Data structures and algorithms

Part 8

Searching and Search Trees

With some Czech slides just for terminology

Petr Felkel

26.11. 2007

Searching – talk overview

Typical operations

Quality measures

Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

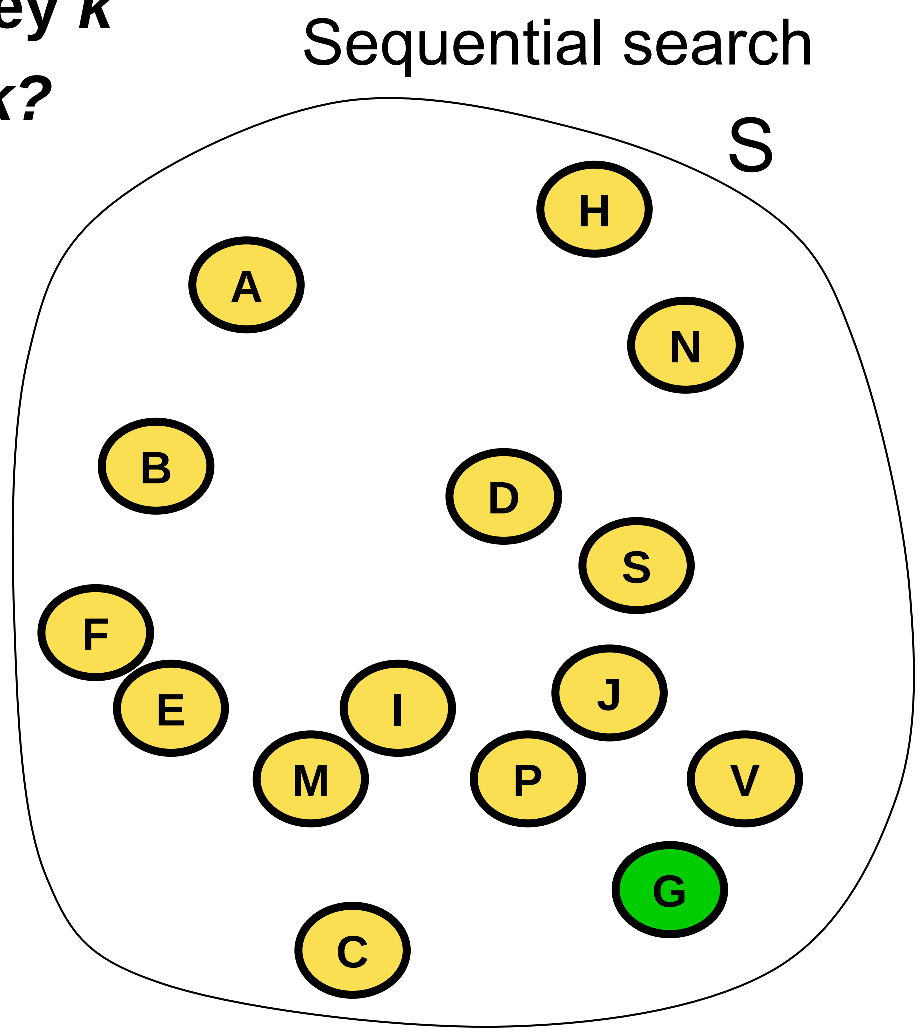
Searching

Input: a set of *n* keys, a query key *k*

Problem description: *Where is k?*

G?

Search was successful



Searching

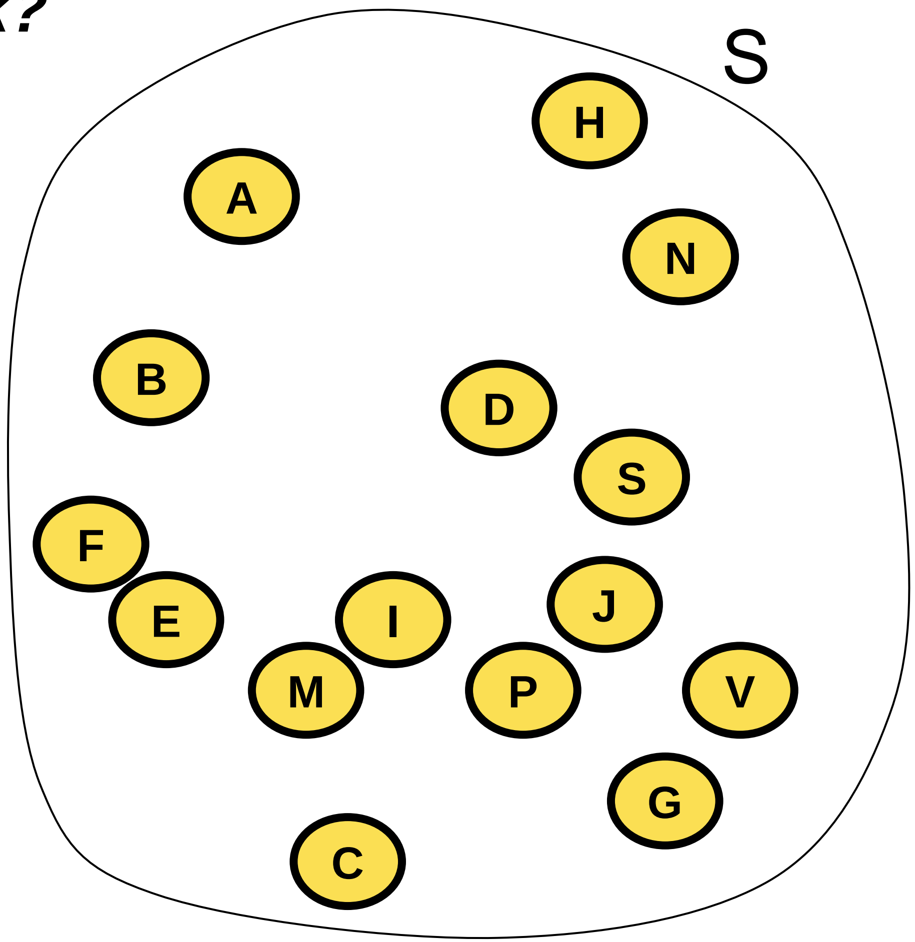
Input: a set of *n* keys, a query key *k*

Problem description: *Where is k?*

L?

Search was unsuccessful

Sequential search

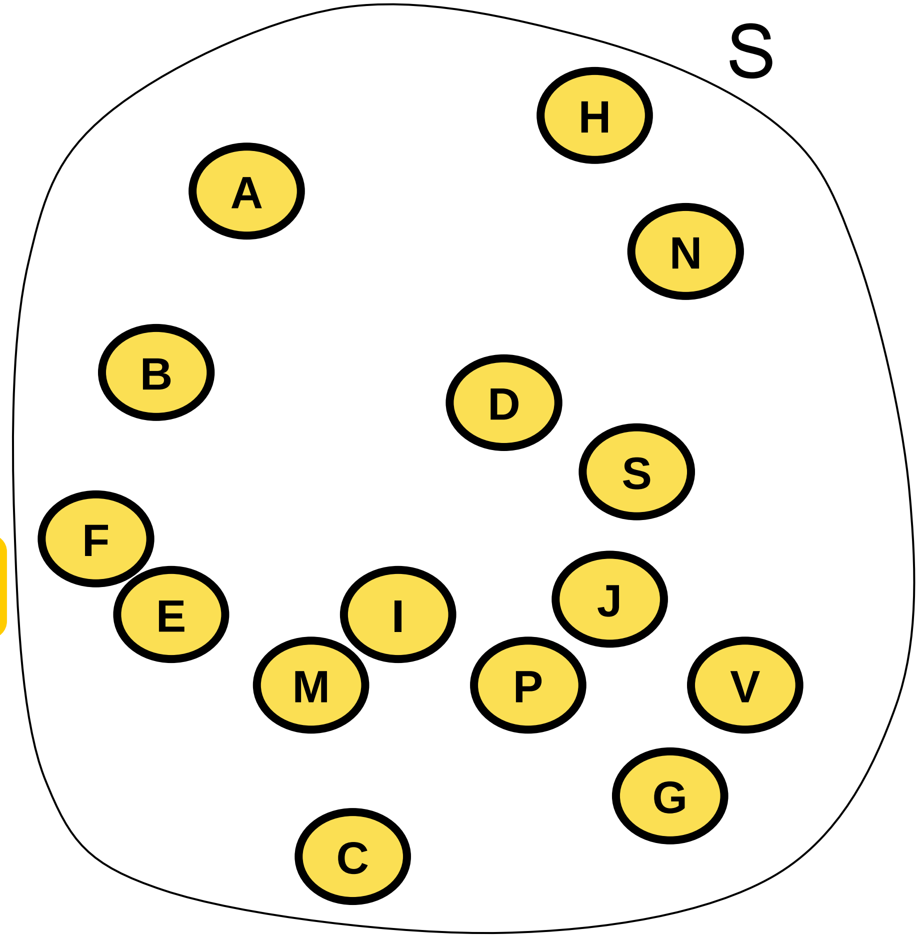


Searching

Search space S

= set of keys where we search

- precisely: set of records with keys we search
- unique keys
- (table, file,…)



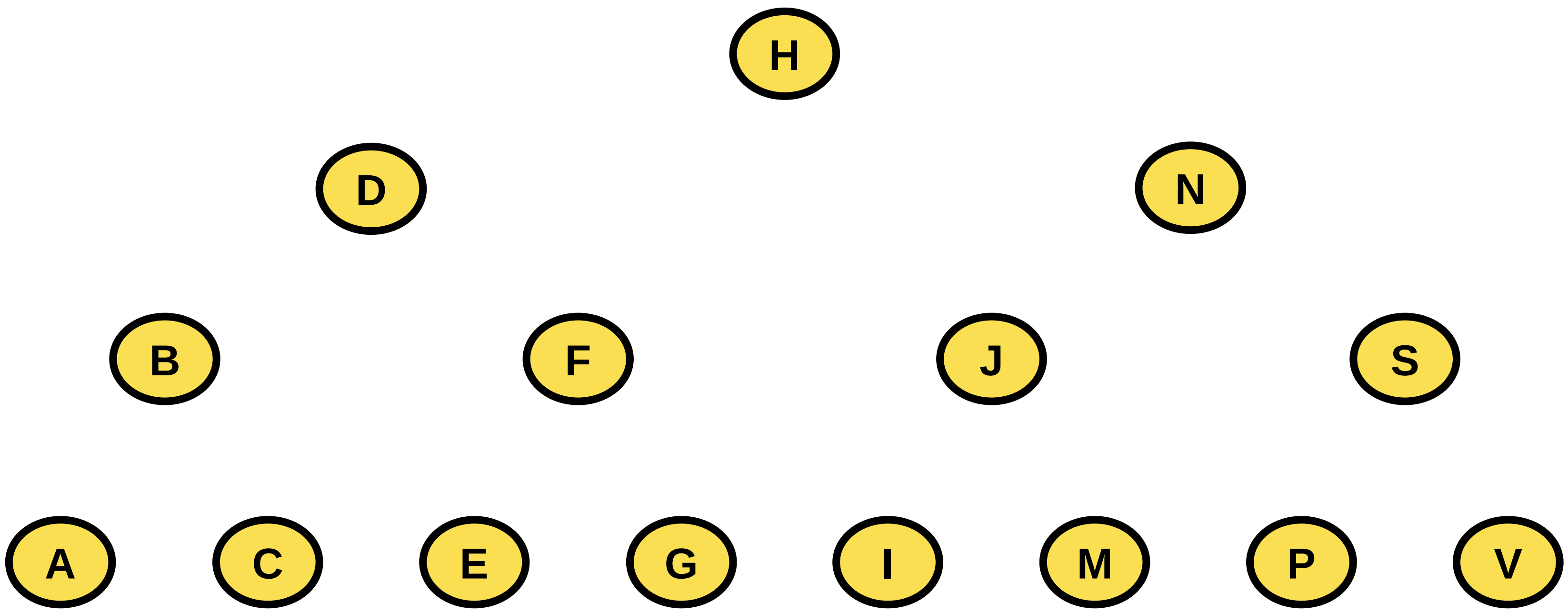
Universum U of the search space

= set of ALL possible keys

$$S \subset U$$

Searching

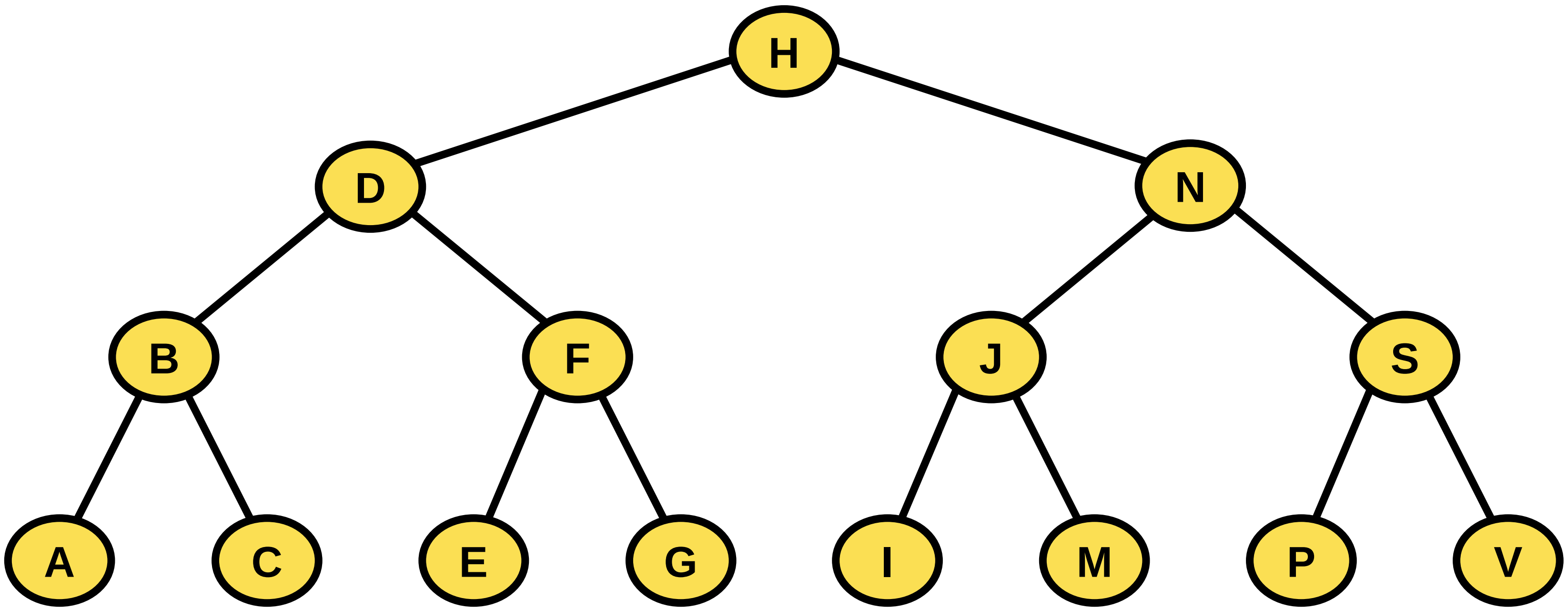
Speed-up



Searching

Input: a set of *n* keys, a query key *k*

Problem description: *Where is k?*

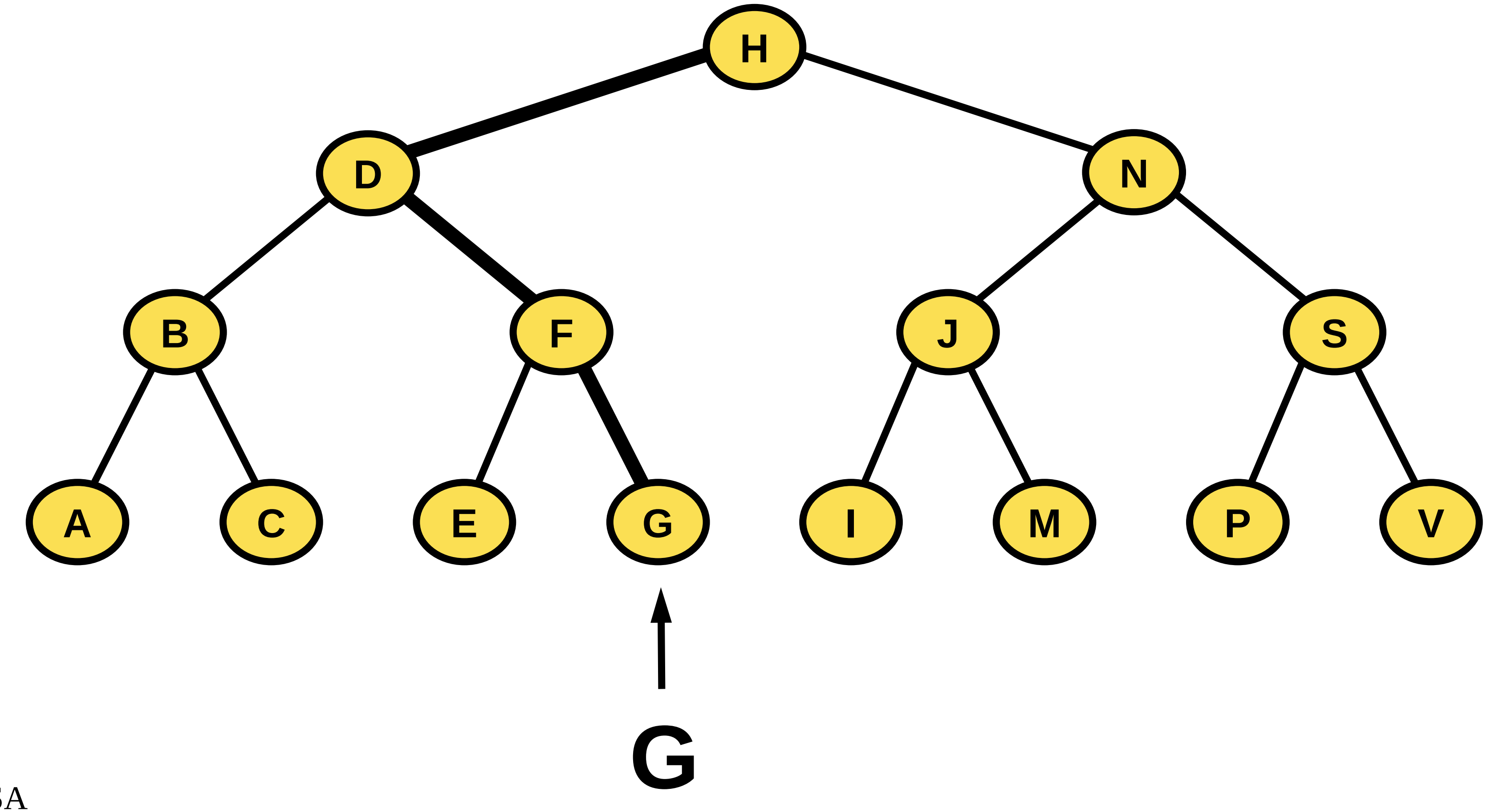


G?

Searching

Input: a set of *n* keys, a query key *k*

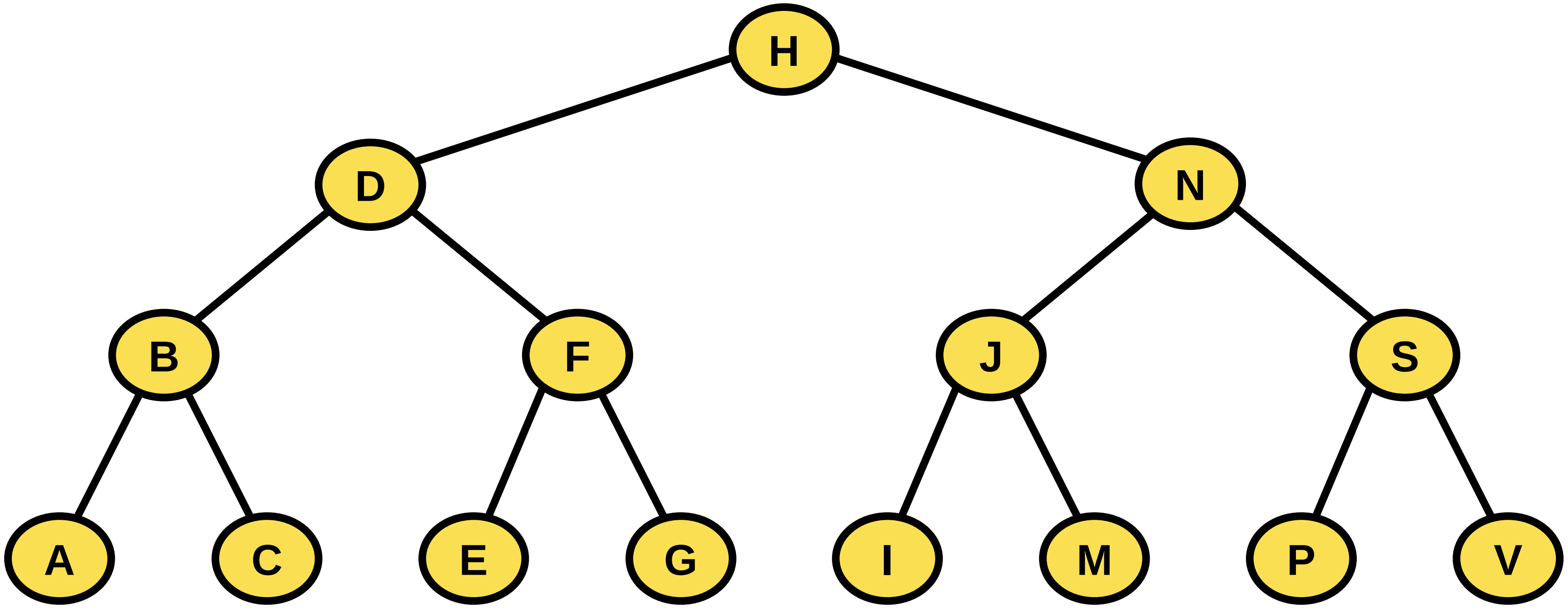
Problem description: *Where is k?*



Searching

Input: a set of *n* keys, a query key *k*

Problem description: *Where is k?*

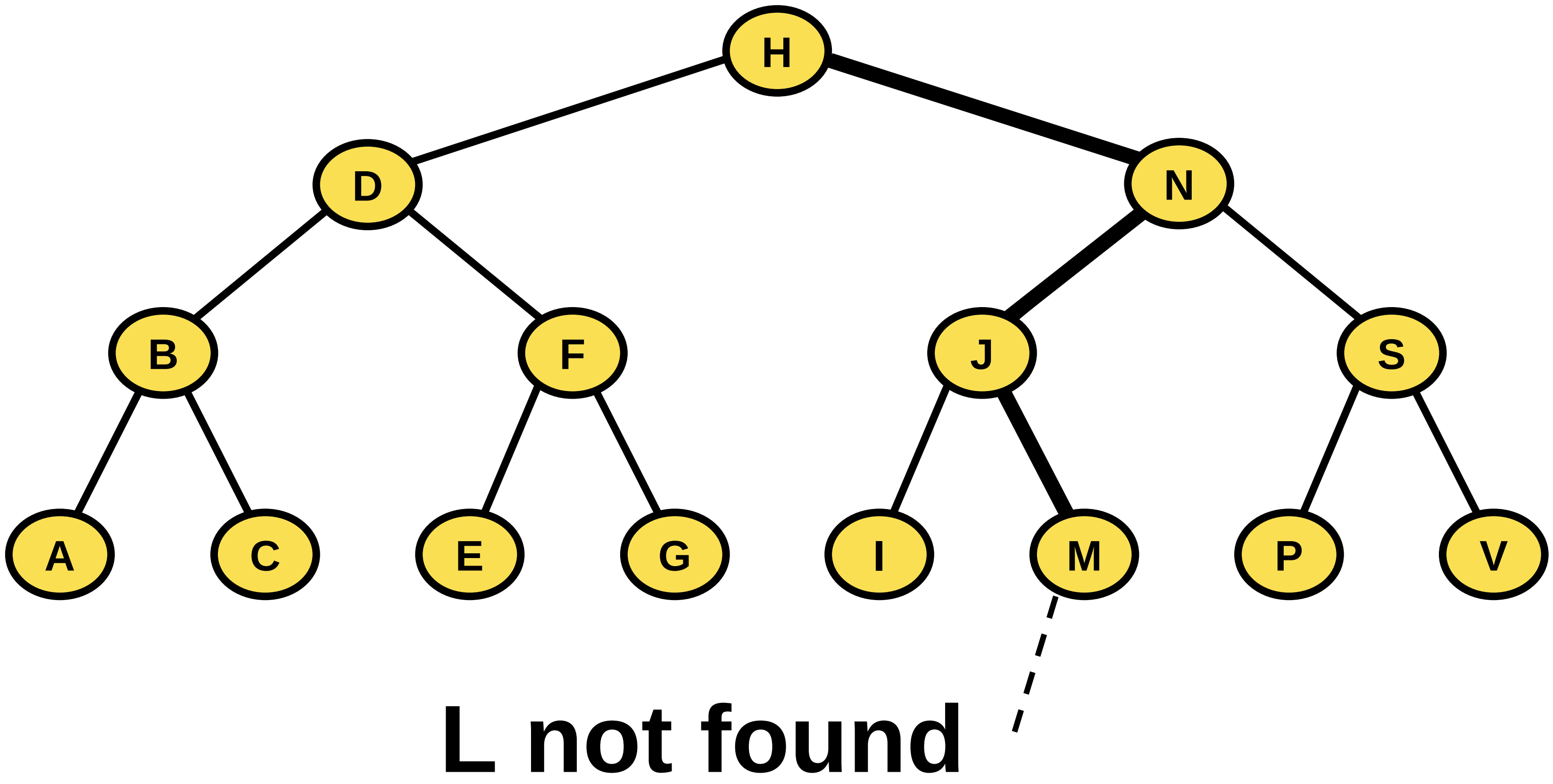


L?

