# Rekurzivní algoritmy

prof. Ing. Pavel Tvrdík CSc.

Katedra počítačových systémů FIT
České vysoké učení technické v Praze

DSA, ZS 2009/10, Předn. 7

http://service.felk.cvut.cz/courses/X36DSA/

# Rekurze

- Rekurze je metoda zápisu algoritmu, kde se **stejný postup** opakuje na částech vstupních dat.
- Výhody:
  - úsporná: zápis kódu je kratší a konečným zápisem lze definovat $\infty$ množinu dat,
  - přirozená: opakování a samopodobnost jsou v přírodě běžné,

  - intuitivní: explicite pojmenovává to, co se opakuje v menším,
  - expresivní: rekurzivní specifikace umožňuje snadnou analýzu složitosti a ověření správnosti (viz např. MergeSort v Přednášce 6).
- Nevýhody:
  - Interpretace nebo provádění rekurzivního kódu používá systémový **zásobník** pro předání parametrů volání a proto vyžaduje **systémovou paměť**.
  - Dynamická alokace systémového zásobníku a ukládání parametrů na něj navíc představuje **časovou režii**.
- Prakticky všechny dnešní VPJ povolují rekurzivní programování.

# Typy rekurze

- **Koncová rekurze**: rekurzivní volání je posledním příkazem algoritmu, po kterém se už neprovádějí žádné další "odložené" operace.

## Příklad

**Největší společný dělitel (Euclid)**: Nechť $n \geq m \geq 0$.

$$\mathrm{GCD}(n,m) = \begin{cases} n & \text{if } m = 0, \\ \mathrm{GCD}(m, \textsc{Remainder}(n,m)) & \text{if } m > 0. \end{cases}$$

- **Lineární rekurze**: V těle algoritmu je pouze jedno rekurzivní volání anebo jsou dvě, ale vyskytují se v disjunktních větvích podmíněných příkazů a nikdy se neprovedou současně.

### Příklad

**Faktoriál**:

$$\textsc{Fac}(n) = \begin{cases} 1 & \text{if } n = 1, \\ n * \textsc{Fac}(n-1) & \text{if } n > 1. \end{cases}$$

### Příklad

**Skalární součin vektorů**:

$$\mathrm{SSV}(A, B, n) = \begin{cases} 0 & \text{if } n = 0, \\ A[n] * B[n] + \mathrm{SSV}(A, B, n-1) & \text{if } n > 0. \end{cases}$$

Strom rekurzivních volání je lineární.

- **Kaskádní rekurze**: V těle algoritmu jsou vedle sebe aspoň dvě rekurzivní volání. Viz MERGESORT, QUICKSORT.

## Příklad

**Fibonacciho čísla**:

$$F(n) = \begin{cases} 1 & \text{if } n = 1, 2, \\ F(n-1) + F(n-2) & \text{if } n > 2. \end{cases}$$

## Příklad

**Lineární rekurence**:

$$F(n) = \begin{cases} 1 & \text{if } n \leq 3, \\ a_1 * F(n-1) + a_2 * F(n-2) + a_3 * F(n-3) & \text{if } n > 3. \end{cases}$$

Strom rekurzivních volání je více-ární.

- **Vnořená rekurze**: Rekurzivní funkce, jejíž argumenty jsou specifikovány rekurzivně.

### Příklad

**Ackermannova funkce**:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Neuvěřitelně rychle rostoucí funkce, viz `http://mathworld.wolfram.com/AckermannFunction.html`.

# Rekurze vs. iterace

- Rekurzivní programování má základní oporu v teoretické informatice, kde bylo dokázáno, že:
  1. Každou funkci, která může být implementovaná na vonNeumannovském počítači, lze vyjádřit rekurzivně bez použití iterace.
  2. Každá rekurzivní funkce se dá vyjádřit iterativně s použitím zásobníku (implicitní zásobník se stane viditelný uživateli).
- Koncovou rekurzi lze vždy nahradit iterací bez nutnosti zásobníku.

# Rekurzivní HEAPIFY

## Algoritmus

**procedure** $\textsc{Heapify}(A, i)$

$\{$

(1) $l \leftarrow \textsc{Left}(i)$;

(2) $r \leftarrow \textsc{Right}(i)$;

(3) **if** $(l \leq Heap\_Size(A)\ \&\ A[l] > A[i])$

(4) **then** $Largest \leftarrow l$ **else** $Largest \leftarrow i$;

(5) **if** $(r \leq Heap\_Size(A)\ \&\ A[r] > A[Largest])$

(6) **then** $Largest \leftarrow r$;

(7) **if** $(Largest \neq i)$

(8) **then** $\{A[i] \leftrightarrow A[Largest]; \textsc{Heapify}(A, Largest)\}$

$\}$

# Iterativní HEAPIFY

## Algoritmus

```
procedure ITERHEAPIFY(A, i)
{
    while (i ≤ ⌊Heap_Size(A)/2⌋)
      do {l ← LEFT(i);
          r ← RIGHT(i);
          if (l ≤ Heap_Size(A) & A[l] > A[i])
             then Largest ← l else Largest ← i;
          if (r ≤ Heap_Size(A) & A[r] > A[Largest])
             then Largest ← r;
          if (Largest = i) return();
          A[i] ↔ A[Largest];
           i ← Largest;
    }      };
```

