Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Vyhľadávacie stromy a ich vizualizácia

(bakalárska práca)

Kubo Kováč

Vedúci: Mgr. Janka Katreniaková

Bratislava, 2007

WUNDER-
BAUM
®
erfrischt die Luft
in jedem Raum!

Prehlásenie

Copy from one, it's plagiarism;
copy from two, it's research.
Wilson Mizner

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím citovaných zdrojov. ....

Venovanie

Túto prácu venujem Korešpondenčnému semináru z programovania a podenkám.

Abstrakt

People who like this sort of thing
will find this the sort of thing they like.
Abraham Lincoln, in a book review

V tejto práci sa zaoberáme vizualizáciou stromovitých dátových štruktúr, presnejšie vyhľadávacích stromov. V prvej časti uvádzame prehľad binárnych vyhľadávacích stromov, vyvážených stromov (AVL-strom, 2-3-strom, B-strom, červeno-čierny strom, AA-strom), znáhodnených metód (skip list, treap) a vyhľadávacích stromov s amortizovanou zložitosťou (GB-strom, splay strom). V druhej časti sa venujeme vizualizácií týchto dátových štruktúr – popisujeme vytvorený program a jeho implementáciu.

Výsledkom práce je prehľad vyhľadávacích stromov a Java applet priložený na CD, tiež prístupný cez internet na adrese

`http://people.ksp.sk/~kuko/bak/`

Kľúčové slová: dátové štruktúry, vizualizácia, vyhľadávacie stromy, vyvážené stromy.

Obsah

Abstrakt **vii**

I Vyhľadávacie stromy 1

1 Úvod **3**
1.1 Základné pojmy . . . 3
Notácia . . . 3
Vyhľadávanie . . . 4
Klasifikácia vyhľadávacích algoritmov . . . 4
Problém slovníka . . . 5
1.2 Implementácia slovníka . . . 6
Jednoduchá reprezentácia . . . 6
Efektívne reprezentácie . . . 8
Slovníky v programovacích jazykoch . . . 9

2 Binárny vyhľadávací strom **11**
2.1 Strom – matematická štruktúra . . . 11
Usporiadaný strom . . . 11
Binárny strom . . . 12
Rozšírený binárny strom . . . 12
Vzťahy medzi definovanými veličinami . . . 12
Viac-cestné stromy . . . 13
2.2 Reprezentácia stromov . . . 13
Reprezentácia stromov v počítači . . . 13
Traverzovanie . . . 14
Prepletené stromy . . . 14
2.3 Binárny vyhľadávací strom. . . . 15
Definícia . . . 15
Vyhľadávanie . . . 16
Vkladanie . . . 16
Vymazávanie . . . 16
Časová zložitosť . . . 17
Výška náhodne generovaného binárneho vyhľadávacieho stromu . . . 18
2.4 Vyvažovanie . . . 19
Vyvažovacie techniky . . . 19
Rotácie . . . 20
Spájanie a delenie vrcholov . . . 20
Prestavanie stromu . . . 21

3 Vyvážené vyhľadávacie stromy 23
3.1 AVL-strom . . . 23
Definícia . . . 23
Výška AVL-stromu . . . 23
Vyvažovanie . . . 23
Spájanie . . . 24
3.2 2-3-strom . . . 24
Definícia . . . 24
Slovníkové operácie . . . 25
3.3 B-stromy . . . 26
Definícia . . . 26
Slovníkové operácie . . . 27
Časová zložitosť . . . 27
Variácie . . . 28
3.4 Červeno-čierny strom . . . 28
Definícia . . . 28
Slovníkové operácie . . . 28
3.5 AA-strom . . . 31

4 Znáhodnené vyhľadávacie stromy 33
4.1 Skip list . . . 33
Definícia . . . 33
Slovníkové operácie . . . 33
Analýza . . . 34
4.2 Treap . . . 34
Definícia . . . 34
Slovníkové operácie . . . 35

5 Amortizované vyhľadávacie stromy 37
5.1 GB-strom . . . 37
Definícia . . . 37
Vkladanie . . . 37
Vymazávanie . . . 37
Amortizovaná zložitosť . . . 38
5.2 Splay strom . . . 38
Samočinné vyvažovanie . . . 38
Operácia *splay* . . . 39
Slovníkové operácie pomocou *splay* . . . 39

II Vizualizácia algoritmov 43

6 Úvod 45
Cieľ . . . 45
Využitie . . . 45
Požiadavky . . . 45

7 Popis 47
Spustenie . . . 47
Hlavné časti . . . 47
Ovládanie . . . 48
Pokročilé ovládanie . . . 48
Špeciálne nastavenia . . . 48
Štatistiky . . . 49

Zobrazenie informácií . . . 49
Applet v akcií . . . 49

8 Implementácia 53
Základné triedy . . . 53
Kreslenie stromu . . . 53
Kreslenie vrcholov . . . 54

9 Záver 55
Existujúce riešenia . . . 55
Výsledok . . . 55
Budúca práca na projekte . . . 55

Časť I

Vyhľadávacie stromy

Kapitola 1

Úvod

1.1 Základné pojmy

We will occasionally use this arrow notation
unless there is danger of no confusion.
Ronald Graham

§ **Notácia.** Logaritmy lg, log a ln sú postupne logaritmy pri základe 2, 10 a e. Znakom Λ označujeme smerník null (nenájdený vrchol, koniec zoznamu, prázdny strom a pod.).

V programoch používame jazyk C s malými typografickými úpravami. Test rovnosti značíme $\equiv$, priradenie je $\leftarrow$. Ak i je celé číslo, potom $++i$ je výraz, ktorý zvýši i o 1 a vráti túto hodnotu. Výraz $i{+}{+}$ najskôr vráti hodnotu i, potom ju zvýši o 1. Obdobne $i{-}{-}$ a $-{-}i$ zníži hodnotu i o 1. Výraz $p\ ?\ e_1 : e_2$ vráti hodnotu e_1, ak p je pravda, inak e_2.

V programoch používame výlučne celočíselný typ – **int** a pole celých čísel. Pole $A[\mathtt{MAX}]$ sa začína od 0, teda indexy sú $0, 1, \ldots, \mathtt{MAX} - 1$. Znakom ∞ budeme označovať číslo väčšie ako ľubovoľné číslo inak použité v programe (tzv. strojové nekonečno, môže to byť napríklad najväčšia hodnota, ktorá sa zmestí do premennej typu **int**).

