

ALGORITMIZACE 2010/03

STROMY, BINÁRNÍ STROMY
VZTAH STROMŮ A REKURZE
ZÁSObNÍK IMPLEMENTUJE REKURZI
PROHLEDÁVÁNÍ S NÁVRATEM (BACKTRACK)

Strom / tree

uzel, vrchol /
node, vertex

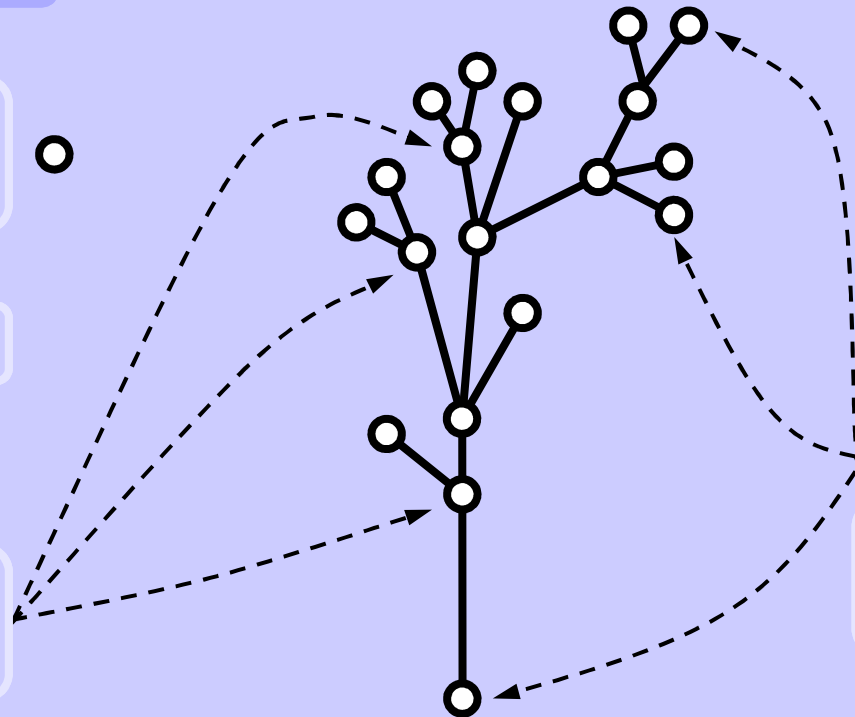


hrana / edge

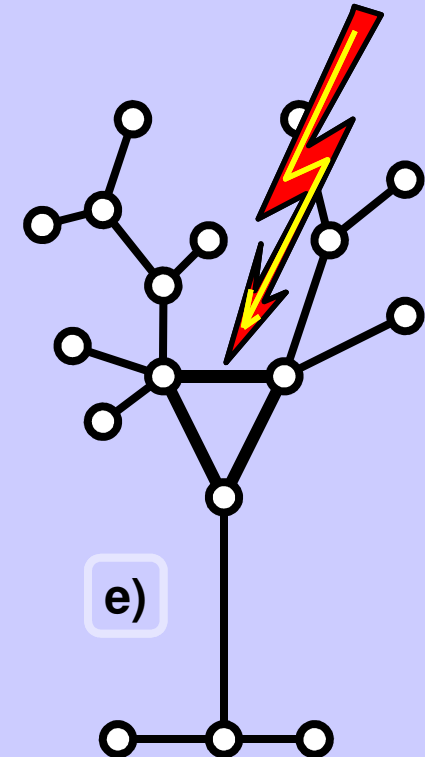
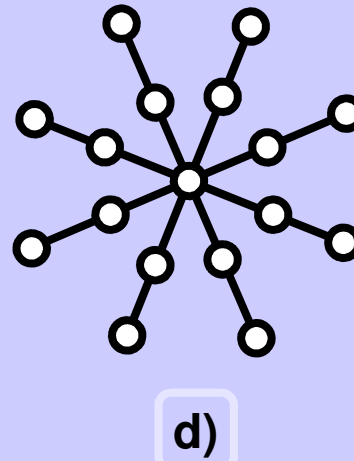
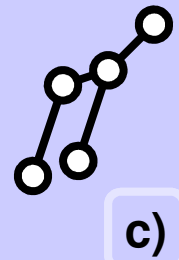
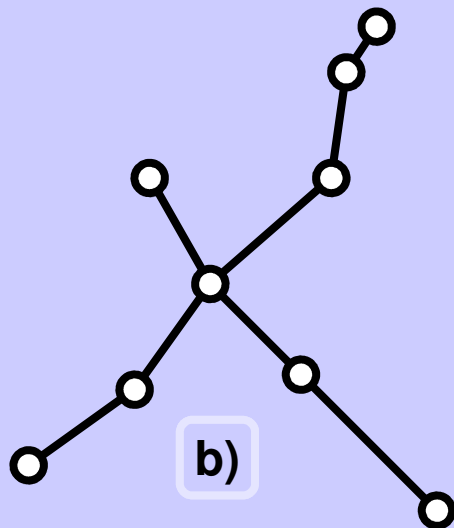


vnitřní uzel /
internal node

list /
leaf (pl. leaves)



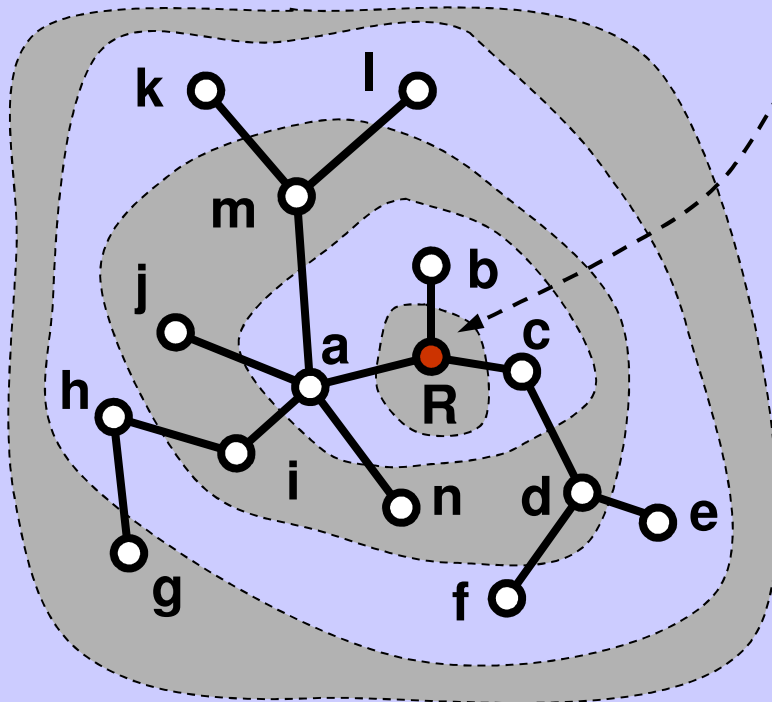
Příklady stromů



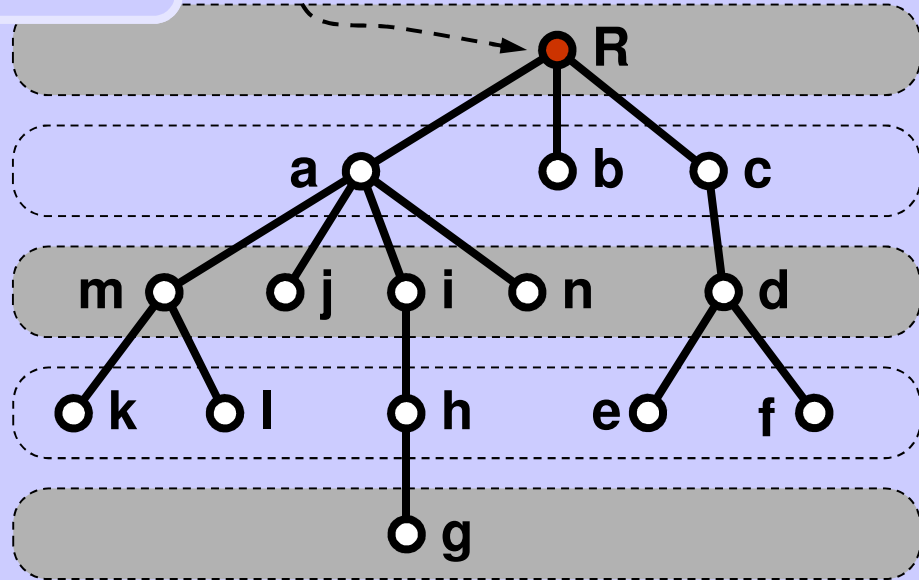
Vlastnosti stromů

1. Strom je souvislý, tj. mezi každými dvěma jeho uzly vede cesta.
2. Mezi každými dvěma uzly ve stromu vede jen jediná cesta.
3. Po odstranění libovolné hrany se strom rozpadá na dvě části.
4. Počet hran ve stromu je vždy o 1 menší než počet uzlů.

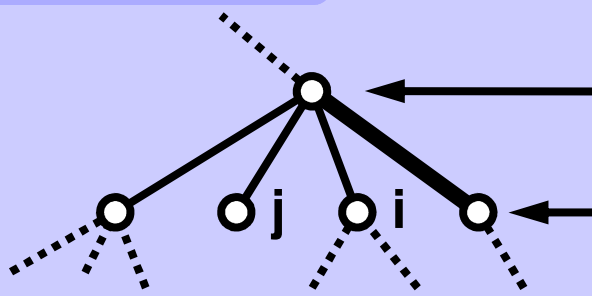
Kořenový strom / rooted tree



Kořen / root

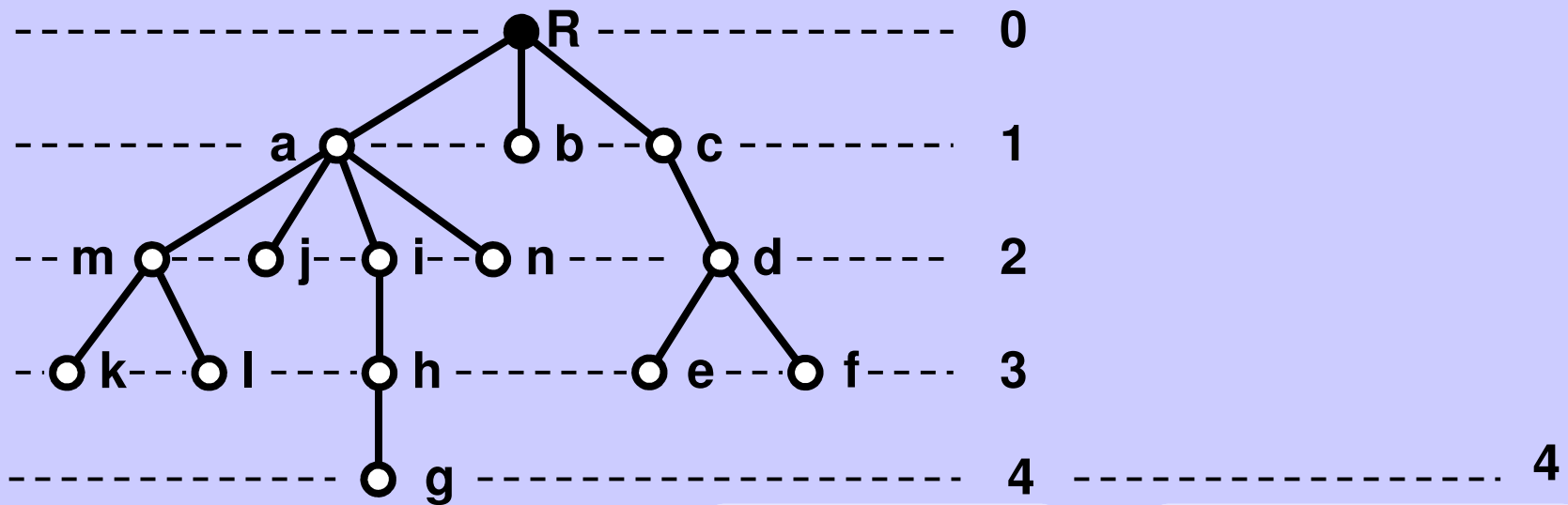


Názvosloví



Předchůdce, rodič / predecessor, parent

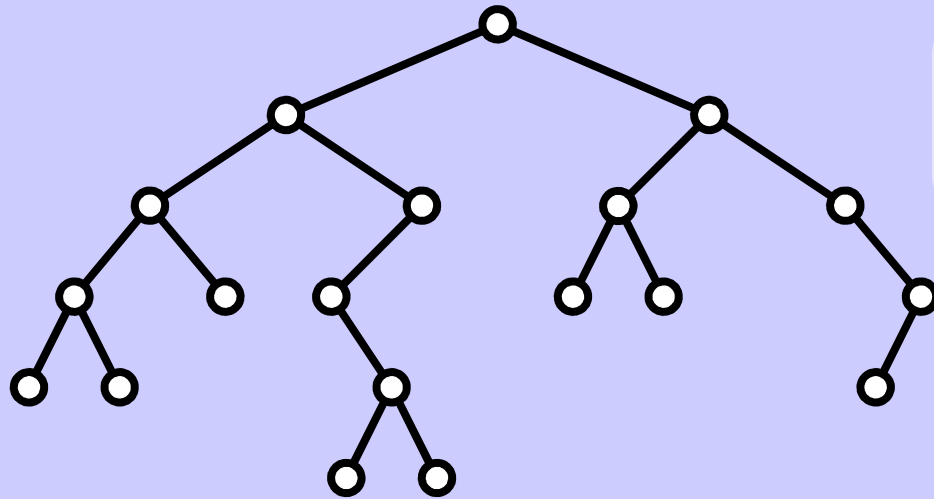
Následník, potomek / successor, child

Hloubka / depth

**Hloubka uzlu /
node depth**

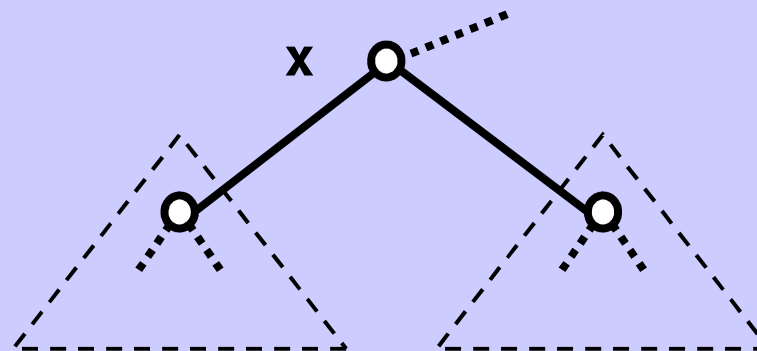
**Hloubka stromu /
tree depth**

Binární (kořenový!!) strom / binary (rooted!!) tree



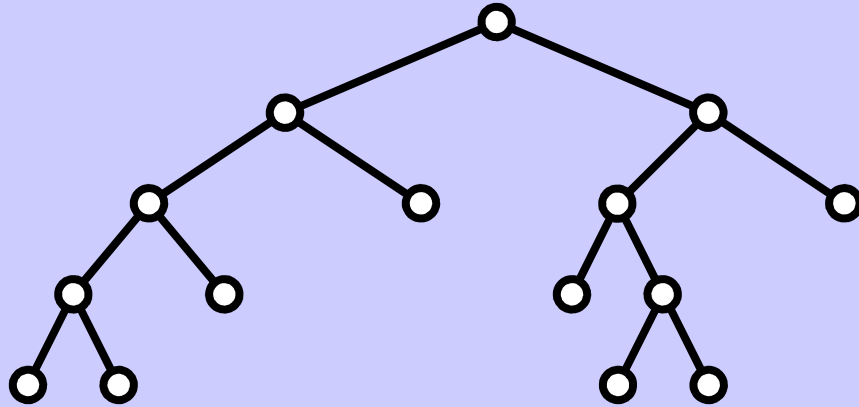
Počet potomků každého uzlu je 0,1, nebo 2.