# Rekurzivní QuickSort

## Algoritmus

**procedure** $\textsc{QuickSort}(A, low, high)$

{

(1) **if** $(low < high)$

(2) **then** { $pivot \leftarrow \textsc{SelectPivot}(A, low, high);$

(3) $mid \leftarrow Rozdel(A, low, high, pivot);$

(4) $\textsc{QuickSort}(A, low, mid);$

(5) $\textsc{QuickSort}(A, mid + 1, high)$ }

}

# Méně rekurzivní $\textsc{QuickSort}$

## Algoritmus

**procedure** $\textsc{QuickSortTail}(A, low, high)$

$\{$

(1) **while** $(low < high)$

(2) **do** $\{\ pivot \leftarrow \textsc{SelectPivot}(A, low, high);$

(3) $mid \leftarrow Rozdel(A, low, high, pivot);$

(4) $\textsc{QuickSortTail}(A, low, mid);$

(5) $low \leftarrow mid + 1\ \}$

$\}$

# Algoritmy nad stromy

- Binární strom je definován rekurzivně (viz Slajd 16 v přednášce 5).
- Proto řada algoritmů nad stromy je rekurzivních.
- Zde se zaměříme na binární stromy s tím, že většina algoritmů je zobecnitelná pro $k$-ární stromy.

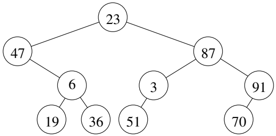
```
struct node
{
  int value; //hodnota uložená v uzlu
  int index; //pořadové číslo uzlu
  node *left; //ukazatel na levý podstrom
  node *right //ukazatel na pravý podstrom
}
```

# Generování náhodného binárního stromu

- BS s náhodnou strukturou: náhodně chybějící levé a pravé podstromy.
- Hodnoty v uzlech jsou náhodná čísla z intervalu $[1, 99]$.

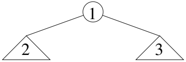
## Algoritmus

**function** $\textsc{RandBinTree}(h)$ *// $h$ je výška*
(1){ **if** $(h \geq 0$ & $\textsc{Random}(1,2) = 1)$
(2) **then** { *vygeneruj dynamicky nový uzel $NewNode$;*
(3) $NewNode.left \leftarrow \textsc{RandBinTree}(h-1)$;
(4) $NewNode.value \leftarrow \textsc{Random}(1,99)$;
(5) $NewNode.right \leftarrow \textsc{RandBinTree}(h-1)$;
(6) **return**$(NewNode)$}
(7) **else return**(`NIL`) }

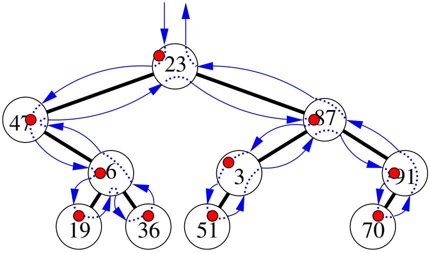


# 3 základní algoritmy procházení stromů a číslování uzlů

- **PreOrder**
- InOrder
- PostOrder



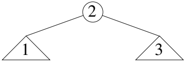

$c \leftarrow 0$; //globální čítač uzlů
**procedure** PREORDERWALK($r$)
{// $r$ je ukazatel na kořen stromu
(1) **if** ($r$ = NIL) **then return**();
(2) $r.index \leftarrow c$; $c \leftarrow c + 1$;
(3) PRINT($r.index, r.value$);
(4) PREORDERWALK($r.left$);
(5) PREORDERWALK($r.right$);
}



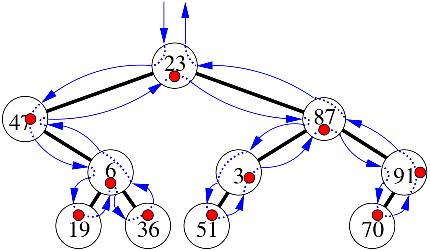

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 23 | 47 | 6 | 19 | 36 | 87 | 3 | 51 | 91 | 70 |

# 3 základní algoritmy procházení stromů a číslování uzlů

- PreOrder
- **InOrder**
- PostOrder



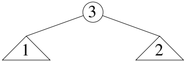

$c \leftarrow 0$; //globální čítač uzlů
**procedure** $\textsc{InOrderWalk}(r)$
$\{//\ r$ je ukazatel na kořen stromu
(1) **if** $(r = \texttt{NIL})$ **then return**();
(2) $\textsc{InOrderWalk}(r.left)$;
(3) $r.index \leftarrow c$; $c \leftarrow c + 1$;
(4) $\textsc{Print}(r.index, r.value)$;
(5) $\textsc{InOrderWalk}(r.right)$;
$\}$




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 47 | 19 | 6 | 36 | 23 | 51 | 3 | 87 | 70 | 91 |

# 3 základní algoritmy procházení stromů a číslování uzlů

- PreOrder
- InOrder
- **PostOrder**




$c \leftarrow 0$; //globální čítač uzlů

**procedure** $\textsc{PostOrderWalk}(r)$

{// $r$ je ukazatel na kořen stromu

(1) **if** $(r = \texttt{NIL})$ **then return**();