Každý príkaz končí bodkočiarkou. Ak $s_1, s_2, \ldots, s_n$ sú príkazy, potom aj $\{\ s_1\ s_2\ \cdots\ s_n\ \}$ je príkaz (presnejšie blok príkazov). Podmienený príkaz sa v C zapisuje **if** (p) s_1 **else** s_2 kde p je podmienka (ak p je celé číslo, tak $p \neq 0$ predstavuje pravdu a $p = 0$ nepravdu) a časť **else** je nepovinná. For-cyklus má tvar **for** $(e_1;\ e_2;\ e_3)$ s – na začiatku sa vykoná inicializácia e_1, následne, kým je e_2 pravdivé, sa vykonáva telo s a výraz e_3. Napríklad **for** $(f \leftarrow 1, i \leftarrow 2;\ i \leq n;\ ++i)$ $f \leftarrow i \cdot f$; najskôr f priradí 1 a i priradí 2; potom kým je $i \leq n$ prenásobuje f číslom i a zvyšuje i. Na konci teda v f bude faktoriál n. While-cyklus má syntax **while** (p) s – kým je podmienka p pravdivá, vykonáva sa s.

Procedúru p s argumentmi $x_1, \ldots, x_n$ typu $t_1, \ldots, t_n$ definujeme **void** $p(t_1\ x_1, \ldots, t_n\ x_n)$ $\{$ *telo procedúry* $\}$. Funkciu f s návratovou hodnotou typu T (a rovnakými argumentmi) definujeme $T\ f(t_1\ x_1, \ldots, t_n\ x_n)$ $\{$ *telo funkcie* $\}$. Príkaz **return** e; vráti hodnotu e ako výsledok funkcie a ihneď ju opustí.

Často tiež budeme využívať preprocesor jazyka C. Direktíva **#define** $id\ r$, kde id je identifikátor a r je reťazec spôsobí, že každý výskyt id v programe sa nahradí r. Existuje tiež parametrizovaná verzia makra. Napríklad **#define** $\mathtt{MIN}(a, b)\quad ((a) < (b)\ ?\ (a) : (b))$ nahradí každý reťazec tvaru $\mathtt{MIN}(a, b)$ výrazom vpravo. Napríklad $c \leftarrow \mathtt{MIN}(2, 1)$; sa nahradí $x \leftarrow ((2) < (1)\ ?\ (2) : (1))$;.

Zavedieme nasledujúce makrá:

Program 1.1 Makrá

```
#define MAX 10000
#define FOR(i,a,b) for (int i = (a); i < (b); ++i)
#define REP(i,n) FOR(i,0,n)
#define ROF(i,b,a) for (int i = (b) - 1; i >= (a); --i)
```

Teda FOR(i, a, b) s; vykoná príkaz s pre všetky (celé) i z rozsahu $i \in \langle a, b)$. FOR(i, a, b) s; vykoná s pre tie isté i ale v opačnom poradí (zhora nadol).

A safe but sometimes chilly way of recalling the past
is to force open a crammed drawer.
If you are searching for anything in particular you don't find it,
but something falls out at the back that is often more interesting.
James M. Barrie

§ **Vyhľadávanie.** Jeden z dôležitých problémov v informatike je organizácia a vyhľadávanie údajov. Tento problém je veľmi zaujímavý jednak z teoretického hľadiska – ako rýchlo vieme vyhľadávať? kde sú hranice? – jednak z praktického hľadiska: programy totiž trávia veľa času práve vyhľadávaním. Špeciálne hľadanie údajov podľa kľúča sa často používa napríklad v databázových systémoch, pri konštrukcií kompilátorov, či pri spracúvaní prirodzeného jazyka.

Majme množinu n údajov. Budeme predpokladať, že súčasťou každého záznamu je položka, ktorú budeme volať *kľúč*. Budeme predpokladať, že záznamy majú rôzne kľúče, teda vieme ich podľa kľúča jednoznačne identifikovať. Vyhľadávací algoritmus dostane na vstupe kľúč k a jeho úlohou je nájsť záznam, ktorého kľúč je k. Na konci hľadania môžu nastať dve možnosti: ak algoritmus našiel daný záznam, hovoríme že vyhľadávanie bolo *úspešné*; v opačnom prípade sa záznam s daným kľúčom v množine nenachádza (algoritmus vráti špeciálnu hodnotu Λ) – v takom prípade hovoríme, že hľadanie bolo *neúspešné*.

§ **Klasifikácia vyhľadávacích algoritmov.** Vyhľadávacie algoritmy môžeme rozdeľovať podľa niekoľkých kritérií. Podľa toho, či sú dáta uložené v operačnej pamäti alebo v externej pamäti, delíme vyhľadávanie na *interné* a *externé*.

Podľa toho, ako sa môže meniť súbor vyhľadávaných dát rozlišujeme *statické, semidynamické a dynamické vyhľadávanie*. Pri statickom vyhľadávaní sa množina dát nemení (alebo len veľmi zriedka), takže sa snažíme dáta predspracovať, aby sme v nich mohli efektívne vyhľadávať. Program na CD so slovensko-anglickým slovníkom môže byť príkladom externého statického vyhľadávania. Pri semidynamickom vyhľadávaní sa môže množina dát zväčšovať, t.j. môžeme pridávať nové prvky. Napríklad kompilátory a assemblery si počas prekladu potrebujú udržiavať tabuľku symbolov (napríklad pre každú premennú jej typ, pre funkciu typ jej argumentov a výstupnej hodnoty). V tomto prípade, ak kompilátor narazí na deklaráciu alebo definíciu novej premennej alebo funkcie, môže sa tabuľka rozrastať. Naopak, uložené symboly väčšinou nepotrebujeme odstraňovať (resp. v prípade lokálnych premenných rušíme celú dátovú štruktúru). Máme tu teda príklad interného semidynamického vyhľadávania. Najvšeobecnejšou metódou je dynamické vyhľadávanie, kde očakávame, že dáta sa budú meniť: budú pribúdať nové dáta a naopak, iné dáta budeme odstraňovať. Rôzne databázové systému môžu slúžiť ako príklad dynamického vyhľadávania.

Tretie možné delenie je na *digitálne* vyhľadávanie a vyhľadávanie *porovnávaním*, podľa toho, aký druh informácií nás smeruje k hľadanému kľúču. V prvom prípade sú to jednotlivé cifry čísla, jednotlivé zložky vektoru, alebo jednotlivé bity v počítačovej reprezentácií kľúča, v druhom prípade sa riadime porovnávaním kľúčov, t.j. informáciu, ktorú získavame je výsledok porovnania: $<$, $>$, alebo $=$.

