acm International Collegiate Programming Contest

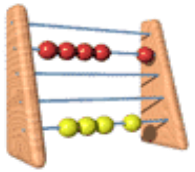
STU FIIT

ACM ICPC je opäť tu!

Lokálne kolo programátorskej súťaže na STU v rámci

CTU Open Contest

27. - 28. 10. 2017



Pošli registračný e-mail na

acm.icpc@fiit.stuba.sk

do stredy 25. 10. 2017 do 18.00 hod.

Viac informácii na:

www.fiit.stuba.sk/acm

ESET GRATEX INTERNATIONAL Nadácia pre rozvoj informatiky

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Vyhľadávanie

3. 10. 2017

zimný semester
2017/2018

Vyhľadávanie

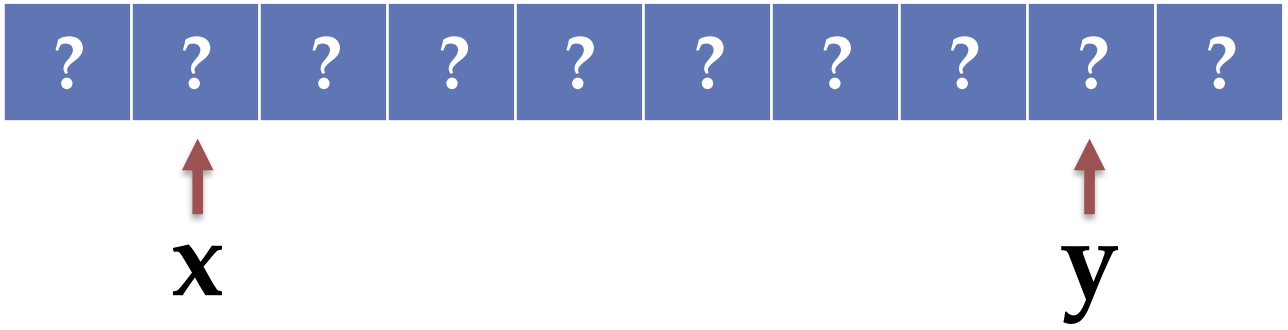
- Vstup:
  - Postupnosť: $a_1, a_2, a_3 ... a_n$
    $k(a_i)$ označíme kľúč k_i prvku a_i
  - Hľadaný kľúč x
  - Čo sú kľúče?

    Definičný obor D – reťazce, reálne čísla, dvojice celých čísel, ...
  - Relácia = (rovnosti) – relácia ekvivalencie nad D
  - Usporiadanie kľúčov < (binárna relácia nad D)

    Lineárne usporiadaná množina K (total ordering)
    Pre $k_1, k_2 \in D$ budeme písať, že $k_1 \leq k_2$ akk $k_1 < k_2$ alebo $k_1 = k_2$.
- Výstup:
  - Index res $\in \{1,2,\ldots,n\}$ takého prvku, že $k(a_{res}) = x$, alebo 0 ak taký prvok neexistuje.

Lineárne vyhľadávanie

- O vstupnej postupnosti kľúčov nemám žiadne znalosti
- Algoritmus: **Prehľadám postupne všetky prvky**
- Čo ma zaujíma: Počet vykonaných operácií
- Vstup: postupnosť N čísel, hľadám x=33
  - hľadám y=42



- Počet vykonaných operácií:
  - najlepší prípad: O(1), najhorší prípad O(n), priemerne (n)

Lineárne vyhľadávanie – zdrojový kód

- Vyhľadanie podľa čísla

```
int hladaj(int *data, int n, int kluc)
{
    int i;
    for (i = 0; i < n; i++)
        if (data[i] == kluc)
            return i;
    return -1;
}
```

- Vyhľadanie podľa mena

```
struct Osoba
{
    char *meno;
    int vek;
};

int hladaj(struct Osoba *data, int n, char *kluc)
{
    int i;
    for (i = 0; i < n; i++)
        if (!strcmp(data[i].meno, kluc))
            return i;
    return -1;
}
```

Existuje lepší algoritmus pre tento problém?

- Nie

  ak o postupnosti kľúčov naozaj nič ďalšie neviem

- Uvažujme teraz, že by sme vstupnú postupnosť mali vzostupne usporiadanú – dodatočná informácia je **usporiadanie**
- **Hľadám y=42**

y

- **Skontrolovaním jedného prvku môžem vylúčiť z ďalšieho hľadania celý interval prvkov**
  - Najlepší prípad?

    kratší z intervalov vľavo alebo vpravo
  - Najhorší prípad?

    dlhší z intervalov vľavo alebo vpravo

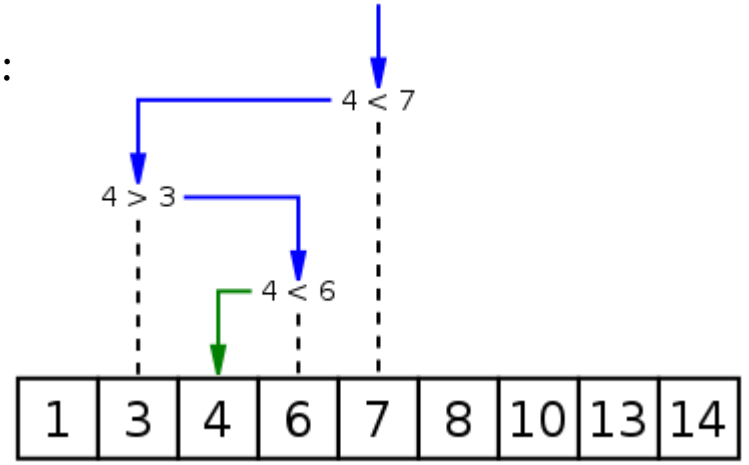
Princíp rýchlejšieho vyhľadávania

- Nejaká informácia navyše o vstupe, napr. usporiadanie
- Skontrolovaním **jedného prvku** môžem vylúčiť z ďalšieho hľadania celý interval prvkov
  - Najlepší prípad? kratší z intervalov vľavo alebo vpravo
  - Najhorší prípad? dlhší z intervalov vľavo alebo vpravo
- Čo keď nechcem riskovať lepší-horší prípad?
- Metóda pólenia intervalu (**binárne vyhľadávanie**)
  - Vždy **skontrolujem prvok v strede intervalu**, v ktorom hľadaný kľúč ešte môže byť
  - Ak som našiel hľadaný kľúč, končím, inak pokračujem v zostávajúcej polovici intervalu
  - Koľko bude porovnaní?

Analýza zložitosti binárneho vyhľadávania

- **Algoritmus: skontrolujem prvok v strede intervalu**, v ktorom hľadaný kľúč ešte môže byť
  - Ak som našiel hľadaný kľúč, končím, inak pokračujem v zostávajúcej polovici intervalu

Príklad (hľadám 4):



- Koľko bude porovnaní?
  - Najlepší prípad: 0(1)
  - Najhorší prípad?
    keď sa hľadaný kľúč v postupnosti nenachádza
  - Koľko razy môžem skrátiť interval na polovicu, až kým nedostanem posledný prvok, ktorý to teoreticky môže byť?
    O(log n)

Porovnanie O(n) vs O(log n)

- Lineárne vyhľadávanie
  - N prvkov = N operácie
  - 2N prvkov = 2N operácií
- Binárne vyhľadávanie
  - N prvkov = log N operácií
  - 2N prvkov = (log N) + 1 operácií

N	log N
10	4
1000	10
1 000 000	20
2 000 000 000	32

Existuje ešte rýchlejší algoritmus pre tento problém?

- Nie

  ak o postupnosti kľúčov naozaj nič ďalšie neviem

- Uvažujme teraz, že by sme pre vstupnú postupnosť mali **ešte nejakú dodatočnú informáciu**
  - Aká by to mohla byť?
- Distribúcia hodnôt kľúčov
  - Predstava o tom, koľko hodnôt ktorého kľúča sa môže v postupnosti nachádzať.
  - Príklad z nedávnej praxe:
    telefónny zoznam – počet priezvisk podľa začiatočného písmena

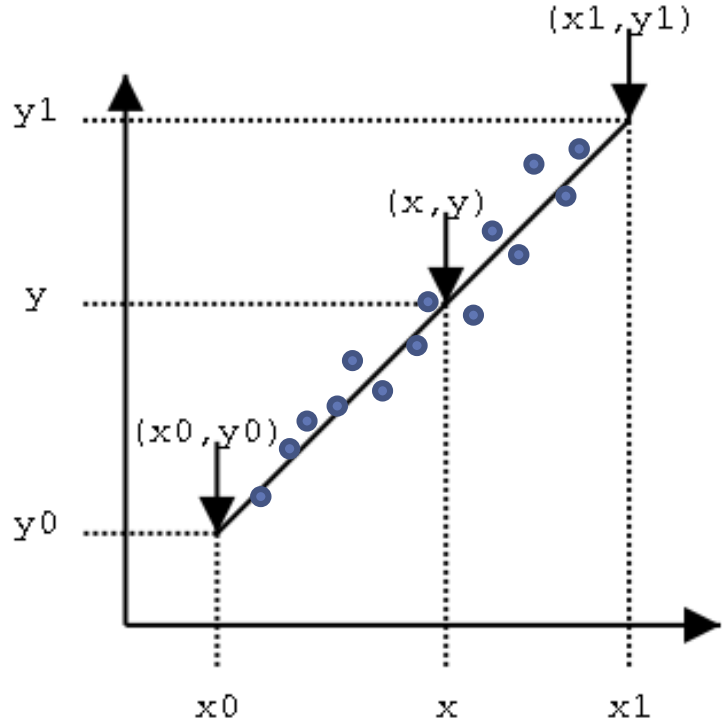
Interpolačné vyhľadávanie

- Predpokladajme rovnomernú distribúciu hodnôt kľúčov
- Na intervale $<x_0,x_1>$ nadobúdajú hodnoty $<y_0,y_1>$
- Hľadám kľúč y, ako určím čo najpravdepodobnejší výskyt – index x, taký, že $k(a_x)$ je blízko y?

$$\frac{y-y_0}{y_1-y_0} = \frac{x-x_0}{x_1-x_0}$$

$$x = x_0 + \frac{(y-y_0)*(x_1-x_0)}{y_1-y_0}$$

- Výpočtová zložitosť pre postupnosť s n prvkami
  - **O(log log n) krokov**



Niektoré ďalšie vyhľadávacie algoritmy

- Ternárne vyhľadávania
- Vyhľadávanie podľa Zlatého rezu
- Exponenciálne vyhľadávanie
- Preskakovacie vyhľadávanie

Nová požiadavka – dynamická množina

- Doteraz sme vyhľadávali v postupnosti, ktorú sme dostali celú na vstupe a ďalej sme ju nemenili
- Čo keby sme chceli postupnosť upravovať?
  - Uvažujme register osôb
  - Pridávať prvky (narodenie dieťaťa)
  - Vyhľadávať prvky (osoba podľa rodného čísla)
  - Odstrániť prvky (úmrtie)

- ADT: dynamická množina

ADT – (Dynamická) množina

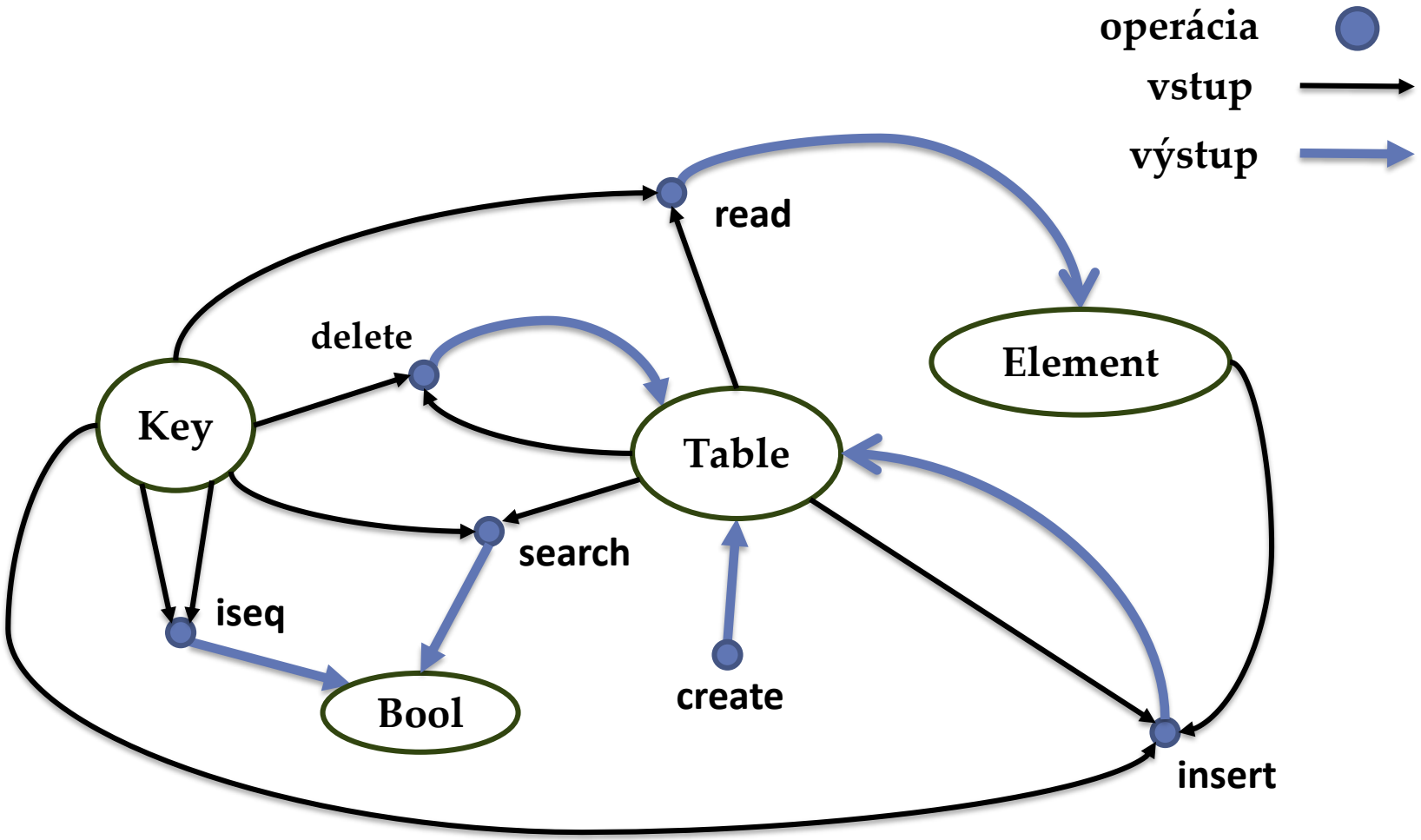
- abstraktný údajový typ množina
  - počet prvkov v údajoch typu množina sa často mení
  - najčastejšie operácie:
    insert – vložiť/pridať prvok do množiny
    search – vyhľadať prvok v množine
    delete – odstrániť prvok z množiny
- Nazývame aj **slovník (dictionary)**
- Napr: tabuľka symbolov v prekladačoch

Tabuľka

- skupina metód, ako implementovať slovník
- bežne aj synonymum pre slovník, najmä ak uvažujeme aj jeho implementáciu
- ale spravidla pomenovanie implementujúceho vektora

- určuje štruktúru pre jednotlivé údaje, ktorá združuje/asociuje hodnotu s kľúčom
- V pamäti sa údaje uchovávajú ako dvojica kľúč – hodnota = položka, prvok tabuľky
- K hodnote sa pristupuje pomocou kľúča – kľúč položku jednoznačne určuje

Slovník / Tabuľka – signatúra



Slovník / Tabuľka – axiómy

- Opisujú vlastnosti – význam (sémantiku) operácií prostredníctvom ekvivalencie výrazov

  Pre všetky $k, k_1, k_2 \in$ KĽÚČ, $e \in$ ELEMENT, $t \in$ TABUĽKA platí:

```
insert(k1,e1, insert(k2,e2,t)) =
    if(iseq(k1, k2))
       then insert(k1,e1,t)
       else insert(k2,e2,insert(k1,e1,t))
```

Slovník / Tabuľka – axiómy (2)

- Pre všetky $k, k_1, k_2 \in$ KĽÚČ, $e \in$ ELEMENT, $t \in$ TABUĽKA platí:

```
search(k, create) = false
search(k1, insert(k2,e,t)) =
  if(iseq(k1, k2))
          then true
          else search(k1, t)

read(k, create) = error_elem
read(k1, insert(k2,e,t)) =
 if(iseq(k1, k2))
    then e
    else read(k1, t)
```

Slovník / Tabuľka – axiómy (3)

- Pre všetky $k, k_1, k_2 \in$ KĽÚČ, $e \in$ ELEMENT, $t \in$ TABUĽKA platí:

```
delete(k, create) = create
delete(k1, insert(k2,e,t)) =
 if(iseq(k1, k2))
    then delete(k1, t)
  else insert(k2,e,delete(k1, t))
```

Dynamická množina poľom

- Implementácia poľom – vektorom
- Pridanie/odstránenie jedného prvku do usporiadanej postupnosti vyžaduje (v najhoršom prípade) posunutie všetkých prvkov
- Teda vieme spraviť dátovú štruktúru, podporujúcu operácie:
  - **Pridanie prvku X** – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti
  - **Vyhľadanie prvku X** – vyžaduje O(log N) operácií, využijeme binárne vyhľadávanie
  - **Odstránenie prvku X** – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti

Dynamická množina spájaným zoznamom

- Implementácia spájaným zoznamom
- Pridanie/odstránenie jedného prvku do usporiadanej postupnosti vyžaduje (v najhoršom prípade) prehľadanie všetkých prvkov
- Teda vieme spraviť dátovú štruktúru, podporujúcu operácie:
  - **Pridanie prvku X** – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti
  - **Vyhľadanie prvku X** – vyžaduje O(N) operácií
  - **Odstránenie prvku X** – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti

Existuje rýchlejší algoritmus pre pridávanie?

