ALG 04

VYHLEDÁVÁNÍ (jednorozměrné vyhledávání)

Vyhledávání v poli

naivní, binární, interpolační

Binární vyhledávací strom (BVS)

operace Find, Insert, Delete

Hledání v seřazeném poli — lineární, POMALÉ

Dané pole

Seřazené pole: →

Velikost = N

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

najdi 993 !

Testů: N

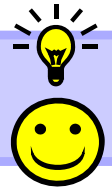
363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

najdi 363 !

Testů: 1

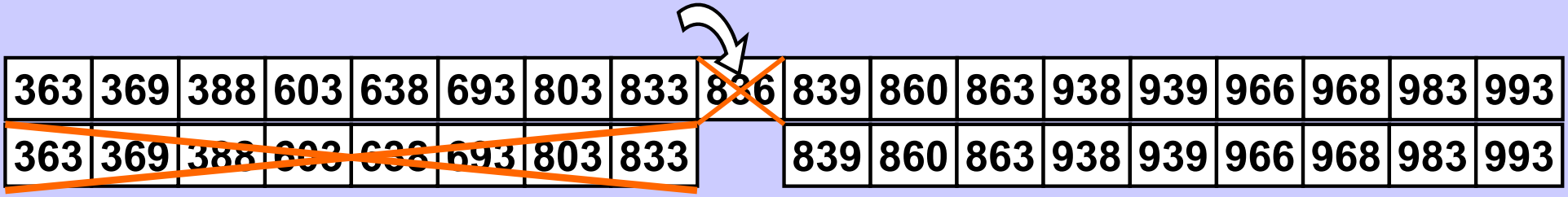
363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Hledání v seřazeném poli — binární, RYCHLEJŠÍ

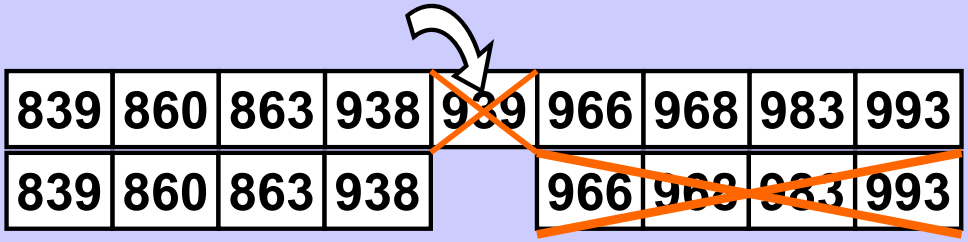


najdi 863 !

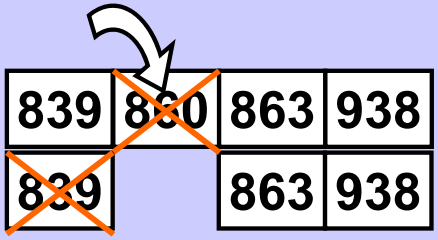
2 testy



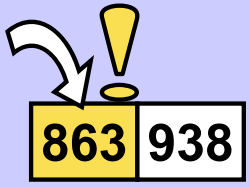
2 testy



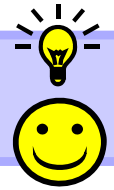
2 testy



1 test

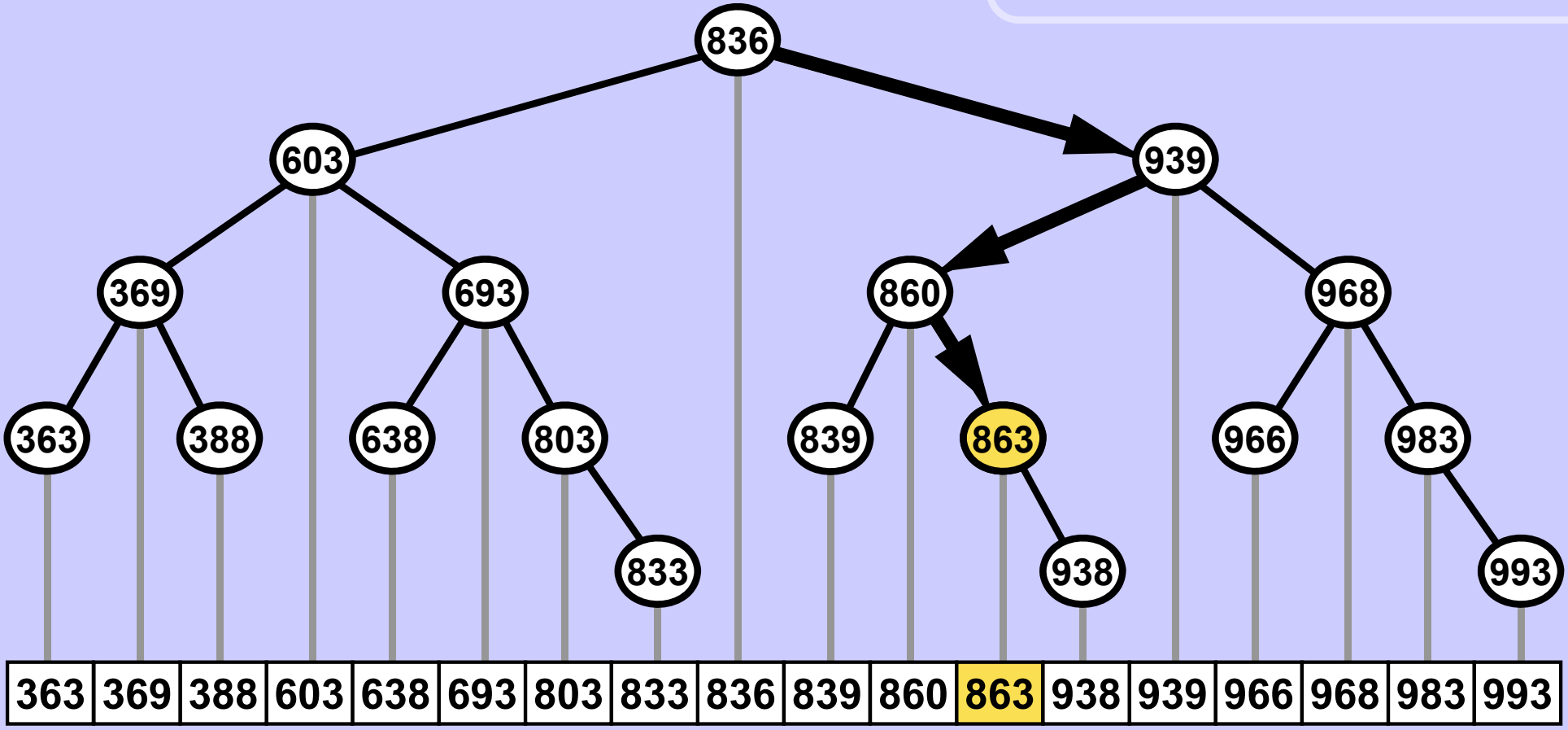


Hledání v seřazeném poli — binární, RYCHLEJŠÍ



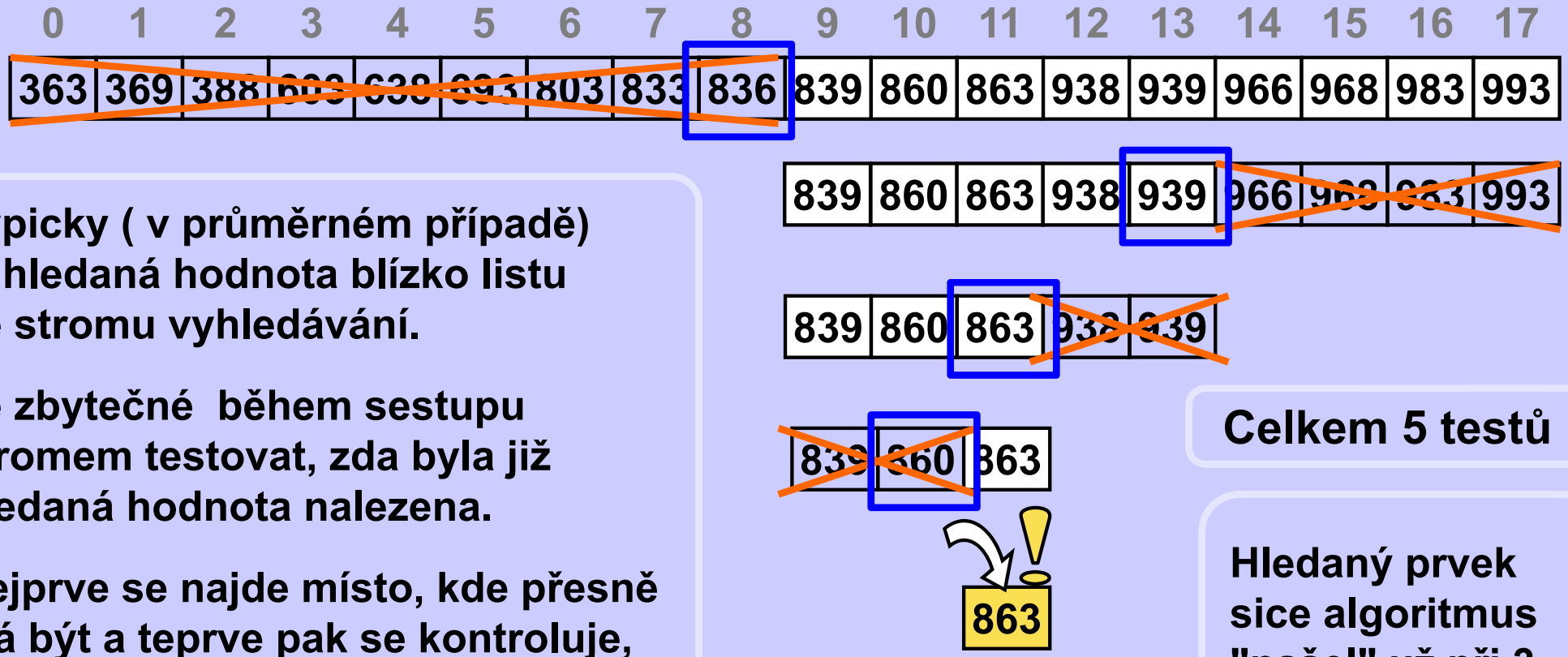
najdi 863 !

