Binární vyhledávací stromy I

DSA - Přednáška 9

Josef Kolář

O čem bude řeč?

- připomeneme si některé operace související s vyhledáváním
- motivace použití vyhledávacích stromů
- operace nad vyhledávacími stromy
  - vyhledání zadané hodnoty
  - minimum a maximum
  - předchůdce a následník
  - vložení a odstranění uzlu
  - a další …
- operační složitost operací nad vyhledávacími stromy

Jaké operace nás zajímají

Připomeňme si operace v ADT Tabulka

```
init:         -> Table
insert(_,_): Elem, Table -> Table
search(_,_): Key, Table -> Elem
delete(_,_): Key, Table -> Table
key(_):       Elem -> Key
```

… a operace v ADT Množina (dynamická, úplně uspořádaná)

```
min(_):       Set -> Elem
max(_):       Set -> Elem
succ(_,_):    Elem, Set -> Elem
pred(_,_):    Elem, Set -> Elem
```

(**Set** je zde i dále možné chápat jako **Table**)

… případně doplněné operacemi

```
sort(_):      Set -> List
select(_,_): Nat, Set -> Elem
join(_,_):    Set, Set -> Set
```

Sekvenční implementace

ADT Tabulka / ADT Množina reprezentovaná polem (spojovým seznamem)

`init, insert, delete, join:`	O(1)
`search, min, max, pred, succ:`	Θ(N)
`sort, select:`	Θ(N . log N)

… reprezentace seřazeným polem (resp. spojovým seznamem)

`init, min, max, pred, succ:`	O(1) (předp. obousměrné spoje)
`search:`	Θ(log N), resp. Θ(N)
`insert, delete:`	Θ(N)
`sort, select:`	O(1), resp. O(N)
`join:`	Θ(M+N)

Můžeme s tím být spokojeni?

Hledání půlením (Binary Search)

Připomínka klasického algoritmu, předpokládá seřazené pole

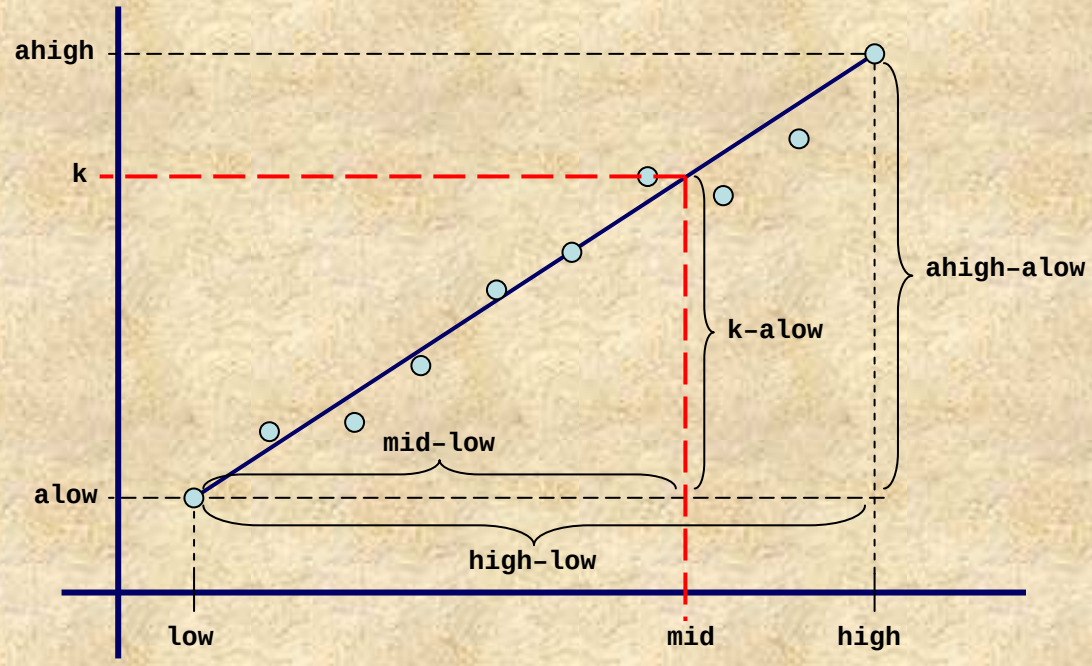
```
Elem searchRec( Elem[] a, int low, int high, Key k) {
  if (low > high) return null;
  int mid = (low+high)/2;
  if (a[mid].key == k) return a[mid];
  if (a[mid].key < k)
       return searchRec( a, mid+1, high, val );
  else return searchRec( a, low, mid-1, val );
}
```

```
Elem searchIter( Elem[] a, int low, int high, Key k) {
  while (high > low) {          // zjednodušené řešení, méně testů
    int mid = (low+high)/2;
    if (a[mid].key < k)
         low = mid+1;
    else high = mid;
  }
  if (a[low].key == k) return a[low];
  else return null;
}
```

