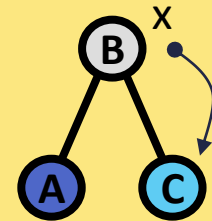
```

Node treeSuccessor( Node x )
{
  if( x == null ) return null;

  if( x.right != null ) // 1. right son exists
    return treeMinimum( x.right );

  y = x.parent;          // 2. right son is null
  while( (y != null) and (x == y.right))
  {
    x = y;
    y = x.parent;
  }
  return y;              // first parent x is left from
}

```



Java-like pseudo code

Předchůdce v BVS

podle uspořádání (in-order tree walk)

x = uzel na cestě, y = jeho rodič

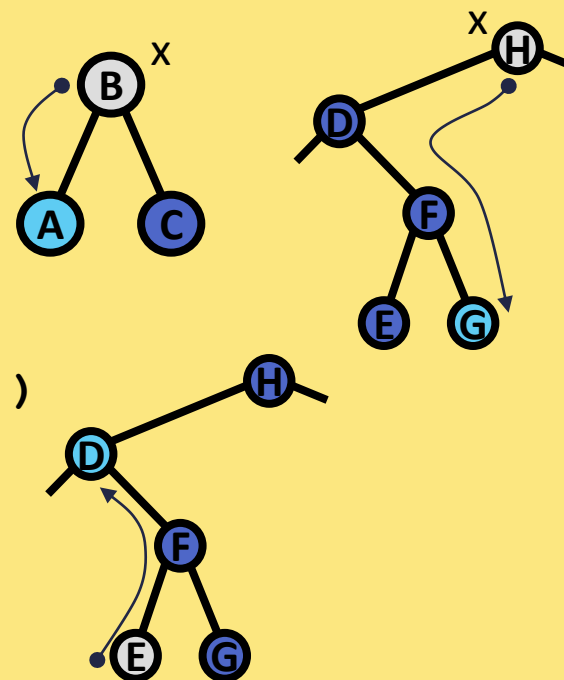
```

Node treePredecessor( Node x )
{
    if( x == null ) return null;

    if( x.left != null )
        return treeMaximum( x.left );

    y = x.parent;
    while( (y != null) and (x == y.left))
    {
        x = y;
        y = x.parent;
    }
    return y;
}

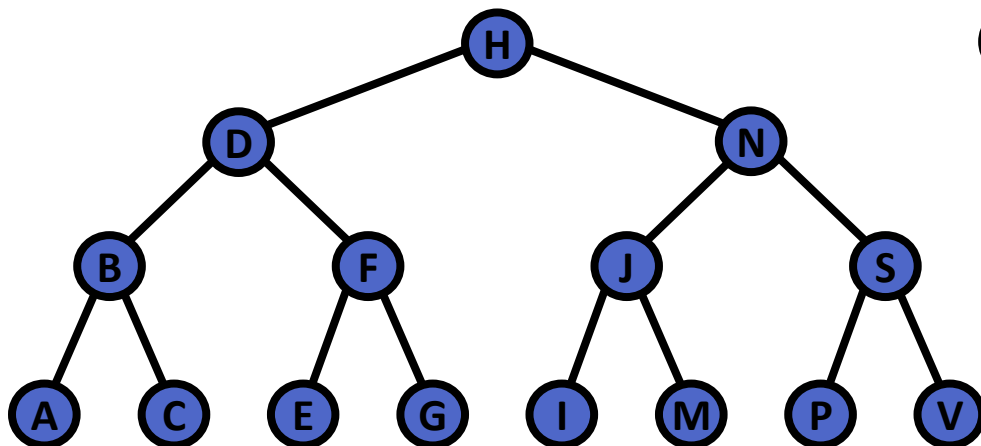
```



Java-like pseudo code

Operační složitost

Dynamické operace **search**, **max**, **min**, **succ**, **pred** mají operační složitost $O(h)$, kde h je hloubka vyhledávacího stromu.

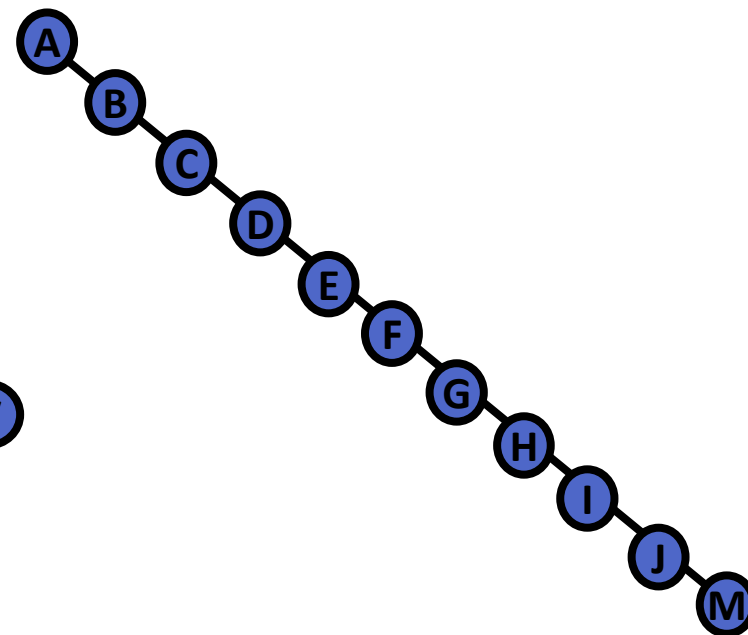


$$h = \log_2(n)$$

$$\Rightarrow O(\log(n))$$



\Rightarrow strom je třeba vyvážit!!!

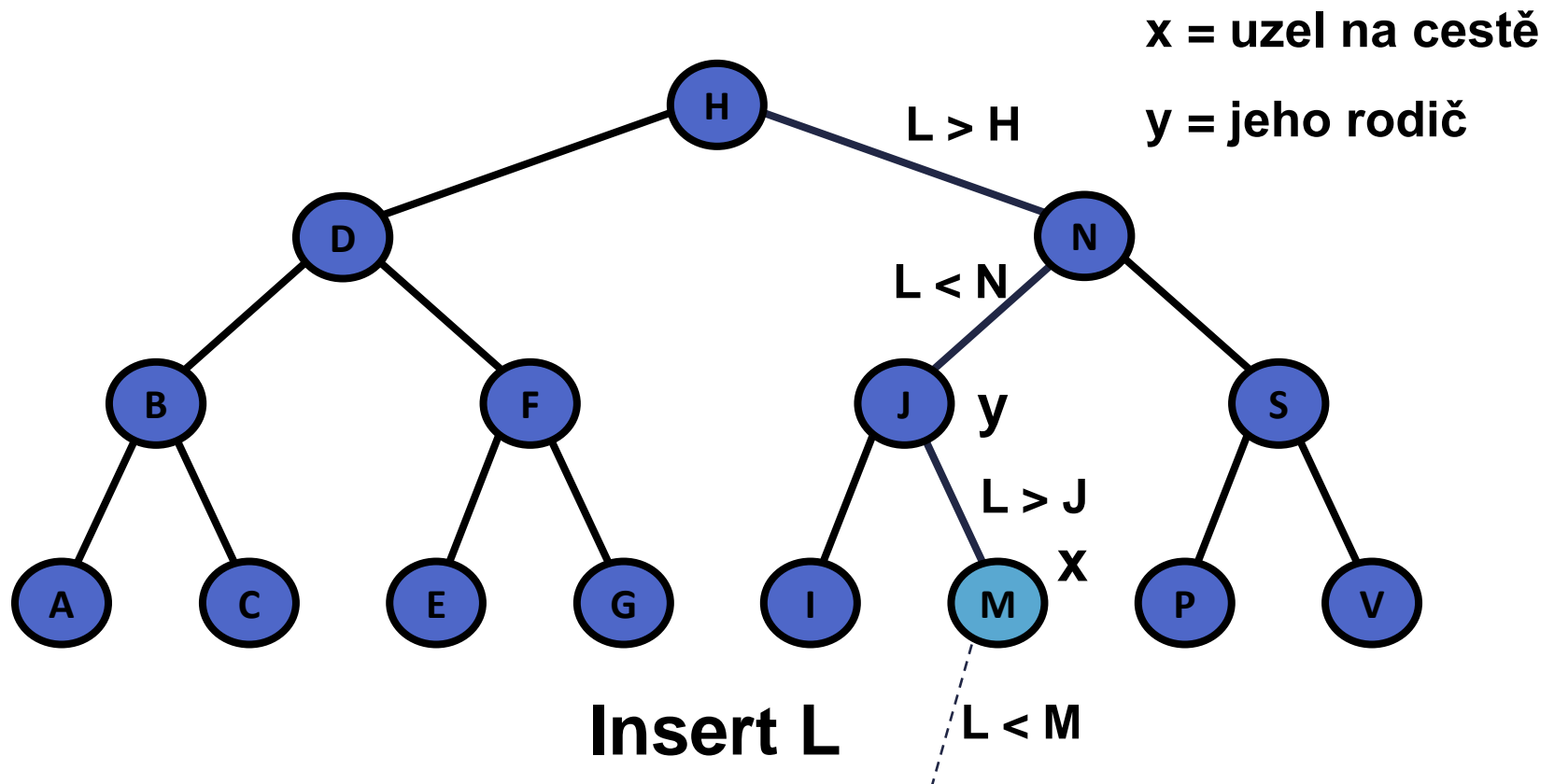


$$h = n$$

$$\Rightarrow O(n) !!!$$



Insert (vložení prvku)



1. Najdi list, který bude rodičem ... M
2. Připoj nový element jako nový list ... M.left

Insert (vložení prvku)

x = uzel na cestě, y = jeho rodič

```

void treeInsert( Tree t, Node e )
{
    x = t.root; y = null; // set x to tree root

    if( x == null )
        t.root = e; // tree was empty
    else {
        while(x != null) { // find the parent leaf
            y = x;
            if( e.key < x.key ) x = x.left;
            else x = x.right;
        }
        if( e.key < y.key ) y.left = e; // add e to parent y
        else y.right = e;
    }
}

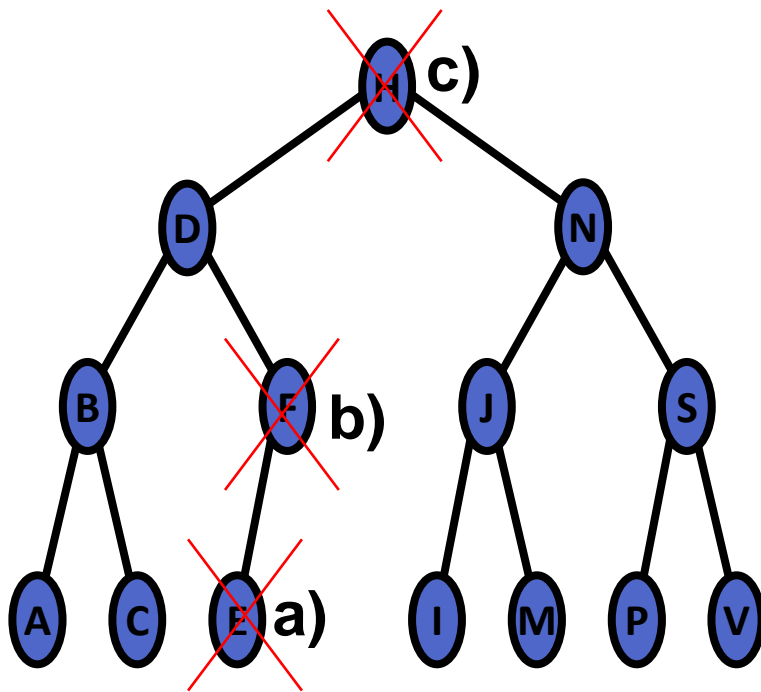
```

Java-like pseudo code

Jednoduchá verze – bez „update“ pro stejné klíče.