Levý a pravý podstrom / left and right subtree



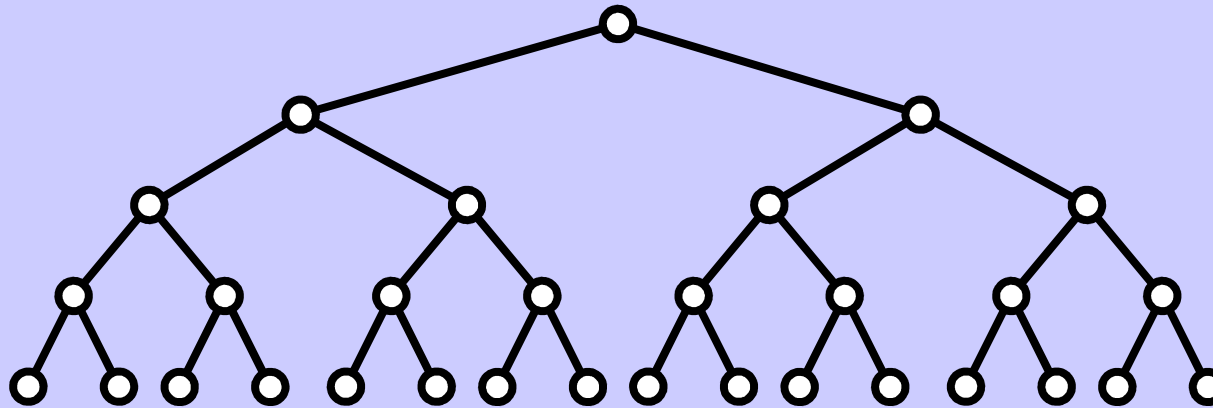
Podstrom uzlu x levý pravý

Pravidelný binární strom / regular binary tree



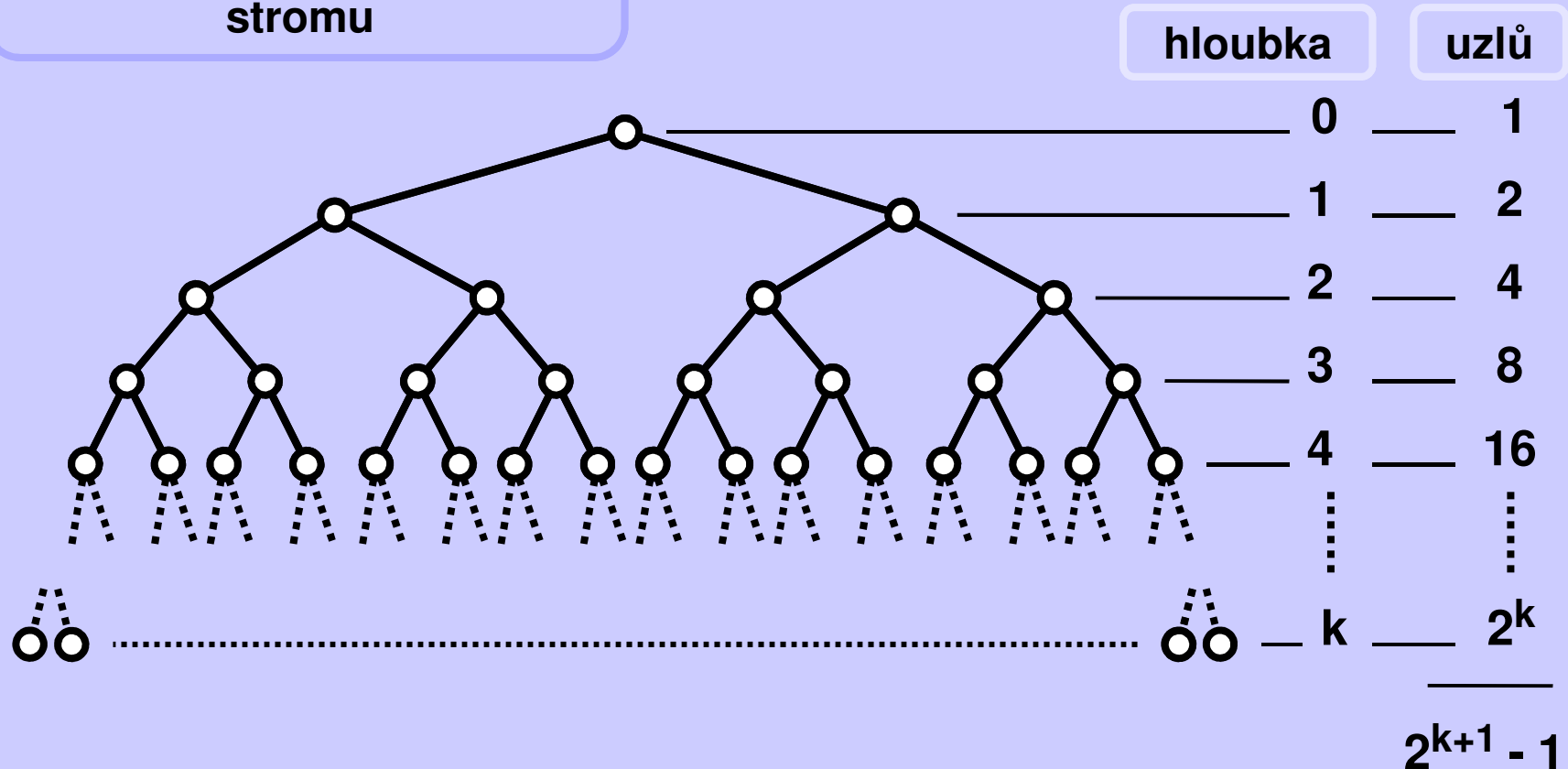
Počet potomků každého uzlu je jen 0 nebo 2.

Vyvážený strom / balanced tree



Hloubky všech listů jsou (víceméně) stejné.

**Hloubka vyváženého
(binárního a pravidelného !)
stromu**



$$(2^{(\text{hloubka vyváženého stromu})+1} - 1) \sim \text{uzlů}$$

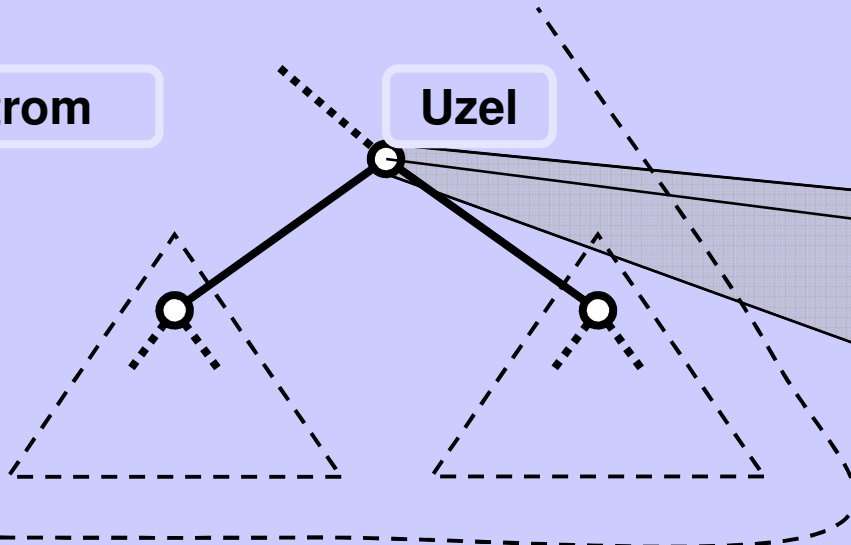
$$\text{hloubka vyváženého stromu} \sim \log_2(\text{uzlů}+1) - 1 \sim \log_2(\text{uzlů})$$

Implementace binárního stromu -- C

Strom

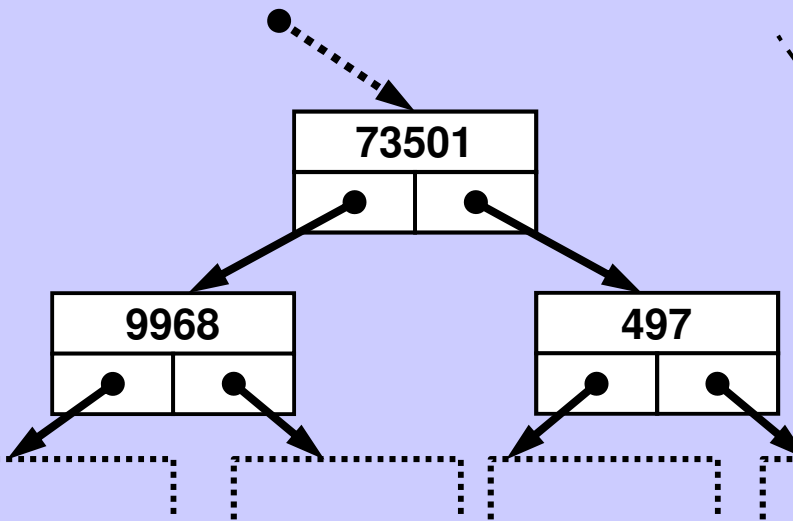
Uzel

Reprezentace uzlu



key	
left	right

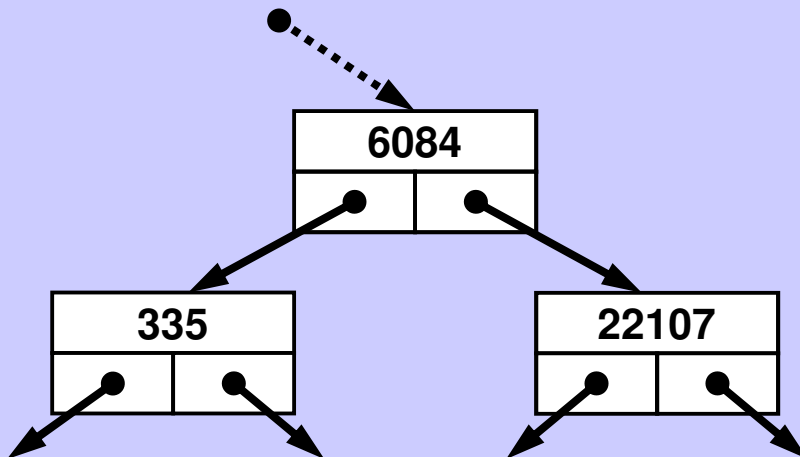
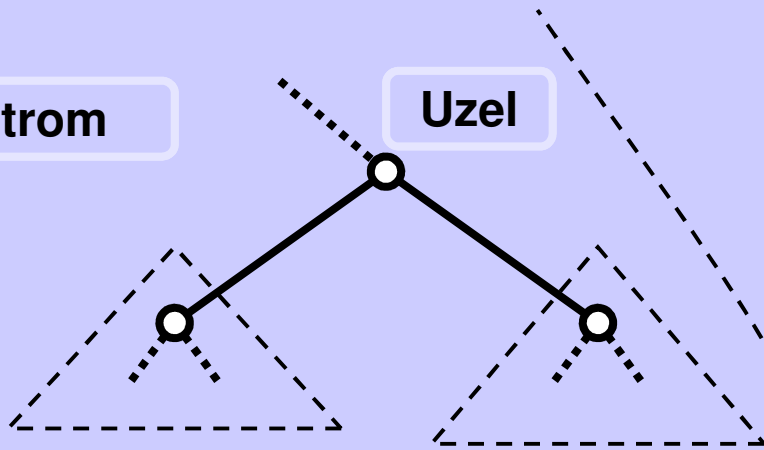
```
typedef struct node {
    int key;
    struct node *left;
    struct node *right;
} NODE;
```



Implementace binárního stromu -- Java

Strom

Uzel



```

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;
    }
}

```

Vybudování náhodného binárního stromu -- C

```

NODE *randTree(int depth) {
    NODE *pnode;
    if ((depth <= 0) || (random(10) > 7))
        return (NULL);           //stop recursion
    pnode = (NODE *) malloc(sizeof(NODE)); // create node
    if (pnode == NULL) {
        printf("%s", "No memory.");
        return NULL;
    }
    pnode->left = randTree(depth-1); // make left subtree
    pnode->key = random(100);        // some value
    pnode->right = randTree(depth-1); // make right subtree
    return pnode;                 // all done
}

```

**Příklad
volání funkce**

```

NODE *root;
root = randTree(4);

```

Poznámka. Volání random(n) vrací náhodné celé číslo od 0 do n-1.
Zde neimplementováno.