Interpolační hledání

Jak hledáme ve slovníku nebo v tlf seznamu? Určitě ne půlením …

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




```
  (k–alow) / (ahigh–alow) =
= (mid-low) / (high-low)
```

```
mid = low + (k–alow)*(high-low)/(ahigh–alow)
```

Interpolační hledání

```
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 searchRec( mid+1, high );
  else return searchRec( low, mid-1 );
}
```

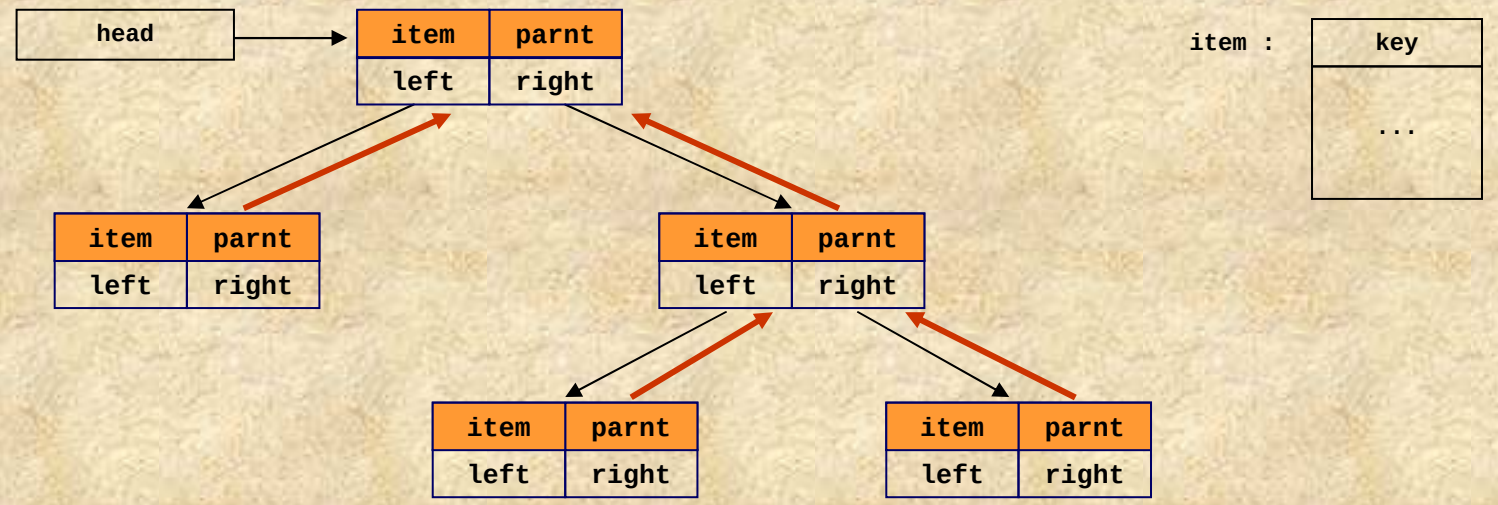
Binární vyhledávací strom (BVS)