Operační složitost: najdi list + vložení = $O(\log_2 n) + O(1) = O(\log_2 n)$

Delete (odstranění prvku)

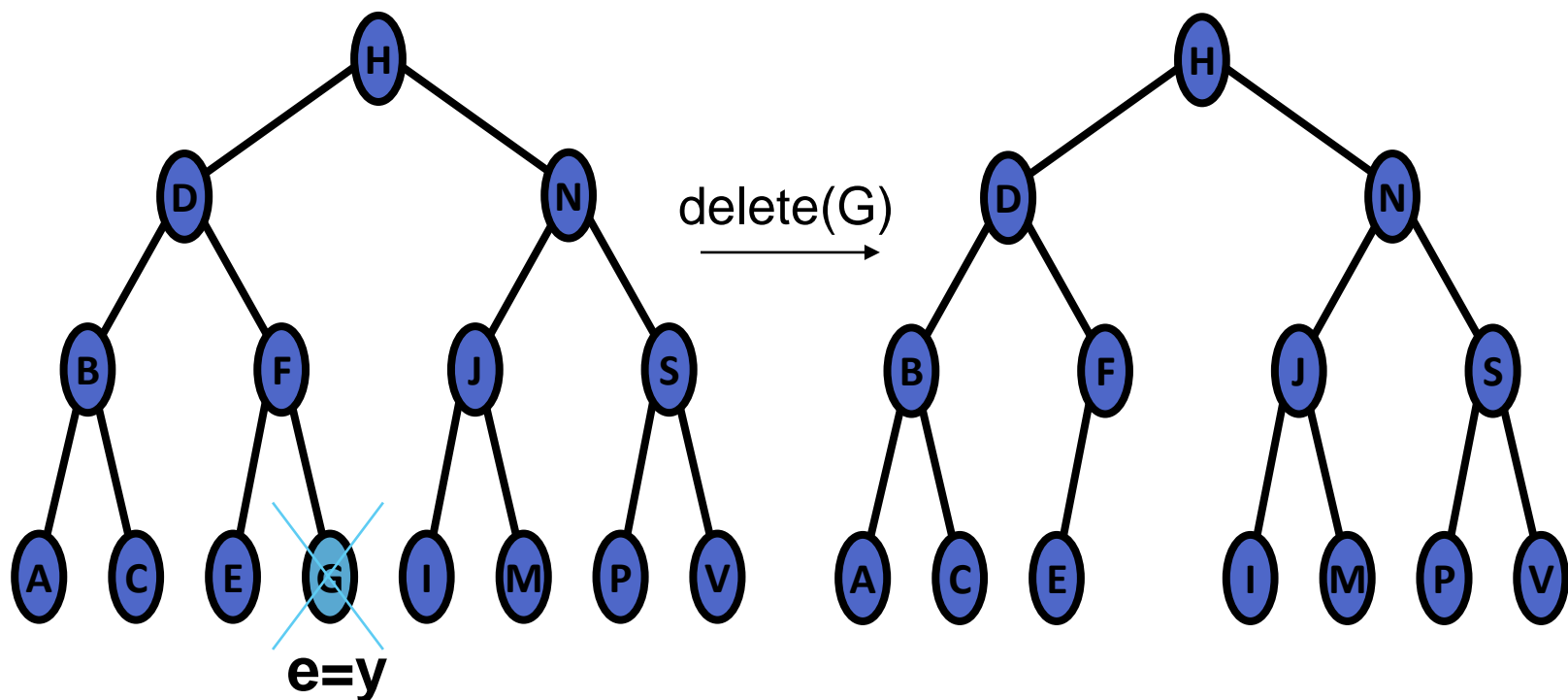


Delete – 3 případy

- rušíme list, který nemá potomky
- rušíme list, který má jednoho potomka
- rušíme list, který má dva potomky

Delete (odstranění prvku)

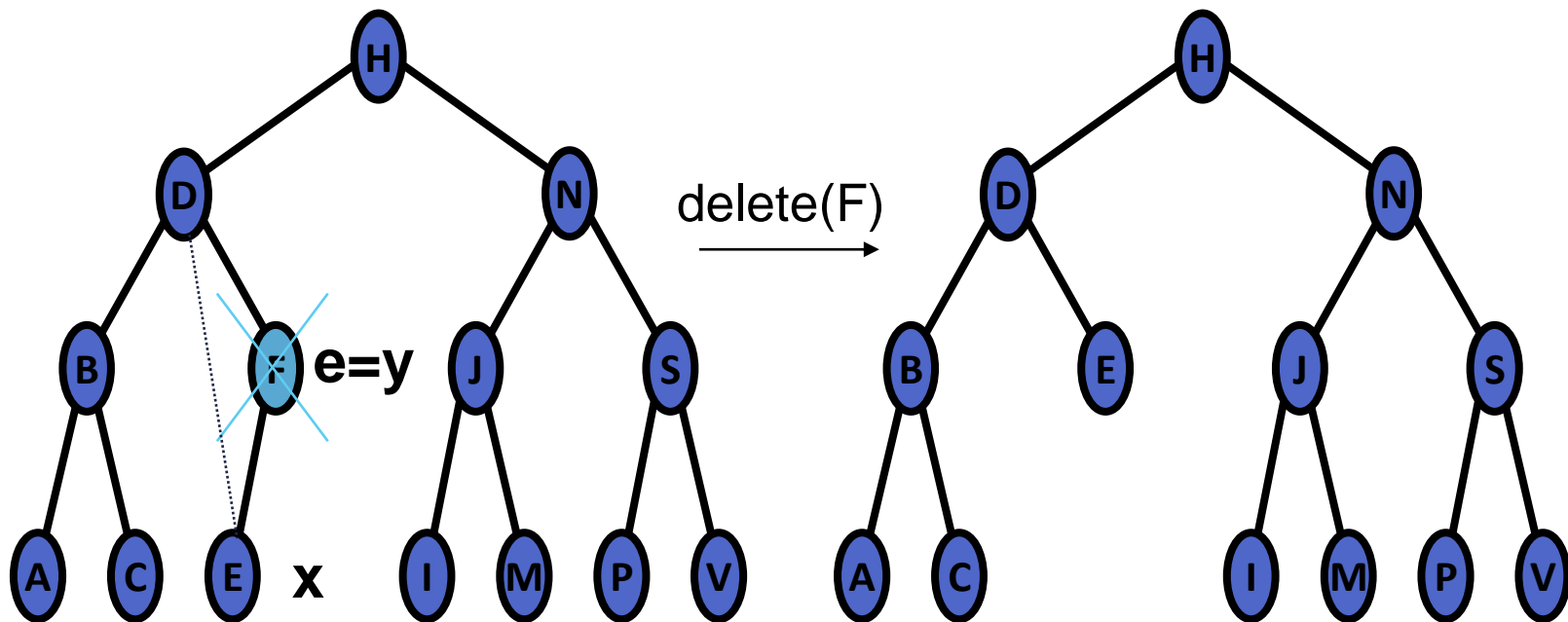
a) smaž list



a) list bez potomků lze jednoduše vypustit

Delete (odstranění prvku)

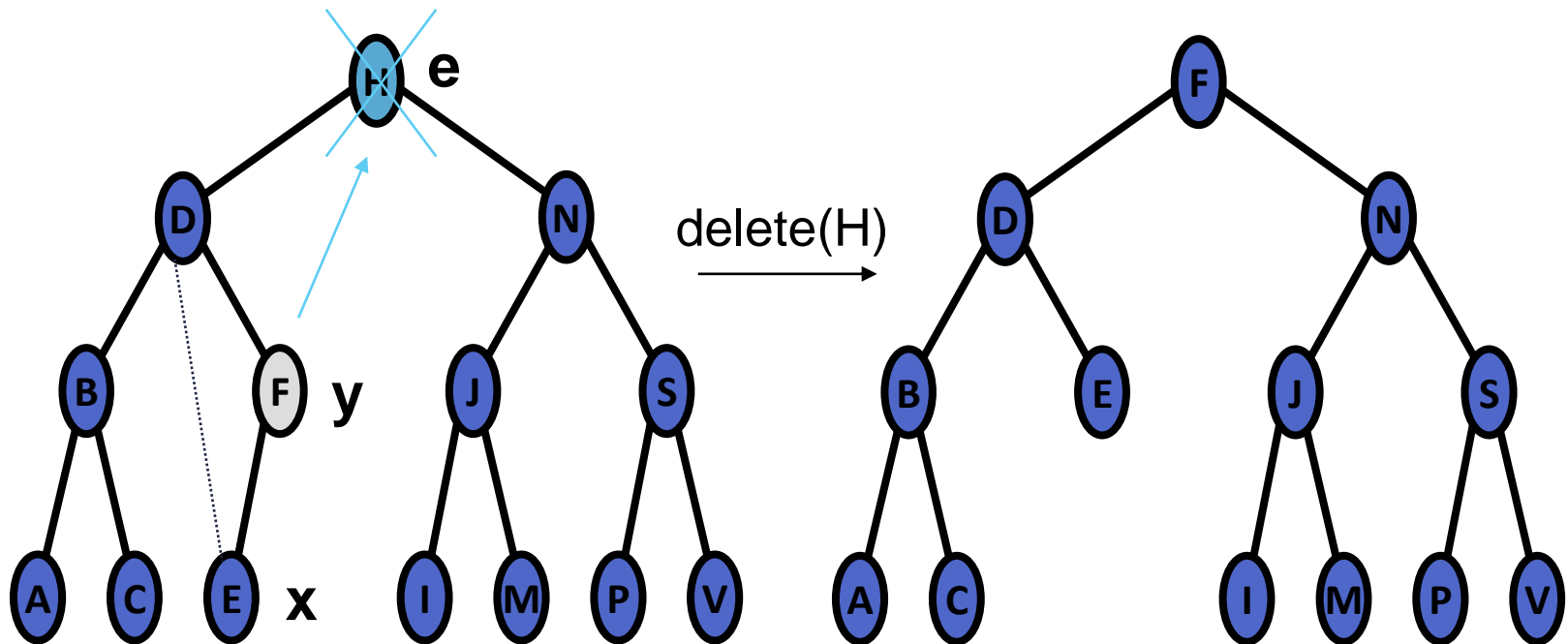
b) vnitřní s potomkem



b) uzel s jedním potomkem - přemostí vymazaný uzel

Delete (odstranění prvku)

c) se 2 potomky



c) uzel se dvěma potomky -> nahrad' uzel jeho předchůdcem (**pred**) a vypust' předchůdce, nebo následníkem (**succ**) a vypust' následníka

Delete (odstranění prvku)

```
Node treeDelete( Tree t, Node e )  
// e...node to logically delete  
// y...node to physically delete  
// x..y' s only son
```

```
{ Node x, y;
```

1. find node y (e or predecessor of e)
2. find $x = y$'s only child or null
3. link x up with parent of y
4. link parent of y down to x
5. replace e by in-order predecessor y
6. return y (for later use)

```
}
```

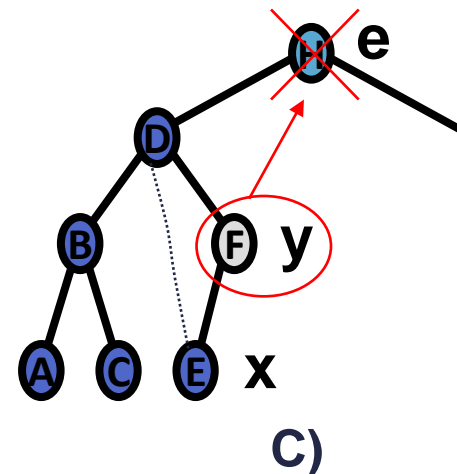
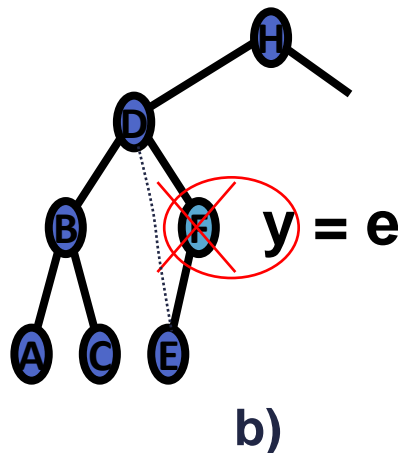
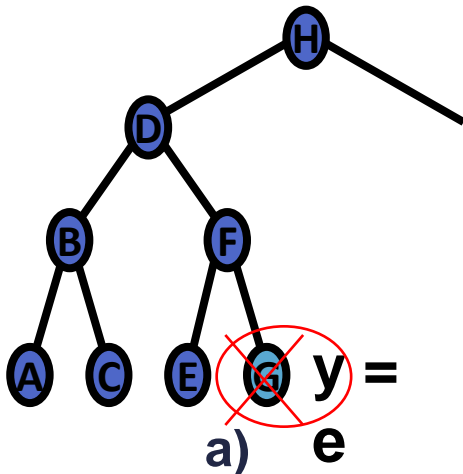
Delete (odstranění prvku) - pokr.

```
Node treeDelete( Tree t, Node e )
// e...node to logically delete
// y...node to physically delete
// x...y's only son
```

```
{ Node x, y;
```

```
1.find node y
```

```
if(e.left == null OR e.right == null)
    y = e; // cases a,b) 0 - 1 child
else
    y = TreePredecessor(e); // case c) 2 children
                                cont...
```



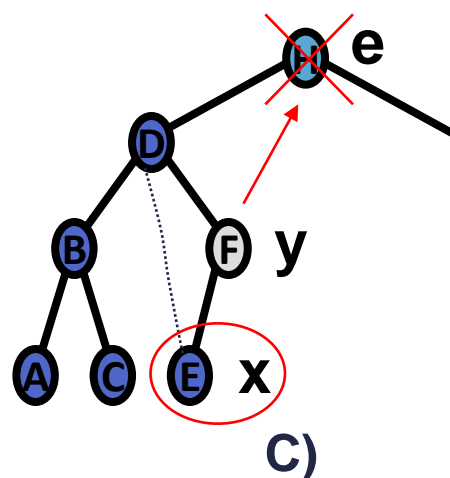
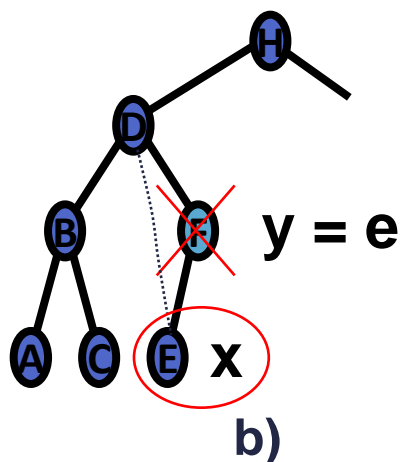
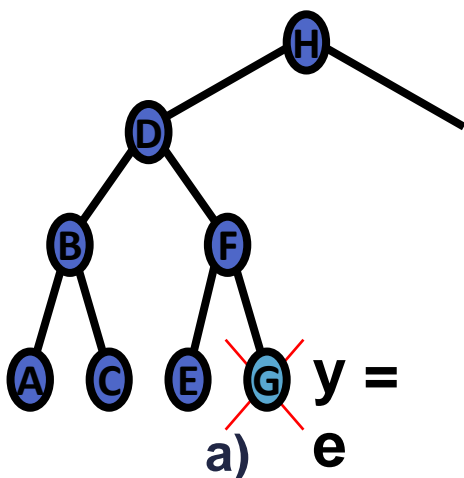
Delete (odstranění prvku)

... cont

```
2. find x = y's only child or null
```

```
if( y.left != null )// a) null, b,c) only child
    x = y.left;
else
    x = y.right;
```

cont...



