ALG 04

Zásobník

Fronta

- **Operace Enqueue, Dequeue, Front, Empty....**
- **Cyklická implementace fronty**

Průchod stromem do šířky

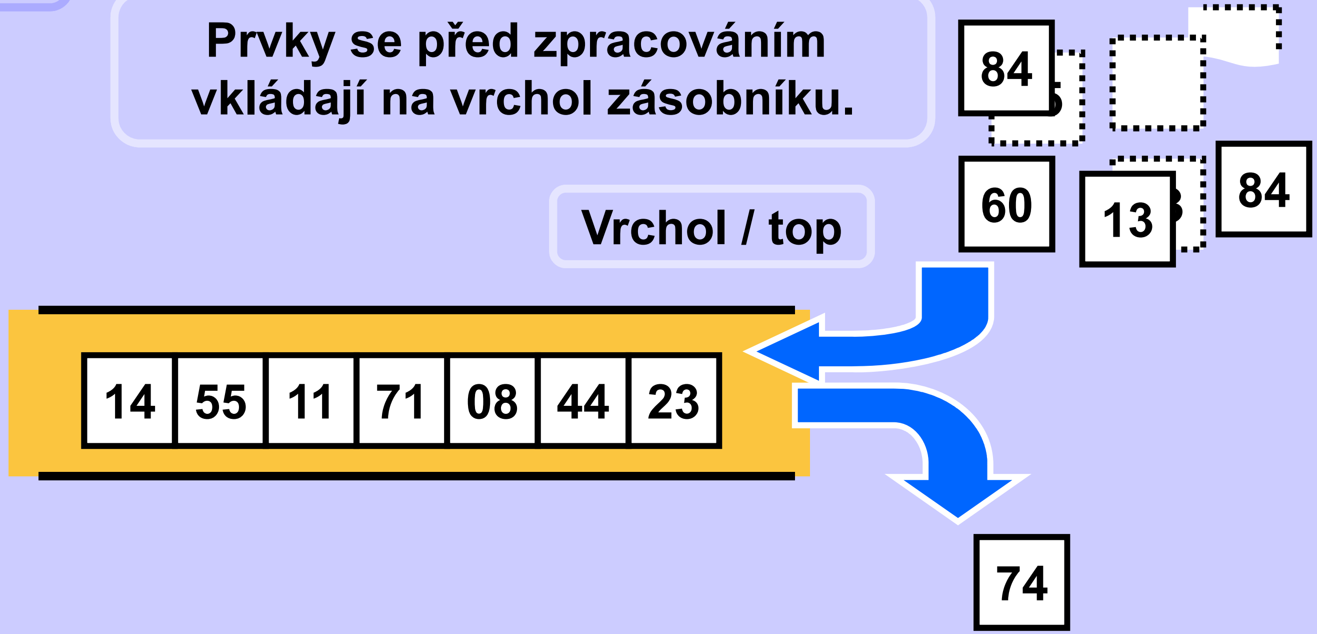
Grafy

- **průchod grafem do šířky**
- **průchod grafem do hloubky**

Ořezávání a heuristiky

Zásobník / stack

Prvky se před zpracováním vkládají na vrchol zásobníku.

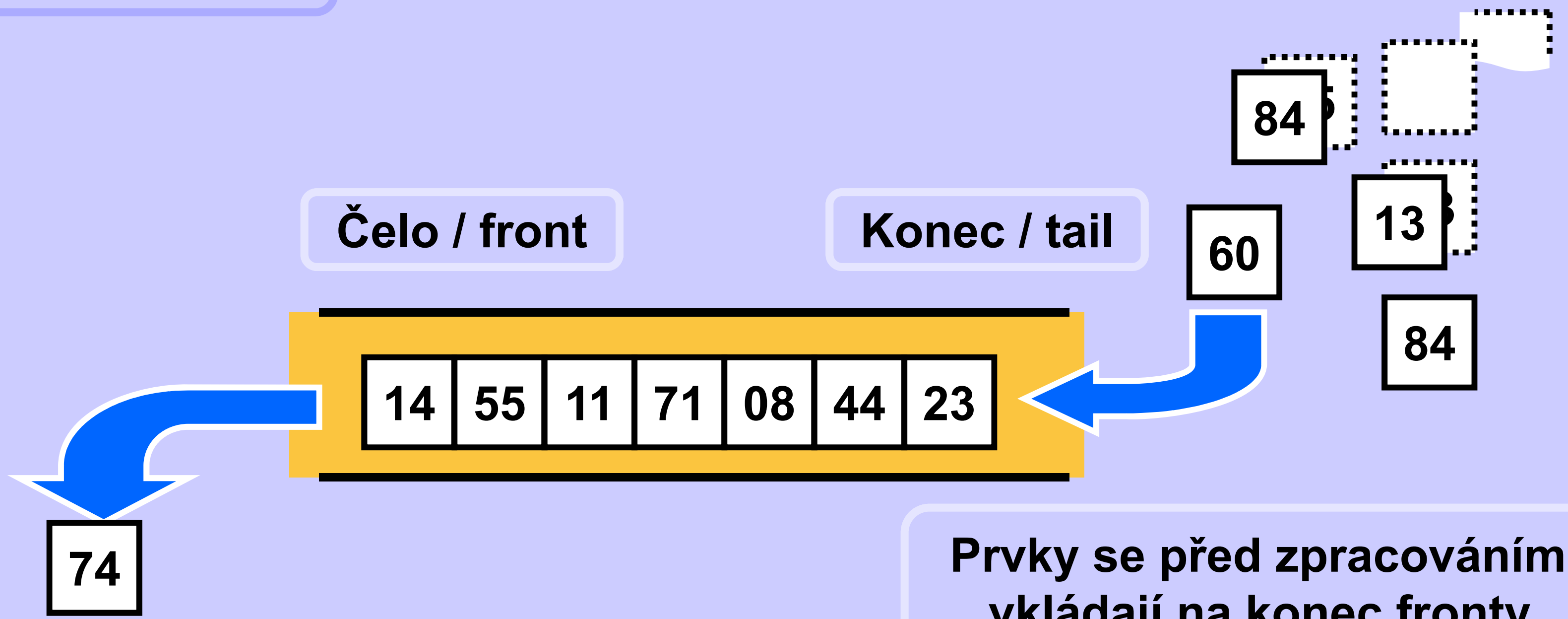


Prvky se odebírají z vrcholu zásobníku a pak se zpracovávají.

Operace

Vlož na vrchol	Push
Odeber z vrcholu	Pop
Čti začátek	Top
Je prázdný?	Empty

Fronta / queue



Prvky se před zpracováním vkládají na konec fronty.

Prvky se odebírají z čela fronty a pak se zpracovávají.

Operace

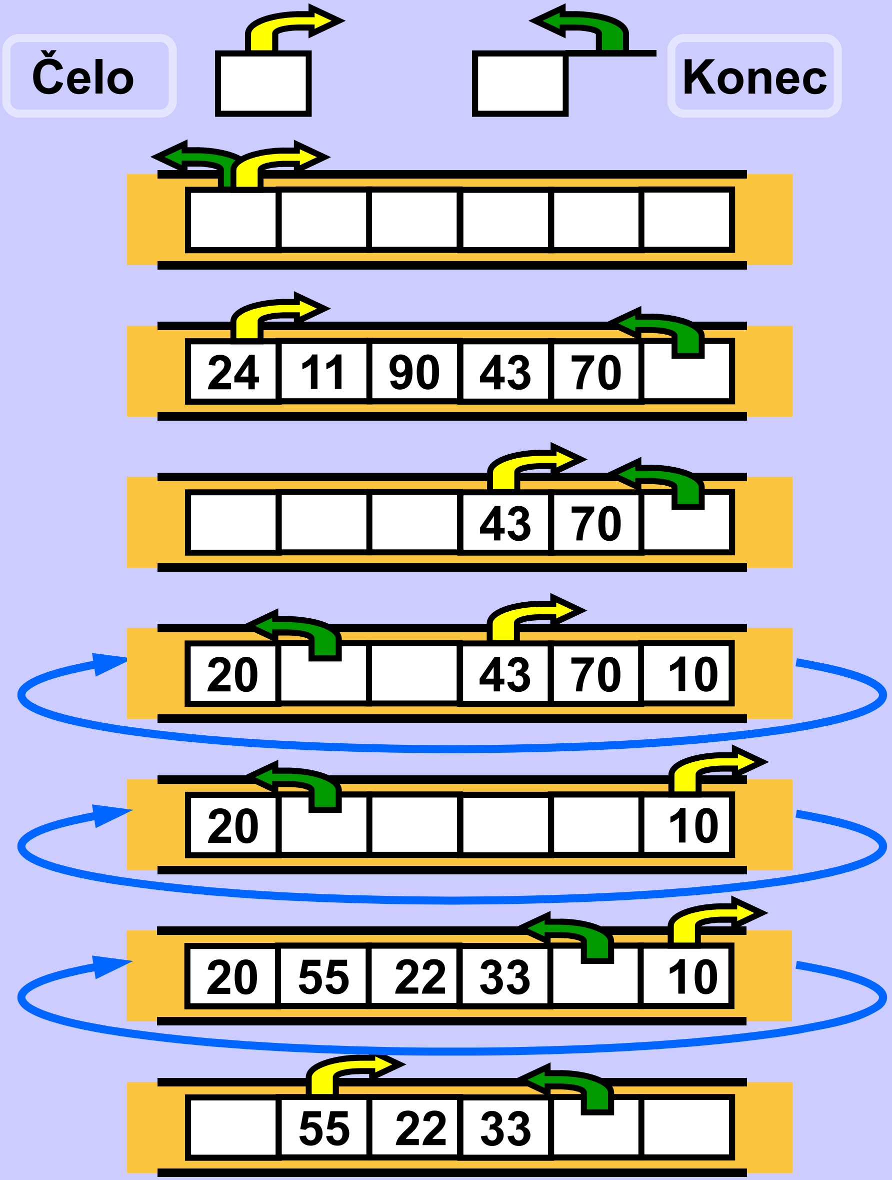
Vlož na konec	Enqueue / InsertLast / Push ...
Odeber ze začátku	Dequeue / delFront / Pop ...
Čti začátek	Front / Peek ...
Je prázdná?	Empty

Fronta

Jednoduchý příklad života fronty

	Čelo			Konec
Prázdná				
Vlož(24)	24			
Vlož(11)	24	11		
Vlož(90)	24	11	90	
Odeber()	11	90		
Vlož(43)	11	90	43	
Odeber()	90	43		
Odeber()	43			
Vlož(79)	43	79		

Cyklická implementace fronty polem



Prázdná fronta
v poli pevné délky

Vlož 24, 11, 90, 43, 70.

Odeber, odeber, odeber.

Vlož 10, 20.

Odeber, odeber.

Vlož 55, 22, 33.

Odeber, odeber.