Hledání kopíruje strukturu binárního stromu



Hledání v seřazeném poli — binární, JEŠTĚ RYCHLEJŠÍ

najdi 863 !



Typicky (v průměrném případě) je hledaná hodnota blízko listu ve stromu vyhledávání.

Je zbytečné během sestupu stromem testovat, zda byla již hledaná hodnota nalezena.

Nejprve se najde místo, kde přesně má být a teprve pak se kontroluje, zda tam opravdu je.

Celkem 5 testů

Hledaný prvek sice algoritmus "našel" už při 3. testu, ale jeho výskyt v poli potvrdil až později.

Prohledávaný úsek se vždy pouze půlí na levou část s hodnotami menšími nebo rovnými q a na pravou část s hodnotami většími než q.

Binární hledání -- rychlá varianta

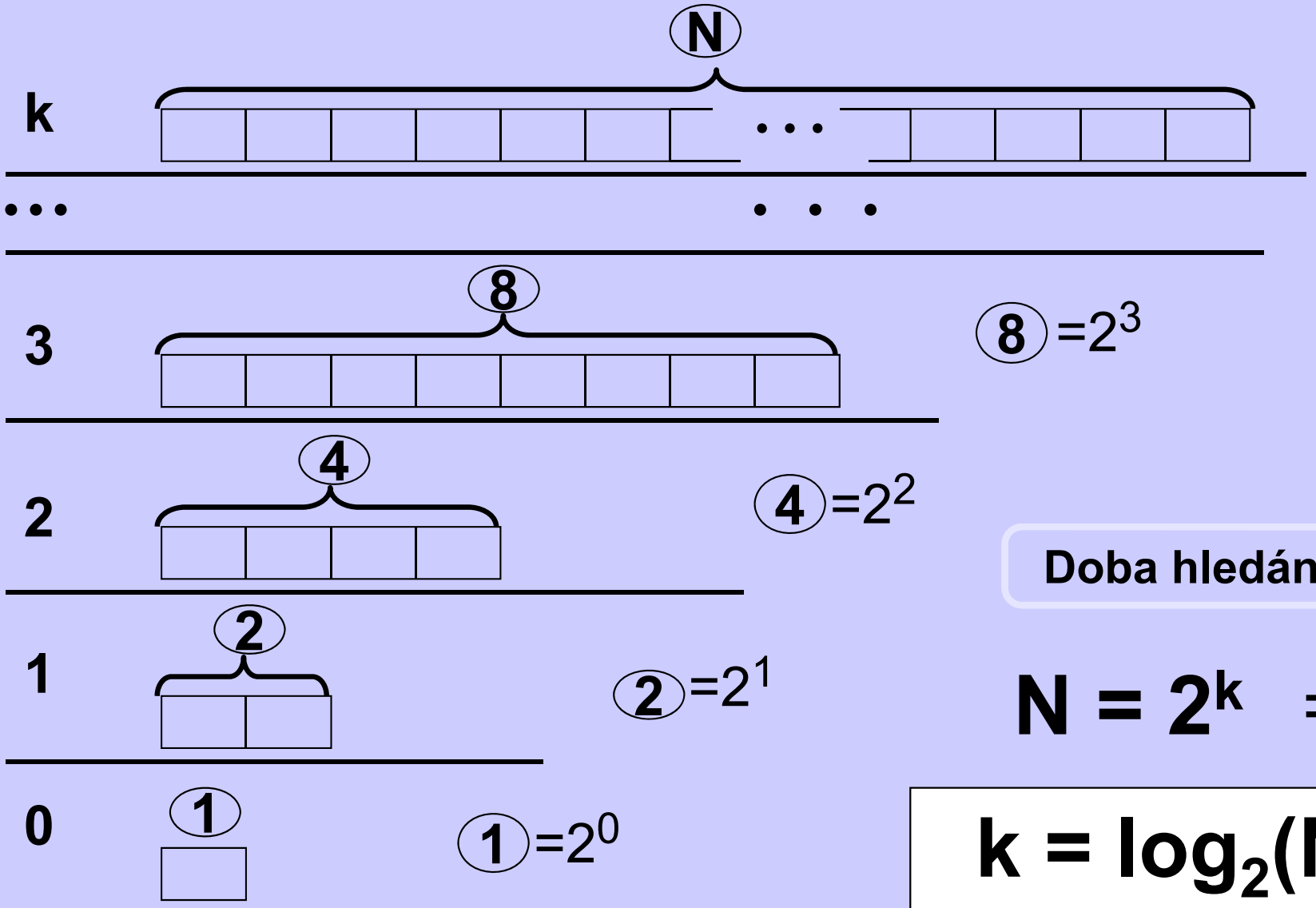
Nejprve je s ohledem na uspořádání hodnot nalezen přesně index, na kterém se má hledaná hodnota vyskytovat. Teprve nakonec se kontroluje, zda na určeném indexu tato hodota opravdu je.
V každé úrovni stromu se tak ušetří jeden test hledané hodnoty.

```
int binSearch( int [] arr, int q ) {
  int low = 0, high = length(arr)-1, mid;

  while( low < high ) {
    mid = (low+high)/2 ;          // bug ?
                                  // fix: mid = low + (high-low)/2;
    if( x > arr[mid] ) low  = mid+1;
    else               high = mid;
  }
  if( arr[low] == x ) return low;
  else return -1;
}
```

Bug? : Pro low + high > max int nastává chyba přetečení
https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

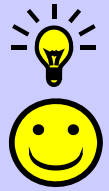
Hledání v seřazeném poli — binární, RYCHLEJŠÍ



Doba hledání

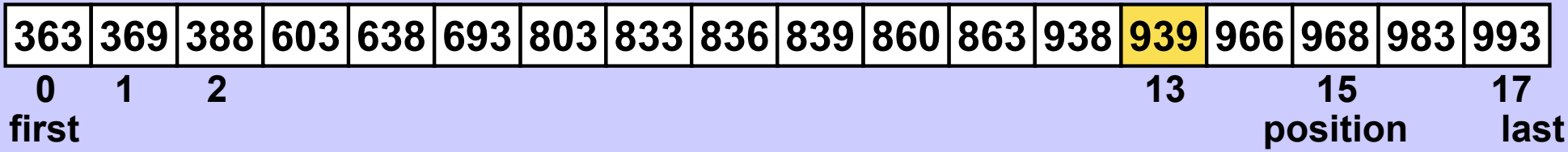
$$N = 2^k \Rightarrow$$

$$k = \log_2(N)$$



Interpolační hledání

Pole a[ ] **Najdi q = 939**



Mají-li hodnoty v poli víceméně rovnoměrně náhodné rozložení, je možno použít lineární interpolaci.
Poloha prvku v poli by měla přibližně odpovídat jeho velikosti.

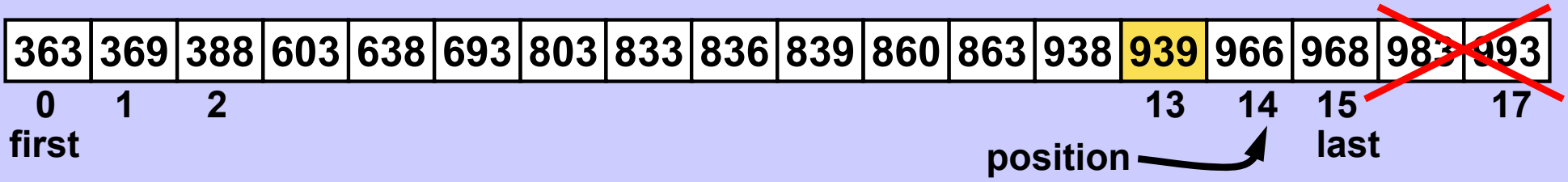
$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

Celá část!

$$position = 0 + \frac{939 - 363}{993 - 363} * (17 - 0) = 15.54$$

Interpolační hledání

Pole a[ ] **Najdi q = 939**



Když se na vypočtené pozici prvek nenalézá, je buď vlevo nebo vpravo od ní a pak lze (rekurzivně) vzít za výchozí interval příslušnou levou nebo pravou část pole a výpočet opakovat.

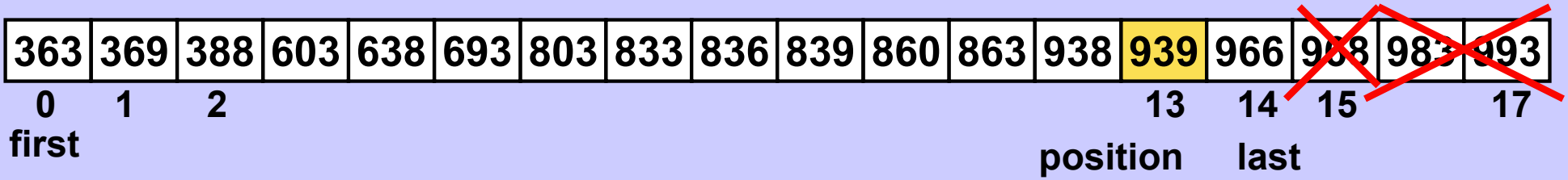
$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

Celá část!

$$position = 0 + \frac{939 - 363}{968 - 363} * (15 - 0) = 14.12$$

Interpolační hledání

Pole a[ ] **Najdi q = 939**



Když se na vypočtené pozici prvek nenalézá, je buď vlevo nebo vpravo od ní a pak lze (rekurzivně) vzít za výchozí interval příslušnou levou nebo pravou část pole a výpočet opakovat.

$$position = first + \frac{(q - a[first])}{a[last]-a[first]} * (last - first)$$

Celá část!