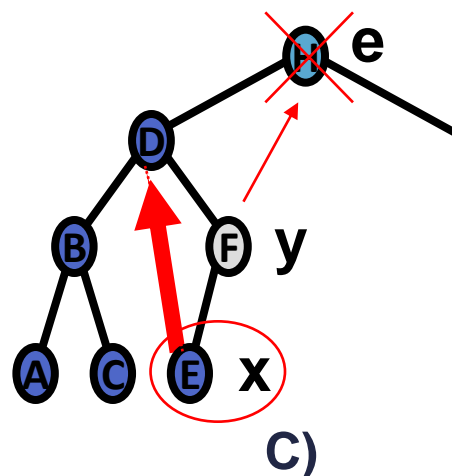
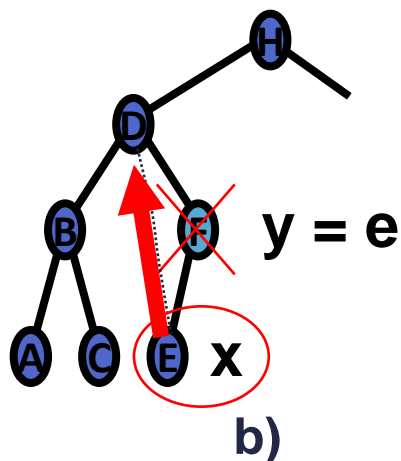
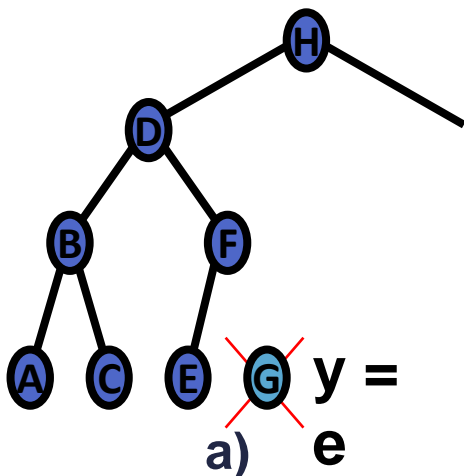
Delete (odstranění prvku)

... cont

```
3. link x up with its new parent (former parent of y)
```

```
if( x != null ) x.parent = y.parent; // b,c)
```

cont ...



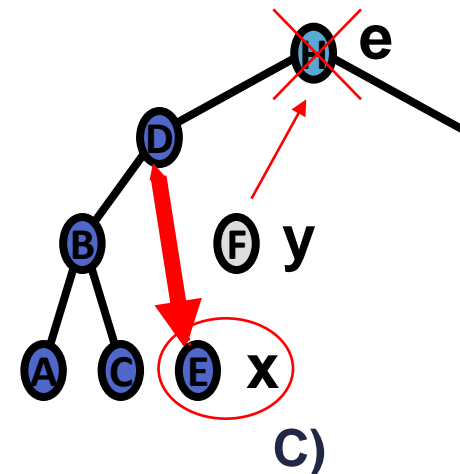
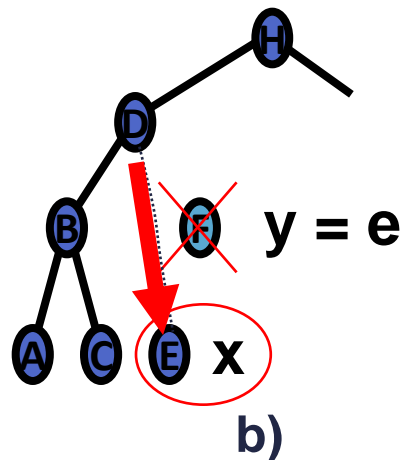
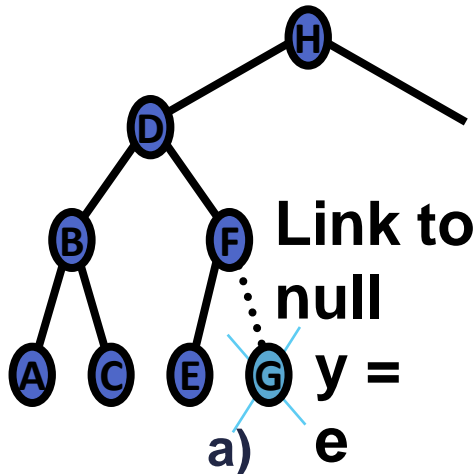
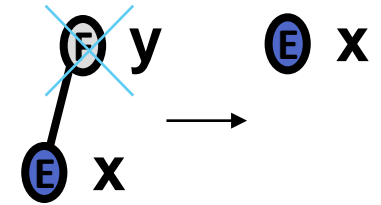
Delete (odstranění prvku)

4. link parent of y down to x

```

if( y.parent == null )
    t.root = x // y was root
else if( y == (y.parent).left )
    (y.parent).left = x; // y was left son
else
    (y.parent).right = x; // y was right son
    
```

cont...



Delete (odstranění prvku)

...

5. replace e with in-order predecessor

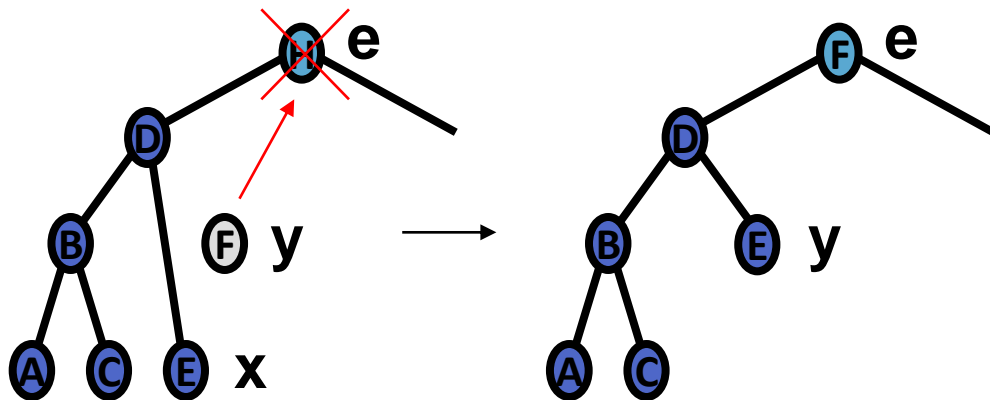
```

if( y != e ) // replace e with in-order predecessor
{
    e.key = y.key;           // copy the key
    e.data = y.data;        // copy other fields too
}

```

6. return y (for later use)

}



Delete (celkový kód)

```

Node treeDelete( Tree t, Node e )
{ Node x, y;
// e..node to logically delete
// y...node to physically delete, x...y's only son

    if(e.left == null OR e.right == null)
        y = e; // cases a, b) 0 to 1 child
    else y = TreePredecessor(e); // c) 2 children
    if( y.left != null ) // a) null, b,c) only child
        x = y.left;
        else x = y.right;
    if( x != null ) x.parent = y.parent; // b,c)
    if( y.parent == null ) t.root = x // y-root
    else if( y == (y.parent).left ) (y.parent).left = x; // y-L son
        else (y.parent).right = x; // y-R son
    if( y != e ) { // replace e with in-order predecessor
        e.key = y.key;
        e.dat = y.data; // copy other fields too
    }
    return y; // instead of delete
}

```

Vyvažování stromu (Tree balancing)

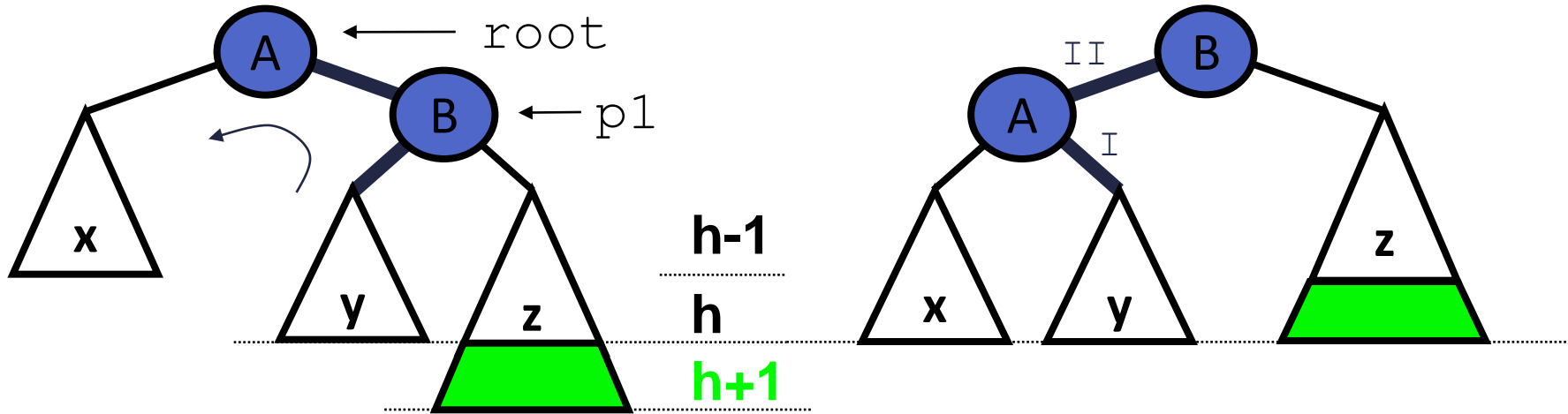
Operace nad BVS jsou závislé na míře vyváženosti stromu – pro vyvážený strom dosahují $O(\log_2 n)$, pro nevyvážený $O(n)$.

Vyváženost:

- Silná podmínka – shoda h (Ideální případ)
Pro všechny uzly platí:
počet uzlů vlevo = počet uzlů vpravo
- Slabší podmínka – násobek h
 - **výška** podstromů - AVL strom (jméno pochází z iniciál jeho objevitelů)
 - **výška** + počet potomků - 1-2 strom, ...
 - **váha** podstromů (počty uzlů) - váhově vyvážený strom
 - stejná **černá výška** – Červeno-černý strom

Pomocné operace pro udržení vyvážení stromu – rotace

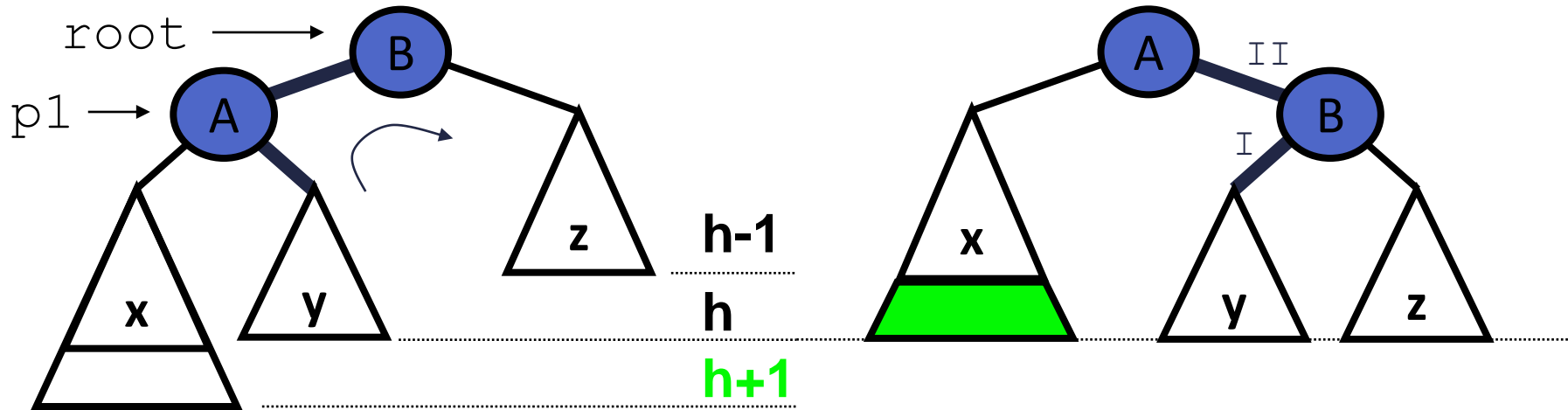
L rotace (Left rotation)



```
Node leftRotation( Node root ) { // subtree root!!!
    if( root == null ) return root;
    Node p1 = root.right;      (init)
    if (p1 == null) return root;
    root.right = p1.left;     (I)
    p1.left = root;          (II)
    return p1;
}
```

Java-like pseudo code

R rotace (right rotation)