Cyklická implementace fronty polem

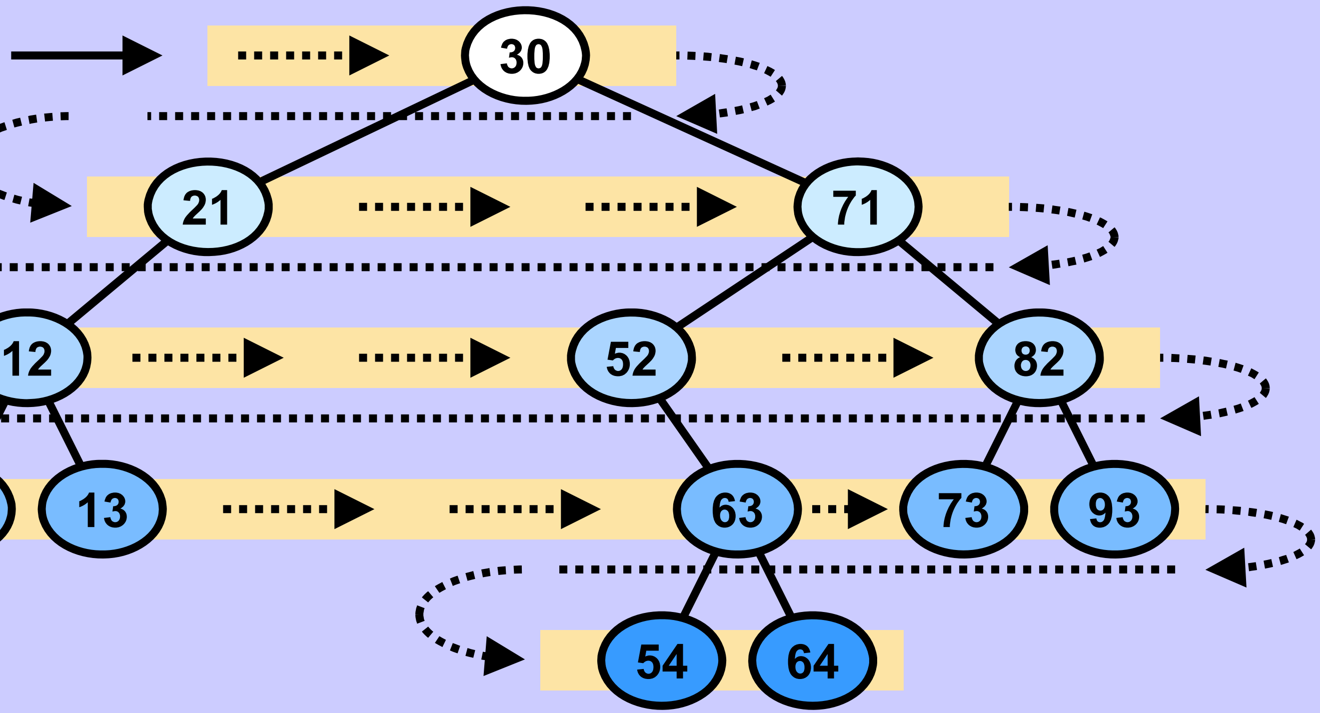
Index/ukazatel konce fronty ukazuje na první volnou pozici za posledním prvkem fronty. Index/ukazatel čela fronty ukazuje na první obsazenou pozici. Pokud oba ukazují tamtéž, fronta je prázdná.

```
class Queue {
  Node q [];
  int size;
  int front;
  int tail;

 Queue( int qsize ){
   size = qsize;
   q = new Node[size];
   front = 0;
   tail = 0;
   }

 boolean Empty() {
   return ( tail==front );
 }
```

```
void Enqueue( Node node ){
  if( (tail+1 == front) ||
      (tail-front == size-1) )
      ... // queue full, fix it

  q[t++] = node;
  if( tail==size ) tail = 0;
}

Node Dequeue() {
  Node n = q[front++];
  if( front == size ) front = 0;
    return n;
}

} // end of Queue
```

Průchod stromem do šířky

Strom s naznačeným směrem průchodu



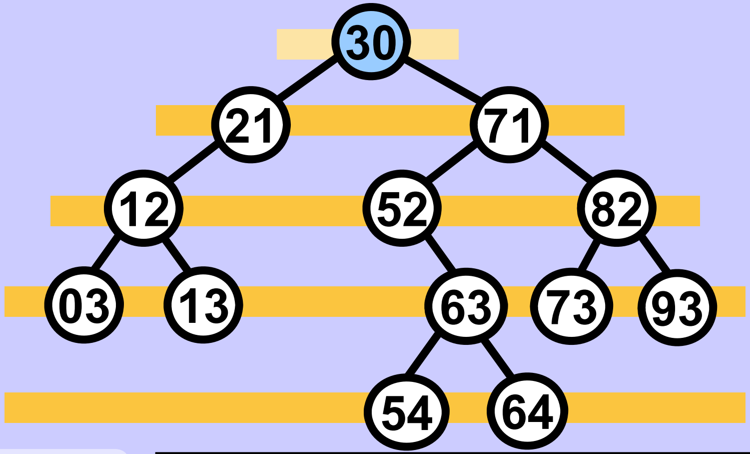
Pořadí uzlů

30	21	71	12	52	82	03	13	63	73	93	54	64

Struktura stromu ani rekurzivní přístup tento průchod nepodporují.

Průchod stromem do šířky

Inicializace



Výstup

2.

Vytvoř prázdnou frontu

Do fronty vlož kořen stromu

Čelo

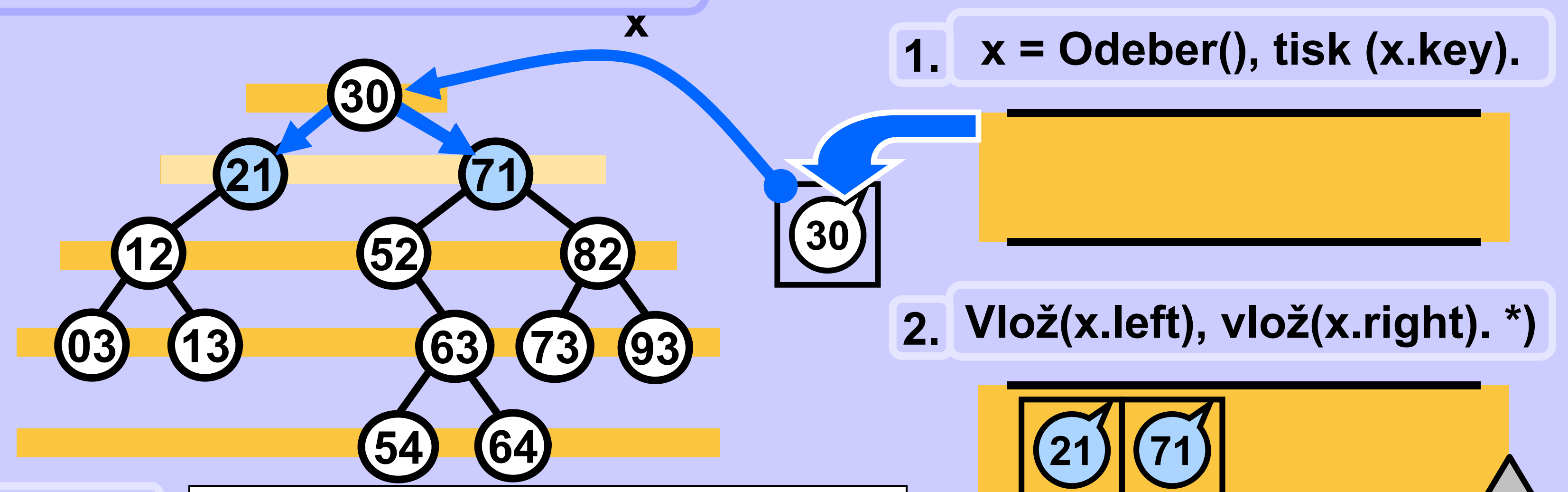
Konec

Hlavní cyklus

Dokud není fronta prázdná, opakuj:

1. Odeber první uzel z fronty a zpracuj ho.
2. Do fronty vlož jeho potomky, pokud existují.

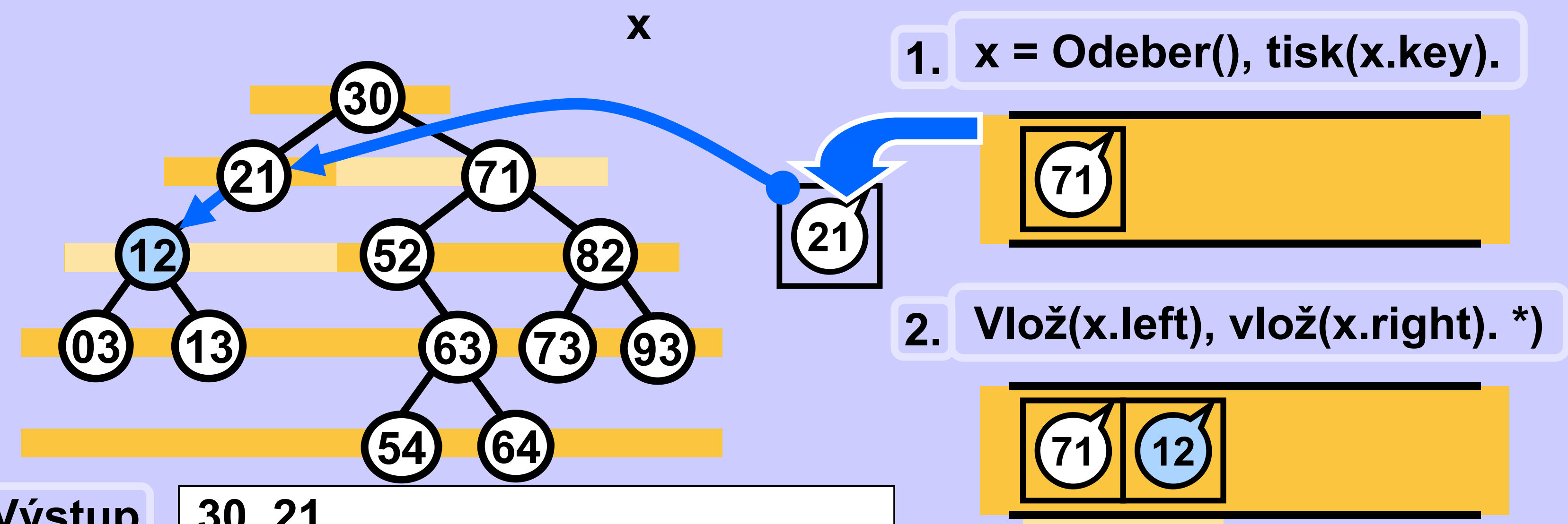
Průchod stromem do šířky



1. x = Odeber(), tisk (x.key).

2. Vlož(x.left), vlož(x.right). *)

Výstup 30



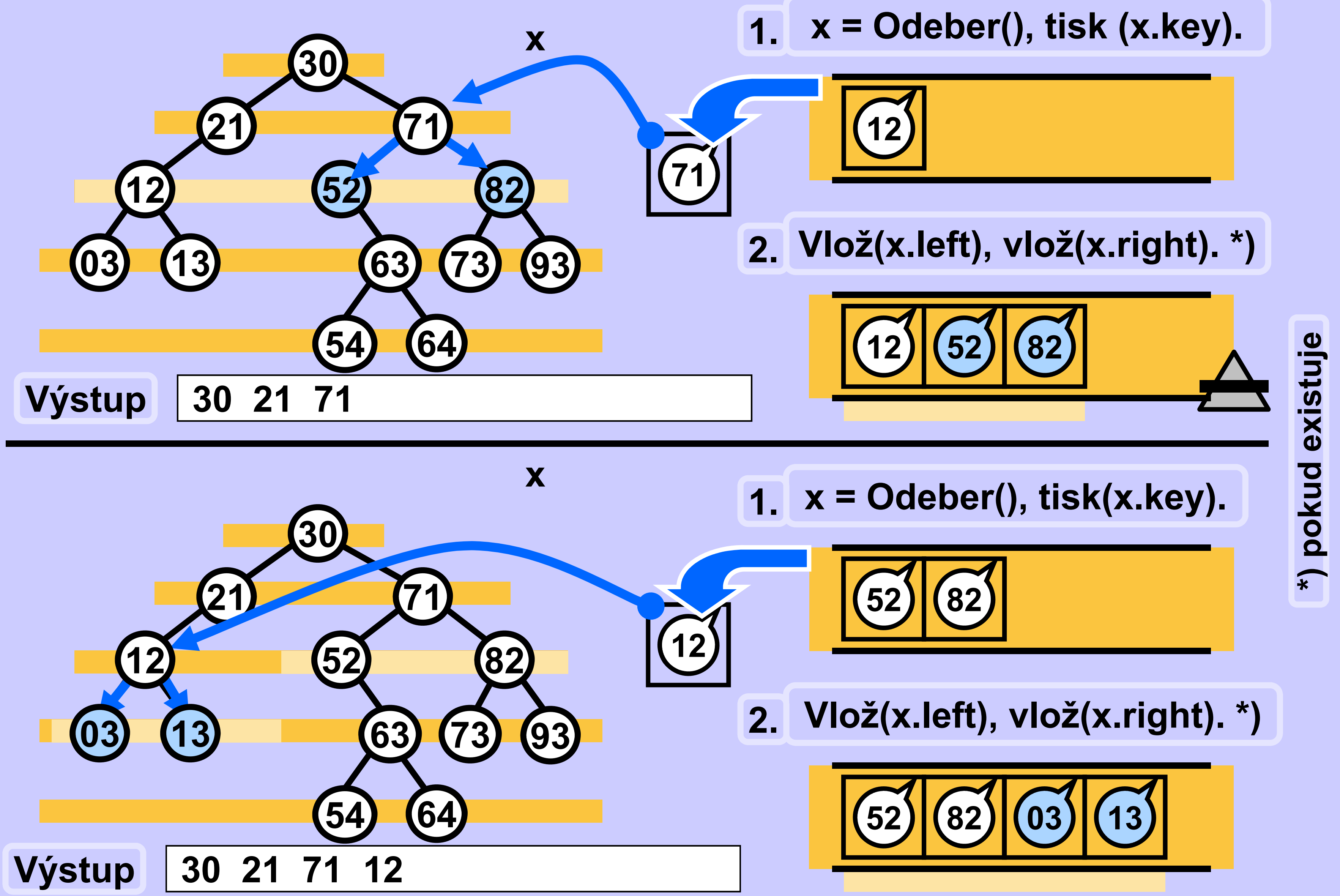
1. x = Odeber(), tisk(x.key).