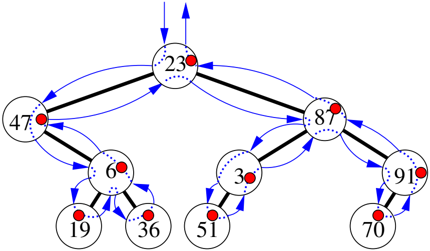
(2) $\textsc{PostOrderWalk}(r.left)$;

(3) $\textsc{PostOrderWalk}(r.right)$;

(4) $r.index \leftarrow c$; $c \leftarrow c + 1$;

(5) $\textsc{Print}(r.index, r.value)$;

}




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 19 | 36 | 6 | 47 | 51 | 3 | 70 | 91 | 87 | 23 |

# Shrnutí

- Všechny 3 průchody jsou založeny na **průchodu do hloubky** = DFS (*Depth-First-Search*).
- Liší se pouze podmínkou, kdy je procházený uzel označen (zpracován).
  - PreOrder: při **první** návštěvě.
  - InOrder: při přecházení z levého do pravého podstromu.
  - PostOrder: při **poslední** návštěvě.
- PreOrder a PostOrder lze zobecnit pro libovolné $k$-ární stromy.
- Binární strom lze tedy **linearizovat** 3 základními způsoby.
- Protože v každém souvislém grafu $G$ lze zkonstruovat **kostru**, lze takto linearizovat (očíslovat) uzly lib. souvislého grafu.

# Velikost a hloubka stromu

## Algoritmus

**function** $\textsc{TreeSize}(r)$
$\{$*// $r$ je ukazatel na kořen stromu*
(1) **if** $(r = \texttt{NIL})$ **then return**$(0)$;
(2) **return**$(\textsc{TreeSize}(r.left) + \textsc{TreeSize}(r.right) + 1)$ $\}$

## Algoritmus

**function** $\textsc{TreeDepth}(r)$
$\{$*// $r$ je ukazatel na kořen stromu*
(1) **if** $(r = \texttt{NIL})$ **then return**$(0)$;
(2) **return**$(\max(\textsc{TreeDepth}(r.left), \textsc{TreeDepth}(r.right)) + 1)$ $\}$

# Rekurzivně definované struktury

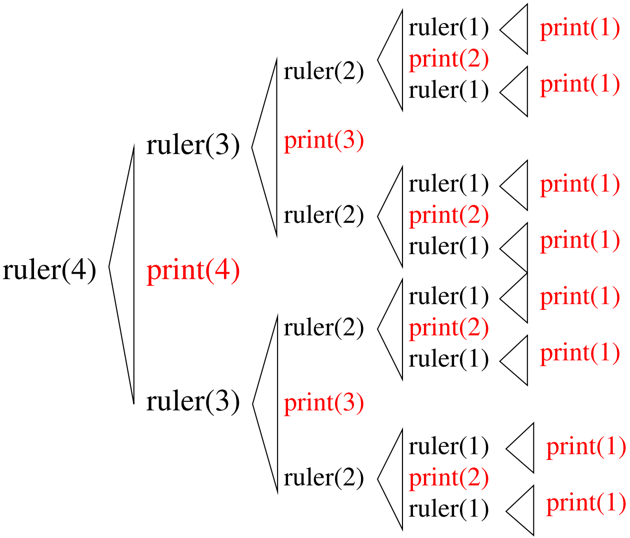
## Algoritmus

```
procedure RULER(h)
{
    if (h < 1)
        then return();
    RULER(h − 1);
    PRINT(h);
    RULER(h − 1);
}
```

Pokud zavoláme $\textsc{Ruler}(4)$, na výstupu se objeví posloupnost čísel v tomto pořadí

$$1\ 2\ 1\ 3\ 1\ 2\ 1\ 4\ 1\ 2\ 1\ 3\ 1\ 2\ 1.$$

# Strom rekurzivních volání



# Skutečná paměťová složitost rekurzivních výpočtů

- V rekurzivní proceduře se při rekurzivním volání téže procedury (totéž pro funkce) musí na systémový zásobník uložit hodnoty parametrů a lokálních proměnných volající procedury.
- Při návratu po ukončení této vnořené procedury se tyto hodnoty obnoví, aby volající procedura mohla pokračovat se stejným kontextem jako před vstupem do rekurzivního volání.
- Výška zásobníku se rovná hloubce stromu rekurzivních volání procedury.
- Rekurzivní výpočet má tedy skrytou paměťovou náročnost úměrnou hloubce tohoto stromu.

**Poznámka**: Moderní VPJ všechna volání podprogramů (nejen rekurzivních funkcí a procedur) realizují přes systémový zásobník.

# Skutečná paměťová složitost rekurzivních algoritmů: Příklady

- MERGESORT (Přednáška 6, Slajd 3): vyžaduje $\Theta(\log n)$ paměti na zásobník.

