1. Dátové štruktúry pre geometrické algoritmy

1.1 Základné definície, asymptotická zložitosť

Algoritmus:
- Konečný návod ako riešiť problém s použitím daných elementárnych operácií.
- Dobre definovaná procedúra, ktorá pre nejakú množinu vstupov vyprodukuje výstupnú hodnotu alebo možinu hodnôt.
- Postupnosť krokov výpočtu, ktorý transformuje vstup na výstup.

Algoritmus nazývame správny, ak pre každý príklad vstupu sa zastaví so správnym výstupom.

Uvažujme mieru zložitosti X (čas, pamäť, počet aritmetických operácií,...)
Zložitosť algoritmu A: je funkcia veľkosti vstupných dát udávajúca množstvo miery zložitosti X spotrebovanej algoritmom A pri riešení problému daného rozsahu.
Ozn: f(n), n – **veľkosť vstupu:** tento pojem závisí od typu problému. Napr.:
- počet prvkov vstupu (napr. triedenie)
- počet bitov (násobenie dvoch čísel)
- dva a viac parametrov (napr. Graf: počet vrcholov, počet hrán)

Pozn: My budeme uvažovať časovú zložitosť algoritmu, t.j.
zložitosť algoritmu ≅ čas výpočtu algoritmu (zjednodušene: počet krokov algoritmu).

Môžeme uvažovať zložitosť: najhoršieho, najlepšieho, priemerného prípadu.

Asymptotická zložitosť

Ak uvažujeme čas výpočtu algoritmu pri dostatočne veľkom vstupe, t.j. zaujíma nás miera rastu časovej zložitosti algoritmu, hovoríme o **asymptotickej zložitosti** algoritmu. Väčšinou algoritmus, ktorý má asymtotickú zložitosť nižšiu, bude výhodnejší aj pri vstupoch menšej
veľkosti. Na popis asymptotickej zložitosti algoritmov sa používa asymptotická notácia.

θ-notácia

Uvažujme funkciu *g(n)* definované na N. Potom

θ(g(n)) = {f(n): ∃ kladné konštanty c_1, c_2 a n_0 také, že $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ pre ∀ $n \ge n_0$}

- píšeme f(n) = θ(g(n)), aj keď θ(g(n)) označuje množinu, t.j. f(n) ∈ θ(g(n)).
- Pre $n \ge n_0$ funkcia f(n) je rovná funkcii g(n), až na konštantný faktor.
- Hovoríme: g(n) je asymptoticky tesná hranica pre f(n).
- Každý člen θ(g(n)) je asymptoticky nezáporný, t.j. f(n) ≥ 0 pre všetky dostatočne veľké n.
  Ako dôsledok, aj g(n) musí byť asymptoticky nezáporná , inak by θ(g(n)) bola prázdna.

Príklad 1: Ukážeme, že $\frac{1}{2} n^2 - 3n = \theta(n^2)$.
Musíme nájsť c_1, c_2 a n_0 nezáporné také, že

$c_1 n^2 \le ½\, n^2 - 3n \le c_2 n^2$ pre $\forall\ n \ge n_0$. / $:n^2$.

$c_1 \le ½ - 3/n \le c_2$.

Druhá nerovnosť platí pre všetky $n \ge 1$ pri voľbe $c_2 \ge ½$. Prvá nerovnosť platí pre všetky $n \ge 7$ pri voľbe $c_1 \le 1/14$.
Voľbou $c_1 = 1/14$, $c_2 = ½$, $n_0 = 7$ ľahko overíme, že $½\, n^2 - 3n = \theta(n^2)$.

Príklad 2: Ukážeme, že $6n^3 \ne \theta(n^2)$.
Nech $\exists\ c_2 > 0, n_0 > 0$: $6n^3 \le c_2 n^2$, pre $\forall\ n \ge n_0$.
$n \le c_2/6$, čo je spor.

Príklad 3: Uvažujme kvadratickú funkciu $f(n) = an^2 + bn + c$, kde a, b, c sú konštanty, $a > 0$. Potom $f(n) = \theta(n^2)$.
Nasledujúcou voľbou konštánt: $c_1 = a/4$, $c_2 = 7a/4$, $n_0 = 2.\max(|b|/a, \sqrt{(|c|/a)})$ ľahko overíme, že $0 \le c_1 n^2 \le an^2 + bn + c \le c_2 n^2$ pre $\forall\ n \ge n_0$.
Vo všeobecnosti, pre ľubovoľný polynóm p(n) r-tého stupňa platí: $p(n) = \theta(n^r)$.
Každú konštantu (konštantnú funkciu) – ako polynóm nultého stupňa – môžeme vyjadriť v tvare $\theta(n^0)$ alebo $\theta(1)$.

- ***O-notácia*** (asymptotická horná hranica):

Uvažujme funkciu *g(n)* definované na N. Potom

O(g(n)) = {f(n): $\exists$ kladné konštanty c a n_0 také, že $0 \le f(n) \le cg(n)$ pre $\forall\ n \ge n_0$}

Označenie: $f(n) = O(g(n))$.
Platí: $f(n) = \theta(g(n)) \Rightarrow f(n) = O(g(n))$, t.j. $\theta(g(n)) \subset O(g(n))$.
$an^2 + bn + c = O(n^2)$, ale aj $an + b = O(n^2)$. Stačí zvoliť $c = a + |b|$; $n_0 = 1$.
Ak použijeme O-notáciu na ohraničenie zložitosti najhoršieho prípadu, získame tým horné ohraničenie zložitosti algoritmu pre ľubovoľný vstup veľkosti n. Toto však neplatí pre θ-notáciu.

- ***Ω-notácia*** (asymptotická spodná hranica):

Uvažujme funkciu *g(n)* definované na N. Potom

Ω(g(n)) = {f(n): $\exists$ kladné konštanty c a n_0 také, že $0 \le cg(n) \le f(n)$, pre $\forall\ n \ge n_0$}.