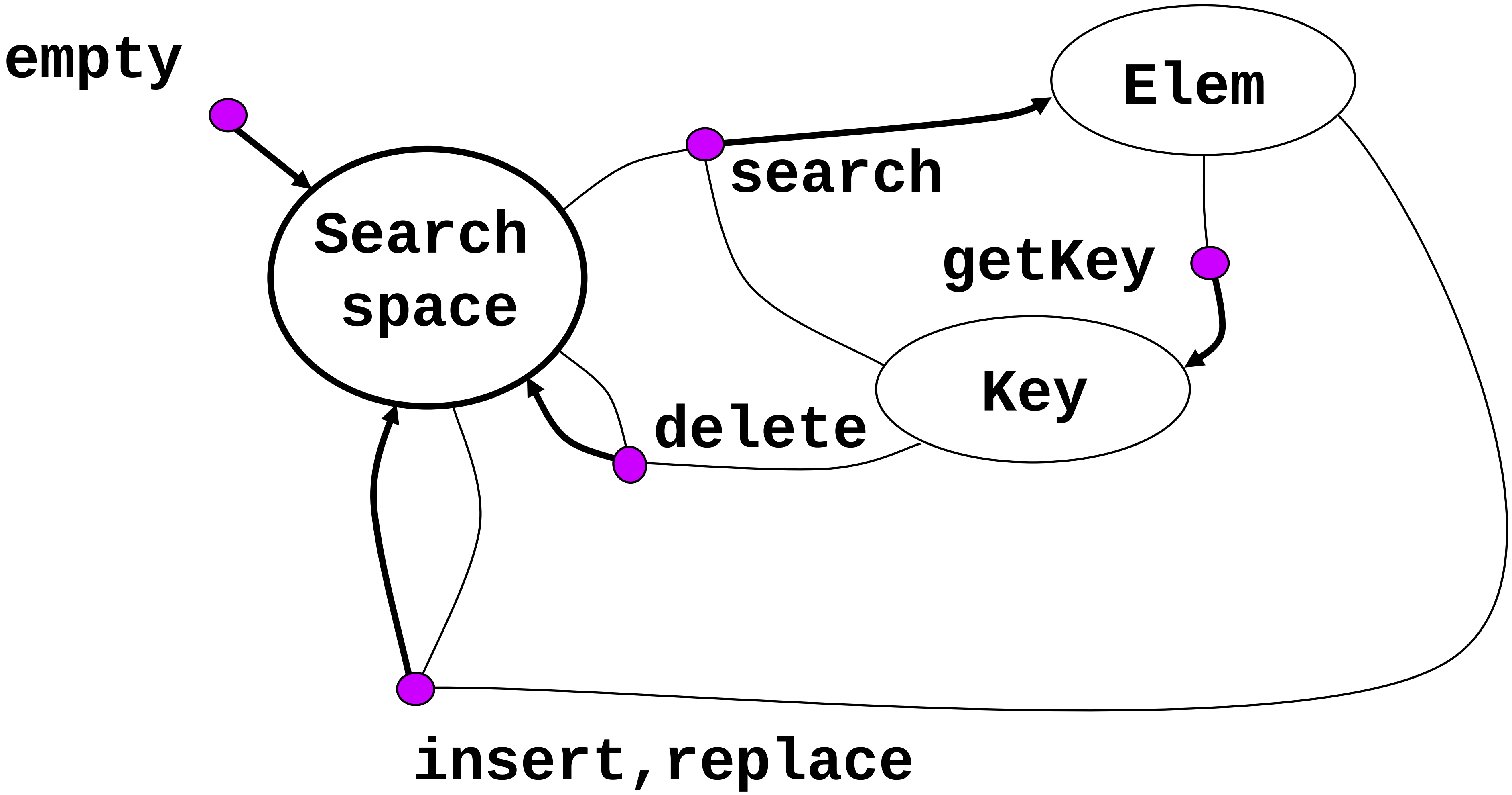
Searching

Input: a set of *n* keys, a query key *k*

Problem description: *Where is k?*



Search space



Searching

Search space *(lexicon)*

– Static - fixed search space

-> simpler implementation

-> change => new release

-> example: Phonebook, printed dictionary

– Dynamic - search space changes in time

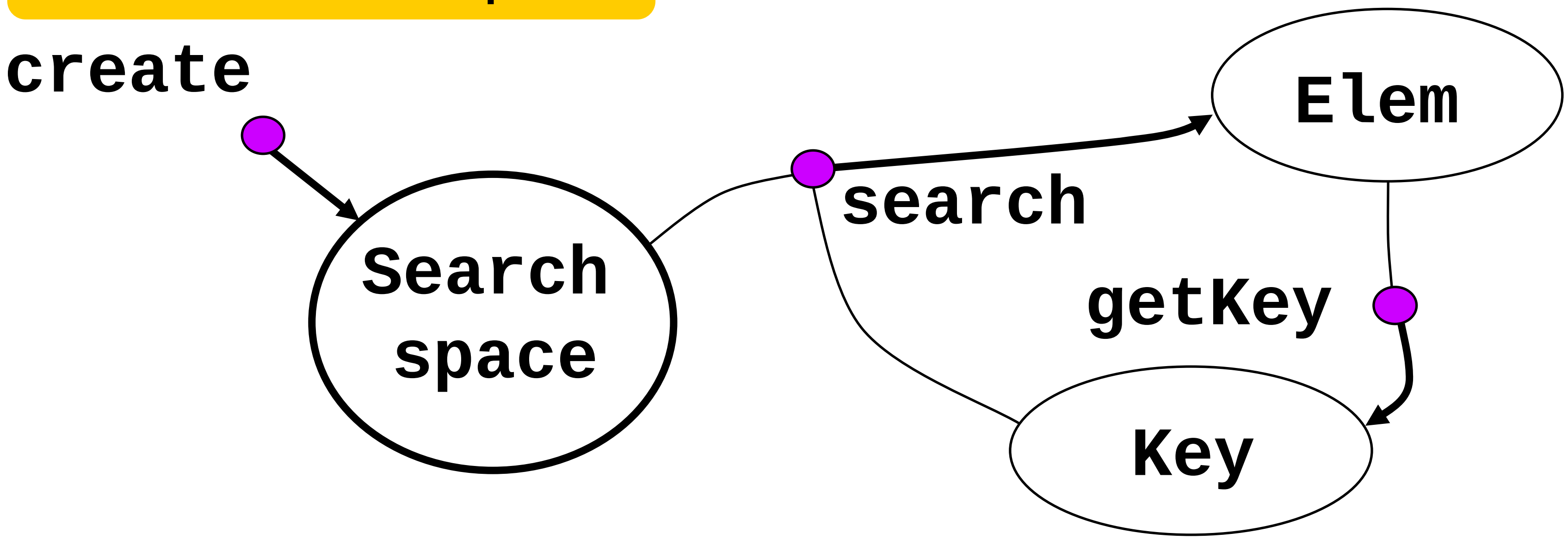
-> more complex implementation

-> change by `insert`, `delete`, `replace`

-> table of symbols in compiler, dictionary,…

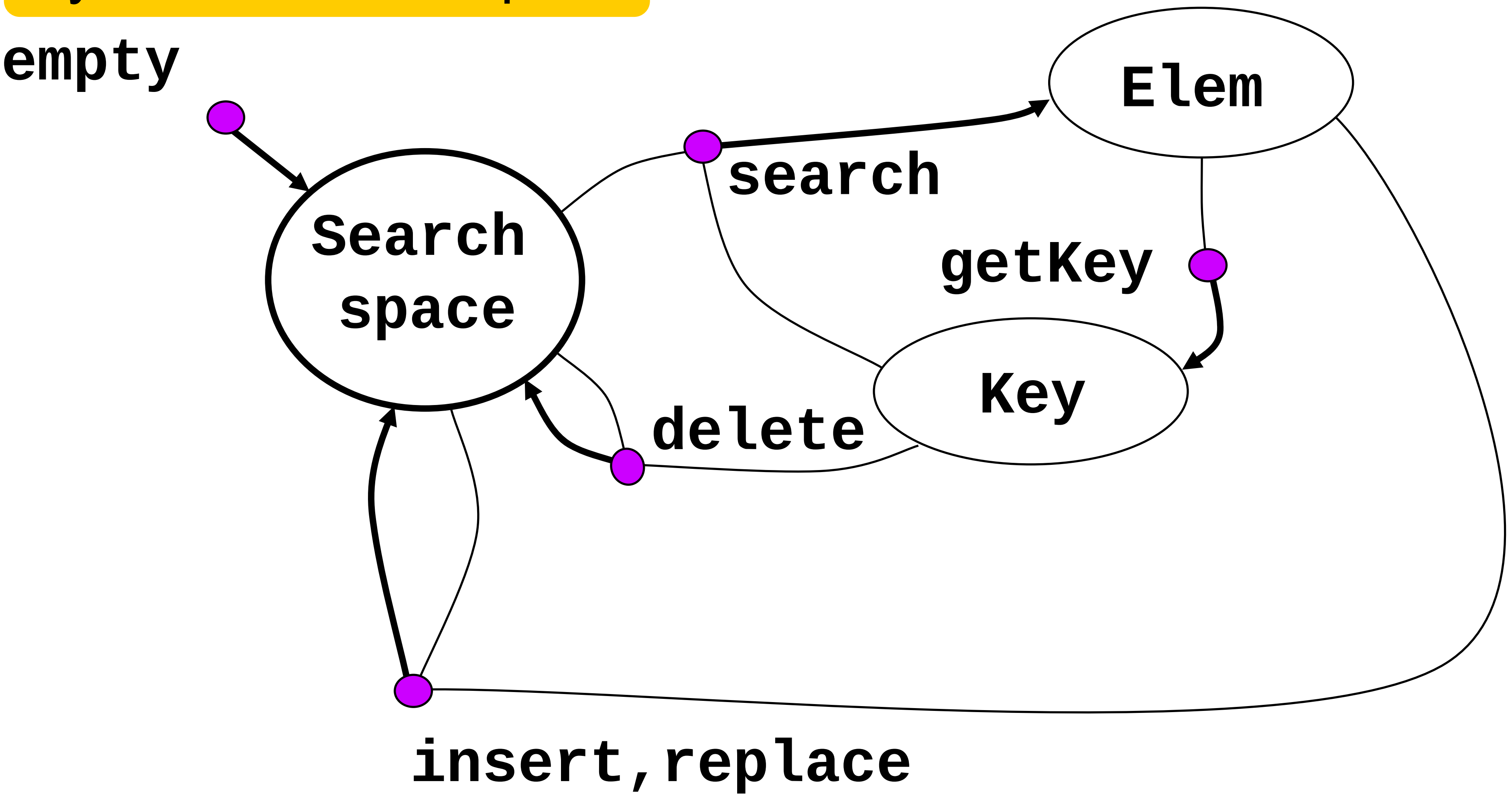
Search space

Static search space



Search space

Dynamic search space



Searching

Variables: `k … key`

`e … element with key k`

`s … data set`

Operations (Informal list):

selectors

- **`search(k,s)`**
- `min(s), max(s)`
- `pred(e,s), succ(e,s)`

extension

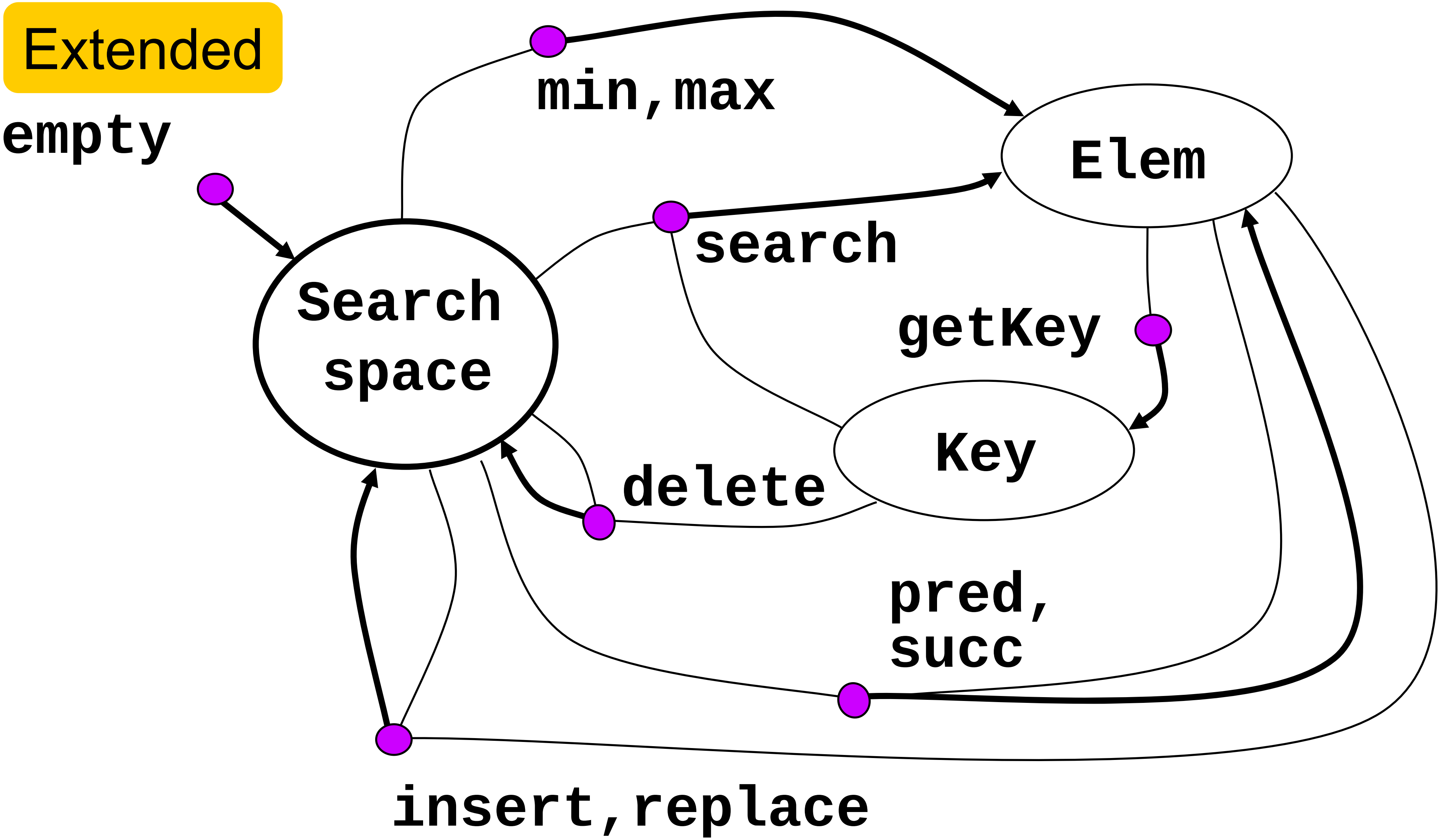
modifiers

Key of element to replace is part of the new element e

- `insert(e,s), delete(k,s), replace(e,s)`

Search space

Extended



Another classification

Address search - based on digital properties of keys

– Compute position from key pos = f(k)

– Direct access (*přímý přístup*), hashing

– Array, table,…

– Direct => FAST (see lecture 11) … O(1)

Associative search - based on comparison between el.

– Element is located in relation to others

– Sequential, binary search, search trees

– Needs searching => SLOWER … O(log n) to O(n)

Another classification

Internal or external

- **internal in the memory**
- external in files on disk or tape

Dimensionality of keys

- **One dimensional - k**
- Multidimensional - [x,y,z]

Searching – talk overview

Typical operations

Quality measures

Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*)

- Node representation
- Operations
- Tree balancing

Quality measures

Space for data

P(n) = memory complexity

Time / Number of operations

Q(n) = complexity of **`search`**, **`query`**

I(n) = complexity of **`insert`**

D(n) = complexity of **`delete`**

Searching – talk overview

Typical operations

Quality measures

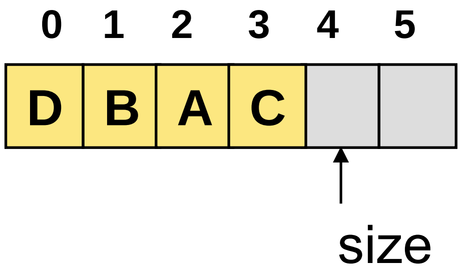
Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

Searching in unsorted array

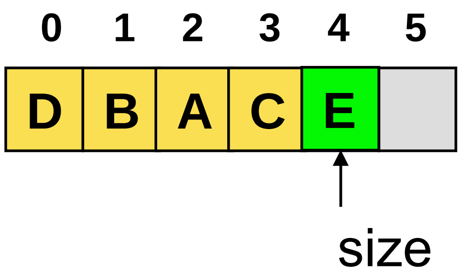


Unsorted array	$P(n) = O(n)$
Sequential search	$Q(n) = O(n)$
insert	$I(n) = O(1)$
delete	$D(n) = O(n)$
min, max	$Q_m(n) = O(n)$

```
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

Searching in unsorted array



Unsorted array with sentinel (zarážka)

Sequential search still Q(n) = O(n)

But saves one test per step

```
search("E", a)
```

```
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

Searching – talk overview

Typical operations

Quality measures

Implementation in an array

– Sequential search

– Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

– Node representation

– Operations

– Tree balancing

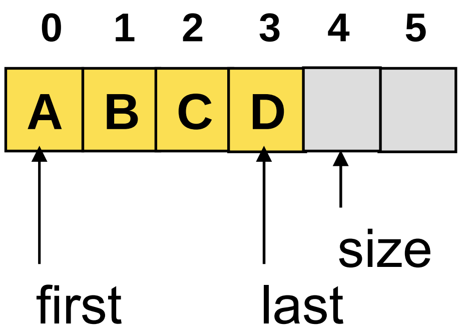
Searching in sorted array

Binary search


```
search("A", a)
```

Java-like pseudo code

Searching in sorted array



Sorted array	$P(n) = O(n)$
Binary search	$Q(n) = O(\log(n))$
insert	$I(n) = O(n)$
delete	$D(n) = O(n)$
min, max	$Q_m(n) = O(1)$

```
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

Binary search <,=,>

```
//Recursive version                 Stop if found ->   O(log(n))
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                Stop if found ->   O(log(n))
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) last = mid - 1;
            else if (key > a[mid].key) first = mid + 1;
                 else return mid; // found
      } return -(first + 1); // failed to find key
}
```

Java-like pseudo code

Binary search <=, >

```
// Iterative fix length version              ->   Θ(log(n))
// with just one test, stop after log(n) steps
int bs(key k, nodeT a[], int first, int last) {
      while (first < last) {
            int mid = (first + last) / 2;
            if (key > a[mid].key) first = mid + 1;
            else //can't be last = mid-1: here A[mid] >= key
                 //so last can't be < mid if A[mid] == key
                 high = mid;

      } return -(first + 1); // failed to find key

      if (first < N) and (A[first ] == value)
         return first
      else return not_found
```

Java-like pseudo code

Where is the BUG?

Binary search bug

Binary search bug

[pointed out by Ondřej Karlík/Joshua Bloch]
[Sun JDK 1.5.0 beta, 2004]

```
int mid = (first + last) / 2;
int mid = (first + last) >> 1;
```

gibibyte

Signed arithmetic overflow for large arrays

– number larger than 2^{30} !!! ~ 1 GiB
– negative index out of bouds

Solution:

```
int mid = first + ((last - first) / 2);
int mid = (first + last) >>> 1; // unsigned shift
int mid = ((unsigned) (first + last)) >> 1;
```

Interpolation search

Interpolation search

- parallels how humans search through a phone book
- estimates position based on values of bounds `a[first]` and `a[last]`

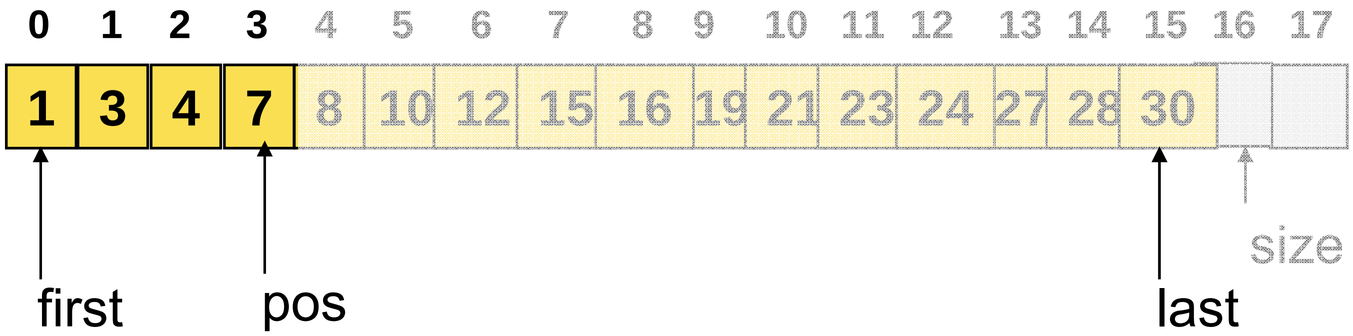
```
                (last - first)
pos = first + ----------------  (x – a[first])
                a[last] – a[first]
```

- O(log log n) average case for uniform distribution
- O(n) maximum for e.g. exponential distribution

Searching in sorted array

Interpolation search

`search("7", a)`




```
                (last - first)
pos = first +  -----------------   (x – a[first])
                a[last] – a[first]