- MS(A,Aux,1,16)
  - MS(A,Aux,1,8)
    - MS(A,Aux,1,4)
      - MS(A,Aux,1,2)
        - MS(A,Aux,1,1); MS(A,Aux,2,2)
        - M(A,Aux,1,2)
      - MS(A,Aux,3,4)
        - MS(A,Aux,3,3); MS(A,Aux,4,4)
        - M(A,Aux,3,4)
      - M(A,Aux,1,4)
    - MS(A,Aux,5,8)
      - MS(A,Aux,5,6)
        - MS(A,Aux,5,5); MS(A,Aux,6,6)
        - M(A,Aux,5,6)
      - MS(A,Aux,7,8)
        - MS(A,Aux,7,7); MS(A,Aux,8,8)
        - M(A,Aux,7,8)
      - M(A,Aux,5,8)
    - M(A,Aux,1,8)
  - MS(A,Aux,9,16)
    - MS(A,Aux,9,12)
      - MS(A,Aux,9,10)
        - MS(A,Aux,9,9); MS(A,Aux,10,10)
        - M(A,Aux,9,10)
      - MS(A,Aux,11,12)
        - MS(A,Aux,11,11); MS(A,Aux,12,12)
        - M(A,Aux,11,12)
      - M(A,Aux,9,12)
    - MS(A,Aux,13,16)
      - MS(A,Aux,13,14)
        - MS(A,Aux,13,13); MS(A,Aux,14,14)
        - M(A,Aux,13,14)
      - MS(A,Aux,15,16)
        - MS(A,Aux,15,15); MS(A,Aux,16,16)
        - M(A,Aux,15,16)
      - M(A,Aux,13,16)
    - M(A,Aux,9,16)
  - M(A,Aux,1,16)

# Skutečná paměťová složitost rekurzivních algoritmů: Příklady

- QuickSort v nejhorším případě dělení (Přednáška 6, Slajd 11, vstupní posloupnost je seřazená inverzně): vyžaduje $\Theta(n)$ paměti na zásobník!!!

QS(A,1,16) < QS(A,1,1)
QS(A,2,150) < QS(A,2,2)
QS(A,3,150) < QS(A,3,3)
QS(A,4,150) < QS(A,4,4)
QS(A,5,150) ......
< QS(A,149,14
QS(A,150,15

- QuickSortTail na Slajdu 6 v tomto nejhorším případě vyžaduje $\Theta(1)$ paměti na zásobník.

# Rekurze vs. iterace revisited

- Koncová rekurze se dá snadno nahradit iterací.
- Rekurzivní algoritmy typu rozděl a panuj, např.
    - průchody stromy,
    - ostatní rekurzivní výpočty nad stromy,
    - podobné výpočty nad rekurzivně strukturovanými daty,

  jsou geniálně jednoduché, ale vyžadují **systémový zásobník**.
- Nerekurzivní ekvivalenty musejí použít **explicitní zásobník**.

# Nerekurzivní PreOrder průchod

## Algoritmus

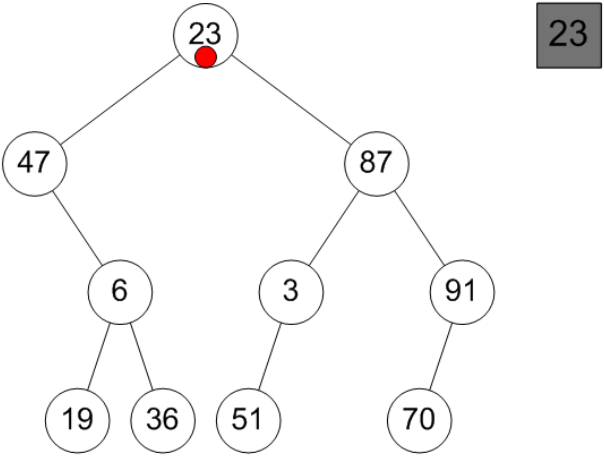
$c \leftarrow 0$; *//globální čítač uzlů*<br>
**stack** $S$; $\textsc{Init}(S)$;<br>
**procedure** $\textsc{PreOrderIterWalk}(r)$<br>
$\{$*// $r$ je ukazatel na kořen stromu*<br>
$\textsc{Push}(r, S)$;<br>
**while** $(\textsc{Not}(\textsc{Empty}(S)))$<br>
**do** $\{R \leftarrow \textsc{Top}(S)$; $\textsc{Pop}(S)$;<br>
    **if** $(R \neq \texttt{NIL})$<br>
        **then** $\{$<br>
            $R.index \leftarrow c$; $c \leftarrow c + 1$;<br>
            $\textsc{Print}(R.index, R.value)$;<br>
            $\textsc{Push}(R.right, S)$;<br>
            $\textsc{Push}(R.left, S)$; $\}$ $\}$<br>
$\}$

# Animace nerekurzivního PreOrder průchodu

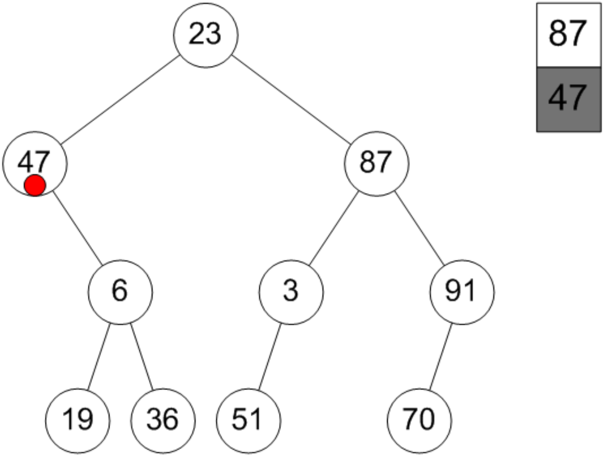
Pravidla animace:

1. Červená kolečka označují momentálně zpracovávané uzly a na zásobníku jsou označeny šedou barvou.
2. Změny stavu zásobníku spočívající v odstraňování položek `NIL` nejsou vykresleny.
3. Proto pokud jsou na vrcholu zásobníku `NIL`y, pak jsou šedou barvou a kolečky označeny položky zásobníku až po první ne`NIL`.