$$position = 0 + \frac{939 - 363}{966 - 363} * (14 - 0) = 13.37$$

Hotovo

Interpolační hledání

```
int interpol( int [] arr, int q ) {
  int first = 0;
  int last = length(arr)-1;
  do{
     pos = first + round( (q-arr[first])/(arr[last]-arr[first])
                          *(last-first) );
     if( arr[pos] < q )      first = pos+1;  // check left side
     else if( arr[pos] > q ) last = pos-1;  //check right side
  }while( (arr[pos] != q) && (first < last) );

  if( arr[pos] == q ) return pos;
  else return -1;  // not found
}
```

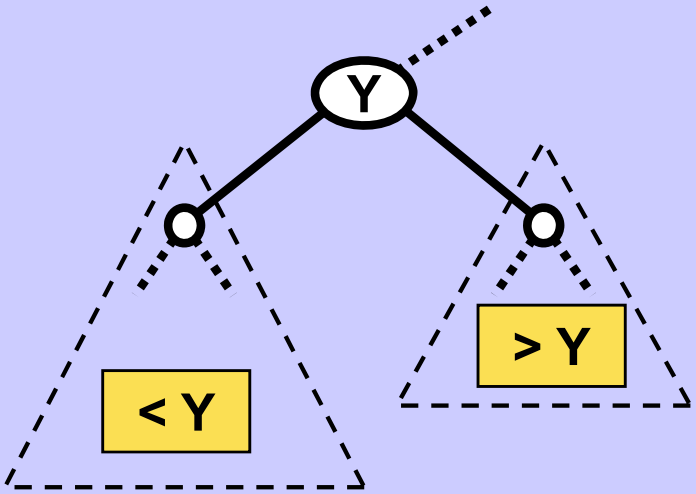
Hledání v seřazeném poli — srovnání rychlostí

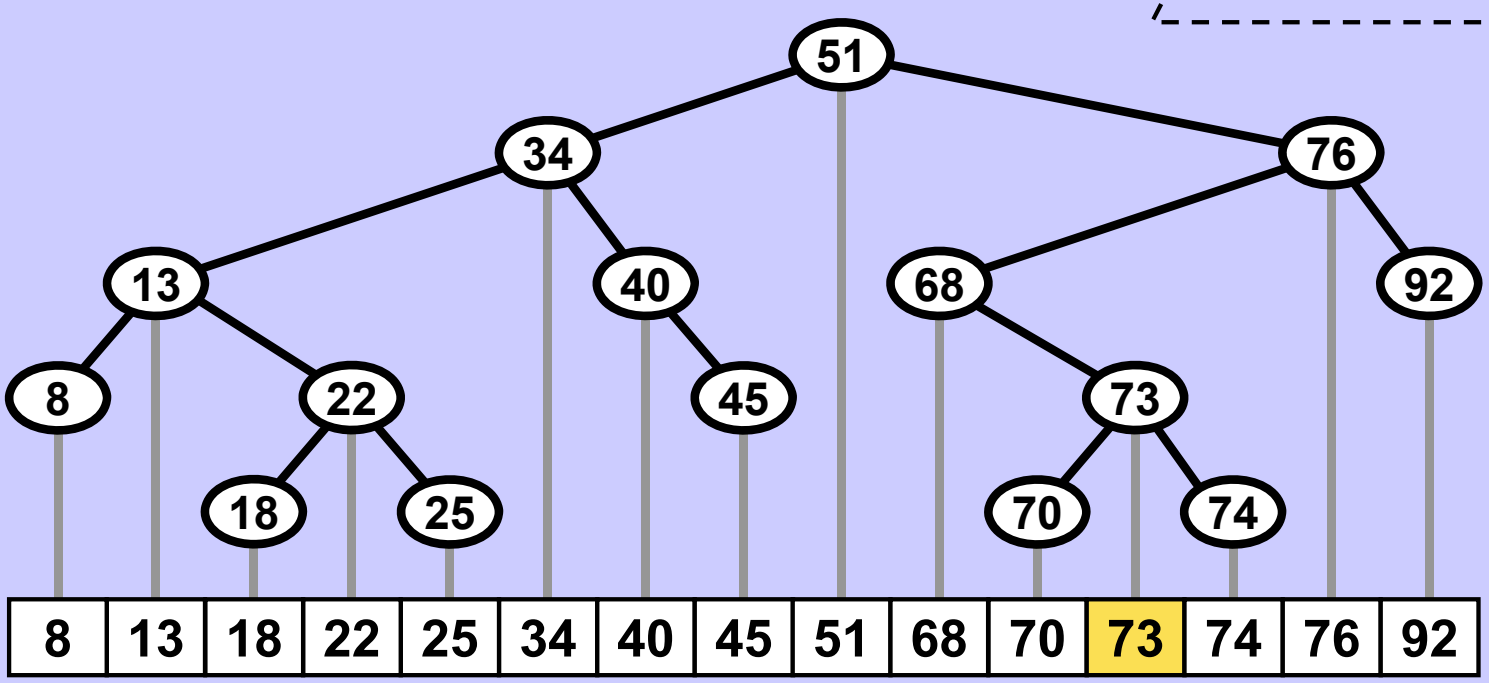
Velikost pole N	Lineární hledání průměrný případ	Interpolační hledání průměrný případ	Binární hledání pomalé nejhorší případ	Binární hledání rychlé pokaždé
10	5.5	1.60	9	4
30	15.5	2.12	11	5
100	50.5	2.56	15	7
300	150.5	2.89	19	9
1 000	500.5	3.18	21	10
3 000	1 500.5	3.41	25	12
10 000	5 000.5	3.63	29	14
30 000	15 000.5	3.80	31	15
100 000	50 000.5	3.96	35	17
300 000	150 000.5	4.11	39	19
1 000 000	500 000.5	4.24	41	20
	Zřejmě $\Theta(n)$	Na náhodném rovnoměrném rozdělení, teoreticky $\Theta(\log(\log(N)))$	Podle struktury binárního stromu $\Theta(\log(n))$	

Binární vyhledávací strom

V levém podstromu každého uzlu jsou všechny klíče menší.

V pravém podstromu každého uzlu jsou všechny klíče větší.



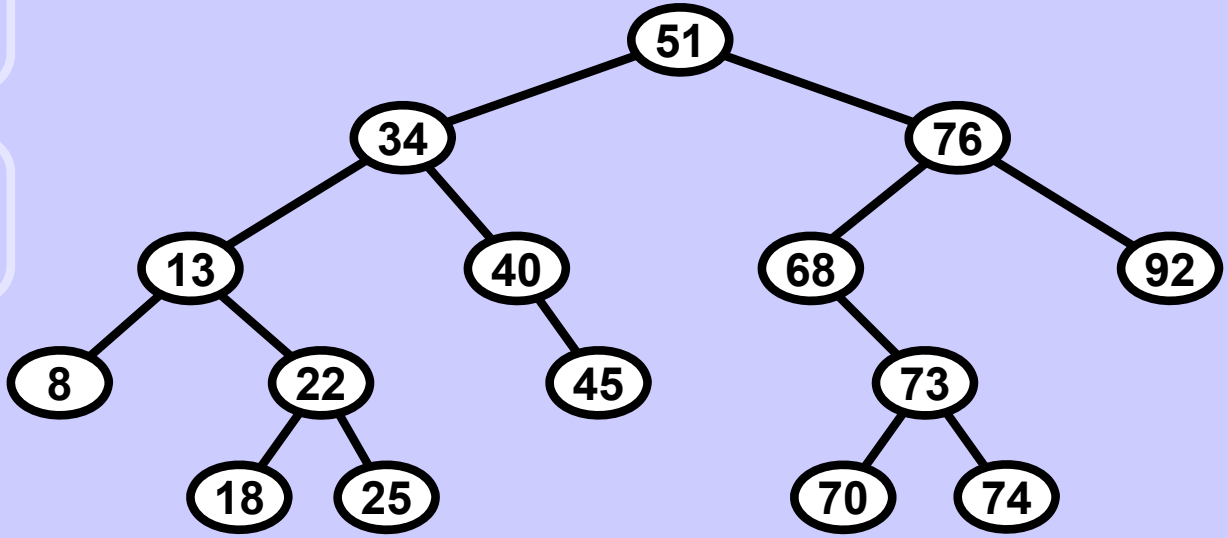


Binární vyhledávací strom

BVS nemusí být
a nebývá vyvážený.

BVS nemusí být
a nebývá pravidelný.