The only place where success comes before work
is a dictionary.
Vidal Sassoon

§ **Problém slovníka.** V práci sa budeme venovať abstraktnému dátovému typu *slovník (dictionary)*. Dátová štruktúra D je slovník, ak podporuje nasledujúce operácie:

- $\mathit{insert}(D, x)$ do dátovej štruktúry D vloží záznam x (ak v nej ešte nie je záznam s daným kľúčom),
- $\mathit{find}(D, k)$ vráti (smerník na) záznam x s kľúčom k, alebo špeciálnu hodnotu Λ, ak sa taký záznam v D nenachádza,
- $\mathit{delete}(D, k)$ vymaže záznam s kľúčom k z D,

kde každý záznam x obsahuje položku $\mathit{key}(x)$ – kľúč jedinečný pre tento záznam.

Význam operácií *insert*, *find* a *delete* sa dá presne popísať pomocou axiomatickej sémantiky: Nech $\emptyset$ označuje prázdny slovník, definujeme:

- $\mathit{insert}(\mathit{insert}(D, x), x') = \mathit{insert}(D, x)$, ak $\mathit{key}(x) = \mathit{key}(x')$,
- $\mathit{insert}(\mathit{insert}(D, x), x') = \mathit{insert}(\mathit{insert}(D, x'), x)$, ak $\mathit{key}(x) \neq \mathit{key}(x')$,
- $\mathit{find}(\emptyset, k) = \Lambda$,
- $\mathit{find}(\mathit{insert}(D, x), k) = x$, ak $\mathit{key}(x) = k$,
- $\mathit{find}(\mathit{insert}(D, x), k) = \mathit{find}(D, k)$, ak $\mathit{key}(x) \neq k$,
- $\mathit{delete}(\emptyset, k) = \emptyset$,
- $\mathit{delete}(\mathit{insert}(D, x), k) = D$, ak $\mathit{key}(x) = k$
- $\mathit{delete}(\mathit{insert}(D, x), k) = \mathit{insert}(\mathit{delete}(D, x), x)$, ak $\mathit{key}(x) \neq k$.

Iný názov pre slovník je *asociatívne pole* (*associative array*, tiež *map*, a *lookup table*). Z programátorského hľadiska sa dá totiž na slovník pozerať ako na zovšeobecnenie poľa. Kým v obyčajnom poli A prislúcha číslu $i \in \mathcal{N}$ nejaký objekt $A[i]$, v slovníku D prislúcha nejakému *objektu* $k \in K$ nejaký objekt $D[k] = \mathit{find}(D, k)$.

V praxi väčšinou na množine kľúčov existuje úplne usporiadanie a tak by sme mohli požadovať, aby sme cez slovník mohli „prechádzať" (iterovať). Nech sú teda kľúče z úplne usporiadanej množiny $(K, <)$, definujeme nové operácie

- $\mathit{minimum}(D)$ vráti záznam s najmenším kľúčom v D,
- $\mathit{maximum}(D)$ vráti záznam s najväčším kľúčom v D,
- $\mathit{predecessor}(D, x)$ vráti predchádzajúci záznam, teda záznam, ktorého kľúč je najväčší menší ako $\mathit{key}(x)$,
- $\mathit{successor}(D, x)$ vráti nasledujúci záznam, teda záznam, ktorého kľúč je najmenší väčší ako $\mathit{key}(x)$.

Abstraktný dátový typ, ktorý okrem slovníkových operácií podporuje aj tieto, voláme *usporiadaný slovník*.

Navyše pre usporiadané slovníky môžeme potrebovať tieto, trochu drastickejšie, operácie:

- $\mathit{join}(D_1, D_2)$ vráti slovník D, ktorého záznamy pozostávajú zo všetkých záznamov D_1 a zo všetkých záznamov D_2; predpokladáme, že všetky záznamy z D_1 majú kľúč menší ako všetky záznamy z D_2; očakávame, že operácia zničí (uvoľní pamäť) D_1 a D_2,
- $\mathit{split}(D, x)$ rozdelí dátovú štruktúru na D_1, x a D_2 tak, že všetky záznamy v D_1 budú mať kľúč menší ako $\mathit{key}(x)$ a všetky záznamy v D_2 budú mať kľúč väčší ako $\mathit{key}(x)$; operácia teda vráti dvojicu (D_1, D_2), pričom zničí D.

Hoci hlavným cieľom vyhľadávania je nájsť dáta prislúchajúce danému kľúču, budeme všetky dáta okrem kľúča zanedbávať. Cieľom práce je ukázať spôsoby hľadania kľúča a pevne veríme, že ak nájdeme kľúč, podarí sa nám nájsť aj prislúchajúce dáta. V programoch preto budeme do slovníka vkladať a vymazávať iba kľúče.

1.2 Implementácia slovníka

§ **Jednoduchá reprezentácia.** Slovník je abstraktný dátový typ. V terminológií objektovo orientovaného programovania by sme mohli hovoriť o abstraktnej triede alebo o interface. Slovník definuje iba podporované operácie, je však viacero možných spôsobov, akými ho implementovať.

Asi najjednoduchšia implementácia je pomocou poľa: Majme pole n záznamov A dĺžky MAX. Nový prvok vložíme jednoducho (v čase $O(1)$) na koniec:

```
int A[MAX], n ← 0;
#define INSERT(k)   if (n ≡ MAX) OVERFLOW; else A[n++] ← k
```

v prípade, že $n = \texttt{MAX}$, nastane pretečenie. Jedna možnosť, ako to riešiť, je zvoliť MAX dostatočne veľké a uistiť sa, že pretečenie nenastane (v takom prípade stačí $A[n{+}{+}] \leftarrow k$). Druhou možnosťou je pri pretečení pole realokovať: alokujeme nové pole dvojnásobnej dĺžky ($\texttt{MAX} \leftarrow 2\texttt{MAX}$), staré pole tam prekopírujeme a uvoľníme jeho pamäť. V prípade, že počet prvkov klesne pod štvrtinu ($n < \texttt{MAX}/4$), môžeme pole realokovať späť na polovičnú veľkosť. Touto technikou dosiahneme amortizovanú časovú zložitosť $O(1)$ a pamäťovú zložitosť $O(n)$ (namiesto $O(\texttt{MAX})$) v najhoršom prípade. Vymazávanie je tiež jednoduché, daný prvok nahradíme prvkom na konci:

```
#define DELETE(k)   A[find(k)] ← A[−−n]
```

It's important to begin a search on a full stomach.
Henry Bromel

Na nešťastie, *insert* a *delete* sú v tomto prípade jediné efektívne operácie. Pri vyhľadávaní nám neostáva nič iné ako prezerať postupne všetky záznamy, kým daný kľúč nenájdeme, alebo nedôjdeme na koniec poľa (v tom prípade vieme, že záznam s daným kľúčom sa v poli nenachádza):

```
#define Λ   −1
int find(int k) { REP(i, n) if (A[i] ≡ k) return i; return Λ; }
```

Ostatné operácie sú pomalé podobne ako *find*: či už hľadáme minimum, maximum, predchodcu alebo následníka, neostáva nám nič iné, len sekvenčné vyhľadávanie.