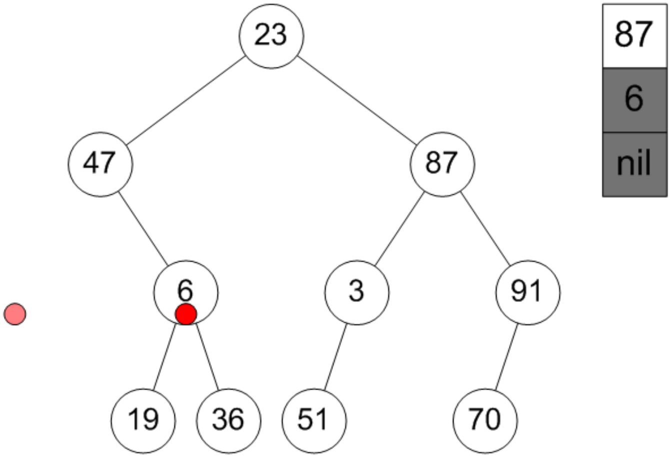
# Animace nerekurzivního PreOrder průchodu



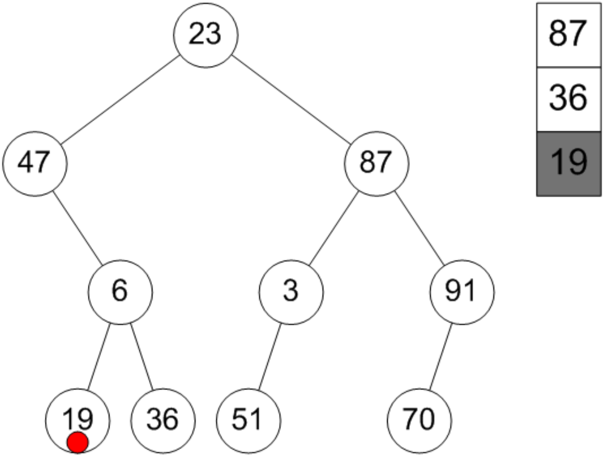
# Animace nerekurzivního PreOrder průchodu



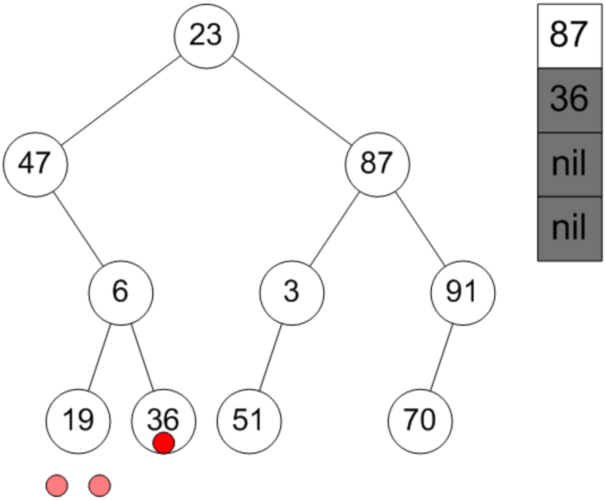
# Animace nerekurzivního PreOrder průchodu



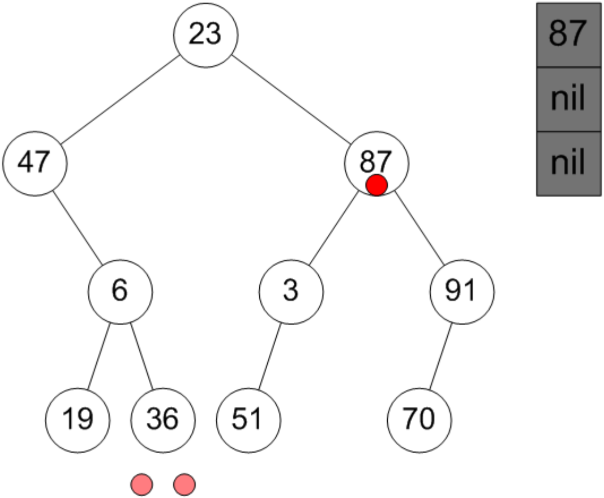
# Animace nerekurzivního PreOrder průchodu



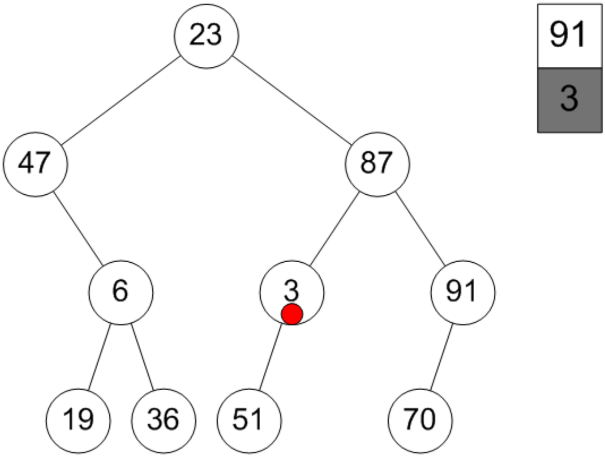
# Animace nerekurzivního PreOrder průchodu