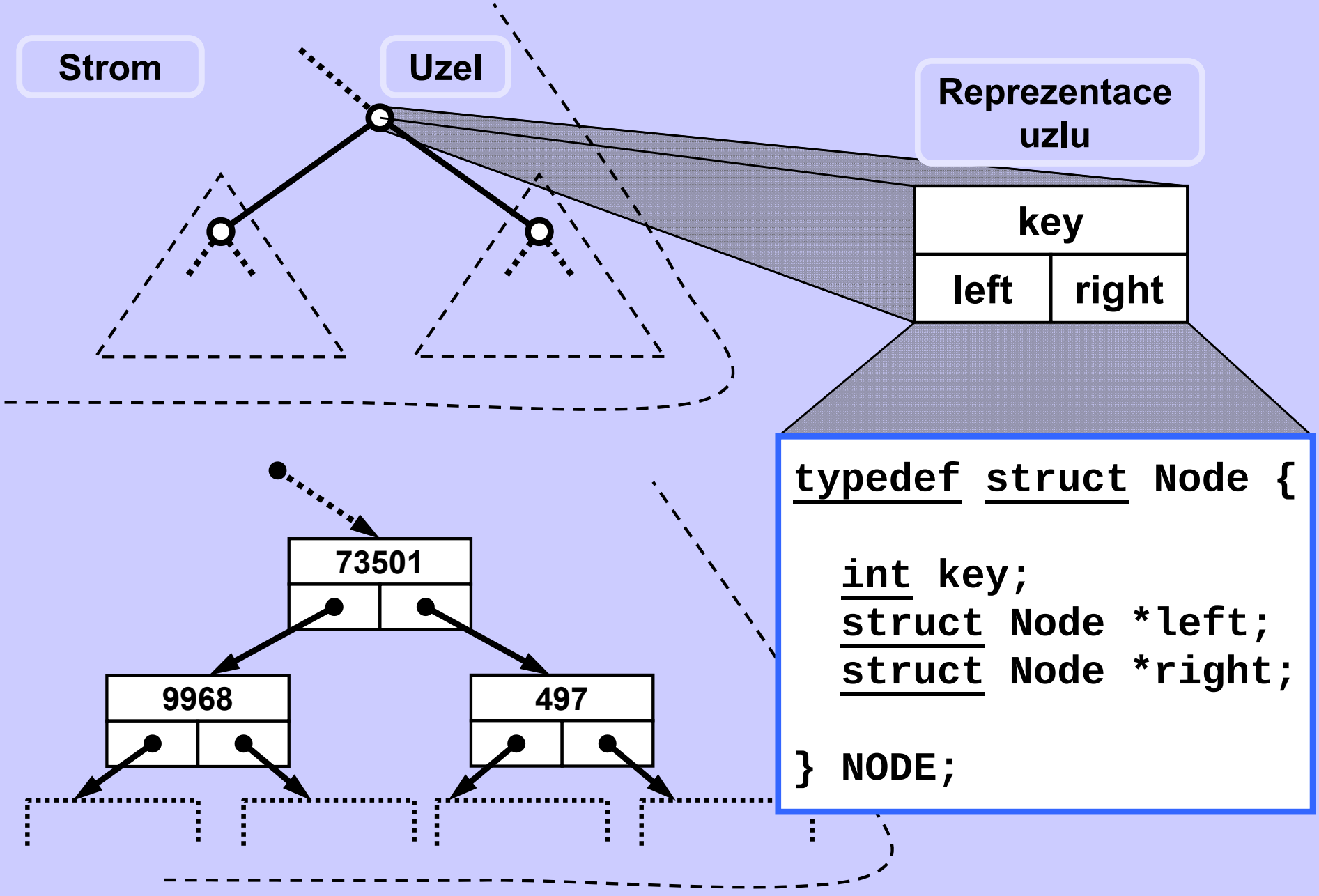
Výpisem prvků BVS
v pořadí INORDER
získáme uspořádané
hodnoty klíčů.



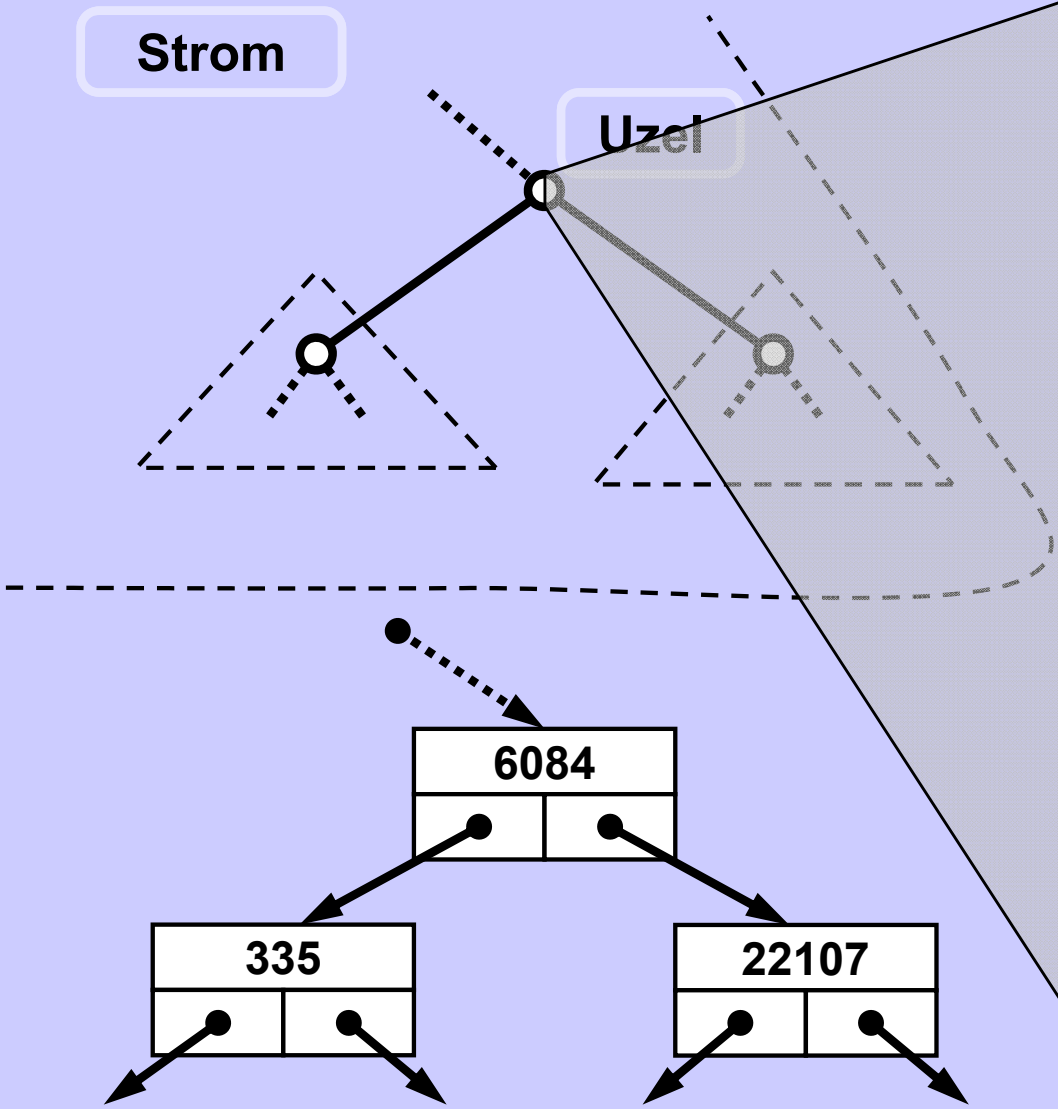
BVS je flexibilní díky operacím:

Find – najdi prvek s daným klíčem
Insert – vlož prvek s daným klíčem
Delete – (najdi a) odstraň prvek s daným klíčem

Implementace binárního stromu -- C



Implementace binárního stromu -- java

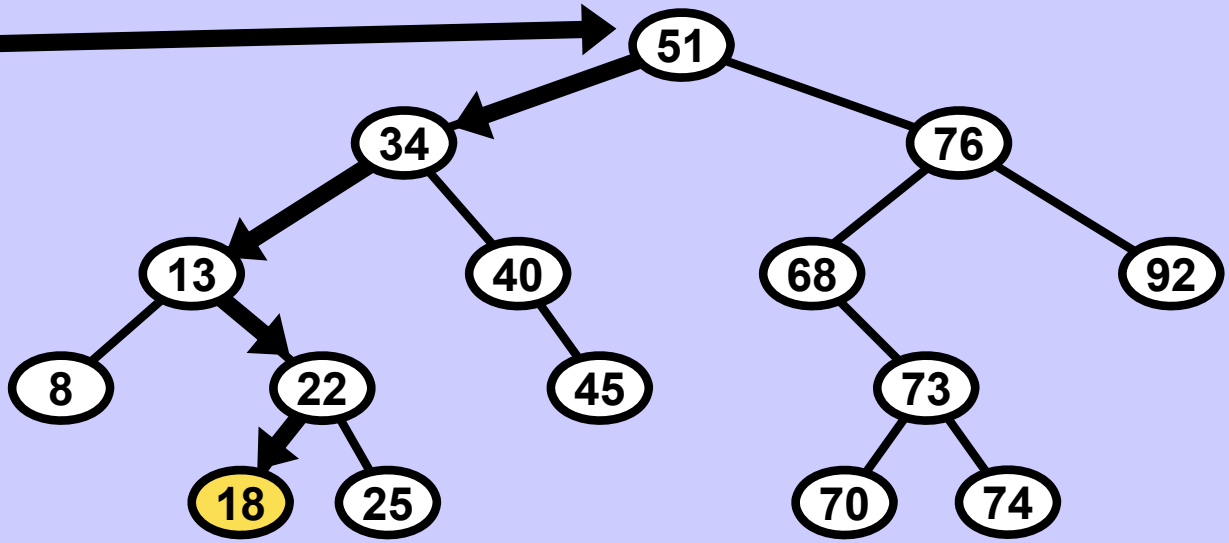



```
public class Node {
   public Node left;
   public Node right;
   public int key;
   public Node(int k) {
      key = k;
      left = null;
      right = null;
   }
}

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

Operace Find v BVS

Najdi 18



Každá operace se stromem začíná v kořeni.

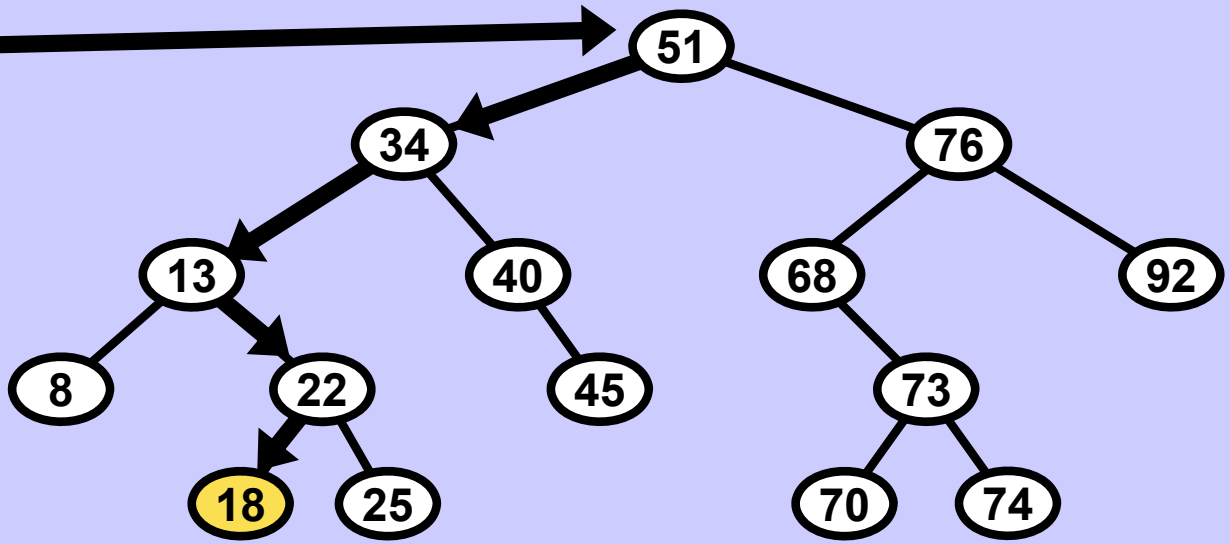
Iterativně