            (15 - 0)
pos = 0 +  ---------  *(7 – 1) = 15/29 * 6 = 3 => found
            30 – 1
```

while `mid = 15-0 = 7`

Searching (*Vyhledávání*)

Typical operations

Quality measures

Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

Binární vyhledávací strom (BVS)

Binární strom (=kořenový, orientovaný, dva následníci) +

= prázdný strom, nebo

trojice: kořen a TL (levý podstrom) a TR (pravý podstrom).

Jeden i oba mohou být prázdné [Kolář]

– uzel má 0, 1, 2 následníky (nemusí být pravidelný)

Binární vyhledávací strom (BVS)

– binární strom, v němž navíc

– Pro libovolný uzel u platí, že

pro všechny uzly u_L z levého podstromu a

pro všechny uzly u_R z pravého podstromu uzlu u platí:

$$\text{klíč}(u_L) < \text{klíč}(u) < \text{klíč}(u_R)$$

Binary search tree (BST)

Binary tree (=rooted, i.e., oriented, two successors,…) +

= empty tree, or

triple: root, TL (left subtree), and TR (right subtree). One or both can be empty [Kolář]

– node has 0, 1, 2 successors (need not to be regular)

Binární vyhledávací strom (BVS)

= Binary tree, and moreover

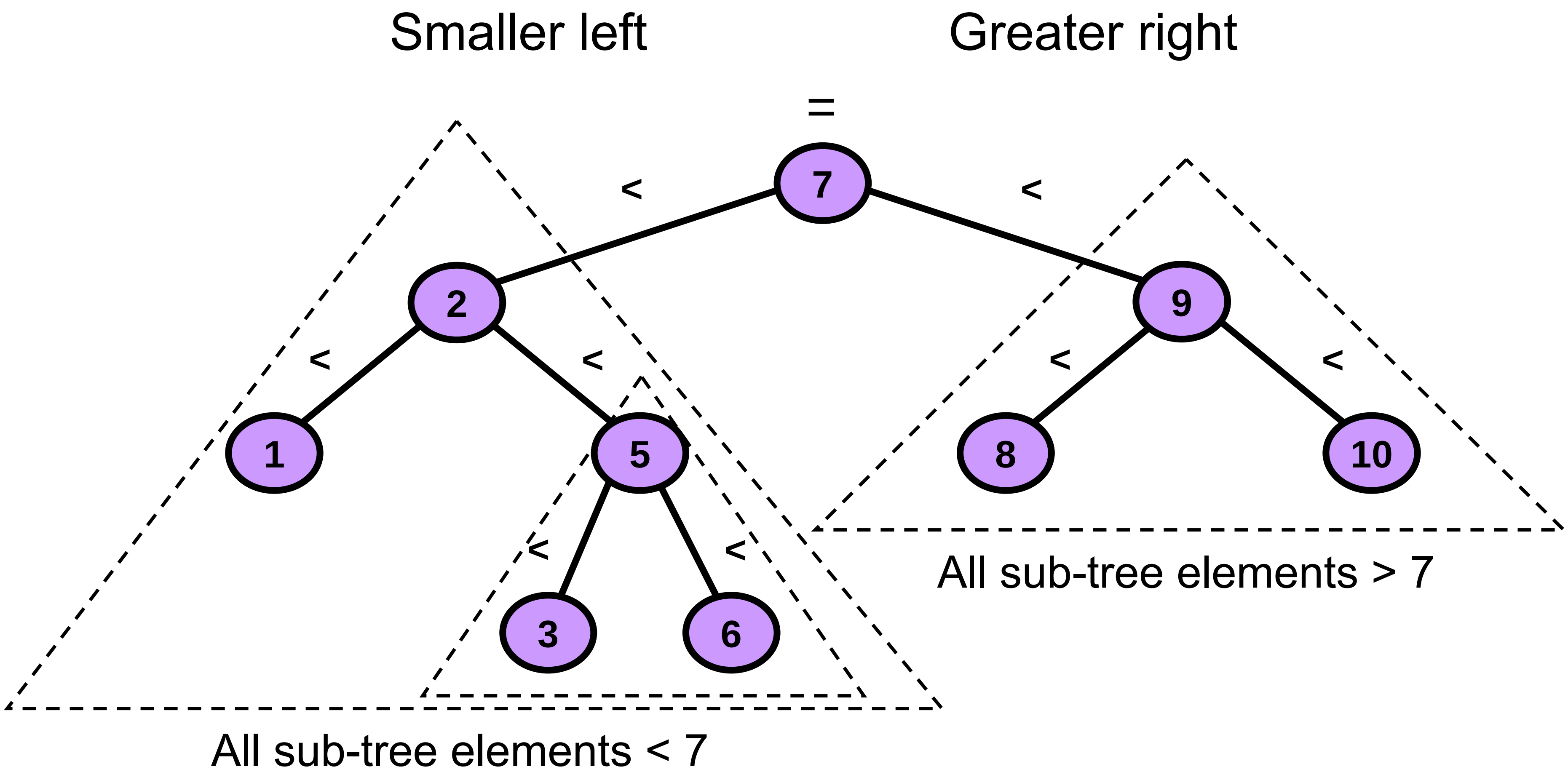
– For any node *u holds*

for all nodes u_L from the left subtree and

for all nodes u_R from the right subtree of node u holds:

$$\text{key}(u_L) < \text{key}(u) < \text{key}(u_R)$$

Binární vyhledávací strom
Binary Search Tree



Searching (*Vyhledávání*)

Typical operations

Quality measures

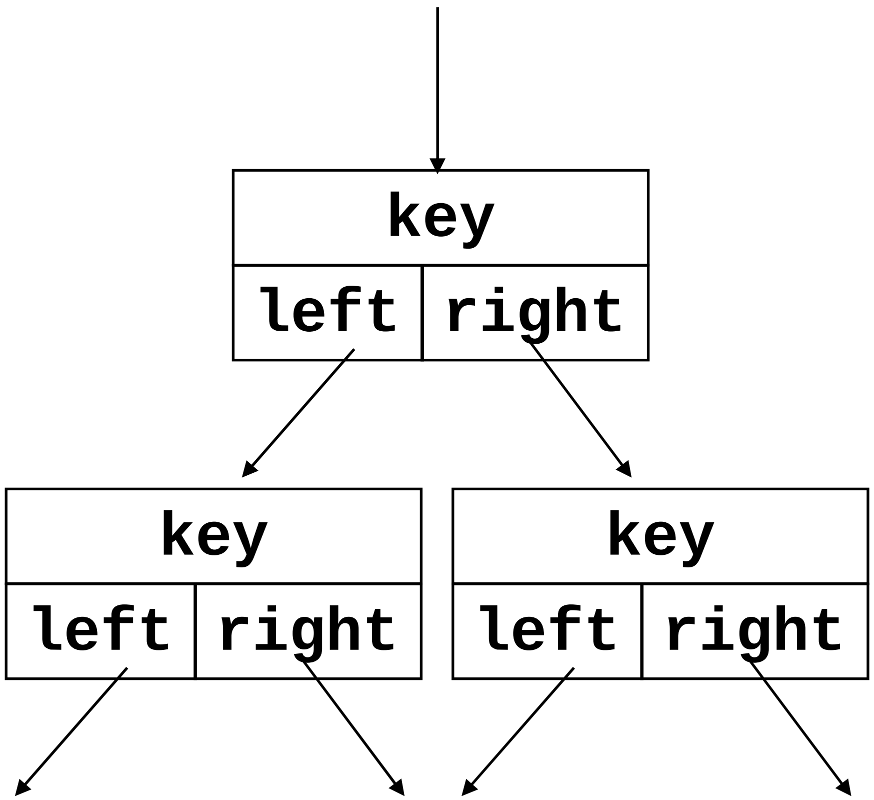
Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

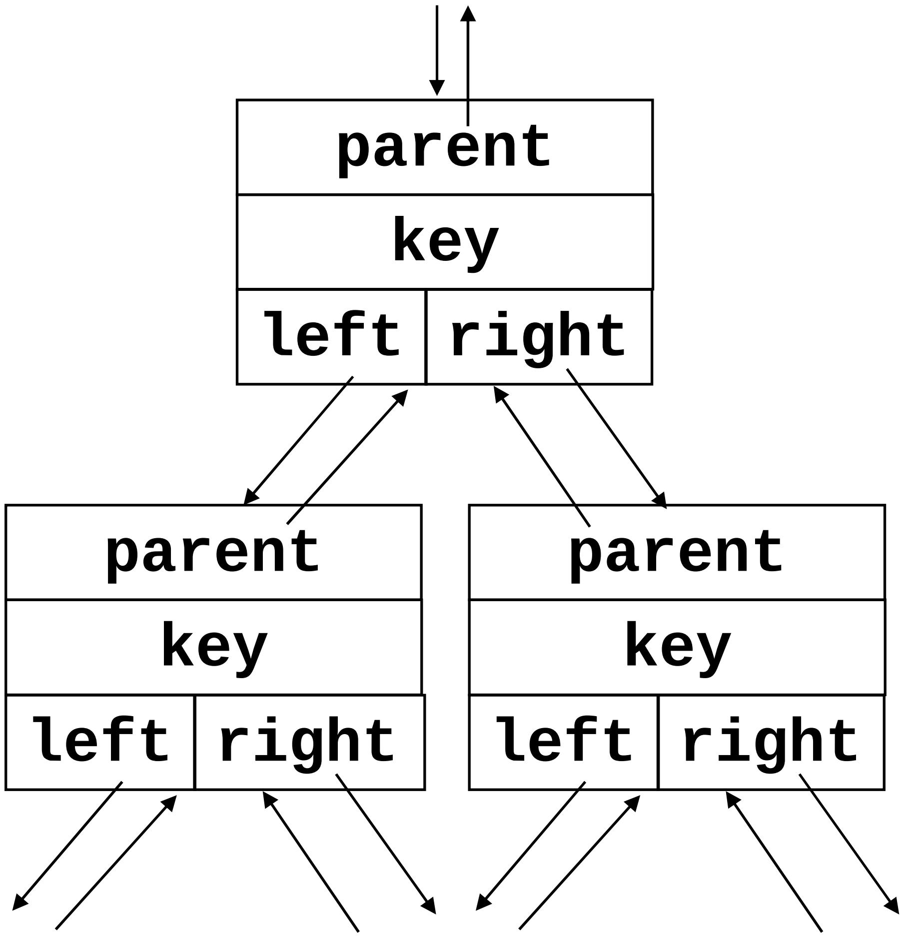
Tree node representation



Good for:

- search
- min, max

Tree node representation



Good for

- search
- min, max
- predecessor, successor

Tree node representation

```
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;
   }}
```

See Lesson 6, page 17-18

```
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 {
   ...
}
```

Searching (*Vyhledávání*)

Typical operations

Quality measures

Implementation in an array

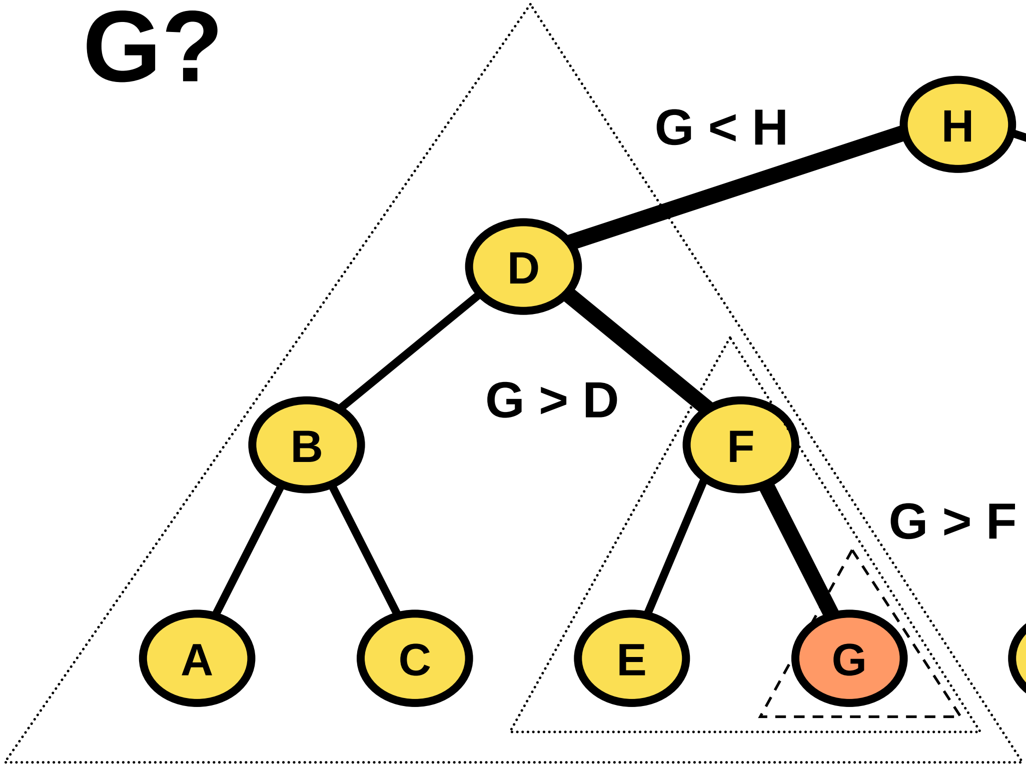
- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

Searching BST

G?



G = G => stop, element G found

Searching BST - recursively

```
//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 );
}
```

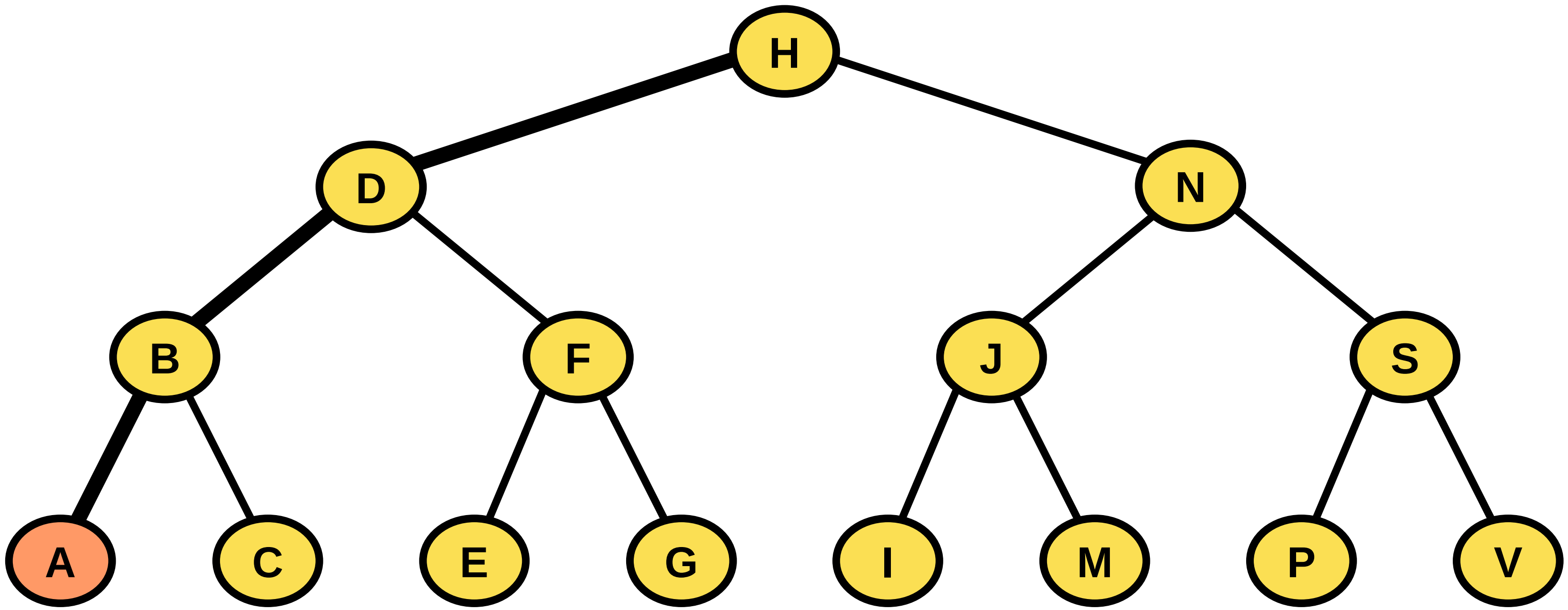
Java-like pseudo code

Searching BST - iteratively

```
//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 in BST



Minimum in BST - iteratively

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

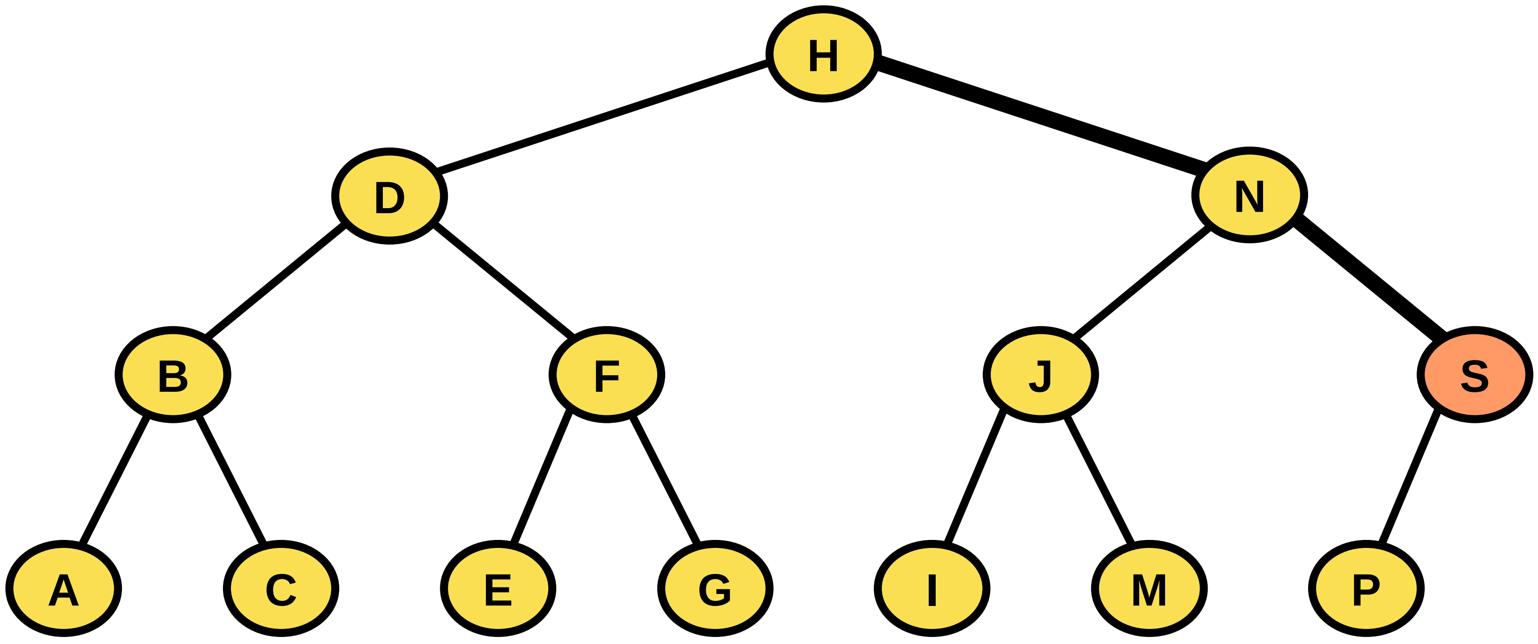
Java-like pseudo code

Maximum in BST - iteratively

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

Java-like pseudo code

Maximum in BST



Successor in BST

in the sorted order (in-order tree walk)

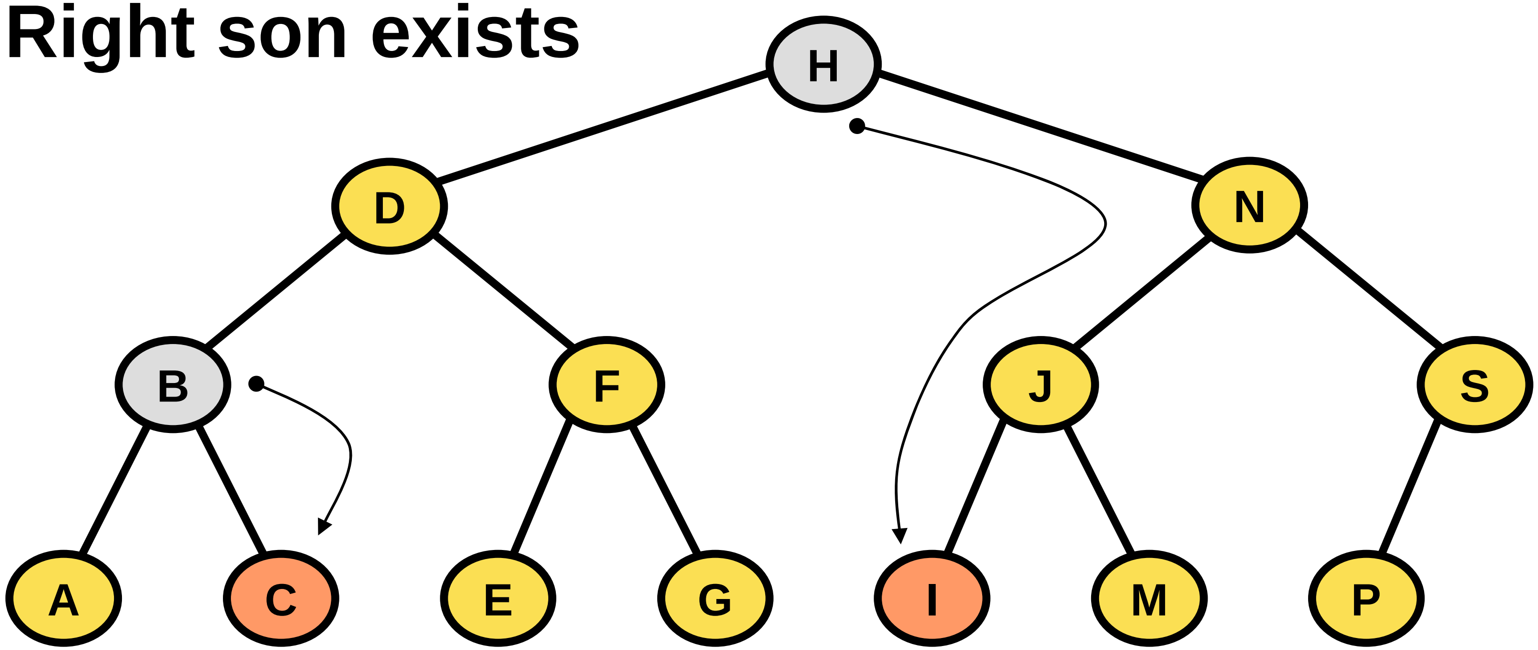
Two cases:

1. Right son exists
2. Right son is null

Successor in BST

in the sorted order (in-order tree walk)

1. Right son exists



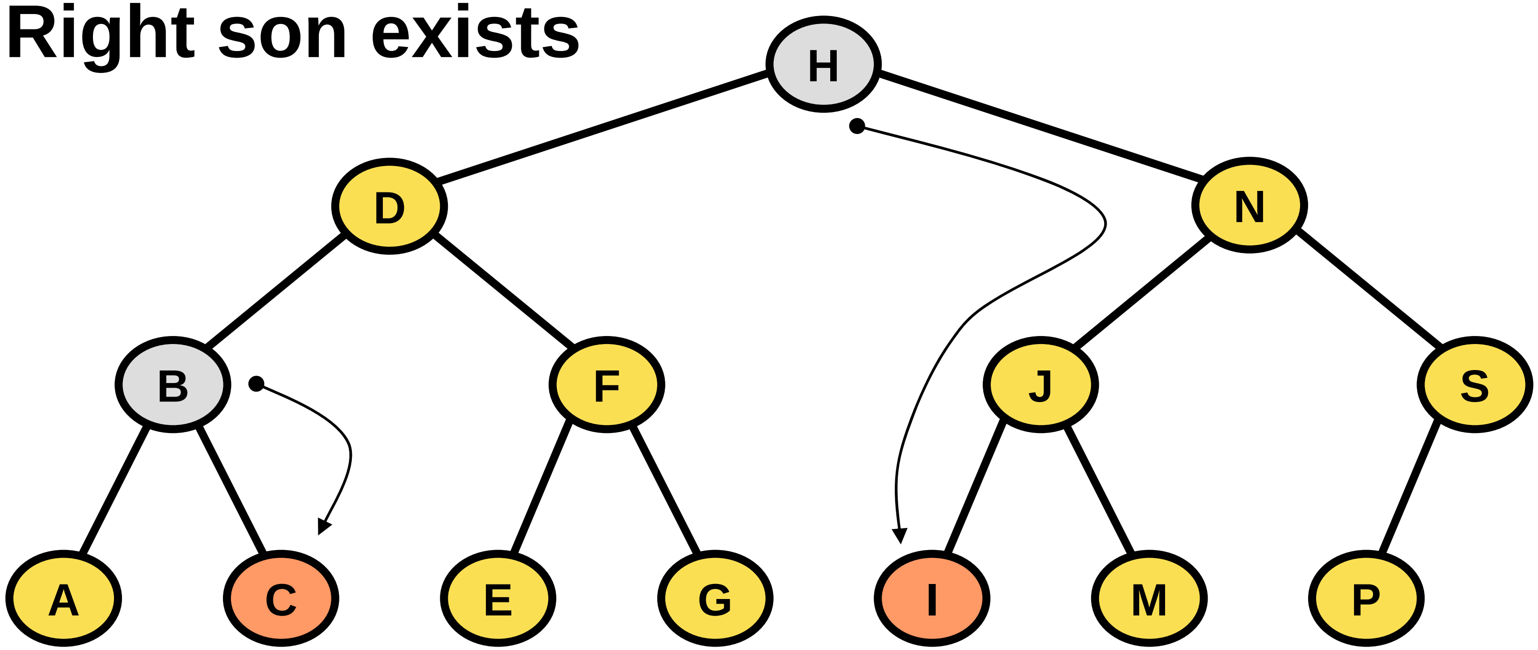

```
succ(B) -> C
succ(H) -> I
```

How?