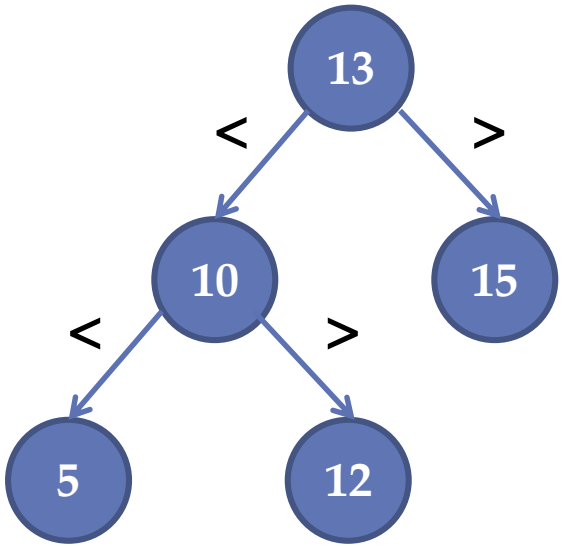
- Vyhľadávanie dokážeme v O(log N)
- Dokážeme prvky aj pridávať v O(log N) pri zachovaní času pre vyhľadávanie O(log N)?
- Spomeňme si na základný princíp rýchlejšieho (aj binárneho) vyhľadávania v usporiadanej postupnosti:

  Skontrolovaním jedného prvku môžem vylúčiť z ďalšieho hľadania celý interval prvkov

- Takéto „rozhodovanie“ vieme vhodne reprezentovať rozhodovacím stromom...

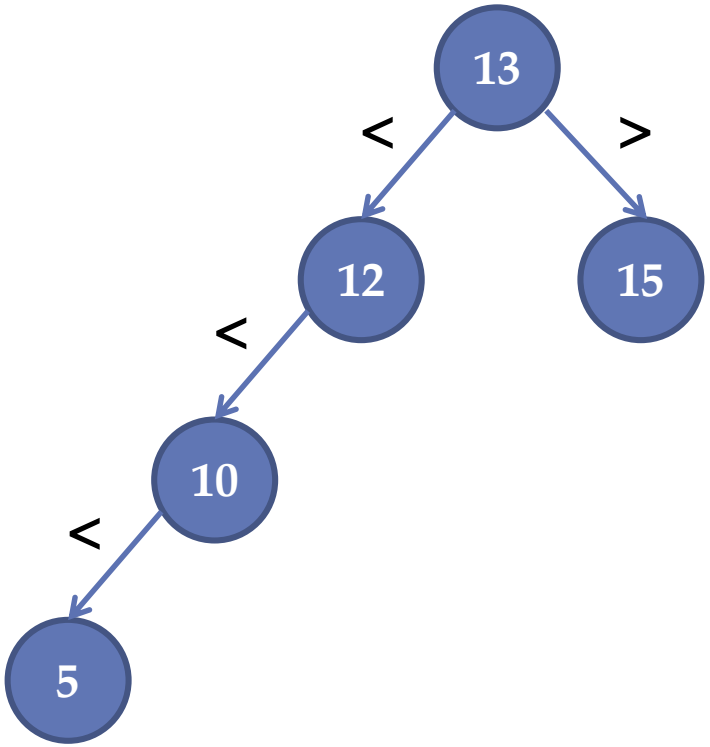
Binárny vyhľadávací strom

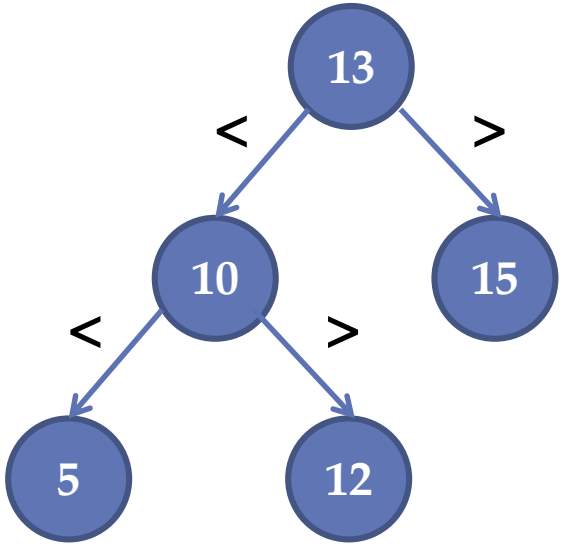
- Každý vrchol má hodnotu a (najviac) dvoch nasledovníkov:
  - Ľavého – kde je menšia hodnota
  - Pravého – kde je väčšia hodnota
- Tá istá množina prvkov môže byť v binárnom vyhľadávacom strome rozlične umiestnená vo vrcholoch
  - Dôležité však je, aby bola splnená podmienka usporiadania hodnôt:
  - ľavý nasledovník každého vrcholu má menšiu hodnotu, pravý väčšiu



Binárny vyhľadávací strom (2)

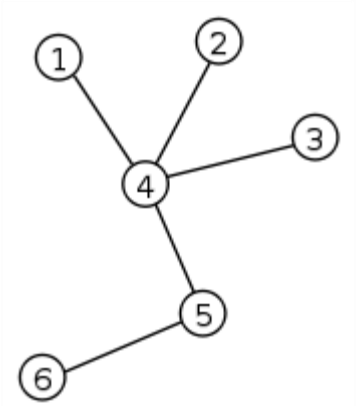
- Rôzna štruktúra stromu, rovnaké prvky, ktorý je lepší? :)

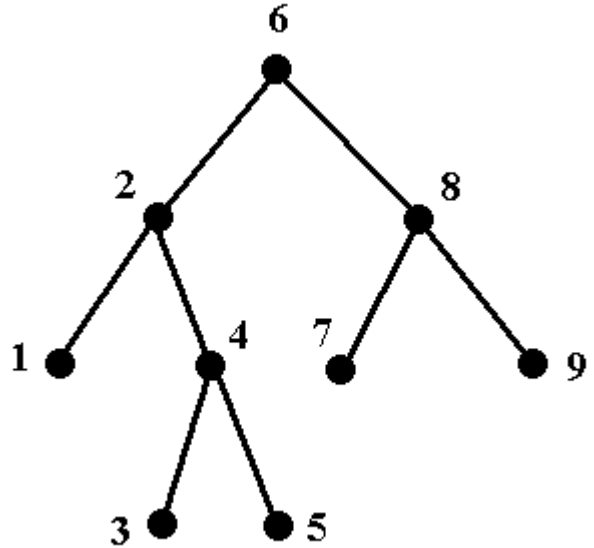




Strom – Definícia (teória grafov)

Strom – Súvislý neorientovaný graf bez cyklov





Graf G = (V, E)
V – množina vrcholov
E – množina hrán (dvojíc vrcholov)

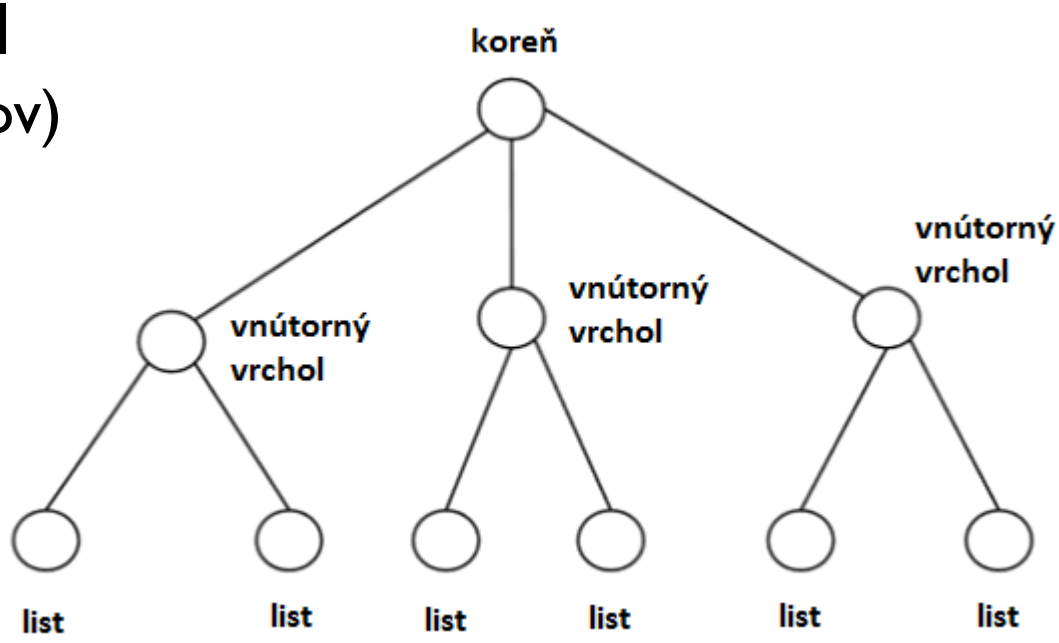
Neorientovaný graf – hrany nemajú orientáciu (smer)

Súvislý graf – po hranách je možné prejsť z ľubovoľného vrcholu do ľubovoľného iného vrcholu v grafe

Cyklus – taký prechod po hranách, že začneme v nejakom vrchole, prejdeme po aspoň jednej hrane a skončíme v počiatočnom vrchole

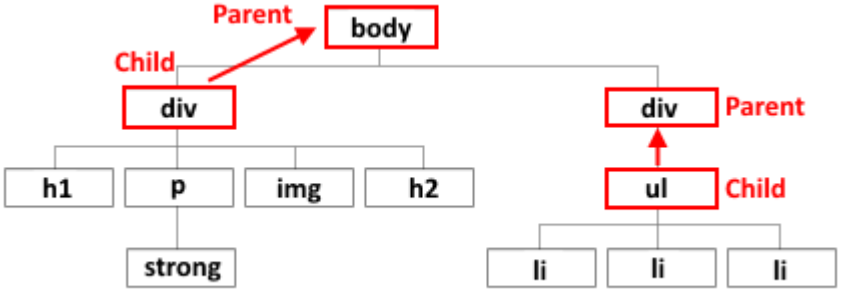
Zakoreňený strom

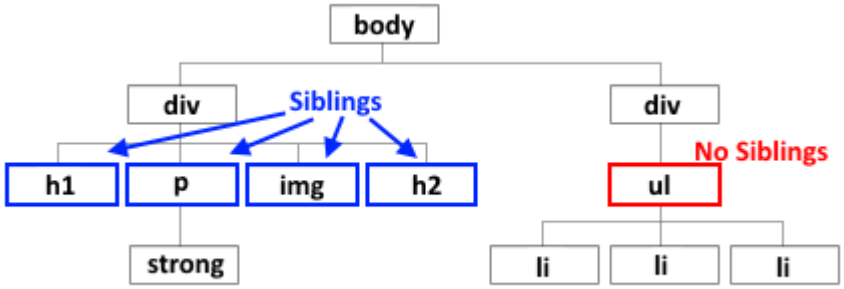
- Strom, v ktorom je význačný vrchol – **koreň** (root)
- Uvažujme vrchol **u**, ktorý leží na ceste z **koreňa** do **v**
  u nazývame **predchodca (ancestor) v**,
  resp. **v** je **nasledovník (descendant) u**
- **List** – koncový vrchol
  (ktorý nemá nasledovníkov)
  - Ostatné vrcholy sú **vnútorné**
- Zvyčajne sa uvažuje orientácia hrán zhora dole (od koreňa k listom)



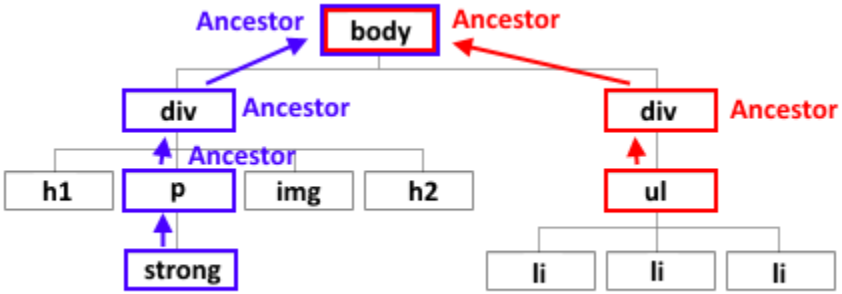
Zakoreňený strom (2) – príklad HTML

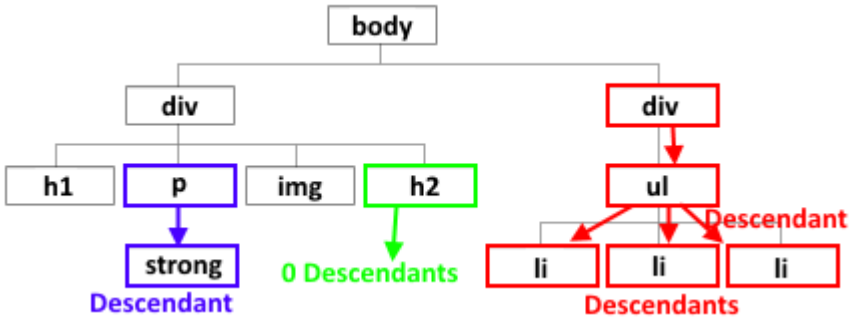
- **Rodič (parent)** – najbližší-priamy predchodca vrcholu
- **Dieťa / potomok (child)** – priamy nasledovník vrcholu
- **Súrodenci (siblings)** – vrcholy s rovnakým rodičom





- **Nasledovník / predchodca je aj nepriamy:**



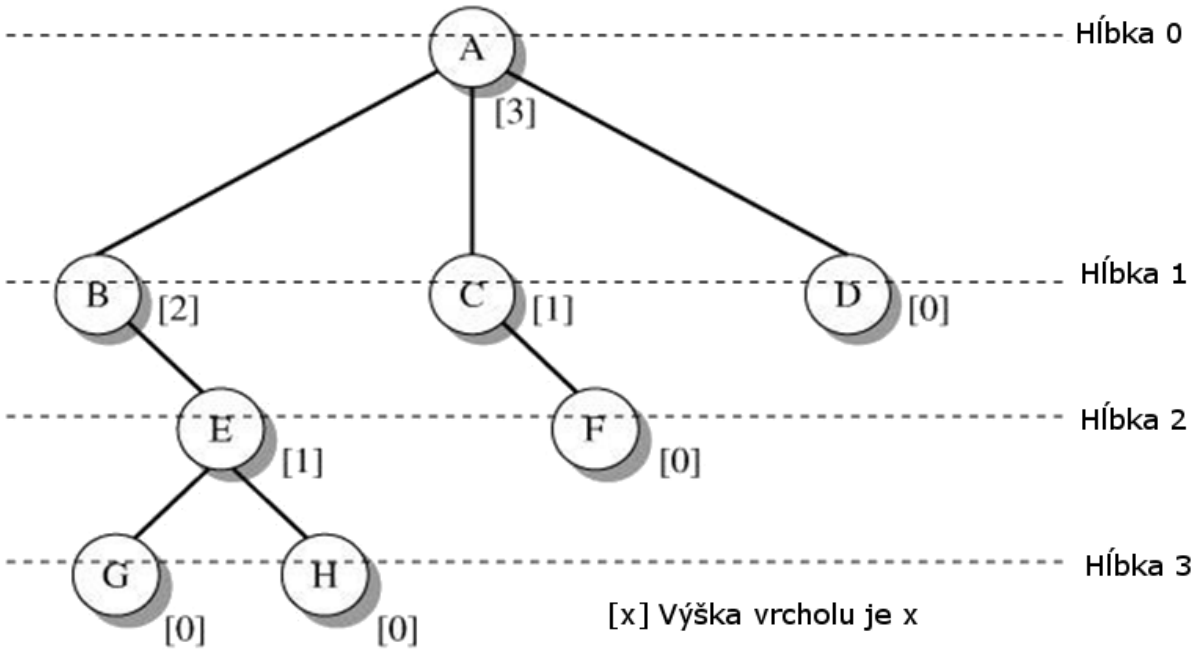


(Zakoreňený) strom – hĺbka, výška

Hĺbka vrcholu – počet hrán od koreňa stromu k danému vrcholu

Výška vrcholu – dĺžka najdlhšej cesty z daného vrcholu k listu (koncovému vrcholu)