```
Node find( int k, Node node ){
  while( true ) {
     if( node == null )   return null;
     if( node->key == k ) return node;
     if( k < node->key )  node = node->left;
        else              node = node->right;
} }
```

Operace Find v BVS

Najdi 18

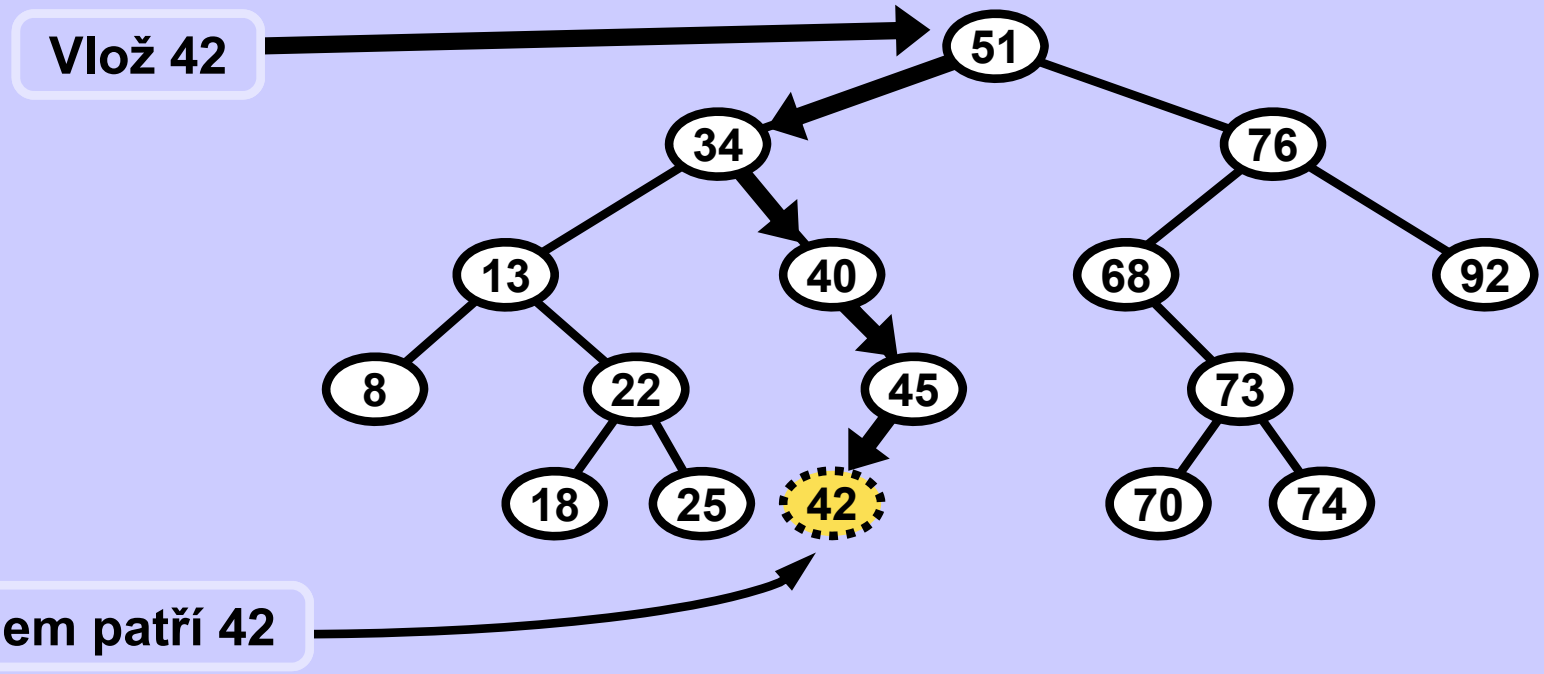
Každá operace se stromem začíná v kořeni.



Rekurzivně

```
Node findRec( int k, Node node ){
  if( node == null )  return null;
  if( node.key == k ) return node;
  if( k < node->key ) return findRec( k, node->left );
     else             return findRec( k, node->right );
} }
```

Operace Insert v BVS



Insert

1. Najdi místo (jako ve Find) pro list, kam patří uzel s daným klíčem.
2. Vytvoř tento uzel a vlož jej do stromu.

Operace Insert v BVS iterativně

```
Node insert( int k, Node node ){
  if( node == null ){                    // empty tree
    Node newNode = ...;                  // create node with key k
    return newNode;
  }
  while( true )
    if( node->key == k ) return null; //can't insert a duplicate
    if( node->key > k )
       if( node->left == null ){
          Node newNode = ...;            // create node with key k
          node->left = newNode;
          return newNode; }
       else node = node->left;
    else                                 // similarly to the right
       if( node->right == null ){
          Node newNode = ...;
          node->right = newNode;
          return newNode; }
       else node = node->right; }
} }
```

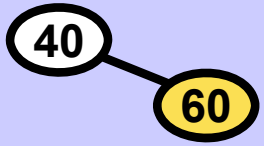
Operace Insert v BVS rekurzivně