2. Vlož(x.left), vlož(x.right). *)

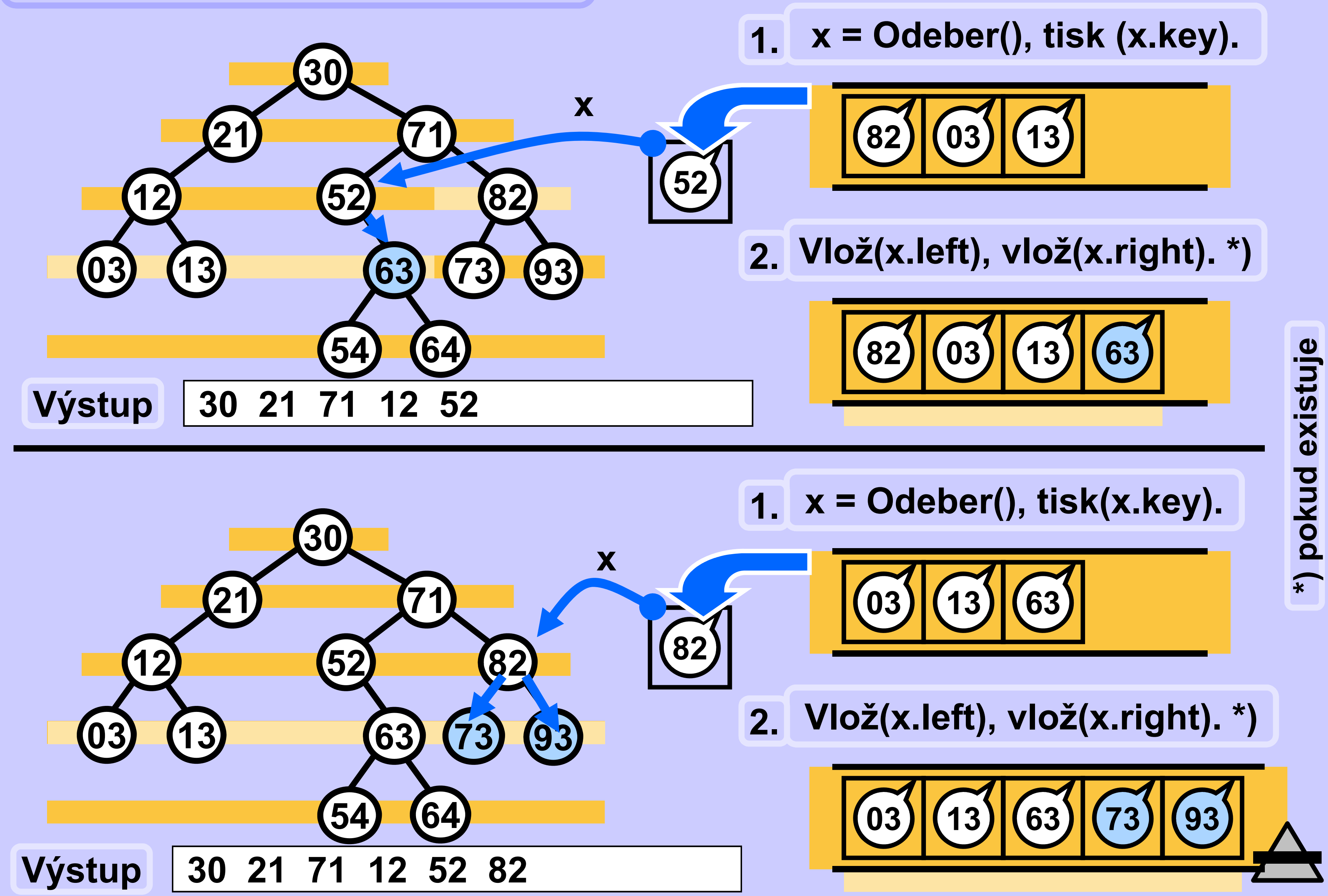
Výstup 30 21

*) pokud existuje

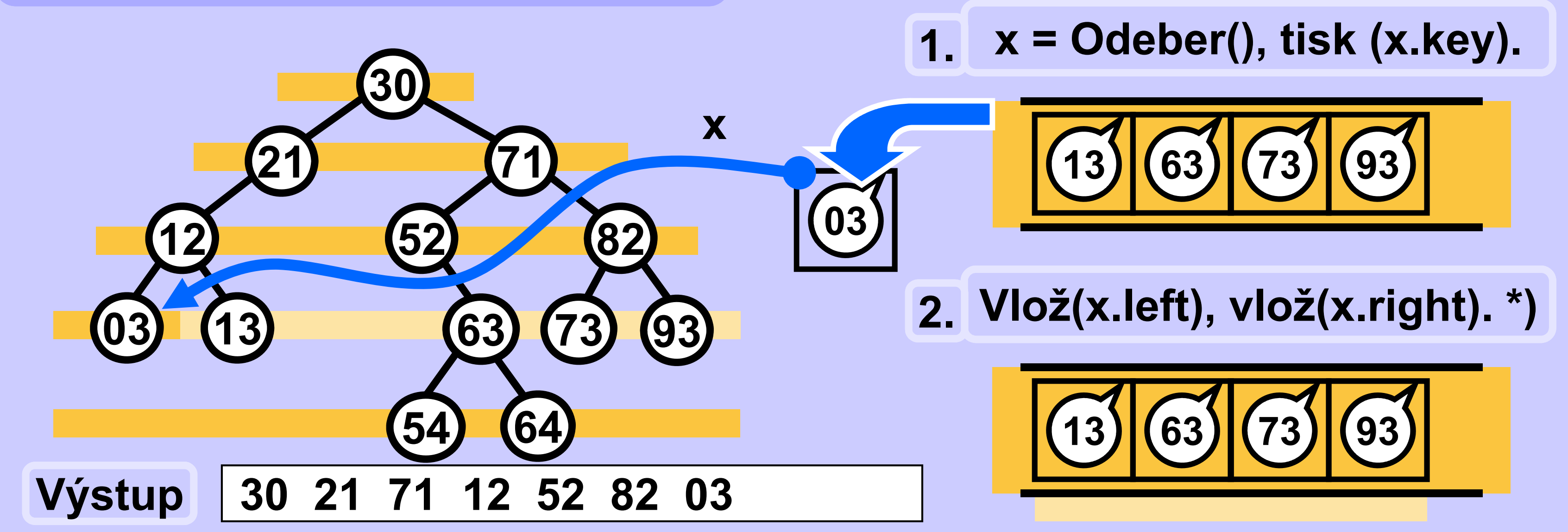
Průchod stromem do šířky

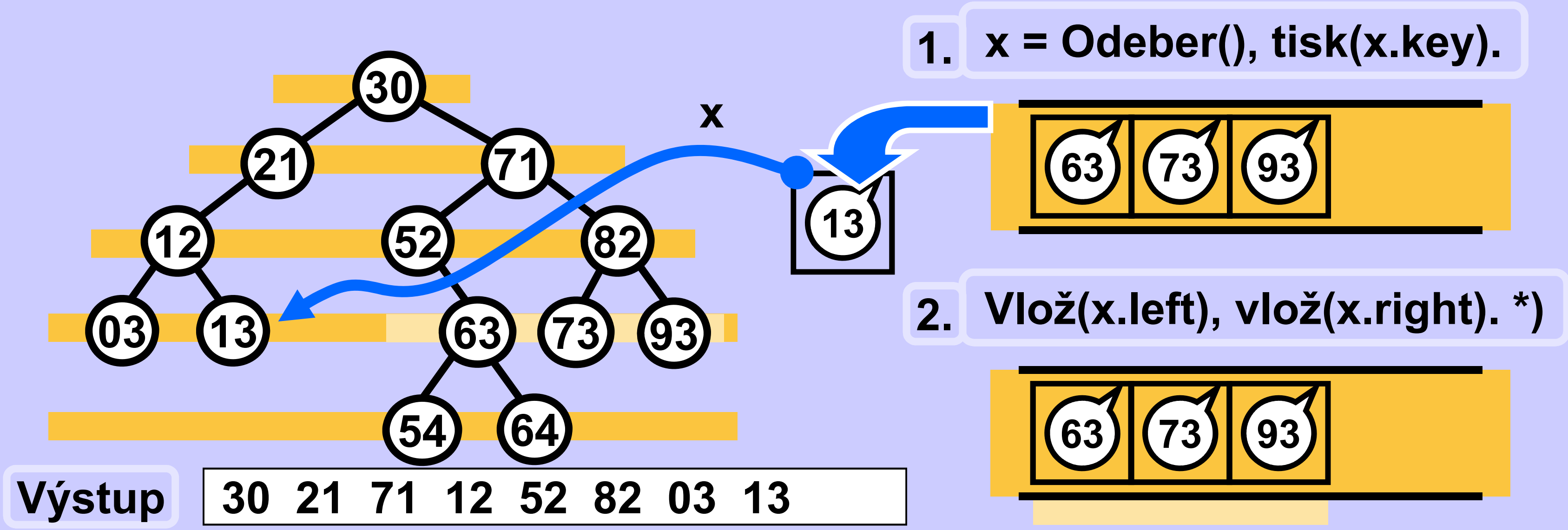


Průchod stromem do šířky



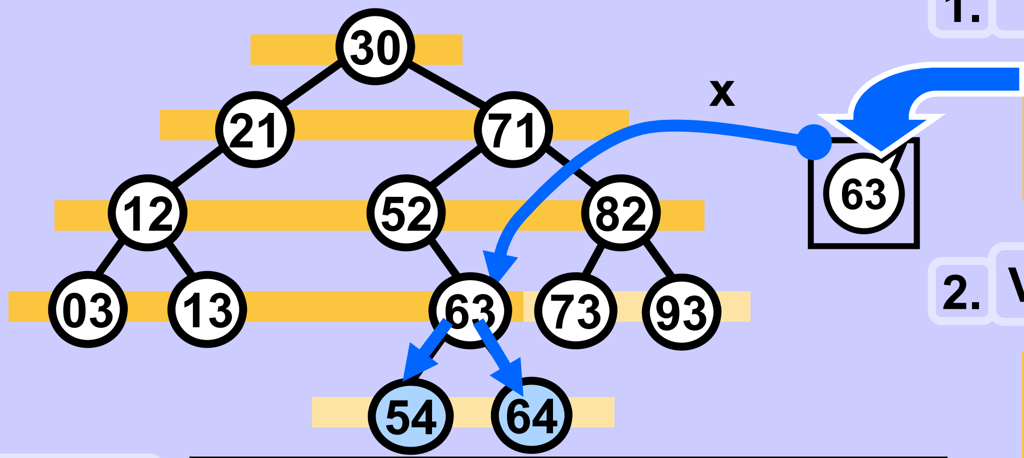
Průchod stromem do šířky



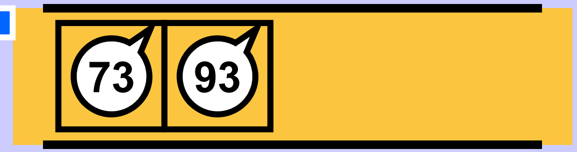


*) pokud existuje

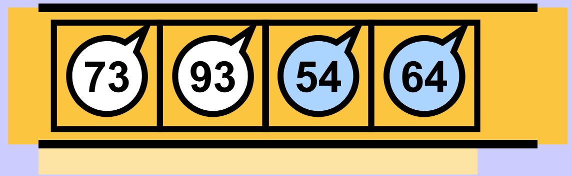
Průchod stromem do šířky



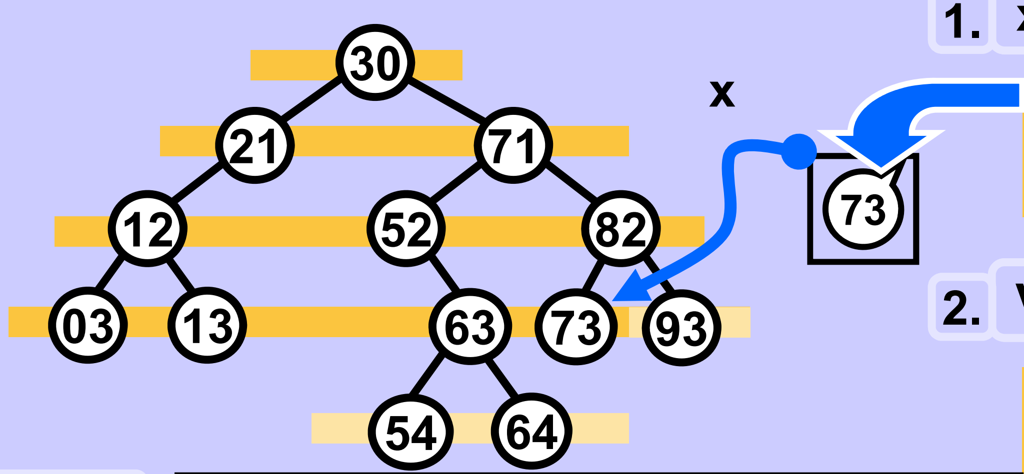
1. x = Odeber(), tisk (x.key).