BVS – binární strom, který má v každém uzlu hodnotu (klíč) splňující podmínku
- hodnota v uzlu je větší (nebo rovna) všem hodnotám v levém podstromu
- hodnota v uzlu je menší (nebo rovna) všem hodnotám v pravém podstromu

Jak budeme reprezentovat / implementovat BVS?

```
class BVS {
  private class Elem
    { Key key;                        // + další případné složky
    }
  private class Node
    { Elem item;
      Node left, right;
      // Node parent;                 // případný odkaz na rodiče
      Node(Item x) {item = x;}
    }
  private Node head;
  BVS (int maxN)                      // ekvivalent operace init()
    { head = null; }
  void insert (…)
  …
}
```

Binární vyhledávací strom (BVS)



BVS – `insert` a `search` rekurzivně

Jak navrhneme vkládání:

- do prázdného stromu vložíme tak, že vrátíme nově vytvořený uzel
- jinak zjistíme, zda máme vkládat nalevo nebo napravo od kořene
  - rekurzivně vložíme na správnou stranu
  - upravíme příslušný odkaz na levého/pravého potomka

Podobně navrhneme vyhledávání (analogie hledání půlením):

- v prázdném stromu se nic nenajde
- v neprázdném zjistíme, zda
  - je hledaná hodnota v kořeni – hotovo
  - nebo je menší než hodnora v kořeni – rekurze na levý podstrom
  - nebo rekurze na pravý podstrom

BVS – insert a search rekurzivně

```
void insert (Elem x)
  { head = insertRec(head, x); }

private insertRec (Node h, Elem x) {
    if (h == null)
        return new Node(x);
    if (x.key < h.item.key)
         h.left = insertRec(h.left, x
    else h.right = insertRec(h.right,
    return h;
  }

Node search (Key val)
  { return searchRec(head, val); }

private Node searchRec (Node h, Key v
    if (h == null) return null;
    if (val == h.item.key) return h;   // obecnější než return h.item;
    if (val < h.item.key)
         return searchRec(h.left, val);
    else return searchRec(h.right, val);
  }
```

```
Elem searchRec( Elem[] a,
                int low,
                int high,
                Key val) {
  if (low > high) return null;
  int mid = (low+high)/2;
  if (val == a[mid].key) return a[mid];
  if (val < a[mid].key)
       return searchRec(low, mid-1);
  else return searchRec(mid+1, high);
}
```

složitost O(h)

h = výška BVS (log N?)

BVS – insert a search iterativně

```
void insert (Item x)                    // iterative version of insert
  { if (head == null) { head = new Node(x); return; }
    Node p = head; q = p;
    while (q != null) {
      p = q;
      if (x.key < q.item.key) q = q.left;
      else                    q = q.right;
    }
    if (x.key < p.item.key) p.left = new Node(x);
    else                   p.right = new Node(x);
  }

Node search (Key val)                   // iterative version of search
  { Node h = head;
    while(( h != null ) and ( val != h.item.key ))
      if( val < h.item.key ) h = h.left;
      else                   h = h.right;
    if ( h == null ) return null;
    else             return h;
  }
```

Jak by se projevilo zahrnutí odkazu na rodiče???

Jak se projeví existence uzlů se stejným klíčem?

BVS – počet uzlů a řazení

```
int count () { return countRec(head) };

private int countRec (Node h)           // lazy node count
  { if (h == null) return 0;
    return 1 + countRec(h.left) + countRec(h.right);
  }

String sortS () { return sortSRec(head); }

private String sortSRec (Node h)        // inorder tree traversal
  { if  (h == null) return "";
    String s = sortSRec(h.left);
    s += h.item.toString() + "\n";
    s += sortSRec(h.right);
    return s;
  }
```

složitost O(N)

Jak bychom implementovali řazení prvků s výsledkem ve spojovém seznamu?