```
Node insertRec( int k, Node node, Node parentNode ){
  if( node == null ){                    // empty tree
    Node newNode = ...;                  // create node with key k
    if(parentNode != null ){
       if(parentNode->key > k)
          parentNode->left = newNode;
       else
          parentNode->right = newNode;
    return newNode;
  }
  if( node->key == k ) return null;      //can't insert a duplicate
  if( node->key > k )                    // chose direction
     return insertRec( k, node->left, node );
  else
     return insertRec( k, node->right, node );
}
```

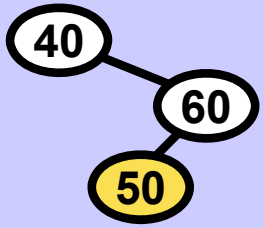
Stavba BVS operací Insert

insert 40

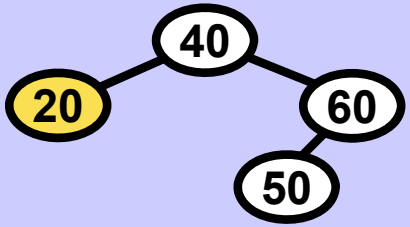
insert 60



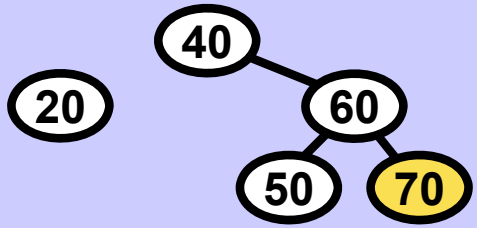
insert 50



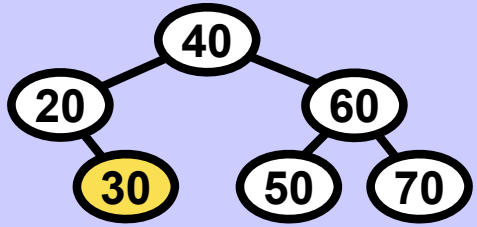
insert 20



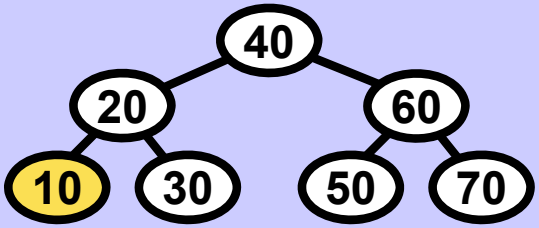
insert 70



insert 30



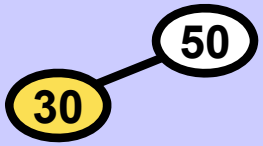
insert 10



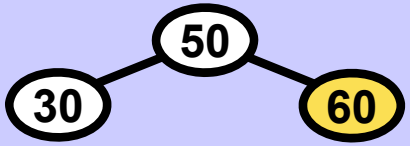
Tvar budovaného BVS závisí na pořadí vkládání dat.

insert 50

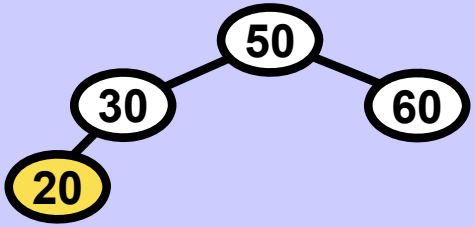
insert 30



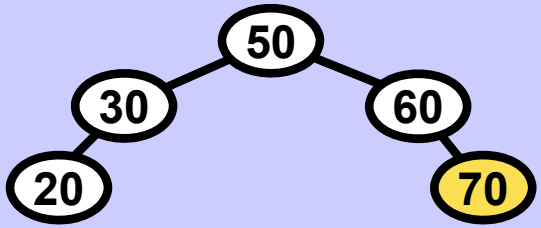
insert 60



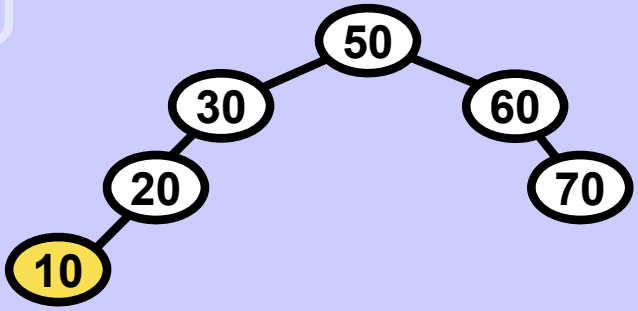
insert 20



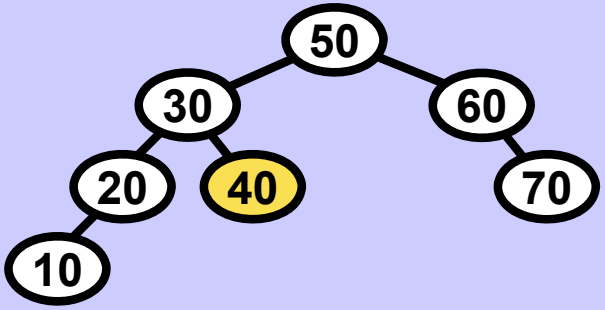
insert 70



insert 10

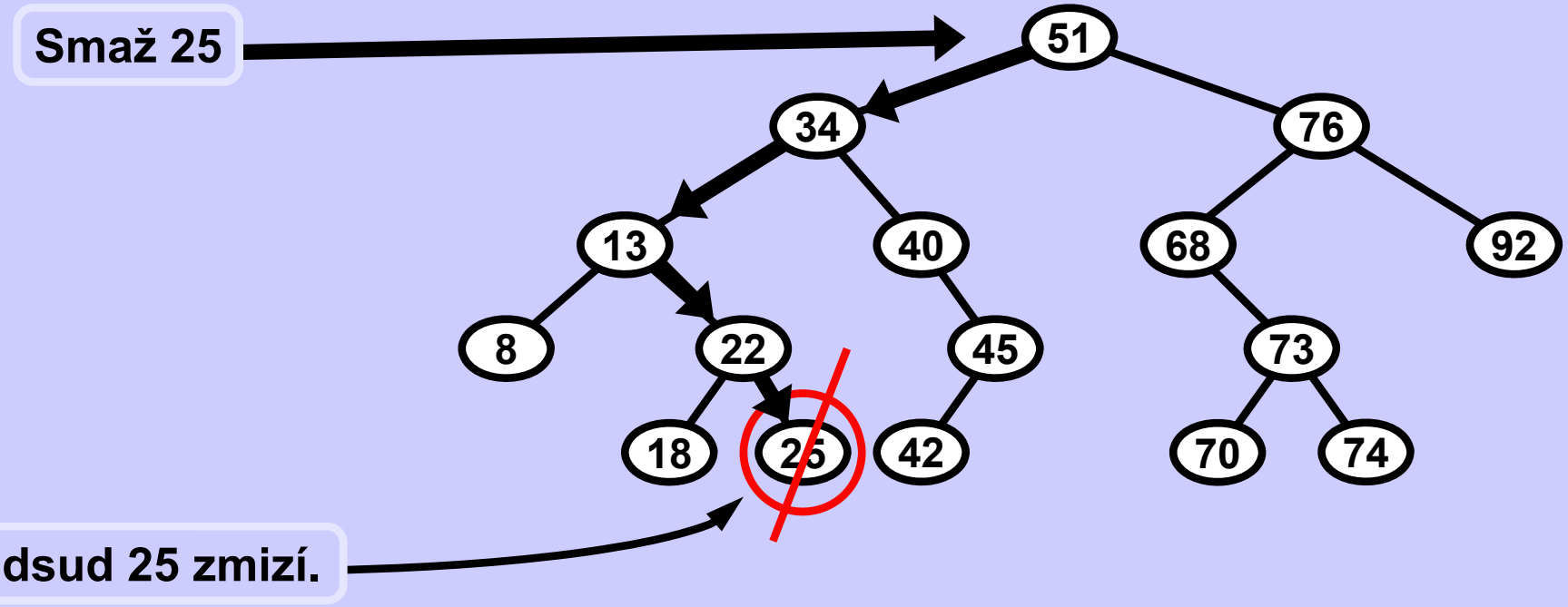


insert 40



Operace Delete v BVS (I.)

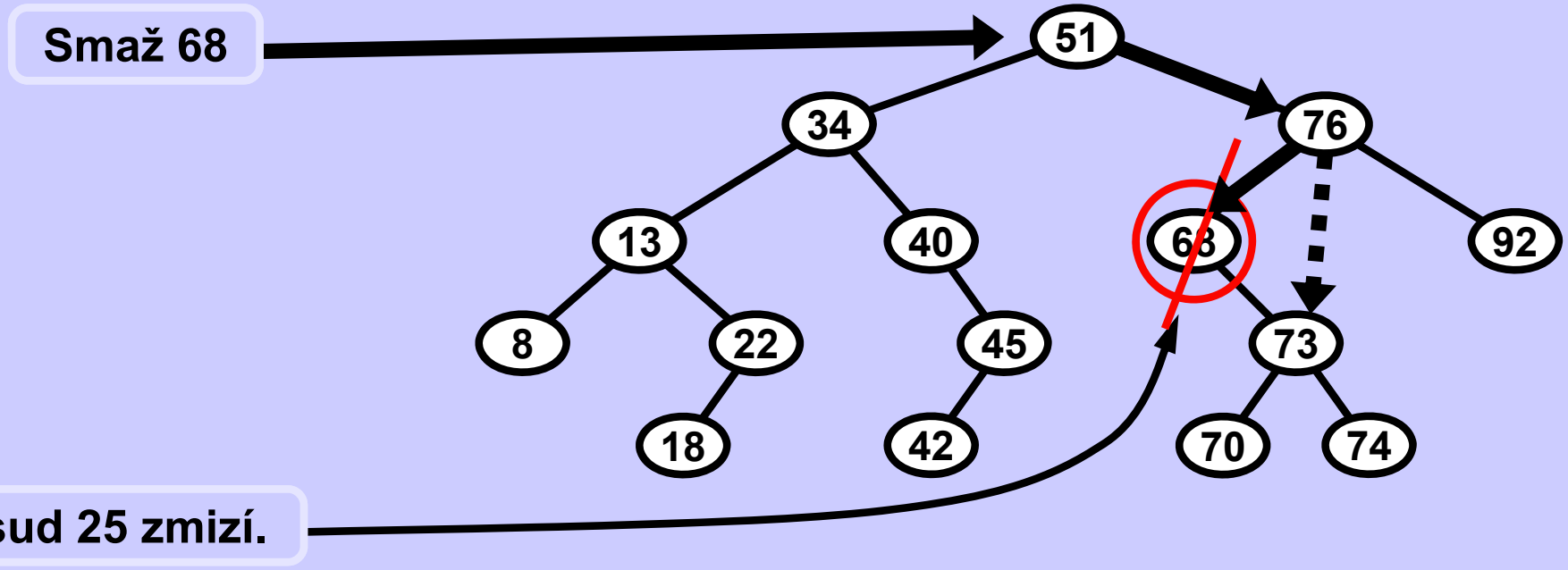
Smazání uzlu s 0 potomky (= listu)



Delete I.