Výška stromu – výška jeho koreňa



Strom – rekurzívna definícia

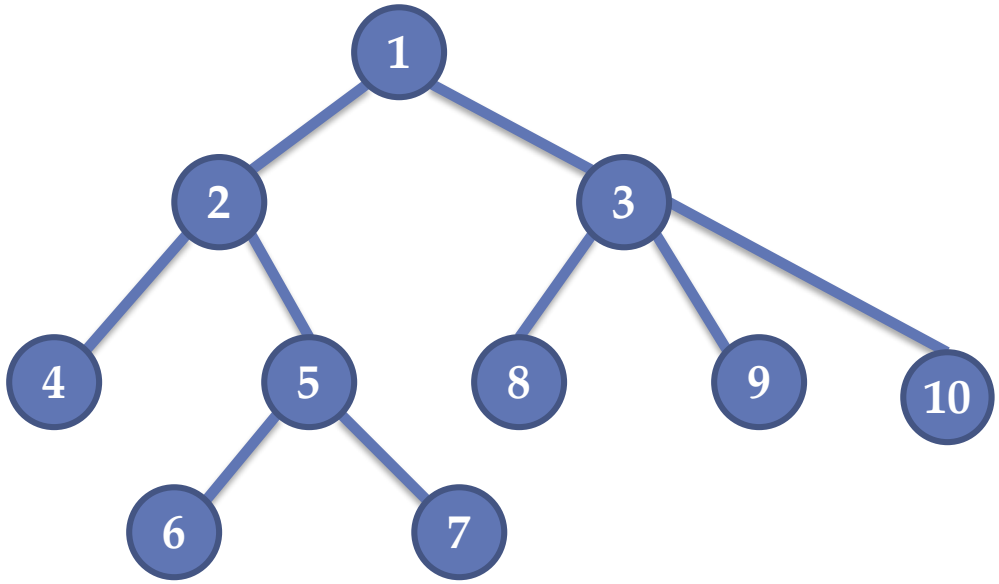
1. Jeden vrchol je strom – tento vrchol je zároveň koreň tohto stromu
2. Nech **V** je vrchol a $\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}$ sú stromy s koreňmi $\mathbf{V_1}, \mathbf{V_2}, \ldots, \mathbf{V_n}$ potom nový strom môžeme zostrojiť tak, že vrchol V urobíme **PREDCHODCOM** vrcholov $\mathbf{V_1}, \mathbf{V_2}, \ldots, \mathbf{V_n}$.
   V tomto novom strome je:
   vrchol V koreň a $\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}$ sú jeho podstromy a vrcholy $\mathbf{V_1}, \mathbf{V_2}, \ldots, \mathbf{V_n}$ sú **NASLEDOVNÍCI** vrcholu V.

ADT Strom – Formálna špecifikácia operácií

- **EMPTY** – vytvorenie prázdneho stromu
- **$EMPTY_N$** – nekonečná rodina operácií.
  - **$EMPTY_i(a, S_1, S_2, ..., S_i)$** vytvorí nový vrchol V s hodnotou a, ktorý má i nasledovníkov – sú to korene stromov S_1, ..., S_i
- **KOREŇ** – Nájdenie koreňa stromu
- **PREDCHODCA** – Nájdenie predchodcu daného vrcholu
- **LNASLEDOVNIK** – Nájdenie najľavejšieho nasledovníka
- **PSUSED** – Nájdenie vrcholu, ktorý má rovnakého predchodcu ale v usporiadaní stromu je vpravo za daným vrcholom.
- **HODNOTA** – Získanie Ohodnotenia vrcholu

Strom – Reprezentácia poľom

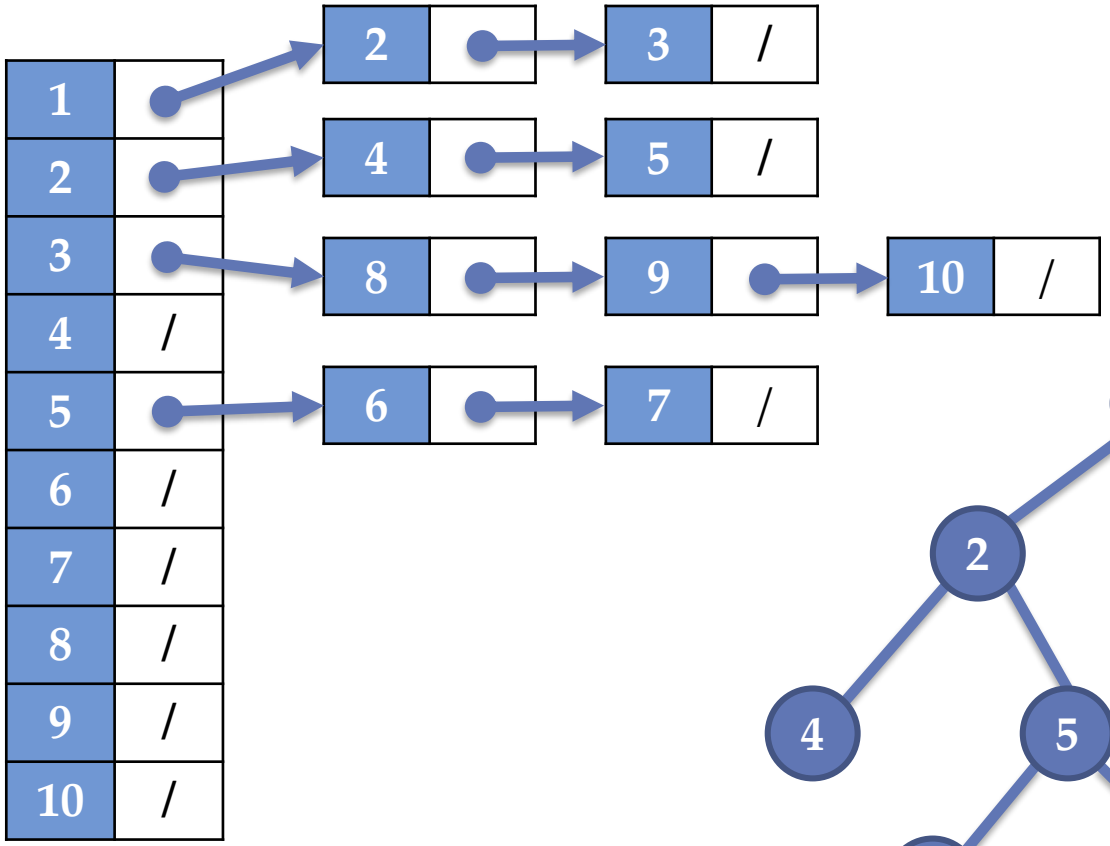
- Index do poľa = číslo vrcholu
- Hodnota prvku poľa = ukazovateľ na rodiča

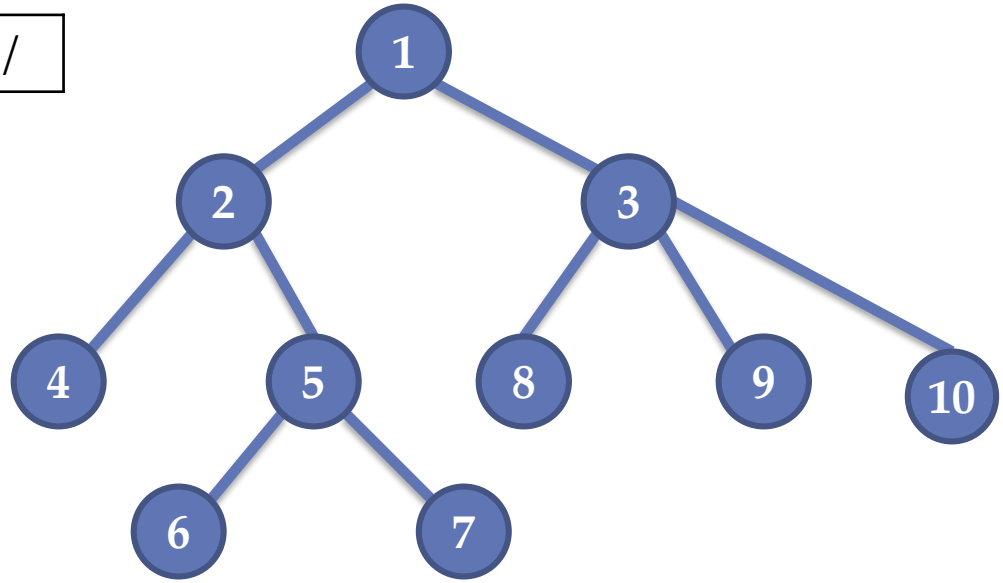


pole[i]	0	1	1	2	2	5	5	3	3	3
index	1	2	3	4	5	6	7	8	9	10

Strom – Reprezentácia spájaným zoznamom

- Každý vrchol má spájaný zoznam priamych nasledovníkov





Rôzne typy stromov (terminologicky)

- Strom (**príroda**)

- Strom (**teória grafov**)
  - Súvislý neorientovaný graf bez cyklov
  - Zakoreňený strom
- Strom (**abstraktná dátová štruktúra**)
  - Reprezentácia hierarchických vzťahov
- Strom (**teória množín**)
  - Čiastočne usporiadaná množina
    (keď nie je nutné, aby sa dali porovnať všetky dvojice prvkov)

Binárny strom

- Strom, v ktorom každý vrchol má najviac **dvoch priamych nasledovníkov** (potomkovia)
- Potomkovia sa označujú ako ĽAVÝ a PRAVÝ
- Rekurzívna definícia:
  - Jeden vrchol je binárny strom a súčasne koreň.
  - Ak u je vrchol a T_1 a T_2 sú stromy s koreňmi v_1 a v_2, tak usporiadaná trojica (T_1, u, T_2) je binárny strom, ak v_1 je **ľavý potomok** koreňa u a v_2 je jeho **pravý potomok**.

Binárny strom – Operácie

- **CREATE**: vytvorenie prázdneho binárneho stromu
- **MAKE**: vytvorenie binárneho stromu z dvoch už existujúcich binárnych stromov a hodnoty
- **LCHILD**: vrátenie ľavého podstromu
- **DATA**: vrátenie hodnoty koreňa v danom binárnom strome
- **RCHILD**: vrátenie pravého podstromu
- **ISEMPTY**: test na prázdnosť

Binárny strom – Formálna špecifikácia

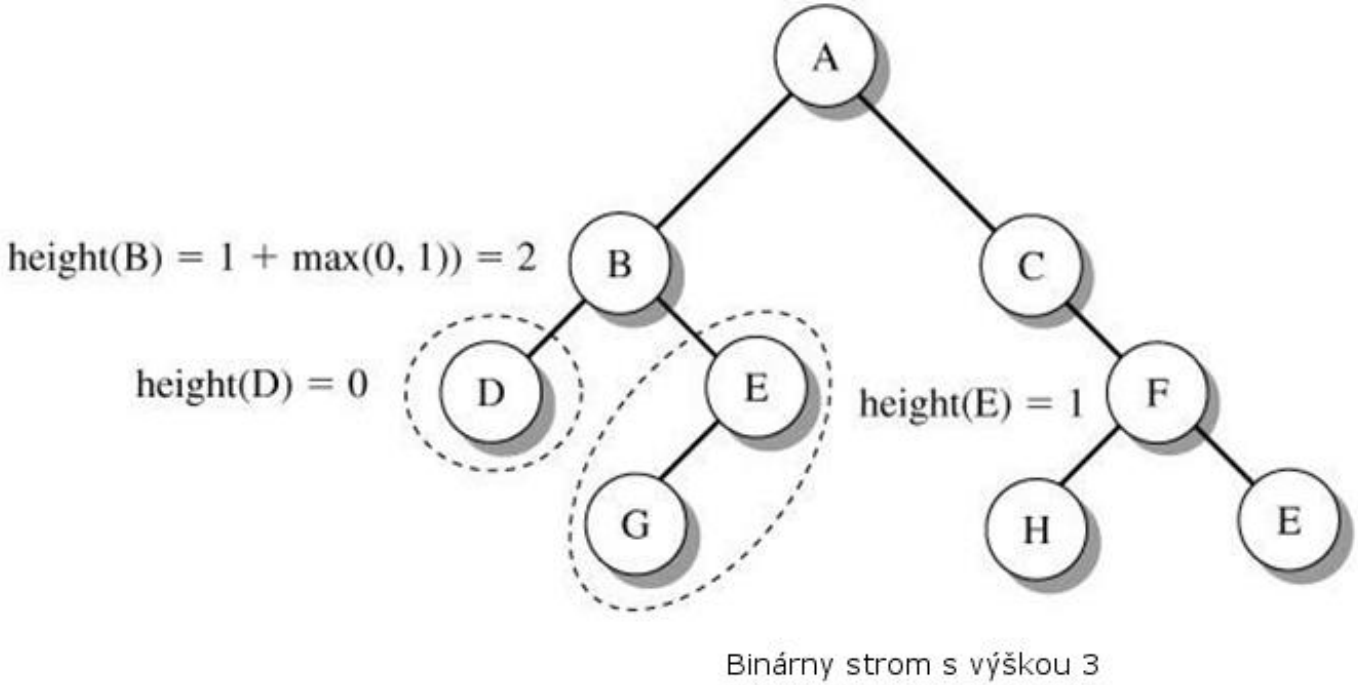
CREATE() → bintree
MAKE(item,bintree,item) → bintree
LCHILD(bintree) → bintree
DATA(bintree) → item
RCHILD(bintree) → bintree
ISEMPTY(bintree) → boolean

Pre všetky p,r ∈ bintree, i ∈ item platí:
ISEMPTY(CREATE) = true
ISEMPTY(MAKE(p,i,r)) = false
LCHILD(MAKE(p,i,r)) = p
LCHILD(CREATE) = error
DATA(MAKE(p,i,r) = i
DATA(CREATE) = error
RCHILD(MAKE(p,i,r)) = r
RCHILD(CREATE) = error

Binárny strom – Výpočet výšky stromu

- Výšku (height) stromu je možné vypočítať rekurzívne:

$$\text{výška}(T) = \begin{cases} -1 & \textit{ak podstrom T je prázdny} \\ 1 + \max(\text{výška}(T_L), \text{výška}(T_R)) & \textit{ak podstrom T nie je prázdny} \end{cases}$$



Binárny strom s výškou 3

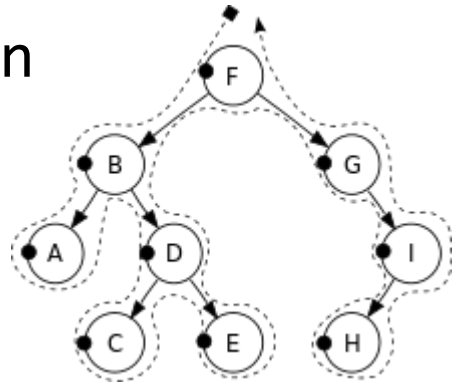
Prehľadávanie binárnych stromov

Tri základné algoritmy:

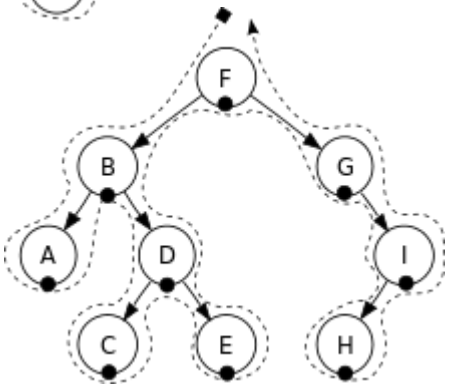
- **pre-order** poradie prehľadávania:
  koreň → ľavý podstrom → pravý podstrom
- **in-order** poradie prehľadávania
  ľavý podstrom → koreň → pravý podstrom
- **post-order** poradie prehľadávania:
  ľavý podstrom → pravý podstrom → koreň

Prehľadávanie binárnych stromov (pseudokód)

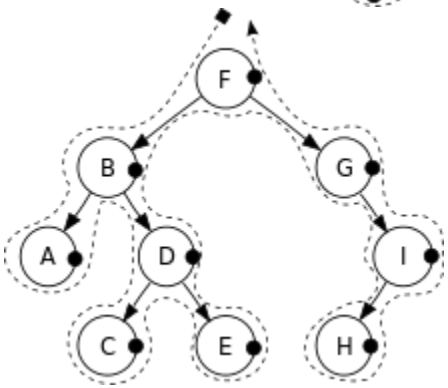
```
PREORDER(T): if T <> nil then
  OUTPUT(DATA(T))
  PREORDER(LCHILD(T))
  PREORDER(RCHILD(T))
```



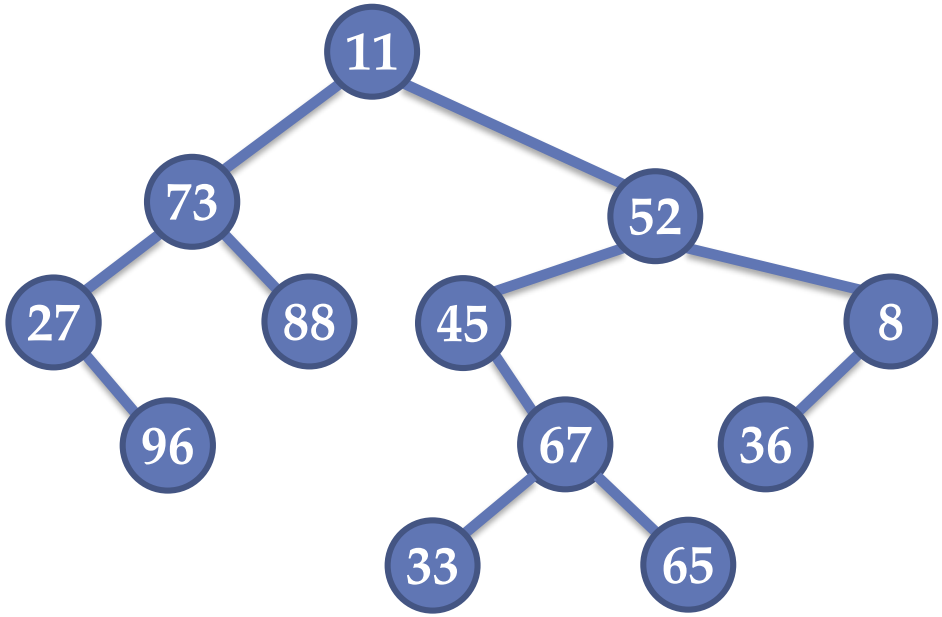

```
INORDER(T): if T <> nil then
  INORDER(LCHILD(T))
  OUTPUT(DATA(T))
  INORDER(RCHILD(T))
```




```
POSTORDER(T) if T <> nil then
  POSTORDER (LCHILD(T))
  POSTORDER (RCHILD(T))
  OUTPUT(DATA(T))
```



Prehľadávanie binárnych stromov (ukážka)



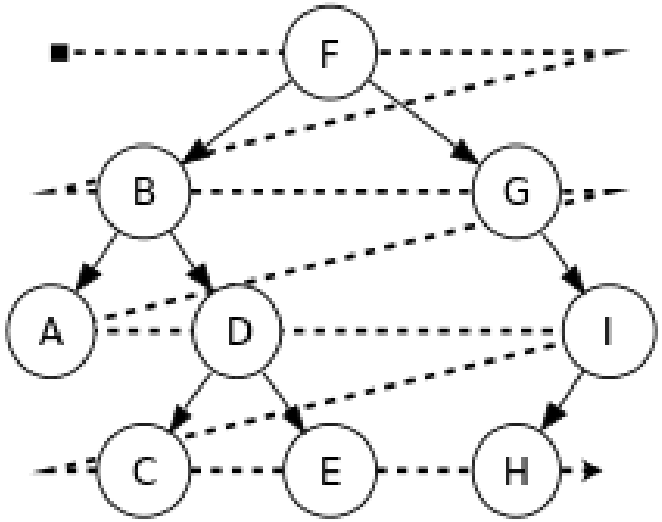
Preorder: 11,73,27,96,88,52,45,67,33,65,8,36

Inorder: 27,96,73,88,11,45,33,67,65,52,36,8

Postorder: 96,27,88,73,33,65,67,45,36,8,52,11

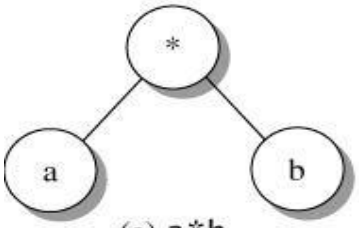
Prehľadávanie po úrovniach (level-order)

- Začína sa koreňom, postupne sa prechádzajú vrcholy po úrovniach (najskôr všetky vrcholy 1. úrovne, potom všetky vrcholy 2. úrovne, atď.)
- Pri prehľadávaní je možné využiť rad(front):
  - Prvý krok: **do radu sa vloží koreň**
  - Opakuje sa, kým nie je rad prázdny: **vyberie sa prvok z radu, zistia sa jeho potomkovia a (pokiaľ existujú) vložia sa do radu**

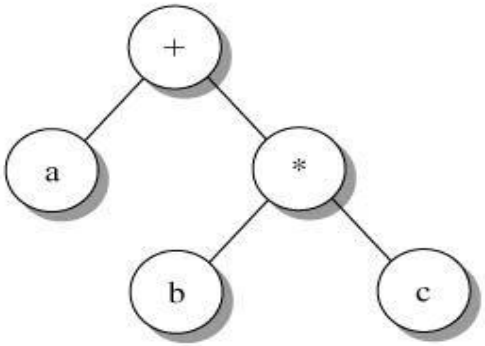


Reprezentácia aritmetických výrazov

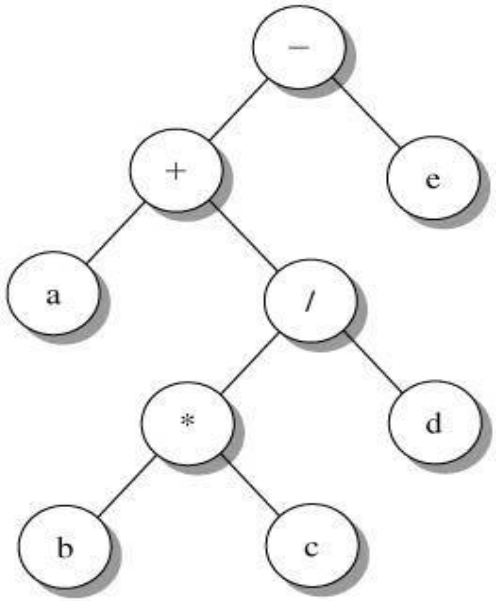
- Základné využitie binárnych stromov v informatike
- Operátor je vnútorný vrchol a jeho potomkom môže byť:
  - Podstrom predstavujúci ďalší výraz
  - Operand



(a) a*b



(b) a + b* c

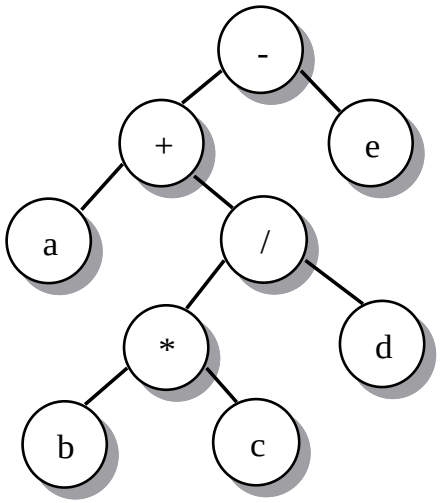


(c) a + b*c/d − e

Prehľadávanie stromu aritmetického výrazu

- Pre-order prechádzanie stromu poskytne prefixový zápis výrazu
- Post-order prechádzanie stromu poskytne postfixový zápis výrazu
- In-order prechádzanie stromu poskytne infixový zápis výrazu (bez zátvoriek)

```
Preorder(Prefix):   - + a / * b c d e

Inorder(Infix):     a + b * c / d – e

Postorder(Postfix): a b c * d / + e -
```



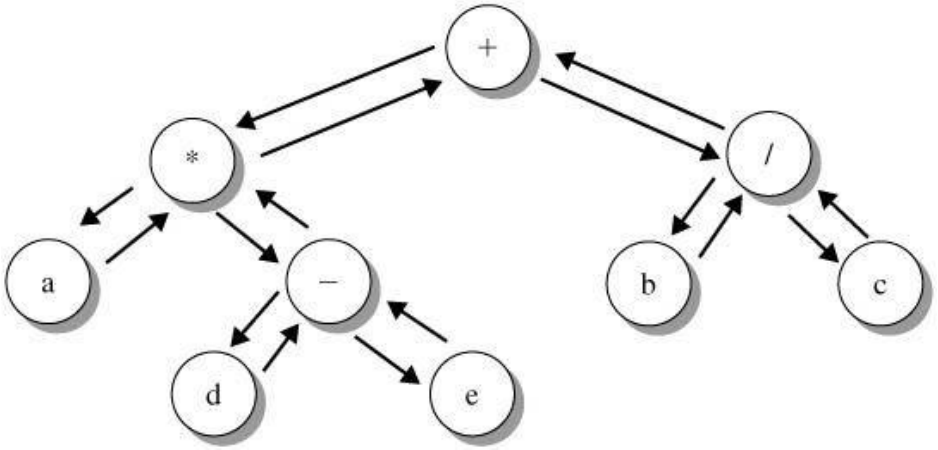
a + b*c/d - e

Eulerov ťah (Euler tour)

- Predchádzajúce spôsoby prechádzali binárny strom vždy tak, že každý vrchol navštívili iba raz.
- Uvažujme **prehľadávanie, ktoré prejde každú hranu raz** (v každom smere)
- Každý vrchol, ktorý má potomkov sa prechádza vždy tri krát:
  - pri prechode od rodiča
  - pri prechode od ľavého potomka
  - pri prechode od pravého potomka

Eulerov ťah v binárnom strome (pseudokód)

```
eulerTour(t):
if t ≠ null
  if t is a leaf node
     visit t
  else
     visit t;
     eulerTour(t.left);
     visit t;
     eulerTour(t.right);
     visit t;
```

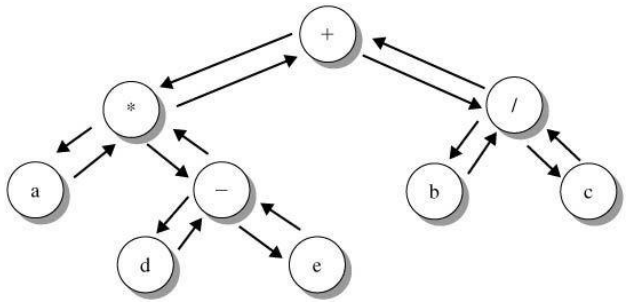


Eulerov ťah: + * a * - d - e - * + / b / c / +

Úplne uzátvorkovaný výraz

- Upravený algoritmus pre Eulerov ťah
- Vstup: matematický výraz reprezentovaný binárnym stromom
- Postup:
  - Pri prechode operandu sa vloží do výstupu operand
  - Pri prechode operátora sa vloží do výstupu:
    - Ľavá zátvorka (pri prechode od rodiča
    - Pravá zátvorka) pri prechode z pravého potomka
    - Operátor pri prechode z ľavého potomka
- Výstup takto upraveného algoritmu:

((a*(d-e))+(b/c))



Binárny vyhľadávací strom (BVS)

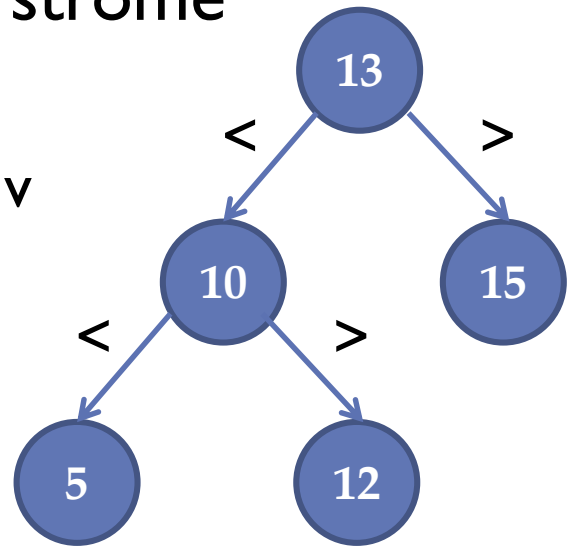
- BVS je binárny strom
- BVS môže byť prázdny
- Ak BVS nie je prázdny, tak spĺňa tieto podmienky:
  - každý prvok má kľúč a všetky kľúče sú rôzne,
  - všetky kľúče v ľavom podstrome sú menšie ako kľúč v koreni stromu
  - všetky kľúče v pravom podstrome sú väčšie ako kľúč v koreni stromu,
  - ľavý aj pravý podstrom sú tiež BVS.

Binárny vyhľadávací strom – Operácie

- Implementácia ADT Dynamická množina
  - **create** – vytvor prázdny strom
  - **insert** – vložiť/pridať prvok do stromu
  - **search** – vyhľadať prvok v strome
  - **delete** – odstrániť prvok zo stromu

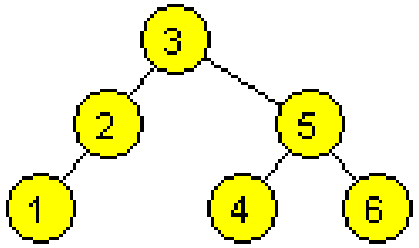
Binárny vyhľadávací strom – intuitívna implemenetácia

- **insert(x)** – vložiť prvok X do stromu
  - Najskôr sa pokúsim X vyhľadať, a potom vložím na miesto kde by mal byť (ale nebol)
- **search(x)** – vyhľadaj prvok X v strome
  1. Začnem v koreni ... vrchol v
  2. Porovnám kľúč X s hodnotou vo v
  3. Ak je rovný, našiel som.
  4. Inak, presuniem sa do príslušného potomka, nastavím ho ako nový v, choď na 2.