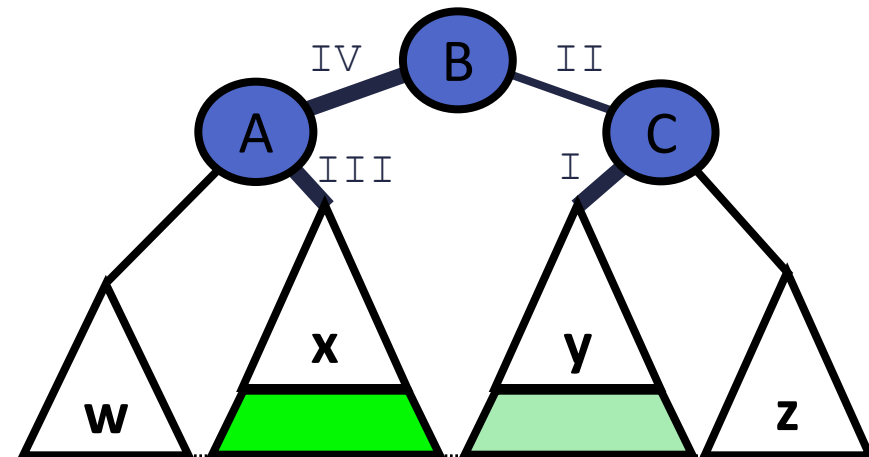
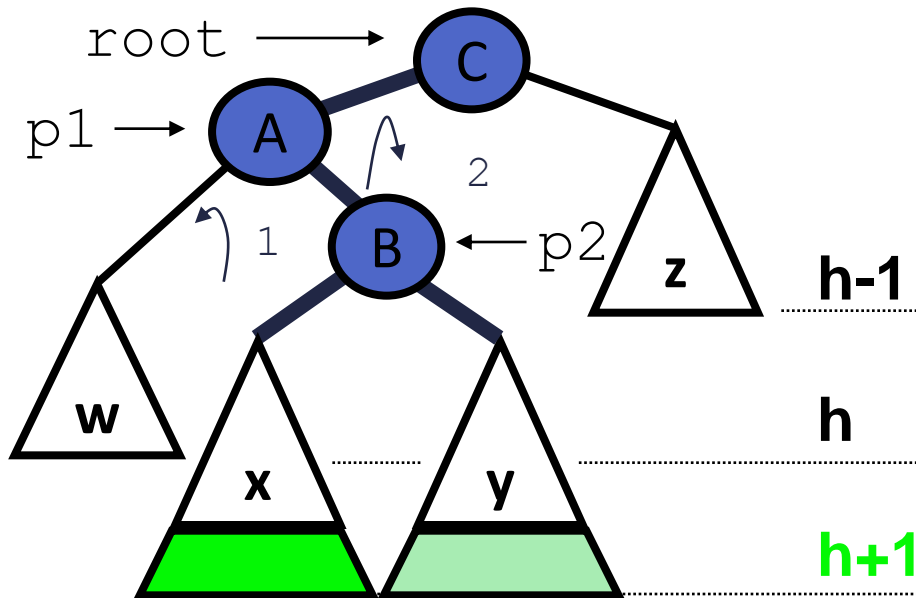
```

Node rightRotation( Node root ) { // subtree root!!!
    if( root == null ) return root;
    Node p1 = root.left;          (init)
    if (p1 == null) return root;
    root.left = p1.right;        (I)
    p1.right = root;             (II)
    return p1;
}

```

Java-like pseudo code

LR rotace (left-right rotation)



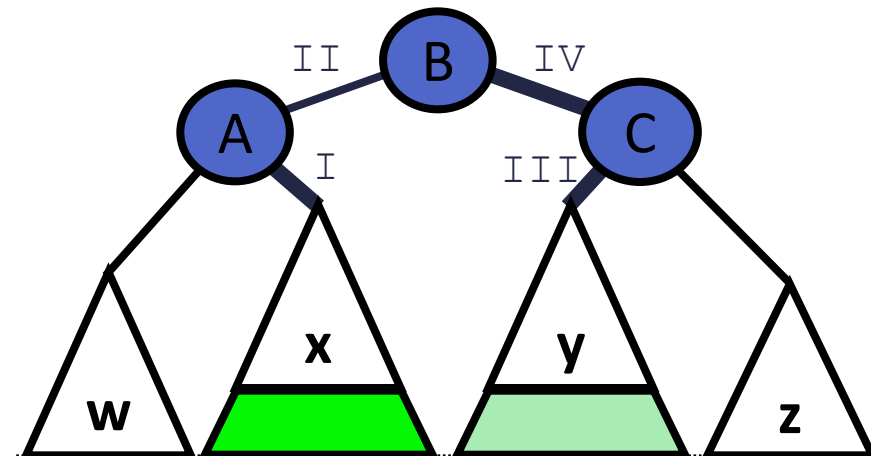
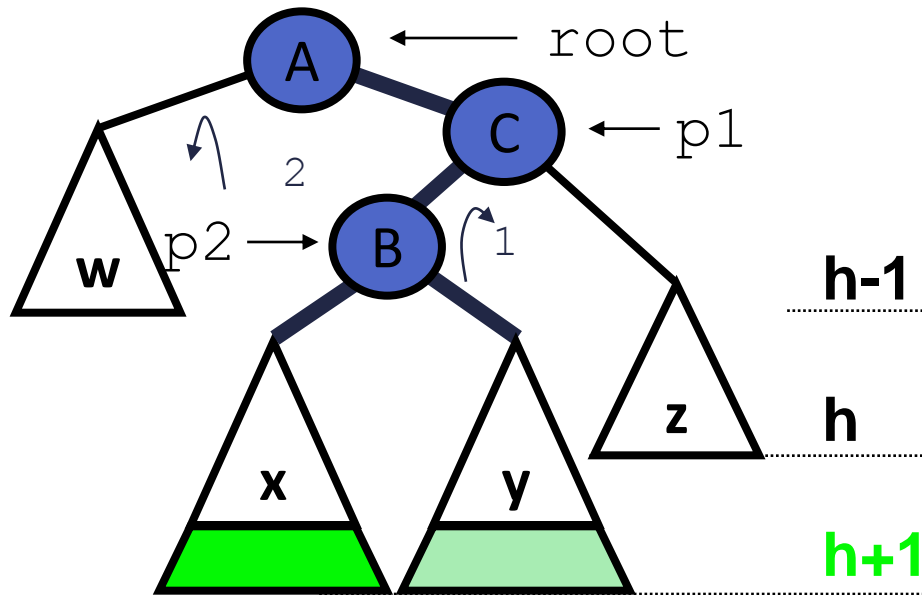
```

Node leftRightRotation ( Node root ) {  if(root==null)..;
Node p1 = root.left; Node p2 = p1.right;    (init)
root.left = p2.right; (I)
p2.right = root;      (II)
p1.right = p2.left;   (III)
p2.left = p1;         (IV)
return p2;    }

```

Java-like pseudo code

RL rotace (right- left rotation)



```

Node rightLeftRotation( Node root ) { if(root==null)....;
  Node p1 = root.right; Node p2 = p1.left;      (init)
  root.right = p2.left; (I)
  p2.left = root;      (II)
  p1.left = p2.right;  (III)
  p2.right = p1;      (IV)
  return p2;      }

```

Java-like pseudo code

Kdy použijeme kterou rotaci?

- AVL strom [Richta90]
 - Výškově vyvážený strom (Georgij Maximovič **Adelson-Velskij** a Evgenij Michajlovič **Landis**)
 - Výška:
 - Prázdný strom: výška = -1
 - neprázdný: výška = výška delšího potomka
 - Vyvážený strom:
rozdíl výšek potomků **bal** = {-1, 0, 1}

AVL strom (AVL tree)

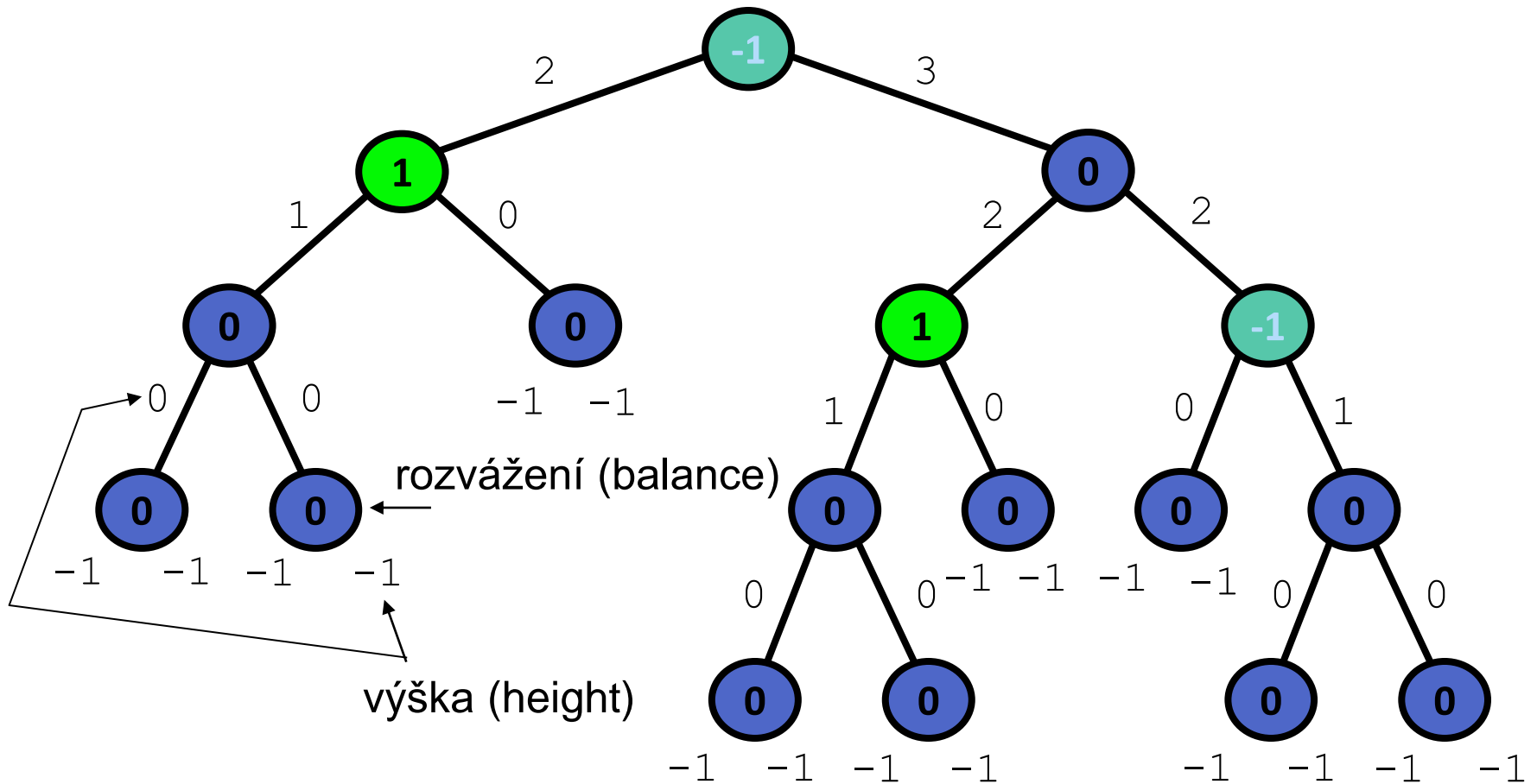
- // A very inefficient recursive definition

```
int height( Node t )
{
    if( t == null )
        return -1;    //leaf
    else
        return 1 + max( height( t.left ),
                       height( t.right ) );
}
```

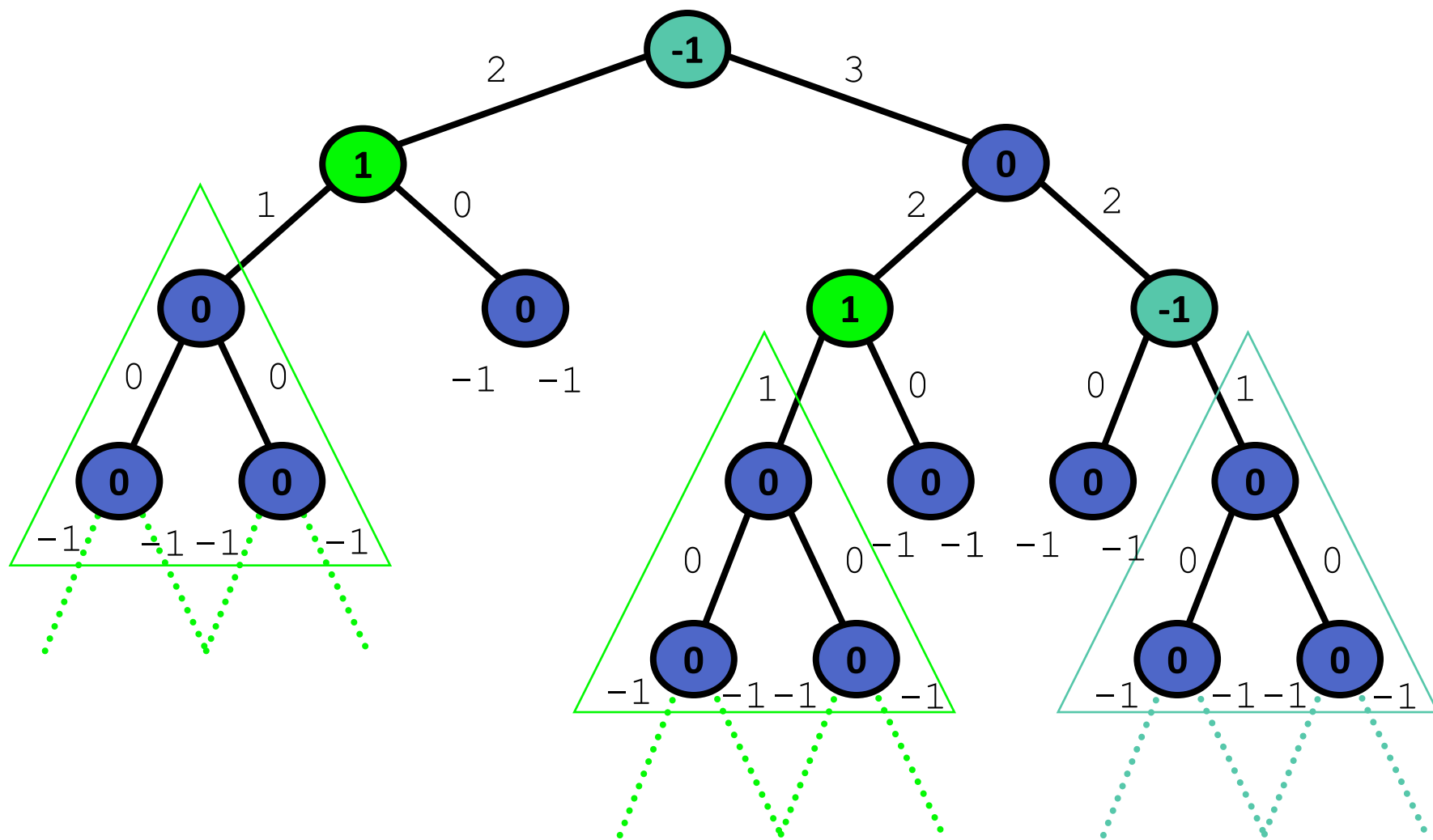
```
int bal( Node t )
{
    return height( t.left ) - height( t.right );
}
```