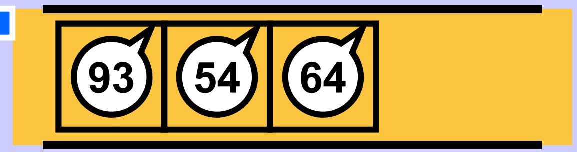
2. Vlož(x.left), vlož(x.right). *)



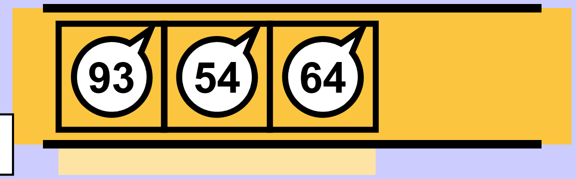
Výstup 30 21 71 12 52 82 03 13 63



1. x = Odeber(), tisk(x.key).



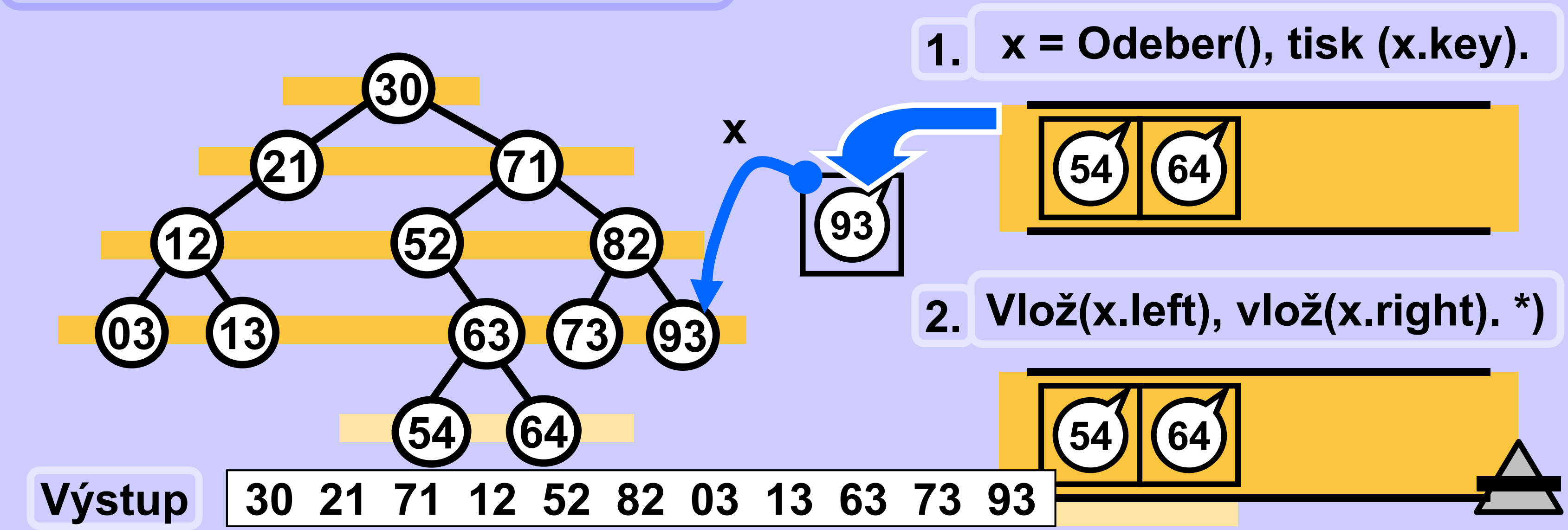
2. Vlož(x.left), vlož(x.right). *)

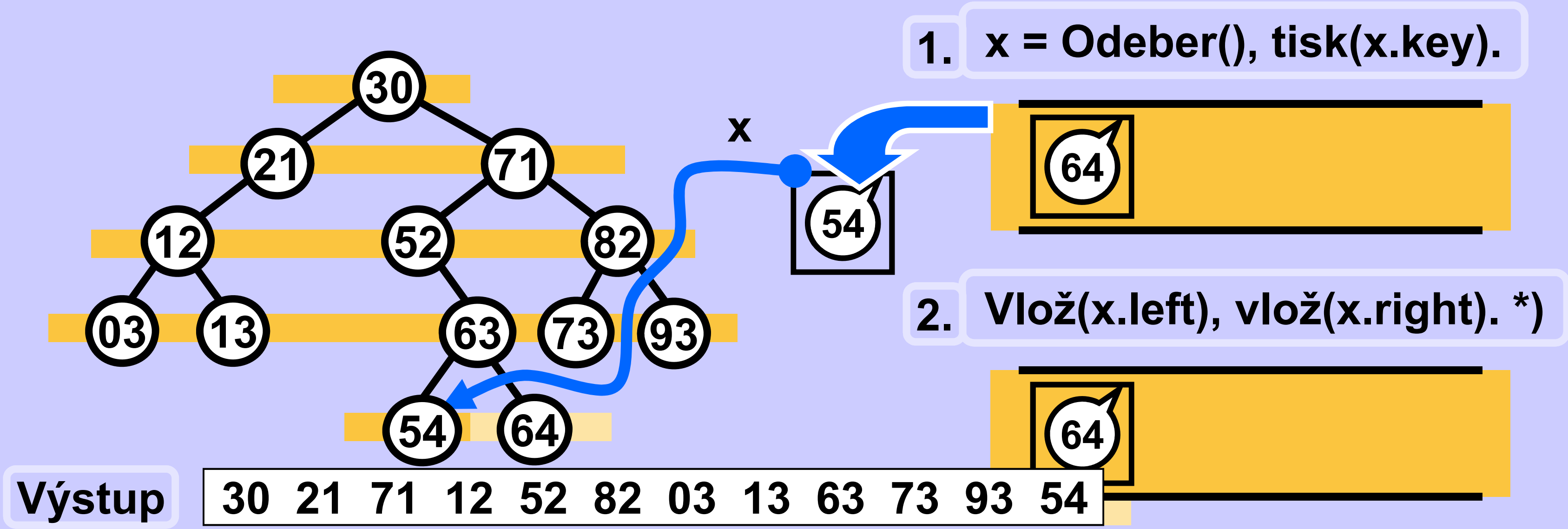


Výstup 30 21 71 12 52 82 03 13 63 73

*) pokud existuje

Průchod stromem do šířky





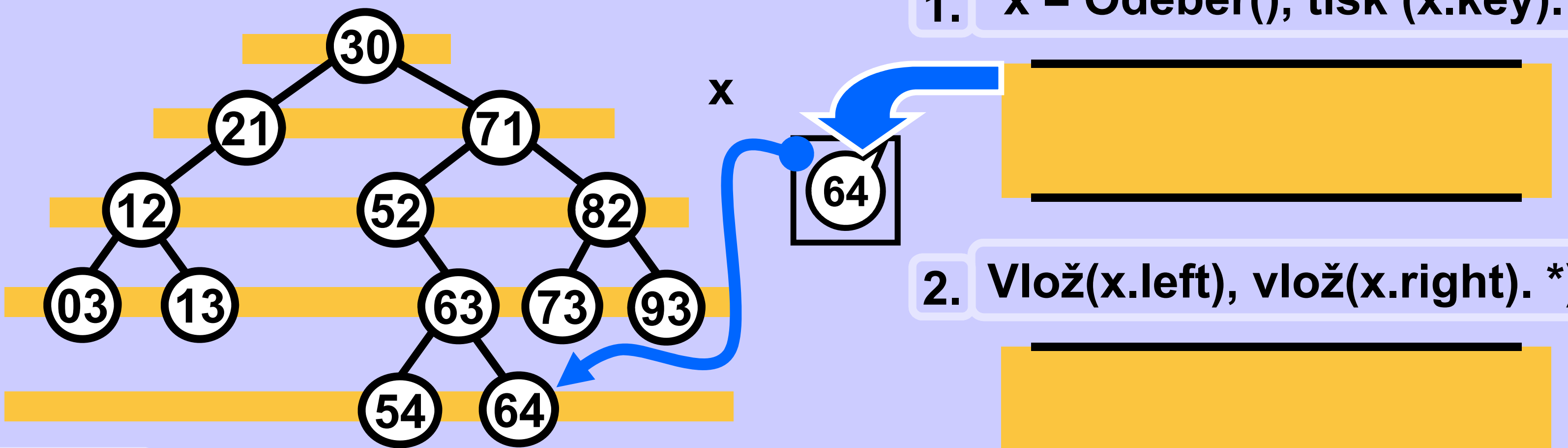
*) pokud existuje

Průchod stromem do šířky

1. x = Odeber(), tisk (x.key).

2. Vlož(x.left), vlož(x.right). *)

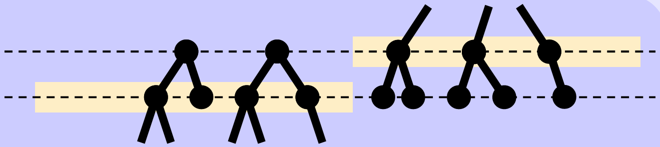
*) pokud existuje



Výstup | 30 21 71 12 52 82 03 13 63 73 93 54 64

Fronta je prázdná,
průchod stromem končí.

V neprázdné frontě jsou vždy právě
-- některé (třeba všechny) uzly jednoho patra
-- a všichni potomci těch uzlů tohoto patra, které už nejsou ve frontě.



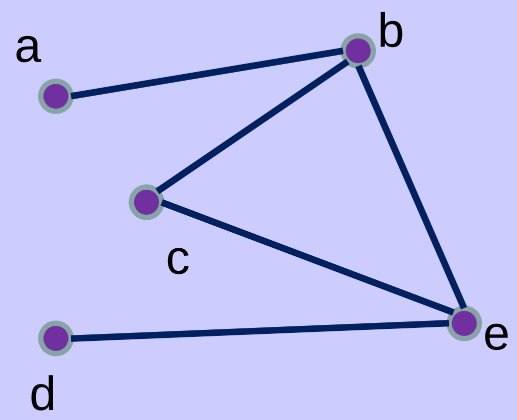
Někdy jsou ve frontě přesně všechny uzly jednoho patra. Viz výše.

Průchod stromem do šířky

```
void traverseTreeBreadthFirst( Node node ) {
    if( node == null ) return;
    Queue q = new Queue();          // init
    q.Enqueue( node );              // root into queue
    while( !q.Empty() ) {
        node = q.Dequeue();
        print( node.key );          // process node
        if( node.left  != null ) q.Enqueue( node.left );
        if( node.right != null ) q.Enqueue( node.right );
    }
}
```

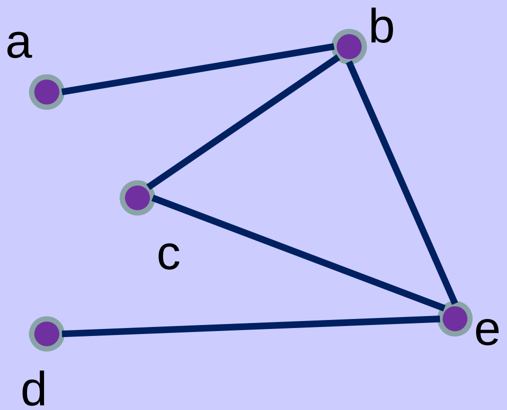
Grafy

- **graf je uspořádaná dvojice**
  - o **množiny vrcholů** $\mathcal{V}$ **a**
  - o **množiny hran** $\mathcal{E}$
- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- **příklad:**
  - **V = {a, b, c, d, e}**
  - **E = {{a,b},{b,e},{b,c}, {c,e},{e,d}}**