Binárny vyhľadávací strom – search (pseudokód)

Rekurzívna verzia




```
bintree TREE-SEARCH(T,k):
if T=nil or k=DATA(T)
      then return T
if k<DATA(T) then return TREE-SEARCH(LCHILD(T),k)
      else return TREE-SEARCH(RCHILD(T),k)
```

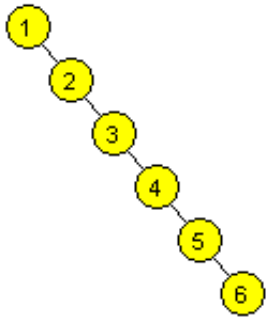
Iteratívna verzia

```
bintree ITERATIVE-TREE-SEARCH(T,k):
while T <> nil and k<>DATA(T) do
      if k<DATA(T) then T ← LCHILD(T)
            else T ←RCHILD(T)
return T
```

Analýza zložitosti – search

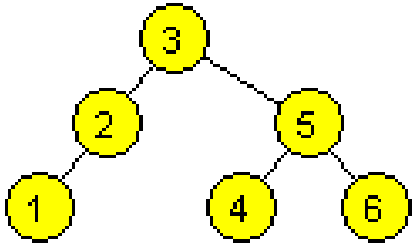
- Závisí od hĺbky h – O(h)
  - Najhorší prípad O(n)

    nájdenie uzla 6

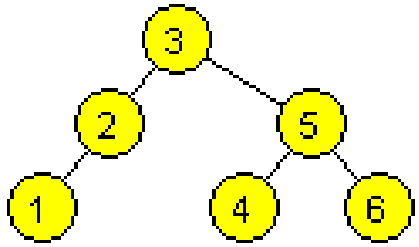


  - Priemerný a zároveň najlepší prípad O(log n)

    nájdenie uzla 6



Binárny vyhľadávací strom – insert (pseudokód)

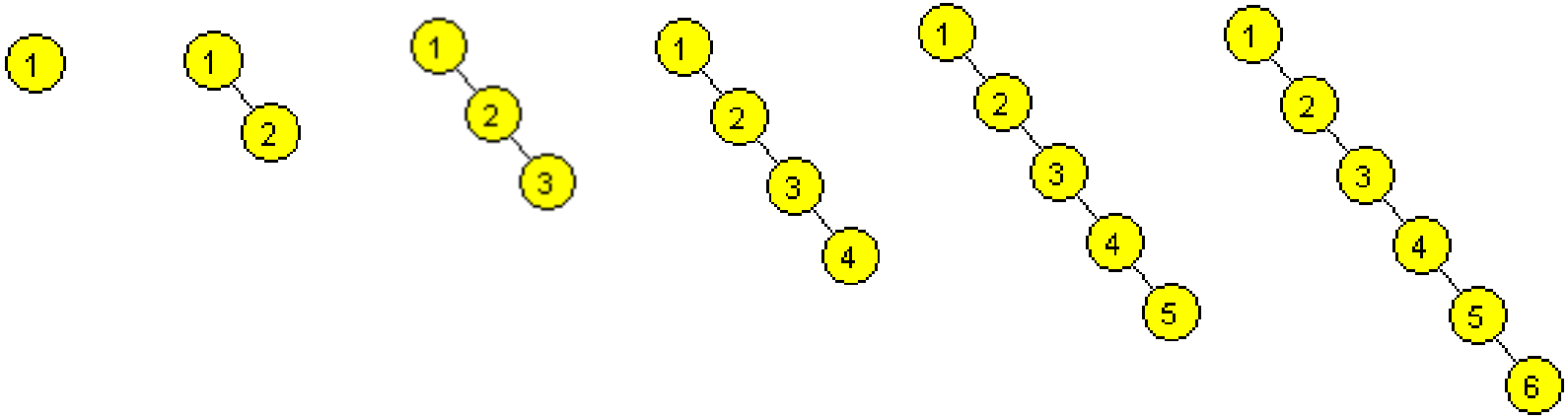


TREE-INSERT(T,n):

```
Y ← nil; X ← ROOT(T)
while X <> nil do
      Y←X
      if DATA(n) < DATA(X) then X ← LCHILD(X)
            else X ← RCHILD(X)
PARENT(n) ← Y
If Y = nil then ROOT(T) ← n
else if DATA(n) < DATA(Y) then LCHILD(Y) ← n
            else RCHILD(Y) ← n
```

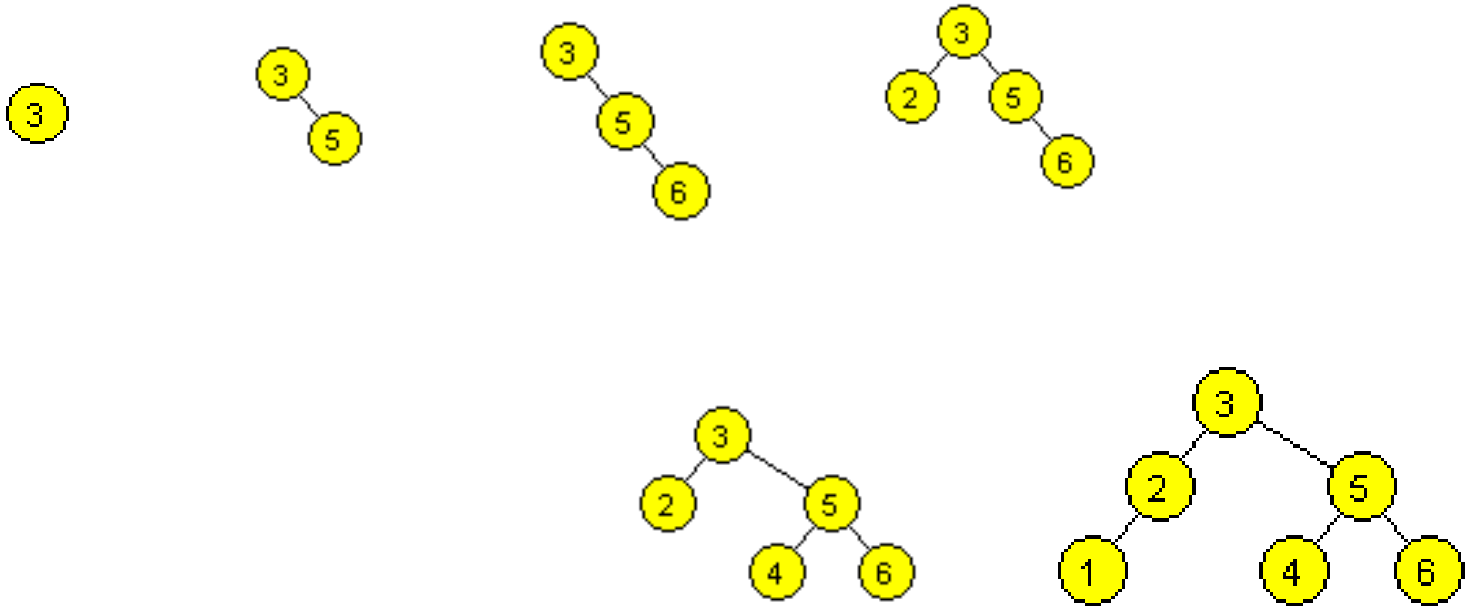
Analýza zložitosti – insert

- Musíme nájsť miesto, kde môžeme prvok vložiť – časová zložitosť závisí od hĺbky stromu – O(h)
  - Najhorší prípad O(n): zoradená postupnosť – vytvárame nevyvážený strom – rýchle zväčšovanie hĺbky stromu Napr. 1, 2, 3, 4, 5, 6



Analýza zložitosti – insert (2)

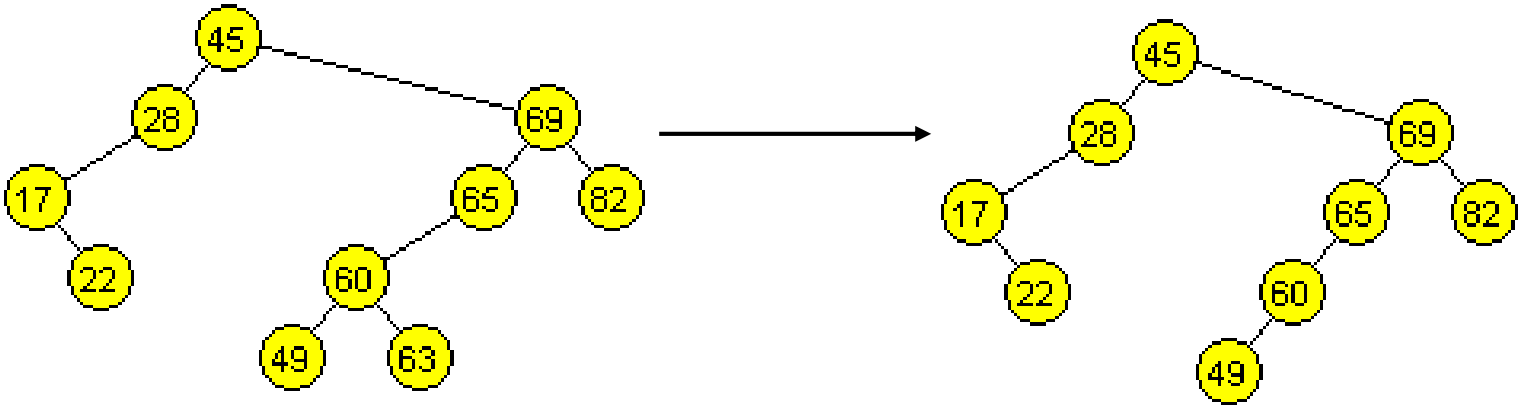
- Priemerný prípad O(log n): náhodná postupnosť – vytvárame väčšinou „dobre“ vyvážený strom – pomalé zväčšovanie hĺbky stromu
  Napr. 3, 5, 6, 2, 4, 1



Binárny vyhľadávací strom – delete

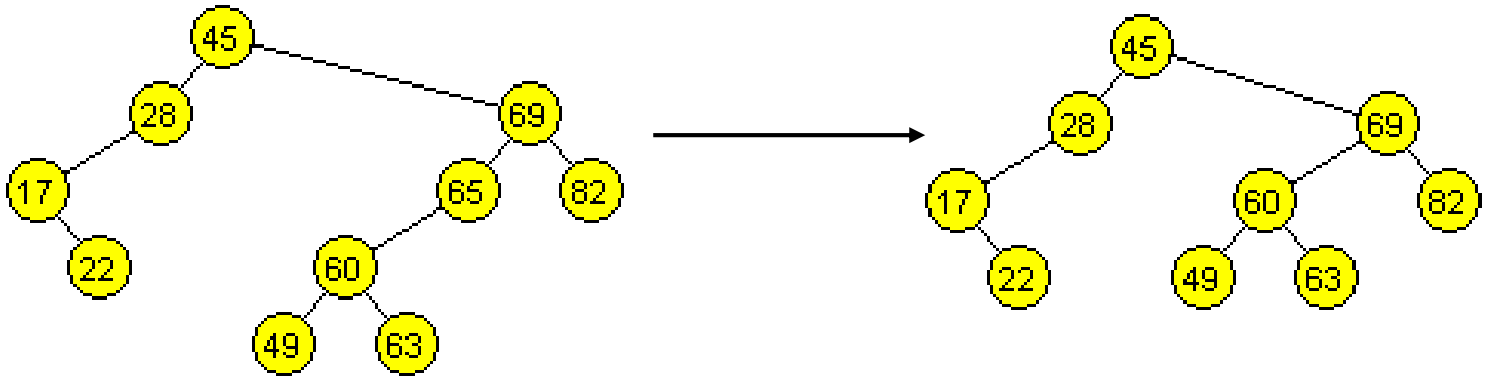
- Môžu nastať tri prípady

1. **vrchol na odstránenie nemá žiadny podstrom:**
   jednoduché odstránenie vrcholu, napr. odstránenie 63



Binárny vyhľadávací strom – delete (2)

2. **vrchol na odstránenie má jeden podstrom:** odstránenie vrcholu, prepojenie koreňa jeho podstromu s jeho rodičom, napr. odstránenie 65

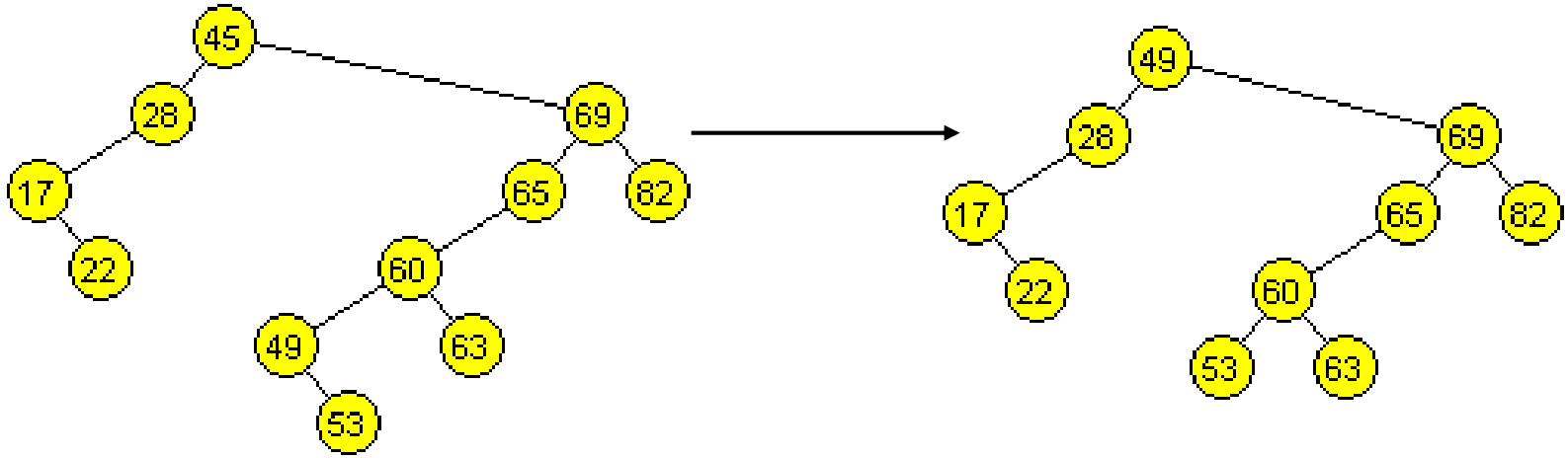


Binárny vyhľadávací strom – delete (3)

3. vrchol V na odstránenie má dva podstromy:

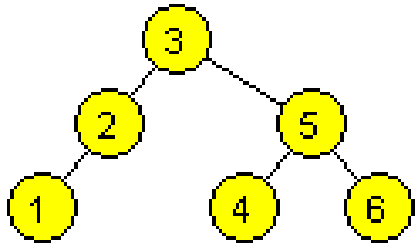
- nájsť za neho náhradu Y (najmenší väčší prvok – successor: najľavejší z jeho pravého podstromu),
- skopírovať kľúč z Y do V, odstrániť zo stromu vrchol Y a (ak existuje) prepojiť jediné dieťa Y s rodičom Y
- (náhrada môže byť aj najväčší menší prvok – predecessor)

Napr. odstránenie 45 (náhrada bude 49)



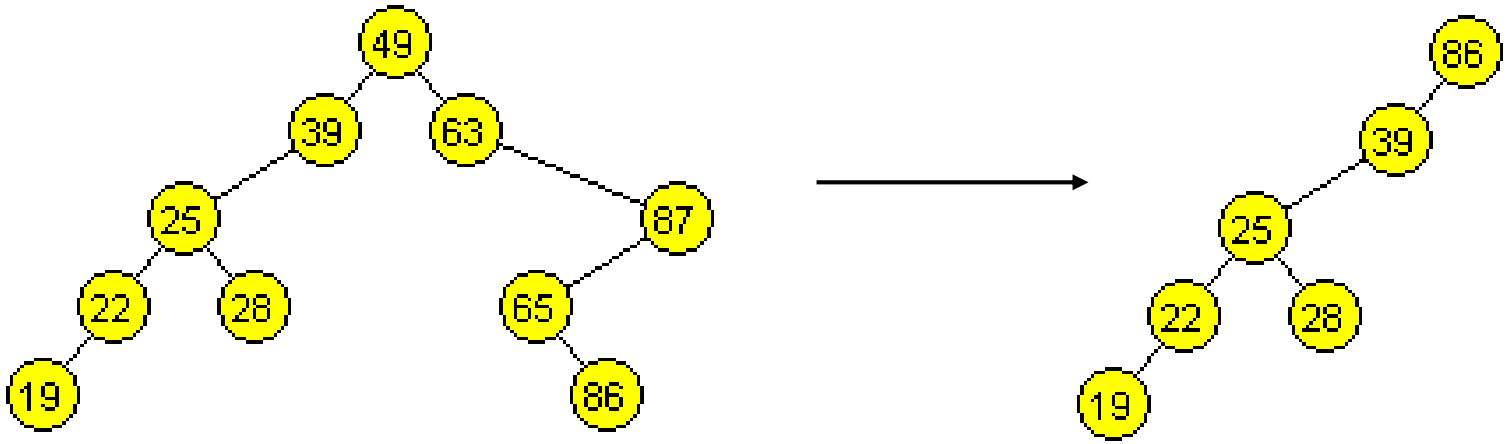
Binárny vyhľadávací strom – delete (pseudokód)

```
bintree TREE-DELETE(T,v):
    if LCHILD(v) = nil or RCHILD(v) = nil then Y ← v
          else Y ← TREE-SUCCESSOR(v)
    if LCHILD(Y) <> nil then X ← LCHILD(Y)
          else X ← RCHILD(Y)
    if X <> nil
          then PARENT(X) ← PARENT(Y)
    if PARENT(Y) = nil then ROOT(T) ← X
          else if Y = LCHILD(PARENT(Y))
                    then LCHILD(PARENT(Y)) ← X
                    else  RCHILD(PARENT(Y)) ←X
    if Y <> v
          then DATA(v) ← DATA(Y)
return Y
```