Ak potrebujeme rýchlo vyhľadávať, lepšia možnosť je udržiavať pole utriedené:

Program 1.2 Implementácia slovníka pomocou utriedeného poľa

```
void init() { A[0] ← ∞; n ← 0; }
void insert(int x) { ++n; int i ← 0; while (A[i] < x) i++;
  ROF(j, n, i) A[j + 1] ← A[j]; A[i] ← x; }
void delete(int k) { int p ← find(k) FOR(j, p, n) A[j] ← A[j + 1]; −−n; }
```

Pri vkladaní musíme novému záznamu „spraviť miesto“ – nájdeme miesto, kam nový záznam vložiť (**while**-cyklus; ten vždy skončí, pretože sme použili zarážku ∞ na konci poľa) a počnúc daným miestom posunieme všetky záznamy o 1 doprava (FOR-cyklus). Pri vymazávaní posunieme všetky prvky počnúc $(i+1)$-vým doľava.

Pridávanie a vymazávanie je teraz zložitejšie. Za odmenu však vieme rýchlejšie výhľadávať. Myšlienkou binárneho vyhľadávania je, že začíname hľadať v intervale $\langle \ell, r\rangle$ a pozrieme sa na stredný prvok $A[m]$, kde $m = \lfloor (\ell + r)/2 \rfloor$. Ak $A[m] = k$, našli sme hľadaný záznam a sme hotoví. Ak $k < A[m]$, potom zjavne hľadaný kľúč patrí do prvej polovice, $\langle \ell, m)$, a ak $A[m] < k$, zjavne patrí do druhej polovice, $(m, r\rangle$, intervalu $\langle \ell, r\rangle$. Takto sa nám každým krokom zmenší hľadaný interval na polovicu a preto stačí $O(\log n)$ takých krokov.

Program 1.3 Binárne vyhľadávanie

```
int find(int k) {
   int ℓ ← 0, r ← n − 1, m;
   while (ℓ ≤ r) {
      m ← ⌊(ℓ + r)/2⌋; if (A[m] ≡ k) return m;
      if (A[m] < k) ℓ ← m + 1; else r ← m − 1;
   }
   return Λ;
}
```

Iný spôsob ako implementovať binárne vyhľadávanie používa napoly otvorené intervaly tvaru $\langle \ell, r)$ a iba dvojhodnotové porovnanie (výsledok porovnanie je iba $<$, $\geq$, nie $<$, $>$, $=$):

Program 1.4 Binárne vyhľadávanie pomocou dvojhodnotového porovnania

```
int find(int k) {
   int ℓ ← 0, r ← n, m;
   while (r − ℓ > 1) if (k < A[m ← ⌊(ℓ + r)/2⌋]) r ← m; else ℓ ← m;
   return (A[ℓ] ≡ k) ? ℓ : Λ;
}
```

Netreba snáď ani dodávať, že v utriedenom poli sú operácie *minimum*, *maximum*, *predecessor* a *successor* triviálne (jednoducho vrátia prvky $A[0]$, $A[n-1]$, $A[i-1]$, resp. $A[i+1]$).

Tretí možný spôsob je použiť utriedený obojsmerný kruhový spájaný zoznam. Každý záznam bude obsahovať smerník na nasledujúci (najmenší väčší) a na predchádzajúci prvok. Následník posledného prvku je prvý a predchodca prvého je posledný prvok. Nevýhodou oproti predchádzajúcemu riešeniu je, že napriek tomu, že sú prvky utriedené, jediný spôsob, ako prechádzať štruktúrou je pomocou smerníkov, ktoré neumožňujú náhodný prístup a teda ostáva nám iba sekvenčné vyhľadávanie. Na druhej strane výhodou je flexibilita zoznamu: v konštantnom čase, prehodením pár smerníkov (bez nutnosti posúvať značnú časť poľa), vieme vložiť nový záznam medzi dva dané záznamy, vymazať daný prvok. Dokonca vložiť medzi dva prvky celý jeden zoznam alebo vymazať úsek prvkov vieme v $O(1)$.

Tu si ukážeme implementáciu zoznamu pomocou poľa. Pole A obsahuje kľúče, L a R obsahujú index predchodcu, resp. následníka. Voľné políčka aj $L[f]$ je voľné políčko – takto vieme rýchlo nájsť miesto, kam nový záznam uložiť. Pri vymazávaní vložíme políčko na začiatok tohto zoznamu. Pri programovaní spájaného zoznamu si treba dať vždy pozor na špeciálny prípad prázdneho zoznamu; tento problém sa dá elegantne riešiť technikou zarážky: pridáme jeden prvok na začiatok zoznamu, ktorý nebude predstavovať žiaden záznam, ale bude prítomný, aj keď je zoznam prázdny (v našom prípade je to nultý prvok). Pomocná funkcia *findgeq*(k) nájde najmenší prvok, ktorý je väčší alebo rovný k (využívame ju v *insert* aj *find*; vďaka zarážke taký prvok vždy existuje).