BVS – řazení do spojového seznamu

Pro jednoduchost použijeme opět uzly typu Node, odkaz na další uzel ve složce `right`

```
Node sort ()
  { Node s = null;
    return sortRec(head, s); }

private Node sortRec (Node h, Node s)   // reversed inorder tree traversal
  { if  (h == null) return s;
    s = sortRec(h.right, s);            // sort the right subtree
    Node x = new Node(h.item);
    x.right = s;                        // insert root item in front
    return sortRec(h.left, x);          // sort the left subtree
  }
```

složitost O(N)

BVS – `select` (výběr k-tého prvku)

Předpokládáme k = 0, 1, ..., N-1 a existenci počítadla uzlů podstromu uloženého v každém uzlu ve složce `cnt`.

Použijeme podobného triku jako QuickSort/select:

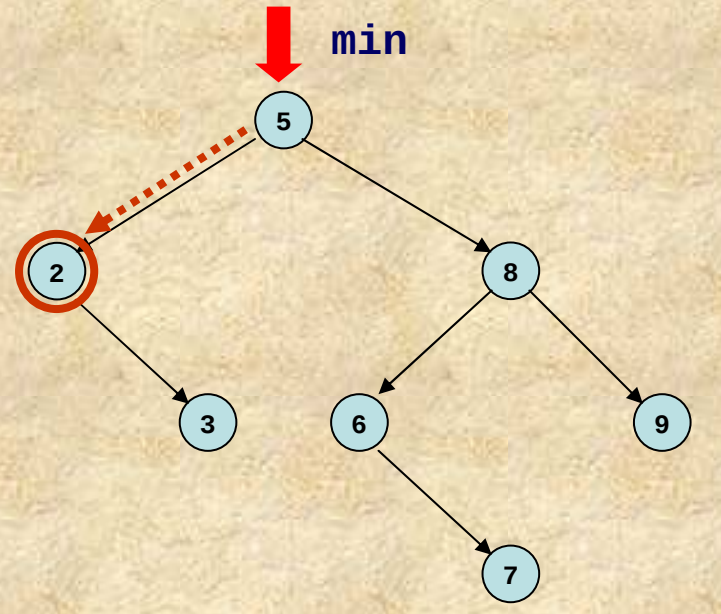
- je-li počet uzlů v levém podstromu větší než k, hledáme rekurzivně v levém
- je-li menší, hledáme v pravém s úpravou pořadového čísla
- jinak jsme nalezli v kořeni

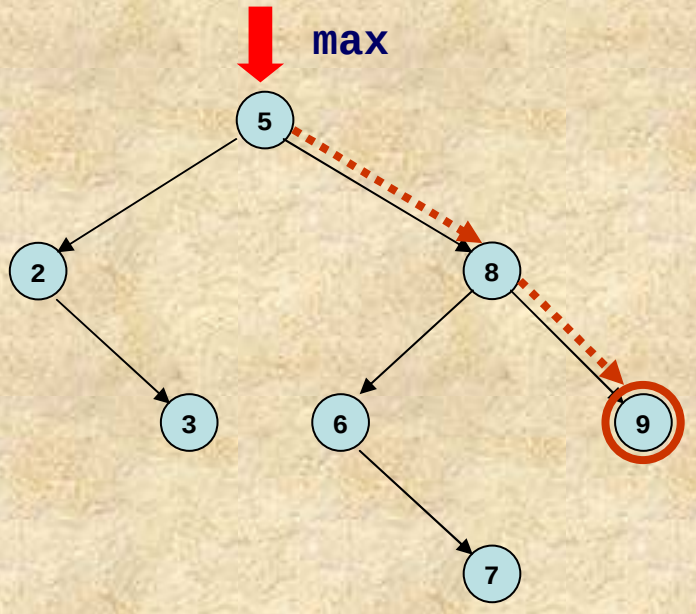
```
Node select (int k)
  { return selectRec(head, k); }

private Node selectRec (Node h, int k)
  { if  (h == null) return null;
    int t = ( h.left == null ) ? 0 : h.left.cnt;
    if ( t > k ) return selectRec(h.left, k);      // go into left subtree
    if ( t < k ) return selectRec(h.right, k-t-1); // into right subtree
    return h;        // least probable alternative last
  }
```