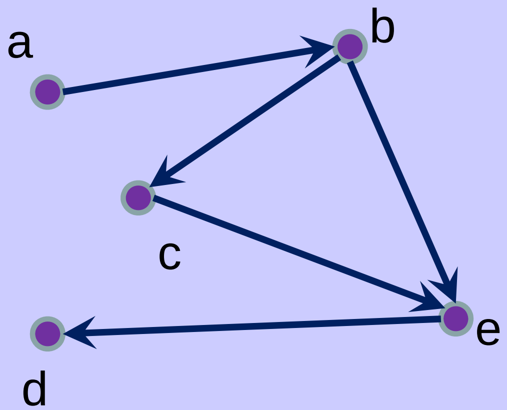


Grafy - orientovanost

- **neorientovaný graf**
  - o **hrana je <u>neuspořádaná</u> dvojice vrcholů**
- **E = {{a,b},{b,e},{b,c}, {c,e},{e,d}}**



- **orientovaný graf**
  - o **hrana je <u>uspořádaná</u> dvojice vrcholů**
- **E = {{a,b},{b,e},{b,c}, {c,e},{e,d}}**



Grafy – matice sousednosti

- Nechť $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ je graf s n vrcholy
- Označme vrcholy $v_1, \ldots, v_n$ (v nějakém libovolném pořadí)
- Matice sousednosti grafu G je čtvercová matice

$$A_G = (a_{i,j})_{i,j=1}^{n}$$

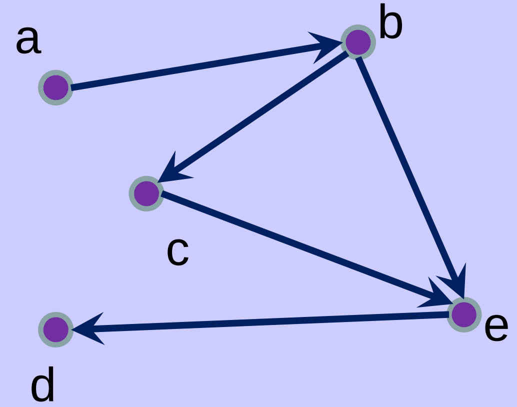
definovaná předpisem

$$a_{i,j} = \begin{cases} 1 & pro\,\{v_i, v_j\} \in E \\ 0 & jinak \end{cases}$$

Grafy – matice sousednosti

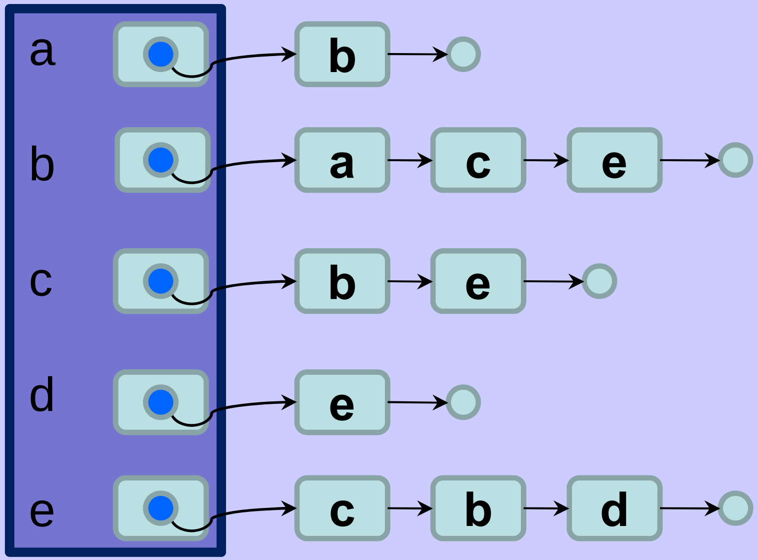
- **pro orientovany graf**

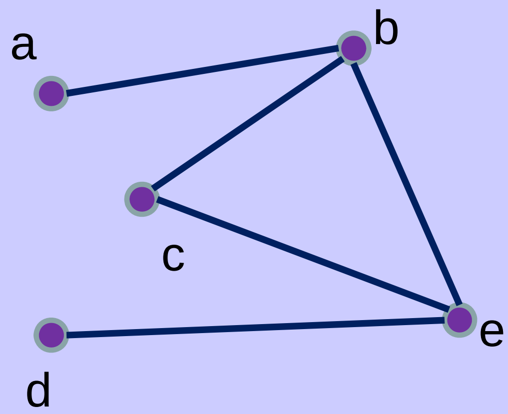
	a	**b**	**c**	**d**	**e**
a	0	1	0	0	0
b	0	0	1	0	1
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	0	1	0



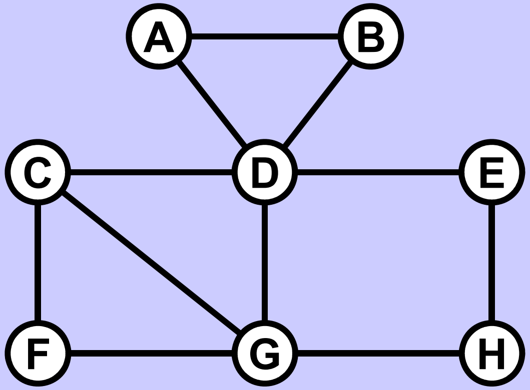
Grafy – seznam sousedů

- Nechť $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ je (ne)orientovaný graf s n vrcholy
- Označme vrcholy $v_1, \ldots, v_n$ (v nějakém libovolném pořadí)
- <u>Seznam sousedů grafu</u> $\mathcal{G}$ je pole $\mathcal{P}$ ukazatelů velikosti n
  - kde $\mathcal{P}[i]$ ukazuje na spojový seznam vrcholů,
    se kterými je vrchol v_i spojen hranou





Průchod grafem do hloubky



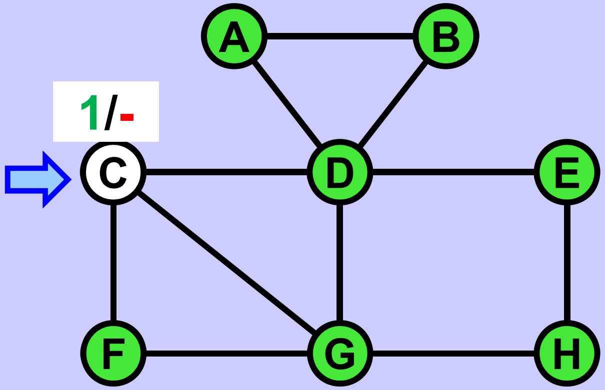
Spojová reprezentace

A → B → D
B → D → A
C → D → F → G
D → C → G → E → B → A
E → H → D
F → C → G
G → C → H → D → F
H → G → E

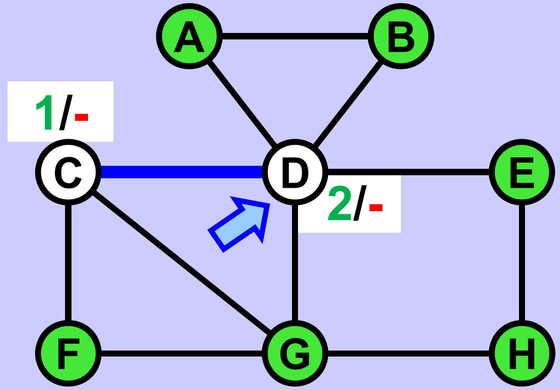
Matice sousednosti

	A	B	C	D	E	F	G	H
A	0	1	0	1	0	0	0	0
B	1	0	0	1	0	0	0	0
C	0	0	0	1	0	1	1	0
D	1	1	1	0	1	0	1	0
E	0	0	0	1	0	0	0	1
F	0	0	1	0	0	0	1	0
G	0	0	1	1	0	1	0	1
H	0	0	0	0	1	0	1	0