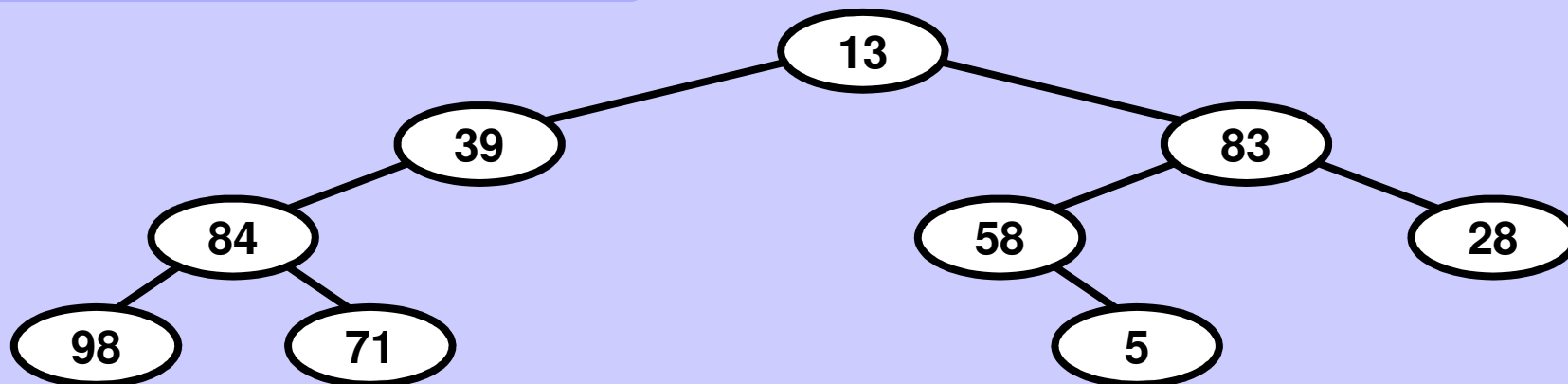
Vybudování náhodného binárního stromu -- Java

```
public Node randTree(int depth) {  
    Node node;  
    if ((depth <= 0) || ((int) Math.random()*10 > 7))  
        return null;  
    // create node with a key value  
    node = new Node((int) (Math.random()*100));  
  
    node.left = randTree(depth-1); // create left subtree  
    node.right = randTree(depth-1); // create right subtree  
    return node; // all done  
}
```

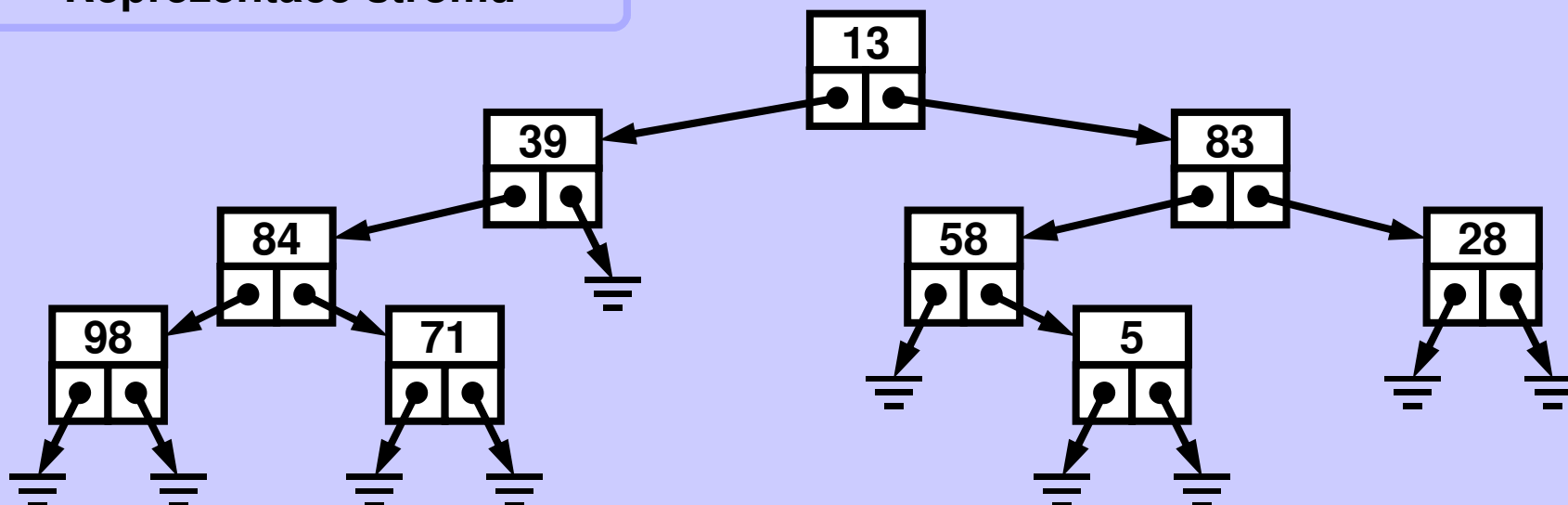
**Příklad
volání funkce**

```
Node root;  
root = randTree(4);
```

Náhodný binární strom

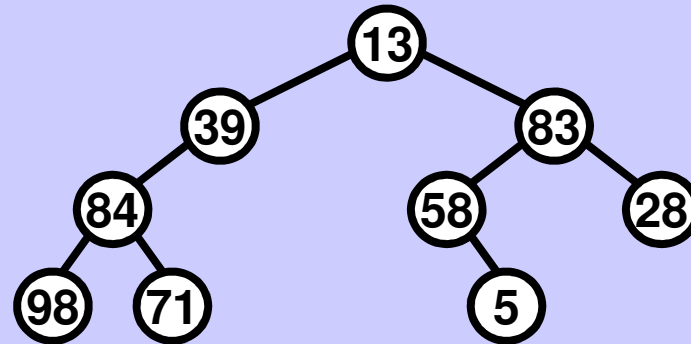


Reprezentace stromu



Průchod (binárním) stromem v pořadí Inorder

Strom



Průchod
stromem
v pořadí
INORDER

```
void listInorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    listInorder(ptr->left);  
    printf("%d ", ptr->key);  
    listInorder(ptr->right);  
}
```

Výstup

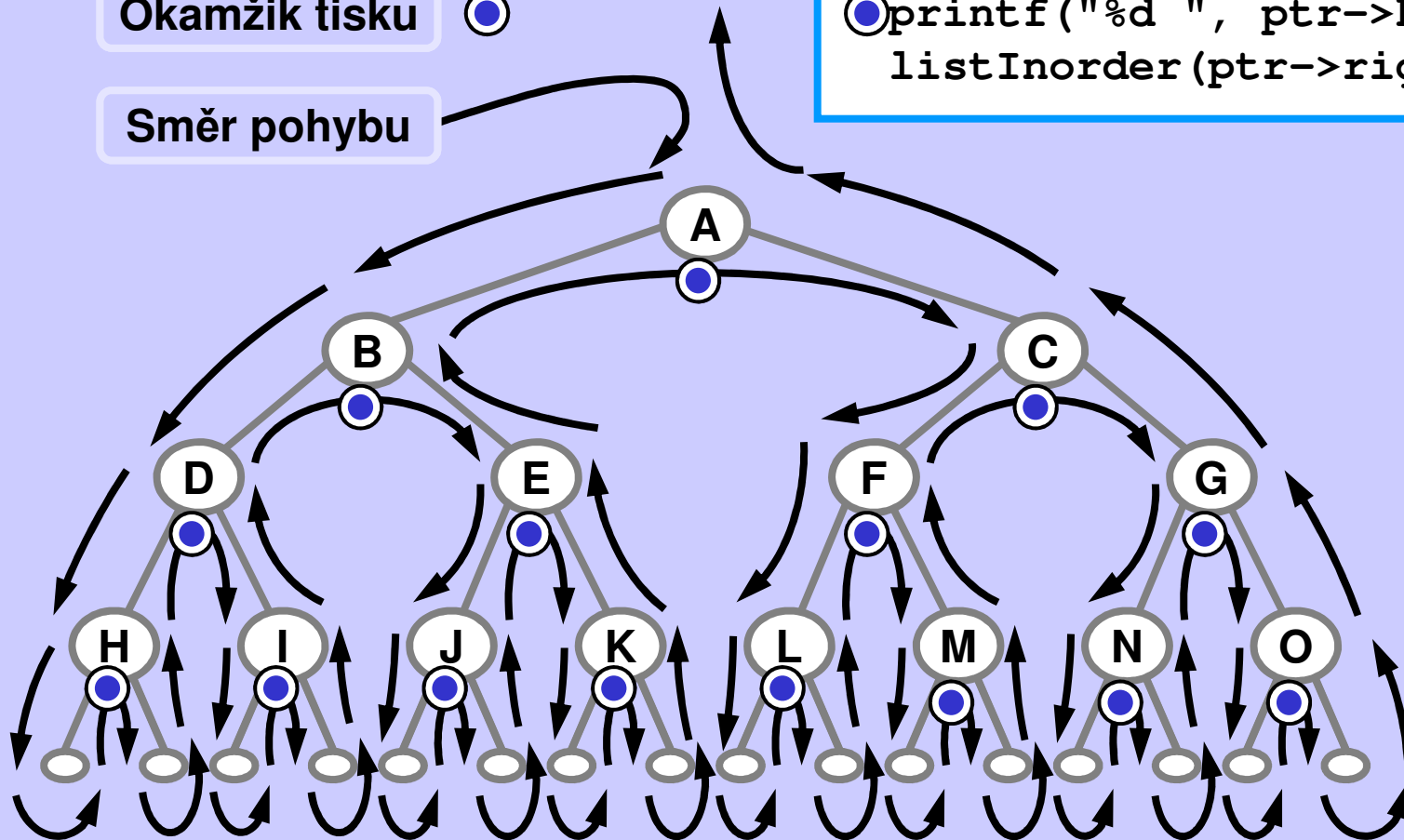
98 84 71 39 13 58 5 83 28

Pohyb ve stromu v průchodu Inorder

Okamžik tisku ○

Směr pohybu

```
listInorder(ptr->left);
○ printf("%d ", ptr->key);
listInorder(ptr->right);
```

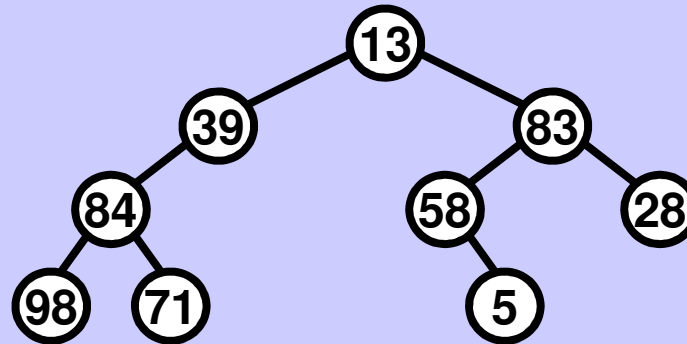


Výstup

H D I B J E K A L F M C N G O

Průchod (binárním) stromem v pořadí Preorder

Strom



Průchod
stromem
v pořadí
PREORDER

```
void listPreorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    printf("%d ", ptr->key);  
    listPreorder(ptr->left);  
    listPreorder(ptr->right);  
}
```

Výstup

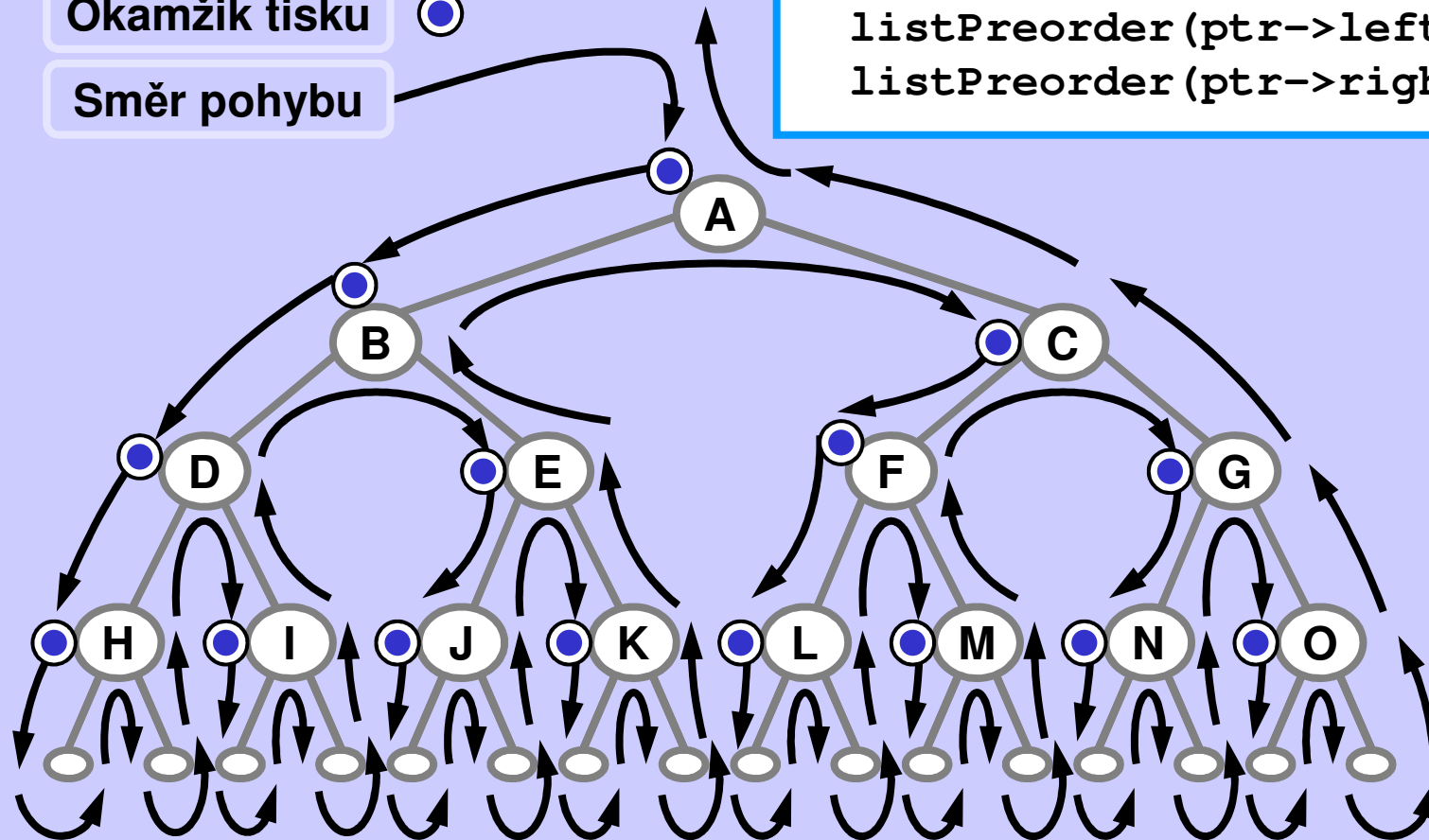
13 39 84 98 71 83 58 5 28

Pohyb ve stromu v průchodu Preorder

Okamžik tisku ○

Směr pohybu →

```
○ printf("%d ", ptr->key);
listPreorder(ptr->left);
listPreorder(ptr->right);
```