Tvrdenie 1. Pre ľubovoľné dve funkcie f(n) a g(n) platí: $f(n) = \theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.
Ak Ω(g(n)) je zložitosť najlepšieho prípadu, tak Ω(g(n)) je zložitosť algoritmu pri ľubovoľnom vstupe veľkosti n.
Ak povieme, že zložitosť algoritmu je Ω(g(n)), to znamená, že pri ľubovoľnom vstupe veľkosti n je zložitosť algoritmu najmenej konštantný násobok g(n) pre dostatočne veľké n.

Uvedené asymptorické označenia znamenajú, že až na multiplikatívnu konštantu a z asymptotického hľadiska, v prvom prípade funkcia f rastie približne rovnako rýchlo ako g, v druhom prípade f nerastie rýchlejšie ako g a v poslednom prípade f nerastie pomalšie ako g. Keďže konštanty nemajú podstatný vplyv na celkovú efektívnosť algoritmu, uspokojíme sa s asymptotickým odhadom zložitosti.

Hovoríme, že *asymptotická zložitosť algoritmu A* je $\theta(g(n))$, ak pre zložitosť $f(n)$ algoritmu A platí: $f(n) = \theta(g(n))$. Podobne pre O a Ω.

Asymptotická notácia v rovnostiach.

	rovnosť	*interpretácia*
(1)	$n = O(n^2)$	$n \in O(n^2)$
(2)	$2n^2 + 3n + 1 = 2n^2 + \theta(n)$	$2n^2 + 3n + 1 = 2n^2 + f(n)$, kde $f(n)$ je nejaká funkcia z množiny $\theta(n)$.
(3)	$2n^2 + \theta(n) = \theta(n^2)$, t.j. ak sa asympt. not. objaví naľavo rovnosti	Pre $\forall$ funkciu $f(n) \in \theta(n)$ $\exists$ nejaká funkcia $g(n) \in \theta(n^2)$ taká, že $2n^2 + f(n) = g(n)$, pre $\forall n$.

(2) a (3) $\Rightarrow 2n^2 + 3n + 1 = \theta(n^2)$.

- ***o-notácia***

O-notácia môže, ale nemusí byť asymptoticky tesná. Napr. $2n^2 = O(n^2)$ je, ale $2n = O(n^2)$ nie je asymptoticky tesná. o-notáciu používame na označenie horného ohraničenia, ktoré nie je
asymptoticky tesné. Formálna definícia je nasledovná:

o(g(n)) = $\{f(n)$: $\forall\ c > 0\ \ \exists\ n_0 > 0$ také, že $\ 0 \le f(n) < cg(n)$ pre $\forall\ n \ge n_0\}$.

Napr. $2n = o(n^2)$, ale $2n^2 \neq o(n^2)$.
Platí: $\lim_{n\to\infty} f(n)/g(n) = 0$.
Niektorí autory toto uvádzajú ako definíciu o-notácie.

- ***ω-notácia***

ω-notácia označuje dolné ohraničenie, ktoré nie je asymptoticky tesné.
Možnosti definície:
1. $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$.
2. **ω(g(n))** = $\{f(n)$: $\forall\ c > 0\ \ \exists\ n_0 > 0$ také, že $\ 0 \le cg(n) < f(n)$ pre $\forall\ n \ge n_0\}$.

Napr. $n^2/2 = \omega(n)$, ale $n^2/2 \neq \omega(n^2)$.
Platí: $\lim_{n\to\infty} f(n)/g(n) = \infty$, ak limita existuje.

Relácie

Mnohé z relačných vlastností reálnych čísel môžeme aplikovať aj na asymptotické porovnania. Nech $f(n)$ a $g(n)$ sú asymptoticky pozitívne funkcie. Potom platia nasledovné vlastnosti:

- *Tranzitívnosť:* $f(n) = \theta(g(n)) \wedge g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$. // platí aj pre O, Ω, o, ω.
- *Reflexívnosť:* $f(n) = \theta(f(n))$. // platí aj pre O, Ω.
- *Symetria:* $f(n) = \theta(g(n)) \Leftrightarrow g(n) = \theta(f(n))$.
- *Obrátená symetria:* $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$.
  $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$.

Analógia medzi asymptotickým porovnávaním dvoch funkcii a porovnávaním dvoch reálnych čísel:

$f(n) = O(g(n)) \approx a \leq b$
$f(n) = \Omega(g(n)) \approx a \geq b$
$f(n) = \theta(g(n)) \approx a = b$
$f(n) = o(g(n)) \approx a < b$
$f(n) = \omega(g(n)) \approx a > b$

Jedna z vlastností reálnych čísel sa však nedá aplikovať na asymptotickú notáciu.
A síce:
Trichotómia: pre $\forall$ a, b $\in$ R platí práve jeden vzťah: a = b, a < b, a > b.

Nie všetky funkcie sú asymptoticky porovnateľné, t.j. môže nastať, že pre dve funkcie f(n) a g(n) neplatí ani jeden zo vzťahov: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$.
Napr. $f(n) = n$, $g(n) = n^{1+\sin n}$, pretože exponent funkcie g(n) osciluje medzi 0 a 2 nadobúdajúc všetky hodnoty intervalu $\langle 0, 2 \rangle$.

Príklad na algoritmus a jeho zložitosť: Spomenúť druhy triedení a zložitosti pri jednotlivých úkonoch. Quicksort, bubblesort, heapsort, ..., hash table

Cvičenia
1. Nech f(n) a g(n) sú asymptoticky nezáporné funkcie. Pomocou základných definícií dokážte: $\max(f(n),g(n)) = \theta(f(n) + g(n))$.
2. Dokážte, že pre ľubovoľné a, b$\in$ R; b > 0: $(n + a)^b = \theta(n^b)$.
3. Platí: $2^{n+1} = O(2n)$? $2^{2n} = O(2^n)$?
4. Dokážte, že $o(g(n)) \cap \omega(g(n))$ je prázdna množina.

1. 2 Základné dátové štruktúry

Manipulácia so vstupnými, výstupnými dátami a medzivýsledkami často zaberá značnú časť z celkového času výpočtu algoritmu. Účinnosť algoritmu preto do značnej miery závisí od spôsobu uloženia spracovávaných dát. Je známych množstvo dátových štruktúr, ktoré poskytujú rôzne výhody a nevýhody pri manipulácií s dátami.