Program 1.5 Implementácia slovníka pomocou utriedeného obojsmerného kruhového zoznamu

```
int A[MAX], L[MAX], R[MAX], avail, t;
void init() { REP(i, MAX) L[i] ← i + 1; R[0] ← L[0] ← Λ; A[0] ← ∞; avail ← 1; }
int findgeq(int k) { int i ← R[0]; while (A[i] < k) i ← R[i]; return i; }
#define FIND(k)   (A[t ← findgeq(k)] ≡ k ? t : Λ)
void insert(int k) { int r ← findgeq(k), i ← avail; avail ← L[i];
   A[i] ← k; R[i] ← r; L[i] ← L[r]; L[r] ← R[L[r]] ← i; }
void delete(int i) { R[L[i]] ← R[i]; L[R[i]] ← L[i]; A[i] ← avail; avail ← i; }
```

Nevýhodou obojsmerného spájaného zoznamu je, že potrebujeme až dva smerníky na jeden záznam, zatiaľ čo na pole žiaden; spájaný zoznam je teda pamäťovo náročnejší. Menej pamäte potrebuje jednosmerný spájaný zoznam (jeden smerník na záznam). Prehľad časových zložitostí

jednotlivých implementácií je v Tabuľke 1.1. Treba zvážiť, ktoré operácie aplikácia vyžaduje a podľa toho vybrať vhodnú štruktúru. Vo všeobecnosti sú však tieto riešenia pomalé a teda vhodné len pre malé n.

	pole		spájaný zoznam			
			jednosmerný*		obojsmerný†	
	netriedené	triedené	netriedený	triedený	netriedený	triedený
insert	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
find	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
delete	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
minimum	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
maximum	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
predecessor	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
successor	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
join	$O(m)$	$O(m)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
split	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Tabuľka 1.1: Porovnanie implementácií pomocou poľa a spájaného zoznamu
* predpokladáme, že máme smerník na začiatok aj koniec zoznamu
† predpokladáme že je to kruhový zoznam

§ **Efektívne reprezentácie.** Efektívna implemntácia je pomocou vyhľadávacích stromov, ktoré popíšeme v nasledujúcich kapitolách a hašovanie. Na tomto mieste si len v krátkosti predstavíme druhú metódu a porovnáme ju s vyhľadávacími stromami.

Hašovanie je metóda digitálneho vyhľadávania. Základnou myšlienkou hašovania je použiť funkciu, ktorá *vypočíta*, kam uložíme prvok, resp. kde ho máme hľadať. Konkrétne, nech K je množina všetkých kľúčov; *hašovacia tabuľka* je pole $T[\texttt{MAX}]$ spolu s hašovacou funkciou $h : K \to \{0, 1, \ldots, \texttt{MAX} - 1\}$ (teda funkciou transformujúcou kľúče na indexy do poľa T). V triviálnom prípade, ak $K = \{0, 1, \ldots, \texttt{MAX} - 1\}$, stačí za h zvoliť identitu. V praxi však býva množina K príliš veľká; v tom prípade je jednou z najjednoduchších možností zobrať $h(k) = k \bmod \texttt{MAX}$ (je dobré, ak v tomto prípade MAX je prvočíslo nie blízko mocniny 2). Od dobrej hašovacej funkcie očakávame, že bude jednoduchá, bude sa dať rýchlo vypočítať a bude kľúče distribuovať do tabuľky T rovnomerne. Môže sa stať, že dva prvky $x \neq x'$ sa zahašujú na rovnakú pozíciu $h(x) = h(x')$. Takémuto javu hovoríme *kolízia*. Existuje viacero spôsobov, ako ju riešiť, jedným z nich je použitie spájaných zoznamov: prvky s rovnakou hašovacou hodnotou jednoducho zreťazíme.

Bližšie pojednanie o hašovacích funkciách či riešení kolízií je nad rámec tejto práce. Porovnajme však hašovacie tabuľky s vyváženými stromami, ktoré opisujeme v nasledujúcich kapitolách:

- Hašovacie tabuľky sú rýchlejšie ako vyvážené stromy. Priemerná časová zložitosť hašovacej tabuľky je $O(1)$ (ak je hašovacia funkcia „dobrá“). V najhoršom prípade je to však až $O(n)$, zatiaľ čo vyvážené stromy garantujú zložitosť $O(\log n)$.

- Vyhľadávacie stromy zachovávajú usporiadanie kľúčov, takže vieme efektívne hľadať predchodcu či následníka. Napríklad vypísanie prvkov utriedených podľa kľúča vieme s vyhľadávacím stromom v lineárnom čase, zatiaľ čo s hašovacou tabuľkou nie. Ak je vyhľadávanie v strome neúspešné, vieme napríklad povedať najbližší väčší alebo najbližší menší prvok. Vieme vyhľadať prvky, ktorých kľúče sú v nejakom intervale a jednoducho ich rozšíriť o ďalšie operácie, napr. hľadanie i-teho najväčšieho prvku. Taktiež ich väčšinou vieme efektívne deliť a spájať. To s hašovacími tabuľkami nevieme.

Hašovacie funkcie sú teda dobrá voľba, ak potrebujeme „obyčajný“, neusporiadaný slovník. Ak je však dôležité usporiadanie prvkov podľa kľúčov, správna voľba sú vyhľadávacie stromy.

§ **Slovníky v programovacích jazykoch.** Slovník je veľmi dôležitý abstraktný dátový typ a tak sa jeho implementácie nachádzajú v mnohých programovacích jazykoch. Napríklad štandardná knižnica jazyka C++, Standard Template Library obsahuje triedy (presnejšie template) *set*, *multiset*, *map* a *multimap*:

- *set* je usporiadaná množina; podporuje operácie vkladania (*insert*), vymazávania (*erase*) a množinové operácie: test na inklúziu (*includes*), zjednotenie (*set_union*), prienik (*set_intersection*), rozdiel (*set_difference*) a symertickú diferenciu (*set_symmetric_difference*)
- *multiset* je podobná ako *set*, ale môže obsahovať aj viacero rovnakých prvkov,
- *map* je usporiadané asociatívne pole; umožňuje vyhľadávanie podľa kľúča, definuje tiež tzv. iterátor („vylepšený smerník"), ktorým sa dá prechádzať cez prvky v *map*
- *multimap* je podobné ako *map*, ale kľúč môže byť asociovaný s viacerými hodnotami.

Všetky uvedené dátové štruktúry sú implementované pomocou vyvážených vyhľadávacích stromov (zväčša pomocou červeno-čiernych stromov).