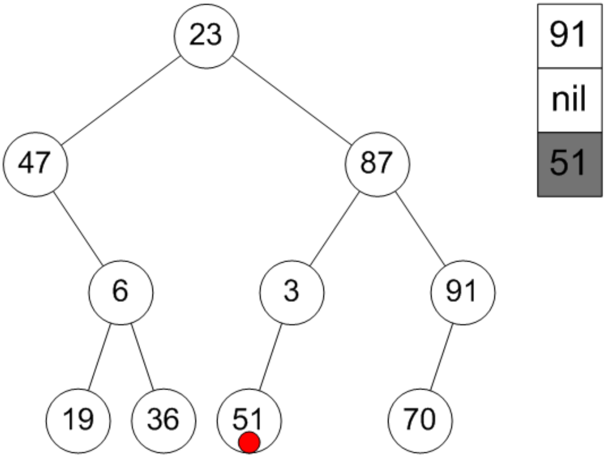
# Animace nerekurzivního PreOrder průchodu



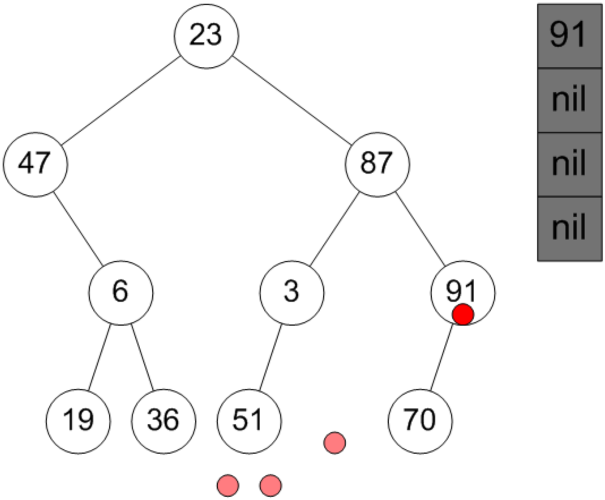
# Animace nerekurzivního PreOrder průchodu



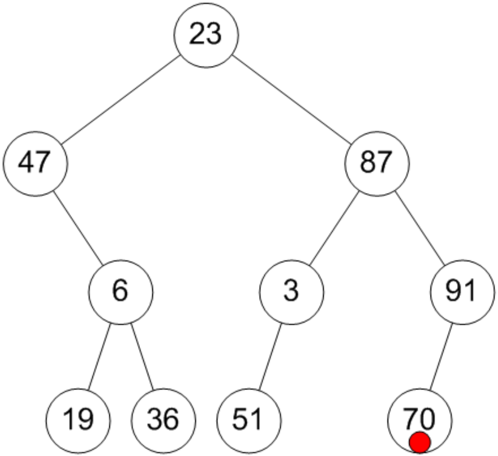
# Animace nerekurzivního PreOrder průchodu

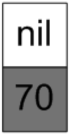


# Animace nerekurzivního PreOrder průchodu

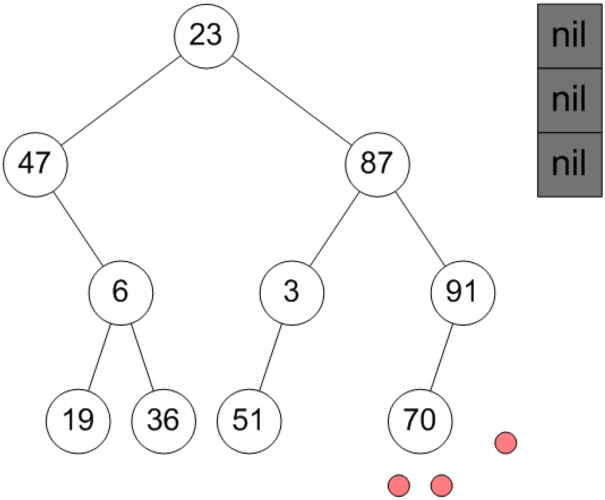


# Animace nerekurzivního PreOrder průchodu



# Animace nerekurzivního PreOrder průchodu



# Efektivnější nerekurzivní PreOrder průchod

## Algoritmus

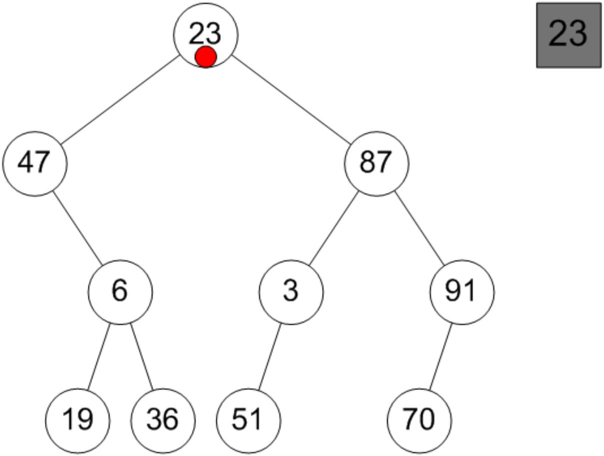
```
c ← 0; //globální čítač uzlů
stack S; Init(S);
procedure BetterPreOrderIterWalk(r)
{// r je ukazatel na kořen stromu
    Push(r, S);
    while (Not(Empty(S)))
    do {R ← Top(S); Pop(S);
        while (R ≠ NIL)
          do {
                R.index ← c; c ← c + 1;
                Print(R.index, R.value);
                Push(R.right, S);
                R ← R.left; } }
}
```