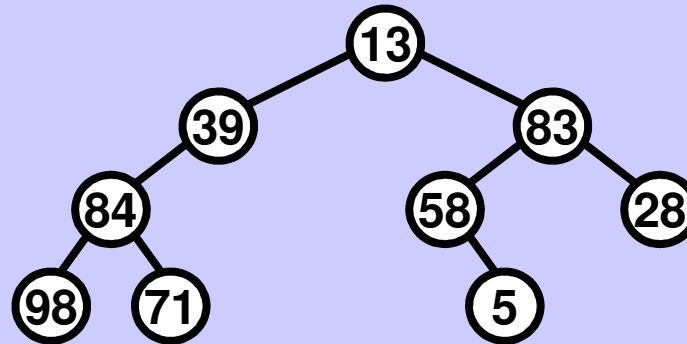


Výstup

A B D H I E J K C F L M G N O

Průchod (binárním) stromem v pořadí Postorder

Strom



Průchod
stromem
v pořadí

POSTORDER

```
void listPostorder( NODE *ptr) {  
    if (ptr == NULL) return;  
    listPostorder(ptr->left);  
    listPostorder(ptr->right);  
    printf("%d ", ptr->key);  
}
```

Výstup

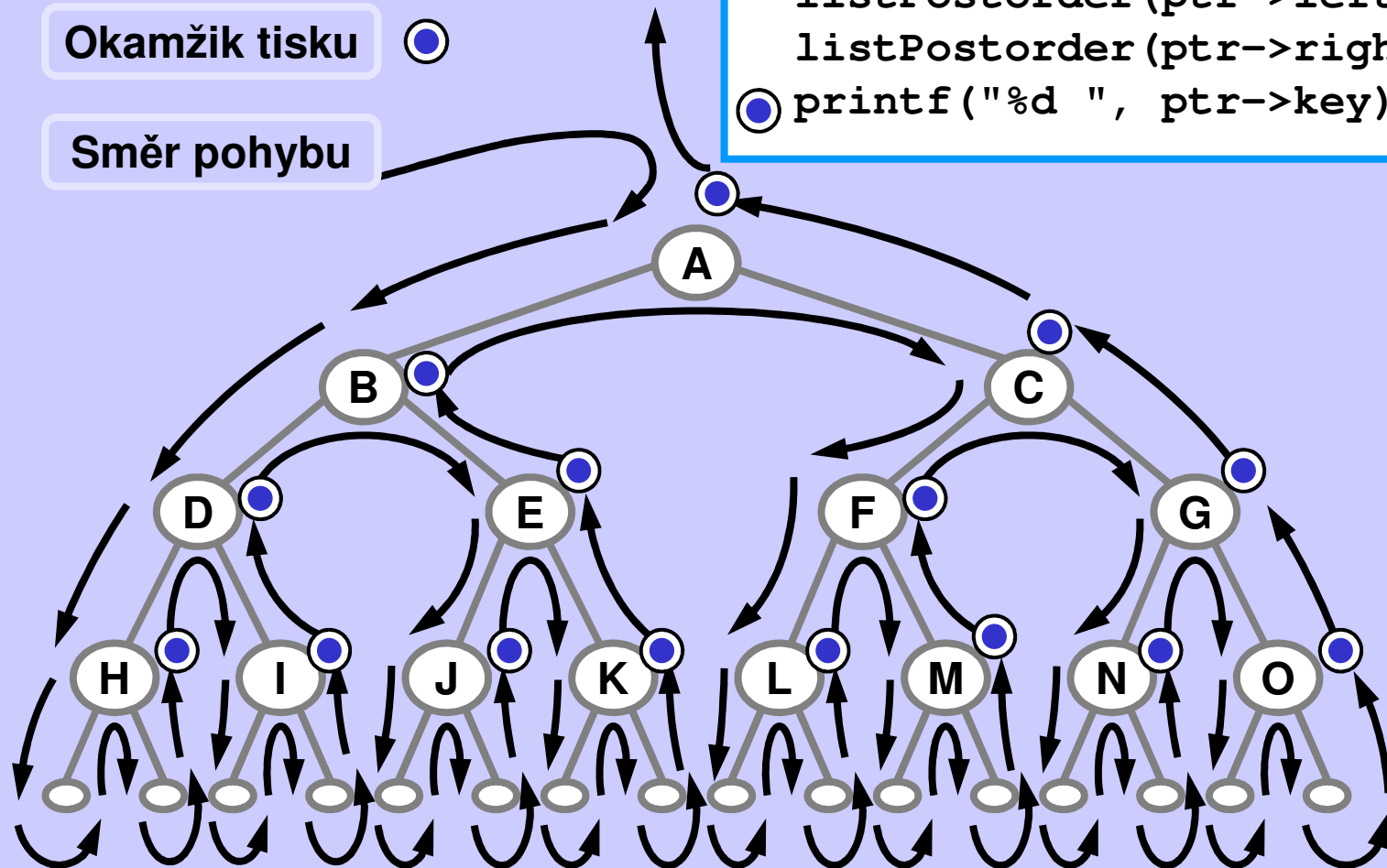
98 71 84 39 5 58 28 83 13

Pohyb ve stromu v průchodu Postorder

Okamžik tisku ○

Směr pohybu

```
listPostorder(ptr->left);
listPostorder(ptr->right);
printf("%d ", ptr->key);
```

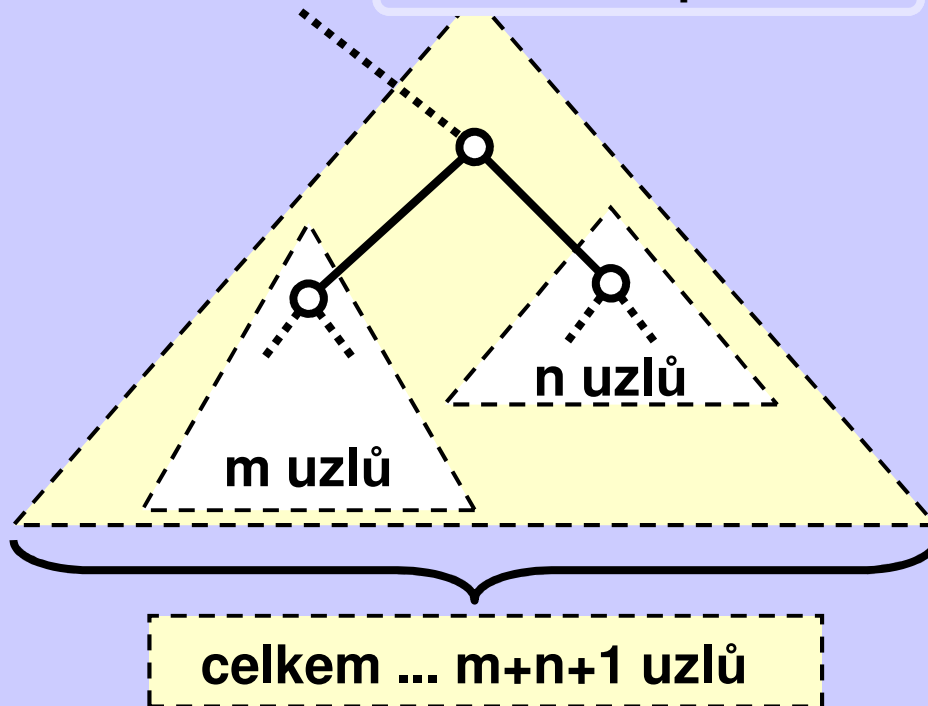


Výstup

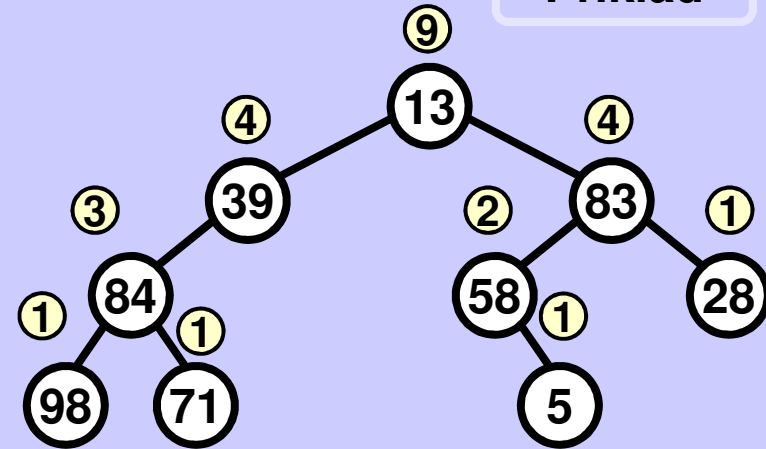
H I D J K E B L M F N O G C A

Velikost stromu (= počet uzlů) rekurzivně

Strom nebo podstrom



Příklad

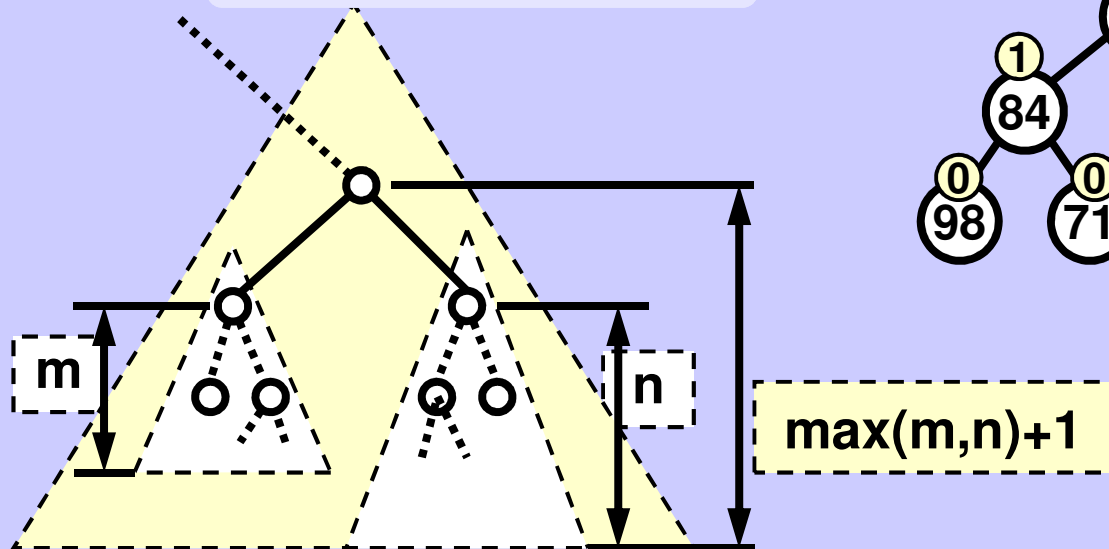


```

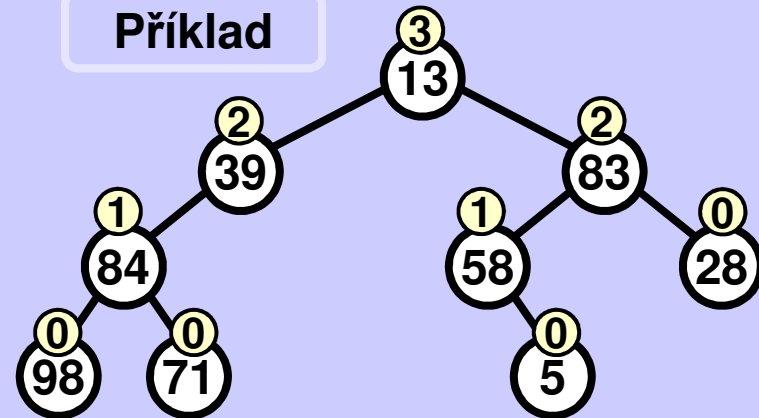
int count(NODE *ptr) {
    if (ptr == NULL) return 0;
    return (count(ptr->left) + count(ptr->right)+1);
}
  
```

Hloubka stromu (= max hloubka uzlu) rekurzivně

Strom nebo podstrom



Příklad



```

int depth(NODE *ptr) {
    if (ptr == NULL) return (-1);
    return ( max(depth(ptr->left), depth(ptr->right) )+1 );
}
  
```

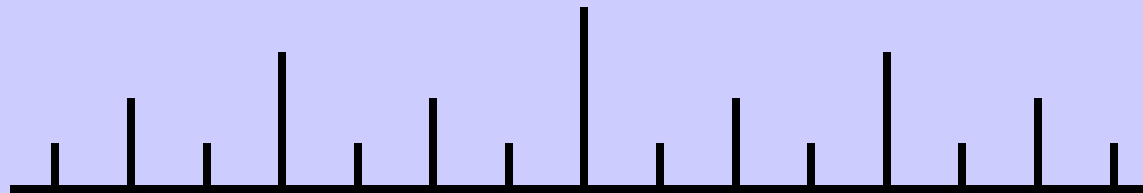
Jednoduchý příklad rekurze

Binární pravítka

Rysky pravítka

Délky rysek

Kód vypíše
délky rysek
pravítka



1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

```
void ruler(int val) {
    if (val < 1) return;

    ruler(val-1);
    print(val);
    ruler(val-1);
}
```

Call: ruler(4);

Domácí úkol: Ternárně!



Jednoduchý příklad rekurze

Binární pravítko vs. průchod inorder

Pravítko

```
void ruler(int val) {  
  if (val < 1) return;  
  
  ruler(val-1);  
  print(val);  
  ruler(val-1);  
}
```

Inorder

```
void listInorder( NODE *ptr) {  
  if (ptr == NULL) return;  
  
  listInorder(ptr->left);  
  printf("%d ", ptr->key);  
  listInorder(ptr->right);  
}
```

Strukturní podobnost, shoda!