Program 1.6 Príklad použitia triedy map v C++

```
#include ⟨iostream⟩
#include ⟨map⟩
#include ⟨string⟩
using namespace std;

int main() {
   map⟨string, string⟩ phone_book;
   phone_book ["Sally Smart"] ← "555-9999";
   phone_book ["John Doe"] ← "555-1212";
   phone_book ["J. Random Hacker"] ← "553-1337";

   map⟨string, string⟩ :: iterator curr;
   for (curr ← phone_book.begin(); curr ≢ phone_book.end(); curr++)
      cout ≪ curr→first + "=" + curr→second ≪ endl;
   return 0;
}
```

Viaceré jazyky majú zabudovanú syntaktickú podporu pre asociatívne polia. Moderné skriptovacie jazyky (awk, Perl, tcl, Javascript, Python, Ruby) majú dokonca asociatívne polia ako základný dátový typ (namiesto obyčajného poľa):

Program 1.7 Príklad použitia slovníka v Pythone

```
phonebook ← {'Sally Smart' : '555-9999',
             'John Doe' : '555-1212',
             'J. Random Hacker' : '553-1337'}
for key in phonebook :
   print key, phonebook [key]
```

Kapitola 2

Binárny vyhľadávací strom

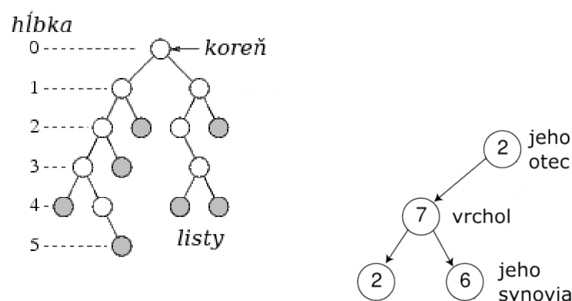
2.1 Strom – matematická štruktúra

Trees like to have kids climb on them,
but trees are much bigger than we are
and much more forgiving.
Diane Frolov & Andrew Schneider

§ **Usporiadaný strom.** *Strom* je súvislý acyklický graf. Strom s n vrcholmi má $n-1$ hrán a práve jednu cestu medzi ľubovoľnou dvojicou vrcholov. *Zakorenený strom* je strom T spolu s vrcholom $r = root(T)$, ktorý nazývame *koreň*. Ak v leží na jedinečnej ceste medzi vrcholom w a koreňom r, potom v je *predok (ancestor)* w a w je *potomok (descendant)* v (ak $v \neq w$, je to *vlastný* predok alebo potomok). Ak sú navyše v a w susedné vrcholy, potom v je *otec (father)*, *rodič (parent)* w, značíme $v = p(w)$ a w je *syn (son)* alebo *dieťa (child)* v. (V tejto analógií sa dá veselo pokračovať: brata otca nazývame *strýko (uncle)*, $p(p(v)) = p^2(v)$ je *starý otec*, $p^{n+1}(v)$ je „pra^n-starý otec“ v; napríklad v má bratratnca w z k-teho kolena, ak $p^{k+1}(v) = p^{k+1}(w)$, ale $p^k(v) \neq p^k(w)$; brat je bratranec z 0-tého kolena).

Každý vrchol okrem koreňa má práve 1 rodiča. *Stupeň (degree)* vrcholu v (značíme $\deg v$) je počet jeho detí (v grafovej terminológií by to bol vstupný vrchol, ak by mal zakorenený strom hrany orientované smerom ku koreňu). Vrchol, ktorý nemá deti ($\deg v = 0$) voláme *konečný vrchol (terminal node)* alebo častejšie jednoducho *list (leaf)*.

Hĺbku (depth) vrcholu v, $d(v)$ definujeme ako dĺžku cesty z v do koreňa r. Teda $d(v) = 0$, ak $v = r$ je koreň a $d(v) = d(p(v)) + 1$ inak – všetci synovia sú o 1 hlbšie ako je ich otec. Vrcholy s rovnakou hĺbkou sú *susedia* a tvoria jednu *úroveň*. *Výšku (height)* vrcholu v, $h(v)$ definujeme: $h(v) = 0$, ak v je list a $h(v) = \max_w\{h(w) \mid v = p(w)\} + 1$ inak. Teda listy sú vo výške nula a vrchol je vždy o 1 vyššie ako jeho najvyšší syn.



Obr. 2.1: Terminológia

Podstrom je podgraf indukovaný potomkami vrcholu. Väčšinou nebudeme rozlišovať medzi vrcholom v a podstromom s koreňom vo vrchole v. Teda napríklad výškou stromu T, $h(T)$ máme na mysli výšku koreňa tohto stromu, $h(root(T))$ a namiesto „podstrom s koreňom vo vrchole v“ budeme hovoriť iba „podstrom v“.

Nakoniec *usporiadaný strom* je taký zakorenený strom, v ktorom je poradie detí každého vrcholu dané (t.j. rozlišujeme prvého, druhého syna, atď).

§ Binárny strom. *Binárny strom (binary tree)* definujeme rekurzívne: binárny strom je buď *prázdny* (značíme Λ), alebo je to trojica $T = (T_L, r, T_R)$, kde T_L a T_R sú binárne stromy. T_L a T_R voláme ľavý a pravý podstrom, r je koreň. Korene ľavého a pravého podstromu v značíme $left(v)$ a $right(v)$ (ak je daný strom prázdny, potom $left(v) = \Lambda$, resp. $right(v) = \Lambda$). Otca vrcholu v značíme $p(v)$ (ak v nemá otca – je to koreň – $p(v) = \Lambda$). Ak $left(v), right(v) \neq \Lambda$, tak platí $p(left(v)) = p(right(v)) = v$.

Hĺbku vrcholu v binárnom strome definujeme rekurzívne: $d(\Lambda) = 0$, a $d(v) = d(p(v))+1$. *Výška* vrcholu je $h(\Lambda) = 0$ a $h(v) = \max(h(left(v)), h(right(v)))+1$. *Počet vrcholov* podstromu s koreňom v značíme $size(v)$: $size(v) = 0$, ak $v = \Lambda$, inak $size(v) = size(left(v)) + size(right(v)) + 1$. Počet vrcholov celého stromu budeme v celej práci značiť n.

Upozorňujeme, že je rozdiel medzi binárnymi stromami a usporiadanými stromami s maximálnym stupňom 2: Ak má vrchol iba jedného syna, v binárnom strome rozlišujeme, či je ľavý alebo pravý – to pri usporiadaných stromoch nerobíme. Existuje však vzťah medzi binárnymi stromami a usporiadanými stromami so stupňom 0 alebo 2.