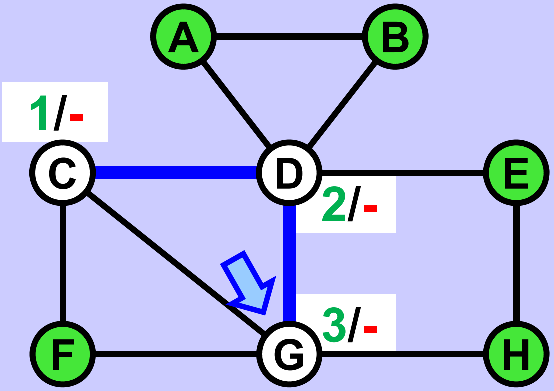
Průchod grafem do hloubky



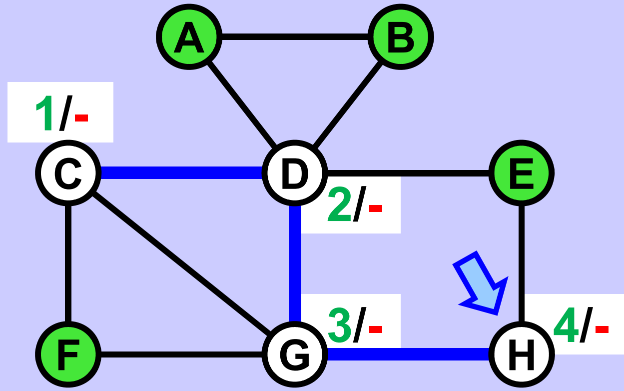
Zásobník	C
Výstup	C



Zásobník	C D
Výstup	C D

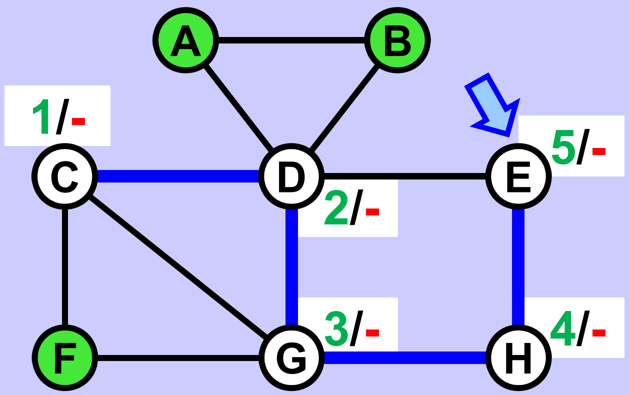


Zásobník	C D G
Výstup	C D G

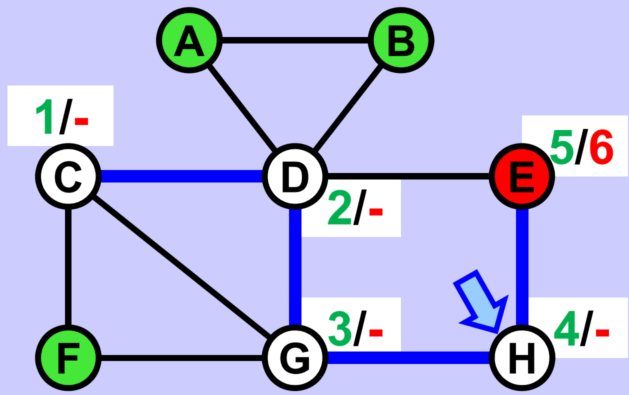


Zásobník	C D G H
Výstup	C D G H

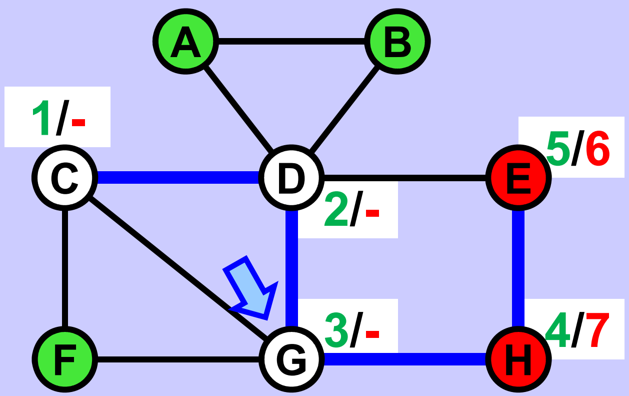
Průchod grafem do hloubky



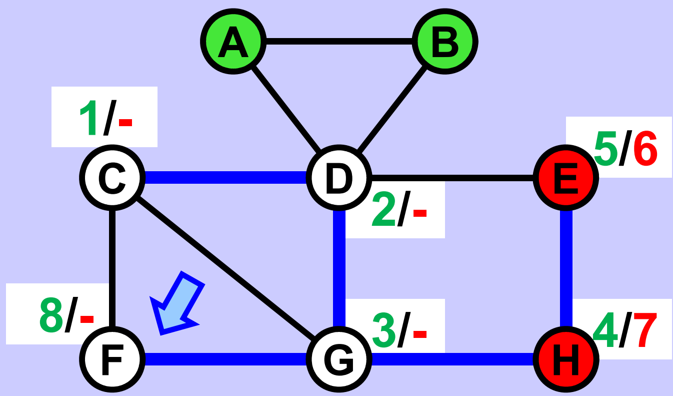
Zásobník	C D G H E
Výstup	C D G H E



Zásobník	C D G H
Výstup	C D G H E

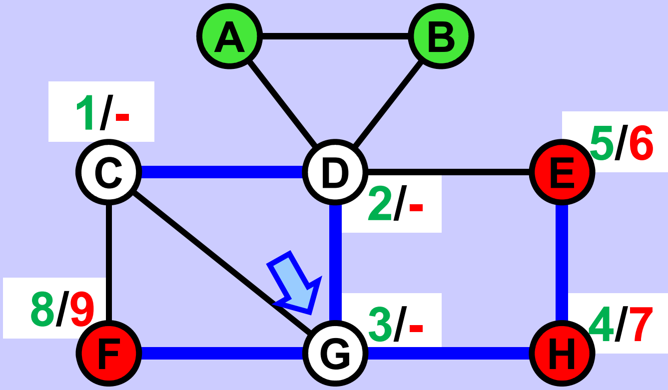


Zásobník	C D G
Výstup	C D G H E



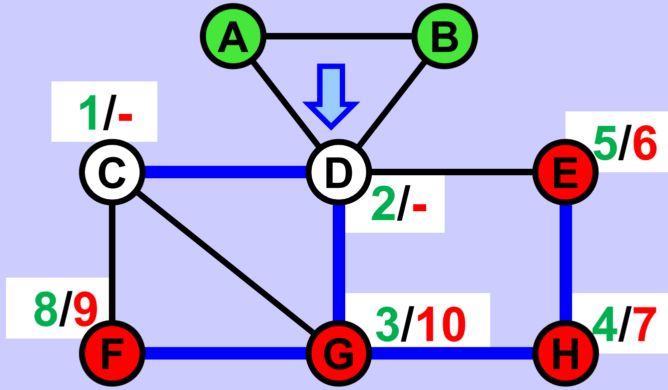
Zásobník	C D G F
Výstup	C D G H E F

Průchod grafem do hloubky



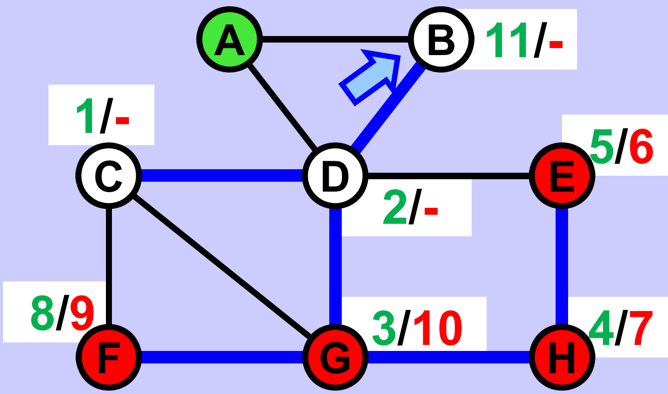
Zásobník C D G

Výstup C D G H E F



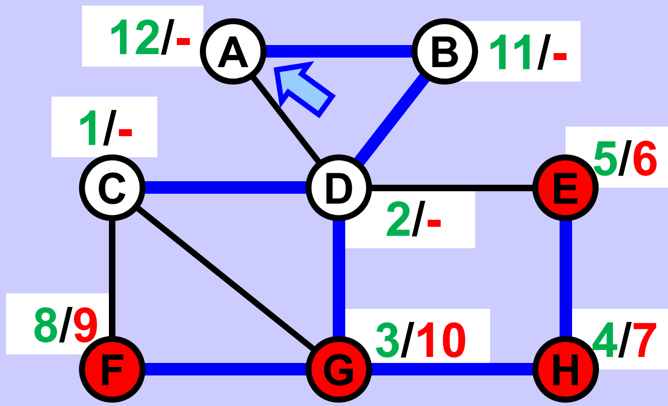
Zásobník C D

Výstup C D G H E F



Zásobník C D B

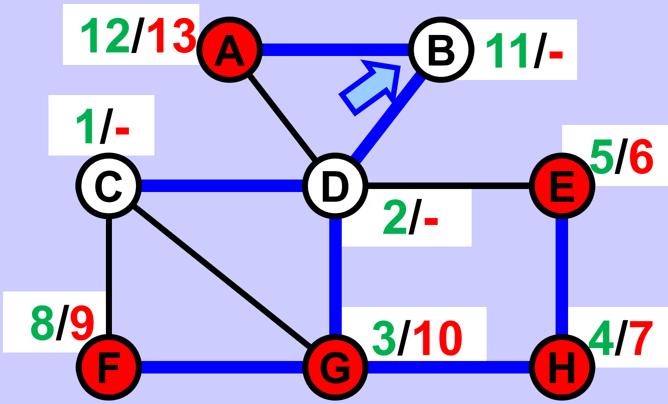
Výstup C D G H E F B



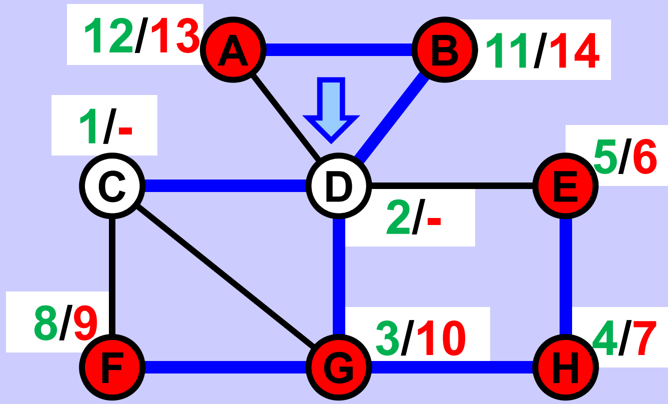
Zásobník C D B A

Výstup C D G H E F B A

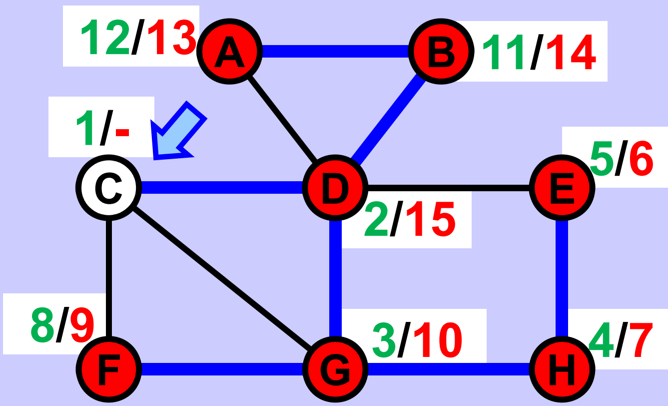
Průchod grafem do hloubky



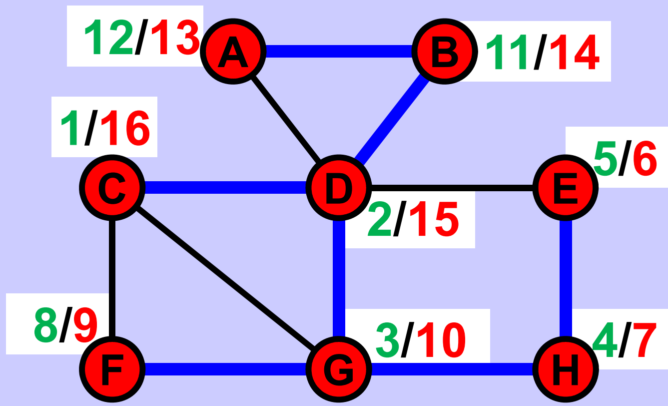
Zásobník	C D B
Výstup	C D G H E F B A



Zásobník	C D
Výstup	C D G H E F B A



Zásobník	C
Výstup	C D G H E F B A



Zásobník	
Výstup	C D G H E F B A

Průchod grafem do hloubky

Životní cyklus uzlu při prohledávání grafu
Fresh - open - closed (čerstvý - otevřený - uzavřený)

Fresh

Čerstvé uzly jsou všechny dosud ani jednou nenavštívené uzly.
Před začátkem prohledávání jsou všechny uzly čerstvé.
Při první návštěvě uzlu se uzel stává otevřeným.
Množina čerstvých uzlů se během prohledávání nikdy nezvětšuje vzhledem k inkluzi.

Open

Otevřené uzly jsou alespoň jednou navštívené uzly, které dosud nebyly uzavřeny.
Množina otevřených uzlů se může během prohledávání zvětšovat i zmenšovat.

Closed

Uzavřené uzly jsou uzly, které už během prohledávání nebudou navštíveny.
Pokud jsou všechny sousedy aktuálního uzlu otevřené nebo uzavřené, aktuální uzel se stává uzavřeným.
Množina uzavřených uzlů se během prohledávání nikdy nezmenšuje vzhledem k inkluzi.
Na konci prohledávání jsou všechny uzly uzavřené.

Průchod grafem do hloubky

Implementační poznámka

Fresh: Čerstvý uzel nemá přiřazen otevírací (ani zavírací) čas.
Open: Otevřený uzel nemá přiřazen zavírací čas.
Closed: Uzavřený uzel má přiřazen zavírací čas.

Otevírací a zavírací časy uzlů v některých případech prohledávání není nutno udržovat.

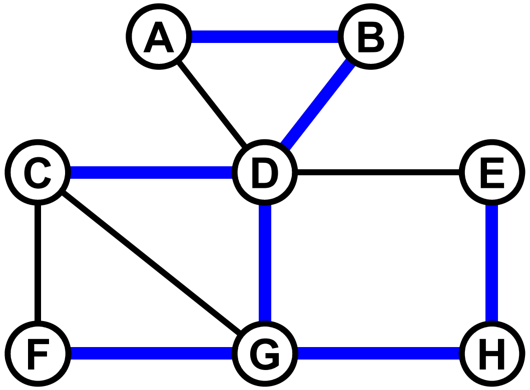
Při iterativním prohledávání s vlastním zásobníkem je ale nutno průběžně udržovat informaci u každého uzlu, zda je čerstvý, otevřený nebo zavřený.

V rekurzivním zpracování není nutno dělat explicitně ani to. Každé volání rekurzivní funkce odpovídá zpracování jednoho uzlu a všem jeho návštěvám. Při zavolání funkce se otevírá uzel, který je aktuálním parametrem volání, a na konci volání se tento uzel uzavírá. V těle funkce probíráme postupně sousedy aktuálního uzlu a voláme rekurzivně prohledávání pouze na ty z nich, které jsou ještě čerstvé (fresh). Stačí pak v každém uzlu udržovat jen informaci jednobitovou -- fresh nebo not fresh.

Průchod grafem do hloubky

Postupný obsah zásobníku

C
C D
C D G
C D G H
C D G H E
C D G H
C D G
C D G F
C D G
C D
C D B
C D B A
C D B
C D
C
--



Výpis (zpracování) uzlu při otevírání uzlu vede na posloupnost

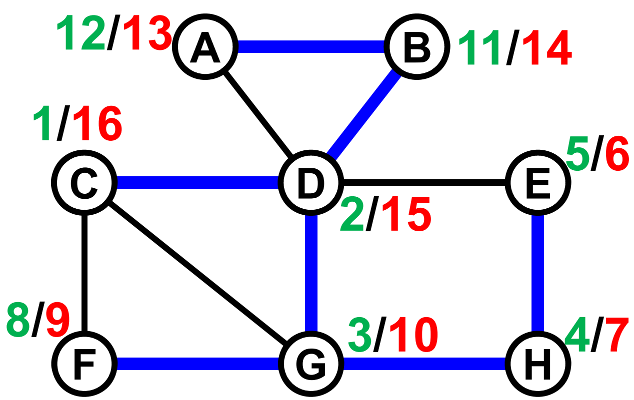
C D G H E F B A

Výpis (zpracování) uzlu při zavírání uzlu vede na posloupnost

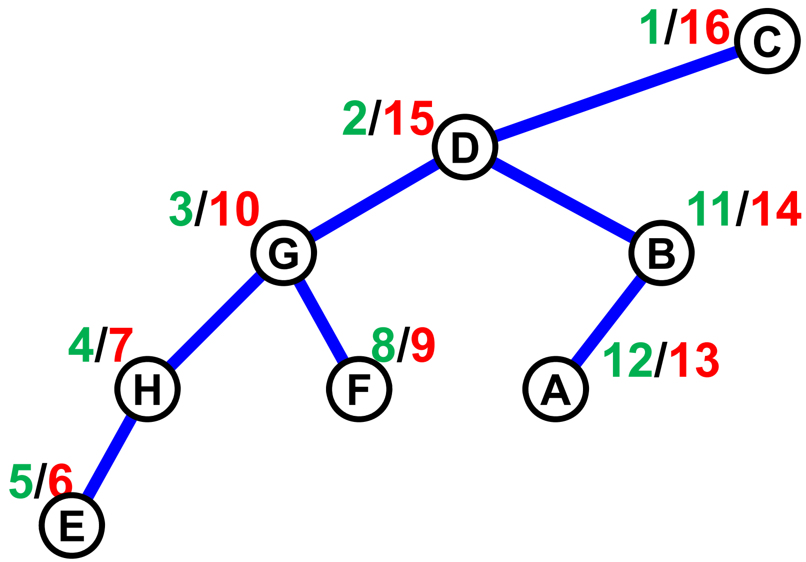
E H F G A B D C

Zpracování uzlu při jeho zavírání se uplatní např. při hledání mostů nebo artikulací v neorientovaném grafu a při detekci silně souvislých komponent v orientovaném grafu.