Java-like pseudo code

AVL strom - výšky a rozvážení

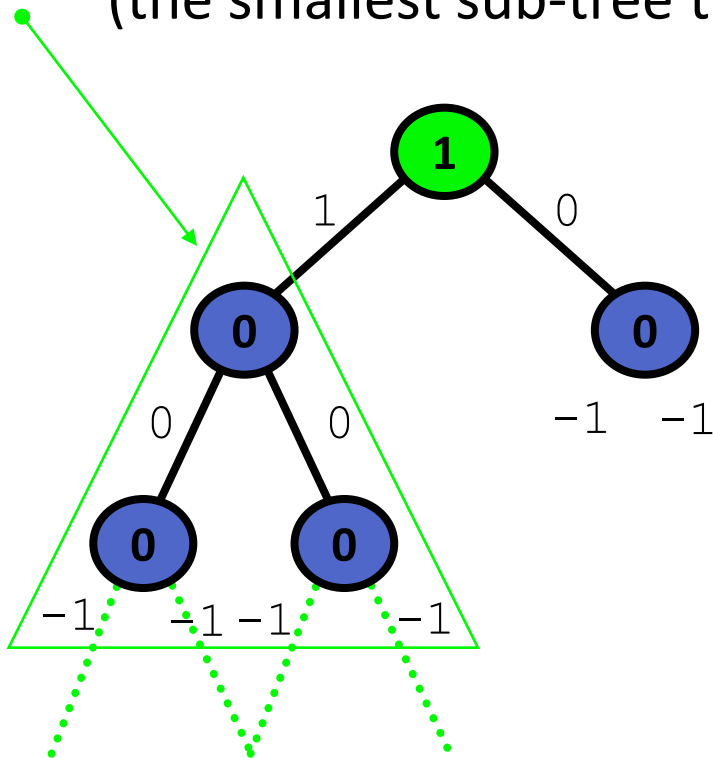


AVL strom před vložením uzlu



AVL strom - nejmenší podstrom

Nejmenší podstrom, který se přidáním uzlu rozváží z 0
(the smallest sub-tree that loses its balance = 0 by insertion)



- Je vyvážený: $bal = 0$
- Po operaci insert zůstává vyvážený

$$bal \in \langle -1, +1 \rangle$$

Rodič (parent)

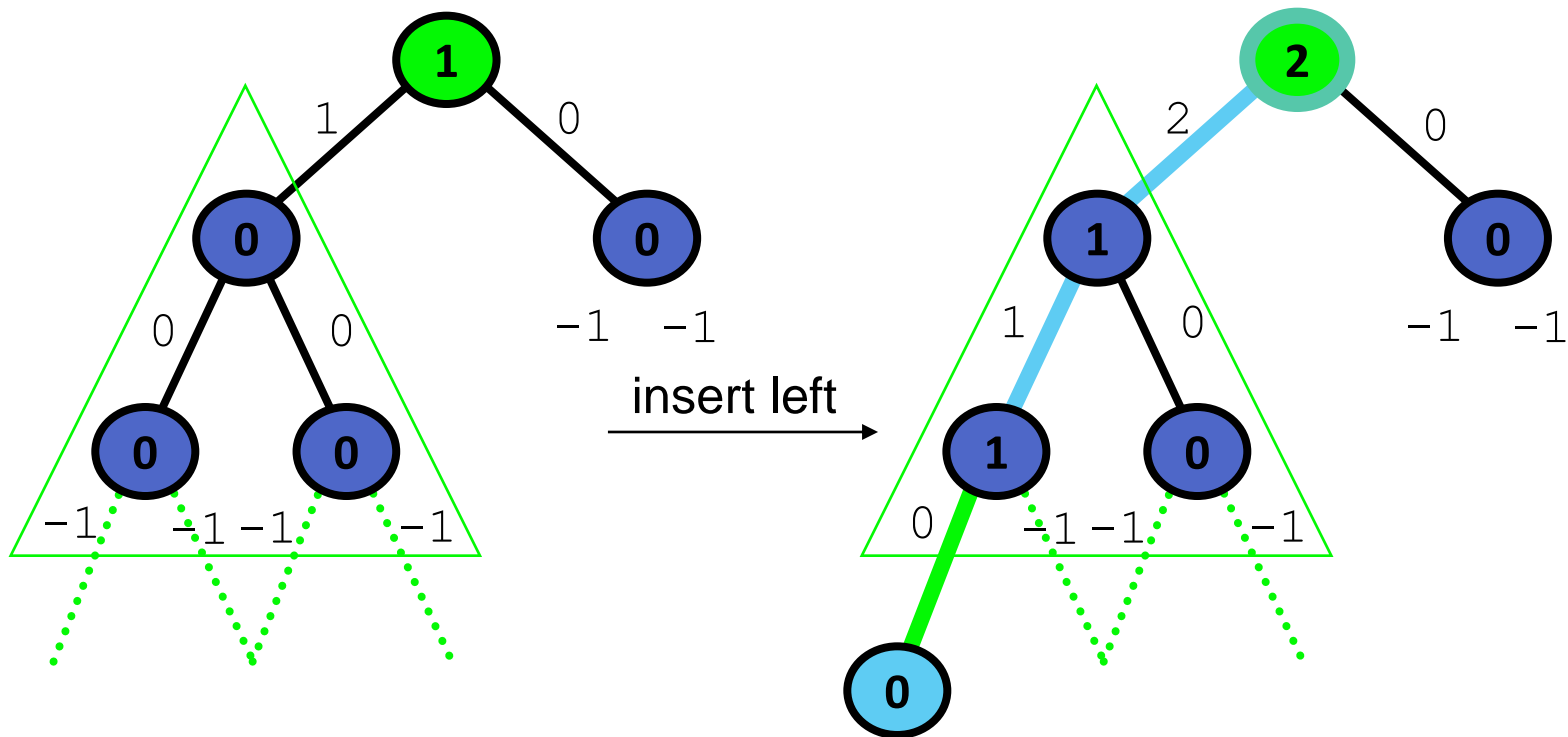
- $1-1 \rightarrow 0$ OK
- $1+1 \rightarrow 2$ není vyvážený

...

Nejmenší – modifikace blízko k listům

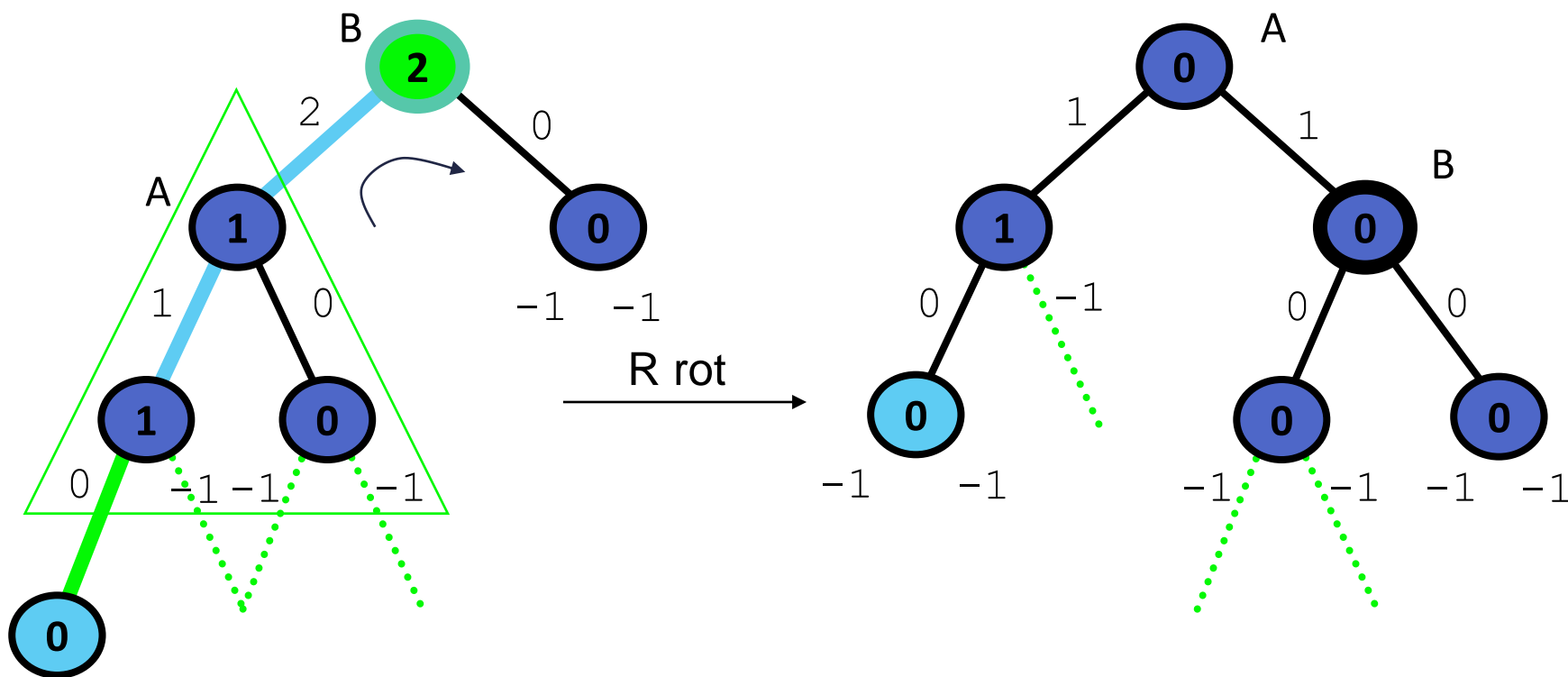
AVL strom - vložení uzlu doleva

- a) Podstrom se přidáním uzlu doleva rozváží
(the sub-tree loses its balance by node insertion – left)



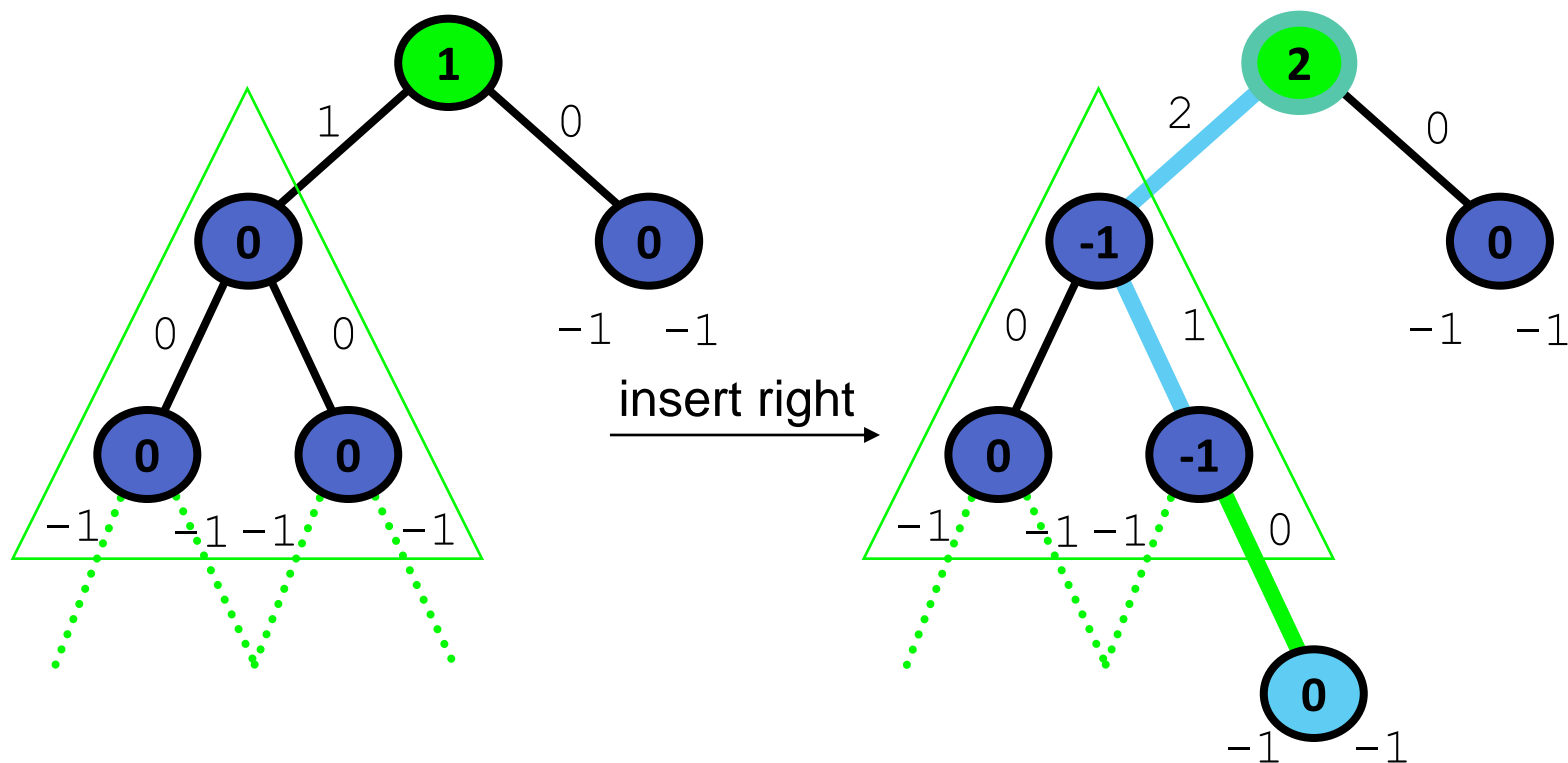
AVL strom - pravá rotace

- a) Vložen doleva – doleva => korekce pravou rotací
(node inserted to the left – left => balance by Right rotation)