Successor in BST

in the sorted order (in-order tree walk)

1. Right son exists



`succ(B) -> C`

`succ(H) -> I`

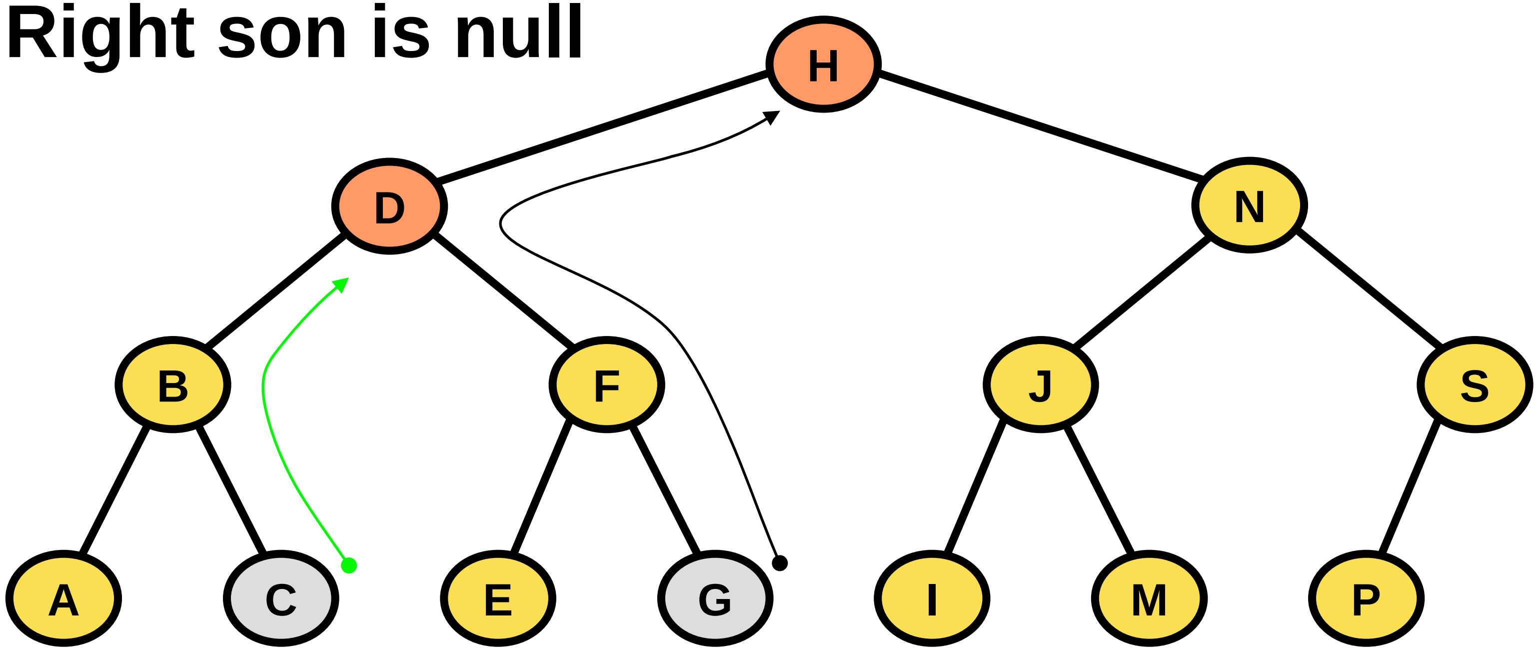
Find the *minimum* in the *right* tree

= min(x.right)

Successor in BST

in the sorted order (in-order tree walk)

2. Right son is null




```
succ(C) -> D
succ(G) -> H
```

How?

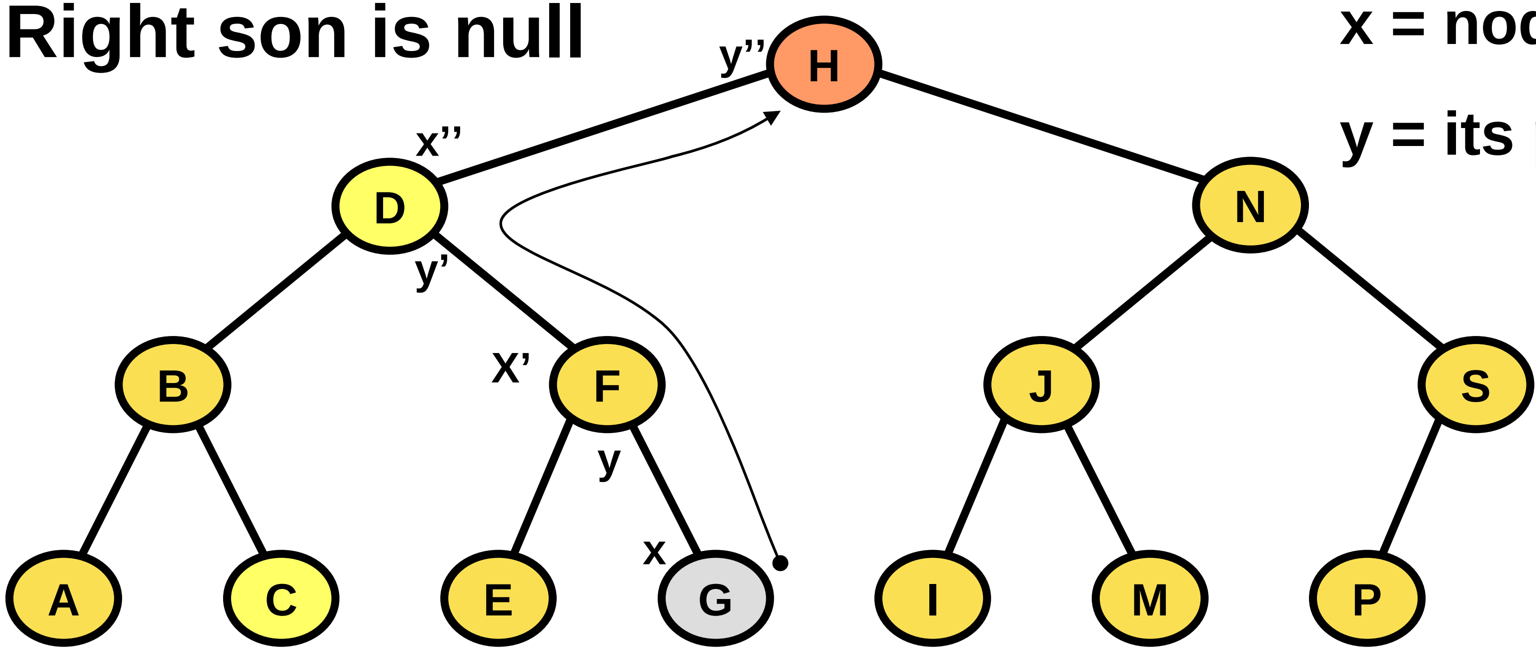
Successor in BST

in the sorted order (in-order tree walk)

2. Right son is null

x = node on path

y = its parent




```
succ(G) -> H
```

Find the *minimal parent to the right*

(the minimal parent the node is left from)

Successor in BST

in the sorted order (in-order tree walk)

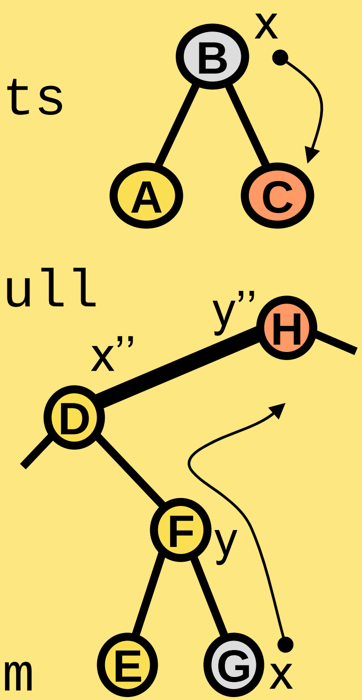
```
Node treeSuccessor( Node x )
{
  if( x == null ) return null;

  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
}
```

x = node on path

y = its parent



Java-like pseudo code

Predecessor in BST

in the sorted order (in-order tree walk)

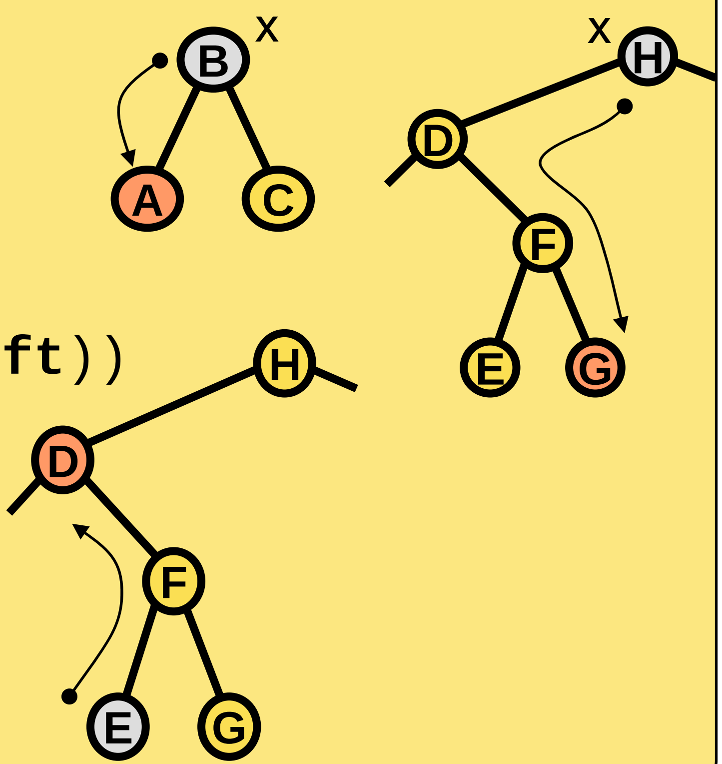
```
Node treePredecessor( Node x )
{
  if( x == null ) return null;

  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;
}
```

x = node on path

y = its parent



Java-like pseudo code

Operational Complexity

The following dynamic-set operations:

Search,

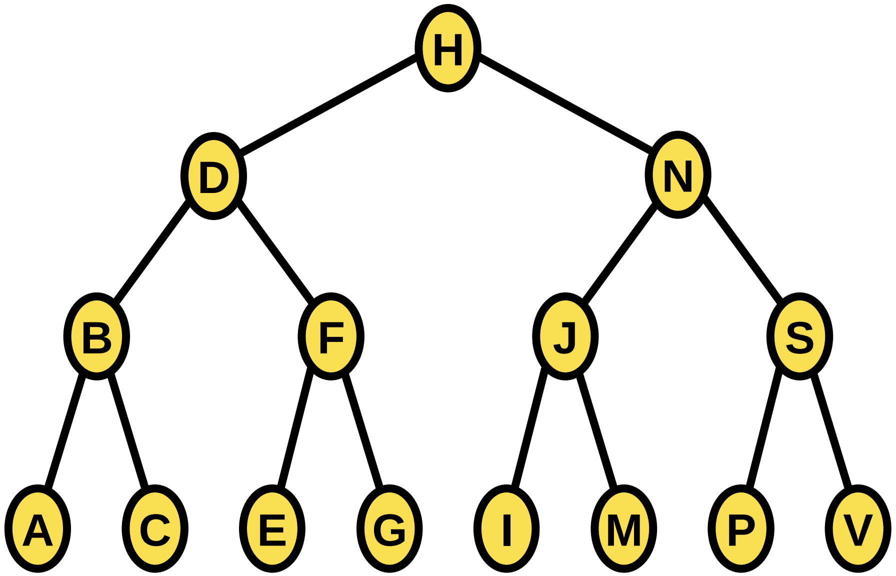
Maximum, Minimum,

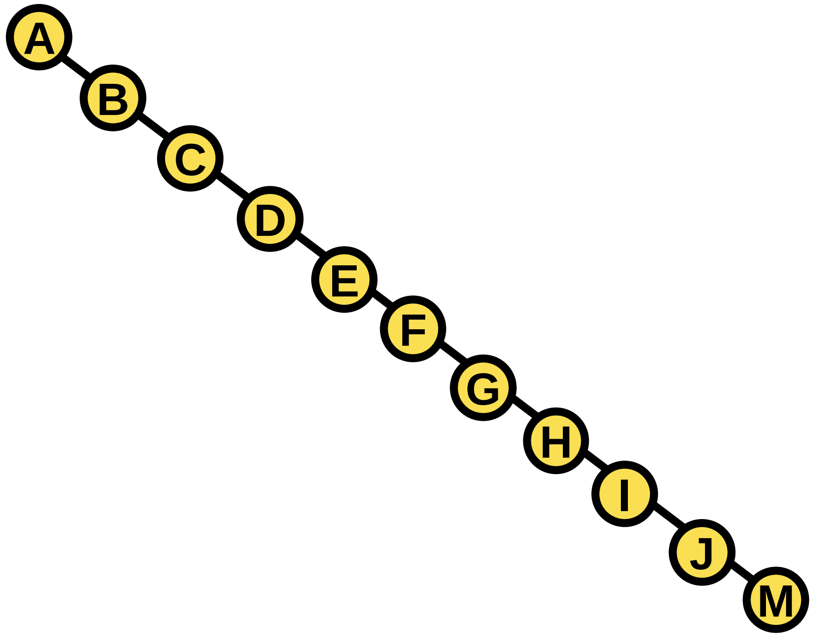
Successor, Predecessor

can run in *O(h)* time

on a binary tree of height *h*. …. *what h?*

Operational Complexity





$h = \log_2(n)$

=> *O(log(n))*

$h = n$

=> *O(n)* !!!

=> balance the tree!!!

Operational Complexity

The following dynamic-set operations:

Search,

Maximum, Minimum,

Successor, Predecessor

can run in O(*n*) time

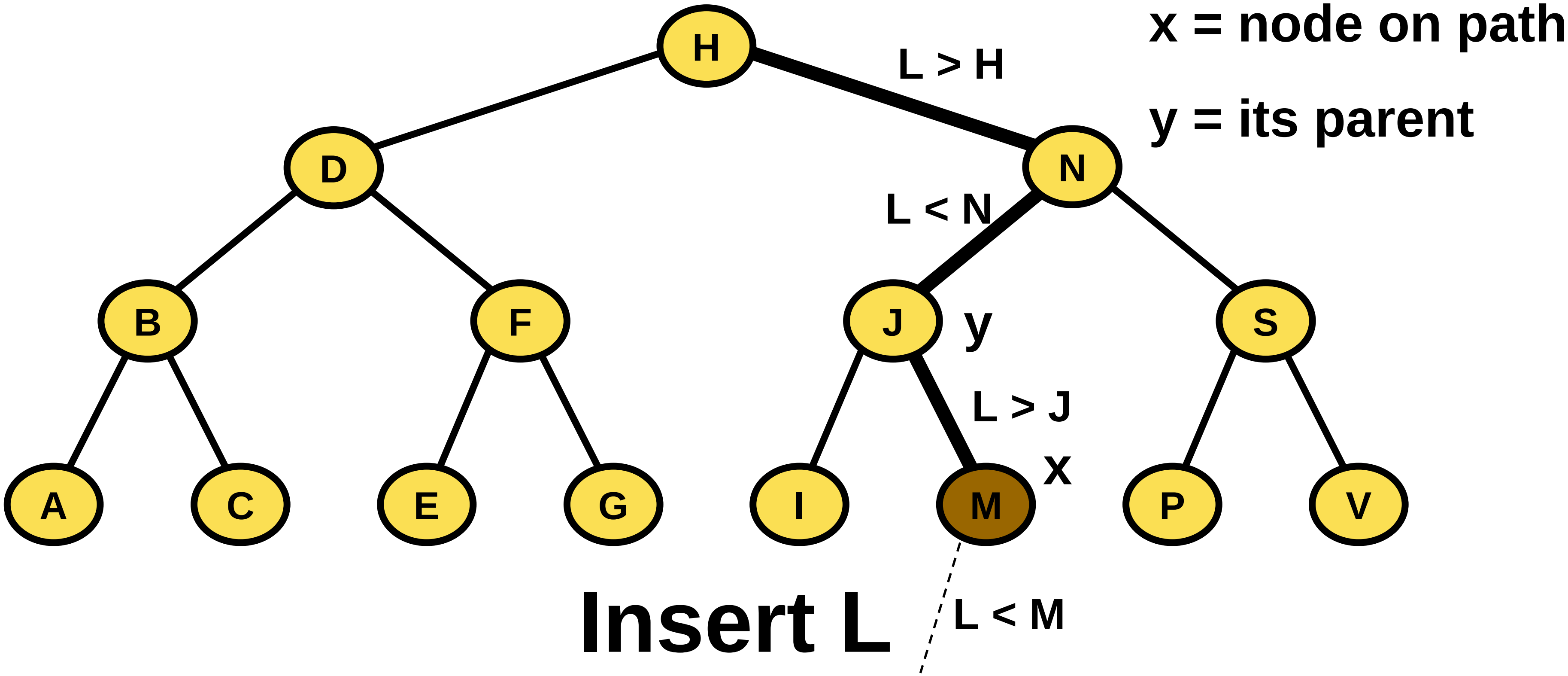
on a not-balanced binary tree with *n* nodes.

and

can run in O(*log*(*n*)) time

on a balanced binary tree with *n* nodes.

Insert (vložení prvku)



Insert L

1. find the parent leaf … M
2. connect new element as a new leaf … M.left

Insert (vložení prvku)