Množina je jedným zo základných pojmov nie len v matematike, ale aj v informatike. Na rozdiel od matematiky, kde väčšinou pracujeme s množinami s pevným počtom prvkov, množiny, s ktorými pracuje algoritmus, môžu narastať, zmenšovať sa alebo ináč sa meniť v čase. Takéto množiny nazývame *dynamické množiny.* Algoritmy tiež požadujú vykonanie niekoľkých typov operácií na množinách. Napr. schopnosť pridať prvok do množiny, vymazať prvok z množiny, zistiť, či daný prvok patrí do množiny. Implementácia dynamickej množiny závisí teda aj od operácií, ktoré musí poskytovať.
V tejto časti sa budeme zaoberať reprezentáciou dynamických množín pomocou jednoduchých dátových štruktúr a operáciami, ktoré tieto dátové štruktúry poskytujú. V prípade niektorých štruktúr je potrebné každý prvok reprezentovať jednoznačnou hodnotou z množiny, ktorá je usporiadaná (najčastejšie sú to reálne čísla). Túto jednoznačnú hodnotu nazývame kľúč. Keď máme pre každý prvok zadaný kľúč, je možné mať štruktúru usporiadanú, čo pomáha pri niektorých operáciách (na druhej strane môže spomaliť iné operácie).

Príklad: Majme danú množinu prvkov (napr. množinu čísel). Na tejto množine chceme efektívne vyhľadávať, vkladať a vyberať prvky. Aké sú možnosti jej implementácie?

1) Triviálne riešenie: **Pole** P: 1 ... n, - pristupujeme ku prvku cez index : P[i].

Zložitosť jednotlivých operácií:
member … O(n), insert … O(1), delete … O(n) - (nájdi, zmaž a posuň)
Výhoda: jednoduchá implementácia.
Nevýhoda: dopredu treba zadať veľkosť poľa, pričom počet položiek sa môže meniť $\Rightarrow$ plytvanie pamäťou, pretečenie...

2) **Vyvážený strom:** Zložitosť jednotlivých operácií … O(log n).
Náročnejšia implementácia, ale rýchlejšie operácie.

Aj jednotlivé prvky systému dát môžu byť štruktúrované - dátová štruktúra **záznam**. Napr. popis vrcholov grafu. Záznam: VRCH, jeho zložky: VRCH.POR (poradové číslo), VRCH.DEG (stupeň), VRCH.HODN (ďalšia číselná hodnota).
Často sa používajú dátové štruktúry, ktorých prvky sú štruktúrované vo forme záznamu. Každý záznam v štruktúre je väčšinou jednoznačne určený jednou z položiek záznamu – tzv. kľúčom. V našom príklade by to bolo poradové číslo vrchola.

Dátová štruktúra je definovaná (určená) vnútornou organizáciou dát + implementáciou jednotlivých operácií.

U - univerzálna množina – množina prvkov, ktoré sa môžu vyskytovať ako prvky dátovej štruktúry,
S – množina prvkov dátovej štruktúry,
$x \in U$.
Funkcie, resp. operácie ktoré sú najčastejšie požadované od dátovej štruktúry:

1. MEMBER (x,S) – určiť, či $x \in S$, ak áno, tak určiť miesto pamäti (resp. smerník), kde je x uložený.
2. INSERT (x,S) – vlož x do S
3. DELETE (x,S) – zmaž x z S

Ak je univerzálna možina usporiadaná, používajú sa aj ďalšie operácie:

4. MIN (S) – vráti najmenší prvok z S
5. MAX(S) – vráti najväčší prvok z S
6. SUCCESSOR (S, x) – nasledovník prevku x v množine S.
7. PREDECESSOR (S, x) – predchodca prevku x v množine S.
8. SPLIT (x,S) rozloží množinu S na 2 disjunktné množiny S_1,S_2 tak, že $S_1 = \{y; y \in S, y \leq x\}$, $S_2 = \{y; y \in S, y > x\}$
9. CONCATENATE (S_1,S_2) – Ak pre $\forall x` \in S_1$, $\forall x`` \in S_2$ platí $x` < x``$ vytvorí sa usporiadaná množina ($S_1 \cup S_2$).

Vnútorná oranizácia dát, najčastejšie používaná pri implementácii dátových štruktúr:

- Pole
- Viazaný zoznam (jednosmerný, obojsmerný)
- Binárny prehľadávací strom, špeciálne vyvážený strom

<u>Pole(Array)</u>

Pole predstavuje najjednoduchšiu dátovú štruktúru, ktorej prvky sú usporiadané v pamäti za sebou. K jednotlivým prvkom poľa P sa pristupuje pomocou indexu, t.j. P[i]. Polia môžeme mať utriedené aj neutriedené. Pre neutriedené pole máme operácie:

```
ARRAY_MEMBER(x, P)
N=PocetPrvkovPola();
for(i=0;i<N;i++)
   if(x==P[i])
      return i;
return N;
```

Zložitosť O(n)

```
ARRAY_INSERT (x, P)
N=PocetPrvkovPola();
RealokujViacPamäte();
P[N]=x;
N++;
```

Zložitosť O(1)

```
ARRAY_DELETE (L, x)
pos=MEMBER(x, P);
N=PocetPrvkovPola();
for(i=pos;i<N-1;i++)
   P[i]=P[i+1];
RealokujMenejPamate();
```

Zložitosť O(n)

Z popísaných algoritmov je zrejmé, že pri pridávaní a odoberaní prvku sa musí meniť veľkosť pamäte určenej pre pole, čo tiež sposobuje časovú záťaž a možné konfliktz v pamäti.