# Animace efektivnějšího nerekurzivního PreOrder průchodu

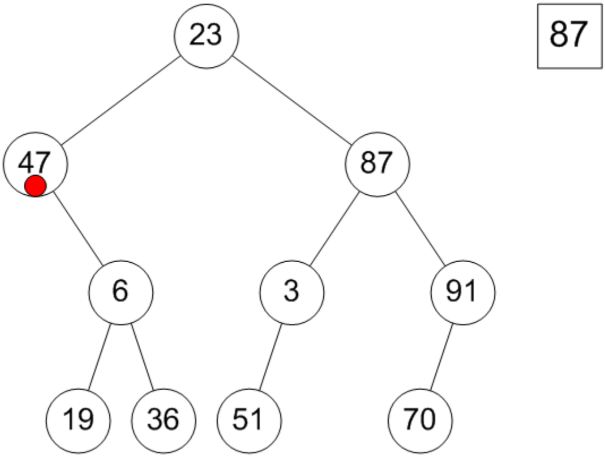
Pravidla animace:

1. Červená kolečka označují momentálně zpracovávané uzly.
2. První uzel označený kolečkem se na zásobník neukládá.
3. Změny stavu zásobníku spočívající v odstraňování položek `NIL` nejsou vykresleny.
4. Proto pokud jsou na vrcholu zásobníku `NIL`y, pak jsou šedou barvou a kolečky označeny položky zásobníku až po první ne`NIL`.

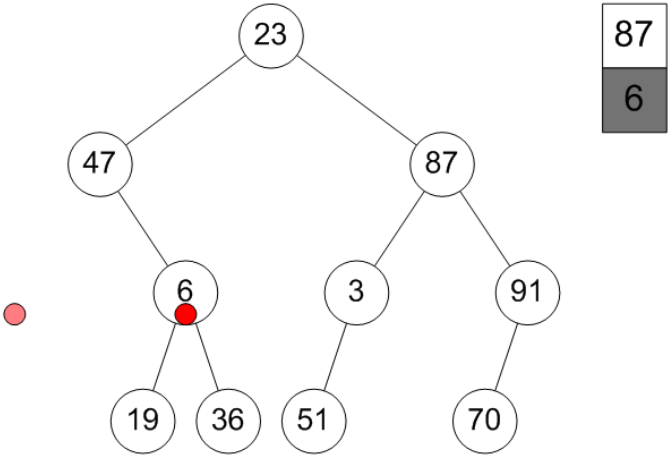
# Animace efektivnějšího nerekurzivního PreOrder průchodu



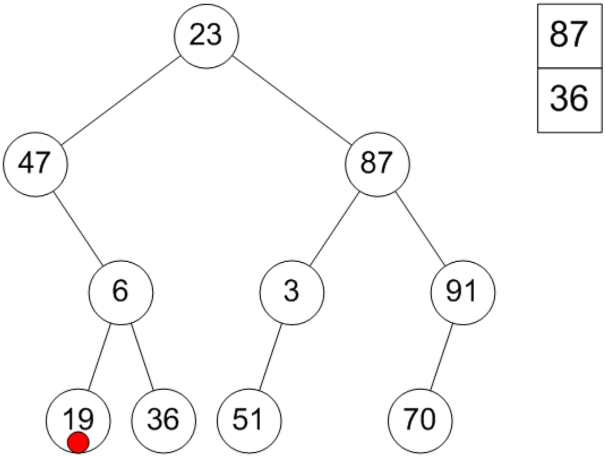
# Animace efektivnějšího nerekurzivního PreOrder průchodu



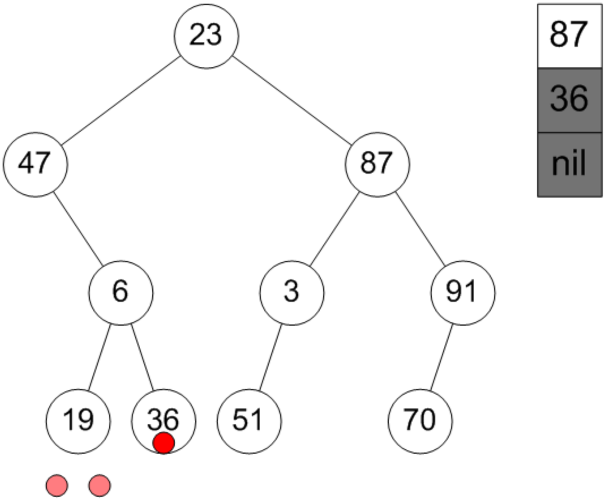
# Animace efektivnějšího nerekurzivního PreOrder průchodu