Najdi daný uzel (jako ve Find) a odstraň ukazatel na něj z jeho rodiče.

Operace Delete v BVS (II.)

Smazání uzlu s 1 potomkem



Z ukazatele 76 --> 68 se stane ukazatel 76 --> 73.

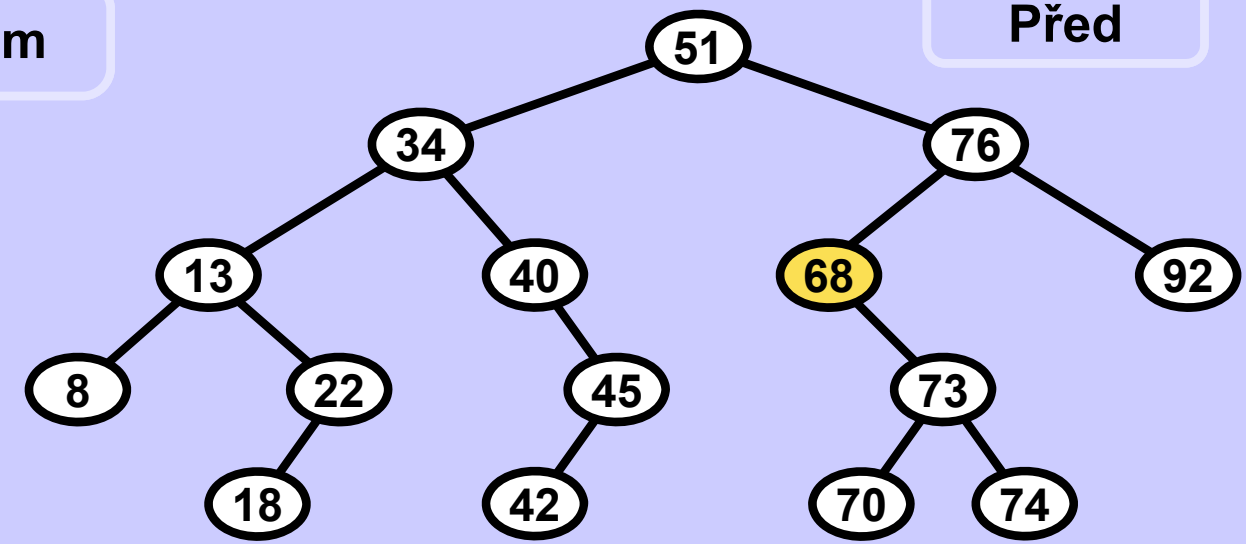
Delete II.
Najdi daný uzel (jako ve Find) a ukazatelem z jeho rodiče na něj ukaž na jeho (jediného!) potomka.

Operace Delete v BVS (II.)

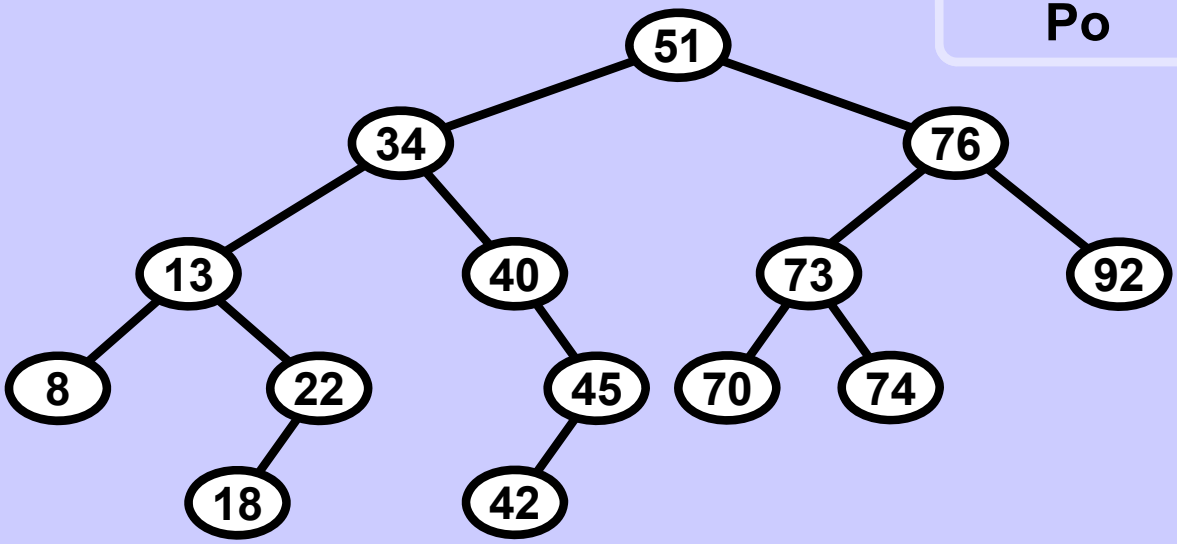
Smazání uzlu s 1 potomkem

Smaž 68

Před



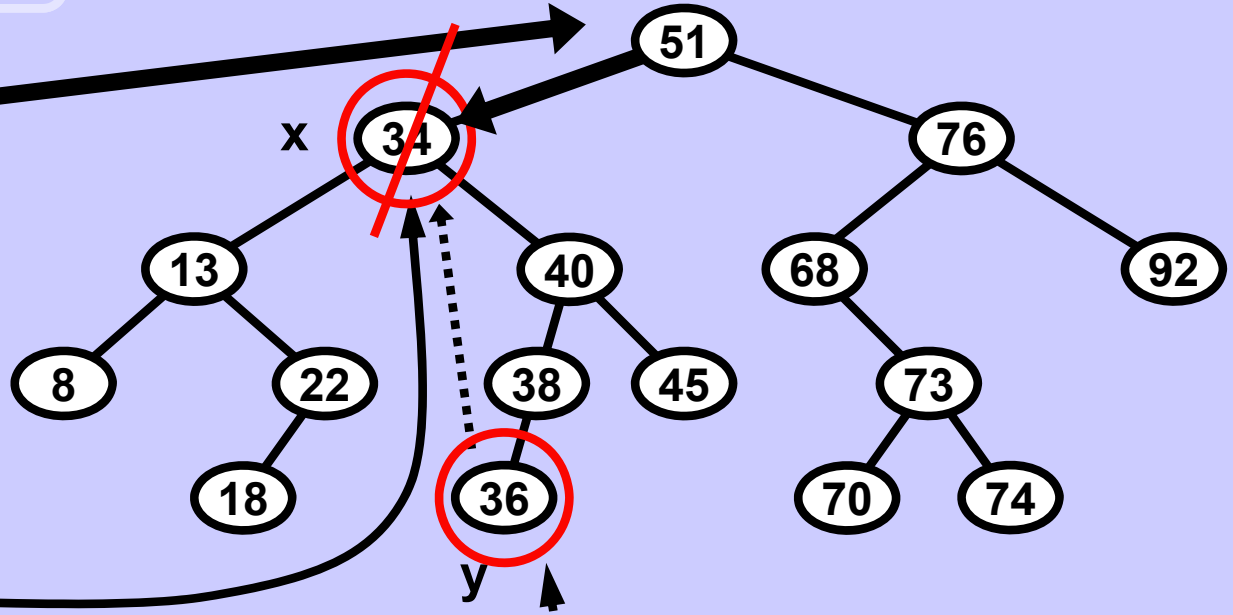
Po



Operace Delete v BVS (IIIa.)

Smazání uzlu s 2 potomky

Smaž 34



Odsud 34 zmizí.

Na jeho místo nastoupí 36.

Delete IIIa.

1. Najdi daný uzel x (jako ve Find) a dále
   najdi ***nejlevější*** (=nejmenší klíč) uzel y v ***pravém*** podstromu x.
2. Z uzlu y ukaž na potomky uzlu x, z rodiče y ukaž na potomka y místo y,
   z rodiče x ukaž na y.

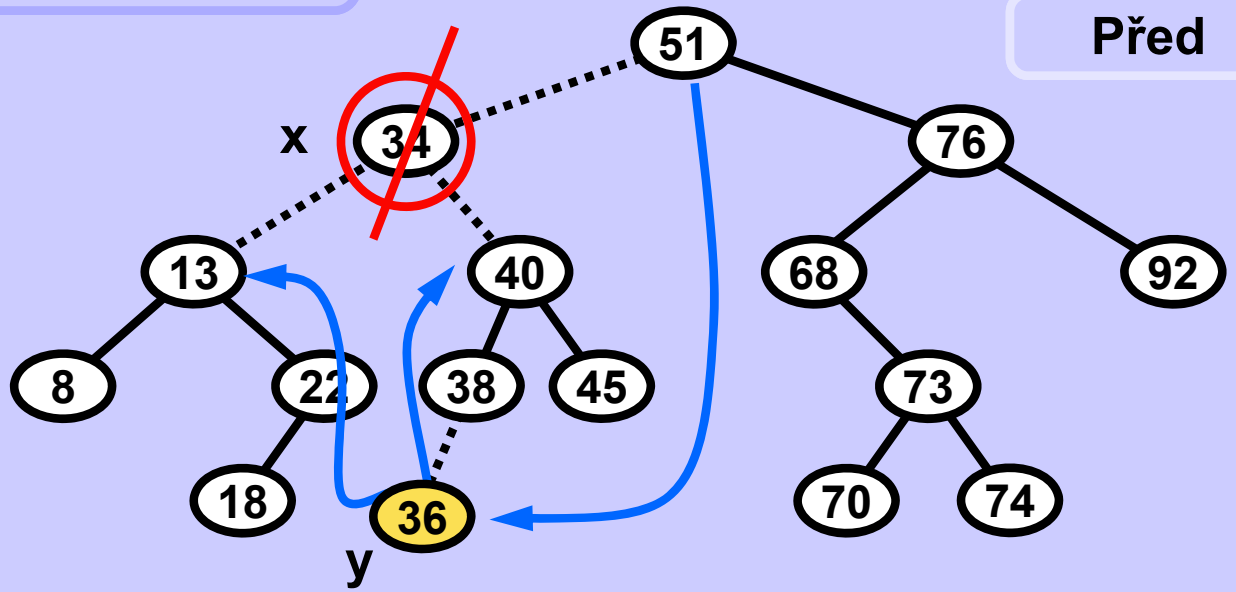
Operace Delete v BVS (IIIa.)

Smaž 34

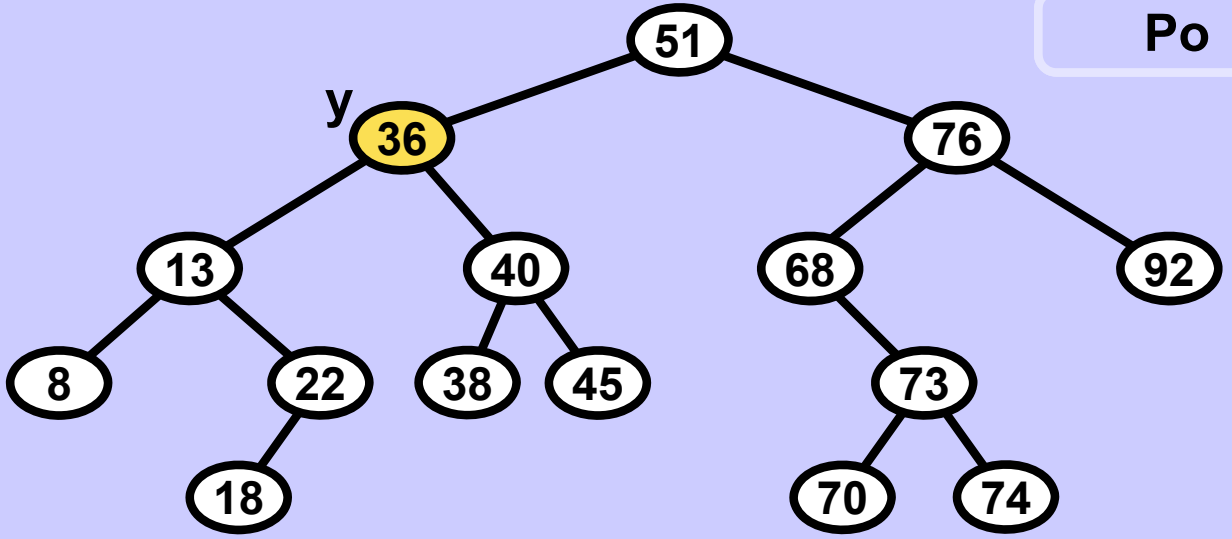
zanikající hrany/ukazatele/reference

vznikající hrany/ukazatele/reference

Před

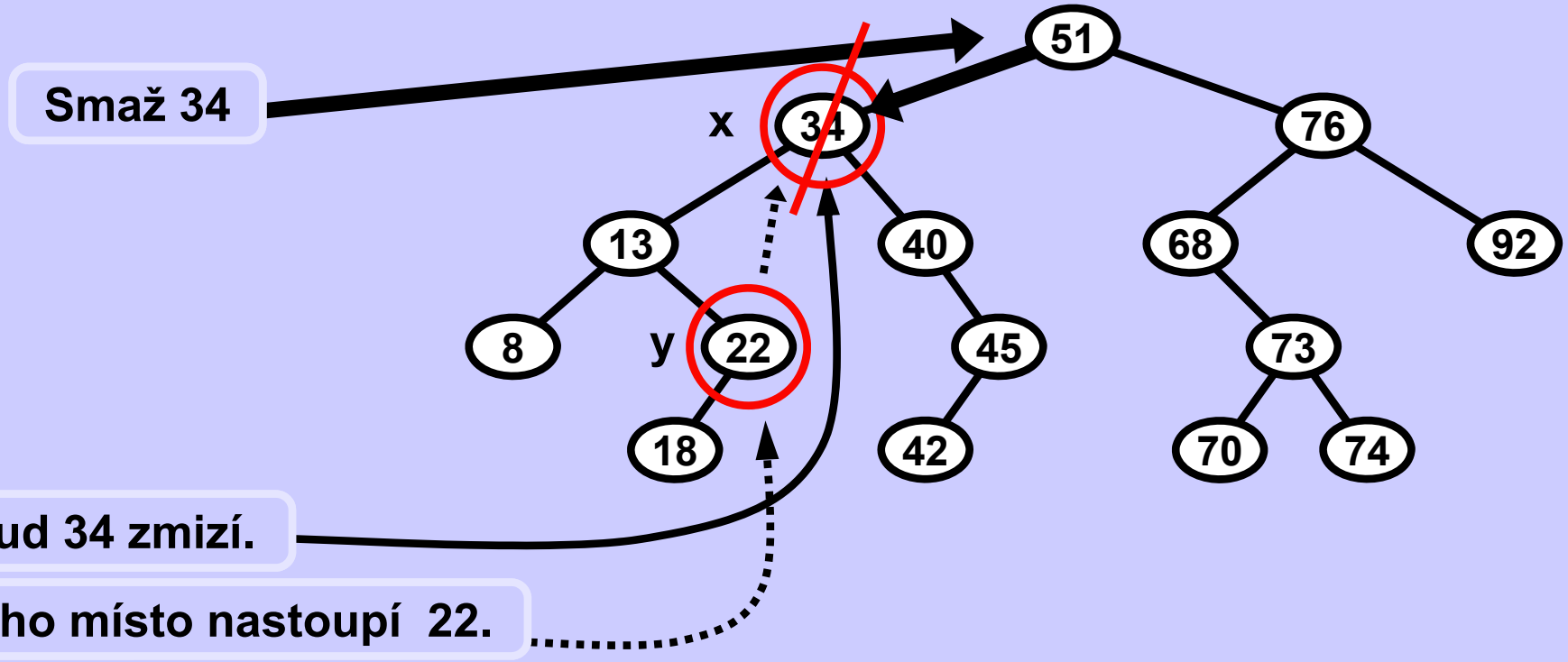


Po



Operace Delete v BVS (IIIb.) je ekvivalentní Delete IIIa.

Smazání uzlu s 2 potomky



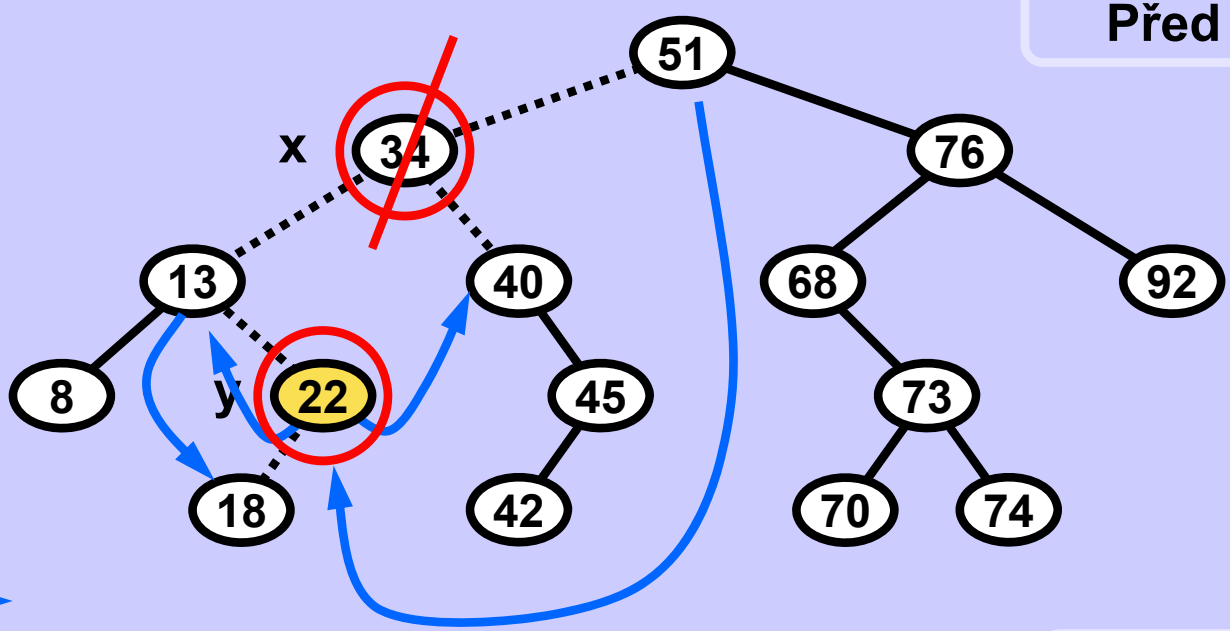
Delete IIIb.

1. Najdi daný uzel x (jako ve Find) a dále
   najdi ***nepravější*** (=největší klíč) uzel y v ***levém*** podstromu x.
2. Z uzlu y ukaž na potomky uzlu x, z rodiče y ukaž na potomka y místo y,
   z rodiče x ukaž na y.

Operace Delete v BVS (IIIb.) je ekvivalentní Delete IIIa.