Zoznam (List)

Veľmi často treba uchovávať v pamäti počítača postupnosť objektov (jednotlivých dát, resp. záznamov). Jednou z dátových štruktúr vhodných na reprezentáciu postupnosti je dátová štruktúra **zoznam**. Základné operácie:

- určiť, či sa daný záznam v zozname nachádza
- pridať daný záznam do zoznamu
- vymazať daný záznam zo zoznamu

Rýchlosť daných operácií závisí od spôsobu implementácie zoznamu, t.j. od spôsobu jeho uloženia v počítači. Dva základné spôsoby:

1. Kompaktný zoznam ≈ zoznam ako pole.

Nevýhoda: dopredu treba zadať veľkosť poľa, pričom počet položiek sa môže meniť ⇒ plytvanie pamäťou, pretečenie...

2. Viazaný zoznam – dynamická dátová štruktúra

- pracujeme so smerníkmi, (smerník – reprezentuje adresu premennej)
- počas behu programu alokujeme (new) a uvoľňujeme (delete) pamäť, ale vždy iba pre aktuálny prvok ⇒ počet prvkov sa môže dynamicky meniť.

Existuje viacej druhov viazaných zoznamov:

- jednosmerný, obojsmerný
- utriedený, neutriedený
- cyklický

Obojsmerný viazaný zoznam, neutriedený

Jednotlivé záznamy (objekty) sú typu Prvok a obsahujú:

- *kľúč* (= meno, identifikácia prvku – nech je to celé číslo): prvok.key;
- *smerník na predchodcu*: prvok.prev (typ Prvok *);
- *smerník na nasledovníka*: prvok.next (typ Prvok *);
- *dáta*

Ak *x.prev* == NIL, element *x* nemá predchodcu, je teda prvým objektom v zozname (**head)**.

Ak *x.next* = NIL, *x* nemá nasledovníka, je posledným prvkom zoznamu (**tail**).
Atribút *head*[*L*] obsahuje smerník na prvý objekt v zozname, t.j. na head. Ak *head*[*L*] = NIL, zoznam je prázdny.

Operácie a ich zložitosť: (uvádzame zložitosť najhoršieho prípadu)
LIST_MEMBER (*L*, *k*) – lineárnym vyhľadávaním nájde prvý objekt s kľúčom *k* v zozname *L* a vráti smerník na ten objekt. Ak sa taký objekt v zozname nenachádza, vráti NIL..... θ(n).
LIST_INSERT (*L*, *x*) – procedúra pripojí objekt *x* (ktorého položka *key* už je nastavená) na začiatok resp. koniec zoznamu.... θ(1).
LIST_DELETE (*L*, *x*) – odstráni prvok *x* zo zoznamu *L*..... θ(1). Vymazanie prvoku zo zoznamu, ak máme daný kľúč *k*, vyžaduje najprv volanie procecúry LIST_MEMBER (*L*, *k*), ktorá nám vráti smerník na prvok s kľúčom *k*. ..... θ(n).

```
LIST_MEMBER(L, k)
x = head[L];
while (x != NIL && x->key != k)
   do x = x->next;
return x;
```

```
LIST_INSERT(L, x)
x->next = head[L];
if (head[L] != NIL)
   head[L]->prev = x;
head[L] = x;
prev[x] = NIL;
```

```
LIST_DELETE(L, x)
if (x->prev != NIL)
   [x->prev]->next = x->next;
else head[L] = x->next;
if (x->next != NIL)
   then [x->next]-> prev = x-> prev;
delete x;
```

```
SENT_LIST_MEMBER(L, k)
x = nil[L]->next
while ((x != nil[L]) && (x->key !=k))
      do x = x->next;
return x;
```

```
SENT_LIST_INSERT(L, x)
x->next = nil[L]->next;
[nil[L]->next]->prev = x;
nil[L]->next = x;
x->prev = nil[L];
```

```
SENT_LIST_DELETE(L, x)
[x->prev]->next = x->next;
[x->next]->prev = x->prev;
```

Sentinel – prvok, ktorý nenesie dáta (prázdny objekt), určuje začiatok, resp. koniec zoznamu, slúži ako smerník na zoznam. K zoznamu *L* uvažujme objekt *nil*[*L*], ktorý bude reprezentovať NIL a bude mať rovnaké položky ako ostatné prvky zoznamu. Položky s obsahom NIL budú teraz obsahovať smerník na *nil*[*L*]. Dostaneme tak cyklický obojsmerný zoznam, pričom sentinel *nil*[*L*] je umiestnený medzi head a tail. Položka *nil*[*L*]->*next* obsahuje smerník na head a *nil*[*L*]->*prev* smerník na tail. Atribút *head*[*L*] je už zbytočný. Prázdny zoznam pozostáva iba zo sentinel-u, pričom *nil*[*L*]->*next* aj *nil*[*L*]->*prev* sú nastavené na *nil*[*L*].
Sentinel málokedy redukuje asymptotickú zložitosť operácií na dátových štruktúrach, ale môže znížiť zložitosť o konštantný faktor. Jedným z hlavných prínosom použitia sentinelu je väčšia prehľadnosť kódov. Avšak nie vždy je vhodné použitie sentinelu, napr. keď pracujeme s veľkým počtom malých zoznamov, použitie sentinelov môže znamenať značnú pamäťovú náročnosť.

Binárny prehľadávací strom (BPS)

Ide o dátovú štruktúru, ktorá umožňuje vykonávať väčšinu operácií v čase úmernom výške stromu.
Prvok binárneho stromu obsahuje:

- kľúč, napr. celé číslo, musí sa dať porovnávať): prvok.key
- smerník na ľavého syna : prvok.left ... typ Prvok * ... smerník na Prvok
- smerník na pravého syna : prvok.right
- smerník na rodiča (nemusí) : prvok.parent
- dáta

Ak prvok nemá syna, daný smerník má hodnotu NIL. Koreň je jediný prvok v strome, ktorého smerník na rodiča má hodnotu NIL. Na koreň stromu ukazuje atribút *root*[T]. Ak *root*[T] = NIL, potom je strom prázdny.

Binárny prehľadávací strom je binárny strom, pre ktorý platí: nech x je ľubovoľný vrchol BPS a y je ľubovoľný vrchol z ľavého (pravého) podstromu vrchola x. Potom platí: y.key < x.key (y.key ≥ x.key).