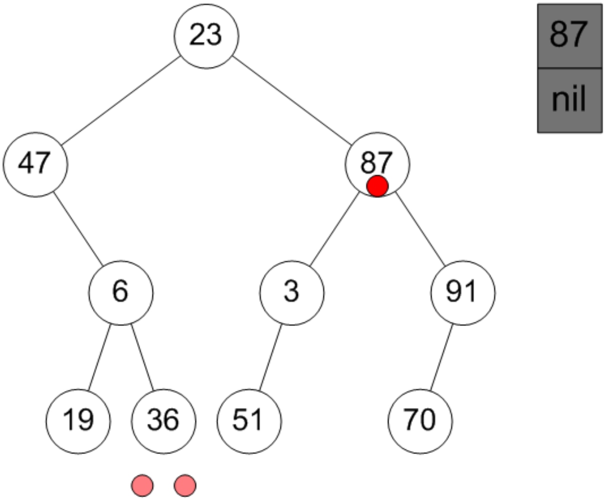
# Animace efektivnějšího nerekurzivního PreOrder průchodu



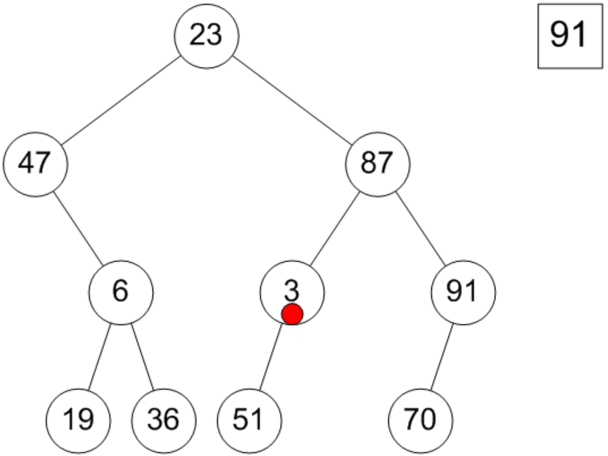
# Animace efektivnějšího nerekurzivního PreOrder průchodu



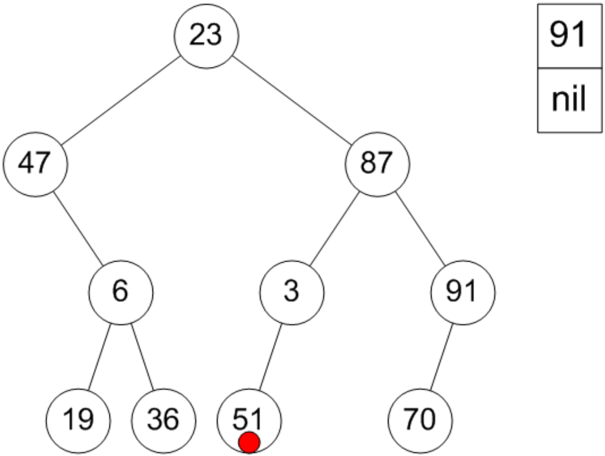
# Animace efektivnějšího nerekurzivního PreOrder průchodu



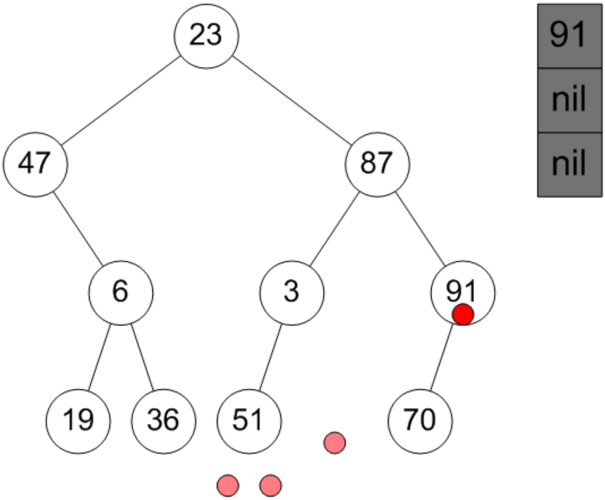
# Animace efektivnějšího nerekurzivního PreOrder průchodu



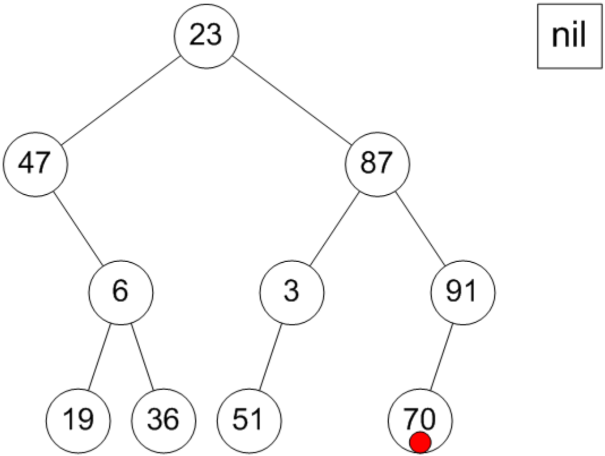
# Animace efektivnějšího nerekurzivního PreOrder průchodu



# Animace efektivnějšího nerekurzivního PreOrder průchodu



# Animace efektivnějšího nerekurzivního PreOrder průchodu



# Animace efektivnějšího nerekurzivního PreOrder průchodu