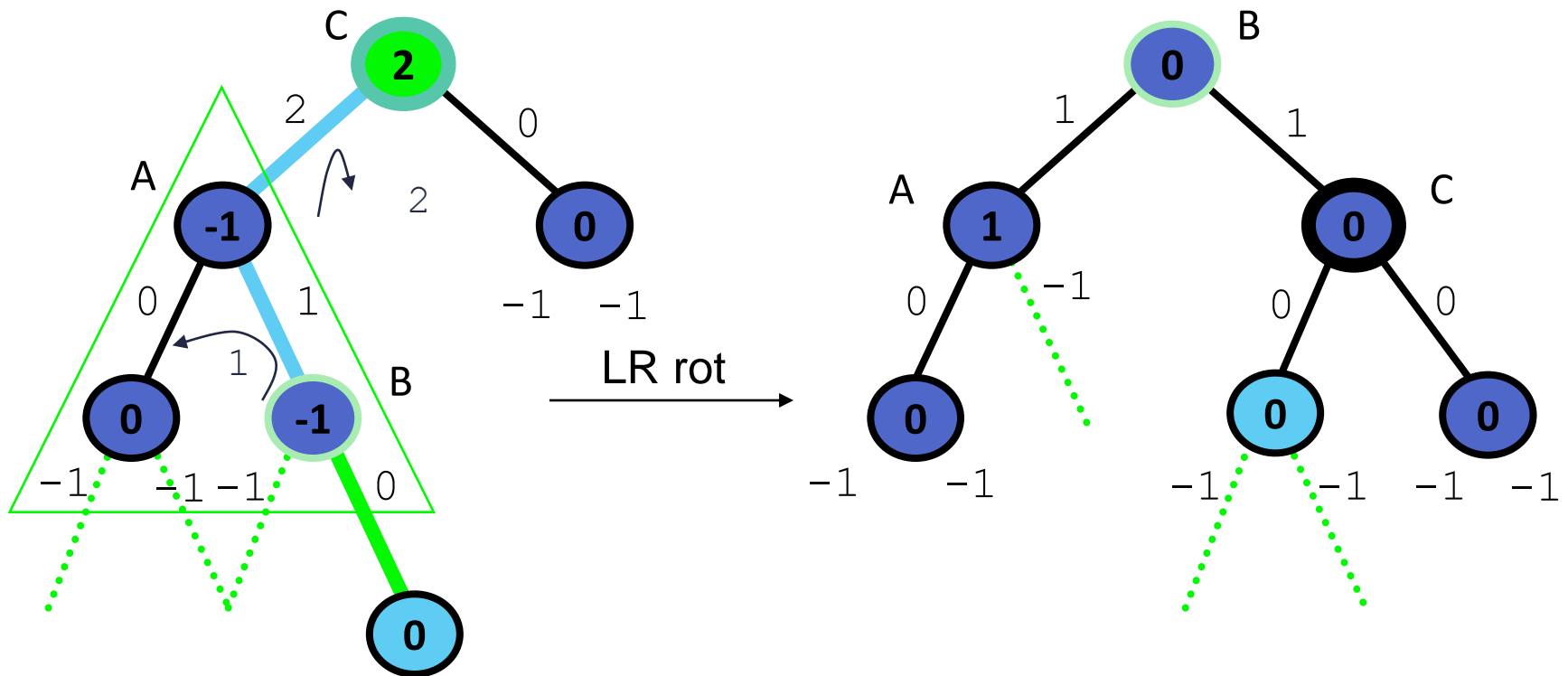
AVL strom - vložení uzlu doprava

- b) Podstrom se přidáním uzlu doprava rozváží
 (The sub-tree loses its balance by node insertion – right)



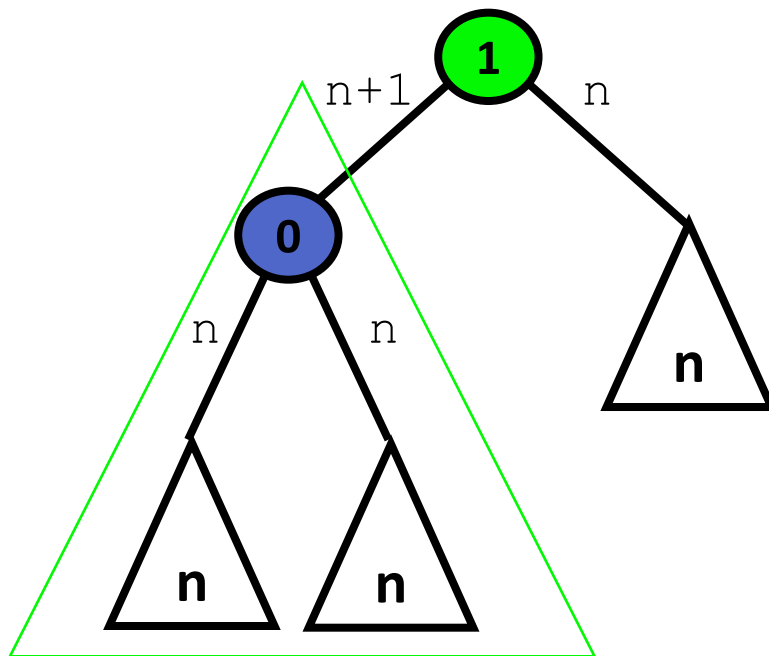
AVL strom - pravá rotace


- b) Vložen doleva – doprava \Rightarrow korekce LR rotací
 (Node inserted left – right \Rightarrow balance by the LR rotation)

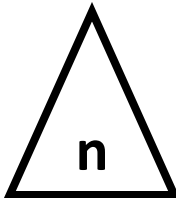



AVL strom - nejmenší podstrom

Nejmenší podstrom, který se přidáním uzlu rozevře
(The smallest sub-tree loses its balance by insertion)



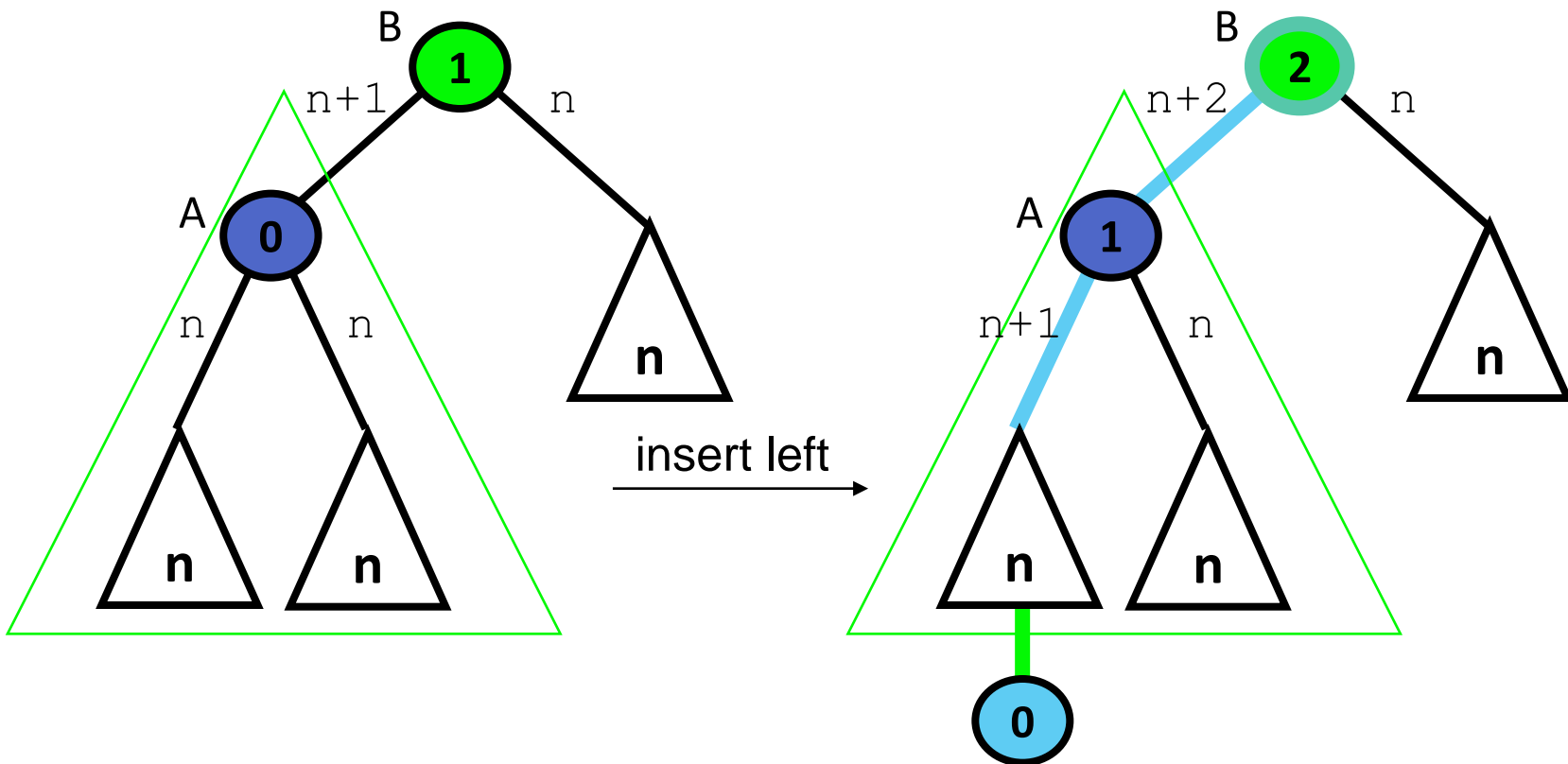
 Uzel s vyvážením 0
(Node with balance 0)

 Podstrom výšky h
(Sub-tree of height n)

 K podstromu výšky n
(Sub-tree below of height n)

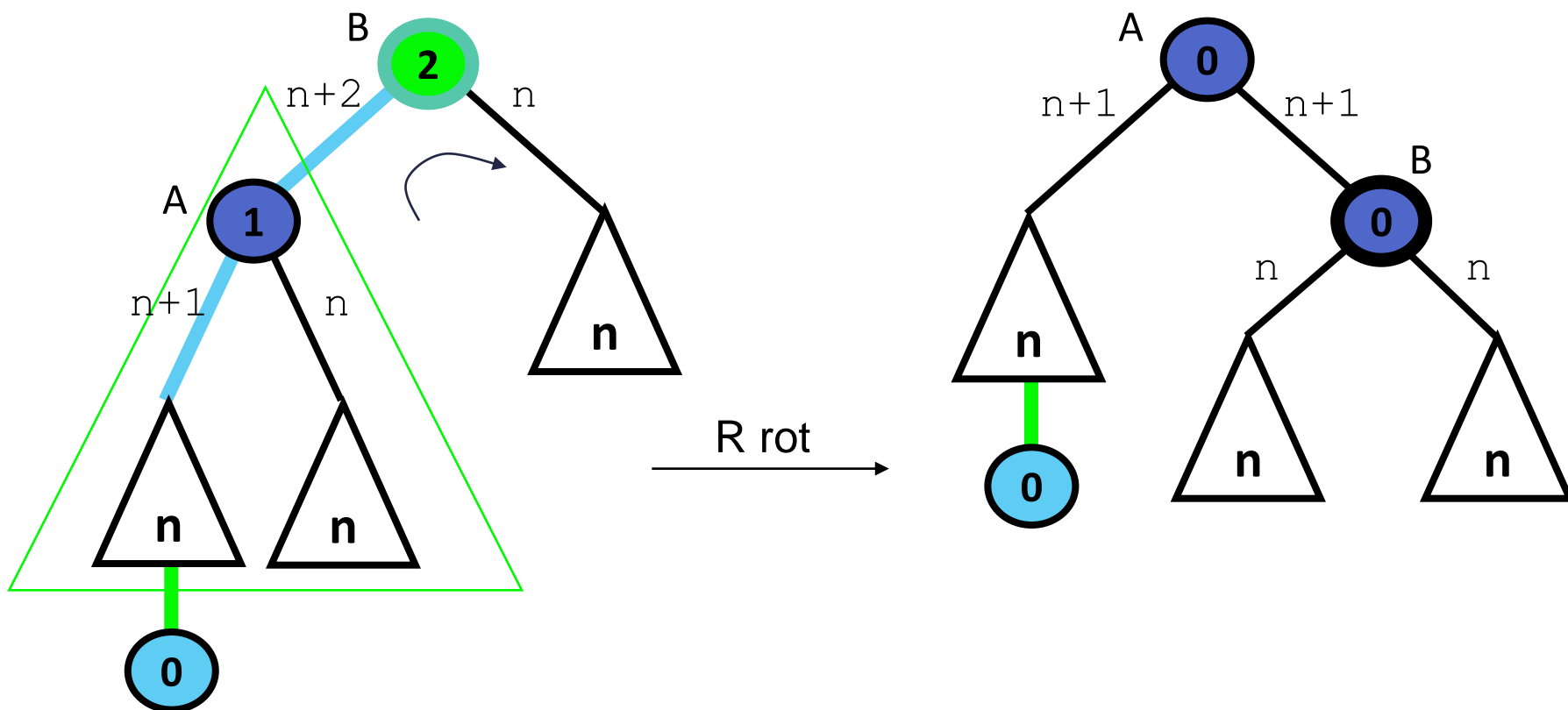
AVL strom - vložení uzlu doleva

- a) Podstrom se přidáním uzlu doleva rozváží
(The sub-tree loses its balance by node insertion – left)