x = node on path

y = its parent

```
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

This is a simple version – with no update for equal keys

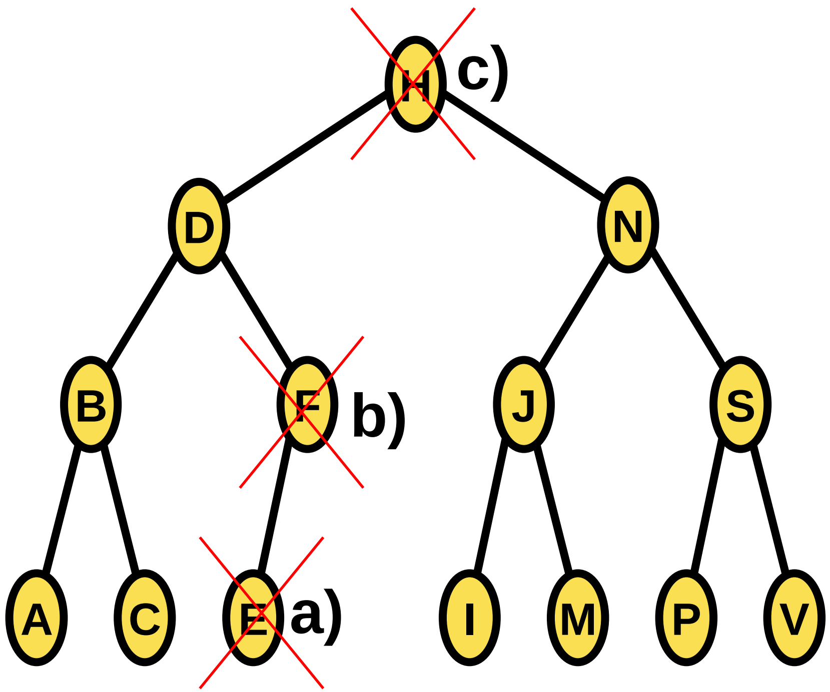
Operational Complexity

Insert

1. find the parent leaf
   O(h), O(log(n)) on balanced tree
2. connect the new element as a new leaf
   O(1)

=> O(h), i.e. O(log(n)) on balanced tree

Delete (odstranění prvku)



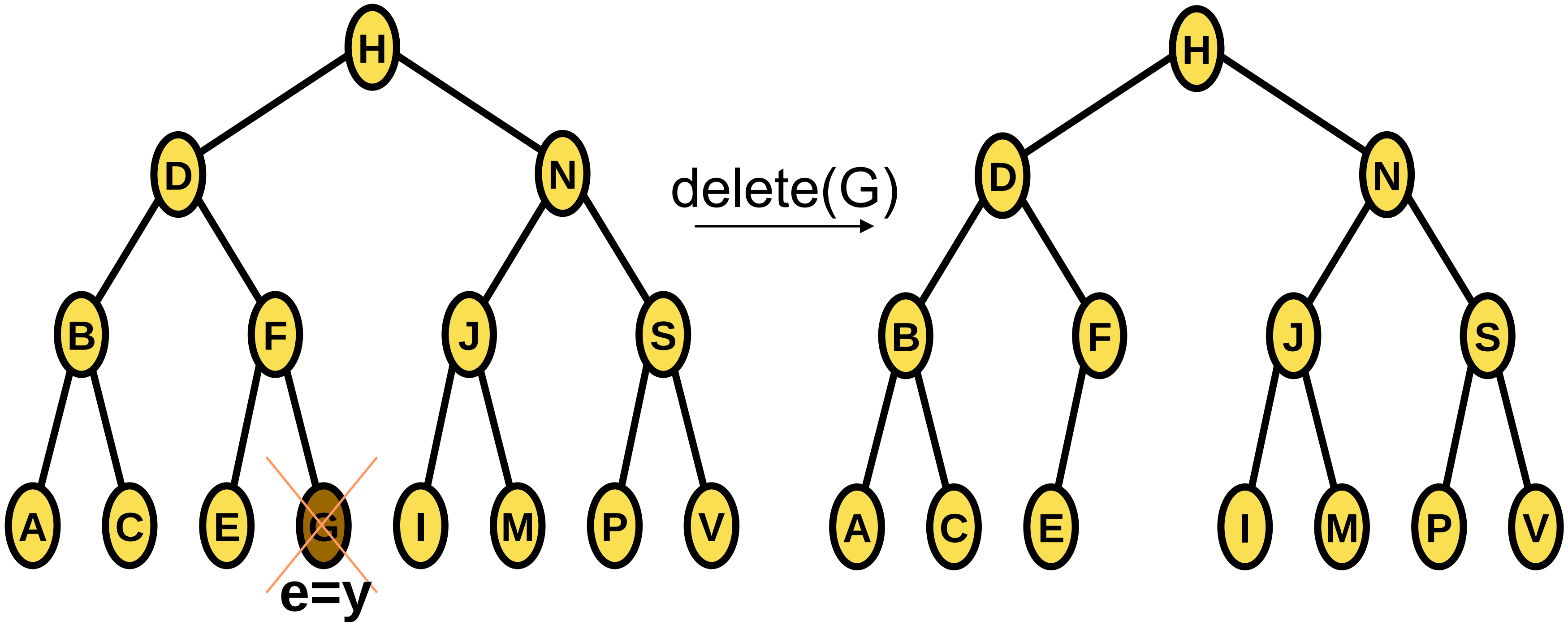
Delete – 3 cases

a) leaf has no children

b) node with one child

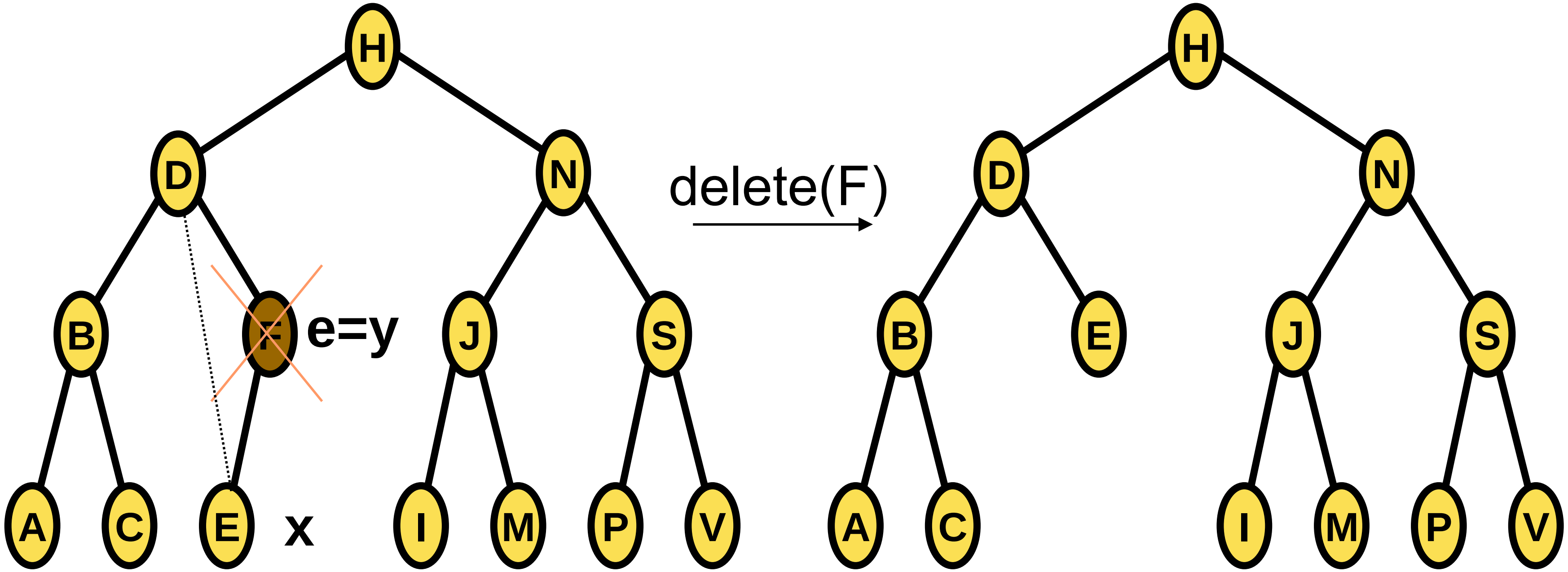
c) node with two children (problem with two subtrees)

Delete (odstranění prvku)
a) leaf (smaž list)



a) leaf has no children -> it is simply removed

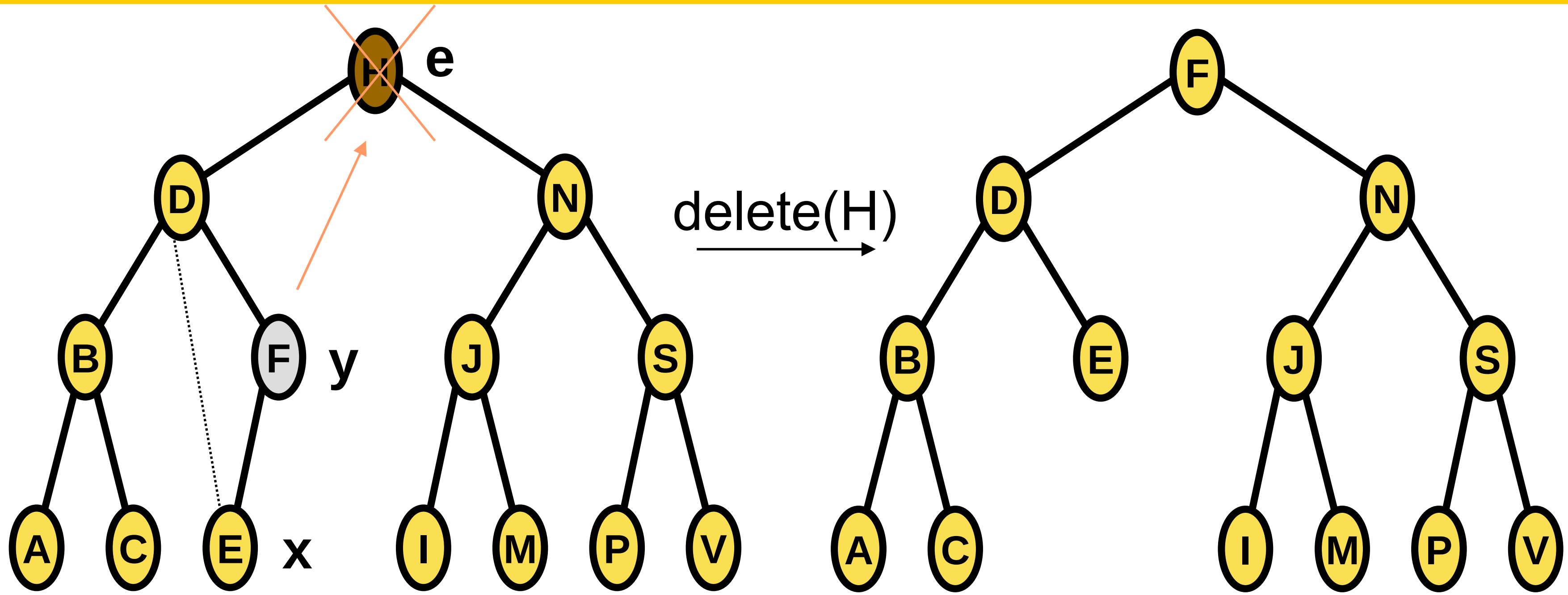
Delete (odstranění prvku)
b) node with one child (vnitřní s 1 potomkem)



b) node has one child -> splice the node out

(přemosti vymazaný uzel)

Delete (odstranění prvku)
c) node with two children (se 2 potomky)



c) node has two children -> replace node with
predecessor (or successor)
(it has no or one child)

and delete the predecessor

Delete (odstranění prvku)

Variables:

`t` tree

`e` element to be *logically* deleted from **`t`**

`y` element to be *physically* deleted from **`t`**

`x` is *y*'s only son or null

– will be connected to y's parent

Delete (odstranění prvku)

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

      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 y)
}
```

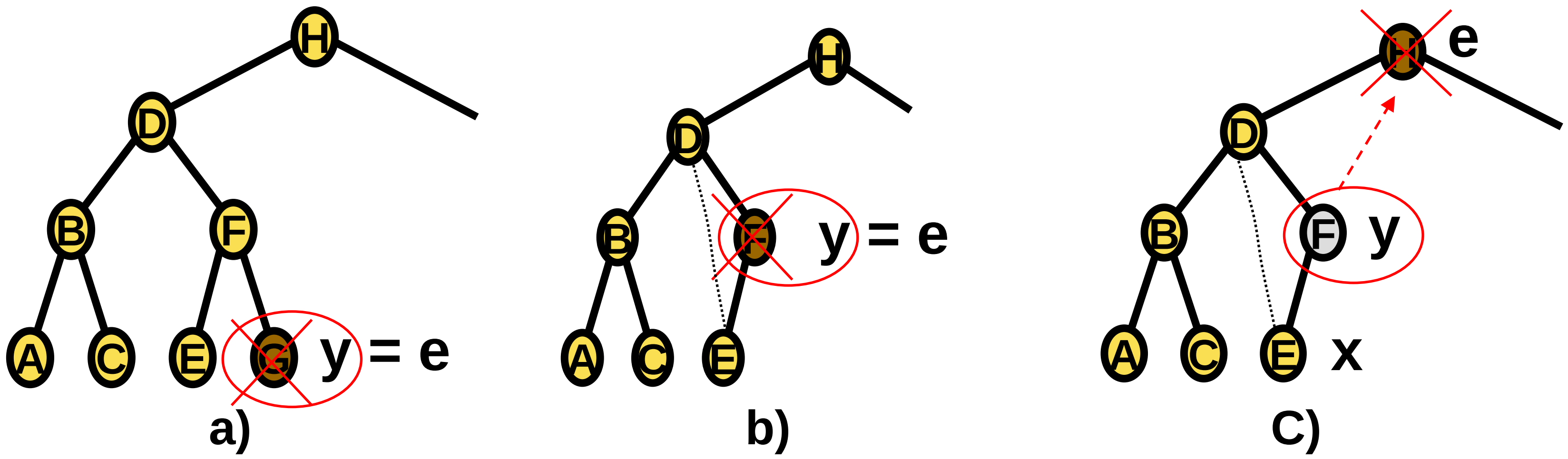
Delete (odstranění prvku)

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

  1.find node y

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

cont…

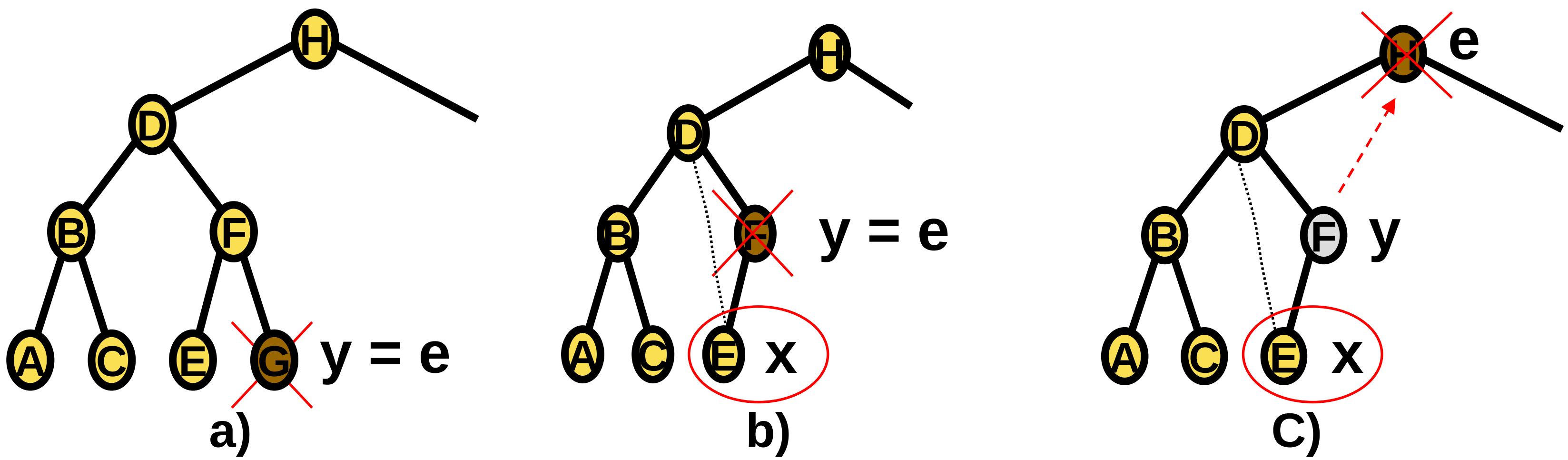


Delete (odstranění prvku)

... Cont // On which side the child is?

```
2. find x = y’s only child (L or R) or null
      if( y.left != null )      // a) null, b,c) only child
    x = y.left;
      else
    x = y.right;
```

cont…

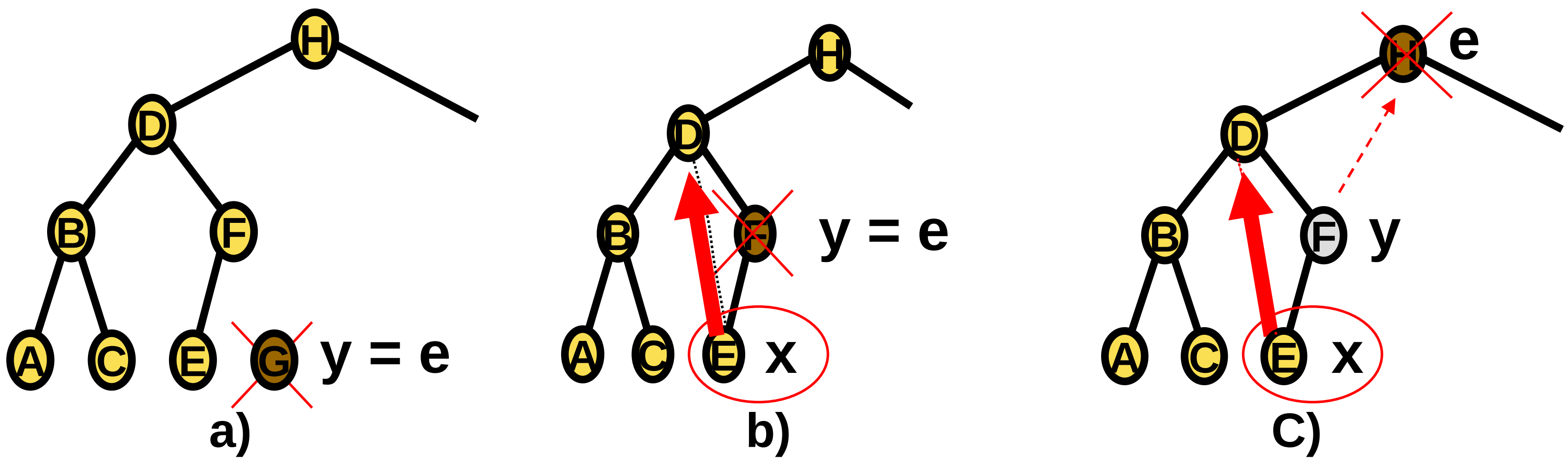


Delete (odstranění prvku)

... cont

```
      ...
 3. link x up with its new parent (former parent of y)
 if( x != null ) x.parent = y.parent;  // b,c)
```

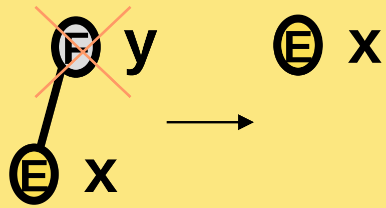
cont…

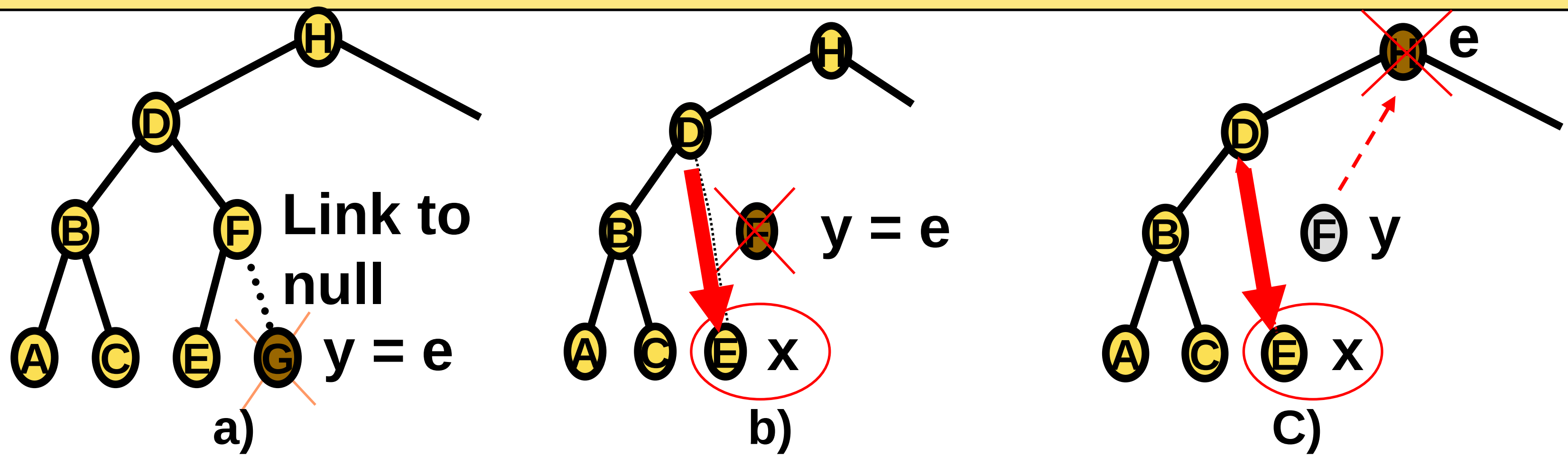


Delete (odstranění prvku)

```
...
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
  ...
```

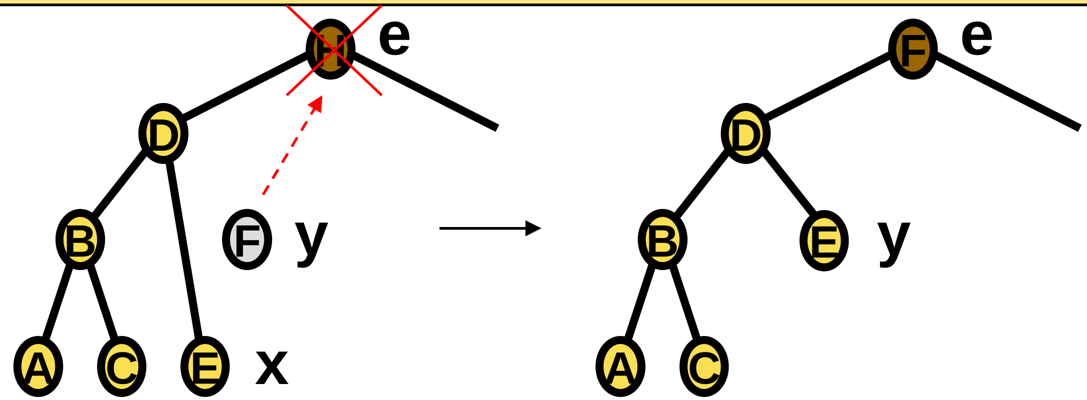
cont...





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)
  return y;          // instead of delete
}
```



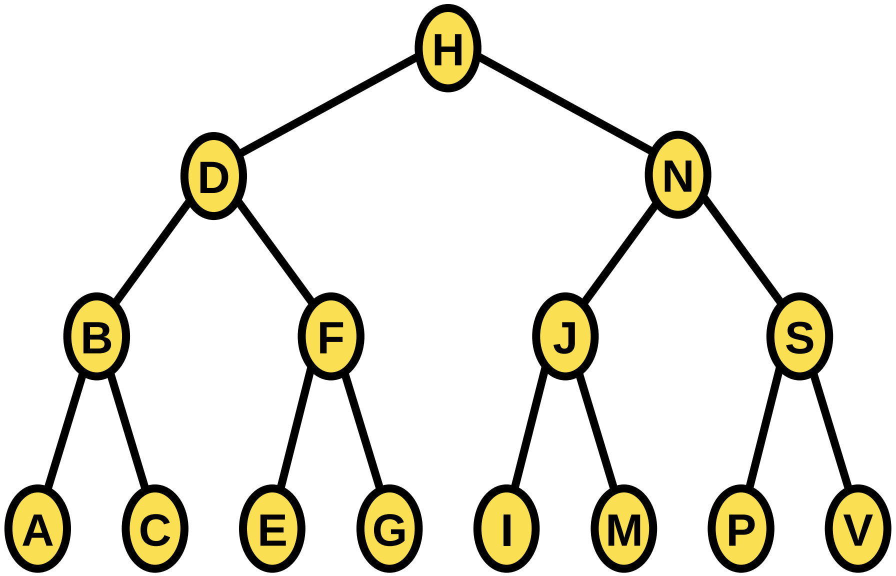
Delete on a single page

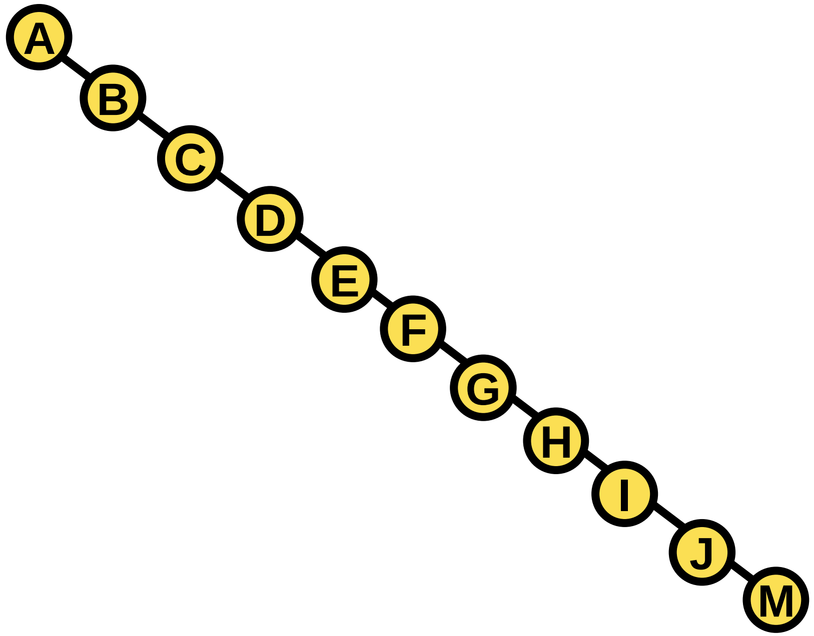
```
Node treeDelete( Tree t, Node e ) // e..node to logically delete
{ Node x, y;                      // y…node to physically delete, x…y's only son