Výstup pravítka

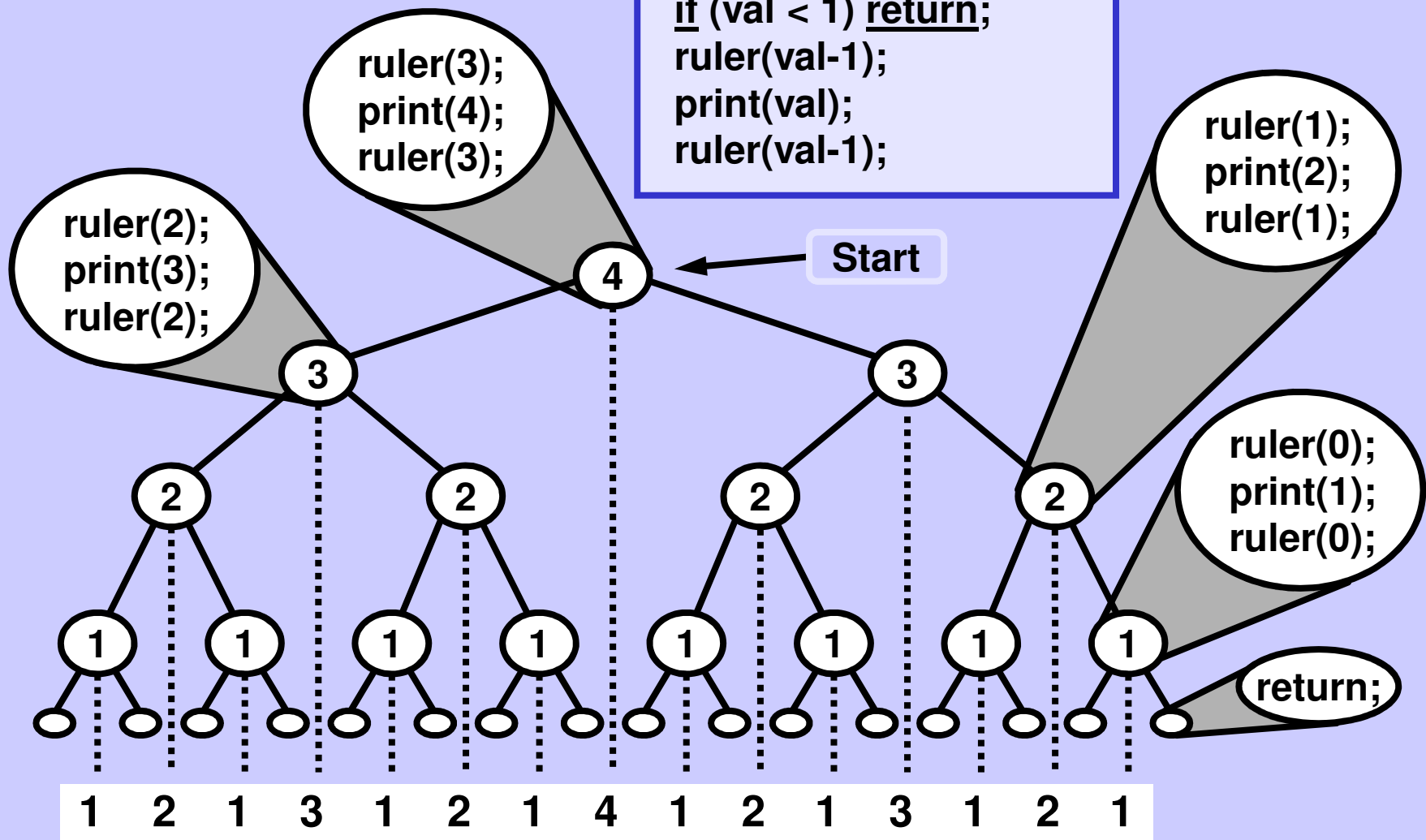
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Jednoduchý příklad rekurze

Činnost binárního pravítka

Kód

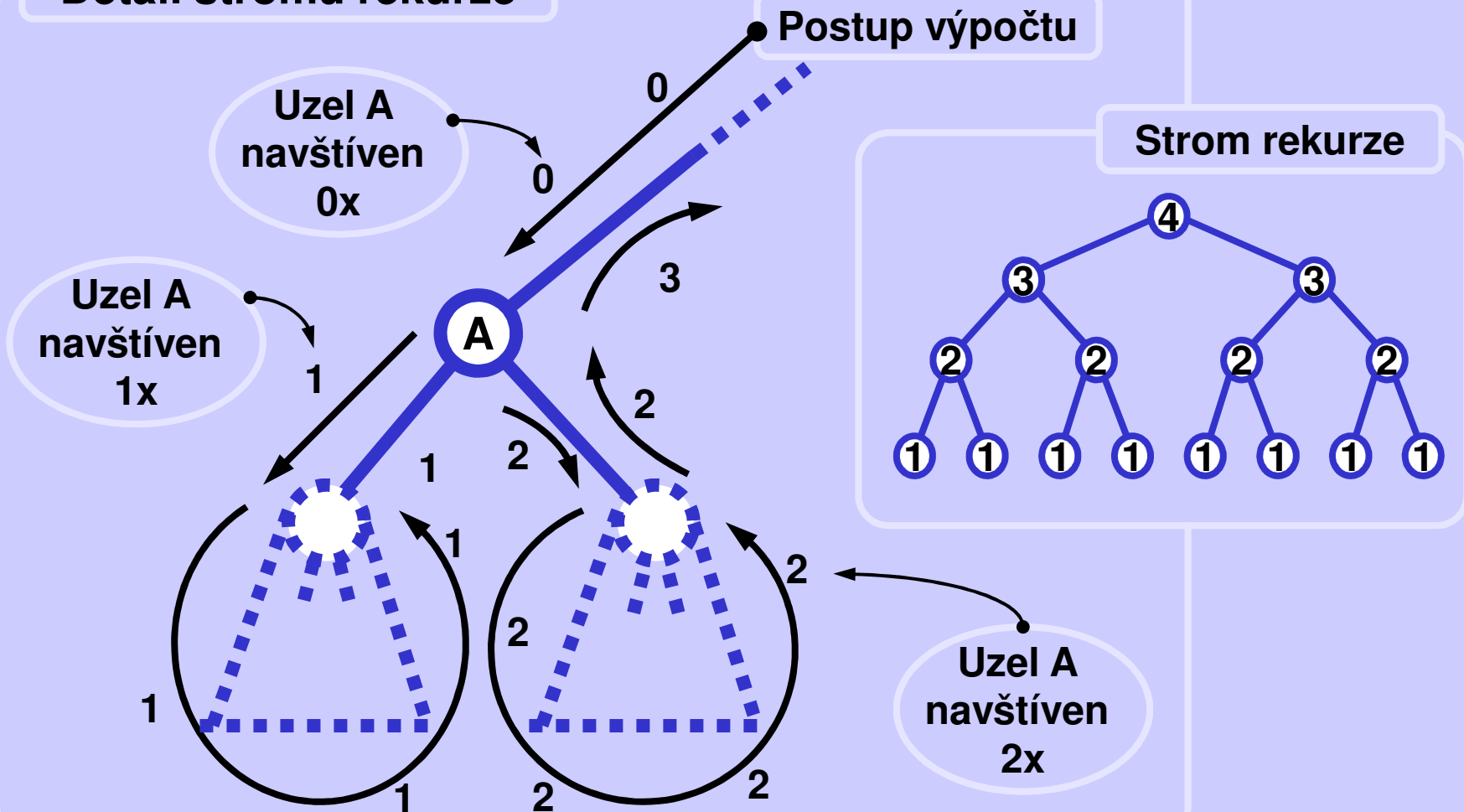
```
if (val < 1) return;
ruler(val-1);
print(val);
ruler(val-1);
```



Zásobník implementuje rekurzi

Binární pravítko bez rekurze

Detail stromu rekurze



Zásobník implementuje rekurzi

Standardní strategie

Při používání zásobníku:


Je-li to možné, zpracovávej jen data ze zásobníku.

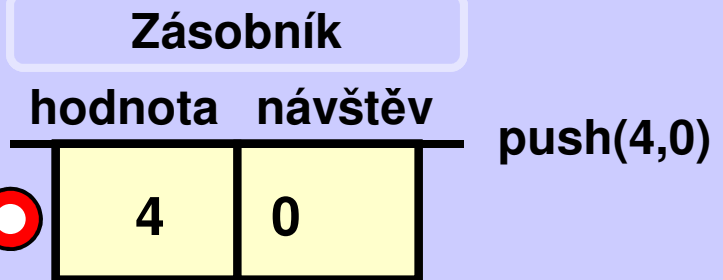
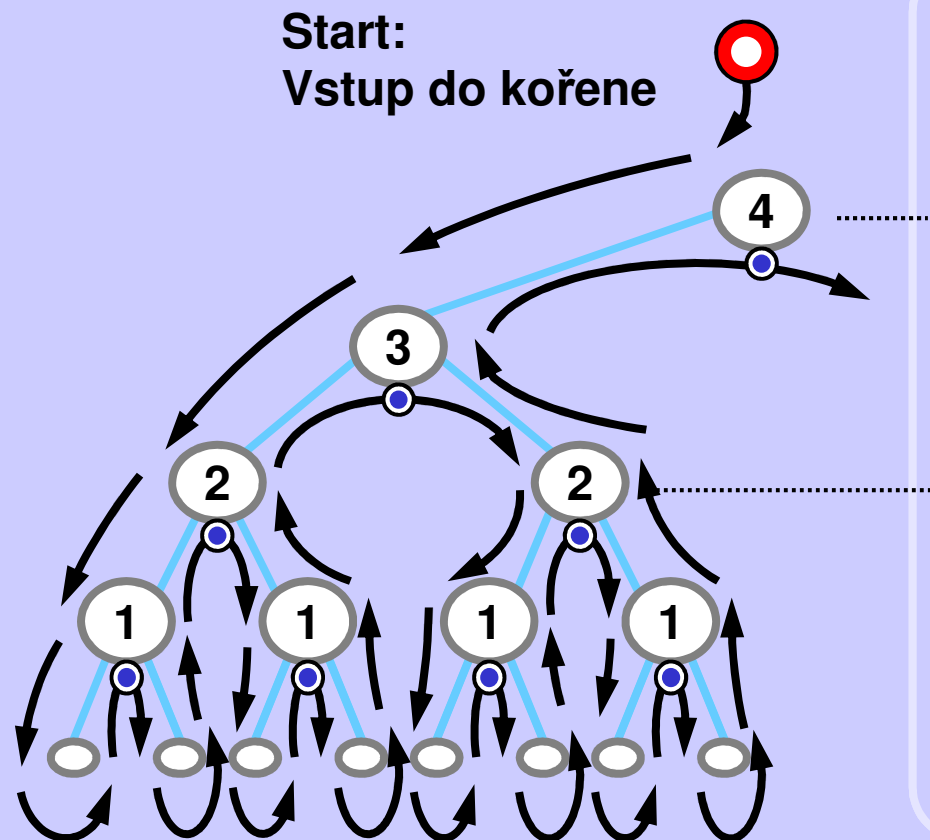
Standardní postup

**Ulož první uzel (první zpracovávaný prvek) do zásobníku.
Každý další uzel (zpracovávaný prvek) ulož také na zásobník.
Zpracovávej vždy pouze uzel na vrcholu zásobníku.
Když jsi s uzlem (prvkem) hotov, ze zásobníku ho odstraň.
Skonči, když je zásobník prázdný.**

Zásobník implementuje rekurzi

Každý záběr v následující sekvenci představí situaci PŘED zpracováním uzlu.

 Aktuální pozice

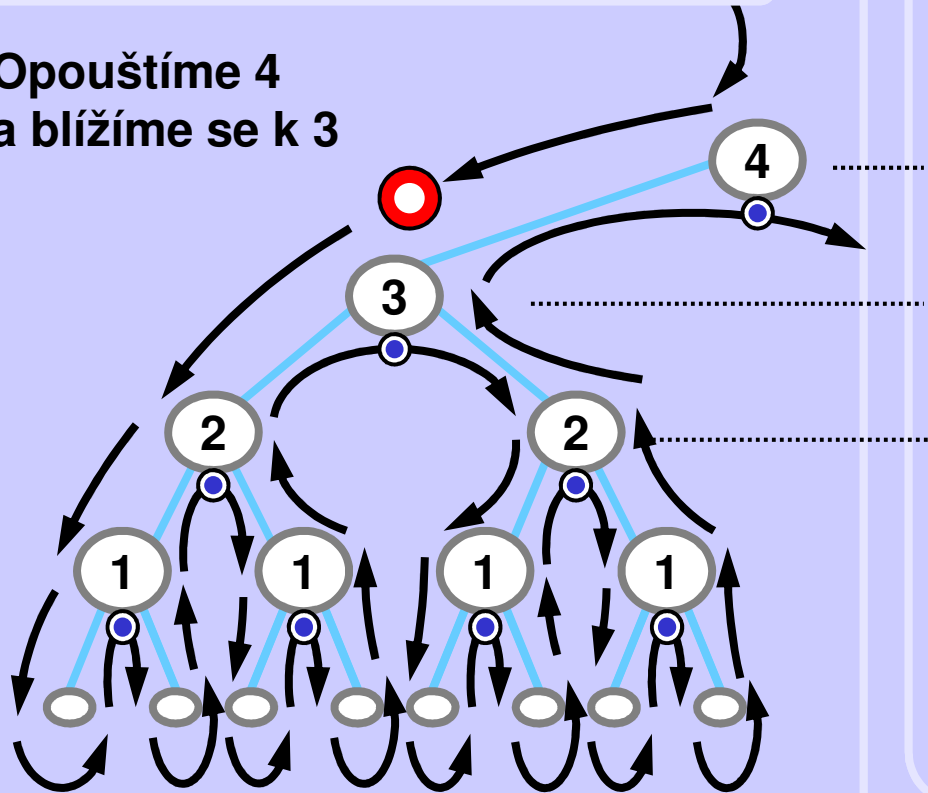


Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze

Opouštíme 4
a blížíme se k 3



Zásobník

hodnota návštěv

hodnota	návštěv
4	1
3	0

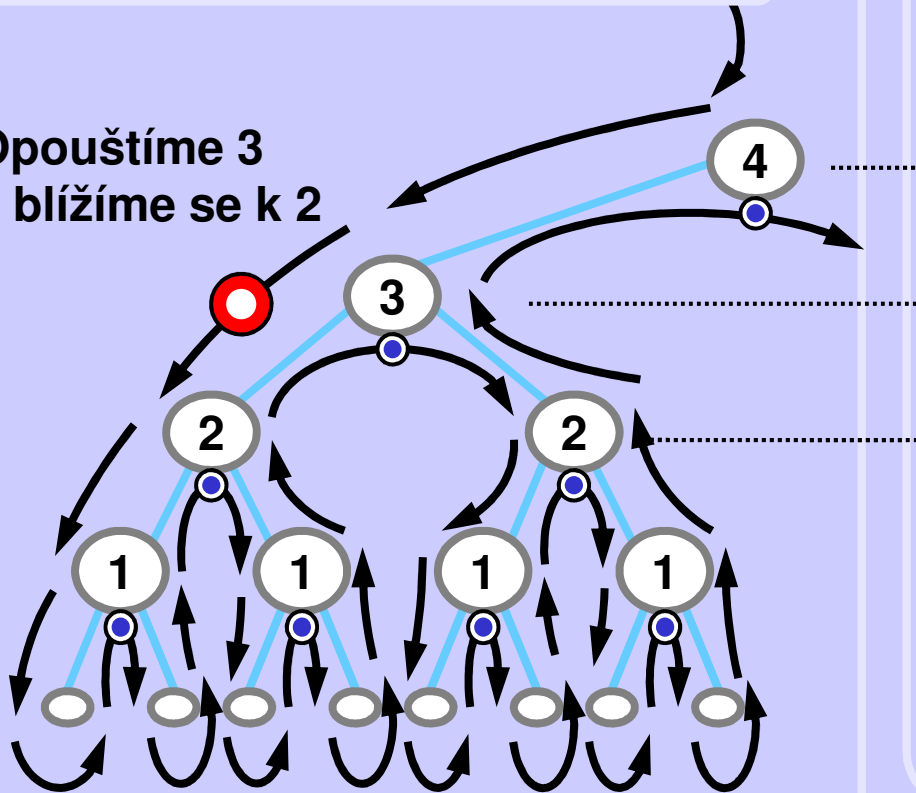
push(3,0)