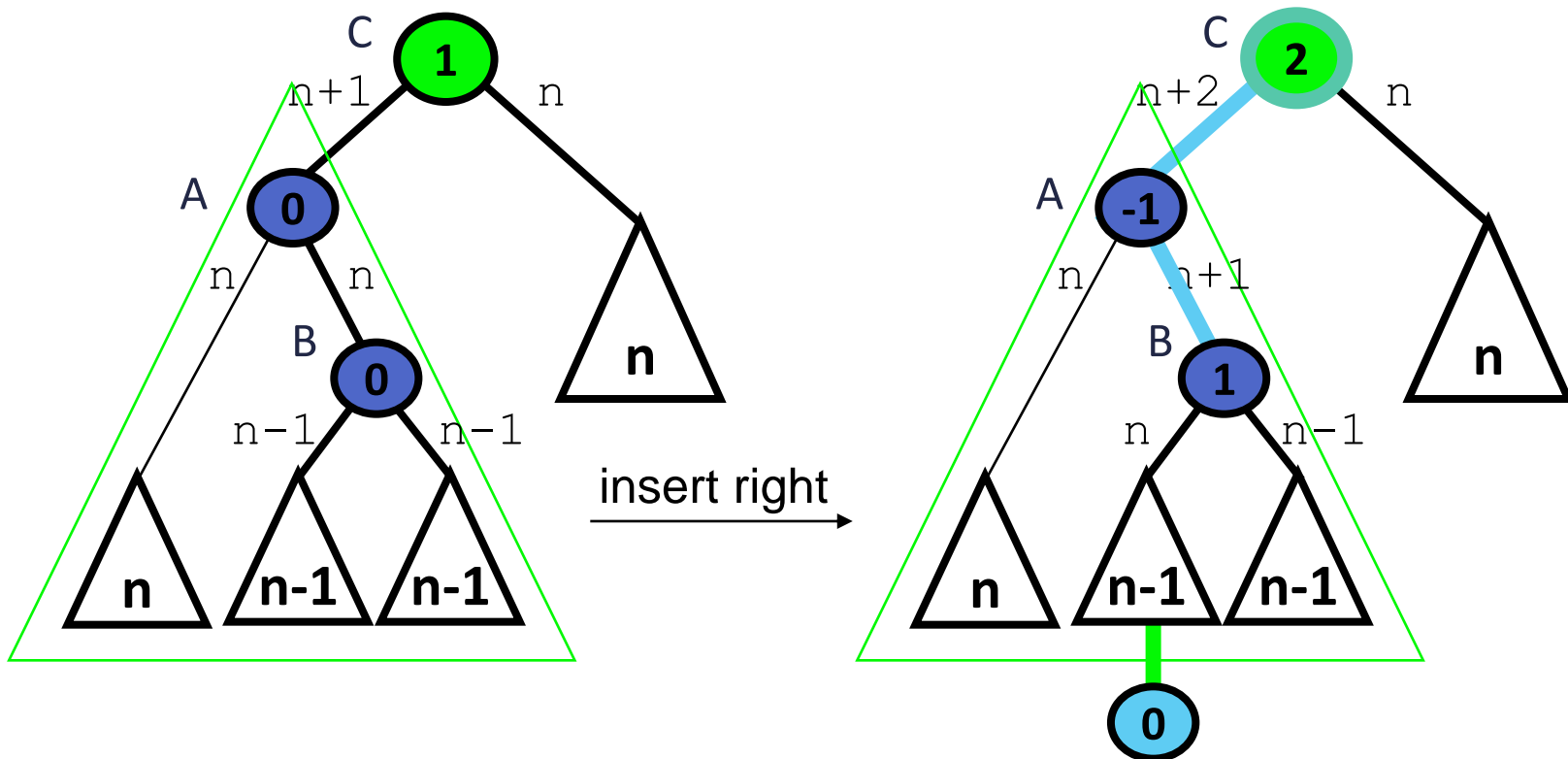
AVL strom - pravá rotace

- a) Vložen doleva – doleva => korekce pravou rotací (R rotací)
 (Node inserted to the left – left => balance by right rotation)



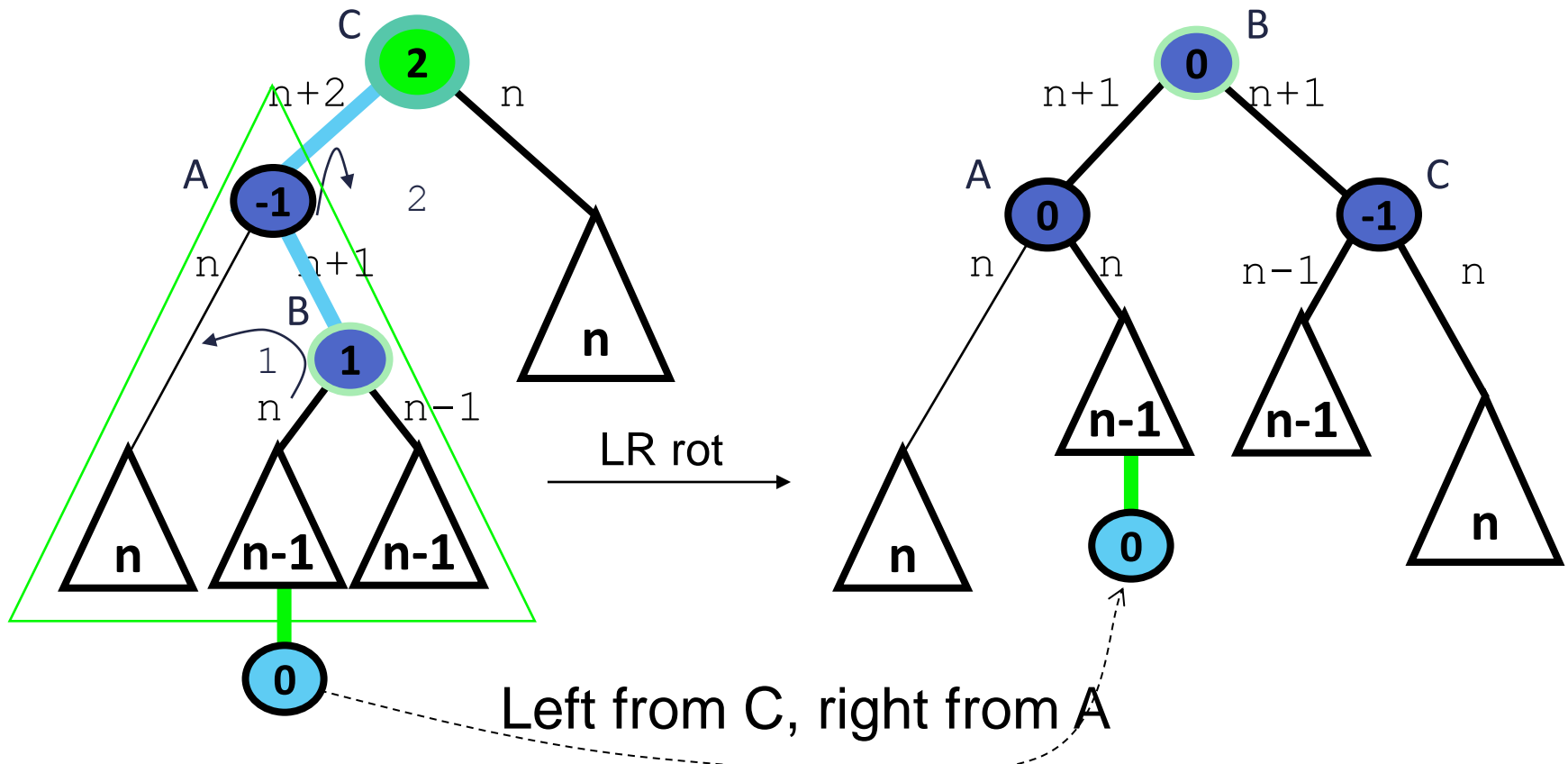
AVL strom - vložení uzlu doprava

b1) Podstrom se přidáním uzlu doprava rozváží
(The sub-tree loses its balance by node insertion – right)



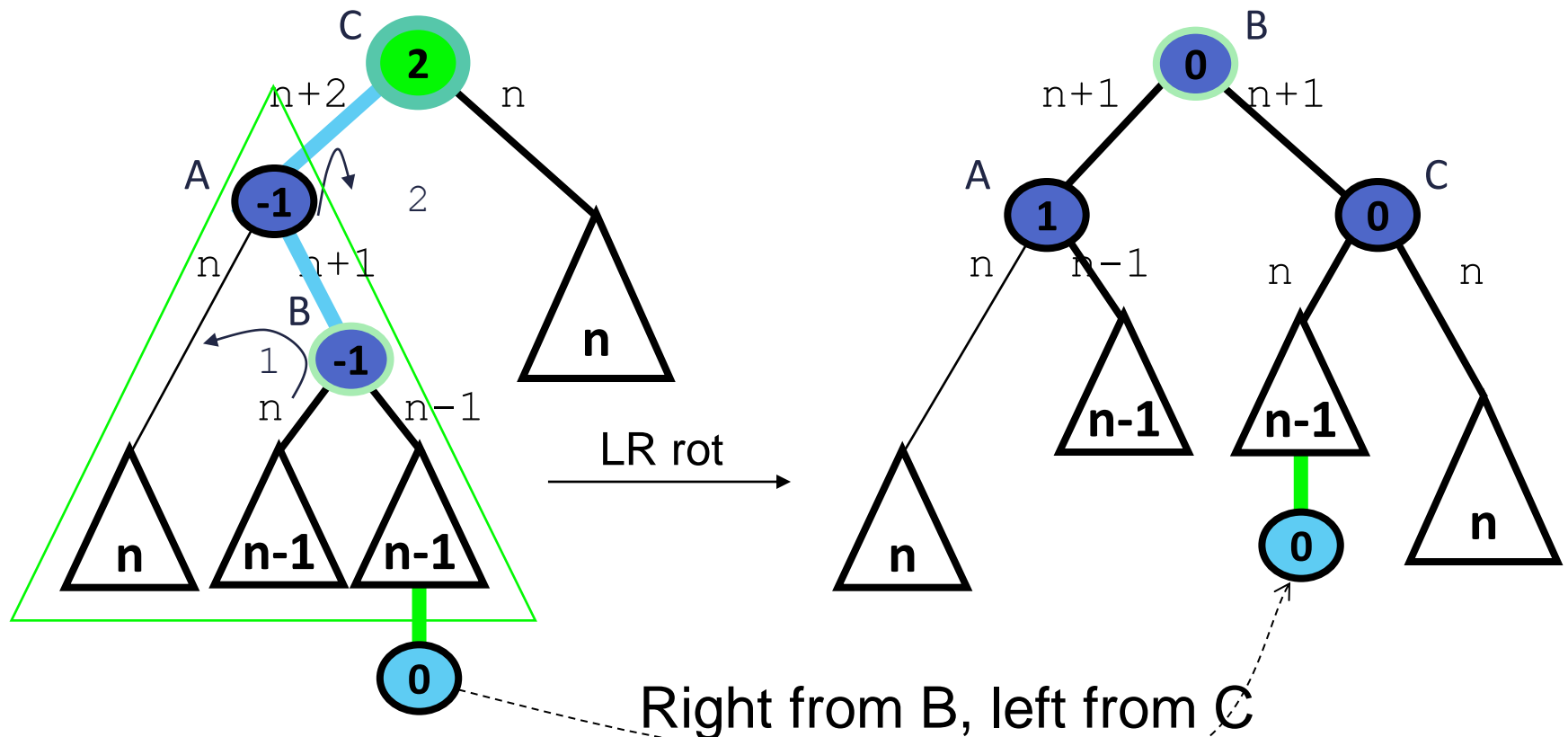
AVL strom - pravá rotace

b1) Vložen doleva – doprava => korekce LR rotací
 (Node inserted left – right => balance by the LR rotation)



AVL strom - pravá rotace

b2) Vložen doleva – doprava => korekce LR rotací
(Node inserted left – right => balance by the LR rotation)



Vkládání (Insert) do BVS bez vyvážení

```
void treeInsert( Tree t, Elem e )
{
    x = t.root;
    y = null;

    if( x == null ) t.root = e;        // single-leaf tree
    else {
        while( x != null ) {          // find the parent leaf y
            y = x;
            if( e.key < x.key ) x = x.left
            else x = x.right
        }
        // add e to parent y
        if( e.key < y.key ) y.left = e
        else y.right = e
    }
}
```

Java-like pseudo code

Vkládání (Insert) do AVL s vyvážením

```
void avlTreeInsert( tree t, elem e )  
{  
    // 1. init  
    // 2. find a place for insert  
    // 3. if( already present )  
    //         replace the node  
    //     else  
    //         insert new node  
    // 4.balance the tree, if necessary  
}
```

Java-like pseudo code

AVL Insert – proměnné & inicializace

```
avlTreeInsert( Tree t, Elem e )  
{  
    Node cur, fcur;    // current sub-tree and its father  
    Node a, b;        // smallest unbalanced tree and its son  
    Bool found;       // node with the same key as e found  
    Node help;        // for search and new node
```

1. init

```
    found = false;  
    cur = t.root; fcur = null;  
    a = cur, b = null;
```

2. find the place for insert

...

Java-like pseudo code

AVL Insert – najdi místo pro vložení

...