složitost O(h)

h = výška BVS (log N?)

BVS – min, max



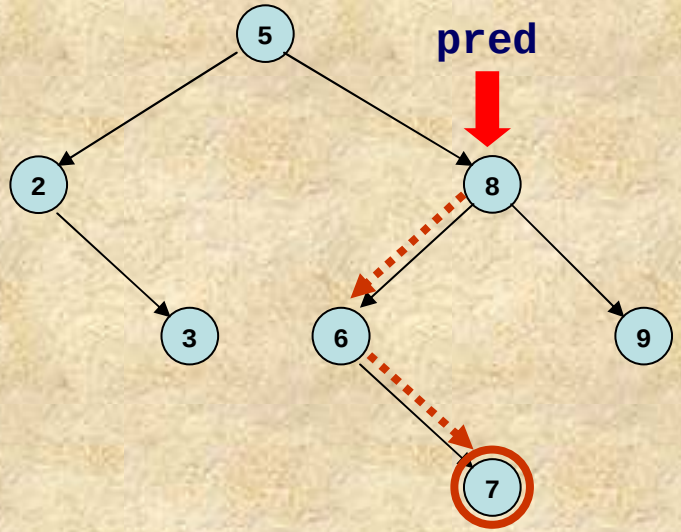


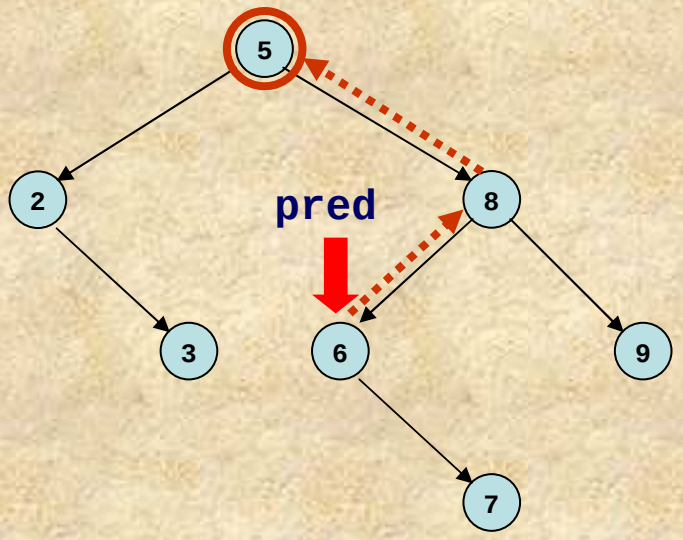

```
Node min (Node x)
  { if (x == null) return null;
    while (x.left != null)
      x = x.left;
    return x;
  }
```

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

složitost ?