§ Rozšírený binárny strom. Pri analýze našich algoritmov sa nám bude viac hodiť takáto definícia: *Rozšírený binárny strom (extended binary tree*, Knuth ho označuje ako *b-strom)* je neprázdny usporiadaný strom, ktorého každý vrchol má stupeň 0 alebo 2. Listy takéhoto stromu voláme *externé vrcholy*, zvyšné sú *interné* (na obrázkoch budeme interné vrcholy budeme značiť krúžkami, externé štvorčekmi). Definujeme *veľkosť* stromu T ako počet externých vrcholov stromu T. Veľkosť T značíme $|T|$. Všimnime si, že existuje bijekcia medzi binárnymi a rozšírenými binárnymi stromami: ak každému vrcholu v binárnom strome pridáme chýbajúceho syna, dostaneme rozšírený binárny strom. Naopak, ak rozšírenému binárnemu stromu listy „ufikneme“, dostaneme „obyčajný“ binárny strom.

Podotknime, že hĺbku a výšku sme definovali tak, že spoločné vrcholy binárneho a príslušného rozšíreného binárneho stromu majú podľa oboch definícií rovnajú výšku aj hĺbku. Preto nebudeme rozlišovať medzi rozšírenými a obyčajnými binárnymi stromami. Ak budeme hovoriť o vrcholoch, máme na mysli interné vrcholy.

Definujme *externú vzdialenosť* v strome *(external path length)* $E(T)$ ako súčet hĺbok všetkých externých vrcholov a podobone *internú vzdialenosť (internal path length)* $I(T)$ ako súčet hĺbok všetkých interných vrcholov. $I(T)$ je teda súčet vzdialeností všetkých interných vrcholov od koreňa a $I(T)/\,size(T)$ je *priemerná hĺbka* (interných) vrcholov. Podobnú interpretáciu má $E(T)$.

Úplný binárny strom je strom, ktorý má všetky úrovne zaplnené, možno až na poslednú, ktorá je zaplnená zľava. Tento strom má najmenšie E aj I.

§ Vzťahy medzi definovanými veličinami. Pre binárne stromy sme definovali počet vrcholov v podstrome v, $size(v)$; pre rozšírené stromy sme definovali veľkosť $|v|$. Medzi týmito dvoma veličinami je jednoduchý vzťah: $|v| = size(v) + 1$. Stačí spočítať hrany rozšíreného stromu dvoma spôsobmi: Každý vrchol okrem koreňa má jedného otca, teda hrán je $|v|+size(v)-1$ a naopak, každý interný vrchol má práve dvoch synov, teda hrán je $2\,size(v)$; odtiaľ dostaneme $|v| = size(v) + 1$.

Ďalej $E = I + 2n$, kde n je počet (interných) vrcholov stromu. Ak T je úplný, potom $I = \sum_{k=1}^{n} \lfloor \lg k \rfloor = |T|q - 2^{q+1} + 2$, kde $q = \lfloor \lg |T| \rfloor$.

Ak definujeme hĺbku stromu $d(T)$ ako hĺbku najhlbšieho externého vrcholu, potom hĺbka stromu sa rovná jeho výške – $h(T) = d(T)$. V hĺbke d je maximálne 2^d vrcholov. Platí: $|T| \leq 2^{h(T)}$, teda $h(T) \geq \lg |T|$. Každý strom má výšku aspoň $\lceil \lg |T| \rceil$. Pre úplné stromy navyše platí $|T| > 2^{h(T)-1}$, teda $h(T) = \lceil \lg |T| \rceil$. Úplné stromy nie sú jediné také: posledná úroveň môže byť

zaplnená ľubovoľne. Stromy, v ktorých $|\,size(left(v)) - size(right(v))| \le 1$ voláme *dokonale vyvážené*. Všeobecnejšie, stromy, pre ktoré je $h(T) = \lceil \lg |T| \rceil$ voláme *optimálne* (úplné a dokonale vyvážené stromy sú príkladom optimálnych stromov). Ak c je konštanta, potom stromy výšky $h(T) \le c \cdot \lceil \lg |T| \rceil$ voláme *vyvážené*.

§ **Viac-cestné stromy.** Uvedené definície sa dajú rozšíriť: *Pozičný strom* je zakorenený strom, kde každý vrchol má priradené kladné celé číslo, ktoré určuje, koľkatý je to syn (ak niektoré číslo chýba, daný podstrom je prázdny). Pozičný strom, s číslami $\le m$ voláme *m-árny strom*. *Rozšírený m-árny strom* je usporiadaný strom, ktorého každý vrchol má 0 alebo m synov. $|v| = (m-1)\,size(v) + 1$. V hĺbke d je maximálne m^d vrcholov a teda $h(T) \ge \lceil \log_m |T| \rceil$, pre úplné stromy nastáva rovnosť.

2.2 Reprezentácia stromov

§ **Reprezentácia stromov v počítači.** Implementácia binárnych stromov je priamočiara: každý vrchol obsahuje kľúč, (prípadne dáta), smerník na ľavého a pravého syna, prípadne, ak potrebujeme, môžeme pridať smerník na otca (ak niektorý syn alebo otec neexistuje, smerník má hodnotu Λ). Smerník na otca môže uľahčiť pohyb v strome, väčšina algoritmov sa však dá naprogramovať bez neho, či už postupom zhora nahol, alebo rekurzívne s pomocou zásobníka, v ktorom sú uložené vrcholy na nejakej ceste od koreňa. Navyše potrebujeme jeden smerník, ktorý ukazuje na koreň; ak je to Λ, strom je prázdny.

My si ukážeme veľmi šikovnú implementáciu v poli. Podobne ako v prípade spájaného zoznamu budú voľné miesta v spájanom zozname. Nulté políčko použijeme ako zarážku. Vyhneme sa tak ošetrovaniu niekoľkých špeciálnych prípadov:

Program 2.1 Reprezentácia binárneho stromu v poli

```
#define Λ  0
int K[MAX], c[2][MAX];
#define L  c[0]
#define R  c[1]
#define ROOT  R[Λ]
#define LEFT  L[Λ]
void init() { ROOT ← Λ; REP(i,MAX) L[i] ← i + 1; }
int newv(int x) { int v ← LEFT; LEFT ← L[v]; K[v] ← x; L[v] ← R[v] ← Λ return v; }
void delv(int v) { L[v] ← LEFT; LEFT ← v; }
```

V poli k sú uložené kľúče, v poli c sú indexy detí vrcholov; vďaka definíciám môžeme písať jednoducho `L`$[v]$ pre $c[0][v]$ – ľavého syna v a `R`$[v]$ pre $c[1][v]$ – pravého syna v (mohli sme zadeklarovať priamo polia L a R, význam takejto implementácie sa objasní ochvíľu). Pole p obsahuje indexy otcov vrcholov. `ROOT` ukazuje na koreň stromu, `LEFT` je prvé voľné miesto. Procedúra *init*() inicializuje prázdny strom a zreťazí voľné miesta do spájaného zoznamu. Funkcia *newv*() alokuje nové miesto pre vrchol, uloží ho tam a vráti jeho index. Procedúra *delv*() naopak uvoľní dané miesto – zaradí ho späť do spájaného zoznamu.