Průchod grafem do hloubky



Strom průchodu do hloubky s otevíracími a zavíracími časy jednotlivých uzlů



Všimněme si, že v podstromu s kořenem X pro každý uzel Y různý od X platí

Open_time(X) < Open_time(Y) < Close_time(Y) < Close_time(X).

Naopak, pokud Y neleží v podstromu s kořenem X, pak platí

Close_time(X) < Open_time(Y) nebo Close_time(Y) < Open_time(X)

Počet uzlů v podstromu s kořenem X je pak

(Close_time(X) + 1 – Open_time(X)) / 2.

Průchod grafem do hloubky

```
void DFS_rec_full( int start ) {
  vector<int> openT( N, 0 );
  vector<int> closeT( N, 0 );
  vector<int> pred( N, -1 );  // -1 == no predecessor
  int time = 0;
  DFS_rec_full( start, time, openT, closeT, pred );

  for( int n = 0; n < N; n++ )
    cout << "node " << n << " open/close "
         << openT[n] << "/"<< closeT[n]
         << " pred " << pred[n] << endl;
}
```

Průchod grafem do hloubky rekurzivně

```
void DFS_rec_full( int currNode, int & time,
                   vector<int> & openT,
                   vector<int> & closeT,
                   vector<int> & pred ){

  int neigh;
  openT[currNode] = ++time;
  for( int j = 0; j < edge[currNode].size(); j++ ){
    neigh = edge[currNode][j];
    if( openT[neigh] == 0 ) {
      pred[neigh] = currNode;
      DFS_rec_full( neigh, time, openT, closeT, pred );
      cout << currNode << " --> " << neigh << endl;
    }
  }
  closeT[currNode] = ++time;
}
```

Průchod grafem do hloubky rekurzivně, základní varianta

```
void DFS_rec_plain( int currNode, vector<bool> & fresh ){
  int neigh;
  fresh[currNode] = false;
  for( int j = 0; j < edge[currNode].size(); j++ ){
    neigh = edge[currNode][j];
    if( fresh[neigh] ) {
      DFS_rec_plain( neigh, fresh );
      cout << currNode << " --> " << neigh << endl;
    }
  }
}

void DFS_rec_plain( int start) {
  vector<bool> fresh( N, true );
  DFS_rec_plain(start, fresh);
}
```

Průchod grafem do šířky

**Životní cyklus uzlu při prohledávání grafu
je koncepčně identický jako při prohledávání do hloubky**

Fresh

Čerstvé uzly jsou všechny dosud ani jednou nenavštívené uzly.
Před začátkem prohledávání jsou všechny uzly čerstvé.
Při první návštěvě uzlu se uzel stává otevřeným.
Množina čerstvých uzlů se během prohledávání nikdy nezvětšuje vzhledem k inkluzi.

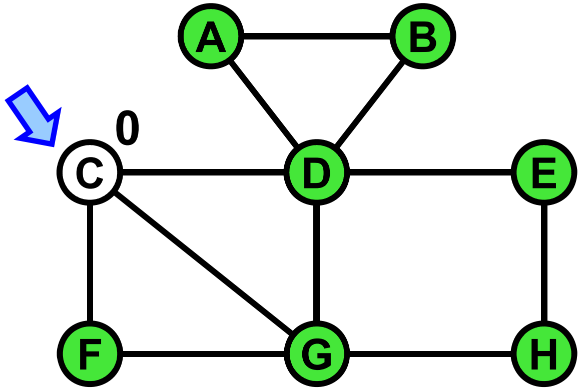
Open

Otevřené uzly jsou alespoň jednou navštívené uzly, které dosud nebyly uzavřeny.
Množina otevřených uzlů se může během prohledávání zvětšovat i zmenšovat.

Closed

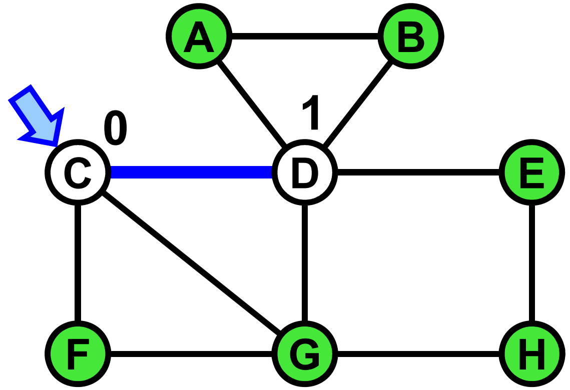
Uzavřené uzly jsou uzly, které už během prohledávání nebudou navštíveny.
Pokud jsou všechny sousedy aktuálního uzlu otevřené nebo uzavřené, aktuální uzel se stává uzavřeným.
Množina uzavřených uzlů se během prohledávání nikdy nezmenšuje vzhledem k inkluzi.
Na konci prohledávání jsou všechny uzly uzavřené.