BVS – pred, succ



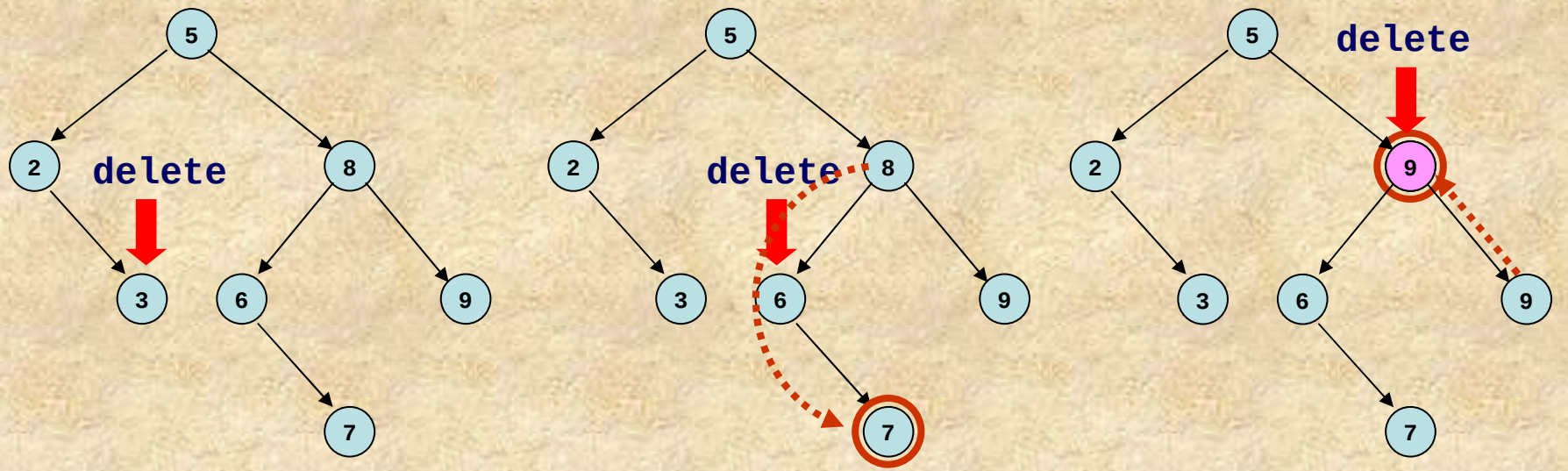


POZOR – pred a succ vyžadují existenci odkazu na rodiče!

```
Node pred (Node x)                                  // we assume x != null
  { if (x.left != null) return max(x.left);         // max of the left subtree
    Node y = x.parent;
    while (( y != null ) && ( x == y.left ))
      { x = y; y = y.parent; }
    return y;
  }

Node succ (Node x)  { ... }                         //  left < -- > right
```

složitost ?

BVS – delete



Vypuštění uzlu rovněž s použitím odkazu na rodiče:

- **je-li to list, přepíše se jen odkaz na něj u jeho rodiče**
- **má-li je jednoho potomka, odkaz na něj u rodiče se přesměruje na jeho potomka**
- **má-li oba potomky, nahradí se jeho hodnota hodnotou succ a vypustí se succ (nebo naopak `pred`)**

BVS – delete

```
Node delete (Node z, Node tree)              // z is a node of tree
  { Node y = z;
    if (( z.left != null ) && ( z.right != null ))
      { y = succ(z); z.item = y.item; }      // z.item replaced by its succ
    if ( y.left != null) x = y.left;
    else                 x = y.right;
    if ( x != null ) x.parent = y.parent;    // adjust parent for x
    if ( y.parent == null )                  // tree root is being deleted
      tree = x;
    else if ( y == y.parent.left )           // y is left son of its parent
              y.parent.left = x;
         else y.parent.right = x;
    return tree;
  }
```

složitost ?

Na efektivní realizaci **delete** (a **pred**, **succ**) jsme potřebovali odkazy na předchůdce.

Uvidíme ale, že to jde i jinak ...