2. find the place for insert

```
while(( cur != null ) and !found )
{
    if( e.key == cur.key ) found = true;
    else {
        if( e.key < cur.key )
            help = cur.left;
        else help = cur.right;
        if(( help != null) and ( bal(help) != 0 )){
            //remember possible place for unbalance
            a = help;
        }
        fcur = cur; cur = help;
    }
} ...
```

AVL Insert – nahrazení nebo vložení

3. if(already present) replace the node value

```
if( found )
    setinfo( cur, e );    // replace the value
else {
    // insert new node to fcur
    help = leaf( e );    // cons ( e, null, null );
    if( fcur == null ) t.root = help;    // new root
    else {
        if( e.key < fcur.key )
            fcur.left = help;
        else
            fcur.right = help;
    }
    ...
}
```

AVL Insert – vyvážení podstromu

4.balance the tree, if necessary

```

if( bal(a) == 2 )      {           // inserted left from 1
    b = a.left;
    if( b.key < e.key )           //and right from its son
        a.left = leftRotation( b ); // L rotation (LR)
    a = rightRotation( a );       // R rotation
}
else if( bal(a) == -2){           //inserted right from -1
    b = a.right;
    if( e.key < b.key )           // and left from its son
        a.right = rightRotation( b );// R rotation(RL)
    a = leftRotation( a );        // L rotation
} // else tree remained balanced
} // !found
}

```

AVL Insert – vyvažování podstromu

4. Shrnutí

| a | b | Rotation |
|---|---|-------------|
| + | + | R rotation |
| + | - | LR rotation |
| - | + | RL rotation |
| - | - | L rotation |

AVL - výška stromu

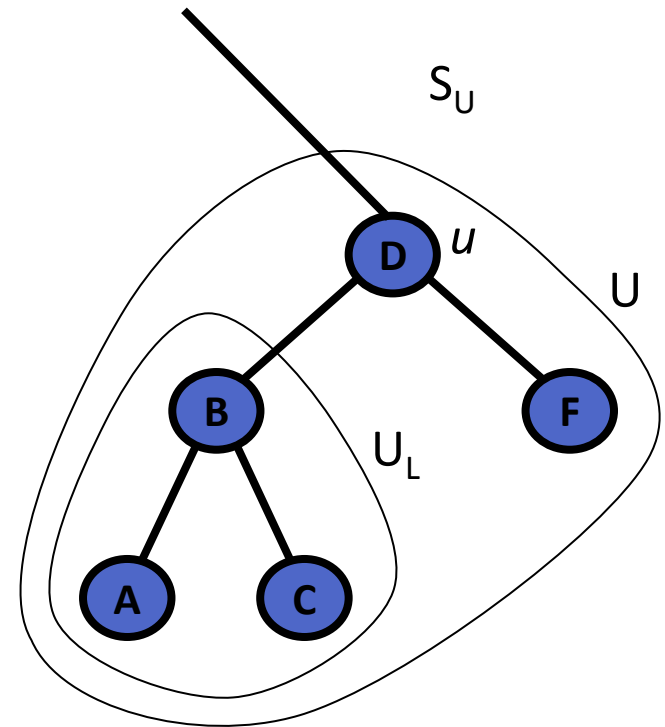
- Pro AVL strom S obsahující n uzlů platí:
- Výška (hloubka) stromu $h(S)$ je maximálně o 45% větší ve srovnání s ideálně vyváženým stromem.
- $\log_2(n+1) \leq h(S) \leq 1.4404 \log_2(n+2) - 0.328$ [Hudec96], [Honzík85]

Váhově vyvážené stromy

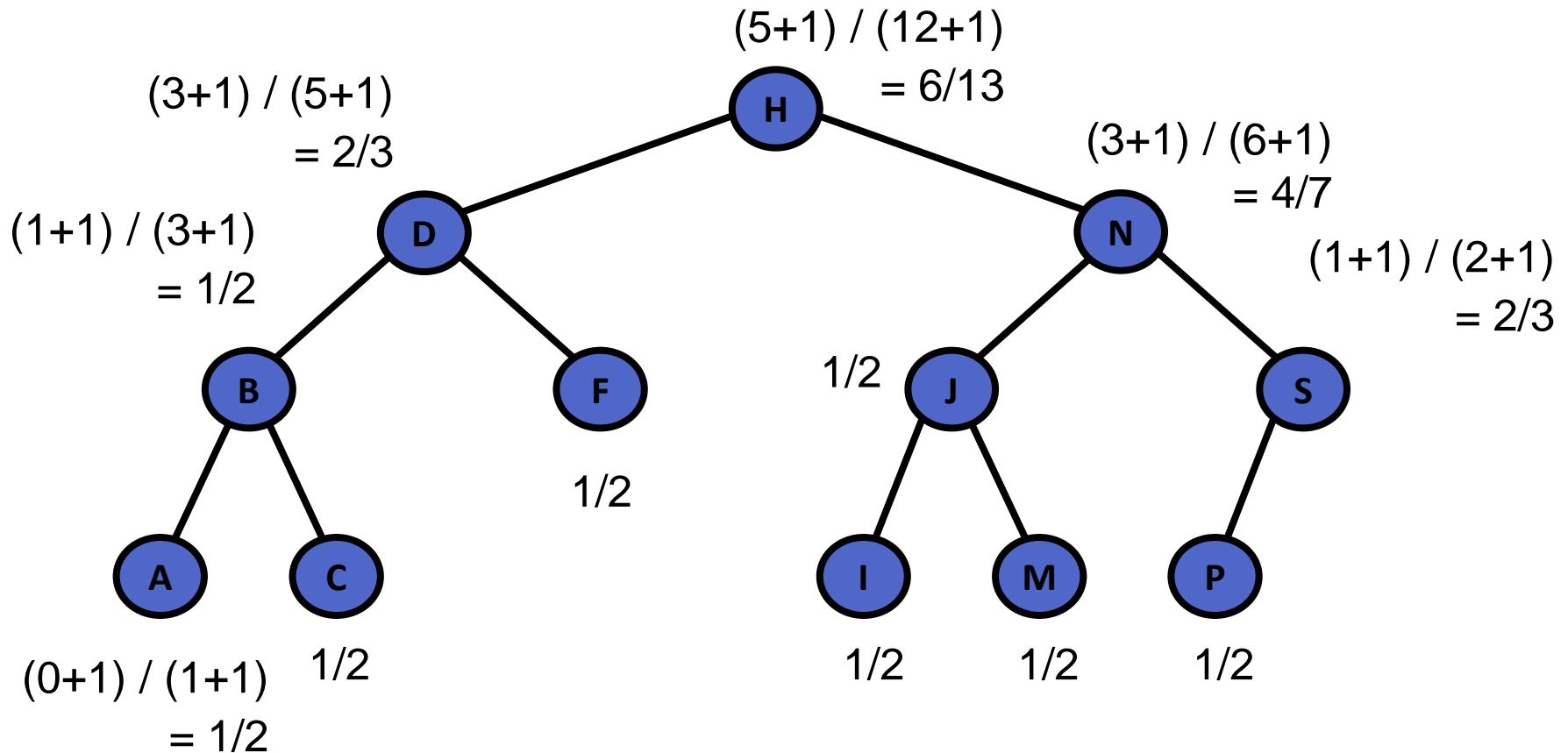
(stromy s ohraničeným vyvážením)

Váha $v(u)$ uzlu u ve stromě S :

- $v(u) = 1/2$, když je u listem
- $v(u) = (|U_L| + 1) / (|U| + 1)$,
když u je kořen podstromu $S_U \subseteq S$
 U_L = množina uzlů levého
podstromu v podstromu S_U
 U = množina uzlů podstromu S_U



Příklad váhově vyváženého stromu



Strom s ohraničeným vyvážením α


Strom S má ohraničené vyvážení α , $0 \leq \alpha \leq 0,5$, jestliže pro všechny uzly S platí:

$$\alpha \leq v(u) \leq 1 - \alpha$$

Výška (hloubka) $h(S)$ stromu S s ohraničeným vyvážením α :

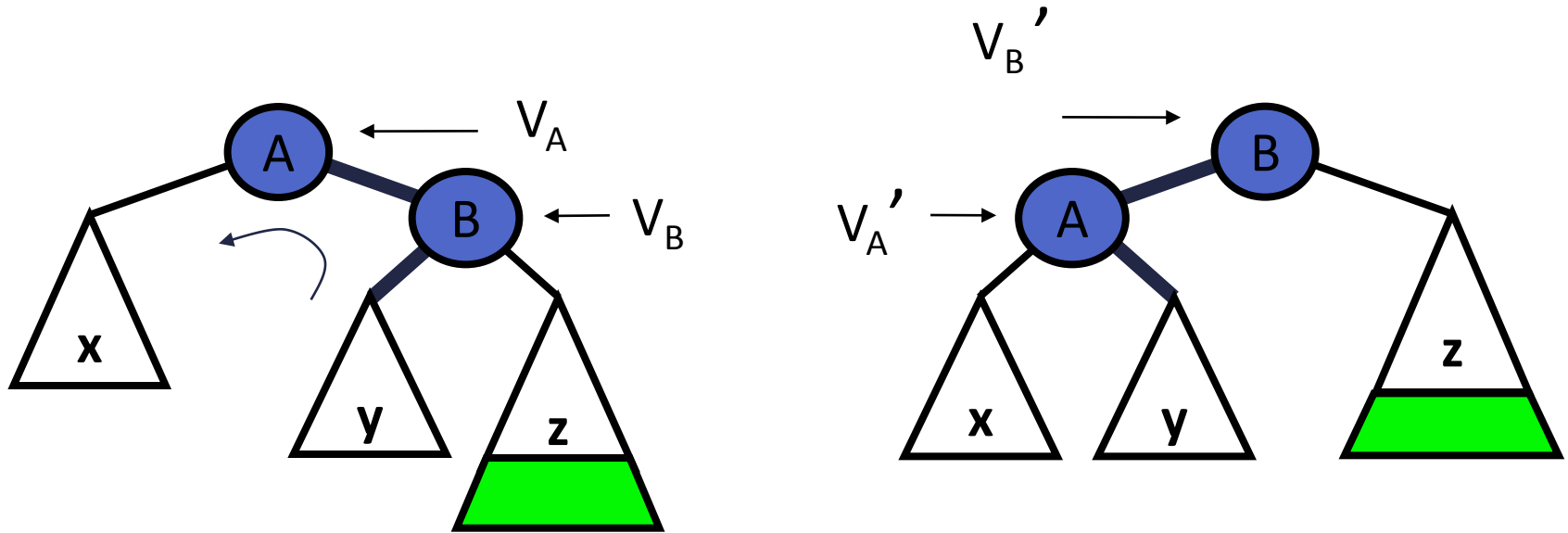
$$h(S) \leq (1 + \log_2(n+1) - 1) / \log_2(1 / (1 - \alpha))$$

[Hudec96], [Mehlhorn84]



Výška ideálně
vyváženého stromu

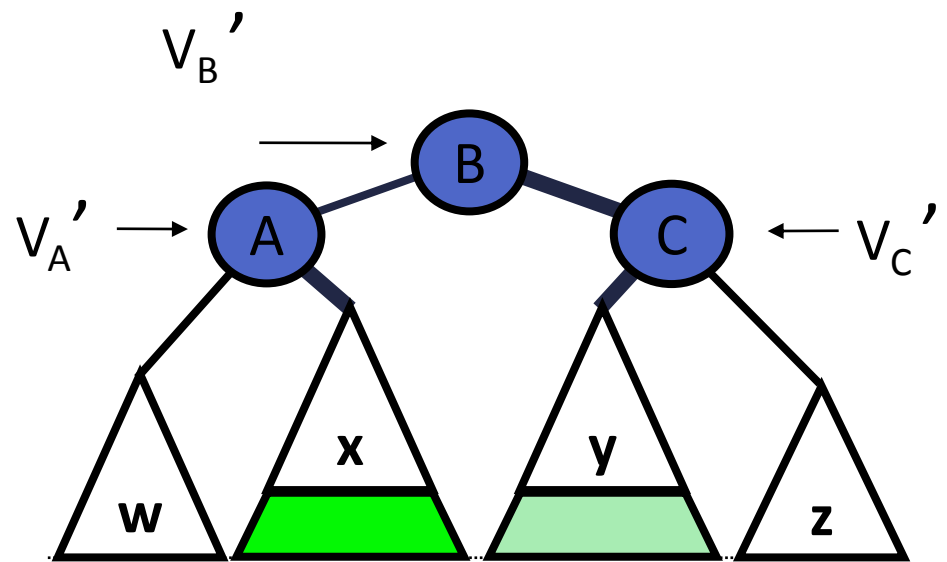
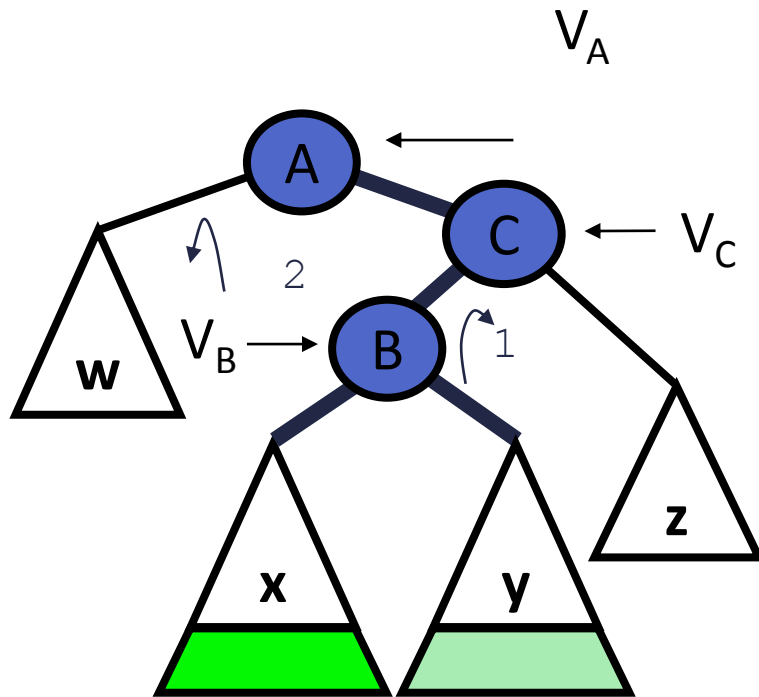
L rotation (Left rotation)



$$V_A' = V_A / (V_A + (1 - V_A) \cdot V_B)$$

$$V_B' = V_A + (1 - V_A) \cdot V_B$$

RL rotation (Right-Left rotation)



$$V_A' = V_A / (V_A + (1 - V_A) V_B V_C)$$

$$V_B' = V_B (1 - V_C) / (1 - V_B V_C)$$

$$V_C' = V_A + (1 - V_A) \cdot V_A V_B$$

[Hudec96]

The End