Operácie:

```
BPS_MEMBER (x, k)
if ((x == NIL) || (k == x->key))
    return x;
if (k < x->key)
    return BPS_MEMBER(x->left, k);
else
    return BPS_MEMBER(x->right, k);
```

```
BPS_MEMBER_2 (x, k)
while ((x != NIL) && (k != x->key))
{  if (k < x->key)
        then x = x->left;
   else x = x->right;
}
return x;
```

```
BPS_SUCCESSOR (x)
if (x->right != NIL)
    return BPS_MINIMUM(x->right);
y = x->parent;
while ((y != NIL) && (x == y->right))
{    x = y;
     y = y->parent;
}
return y;
```

```
BPS_MINIMUM (x)
while (x->left != NIL)
   x = x->left;
return x;

BPS_MAXIMUM (x)
while (x->right != NIL)
   x = x->right;
return x;
```

```
BPS_INSERT (T, z)
y = NIL;
x = root[T];
while (x != NIL)
{    y = x;
     if (z->key < x->key)
          then x = x->left;
     else x = x->right;
}
z->parent = y;
if (y = NIL)
     root[T] = z;
else if (z->key < y->key)
     then y->left = z;
else y->right = z;
```

```
BPS_INSERT_2 (x, y)
{
  // vloží y do podstromu s koreňom x
 if (y→key < x→key)
     if (x→left == NIL)  x→left = y;
     else BPS_INSERT_2(x→left, y);
 if (y→key > x→key)
     if (x→right == NIL)  x→right = y;
     else BPS_INSERT_2 (x→right, y);
}
```

```
BPS_DELETE (T, z)
if  ((z->left == NIL) || (z->right == NIL)
    then y = z;
else y = BPS_SUCCESSOR(z);
if (y->left != NIL)
    x = y->left;
else x = y->right;
if (x != NIL)
    then x->parent = y->parent;
if (y->parent == NIL)
    root[T] = x;
else if (y == [y->parent]->right)
    [y->parent]->left = x;
else [y->parent]->right = x
if (y != z)
{       z->key = y->key;
        Ak má y aj iné položky, skopíruj ich
}
return y;
```

Popis jednotlivych algoritmov (operacii):

MEMBER (x, k) – prechadza sa strom od hora dole s porovnavanim klucov, az kym sa na sany kluc nenaraz. Vráti smerník na prvok s kľúčom k, ak sa nachádza v podstrome s vrcholom x, inak vráti NULL.

INSERT (x) – prechadza sa strom od korena s porovnavanim klucov, az sa dojde k vrcholu bez daneho potomka, ta to miesto sa potom vlozi x ako list stromu.

MIN (MAX.):
postupujeme od koreňa vždy do ľavého (pravého) syna. Prvok, ktorý nemá ľavého (pravého) syna je minimálny (maximálny).

SUCCESSOR (y): - ak y ma praveho syna, potom nasledovnik y podla usporiadana klucov je prvok s minimalnym klucom v pravom podstrome y. Ak y nemam praveho syna, potom sa postupuje v prehladavan hore stromom (inverzny prechod ako pri INSERT) az kymsa v tomto prehladavani predchadzajuci vrchol nenachadza nalavo od aktualneho vrcholu. Podobne aj pre vyhladavanie predchodcu prvku (PREDECESSOR).

DELETE (y) - y je smerník na prvok, ktorý treba zmazať. Ak máme kľúč, použijeme MEMBER.

1. y je list: y→parent→right = NULL
2. y má jedného nasledovníka v: ak y je koreň tak ho vynecháme a novým koreňom bude vrchol v. Ak y je pravým nasledovníkom vrchola u, tak y→parent→right = v
3. y má dvoch synov:
   - nájdeme maximum v ľavom podstrome vrchola y (MAX)
   - vymeníme y a MAX
   - zmažeme y (teraz nemá pravého syna) podľa 1. alebo 2.

Def 1: *Výška stromu* je dľžka najdlhšej cesty od koreňa k listu (+ 1).
Def 2: *Výška stromu* je definovaná ako maximálna úroveň ľubovoľného vrchola stromu. Koreň stromu je na prvej úrovni. Ozn. h(r) výšku stromu s koreňom r.
Pozn: Výška stromu s jedným vrcholom je 1.

Veta: Operácie MEMBER, INSERT, DELETE, MIN, MAX na BPS s výškou h sa vykonajú v čase O(h).
Dôkaz: Pri každej operácií prejdeme cestu z nejakého vrchola (najčastejšie koreňa) smerom dolu. V každom vrchole tejto cesty vykonáme operácie, ktorých trvanie je ohraničené konštantou. Dľžka cesty je max. h.

Vzťah medzi výškou BS (binarneho stromu) a počtom jeho vrcholov (t.j. počtom prvkov množiny, ktorú reprezentuje): Max. počet vrcholov v BS výšky h je $2^h - 1$, takže pre n-prvkovú množinu treba BS s výškou $\lceil \log(n+1) \rceil$; ($2^h - 1 \geq n \Rightarrow h \geq \log(n+1)$). Výška BPS v ideálnom prípade teda bude h = log n, v najhoršom prípade h = n.
Časová zložitosť uvažovaných operácií v najhoršom prípade bude O(n) (pri deformovanom strome). Možnosti zlepšenia:
a) použiť algoritmy na INSERT a DELETE, ktoré zaručia, že BPS bude dobre vyvážený, t.j. bude mať výšku O(log n)
b) pri dostatočne náhodnej voľbe čísel, ktoré vkladáme a vyberáme, sa výška BPS pohybuje okolo 1,4logn, takže priemerná dĺžka operácií nie je omnoho väčšia ako toretické optimum (randomizácia)
c) použiť iné stromové štruktúry, ktoré majú výšku O(logn) za všetkých okolností (2-3 stromy, červeno-čierne stromy,...)

Zložitosť operácií na jednotlivých implementáciách dátových štruktúr

Prvok x z univerzálnej množiny budeme reprezentovať kľúčom $k \in Z$. Prvky teda budeme vedieť usporiadať podľa kľúča.