BVS – rotace (jednoduchá)

Rotace zachovává uspořádání v BST

Vliv rotace vpravo na hloubku:	x, α ... -1	β ... 0	y, γ ... +1
Vliv rotace vlevo na hloubku:	x, α ... +1	β ... 0	y, γ ... -1

```
Node rotateRight (Node h) {
  Node p = h.left; h.left = p.right; p.right = h; return p; }

Node rotateLeft (Node h)  {
  Node p = h.right; h.right = p.left; p.left = h; return p; }
```

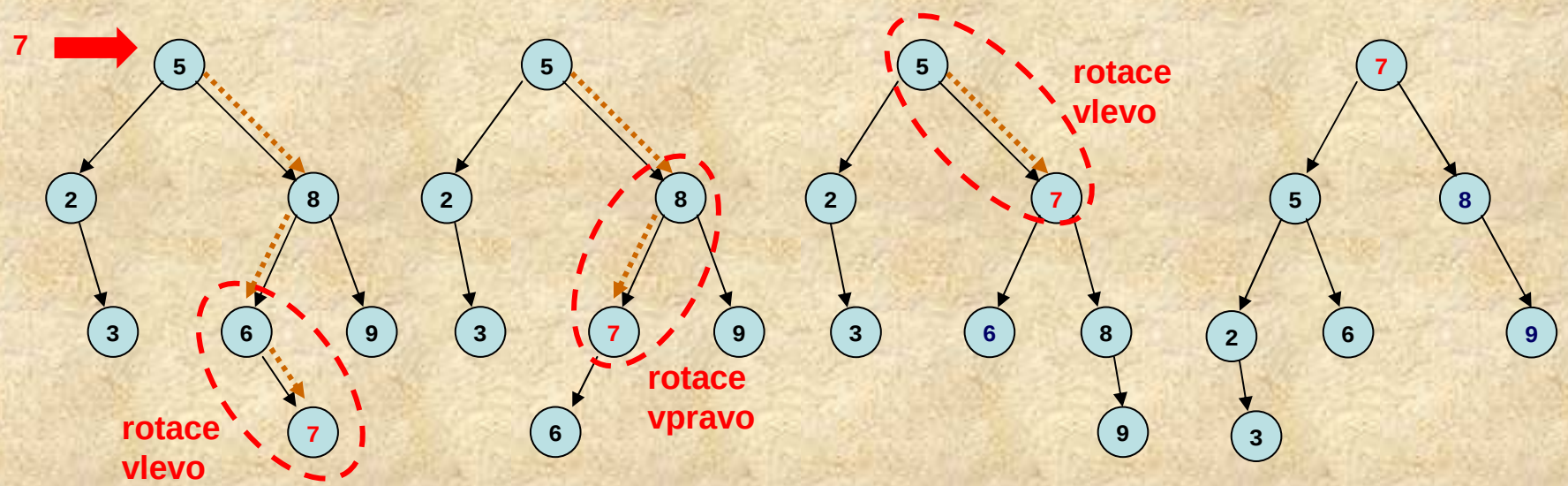
BVS – `insert` do kořene

Jak zařídit, aby se nově vkládaná hodnota x uložila do kořene stromu h?
Bude to zřejmě znamenat změny ve struktuře BVS. Bude to drahé?

Uvažujme následující postup:

- je-li `x.key < h.item.key`, vložíme x do kořene levého podstromu a provedeme rotaci vpravo
- jinak vložíme x do kořene pravého podstromu a provedeme rotaci vlevo

Pokud je strom h prázdný, vytvoříme jenom nový uzel.



BVS – insert do kořene

```
void insert (Elem x)
  { head = insertRoot(head, x); }

private Node insertRoot (Node h, Elem x)
  { if ( h == null ) return new Node(x);
    if ( x.key < h.item.key )
         { h.left  = insertRoot(h.left, x);  h = rotateRight(h); }
    else { h.right = insertRoot(h.right, x); h = rotateLeft(h); }
    return h;
  }
```

složitost O(h)

h = výška BVS (log N?)

Uvidíme, že rotace se nám budou hodit na řadu dalších operací.

Všímáme si, že jedna rotace má konstantní složitost!

BVS – part (rozdělení, k-tý prvek v kořeni)

Opět předpokládáme k = 0, 1, ..., N-1 a počítadla uzlů podstromu uloženého v každém uzlu ve složce cnt jako u operace select.