  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
}
```

And the operational complexity?

Operational Complexity





$$h = \log_2(n)$$

$$h = n \text{ !!!}$$

O(log(n))

O(n) **!!!**

=> balance the tree!!!

Searching – talk overview

Typical operations

Quality measures

Implementation in an array

- Sequential search
- Binary search

Binary search tree – BST (*BVS*) – in dynamic memory

- Node representation
- Operations
- Tree balancing

Tree balancing

- Balancing criteria
- Rotations
- AVL – tree
- Weighted tree

Tree balancing

Why?

To get the *O(log n)* complexity of search,...

How?

By *local modifications* reach the global goal

(*local modifications* = rotations)

Kritéria vyvážení stromu

Silná podmínka – shoda *h* podstromů (Ideální případ)

Pro všechny uzly platí:

počet uzlů vlevo = počet uzlů vpravo

Slabší podmínka – násobek *h* => $c*h = O(\log n)$

- **výška** podstromů - AVL strom
- **výška** + počet potomků - 1-2 strom, ...
- **váha** podstromů (počty uzlů) - váhově vyvážený strom
- stejná černá výška – Červeno-černý strom

Tree balancing criteria

Strong criterion (Ideal case)

For all nodes:

No of nodes to the left = No of nodes to the right

Weaker criterion: => $c*h = O(\log n)$

- subtree heights - AVL tree
- height + number of children - 1-2 tree, ...
- subtree weights (No of nodes) - weighted tree
- equal Black height – Red-Black tree

Tree balancing

Balancing criteria

Rotations

AVL – tree

Weighted tree

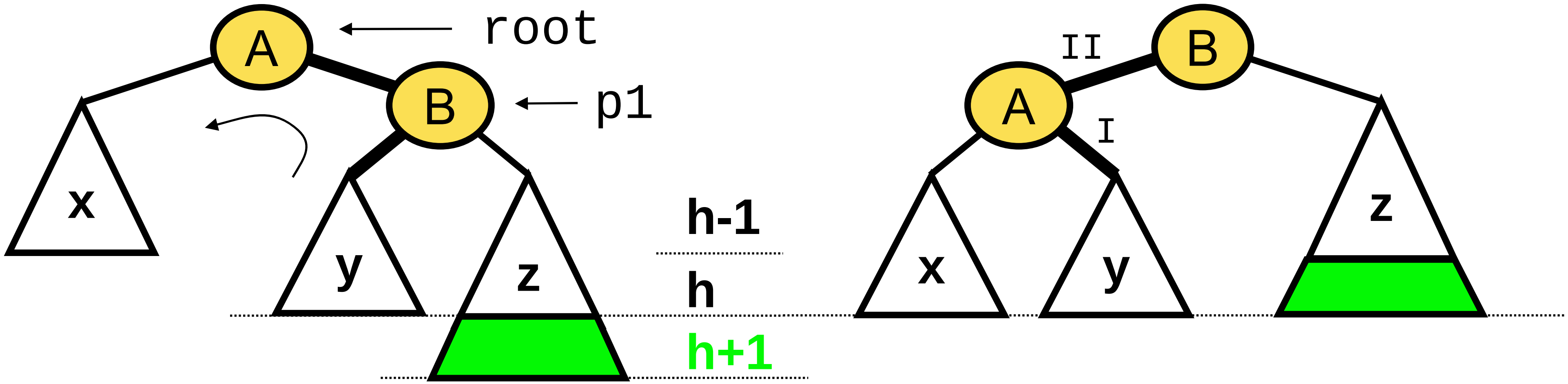
Rotations

- Balance the tree (by changing tree structure)
- Preserve mutual relation of nodes

  - what was left, will stay left, …

  - left son is smaller, right son is larger,…

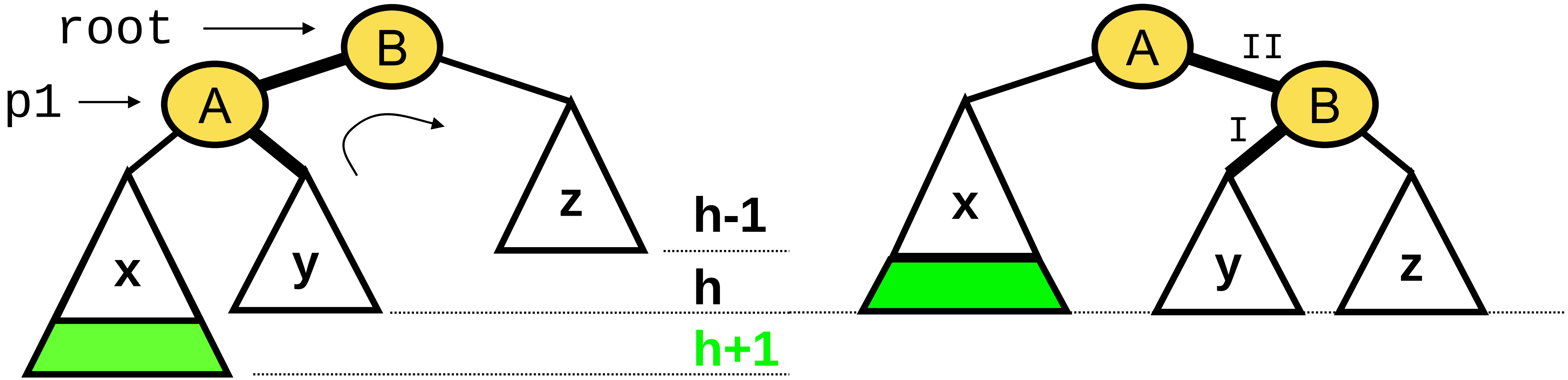
L rotace (Left rotation)




```
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

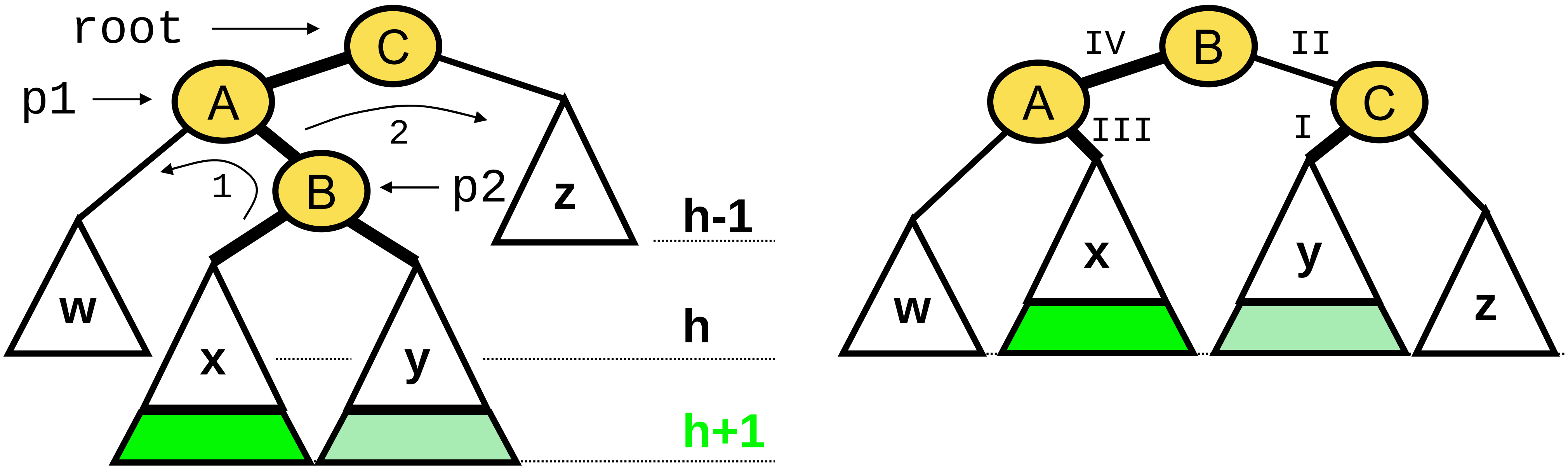
R rotace (right rotation)




```
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

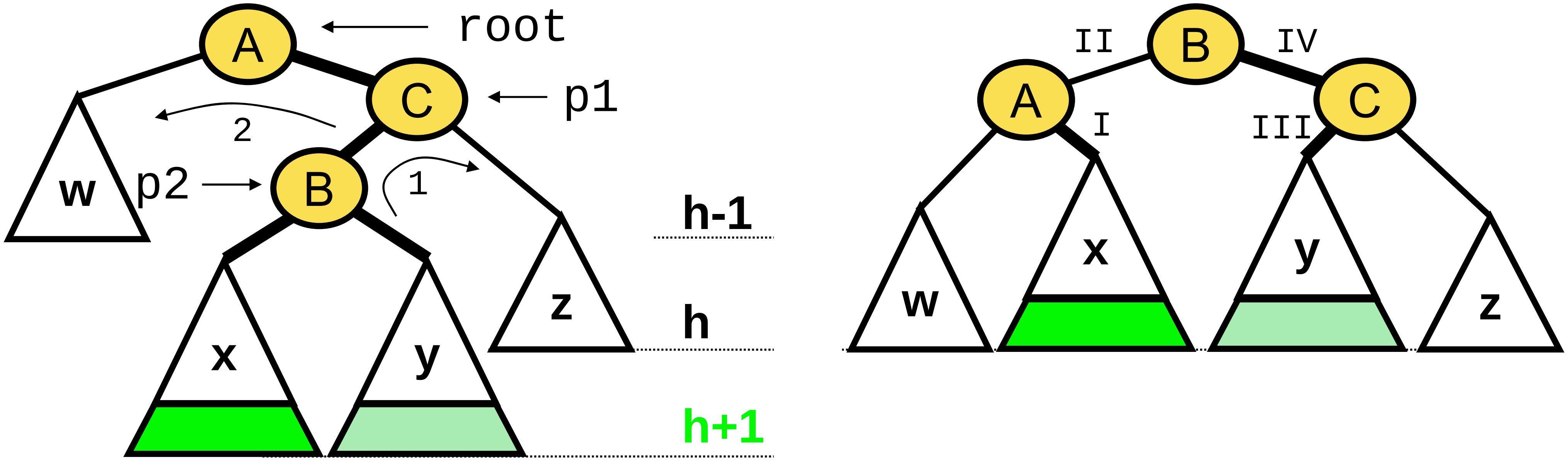
LR rotace (left-right rotation)




```
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

Tree balancing

Balancing criteria

Rotations

AVL Tree

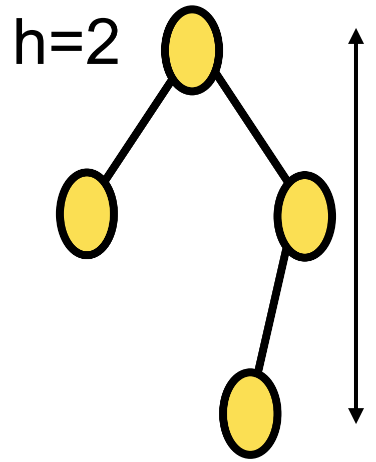
Weighted tree

AVL strom

AVL strom **[Richta90]**

- Výškově vyvážený strom
- Georgij Maximovič Adelson-Velskij a Evgenij Michajlovič Landis 1962
- Výška:
  - Prázdný strom: výška = -1
  - neprázdný: výška = výška delšího potomka
- Vyvážený strom:

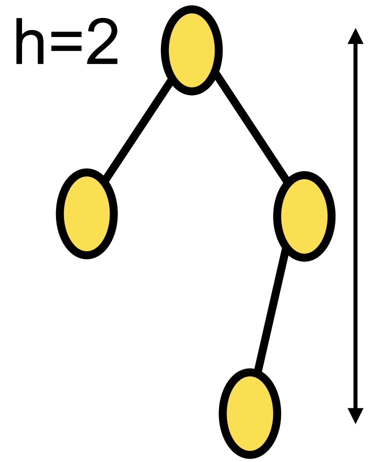
  rozdíl výšek potomků **`bal`** = {-1, 0, 1}



AVL Tree

AVL tree [Richta90]

- Height balanced BST
- Georgij Maximovič Adelson-Velskij and Evgenij Michajlovič Landis, 1962
- Height:
  - Empty tree: height = -1
  - Non-empty: height = height of the highest son



- Height balanced tree:

  difference of son heights in interval

  bal = {-1, 0, 1}

AVL tree

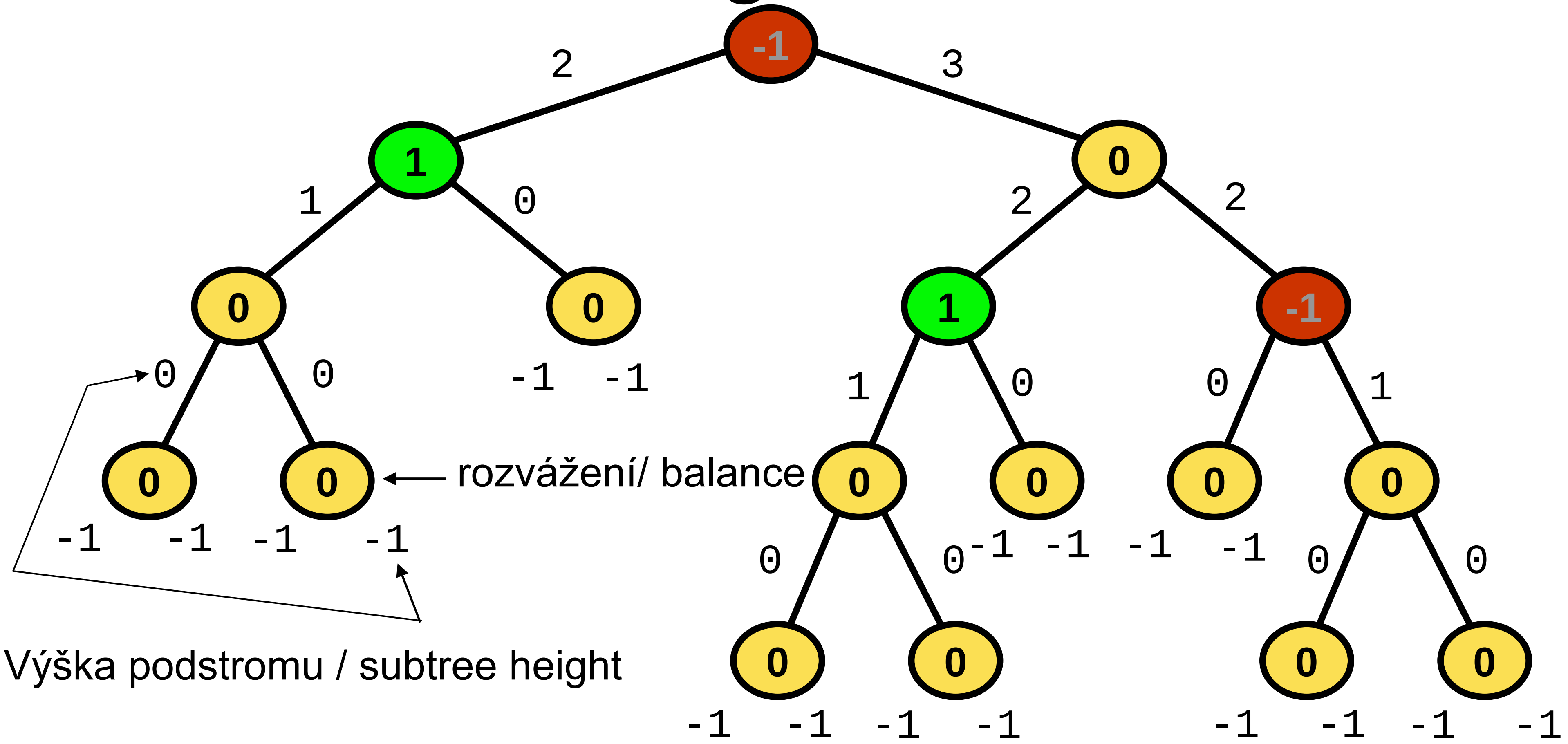
```
// A very inefficient recursive definition
```

```
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

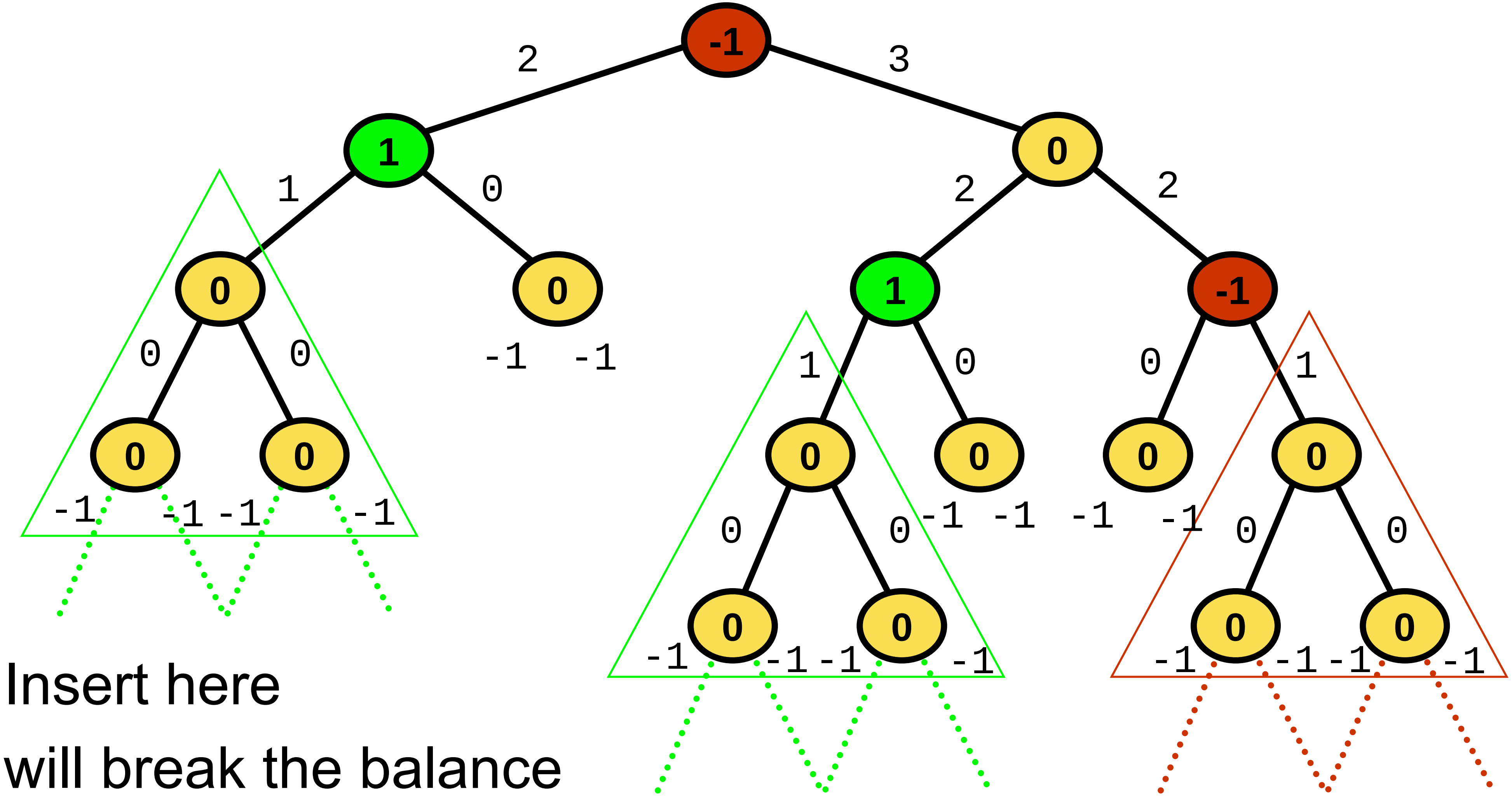
AVL strom - výšky a rozvážení
AVL tree - heights and balance



bal = {-1, 0, 1}

=> nodes with -1 and 1 absorb insertion or break the balance

AVL strom před vložením uzlu
AVL tree before node insertion



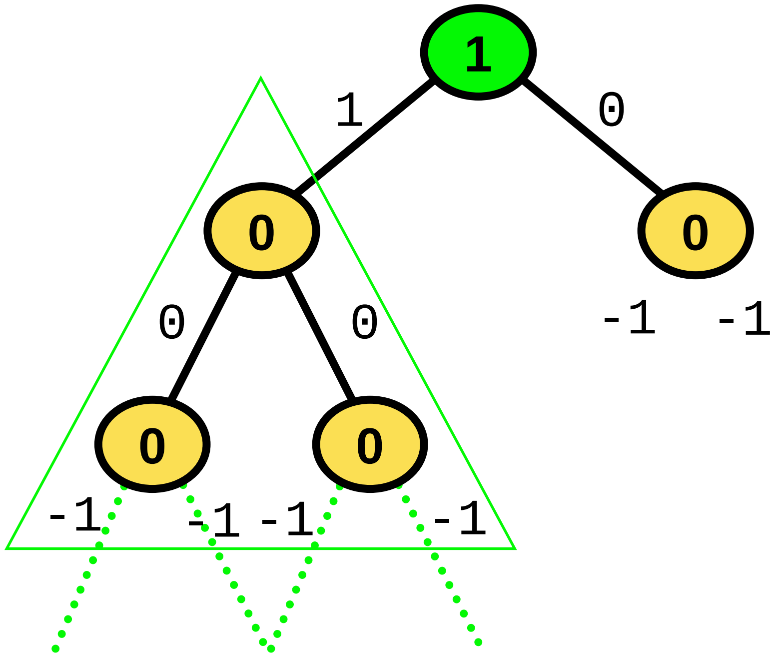
Insert here

will break the balance

AVL strom - nejmenší podstrom
AVL tree - the smallest subtree

Nejmenší podstrom, který se může přidáním uzlu rozvážit

The smallest sub-tree that can loose its balance by insertion



△ its “neutral” subtree

- is balanced: bal = 0
- remains balanced after insert bal∈⟨-1,+1⟩

Subtree with root 1

- absorbs insert right → 0
- breaks balance if insert left → 2

Smallest subtree

- modification near the leaves

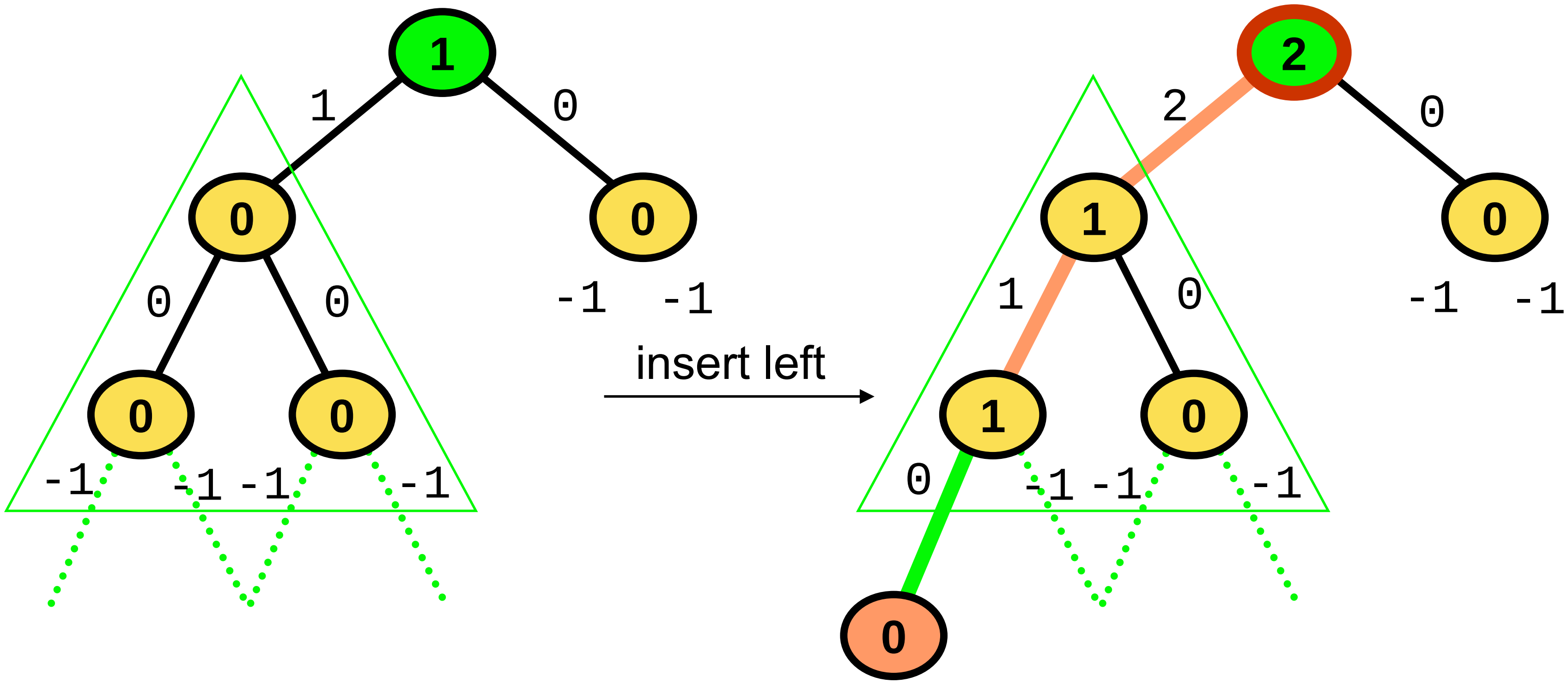
AVL tree

Node insertion – an example

AVL strom - vložení uzlu doleva
AVL tree - node insertion left

a) Podstrom se přidáním uzlu doleva rozváží

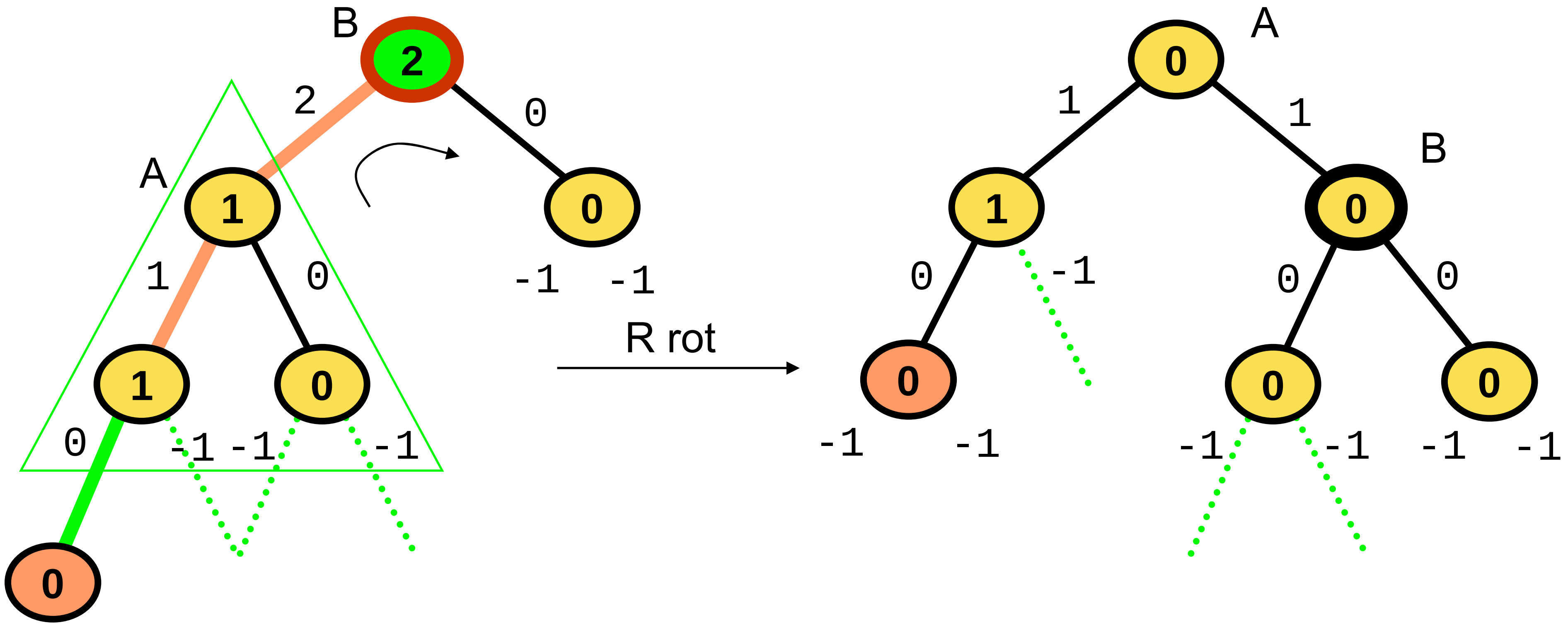
The sub-tree loses its balance by node insertion - left



AVL strom - pravá rotace
AVL tree - right rotation

a) Vložen doleva – doleva => korekce pravou rotací

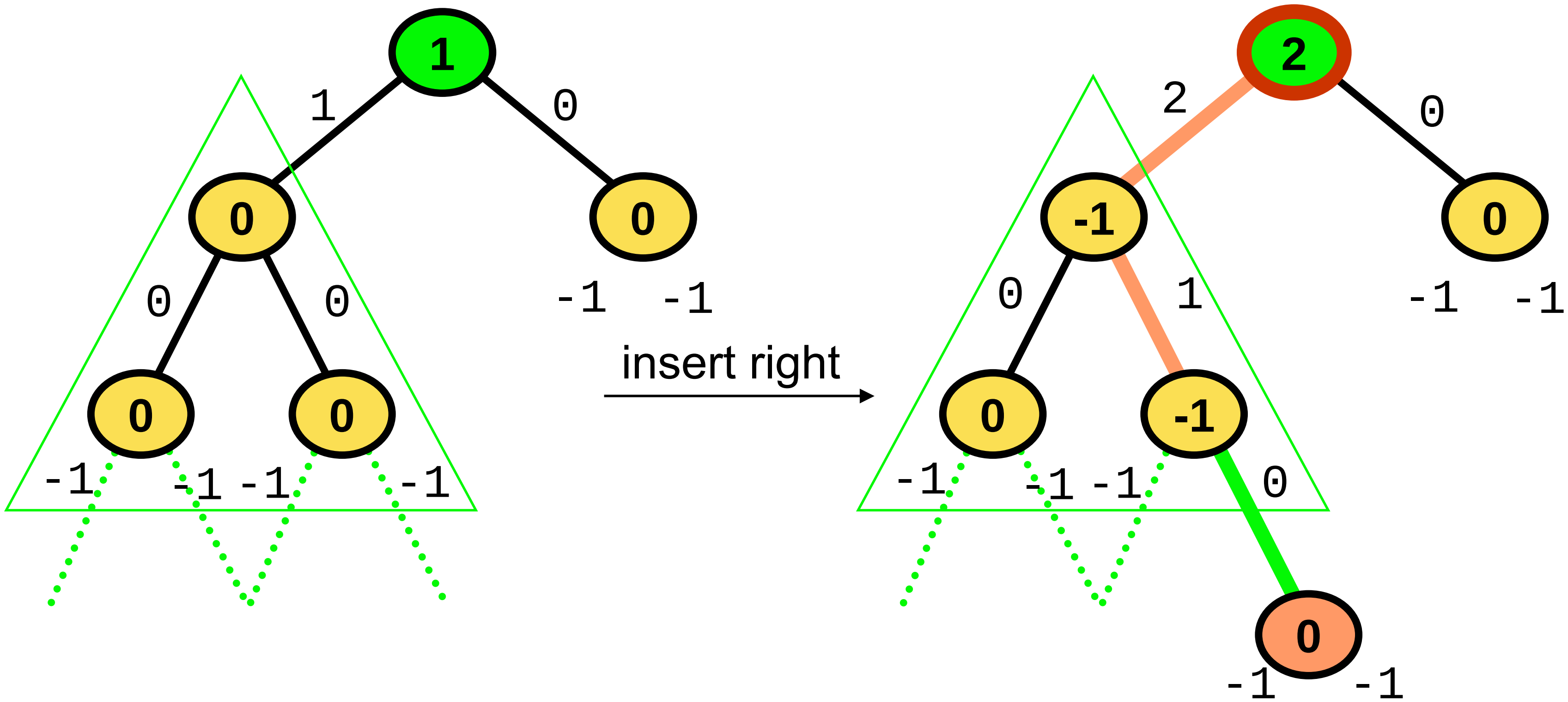
Node inserted to the left – left => balance by Right rotation



AVL strom - vložení uzlu doprava
AVL tree after insertion-right

b) Podstrom se přidáním uzlu doprava rozváží

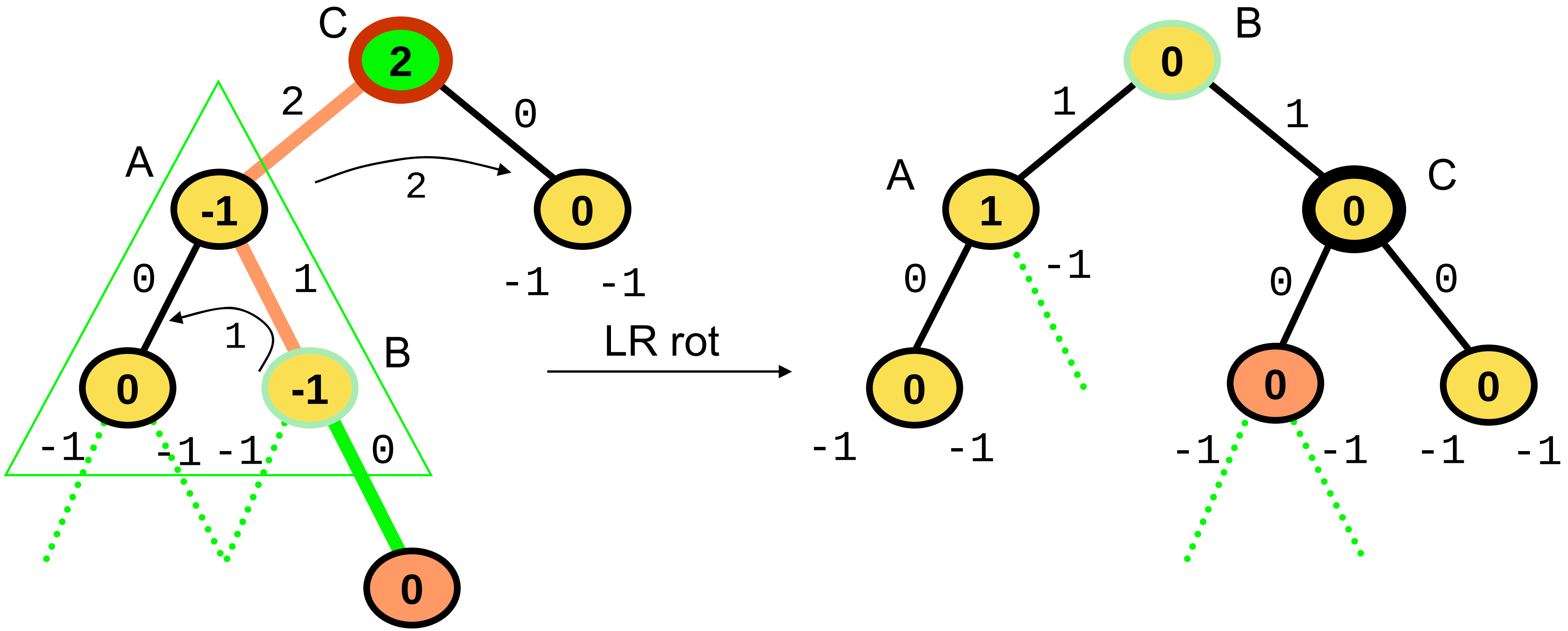
The sub-tree loses its balance by node insertion - right



AVL strom - pravá rotace
AVL tree - right rotation

b) Vložen doleva – doprava => korekce LR rotací

Node inserted left – right => balance by the LR rotation



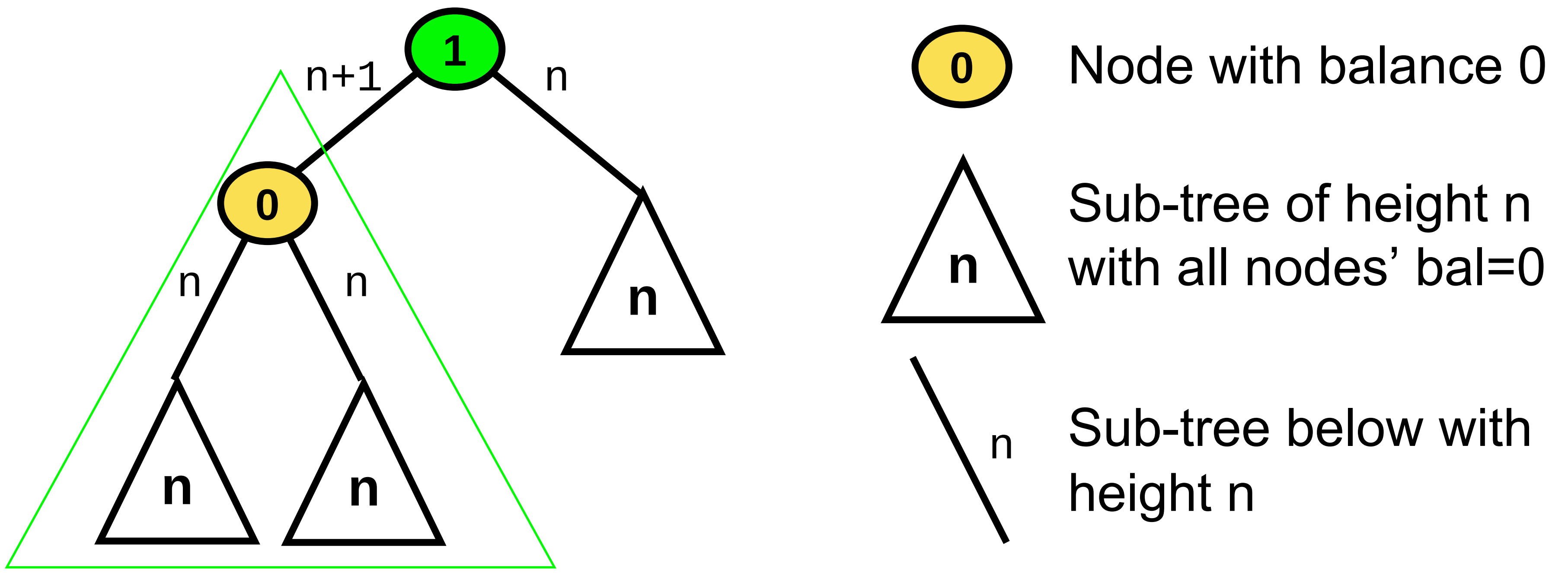
AVL tree

Node insertion - in general

AVL strom - nejmenší podstrom
AVL tree - the smallest subtree

Nejmenší podstrom, který se přidáním uzlu rozváží z bal = 0

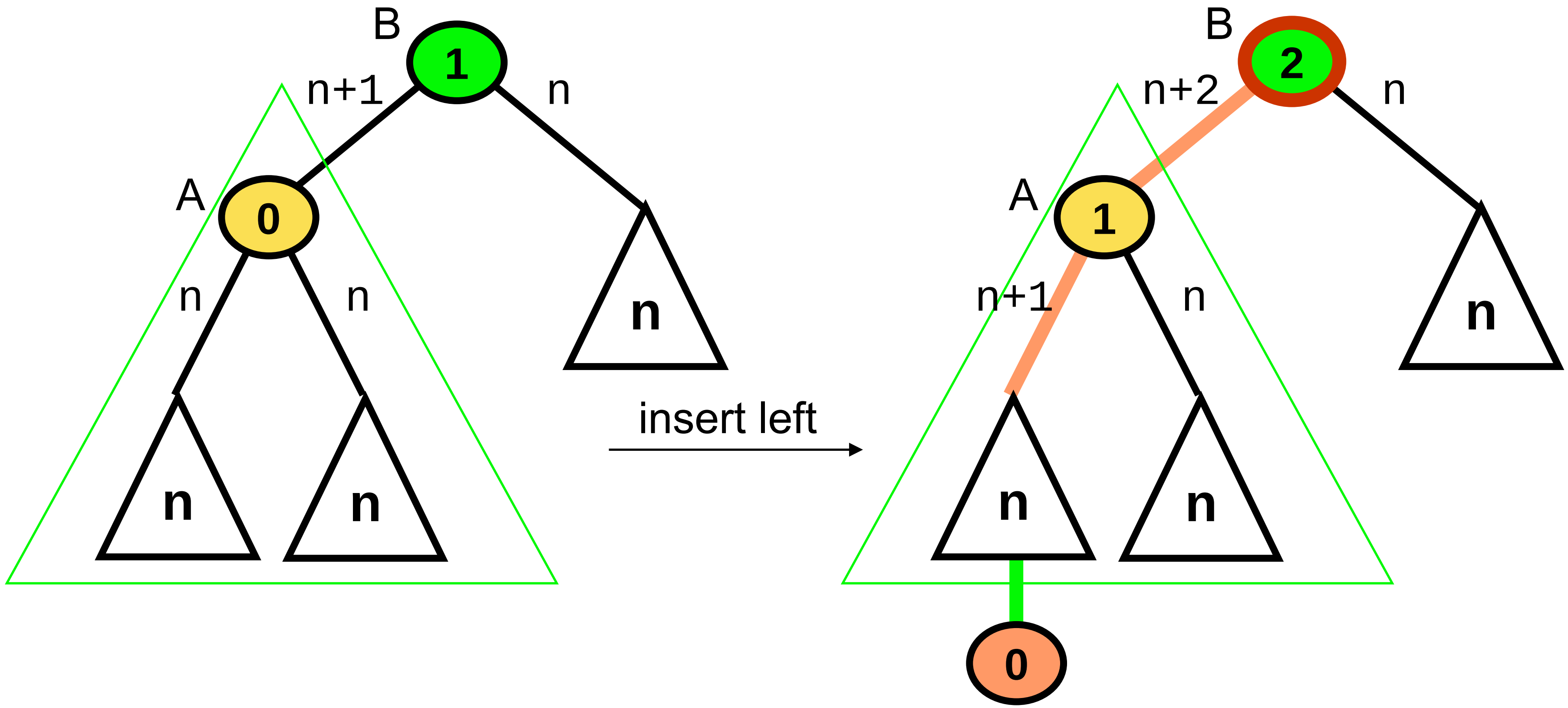
The smallest sub-tree that looses its bal = 0 by insertion



AVL strom - vložení uzlu doleva
AVL tree - node insertion left

a) Podstrom se přidáním uzlu doleva rozváží

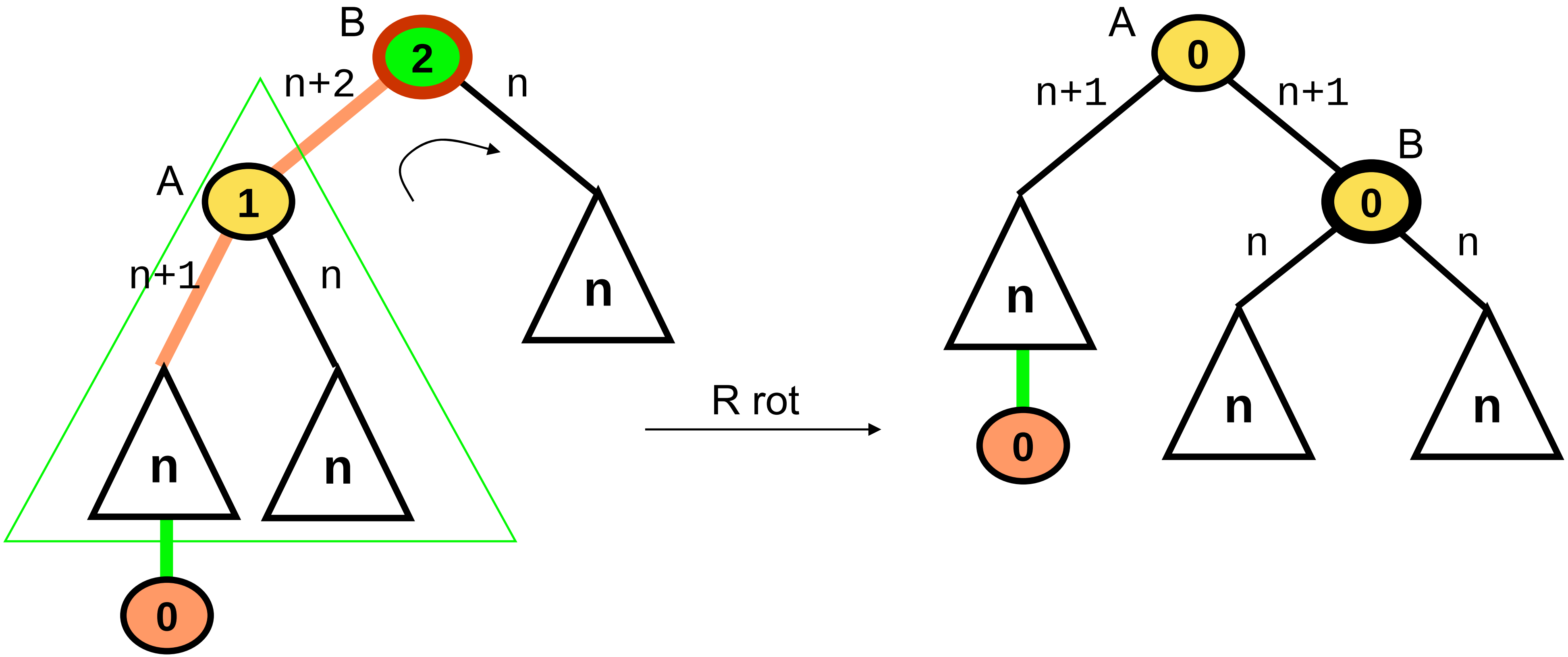
The sub-tree loses its balance by node insertion - left



AVL strom - pravá rotace
AVL tree - right rotation

a) Vložen doleva – doleva => korekce pravou rotací (R rotací)

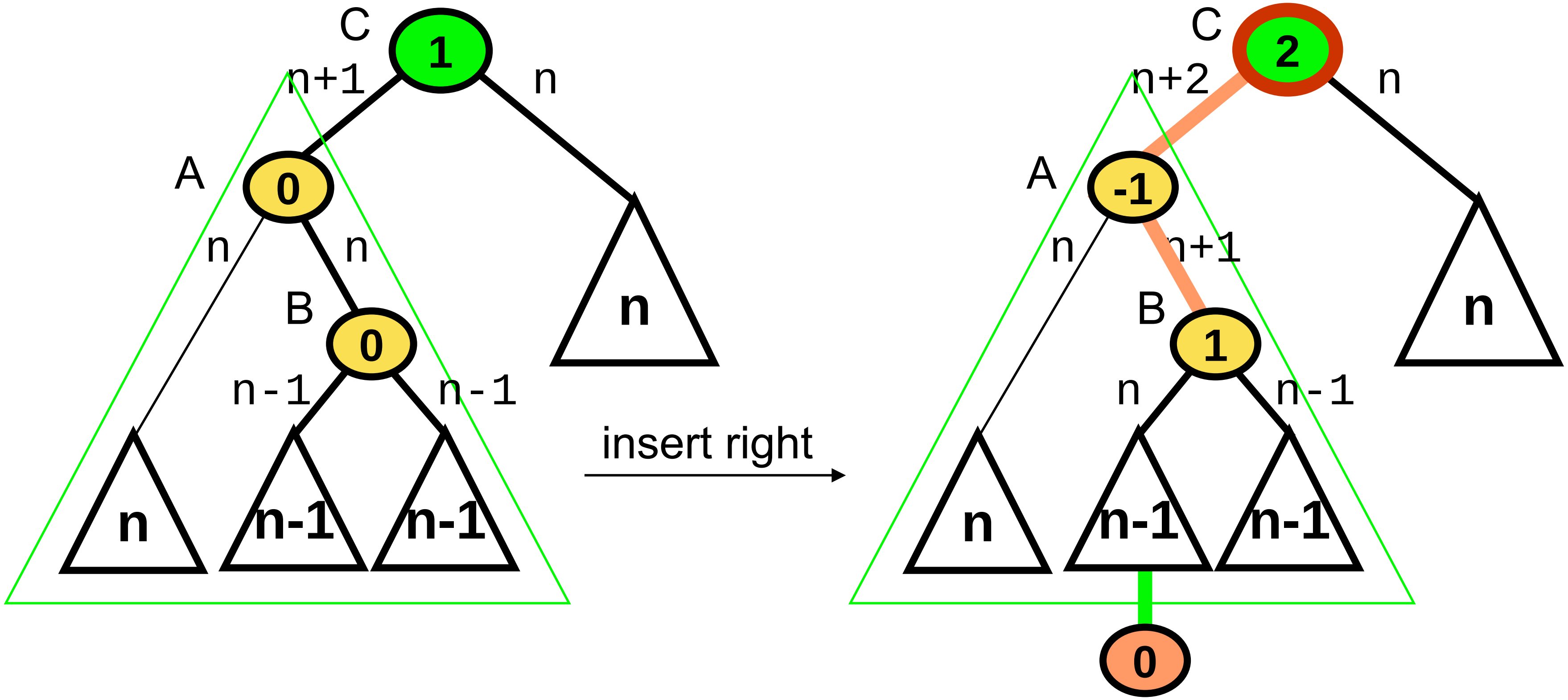
Node inserted to the left – left => balance by right rotation



AVL strom - vložení uzlu doprava
AVL tree after insertion-right

b1) Podstrom se přidáním uzlu doprava rozváží

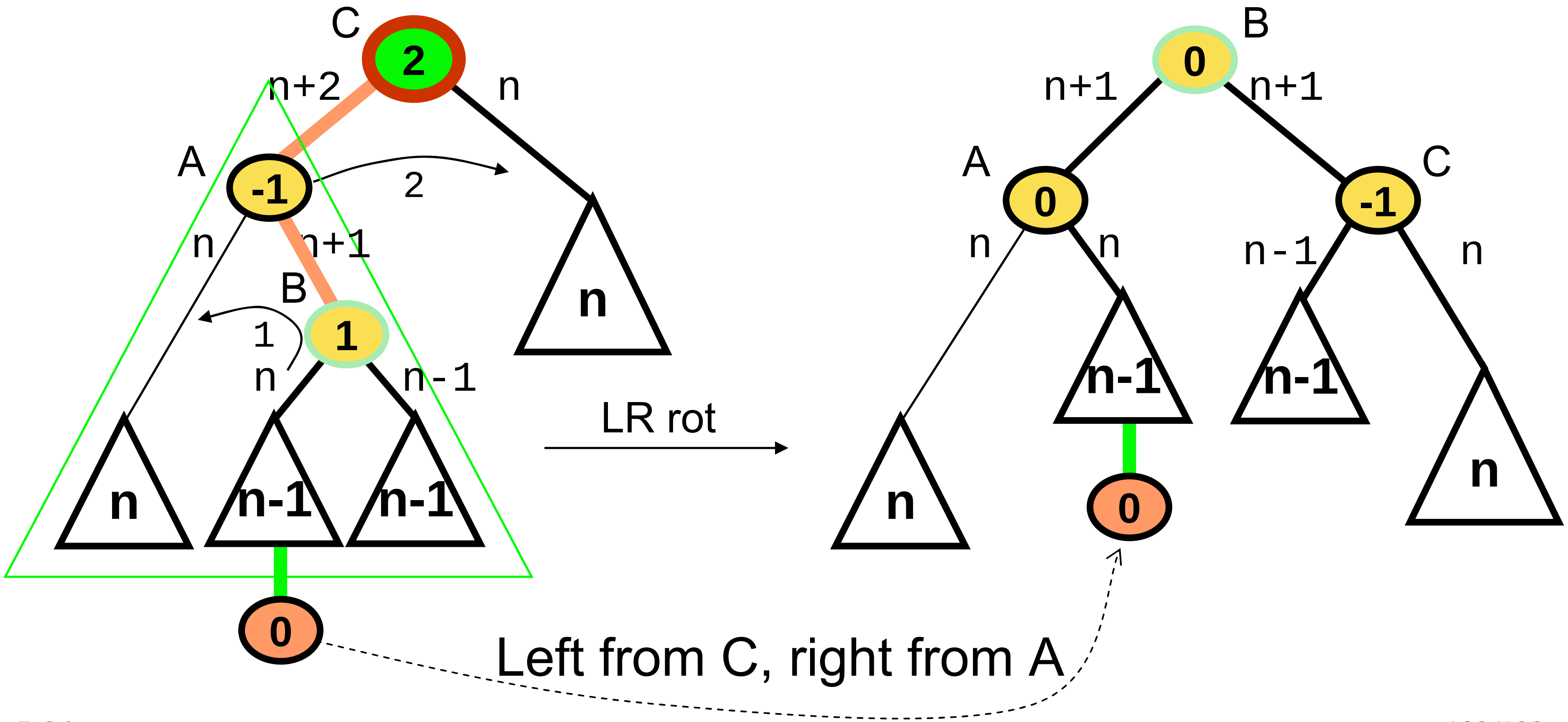
The sub-tree loses its balance by node insertion - right



AVL strom - pravá rotace
AVL tree - right rotation

b1) Vložen doleva – doprava => korekce LR rotací

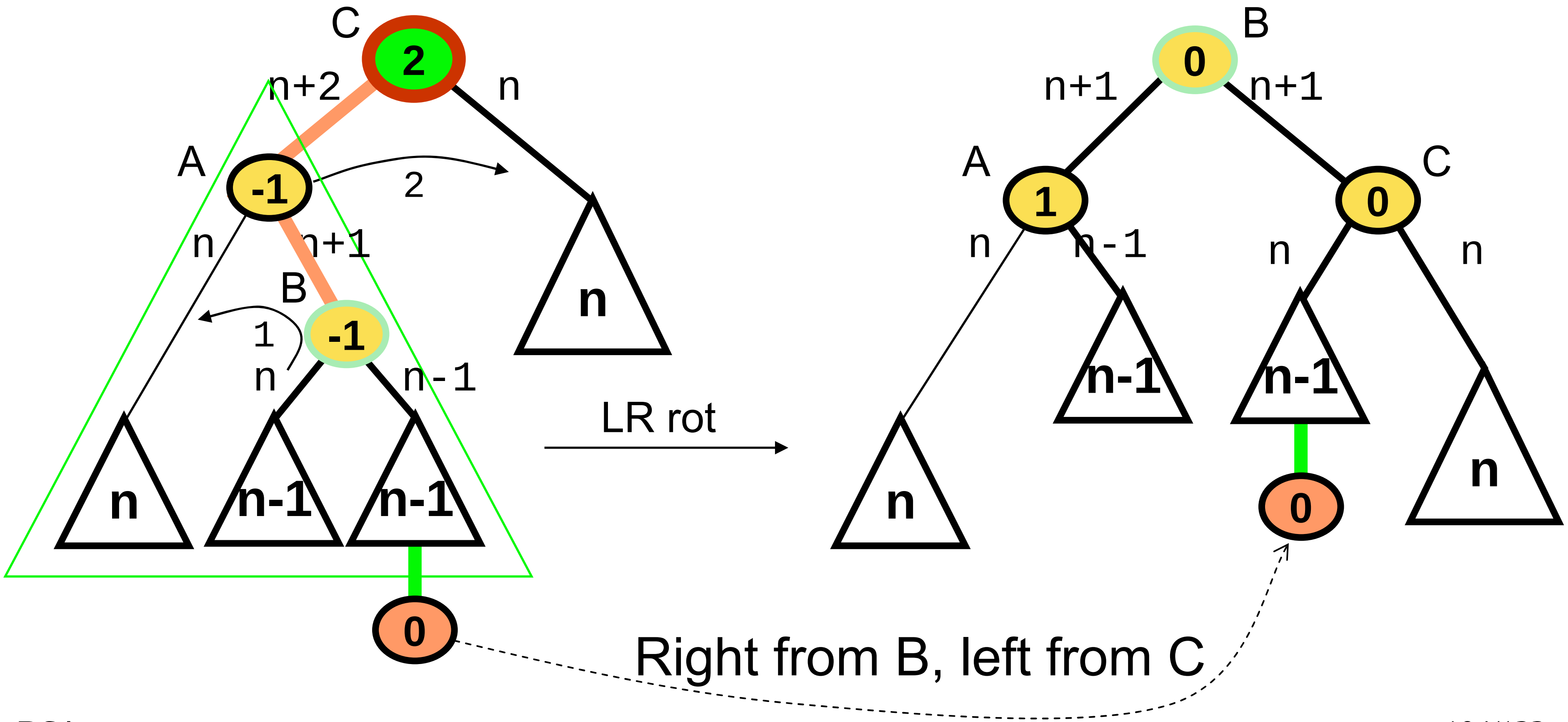
Node inserted left – right => balance by the LR rotation



AVL strom - pravá rotace
AVL tree - right rotation

b2) Vložen doleva – doprava => korekce LR rotací

Node inserted left – right => balance by the LR rotation



BST Insert without balancing