	Zoznam	*Utriedený zoznam*	*BPS*
MEMBER	O(n)	O(n)	O(n)
INSERT	O(1)	O(n)	O(n)
DELETE	O(1)	O(n)	O(n)
MIN, MAX	O(n)	O(1)	O(n)
SPLIT	O(n)	O(n)	
CONCATENATE	O(1)	O(1)	

Poznámky:
1. Pri operácii DELETE predpokladáme, že máme smerník na prvok, ktorý chceme zmazať (napr. ako výsledok operácie MEMBER).
2. SPLIT a CONCATENATE sa pomocou spájaného zoznamu urobia triviálne:
   - SPLIT v zozname: pôjdeme po prvkoch a budeme ich rozďelovať do dvoch zoznamov podľa toho, či sú väčšie alebo menšie ako prvok, s ktorým porovnávame
   - CONCATENATE v zozname: druhý zoznam napojíme na koniec prvého

- SPLIT v utriedenom zozname: pôjdeme po prvkoch od najmenšieho až kým nenarazíme na prvý väčší, pred ním zoznam rozdelíme
- CONCATENATE v utriedenom zozname: napojíme zoznam s väčšími prvkami za zoznam s menšími

3. SPLIT a CONCATENATE pomocou BPS nie je triviálne, budeme sa tomu venovať neskôr.

Prehľadávanie BPS

Úloha: Prejsť cez všetky vrcholy BPS.

```
IN_ORDER (v)
{
1) if (v→left != NIL) IN_ORDER (v→left);
2) Spracuj (v);
3) if (v→right != NIL) IN_ORDER (v→right);
}

IN_ORDER (T)
{
 if (v = root[T])
      IN_ORDER(v);
}
```

Procedúra IN_ORDER spracuje vrcholy v utriedenom poradí.
Pozn. Existujú ďalšie spôsoby prehľadávania BPS:
- PRE_ORDER: postupnosť krokov je 2), 1), 3)
- POST_ORDER: postupnosť krokov je 1), 3), 2)

Zložitosť: O(n).

1. 3. AVL-strom (Adelson-Velskii a Landis)

Def: *AVL-strom* je binárny prehľadávací strom taký, že pre každý jeho vrchol *v* platí:
- buď *v* je list
- alebo *v* má jedného syna, ktorý je list
- alebo *v* má dvoch synov. Ak ľavý podstrom vrchola *v* má výšku h_1 a pravý h_2, potom platí:
  $|h_1 - h_2| \leq 1$.

Teda AVL-strom je BPS, v ktorom výšky dvoch podstromov každého vrchola sa líšia najviac o 1. Preto algoritmy, ktore nemenia strom (MEMBER, MAX, SUCCESSOR, ...) su rovnake ako v pripade BPS.

Veta: AVL-strom s n vrcholmi má výšku najviac **2logn**.

Dôkaz: Nech AVL-strom s n vrcholmi má výšku h. Uvažujme AVL-strom s výškou h (ozn. T_h), ktorý má minimálny počet vrcholov, ozn. n(h). Potom zjavne $n \geq n(h)$. Platí $n(1) = 1$, $n(2) = 2$, $n(3) = 4$, ... Nech ľavý podstrom stromu T_h má výšku $h - 1$ a pravý podstrom má výšku $h - 1$ alebo $h - 2$. Keďže chceme, aby T_h malo minimálny počet vrcholov, jeho ľavý podstrom bude T_{h-1} a pravý T_{h-2}. Teda $n(h) = 1 + n(h - 1) + n(h$

– 2) > 2n(h – 2). Odtiaľ dostávame: $n \geq n(h) > 2n(h-2) > 2^2 n(h-4) > ... > 2^{h/2} \Rightarrow n > 2^{h/2} \Rightarrow$
$h < 2\log n$.

Pozn. Dá sa dokázať aj presnejší odhad, a síce:
$\log(n+1) \leq h \leq 1{,}4404 \log(n+2) - 0{,}3277$; (Dôkaz: cez Fibonacciho stromy, T_h je Figonacciho strom stupňa h + 1, $n(h) = F_{h+2} - 1 \geq \phi^{h+2} / \sqrt{5} - 2$; $\phi = \frac{1}{2}(1+\sqrt{5})$; $F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2}$).

Def: Pre každý vrchol *v* BPS položme
- $p(v)$ ... výška pravého podstromu vrchola *v* (ak ∃)
- $l(v)$ ... výška ľavého podstromu vrchola *v* (ak ∃)
- $p(v) = 0$ resp. $l(v) = 0$, ak *v* nemá pravého resp. ľavého syna.

Označme $b(v) = l(v) - p(v)$. Vrchol *v* nazveme *vyváženým*, ak $b(v) \in \{-1, 0, 1\}$.

AVL-strom je teda BPS, ktorého každý vrchol je vyvážený.

Operácia vkladania vrcholu do AVL-stromu

Pridávaním vrchola do AVL-stromu pomocou algoritmu INSERT pre BPS môžu vzniknúť nevyvážené vrcholy. Pre každý vrchol sa hodnota $b(v)$ môže zväčšiť alebo zmenšiť o 1, prípadne zostať nezmenená. Nevyváženosť vo vrchole *v* nastane, ak sa $b(v)$ zmení z 1 na 2 resp. z -1 na -2.
Budeme sa venovať prípadu $b(v) = 2$, druhý prípad je symetrický.

Na obrázku A a B sú ukázané dva možné spôsoby vzniku nevyváženého vrchola pri operácii INSERT. V oboch prípadoch mali pôvodné stromy (označené ako T1, T2, T3) rovnakú výšku (alebo boli všetky prázdne) a po pridaní nového vrchola sa o 1 zväčšila výška stromu T1 resp. T2, čím sa porušila vyváženosť vrchola *x*. Zmenou tvaru stromu je možné vyváženosť vrchola *x* obnoviť bez toho, aby sme porušili vyváženosť iných vrcholov. Vznikne tak BPS, v ktorom je *x* aj všetci jeho pôvodní nasledovníci vyvážení. Dôležité je, že výška stromu s vrcholom *x*, ktorá sa po pridaní nového vrchola zvýšila o 1, po následnej modifikácii opäť klesla na pôvodnú hodnotu. Takže vyvážením *x* sa automaticky obnoví vyváženosť všetkých jeho predchodcov, ak u nich došlo k narušeniu vyváženosti. Stačí teda vyvažovať ten z nevyvážených vrcholov, ktorý je na ceste z koreňa do nového vrchola najnižšie.