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze

Opouštíme 3
a blížíme se k 2



Zásobník

hodnota návštěv

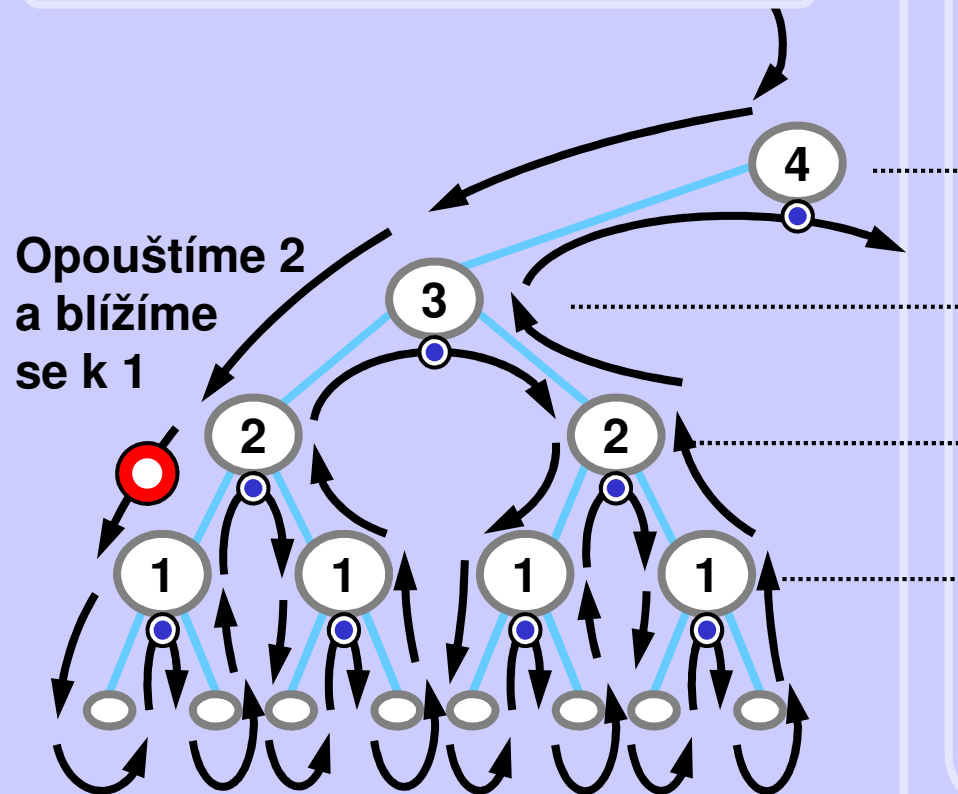
hodnota	návštěv
4	1
3	1
2	0

push(2,0)

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



Zásobník

hodnota návštěv

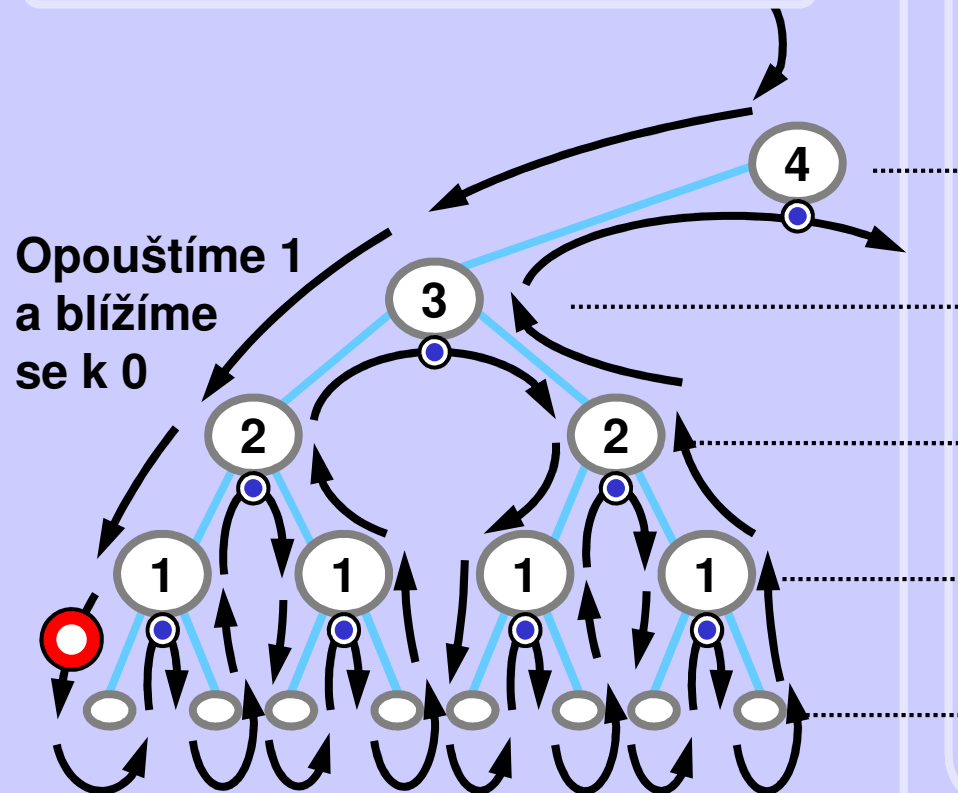
4	1
3	1
2	1
1	0

push(1,0)

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



Zásobník

hodnota návštěv

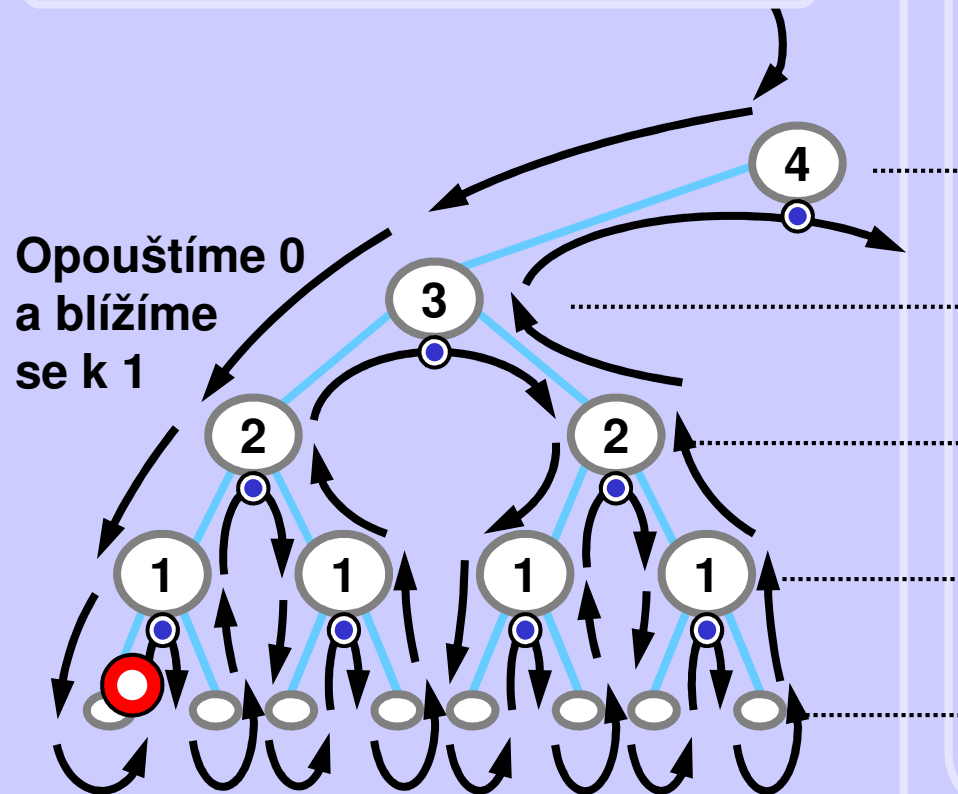
4	1
3	1
2	1
1	1
⋮	⋮
0	0

push(0,0)

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



Zásobník

hodnota návštěv

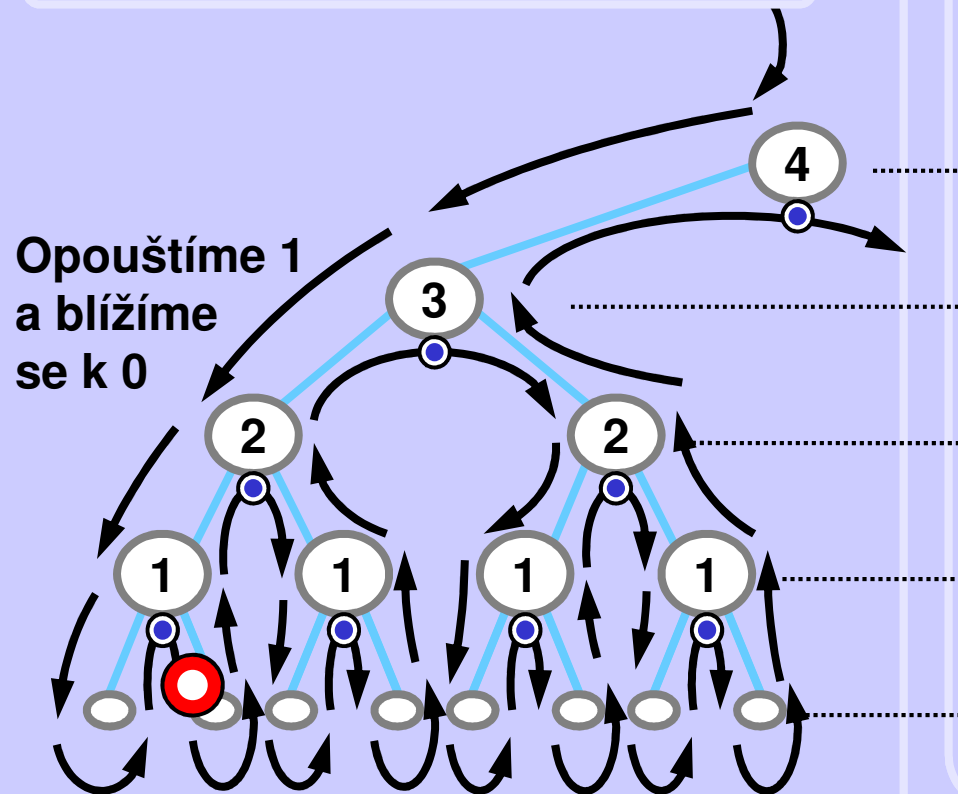
4	1
3	1
2	1
1	1
0	0

pop()

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



1

Zásobník

hodnota návštěv

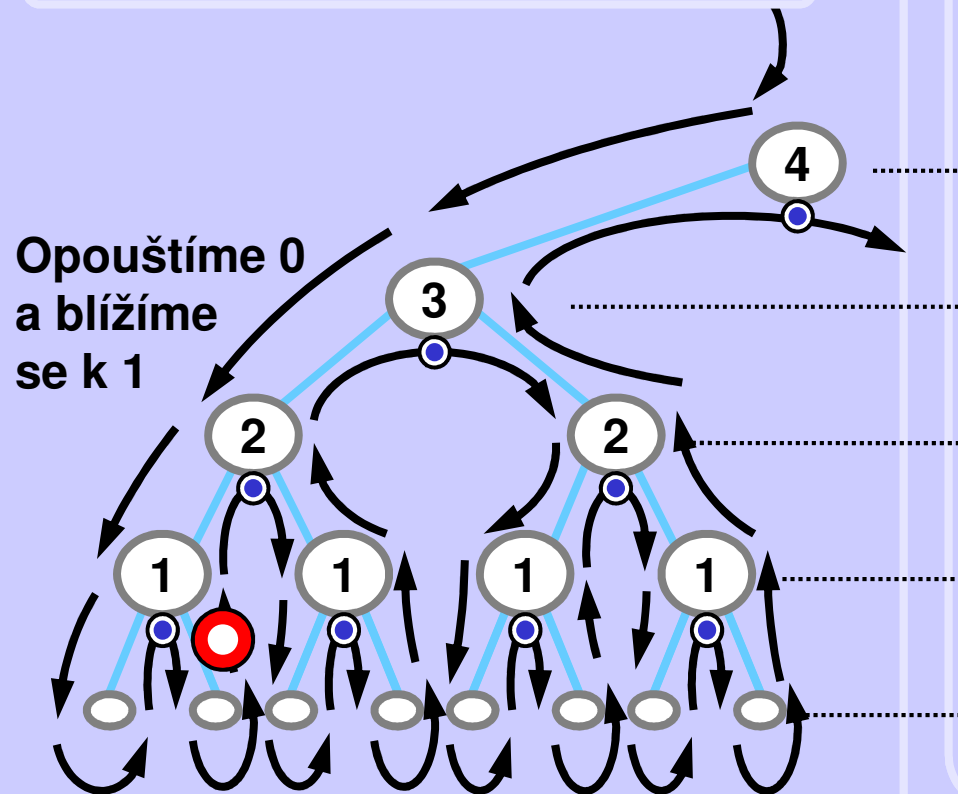
4	1
3	1
2	1
1	2
0	0

push(0,0)

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



Zásobník

hodnota návštěv

4	1
3	1
2	1
1	2
0	0

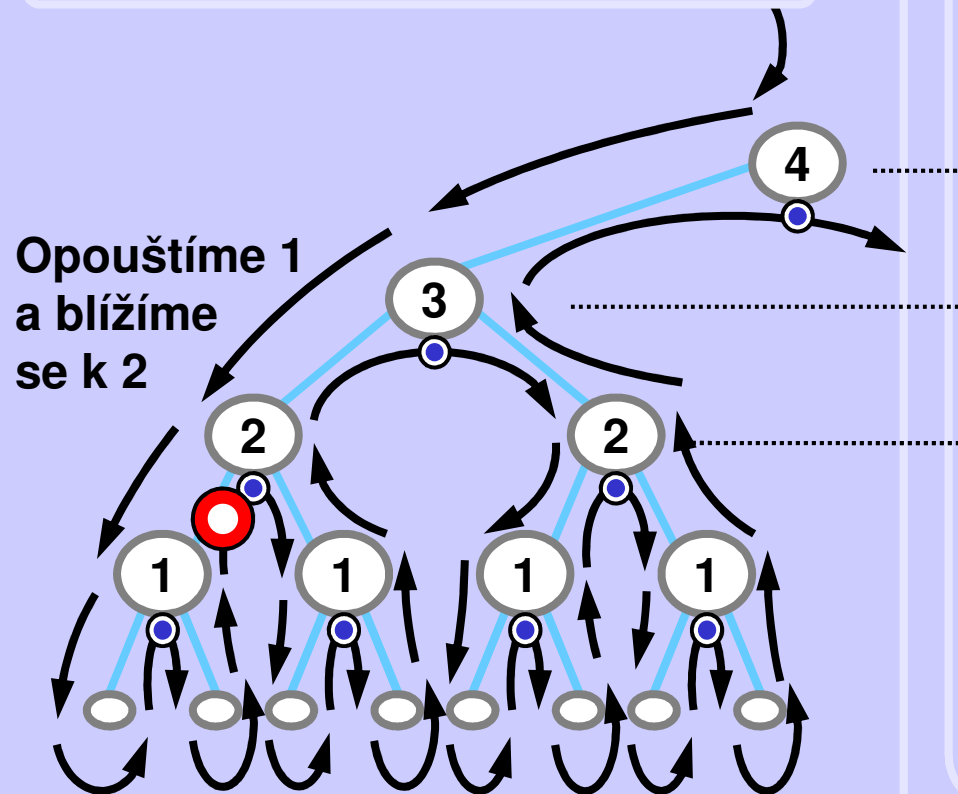
pop()