Smaž 34

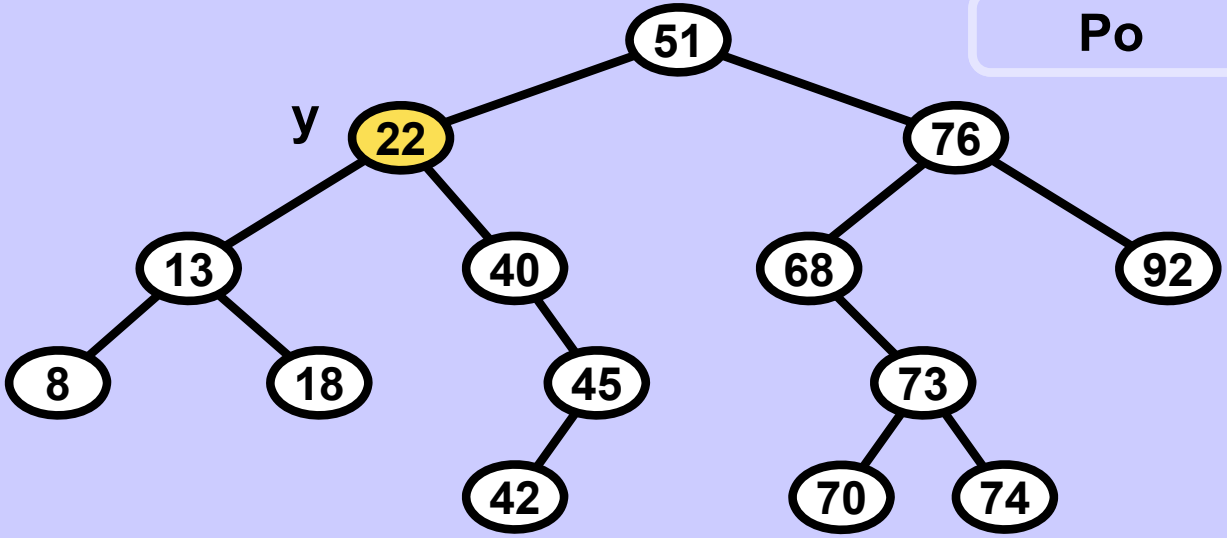
Před



zanikající hrany/ukazatele/reference

vznikající hrany/ukazatele/reference

Po



Je nutno ošetřit případ, kdy přesouvaný uzel y má sám následníka, tedy je na něj nutno aplikovat variantu Delete II.

Operace Delete v BVS rekurzivně

```
Node FindMin( Node root ){
  while( root->left != null ) root = root->left;
  return root;
}

Node Delete( Node root, int data ){
  if( root == null ) return root;
  else if( data < root->data )
                  root->left  = Delete( root->left, data );
  else if( data > root->data )
                  root->right = Delete( root->right, data );
  else {
    // Case 1: No Child
    if( root->left == null && root->right == null ){
      delete root;
      root = NULL;
     ...
```

Operace Delete v BVS rekurzivně

```
    ...
    // Case 2: one child
    } else if( root->left == null ){
      Node temp = root;
      root = root->right;
      delete temp;
    } else if( root->right == null ){
      Node temp = root;
      root = root->left;
      delete temp;
     // Case 3:two children
    } else{
      Node temp = FindMin( root->right );
      root->data = temp->data;
      root->right = Delete( root->right, temp->data );
    }
  }
  return root;
}
```

Asymptotické složitosti operací Find, Insert, Delete v BVS

	BVS s n uzly	
Operace	**Vyvážený**	**Možná nevyvážený**
Find	$O(\log(n))$	$O(n)$
Insert	$\Theta(\log(n))$	$O(n)$
Delete	$\Theta(\log(n))$	$O(n)$