Tvrdenie: Vrchol *x* treba vyvažovať práve vtedy, keď platia nasledujúce dve podmienky:
1. $b(x) = 1$ a nový vrchol je pridaný doľava od *x*,
   $b(x) = -1$ a nový vrchol je pridaný doprava od *x*,
2. pre ∀ *y*, ktorý leží na ceste od *x* ku pridávanému vrcholu platí: $b(y) = 0$.

Dôkaz: Nech *x* je posledný nevyvážený vrchol na ceste z koreňa do nového vrchola a *y* nech je nejaký vrchol na ceste z *x* do nového vrchola. Označme $b(v)$ hodnotu *b* vrchola *v* pred pridaním a $b`(v)$ hodnotu *b* vrchola *v* po pridaní nového vrchola.
$\Rightarrow b(x) \in \{1, -1\}; b`(x) \in \{2, -2\}$, (*).
Chceme ukázať, že $b(y) = 0$. Sporom.
Nech $b(y) = \pm 1$. Potom

1) $b`(y) = \pm 2$ $(1 \rightarrow 2;\ -1 \rightarrow -2)$ nemôže nastať, lebo x je posledný nevyvážený vrchol
2) $b`(y) = 0$ alebo $b`(y) = \pm 1$ ... výška stromu s koreňom x sa nezmenila $\Rightarrow b`(x) = b(x)$, spor s (*)

$\Leftarrow$ Nech platia predpoklady 1) a 2) tvrdenia. Potom $b`(x) = \pm 2$, t.j. $x$ je nevyvážený. Pre ľubovoľný vrchol $y$ na ceste z $x$ do nového vrchola platí, že obidva podstromy vrchola $y$ sú rovnakej výšky ( keďže $b(y) = 0$). Po pridaní nového vrchola sa zmení výška jedného z podstromov, teda $b`(y) = \pm 1 \Rightarrow$ vrchol y netreba vyvažovať $\Rightarrow$ vrchol x je najnižší nevyvážený vrchol.

Teraz uvedieme algoritmus na vkladanie uzla do AVL-stromu.

1) Otestujeme, či strom nie je prázdny. Ak áno, pridáme vrchol, ak nie tak ideme ďalej
2) Vložíme nový vrchol podobne ako pre INSERT v BPS, pritom hľadáme vrchol x ako najnižší potenciálne nevyvážený vrchol (ak $b(u) \neq 0 \Rightarrow x = u$).
3) Nový vrchol je už vložený a je určený aj kandidát na vyvažovanie, vrchol x. Zmeníme hodnoty $b(v)$ pre každý vrchol v na ceste z x do nového vrchola. Ostatné hodnoty $b(v)$ netreba meniť, lebo po vyvážení x budú nadobúdať pôvodnú hodnotu.
4) Ak $b(x) = 2$ vyvážime vrchol x podľa A alebo B.

Algoritmus:

v – nový vrchol, x – kandidát na vyvažovanie, koreň – koreň stromu, (x a *koreň* sú globálne premenné)

```
AVL_INSERT (v)
{       x = koreň;
(1)     if (koreň = = NIL)  { koreň = v;
                              b(v) = 0;}
        else
(2)     {       INSERT (koreň, v);
(3)             MODIFY (x, v);
(4)             If (b(x) = = 2)
                        If (b(x→ľavý) = = 1) vyvažuj podľa A
                        If (b(x→ľavý) = = -1) vyvažuj podľa B
                If (b(x) = = -2)
                        Symetricky ku (b(x) = = 2)
        }
}

INSERT (u, v)
{       If (v→kľúč < u→kľúč)
                If (u→ľavý = = NIL) { u→ľavý = v;
                                      if (b(u) ≠ 0) x = u;  }
                else
                {       if (b(u) ≠ 0) x = u;
                        INSERT (u→ľavý, v);
                }
        If (v→kľúč > u→kľúč)
```

```
                    If (u→pravý = = NIL) { u→pravý = v;
                                           if (b(u) ≠ 0) x = u;  }
                    else
                    {       if (b(u) ≠ 0) x = u;
                            INSERT (u→pravý, v);
                    }
}

MODIFY (u, v)
{       If (v→kľúč < u→kľúč)
                If (u→ľavý ≠ NULL)
                {       b(u) = b(u) + 1;
                        MODIFY (u→ľavý, v);
                }
        If (v→kľúč > u→kľúč)
                If (u→pravý ≠ NULL)
                {       b(u) = b(u) – 1;
                        MODIFY (u→pravý, v);
                }
}
```

Zložitosť: O(log n)
(1) O(1) operácií
(2) O(log n) – 1-krát dolu po ceste
(3) O(log n) – 1-krát dolu po časti cesty
(4) O(1) – lebo vyvažujeme len jedenkrát

Operácia vynechania vrchola v AVL-strome

Operácia vynechania vrchola bude rozšírením DELETE pre BPS.