```
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

AVL Insert (with balancing)

```
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 - variables & init

```
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

  1.init
  cur = t.root; fcur = null;
  a = cur, b = null;

  2. find the place for insert
```

Java-like pseudo code

AVL Insert - find place for insert

```
...

2. find the place for insert
while(( cur != null ) and !found )
{
    if( e.key == cur.key ) found = true;
    else {
          fcur = cur;              // father of cur
          if( e.key < cur.key )
               cur = cur.left;
          else cur = cur.right;
          if(( cur != null) and ( bal(cur) != 0 )){
           //remember possible place for unbalance
                a = cur; // the deepest bal = +1 or -1
          }
    }
} ...
```

AVL Insert - replace or insert new

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

AVL Insert - balance the subtree

```
   ... // !found continues
```

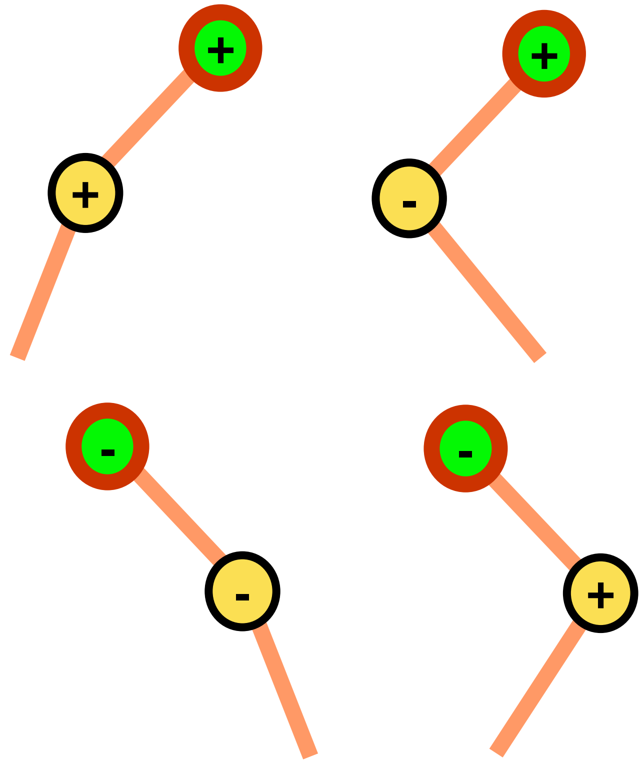
4.balance the tree, if necessary

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

AVL Insert - balance the subtree

4. Balance summary

a	b	Rotation
+	+	R rotation
+	–	LR rotation
–	+	RL rotation
–	–	L rotation



AVL - výška stromu

For AVL tree S with *n* nodes holds

Height *h(S)* is at maximum 45% higher in comparison to ideally balanced tree

$\log_2(n+1) \leq h(S) \leq 1.4404 \log_2(n+2) - 0.328$

[Hudec96], [Honzík85]

Tree balancing

Balancing criteria

Rotations

AVL – tree

Weighted tree

Váhově vyvážené stromy

(stromy s ohraničeným vyvážením)

Váha uzlu *u* ve stromě *S*:

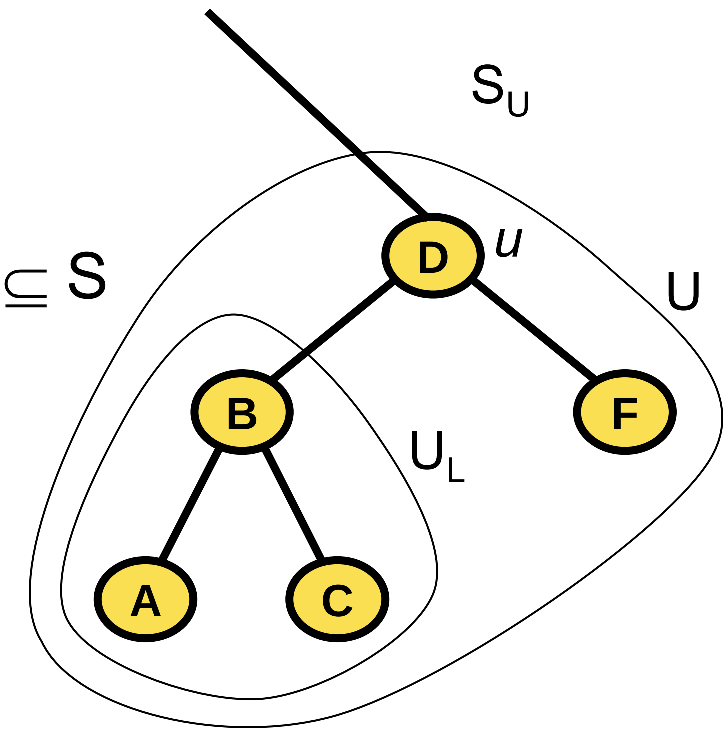
$v(u) = 1/2$, když je *u* listem

$v(u) = (|U_L| + 1) \ / \ (|U| + 1)$,

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



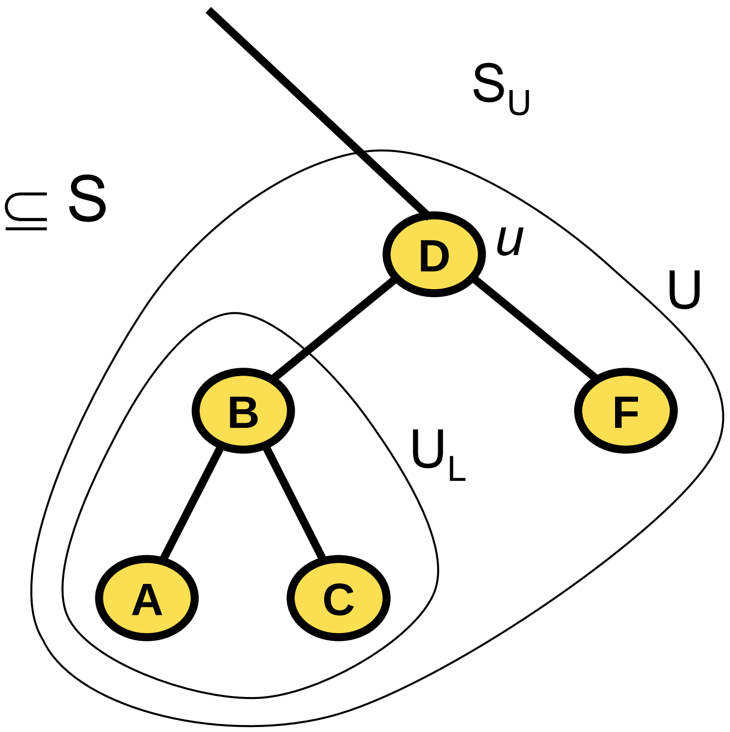
Weight balanced trees

Weight v(*u*) of node *u* in tree *S*

$v\,(u) = 1/2,$ if u is leaf

$v\,(u) = (\ |U_L| + 1)\ /\ (\ |U| + 1\ ),$

if u is the root of sub-tree $S_U \subseteq S$

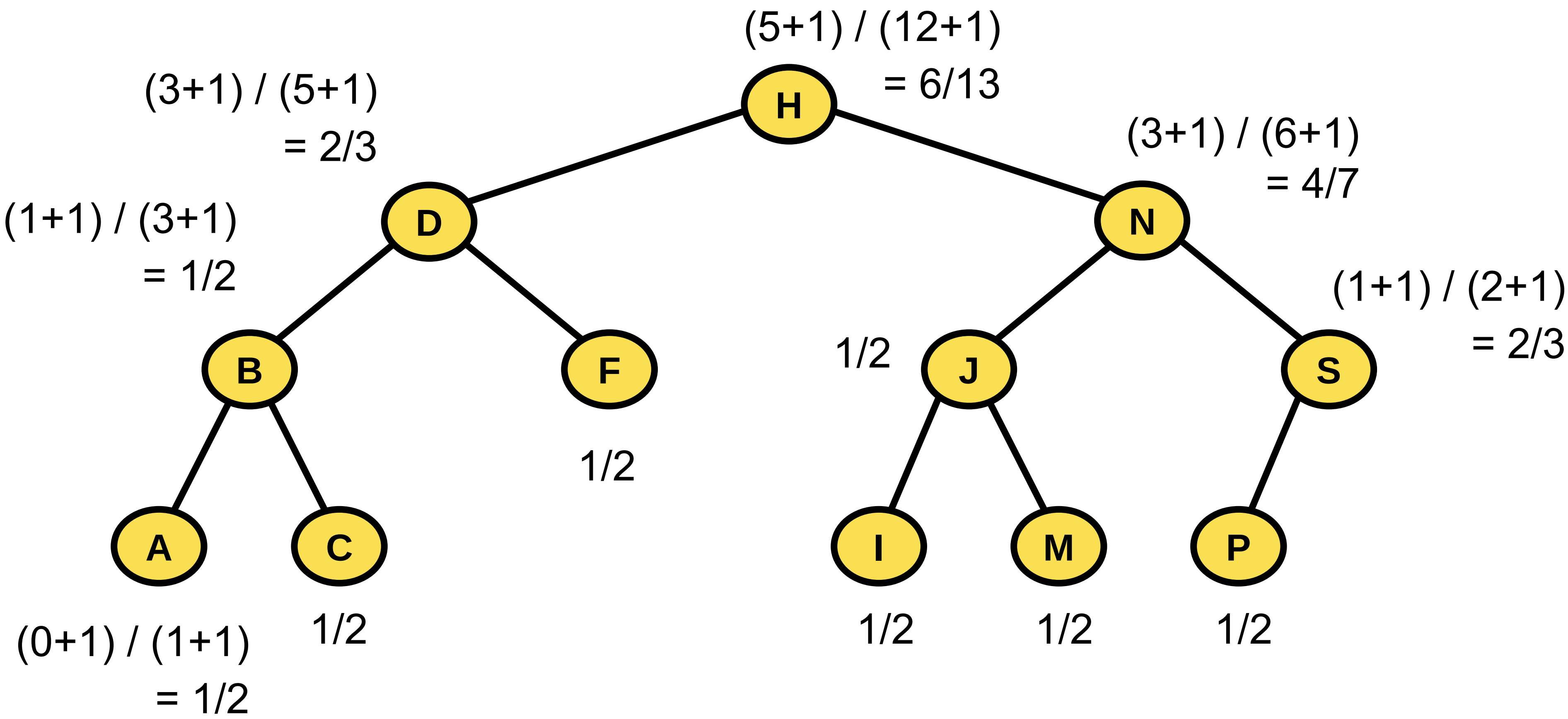


U_L = set of nodes

in the left sub-tree of sub-tree S_U

U = set of nodes in sub-tree S_U

Weight balanced tree example



Váhově vyvážené stromy

Strom s ohraničeným vyvážením α:

Strom S má ohraničené vyvážení α, $0 \le \alpha \le 0{,}5$,
jestliže pro všechny uzly S platí

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

Výška h(S) stromu S s ohraničeným vyvážením α

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

Výška ideálně
vyváženého stromu

[Hudec96], [Mehlhorn84]

Weight balanced trees

Weight balanced tree delimited by α:

Tree S has the balance delimited by α, $0 \leq \alpha \leq 0{,}5$,

if for all nodes S holds

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

Height h(S) of tree S with balance delimited by α:

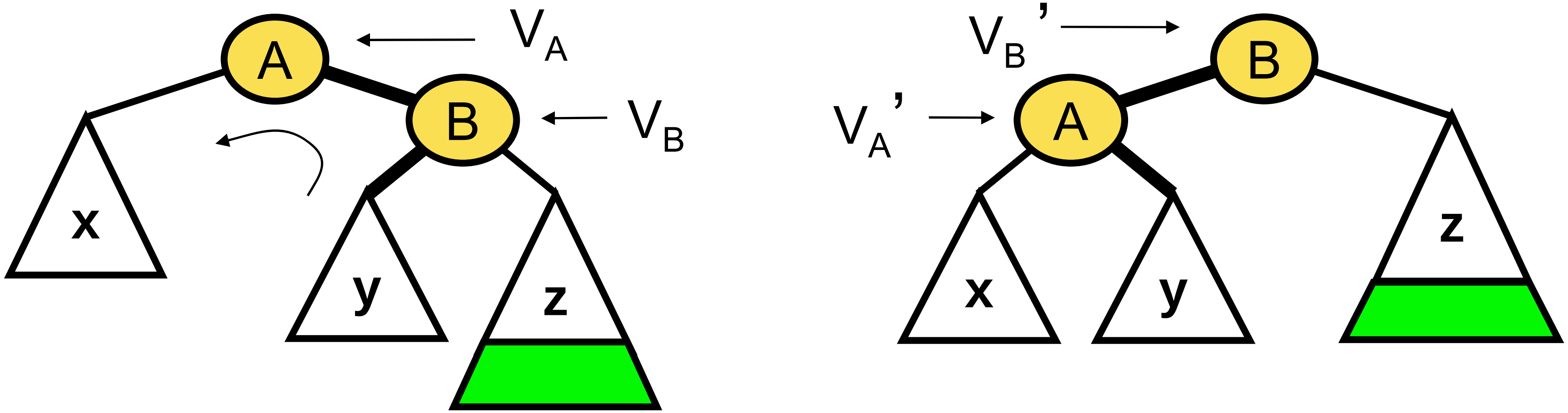
$$h(S) \leq (1 + \log_2(n+1) - 1) / \log_2 (1 / (1- \alpha))$$

balanced tree

height

[Hudec96], [Mehlhorn84]

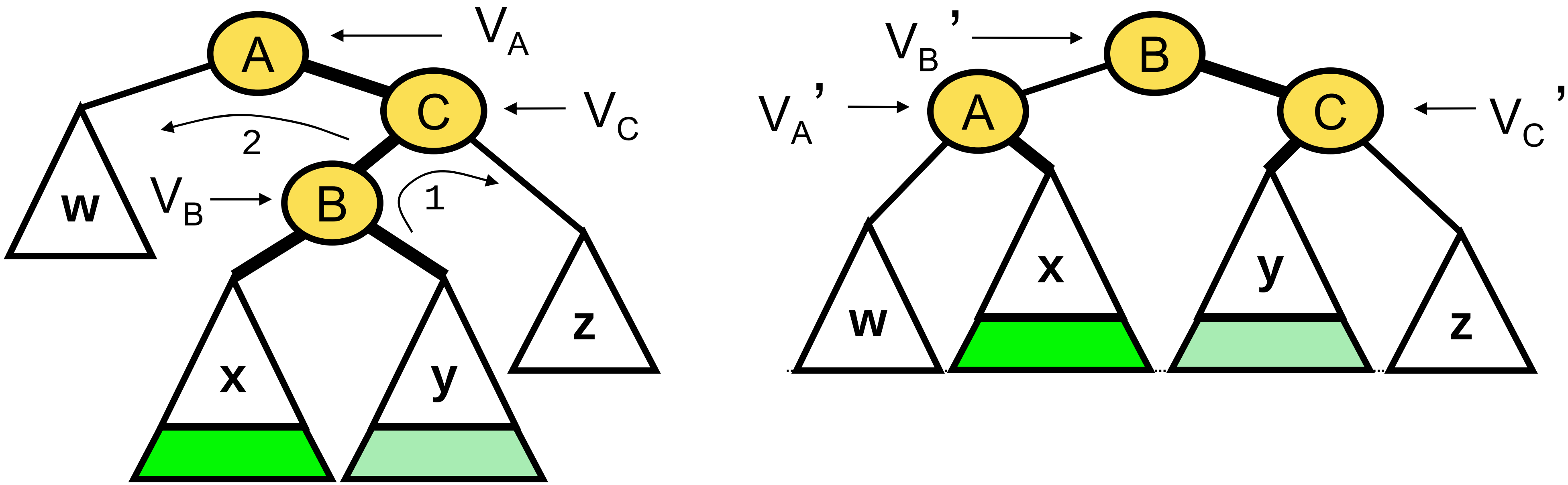
L rotation (Left rotation) [Hudec96]



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

$$V_B' = V_A + (1- V_A) \,.\, V_B$$

RL rotation (Right-Left rotation)



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

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

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

[Hudec96]

Prameny

Bohuslav Hudec: Programovací techniky, skripta, ČVUT Praha, 1993

References

Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*, MIT Press, 1990

AVL tree, http://en.wikipedia.org/w/index.php?title=AVL_tree&oldid=171936487 (last visited Nov. 20, 2007).

Joshua Bloch: *Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken*, http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html