1

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



Zásobník

hodnota návštěv

4	1
3	1
2	1
1	2

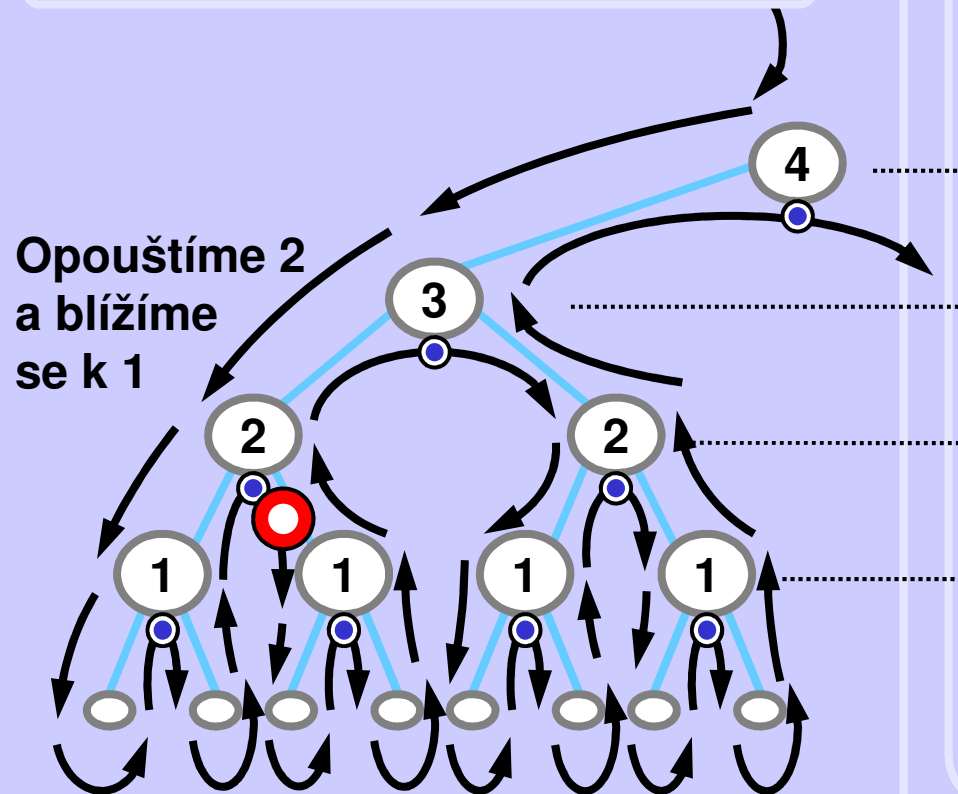
pop()

1

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



1 2

Zásobník

hodnota návštěv

4	1
3	1
2	2
⋮	
1	0

push(1,0)

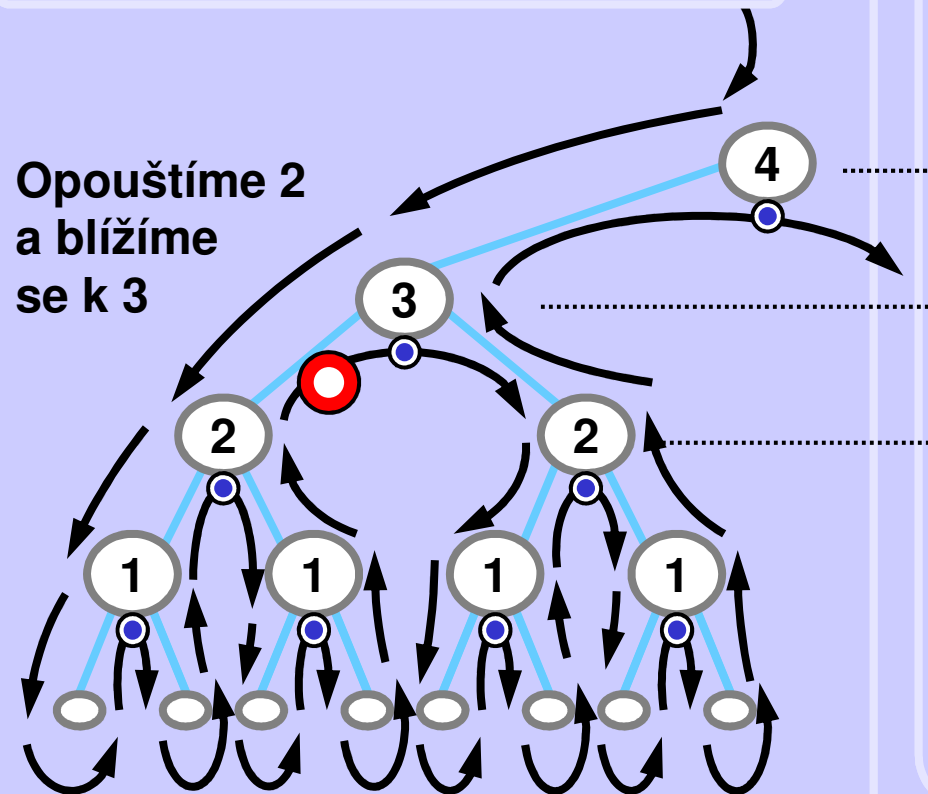
Výstup

atd...

Zásobník implementuje rekurzi

... po chvíli ...

Průchod stromem rekurze



1 2 1

Zásobník

hodnota návštěv

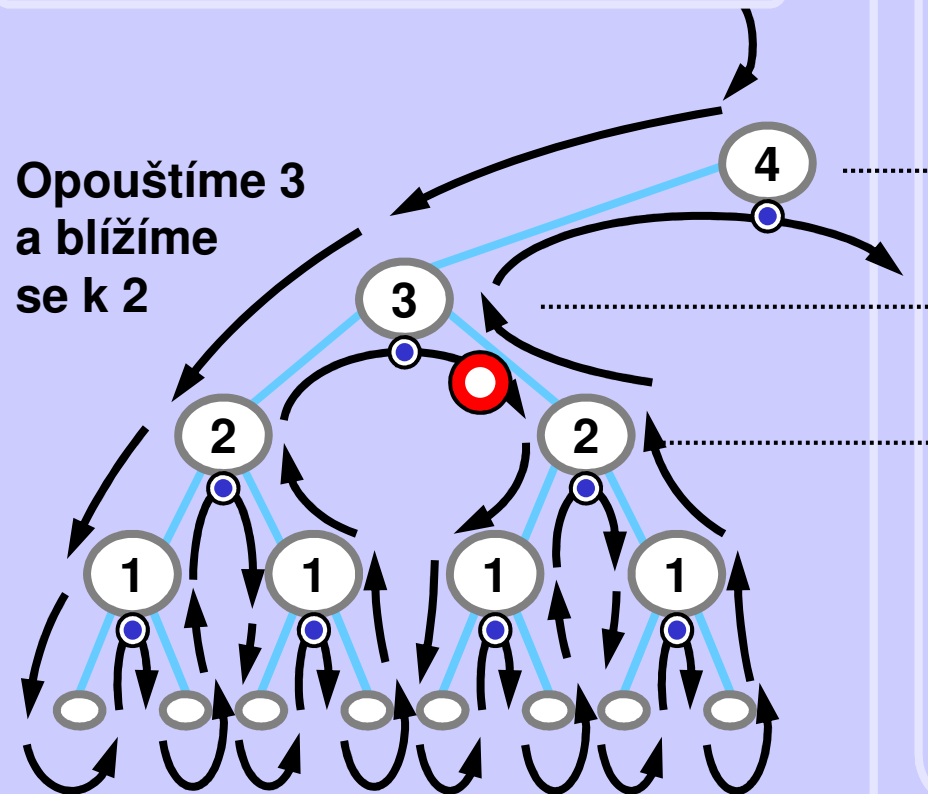
4	1
3	1
2	2

pop()

Výstup

Zásobník implementuje rekurzi

Průchod stromem rekurze



1 2 1 3

Zásobník

hodnota návštěv

4	1
3	2
2	0

push(2,0)

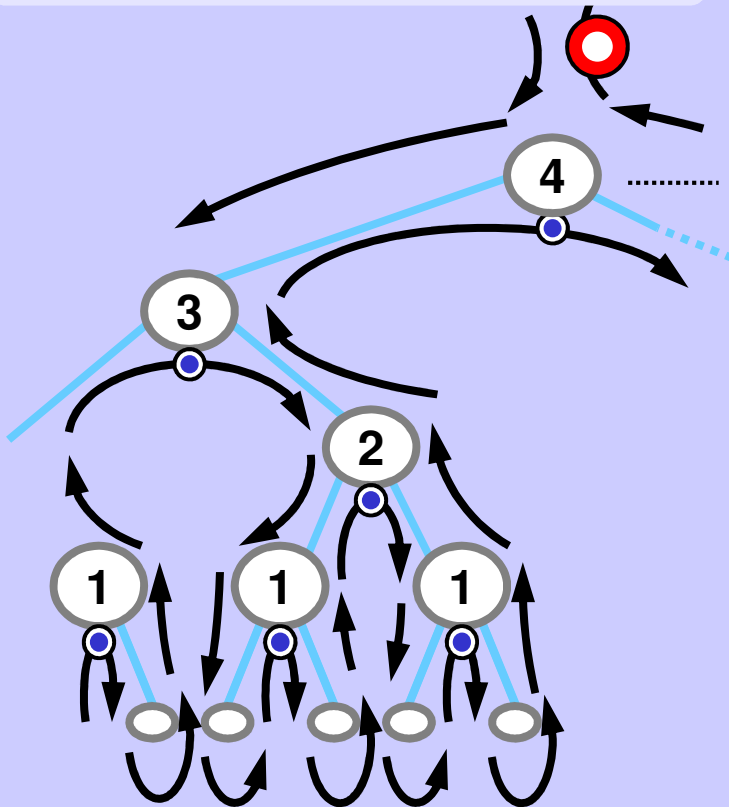
Výstup

atd...

Zásobník implementuje rekurzi

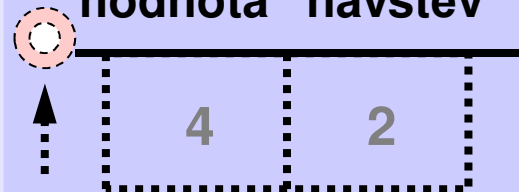
... a je hotovo

Průchod stromem rekurze



Zásobník

hodnota návštěv



(empty == true)

pop()

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Výstup

Zásobník implementuje rekurzi

Rekurzivní pravítko bez rekurzivního volání
Pseudokód (skoro kód :-)) pro objektový přístup

```
stack.init();
stack.top.value = N; stack.top.visits = 0;
while (!stack.empty()) {
    if (stack.top.value == 0) stack.pop();
    if (stack.top.visits == 0) {
        stack.top.visits++;
        stack.push(stack.top.value-1, 0);
    }
    if (stack.top.visits == 1) {
        print(stack.top.value);
        stack.top.visits++;
        stack.push(stack.top.value-1, 0);
    }
    if (stack.top.visits == 2) stack.pop();
}
```

Rekurzivní pravítko
bez rekurzivního volání,
jednoduchá implementace polem

Zásobník implementuje rekurzi

```

int stackVal[10]; int stackVis[10];
void ruler2(int N) {
    int SP = 0; // stack pointer
    stackVal[SP] = N; stackVis[SP] = 0; // init
    while (SP >= 0) { // while unempty
        if (stackVal[SP] == 0) SP--; // pop: in leaf
        if (stackVis[SP] == 0) { // first visit
            stackVis[SP]++; SP++;
            stackVal[SP] = stackVal[SP-1]-1; // go left
            stackVis[SP] = 0;
        }
        if (stackVis[SP] == 1) { // second visit
            printf("%d ", stackVal[SP]); // process the node
            stackVis[SP]++; SP++;
            stackVal[SP] = stackVal[SP-1]-1; // go right
            stackVis[SP] = 0;
        }
        if (stackVis[SP] == 2) SP--; // pop: node done
    }
}