Uvažujeme rekurzivně:

- je-li k-tý prvek v **levém** podstromu, provedeme jeho rozdělení, aby se dostal do kořene a rotujeme **vpravo**
- je-li v **pravém** podstromu, snížíme k o příslušný počet, provedeme rozdělení pravého podstromu, aby se dostal do kořene a rotujeme **vlevo**
- pokud jsme už prvek našli v kořeni, jen vracíme daný podstrom.

```
Node partRec (Node h, int k)
  { int t = ( h.left == null ) ? 0 : h.left.cnt
    if ( t > k )
       { partRec(h.left, k); h = rotateRight(h);
    if ( t < k )
       { partRec(h.right, k-t-1); h = rotateLeft(h);  // part right
  subtree
    return h;        // (t == k)  => k-th element is already in the root
  }
```

složitost O(h)

h = výška BVS (log N?)

BVS – remove (key s použitím rotací)

Uvažujeme rekurzivně:

- je-li vypouštěný prvek v kořeni, slepíme pomocí **joinLR** jeho levý a pravý podstrom (vlevo jsou menší hodnoty, vpravo větší!!)
- je-li vypouštěný prvek v **levém** podstromu, vypustíme jej z něj a výsledek pověsíme vlevo
- je-li vypouštěný prvek v **pravém** podstromu, vypustíme jej z něj a výsledek pověsíme vpravo

```
private Node joinLR (Node a, Node b)  // a is all less than b !!!
  { if ( b == null ) return a;        // simple case
    b = partRec(b, 0);                // smallest of b is in the root
    b.left = a;                       // b had empty left subtree
    return b;
  }
```

BVS – remove (podle klíče s použitím rotací)

```
void remove (Key val)
  { head = removeRec(head, val); }

private Node removeRec (Node h, Key val)
  { if ( h == null ) return null;        // nothing to remove from
    if ( val < h.item.key )      h.left  = removeRec(h.left, val);
    else if ( val > h.item.key ) h.right = removeRec(h.right, val);
    else                         h = joinRL(h.right, h.right);
    updateCnt(h);
    return h;
  }
```

Jak bychom udržovali složky cnt (pro part a select) při remove?

- asi by se prodražilo přepočítávat **cnt** pro celý strom
- udělat malou utilitu na úpravu a použít ji jen pro uzly, kde se měnil levý / pravý podstrom

```
void updateCnt (Node h)
  { int c = 1;
    if (h.left  != null) c += h.left.cnt;
    if (h.right != null) c += h.right.cnt;
    h.cn = c;
  }
```

BVS – join (key s použitím rotací)

Jak implementovat spojení dvou BVS?

- **opakovaně vkládat uzly / prvky jednoho stromu do druhého ... ? O(M . log N) ?**
- **bohužel vyjde draho opakované opravování složek cnt (pokud je používáme**
- **použijeme uvedenou utilitu `updateCnt`**

```
void join (Node b)
  { head = joinRec(head, b); }

private Node joinRec (Node a, Node b)
  { if ( b == null ) return a;
    if ( a == null ) return b;
    b = insertRoot(b, a.item);
    b.left  = joinRec(a.left, b.left);
    b.right = joinRec(a.right, b.right);
    updateCnt(b);
    return b;
  }
```

Prameny

- **Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 1990**
- **Sedgewick, R.: Algorithms in Java (Parts 1 – 4: Fundamentals, Data Structures, Sorting, Searching).Third edition, Addison Wesley / Pearson Education, Boston, 2003**
- Bohuslav Hudec: Programovací techniky, skripta, ČVUT Praha, 1993
- Miroslav Beneš: Abstraktní datové typy, Katedra informatiky FEI VŠB-TU Ostrava, http://www.cs.vsb.cz/benes/vyuka/upr/texty/adt/index.html