Analýza zložitosti – delete

- Musíme nájsť vrchol, ktorý chceme odstrániť a vrchol, ktorý sa stane náhradou – časová zložitosť závisí od hĺbky stromu – O(h)
- Odstraňovanie vrcholov spôsobuje nevyváženosť stromu, pretože vždy vyberáme ako náhradu nasledovníka – počet v pravom podstrome sa znižuje, počet v ľavom podstrome zostáva rovnaký
- preto najhorší prípad má zložitosť O(n), ináč v priemere je to O(log n)
- Napr. po odstránení vrcholov 49, 63, 65, 87, 65



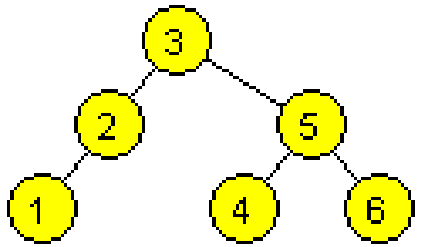
Binárny vyhľadávací strom – Ďalšie operácie

- BVS je NIELEN implementácia ADT Dynamická množina
- Navyše máme binárna reláciu usporiadania < kľúčov
- Ďalšie operácie BVS:
  - **min** / **max** – vyhľadať najmenší / najväčší prvok
  - **successor** – vyhľadať najbližší väčší prvok
  - **predecessor** – vyhľadať najbližší menší prvok

Binárny vyhľadávací strom – min / max (pseudokód)

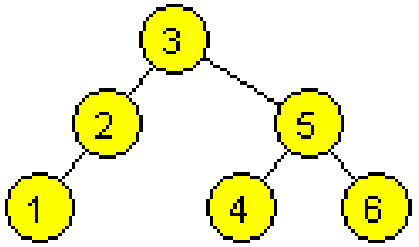
```
bintree TREE-MINIMUM(T):
    while LCHILD(T) <> nil do
                T ← LCHILD(T)
    return T
```

```
bintree TREE-MAXIMUM(T):
    while RCHILD(T) <> nil do
                T ← RCHILD(T)
    return T
```



Binárny vyhľadávací strom – successor (pseudokód)

```
bintree TREE-SUCCESSOR(T):
    if RCHILD(T) <> nil
        then return TREE-MINIMUM(RCHILD(T))
    S ← PARENT(T)
    while S <> nil and T = RCHILD(S) do
        T←S
        S ← PARENT(T)
    return S
```



Triedenie binárnych vyhľadávacím stromom

- Štruktúra vrcholov v BVS umožňuje skonštruovať jednoduchý algoritmus usporadúvania tzv. **Treesort**
- In-order prehľadávanie – usporiadaný výpis obsahu BVS
  - Zložitosť O(n), kde n je počet prvkov v strome

- Máme teda porovnávací algoritmus, ktorý dokáže usporiadať n čísel na vstupe rýchlejšie ako O(n log n)?
  - Nie, pretože vytvorenie BVS trvá O(n log n)
  - Všimnime si, že štruktúra prvkov v BVS je „ekvivalentná“ samotnému problému usporiadania, pretože keď už máme BVS, tak dokážeme výsledné poradie získať v O(n)

Implementácia v jazyku C (ukážka)

- Dve štruktúry: strom, prvok stromu (vrchol)

```
struct Strom
{
    int pocet;
    struct Vrchol *koren;
};

struct Vrchol
{
    int hodnota;
    struct Vrchol *lavy, *pravy;
};


struct Strom *strom_vytvor()
{
    struct Strom *s = (struct Strom *)malloc(sizeof(struct Strom));
    s->pocet = 0;
    s->koren = NULL;
    return s;
}
```

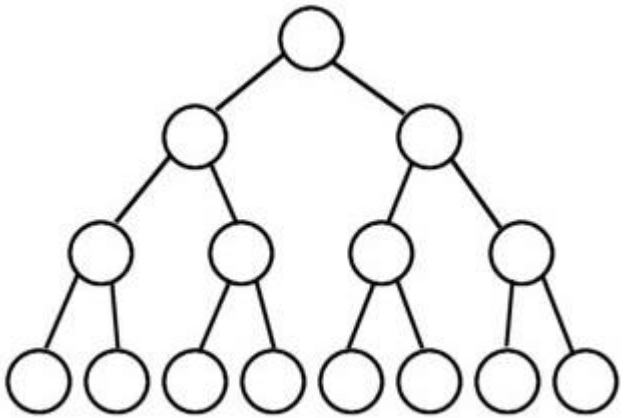
Implementácia v jazyku C (testovanie)

- Ako si rýchlo otestujem moju implementáciu?
  - Vložím tam náhodné čísla a vypíšem ich (mali by byť usporiadané – Treesort)

```c
int main(void)
{
    struct Strom *s = strom_vytvor();

    int i;
    for (i = 0; i < 50; i++)
        strom_pridaj(s, rand()%1000);

    strom_vypis(s);

    return 0;
}
```

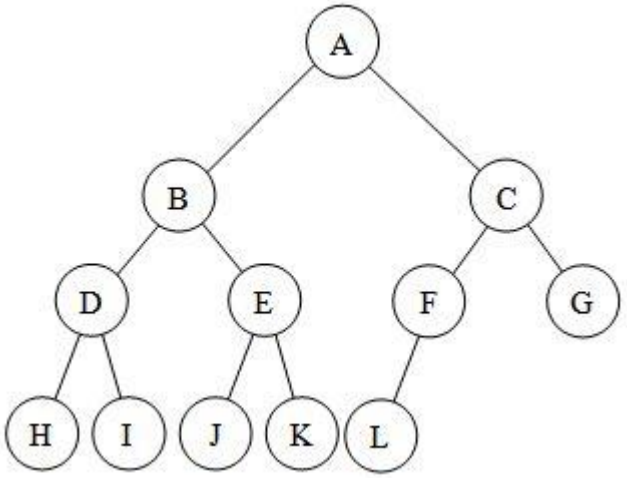
Efektivita vykonania operácií BVS

- Zložitosť operácií nad BVS lineárne závisí od hĺbky stromu – O(h)
- Operácie pracujú lepšie keď je strom „vyvážený“
- Najlepšie, aby bol takýto (tzv. úplný binárny strom)



Halda (heap)

- Využíva špeciálny typ binárneho stromu
  - Tzv. úplný binárny strom (complete binary tree)
  - Na každej úrovni je úplne naplnený, okrem možno poslednej úrovne (rozdiel oproti plnému binárnemu stromu)
- Halda poskytuje len obmedzené operácie
  - **insert** (pridať prvok), **delete** (odstrániť prvok)
  - **getmax** (vyhľadať najväčší prvok)
- Implementácia pomocou BVS:
  - insert / delete **O(n)**, getmax **O(1)**
- Cieľ je vyváženejšia zložitosť:
  - insert / delete / getmax **O(log n)**



ADT Prioritný rad / front (Priority queue)

- Množina prvkov, ktorým je pridelená priorita (kľúč) – je ich možné podľa priority porovnávať.
- Prvky je možné **vkladať v akomkoľvek poradí** s rôznou prioritou avšak, pri výbere sa **vyberá vždy len prvok s najvyššou prioritou**.

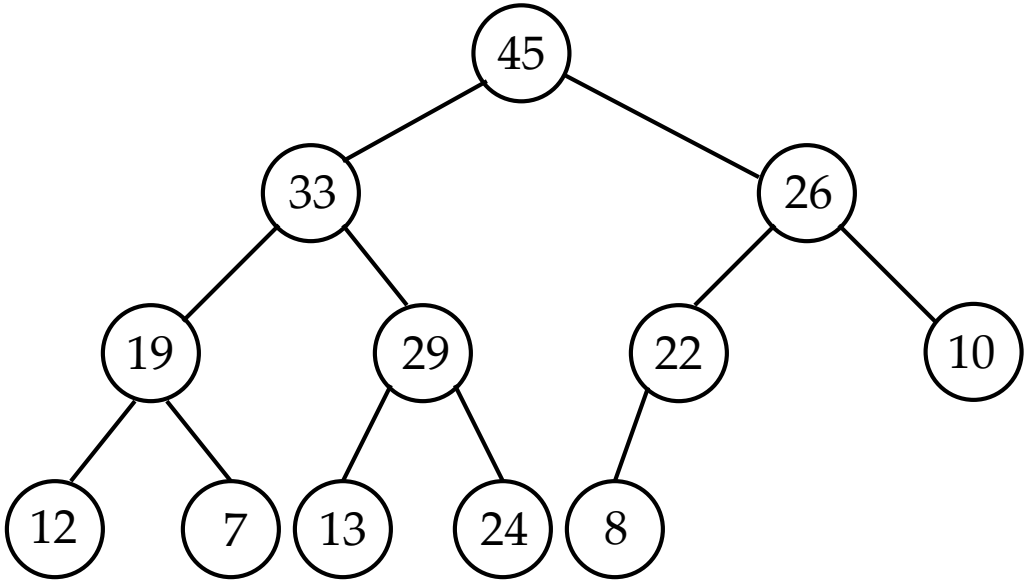
- Operácie:
  - **insert(S,x)** – vloženie prvku x do množiny S
  - **maximum(S)** – vrátenie prvku s najväčším kľúčom
  - **removeMax(S)** – odstránenie prvku s najväčším kľúčom

Implementácia pomocou spájaného zoznamu

- **insert** – pridávanie prvkov na začiatok zoznamu
  - **O(1)**
- **maximum** / **removeMax** – nájdenie prvku s najväčšou prioritou, ten sa vymaže
  - **O(n)**

Implementácia pomocou binárnej haldy

- Binárna halda je úplný binárny strom, pre ktorý platí, že **hodnota kľúča vo vrchole je väčšia alebo rovná hodnotám kľúčov jeho nasledovníkov**



Vlastnosti binárnej haldy

- Binárna halda má voľnejšie pravidlá usporiadania kľúčov (umiestnenie prvkov) ako binárny vyhľadávací strom
- Nemusí platiť, že ľavý podstrom obsahuje prvky s nižšími hodnotami kľúčov ako pravý podstrom!
- Platí tzv. **haldová vlastnosť**:

kľúč(PARENT(i)) ≥ kľúč(i)
pre všetky vrcholy i okrem koreňa

Dôsledok: koreň stromu (binárnej haldy) má vždy najväčšiu hodnotu kľúča (≥ ako ostatné vrcholy).

Binárna halda – Implementácia vektorom

- Koreň stromu na 1. pozícii **`heap[1]`**
- Nasledovníky vrchola zapísaného na i-tej pozícii vektora sú (ak existujú):
  - **`left(i) = 2*i`**
  - **`right(i) = 2*i + 1`**
  - **`parent(i) = |i/2|`**
- **`heap[i..j]`**, kde `i>=1`, je binárna halda práve vtedy, ak každý prvok nie je menší ako jeho nasledovníky.

- Operácia **maximum** – prvok `heap[1]`
  - O(1)

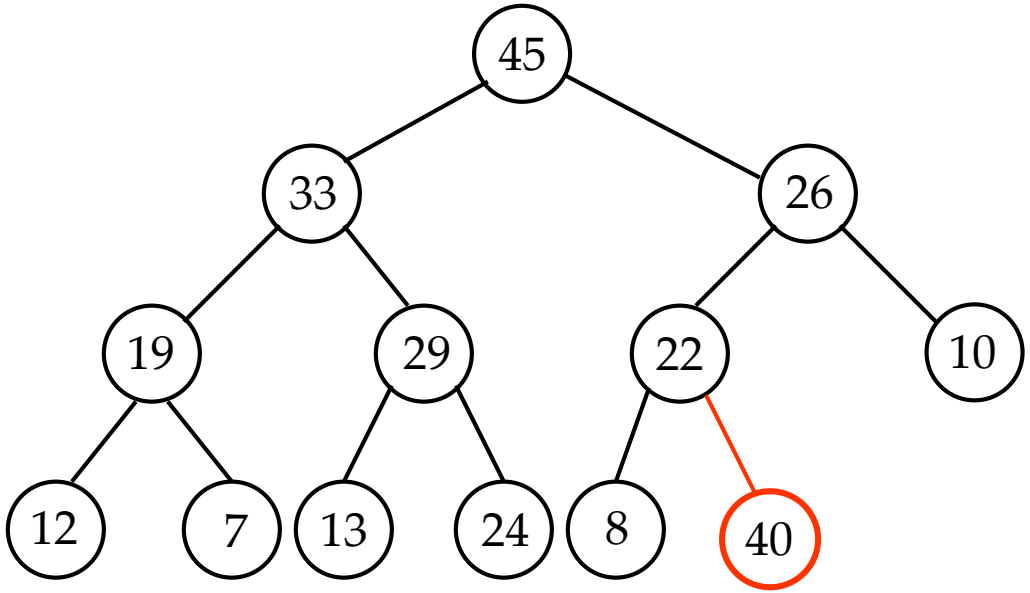
Binárna halda – Operácia insert

1. Vytvorí sa nový vrchol na najnižšej úrovni, označme **v**
2. Ak je **v** koreň stromu (haldy), končíme.
3. Ak kľúč(**v**) ≤ kľúč(parent(**v**)), končíme.
4. Inak (ak kľúč(v) > kľúč(parent(**v**))), vymeníme vrchol **v** so svojím rodičom, a
   pokračujeme na krok 2 pre **v** ← **parent(v)**

   (Ak je kľúč vrchola **v** väčší ako kľúč nového rodiča, vymení sa aj s ním, … opakujeme, kým nie je strom opäť haldou – spĺňa haldovú vlastnosť)

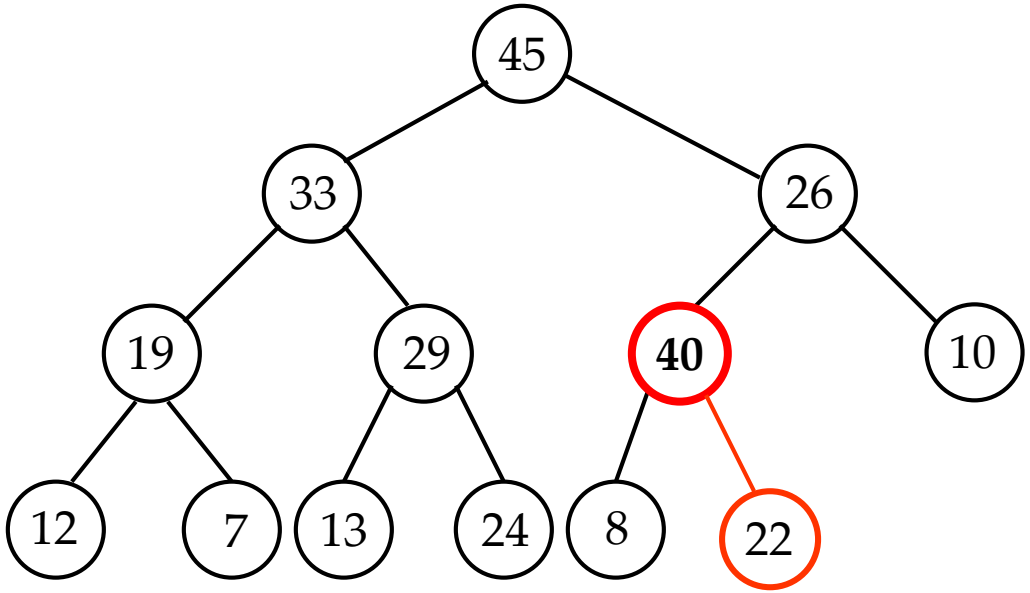
Binárna halda – Vykonanie insert(40)