```


Rekurzivní pravítko
bez rekurzivního volání,
jednoduchá implementace polem

Zásobník implementuje rekurzi

Poněkud kompaktnější kód

```

int stackVal[10]; int stackVis[10];

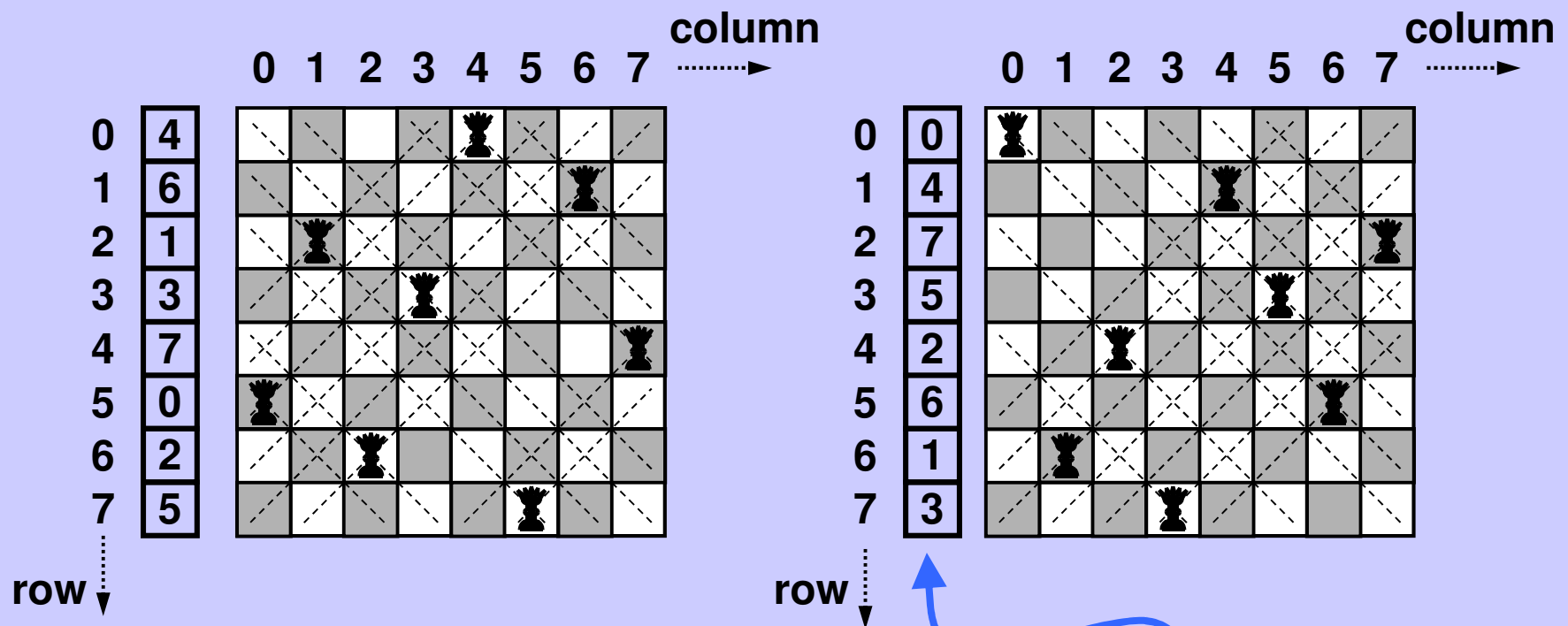
void ruler2(int N) {
    int SP = 0; // stack pointer
    stackVal[SP] = N; stackVis[SP] = 0; // init
    while (SP >= 0) { // while unempty
        if (stackVal[SP] == 0) SP--; // pop: in leaf
        if (stackVis[SP] == 2) SP--; // pop: node done
        if (stackVis[SP] == 1) // if second visit
            printf("%d ", stackVal[SP]); // process the node
        stackVis[SP]++; SP++; // otherwise
        stackVal[SP] = stackVal[SP-1]-1; // go deeper
        stackVis[SP] = 0;
    } }

```

Snadné prohledávání s návratem (Backtrack)

Problém osmi dam na šachovnici

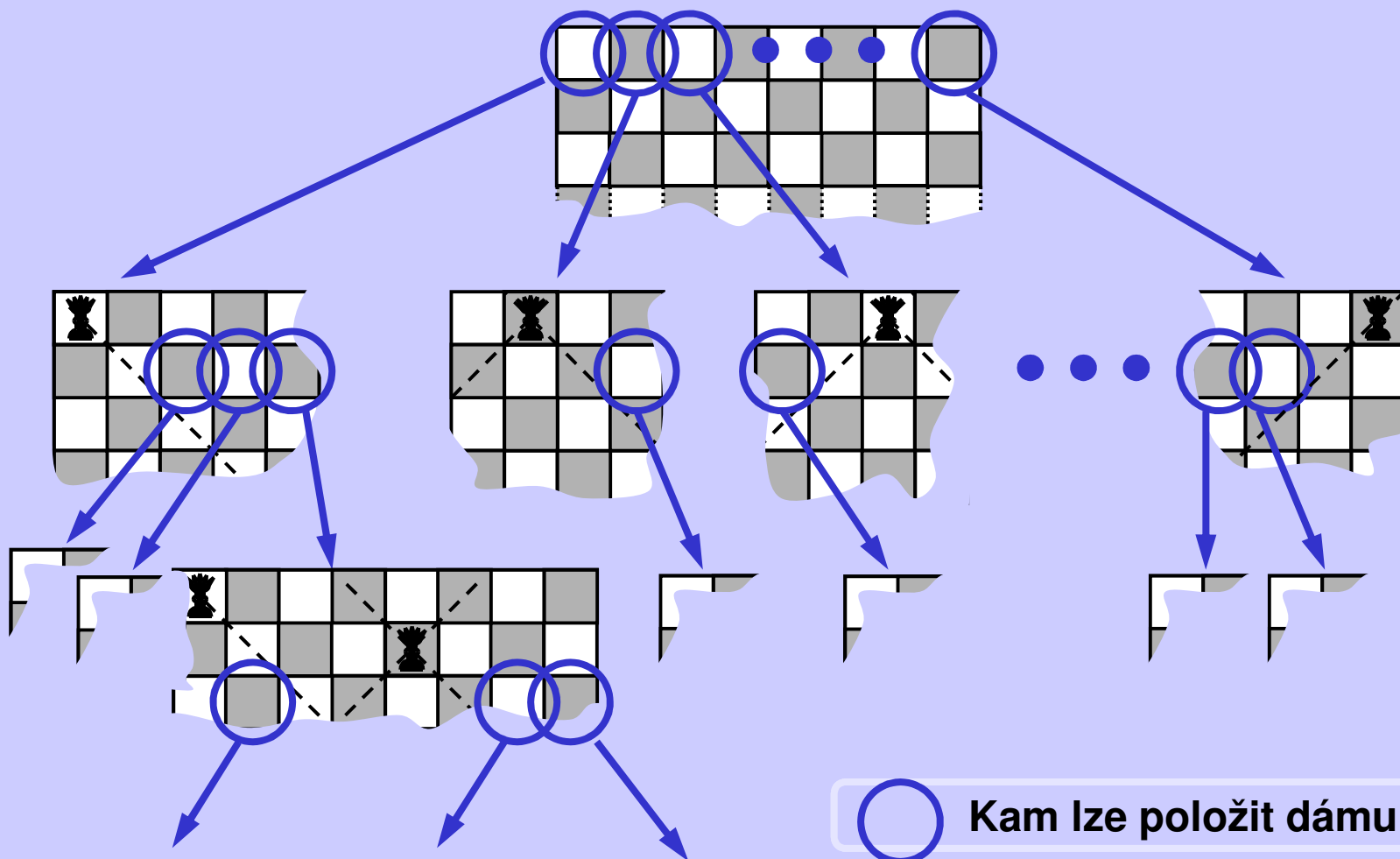
Některá řešení



Jediná datová struktura: pole `queenCol[]` (viz kód)

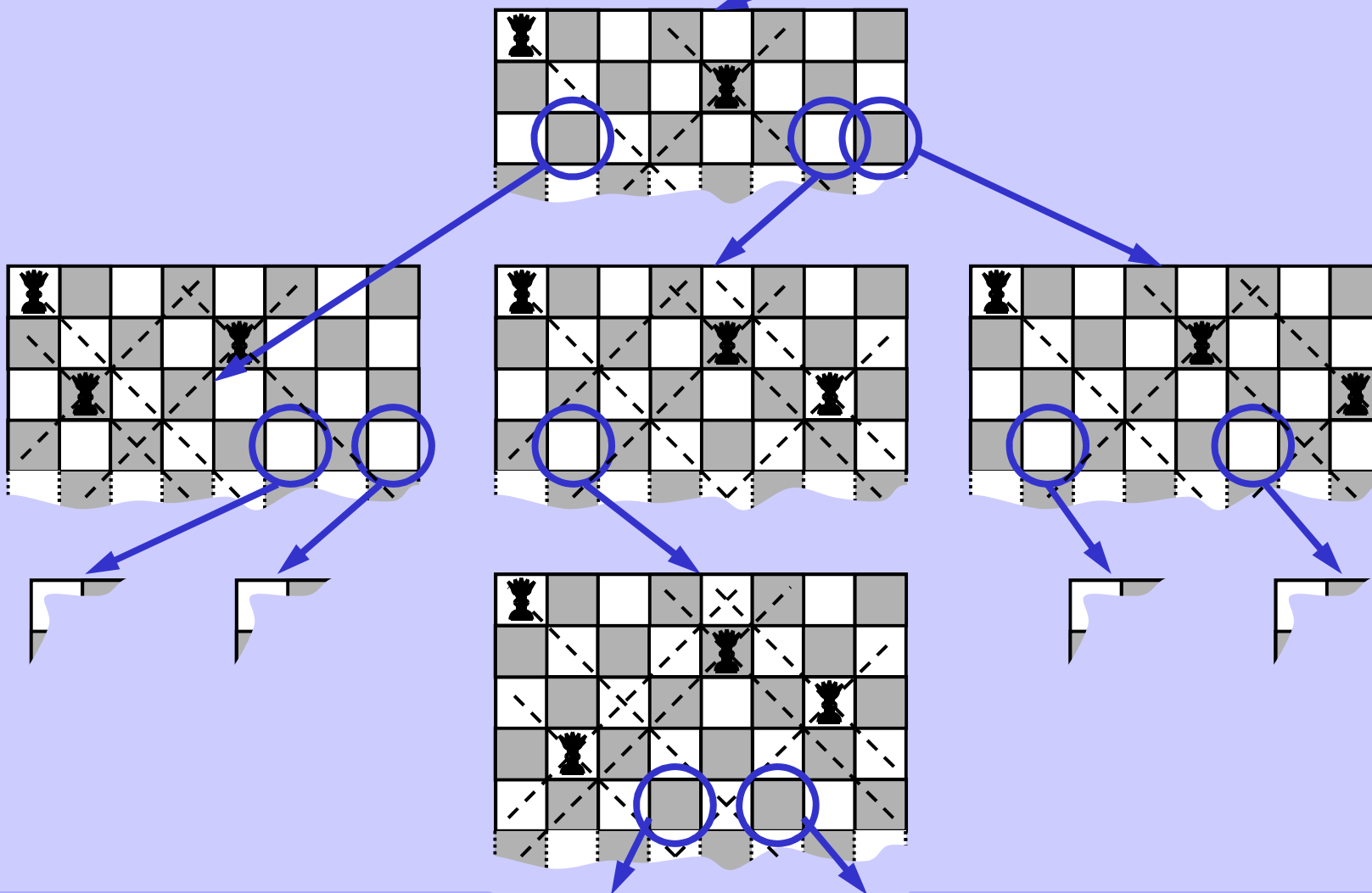
Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (kořen a několik potomků)



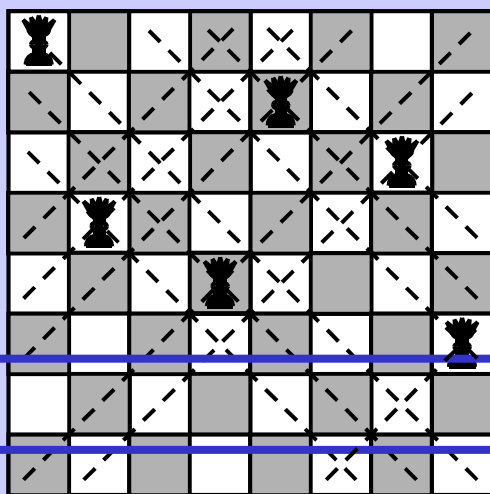
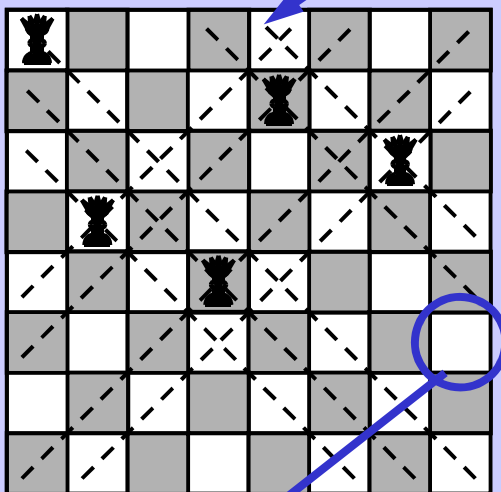
Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (výřez)

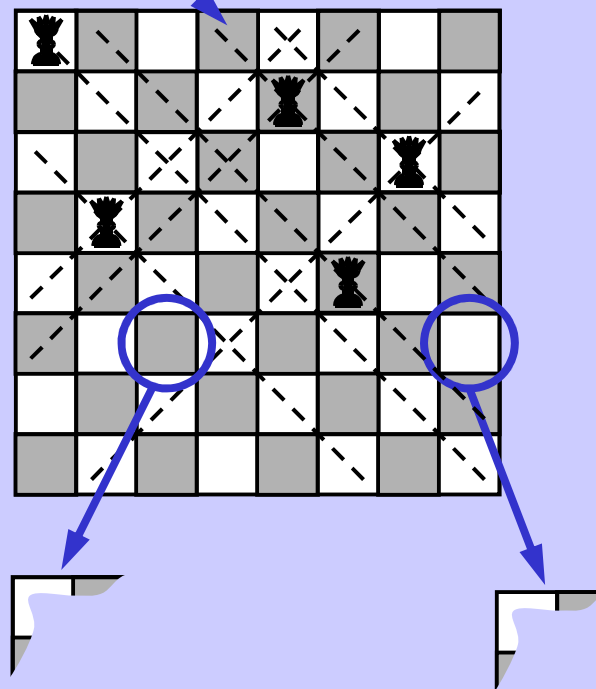


Snadné prohledávání s návratem (Backtrack)

Strom testovaných pozic (výřez)



Stop!



Snadné prohledávání s návratem (Backtrack)

N dam na šachovnici N x N

N poč. dam	Počet řešení	Počet testovaných pozic dámy		Zrychlení
		Hrubá síla (N^N)	Backtrack	
4	2	256	240	1.07
5	10	3 125	1 100	2.84
6	4	46 656	5 364	8.70
7	40	823 543	25 088	32.83
8	92	16 777 216	125 760	133.41
9	352	387 420 489	651 402	594.75
10	724	10 000 000 000	3 481 500	2 872.33
11	2 680	285 311 670 611	19 873 766	14 356.20
12	14 200	8 916 100 448 256	121 246 416	73 537.00

Tab 3.1 Rychlosti řešení problému osmi dam

Snadné prohledávání s návratem, 8 dam (Backtrack)

```

boolean positionOK(int r, int c) {           // r: row, c: column
    for (int i = 0; i < r; i++)
        if ((queenCol[i] == c) ||         // same column or
            (abs(r-i) == abs(queenCol[i]-c))) // same diagonal
            return false;
    return true;
}

```

```

void putQueen(int row, int col) {
    queenCol[row] = col;                 // put a queen there
    if (++row == N)                       // if solved
        print(queenCol);                 // output solution
    else
        for (col = 0; col < N; col++)    // test all columns
            if (positionOK(row, col))    // if free
                putQueen(row, col);      // next row recursion
}

```

```

Call: for (int col = 0; col < 8; col++)
        putQueen(0, col);

```