Průchod grafem do šířky



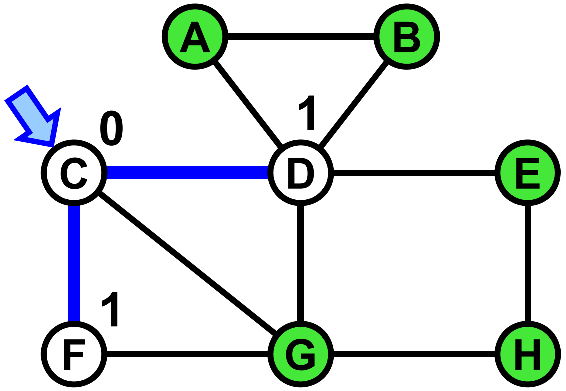
Fronta: C

Výstup: C



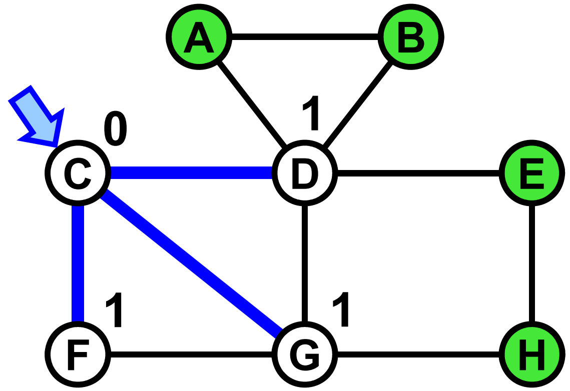
Fronta: D

Výstup: C



Fronta: D F

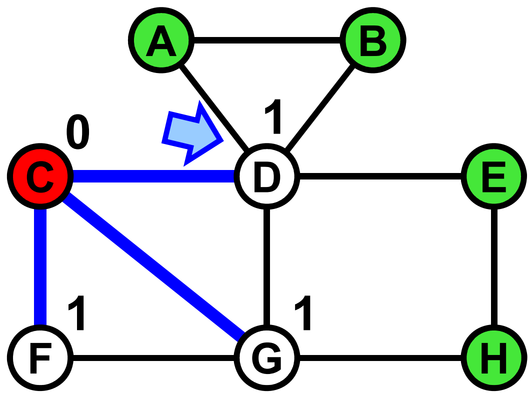
Výstup: C



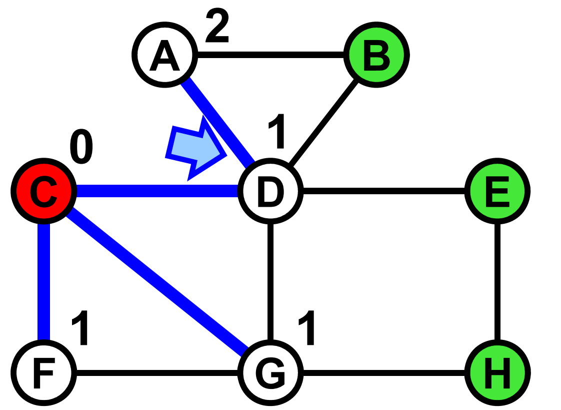
Fronta: D F G

Výstup: C

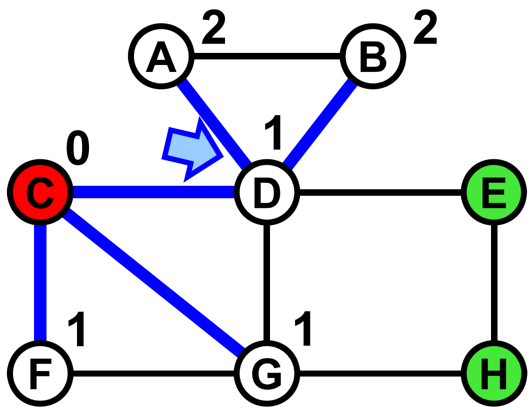
Průchod grafem do šířky



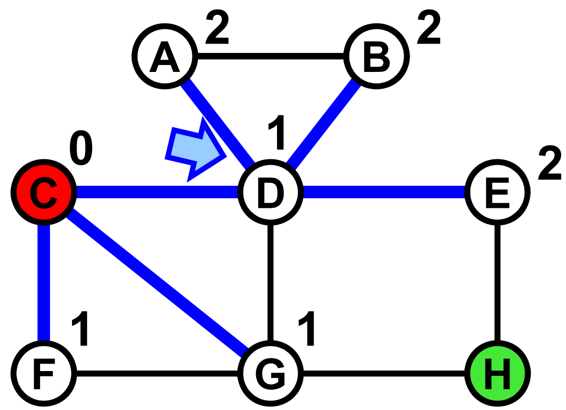
Fronta	D F G
Výstup	C D



Fronta	F G A
Výstup	C D

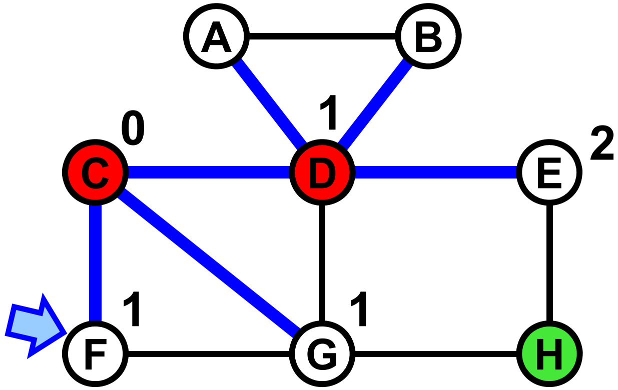


Fronta	F G A B
Výstup	C D



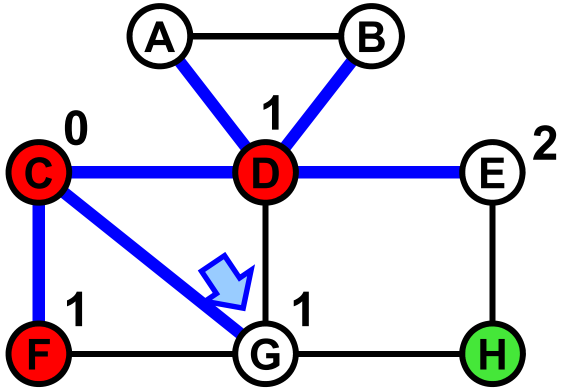
Fronta	F G A B E
Výstup	C D

Průchod grafem do šířky



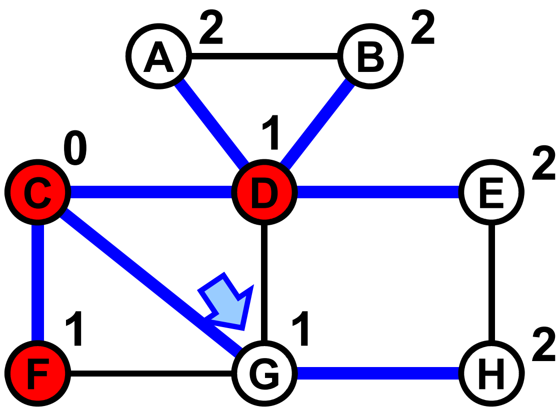
Fronta: F G A B E

Výstup: C D F



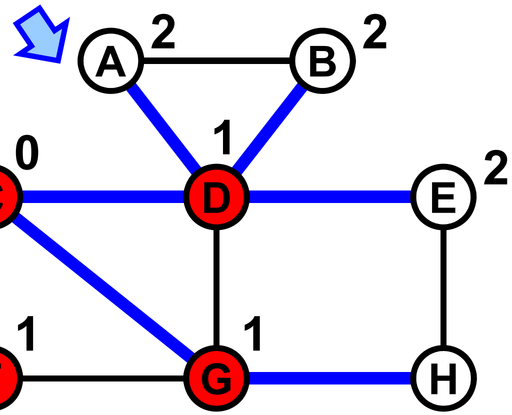
Fronta: G A B E

Výstup: C D F G



Fronta: A B E H

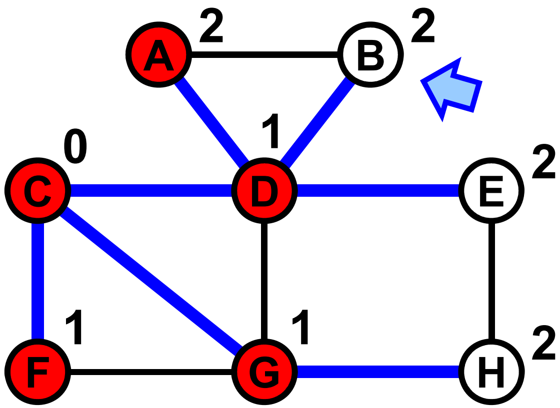
Výstup: C D F G



Fronta: A B E H

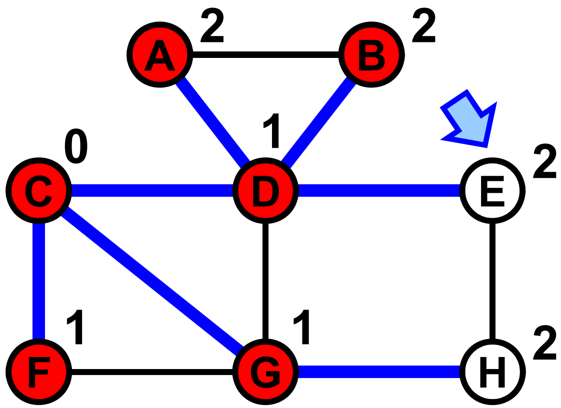
Výstup: C D F G A

Průchod grafem do šířky



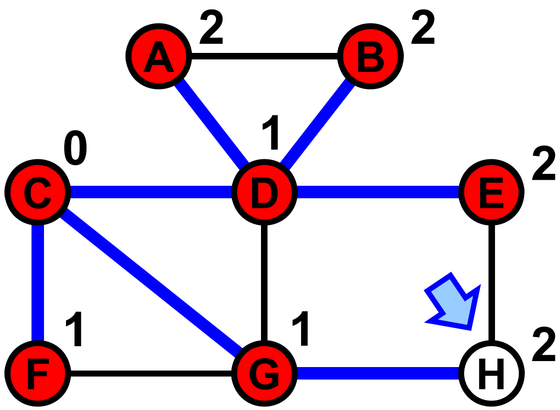
Fronta B E H

Výstup C D F G A B



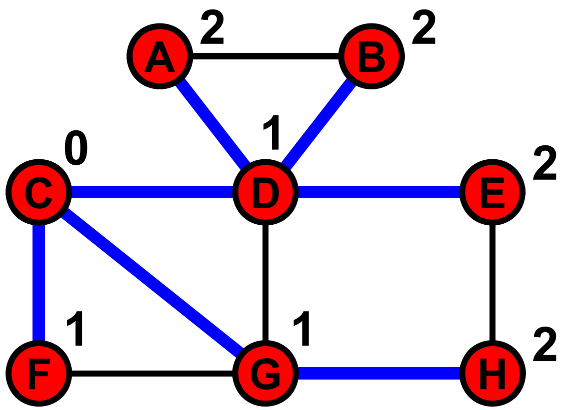
Fronta E H

Výstup C D F G A B E



Fronta H

Výstup C D F G A B E H

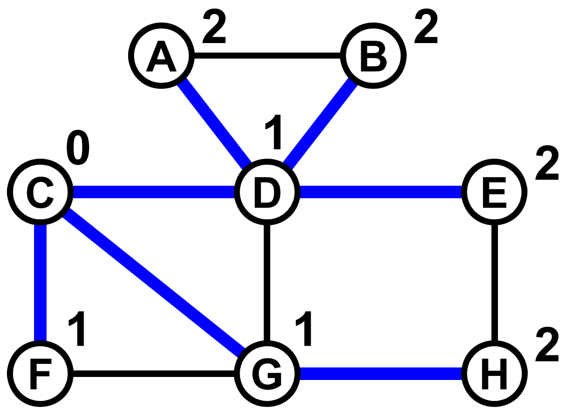


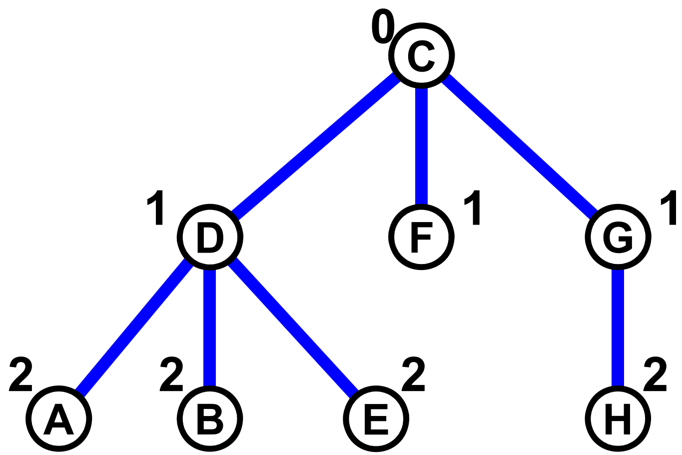
Fronta

Výstup C D F G A B E H

Průchod grafem do šířky

Strom průchodu do šířky
s vzdáleností od kořene jednotlivých uzlů





Hloubka uzlu ve stromu průchodu do šířky
je rovna jeho vzdálenosti od startovního uzlu.

Na rozdíl od průchodu do hloubky nejsou otevírací a zavírací časy důležité.

Průchod do šířky se uplatní např. při testování souvislosti grafu,
testování existence cyklů ve grafu, testování bipartitnosti a zejména
při výpočtu vzdálenosti mezi startovním a cílovým uzlem.

Průchod grafem do šířky

Implementační poznámka

Fresh: Čerstvý uzel nemá přiřazenu vzdálenost od startovního uzlu.
Open: Otevřený uzel má přiřazenu vzdálenost od startovního uzlu a je ve frontě.
Closed: Uzavřený uzel má přiřazenu vzdálenost od startovního uzlu a není ve frontě.

Stavy fresh/open/closed není nutno explicitně udržovat, obsah fronty společně s vzdáleností jednoznačně tyto stavy určují každému uzlu.

Průchod do šířky je iterativní postup, rekurzivní varianta se nepoužívá (byla pouze nepřehlednější).

Průchod grafem do šířky

Pseudokód

```
void graphBreadthFirstSearch( Node startNode ) {
    Set visited = new Set();     // visited == not fresh
    Queue q = new Queue();
    q.Enqueue( startNode );
    visited.add( startNode );
    while( !q.Empty() ) {
        node = q.Dequeue();
        print( node.key );         // process node
        forall x in node.Neighbors()
            if( x not in visited ){
                visited.add( x );
                q.Enqueue( x );
            }
    }
}
```

Průchod grafem do šířky v C++

```
void BFS( int start ){
  queue<int> q;
  vector<int> dist( N, -1 ); // -1 ... fresh node
  vector<int> pred( N, -1 ); // -1 ... no predecessor (yet)

  q.push( start );
  dist[start] = 0;

  int currn, neigh;
  while( !q.empty() ){
    currn = q.front(); q.pop();
    cout << "node " << currn << endl;  // process the node
    for( int j = 0; j < edge[currn].size(); j++ ){
      neigh = edge[currn][j];
      if( dist[neigh] == -1 ){         // neigh is fresh
        q.push( neigh );
        dist[neigh] = dist[currn] + 1;
        pred[neigh] = currn;
        cout << "BFS edge: " << currn << "->" << neigh << endl;
} } } }
```

Asymptotická složitost

Každá jednotlivá operace se zásobníkem/frontou a s použitými datovými strukturami má konstantní složitost pro jeden uzel (včetně inicializace).

Každý uzel jen jednou vstoupí do zásobníku/fronty a jednou z ní vystoupí.
Stav uzlu (fresh/open/closed) se testuje tolikrát, kolik je stupeň tohoto uzlu.
Součet stupnů všech uzlů je roven dvojnásobku počtu hran.

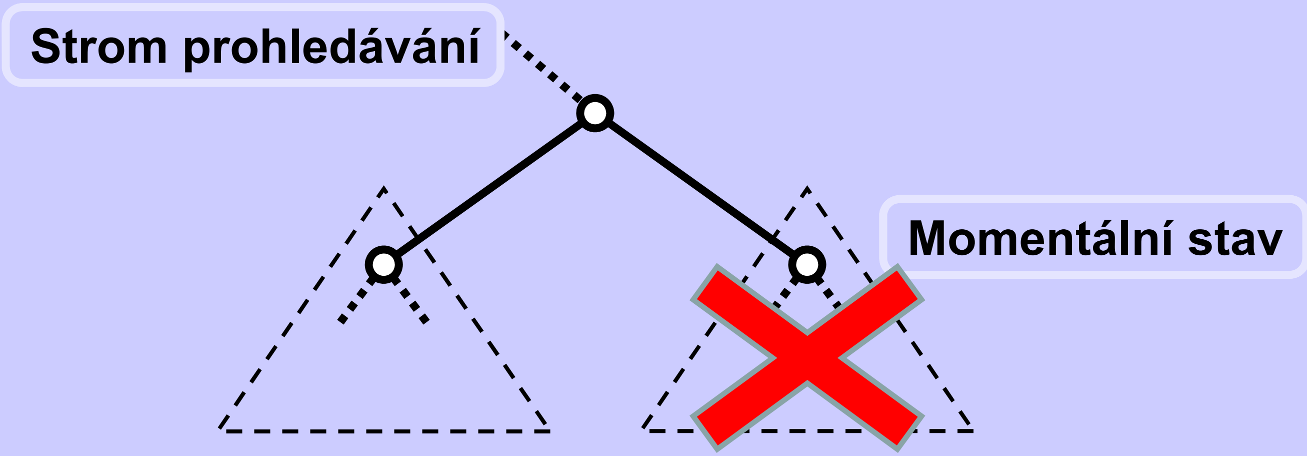
Celkem

$$\Theta(\ |V| + |E|\ ).$$

Ořezávání

- Urychlení prohledávání
- Ořezávání neperspektivních větví
- Pokud jsme schopni na základě vyhodnocení momentálního stavu zjistit,
  - že je to stav neperspektivní a
  - že rozhodně nepovede k řešení úlohy

  - „odřízneme“ ze stromu celý podstrom momentálního stavu



Příklad ořezávání – magický čtverec

- **Magický čtverec řádu $\mathcal{N}$**
  - o **čtvercové schéma čísel velikost $\mathcal{N} x \mathcal{N}$**
  - o **obsahuje právě jednou každé celé číslo od 1 do $\mathcal{N}^2$**
  - o **součet čísel ve všech řádcích a ve všech sloupcích stejný**
- **Příklad**

2	9	4
7	5	3
6	1	8

- **Triviální řešení: generování všech možných rozmístění čísel od 1 do $\mathcal{N}^2$**
- **Ořezávání: kdykoliv je součet na řádku neperspektivní**
  - o **součet všech čísel čtverce je** $\frac{1}{2} \mathcal{N}^2 (\mathcal{N}^2+1)$
  - o **součet čísel na řádku je** $\frac{1}{2} \mathcal{N} (\mathcal{N}^2+1)$

Heuristiky

- <u>Heuristika</u> je návod, který nám říká, jaký postup řešení úlohy vede obvykle k rychlému dosažení výsledku.
- Nezaručuje vždy zrychlení výpočtu.
- Heuristika se používá pro stanovení pořadí,
  - v jakém se zkoumají možné průchody stromem/grafem

- Příklad: úloha projít šachovým koněm celou šachovnici $N \times N$
- účinná heuristika: nejprve se navštíví ta dosud nenavštívená pole, z nichž bude nejméně možností dalšího bezprostředního pokračování cesty koně.
- urychlení na šachovnici 8×8 až stotisíckrát.