- Vloženie na najnižšiu úroveň



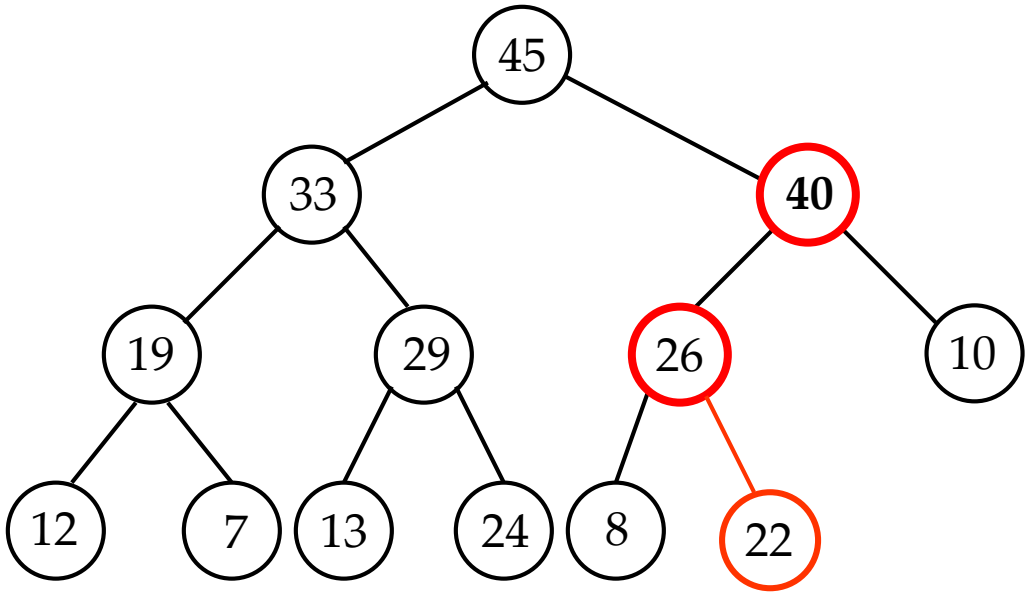
Binárna halda – Vykonanie insert(40)

- Prvá výmena s rodičom (22)



Binárna halda – Vykonanie insert(40)

- Druhá výmena s rodičom (26)



- Hotovo – obnovená haldová vlastnosť
  - v každom vrchole platí **kľúč(PARENT(i))** ≥ **kľúč(i)**

Binárna halda – Insert (pseudokód)

```
Heap-INSERT(heap, key):
  heap-size (heap) = heap-size(heap) + 1
  i = heap-size (heap)
  while i > 1 and heap[PARENT(i)] < key
      do heap[i] = heap[PARENT(i)]
           i = PARENT(i)
  heap[i] = key
```

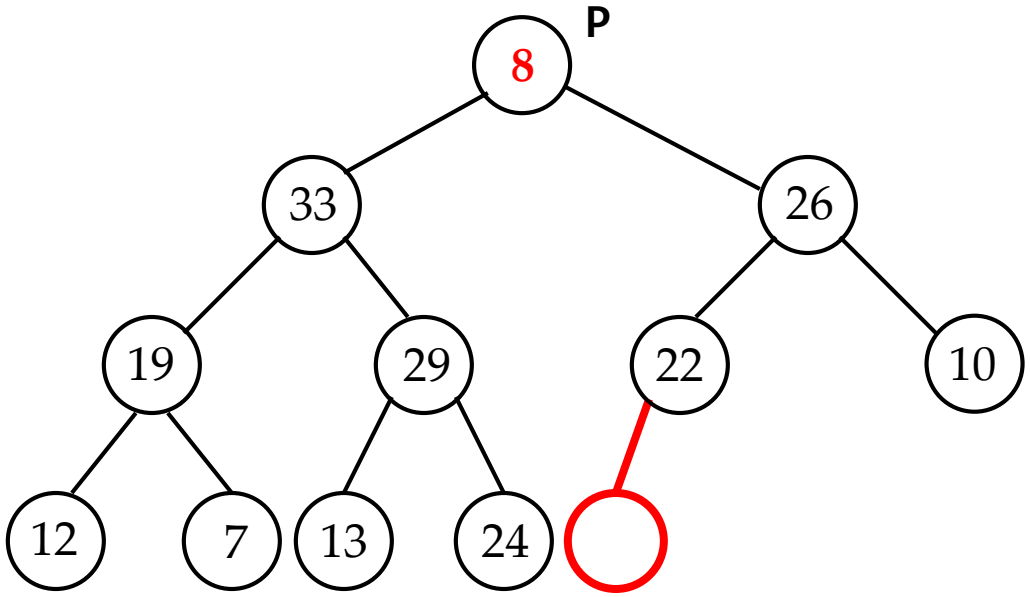
- Zložitosť?
  - O(log n), kde n je počet prvkov v halde
  - Pretože: úplný binárny strom s n prvkami má hĺbku O(log n)

Odstránenie najväčšieho prvku z binárnej haldy

- Odstránime koreň haldy.
- Odstránime najpravejší vrchol na najnižšej úrovni (jeho kľúč označme P) a hodnotu P zapíšeme do koreňa
  - Mohli sme porušiť haldovú vlastnosť!
- Obnovíme haldovú vlastnosť smerom dole
  1. Ak P je list, končíme.
  2. Označme Q najväčšiu z hodnôt priamych potomkov P
     Ak Q ≤ P, teda v potomkoch nie sú väčšie kľúče, končíme.
  3. Inak (ak Q > P) vymeníme vrchol P s vrcholom Q, pokračujeme na krok 1 pre nižšie umiestnený vrchol P.

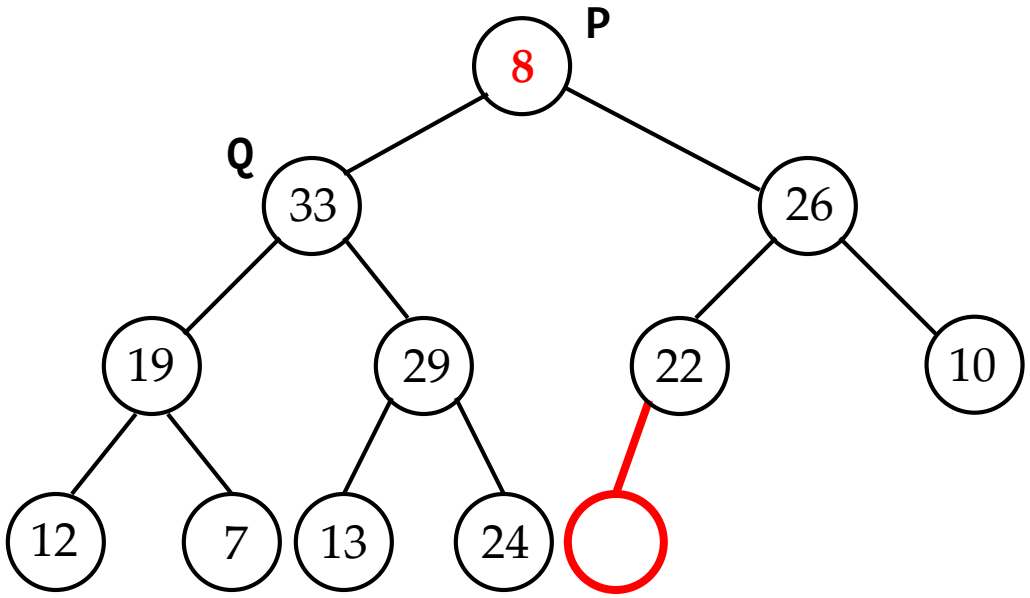
Binárna halda – Vykonanie removeMax

- Odstránime koreň haldy
- Odstránime najpravejší vrchol na najnižšej úrovni (jeho kľúč označme P) a hodnotu P zapíšeme do koreňa



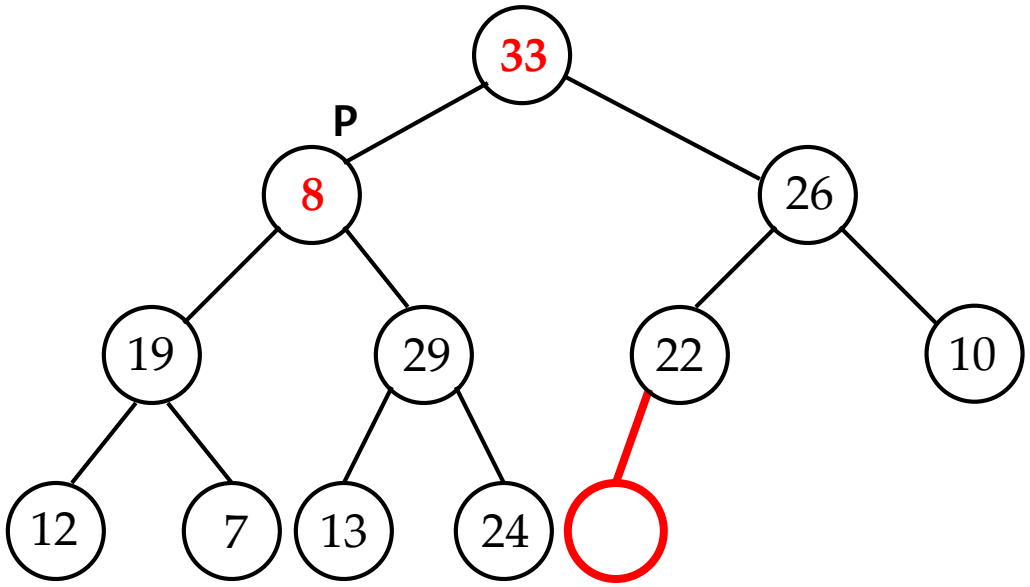
Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P



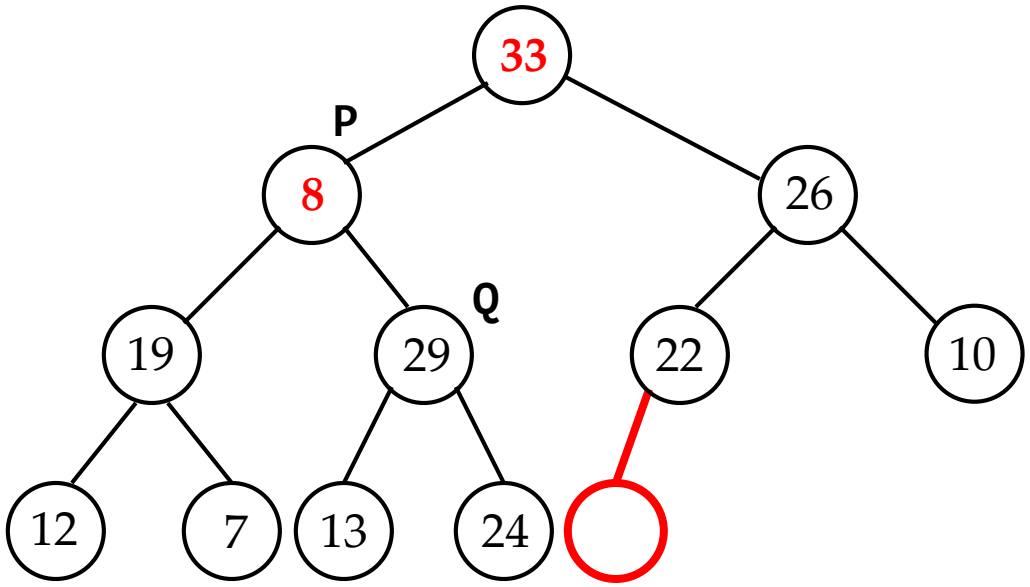
Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q



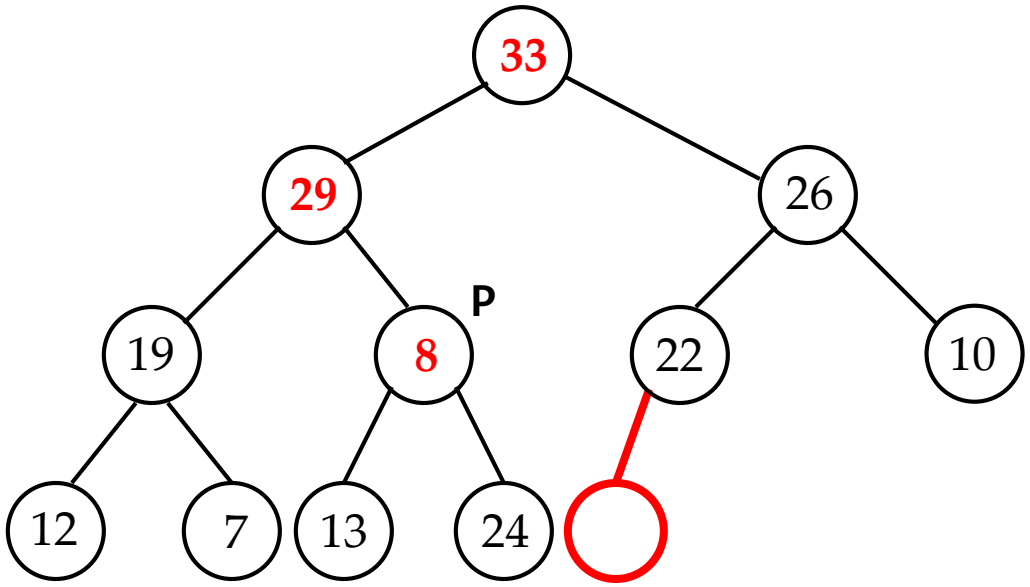
Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P



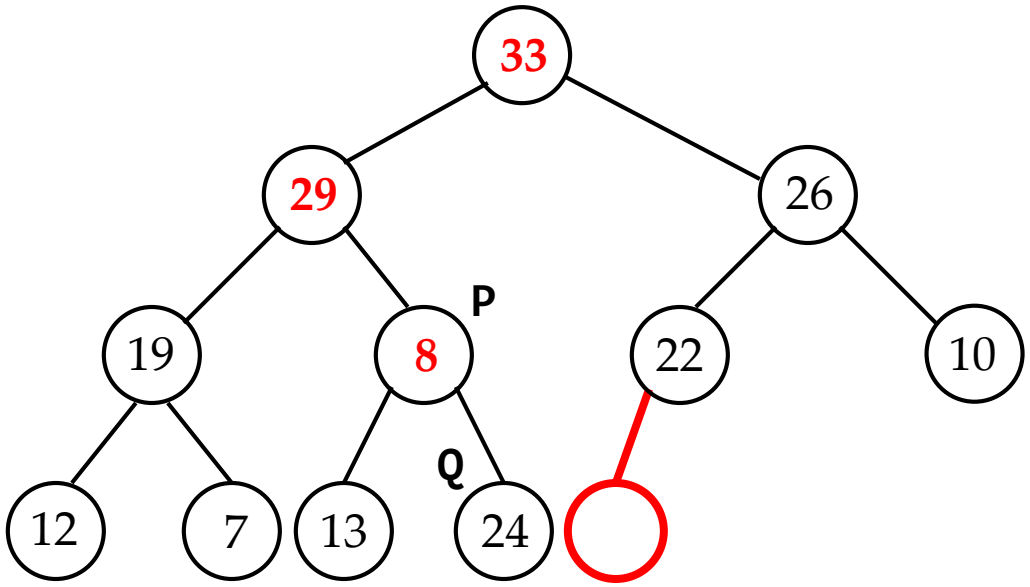
Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q



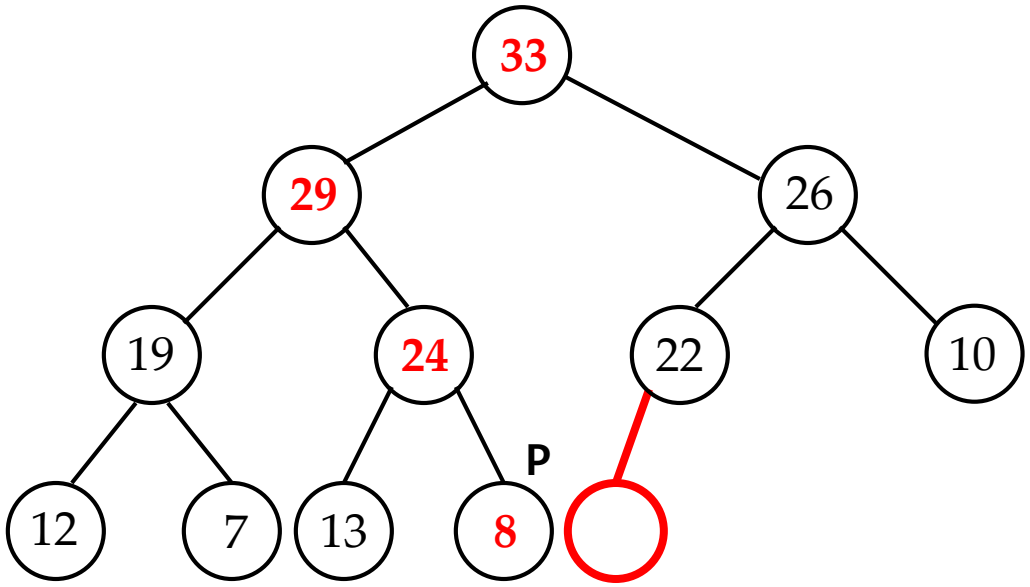
Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
-



Binárna halda – Vykonanie removeMax

- Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q > P vymeníme vrchol P s vrcholom Q
- Ak P je list, končíme.