AVL_DELETE (*v*)
1) *v* má najviac 1 syna:
- vrchol *v* vynecháme ako pri DELETE pre BPS
- *x* = *v*→rodič;
- BALANCE (*x*, *v*);
- delete *v*;

2) *v* má dvoch synov:
- nájdeme maximum v ľavom podstrome vrchola *v* (*MAX*)
- vymeníme *v* a *MAX*
- postupujeme podľa 1)

```
BALANCE (x, v)
{       If (x ≠ NULL)
(a)  {  if (v→kľúč < x→kľúč) and (b(x) ≥ 0)          // do x sme prešli zľava
                {       b(x) = b(x) – 1;
                        If (b(x) = = 0) BALANCE (x→rodič, v);      // zmenila sa
    výška h(x)
                }
```

(b) if ($v \rightarrow kľúč > x \rightarrow kľúč$) and ($b(x) \leq 0$)
{ $b(x) = b(x) + 1$;
If ($b(x) = = 0$) BALANCE ($x \rightarrow rodič$, v);
}
(c) if ($v \rightarrow kľúč > x \rightarrow kľúč$) and ($b(x) = = 1$) situácia **A**, **B**, alebo **C**
(d) if ($v \rightarrow kľúč < x \rightarrow kľúč$) and ($b(x) = = -1$) situácia symetr. ku **A**, **B**, **C**
(e) Ak sme v (c) alebo (d) vyvažovali podľa **A** resp. **B** a v $x \rightarrow rodič$ vznikla nevyváženosť $\Rightarrow$
BALANCE ($x \rightarrow rodič$, v);
}
}

Pozn.:
(a) Ak sme do x prišli zľava a $b(x) \geq 0$, vrchol x netreba vyvažovať,
- ak $b`(x) = 0$, zmenila sa výška stromu s vrcholom x, treba vyvažovať $x \rightarrow rodič$,
- ak $b`(x) = -1$, výška stromu ostala nezmenená, netreba vyvažovať $x \rightarrow rodič$

(b) symetricky ku (a)
Ak sme vyvažovali podľa **A** resp. **B**, po vyvážení sa zmenila výška stromu s pôvodným vrcholom x, a teda vo vrchole $x \rightarrow rodič$ mohla vzniknúť nevyváženosť.

Zložitosť: O(log n) – manipulujeme iba s vrcholmi na ceste z koreňa do zmazávaného vrchola, resp. náhradníka.
1. Vymazanie vrchola – O(log n) ... kvôli hľadaniu náhradníka
2. BALANCE – nanajvyš O(log n) vyvažovaní, ktoré majú konštantný počet krokov.

Operácie MEMBER, MIN a MAX v AVL-strome nemenia tvar stromu, teda netreba vyvažovať. Môžeme použiť algoritmy ako pri BPS.
Zložitosť: O(log n) – zídeme od koreňa k listu.

Veta: Operácie INSERT, DELETE, MEMBER, MIN, MAX na AVL-stromoch sa dajú vykonať v čase O(log n).

Operácia spájania

Majme dané dva AVL-stromy S_1 a S_2, pričom pre: $\forall x` \in S_1, \forall x`` \in S_2$ platí: $x` < x``$.
Operácia CONCATENATE (S_1,S_2) nám vytvorí AVL-strom $S_1 \cup S_2$.
Nech $h(S_1) \geq h(S_2)$.
1. Označme j = vrchol s najmenším kľúčom v strome S_2 (j – juncture node).
2. AVL_DELETE (j) - vymažeme vrchol j zo stromu S_2 – vznikne strom $S_2`$.
3. V strome S_1 ideme od koreňa stále napravo, až kým nenájdeme taký vrchol p, pre ktorý platí: $h(p) - h(S_2`) = 0$ alebo 1, kde $h(p)$ označuje výšku stromu s koreňom p. Uvedomme si, že výška stromu, pri prechode na pravého syna, sa mení nasledovne:
   - ak $b(x) = 0 \vee -1 \Rightarrow h(x \rightarrow pravý) = h(x) - 1$,
   - ak $b(x) = 1 \Rightarrow h(x \rightarrow pravý) = h(x) - 2$.
4. Vrchol p nahradíme vrcholom j a zmodifikujeme podľa obrázku. POZOR! Výška pôvodného stromu s vrcholom $r = p \rightarrow rodič$ sa o 1 zväčšila.

5. Ak je treba, vyvážime *r* ako pri algoritme vkladania vrchola (keďže pri vyvažovaní sa nám v tomto prípade môže meniť výška vyvažovaného vrchola, treba takto postupovať až ku koreňu).

Pre $h(S_1) \leq h(S_2)$ postupujeme analogicky, len symetricky.

Pozn: Pod pojmom výška vrchola *v* rozumieme výšku stromu s koreňom *v*, ozn. h(*v*).

Zložitosť: O(log n)

1.+2.: Vymazanie vrchola s najmenším kľúčom – $O(\log n_2) \leq O(\log n)$

3. Nájdenie vrchola *p*: $O(\log k) \leq O(\log n)$, kde k je rozdiel výšok stromov S_1, S_2`.
4. Spojenie: O(1)
5. Vyváženie: O(log k)

Operácia rozdeľovania

Operácia SPLIT (*S*, *x*), $x \in S$, vytvorí AVL-stromy S_1,S_2, pre ktoré platí:
$\forall x` \in S_1 : x` \leq x$, $\forall x`` \in S_2 : x`` > x$.
Pri operácii SPLIT budeme využívať nasledujúcu, mierne modifikovanú, verziu operácie CONCATENATE:
CONCATENATE (S_1``, *j* ,$S_2$``)

- Ak $h(S_1``) \geq h(S_2``)$, význam parametrov je takýto: $S_1`` = S_1$, *j* = juncture node - vrchol s najmenším kľúčom v strome $S_2$, $S_2`` = S_2`$.
- Ak $h(S_1``) \leq h(S_2``)$, význam parametrov je takýto: $S_1`` = S_1`$, *j* = juncture node - vrchol s najväčším kľúčom v strome $S_1$, $S_2`` = S_2$.

Teda vstupom sú stromy, ktoré vzniknú v pôvodnej funkcii po kroku 2. Ďalej operácia pokračuje ako pôvodná, krokmi 3 – 5.

```
AVL_SPLIT (S, x)
{
        α = ľavý podstrom x;
        AVL_INSERT (α, x);
        β = pravý podstrom x;
        r = x→parent; p = x;
        While (r != NIL) do
        {
                If (p→key < r→key)
                        β = CONCATENATE (β, r, pravý podstrom r);
                If (p→ key > r→ key)
                        α = CONCATENATE (ľavý podstrom r, r, α);
                p = r;
                r = r→parent;
        }
}
```

Zložitosť: Dá sa netriviálne dokázať, že je to O(log n).