Binárna halda – extractMax (pseudokód)

```
Heap-EXTRACT-MAX(heap)
  if heap-size(heap) < 1
     then error
  max = heap[1]
  heap[1] = heap[heap-size(heap)]
  heap-size(heap) = heap-size(heap)-1
  HEAPIFY(heap, 1)
  return max
```

Binárna halda – heapify (pseudokód)

```
HEAPIFY(heap, i)
   lavy = left(i)
   pravy = right(i)
   if lavy <= heap-size(heap) and heap[lavy] > heap[i]
       then largest = lavy
       else largest = i
   if pravy <= heap-size(heap) and heap[pravy] > heap[largest]
       then largest = pravy
   if largest <> i
       then exchange (heap[i], heap[largest])
             HEAPIFY(heap, largest)
```

- Zložitosť?
  - O(log n), kde n je počet prvkov v halde

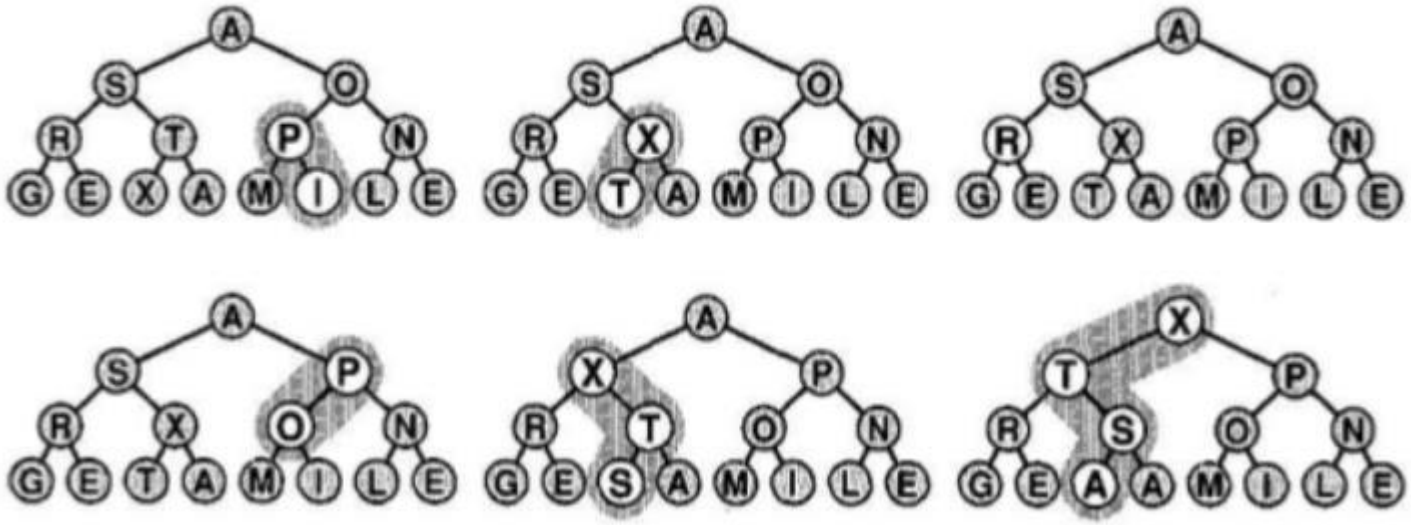
Binárna halda – vytvorenie haldy

- z vektora heap[1..n], kde n=length(heap)
- všetky prvky v podvektore heap[(⌊n/2⌋+1..n] sú listy a teda aj 1-prvkové haldy
- Pseudokód:

```
BUILD-HEAP(heap):
  heap-size(heap) = length(heap)
  for i = ⌊length[heap] / 2⌋ downto 1
     do HEAPIFY(heap, i)
```

- Zložitosť?
  - Vo výške h je najviac $\frac{n}{2^{h+1}}$ vrcholov, heapify haldy výšky h trvá O(h)
  - $T_{\text{BUILD-HEAP}}(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(2n) = O(n)$

Binárna halda – vytvorenie haldy (ukážka)



Usporadúvanie haldou (Heapsort)

- Pomocou haldy dokážeme spraviť efektívny triediaci algoritmus, tzv. **Heapsort**
- Postup: **Vytvorím haldu a postupne z nej vyberiem všetky prvky**
- **HEAP-SORT**(A):

```
    BUILD-HEAP(A);
    for i = length(A) downto 2 do
    { A[1] ⇔ A[i];
      heap-size(A) = heap-size(A)–1;
      HEAPIFY(A, 1)
    }
```

- Zložitosť?
  - Vytvorenie (O(n)) a n krát vybratie max (n*O(log n)) = O(n log n)

Vyvážené vyhľadávacie stromy

- Prioritný rad/front (halda) nie je implementácia všeobecnej dynamickej množiny
- Ako vylepšiť všeobecné vyhľadávacie stromy?
  - Obmedziť ich štruktúru, aby sme mohli o nej prehlásiť nejaké vlastnosti – napr. že bude vždy nízka výška stromu
  - Z týchto garancií (na veľkosť výšky) vyplynú efektívne zložitosti operácií nad takýmito stromami
- Na získanie najlepšej zložitosti O(log n) musíme zabezpečiť, aby strom po vykonaní operácií (insert, delete) zostal vyvážený – použitie samovyvažovacích stromov ako sú AVL stromy alebo červeno-čierne stromy, ktoré automaticky menia svoju štruktúru tak, aby po týchto operáciách bol rozdiel hĺbok ľavého a pravého podstromu „malý“

Vyvážené vyhľadávacie stromy – Prehľad

- Základné vyvažovanie – podrobne
  - AVL strom
  - Splay strom
- Ostatné: definícia, insert, vlastnosti
  - B stromy
  - (a,b) stromy: 2,3 a 2,3,4 stromy
  - Červeno-Čierne (Red-Black) stormy
  - Váhovo vyvážené
- Optimálne binárne vyhľadávacie stromy
- A ďalšie:
  - Trie – dynamická množina reťazcov
  - Radixový strom, lano, ...

Opakovanie – Problém vyhľadávania

- Vstup:
  - Postupnosť: $a_1, a_2, a_3 ... a_n$
    $k(a_i)$ označíme kľúč k_i prvku a_i
  - Hľadaný kľúč x
  - Čo sú kľúče?

    Definičný obor D – reťazce, reálne čísla, dvojice celých čísel, ...
  - Relácia = (rovnosti) – relácia ekvivalencie nad D
  - Usporiadanie kľúčov < (binárna relácia nad D)

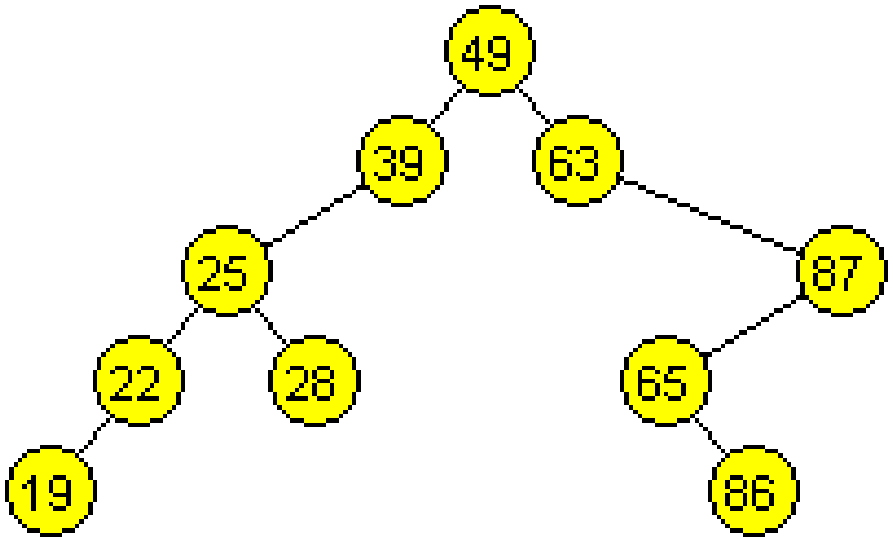
    Lineárne usporiadaná množina K (total ordering)
    Pre $k_1, k_2 \in D$ budeme písať, že $k_1 \leq k_2$ akk $k_1 < k_2$ alebo $k_1 = k_2$.
- Výstup:
  - Index res $\in \{1,2,\ldots,n\}$ takého prvku, že $k(a_{res}) = x$, alebo 0 ak taký prvok neexistuje.

Opakovanie – Základné algoritmy

- Čím viac informácií o vstupnej postupnosti mám k dispozícii, tým rýchlejší algoritmus dokážem vytvoriť
  - Lineárne vyhľadávanie: O(n)
  - Binárne vyhľadávanie: O(log n)
  - Interpolačné vyhľadávanie: O(log log n)
- Binárne vyhľadávacie stromy
  - Priemerný prípad: O(log n)
  - **Najhorší prípad: O(n)**
- Niektoré špecializované typy vyhľadávania
  - Prioritný front (vyhľadávam len najprioritnejší prvok): insert / removeMax : O(log n)

Nová operácia: nájdi k-ty prvok v strome

- Prvé riešenie:
  Využiť in-order usporiadanie, zobrať k-ty prvok
  - Zložitosť O(k)
- Napr. k=5



- In-order: 19, 22, 25, 28, **39**, 49, 63, 65, 86, 87

Nová operácia: nájdi k-ty prvok v strome

- Lepšie riešenie: využiť princíp QuickSelect algoritmu **pri porovnaní vo vrchole pokračovať len v podstrome, v ktorom sa k-ty prvok nachádza**
- Potrebujeme pre každý vrchol x poznať: **počet prvkov v podstrome strome s koreňom x**
- Implementácia ako **rozšírenie štandardnej dátovej štruktúry BVS**, rozšírime údaje pre vrchol:
  - ľavý, pravý, rodič, **počet** (prvkov v podstrome) tzv. **váha**
  - rekurzívna definícia váhy
    váha(v) = váha(ľavýPodstrom(v)) + váha(pravýPodstrom(v)) + 1
- Hodnoty **váha** vo vrcholoch upravujeme pri každej operácií ktorá mení štruktúru stromu: zložitosť O(h), kde h je výška stromu

Order statistic tree

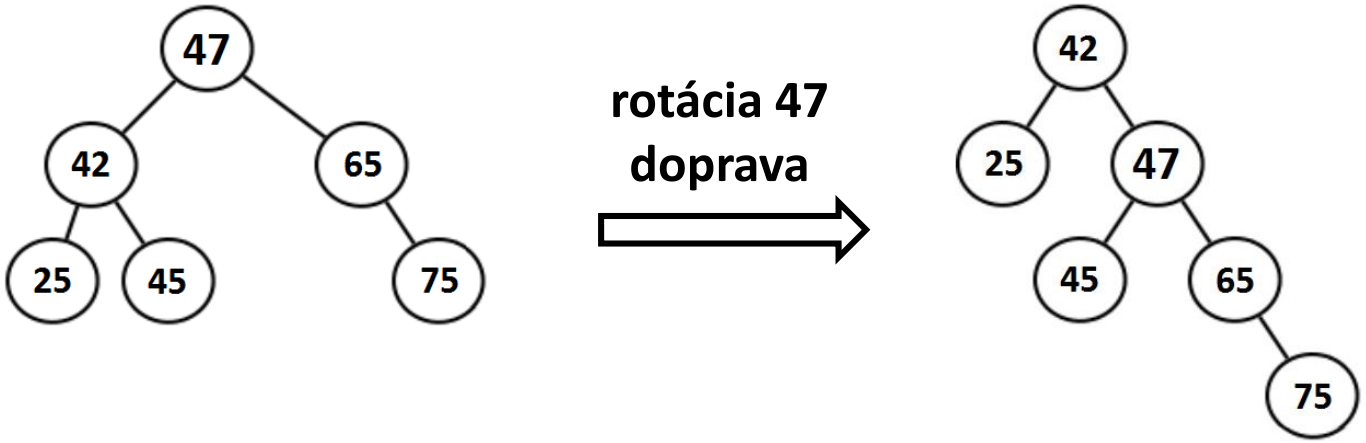
- Rozšírenie BVS stromu
- Pre každý vrchol BVS si navyše pamätáme **počet prvkov v podstrome vrcholu** tzv. váhu
- Hodnoty váhy vo vrcholoch upravujeme pri každej operácií ktorá mení štruktúru stromu (insert, delete)
- Rozšírený strom podporuje navyše operácie:
  - **select(k)** – nájdi k-ty najmenší prvok v množine
  - **rank(x)** – nájdi poradie prvku x v usporiadanej postupnosti prvkov stromu
- Zložitosť operácií O(h), kde h je výška stromu

Ako vylepšiť všeobecné vyhľadávacie stromy?

- Obmedziť ich štruktúru, aby sme mohli o nej prehlásiť nejaké vlastnosti – napr. že bude vždy nízka výška stromu
- Z týchto garancií (na veľkosť výšky) vyplynú efektívne zložitosti operácií nad takýmito stromami
- Na získanie optimálnej zložitosti O(log n) musíme zabezpečiť, aby strom po vykonaní každej operácie zostal vyvážený
- Ako zabezpečiť vyváženie stromu?
  - Hodnoty v strome meniť nemôžeme :)
  - Musíme nejako **upravovať štruktúru** stromu

Rotácia stromu – doprava

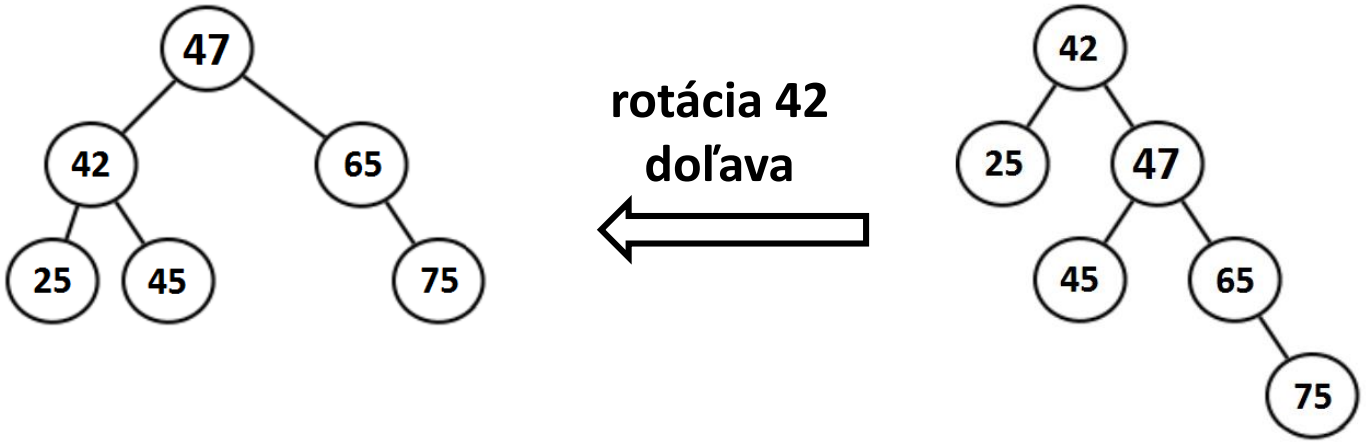
- Operácia, ktorá zmení štruktúru ale zachová usporiadanie
- Zmena tvaru stromu – zmena výšky stromu
- **Rotácia doprava**:
  ľavé dieťa sa presunie (doprava hore) na miesto rodiča



- Zmena hĺbky 25(-1), 42(-1), 47(+1), 65(+1), 75(+1)
- In-order poradie (oba stromy): 25, 42, 45, 47, 65, 75

Rotácia stromu – doľava

- Operácia, ktorá zmení štruktúru ale zachová usporiadanie
- Zmena tvaru stromu – zmena výšky stromu
- **Rotácia doľava**:
  pravé dieťa sa presunie (doľava hore) na miesto rodiča



- Zmena hĺbky 25(+1), 42(+1), 47(-1), 65(-1), 75(-1)
- In-order poradie (oba stromy): 25, 42, 45, 47, 65, 75