Podobne m-árne stromy vieme reprezentovať „veľkými vrcholmi" s poľom $c[m]$[`MAX`] detí. Iná možnosť je použiť tzv. *left-child, right-sibling* (skrátene LCRS) reprezentáciu: každý vrchol si pamätá iba najľavejšieho syna a pravého brata. Inými slovami, všetci bratia sú pospájaný do spájaného zoznamu. Na túto reprezentáciu stačia pre každý vrchol iba dva smerníky: L a R; napríklad tretí syn vrcholu v je $R[R[L[v]]]$. Posledný zo súrodencov má $R[w] = \Lambda$; druhá možnosť je, že každému vrcholu pridáme jeden bit, aby sme vedeli rozoznávať najpravejšie vrcholy zo súrodencov a v takom prípade môžeme položiť $R[w] = p(w)$. Z každého vrcholu teraz máme prístup k otcovi tak, že budeme nasledovať smerníky R, kým sa nedostaneme k najpravejšiemu bratovi a jeho R-smerník ukazuje na otca.

Poznamenajme, že

- táto reprezentácia poukazuje na bijekciu medzi binárnymi a viaccestnými stromami – viaccestné stromy sa dajú kódovať ako binárne stromy,
- pre binárne stromy to znamená, že vieme stromy reprezentovať pomocou 2 smerníkov a 1 bitu na vrchol (namiesto troch smerníkov; a pritom stále vieme v konštantnom čase vypočítať ľavého, pravého syna aj otca; je to však za cenu zložitejšieho kódu), pozri program 2.2,
- pre binárne vyhľadávacie stromy, ktoré o chvíľu definujeme sa vieme „zbaviť“ aj tohto jedného bitu.

Program 2.2 Left-child right-sibling reprezentácia binárneho stromu

```
int K[MAX], l[MAX], r[MAX];
bool isRight[MAX];
#define L(v)   isRight[l[v]] ? Λ : l[v]
#define R(v)   isRight[l[v]] ? l[v] : r[l[v]]
#define P(v)   isRight[v] ? r[v] : r[r[v]]
void init() { r[Λ] ← l[Λ] ← Λ; isRight[Λ] ← 1; }
```

§ **Traverzovanie.** Jednou zo základných operácií na stromoch je traverzovanie. Algoritmy, ktoré manipulujú so stromami často potrebujú postupne prejsť všetky vrcholy stromu (napríklad vypísanie, kopírovanie stromu, kreslenie stromu, pozri kapitolu 8). Existujú tri základné rekurzívne postupy:

- *preorder* – navštívime koreň a potom postupne rekurzívne všetkých synov,
- *postorder* – navštívime najskôr rekurzívne všetkých synov, nakoniec koreň
- pre binárny strom vieme definovať *inorder*, tiež *symetrické usporiadanie* – navštívime ľavého syna, potom koreň, potom pravého syna.

Preorder a postorder je vlastne prehľadávanie do hĺbky, pričom v preorder vrchol spracujeme hneď, ako ho objavíme, v postorder, až keď sa vraciame.

Napríklad strom na obr. 2.3 má preorder, inorder a postorder zápisy:

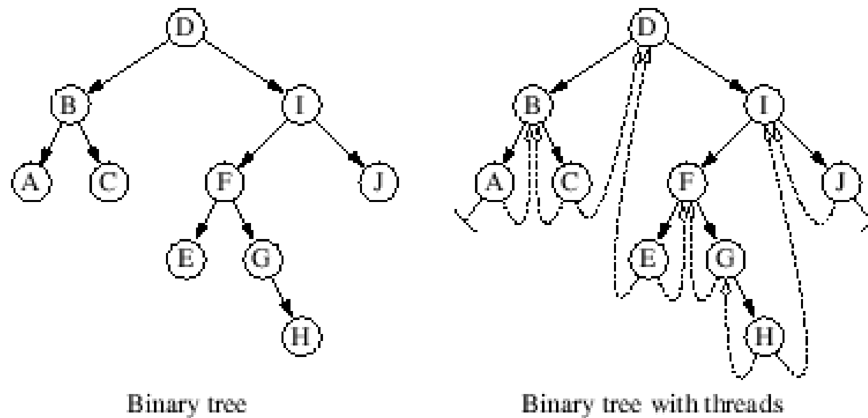
preorder	15 6 3 2 4 7 13 9 18 17 20
inorder	2 3 4 6 7 9 13 15 17 18 20
postorder	2 4 3 9 13 7 6 17 20 18 15

Program 2.3 Traverzovanie binárneho stromu

```
void traverse(int v) { if (¬v)return;
    preorder-visit(v); traverse(left(v)); inorder-visit(v); traverse(right(v)); postorder-visit(v);
}
```

Traverzovanie nám na vrcholoch definuje usporiadanie – má zmysel hovoriť, ktorý vrchol bude spracovaný skôr, ktorý neskôr, ktorý vrchol spracujeme hneď po alebo hneď pred daným vrcholom. Nás bude zaujímať najmä symetrické usporiadanie. Definujeme $succ(v)$ = následník vrcholu v a $pred(v)$ = predchodca vrcholu v v symetrickom usporiadaní (inorder),

§ **Prepletené stromy.** Ukázali sme, že počet externých vrcholov je o 1 väčší ako počet interných vrcholov. Inými slovami, pri klasickej implementácií je viac ako polovica smerníkov rovná Λ – odkazujú na prázdny podstrom. Toto miesto sa však dá rozumne využiť: Ku každému smerníku pridáme 1 bit, ktorý bude indikovať, či je príslušný podstrom prázdny. Ak je podstrom neprázdny, smerník bude ukazovať na koreň príslušného podstromu tak, ako v jednoduchej reprezentácií. Ak je však prázdny, bude $left(v) = pred(v)$, resp. $right(v) = succ(v)$. Takúto reprezentáciu voláme



Obr. 2.2: Prepletený strom

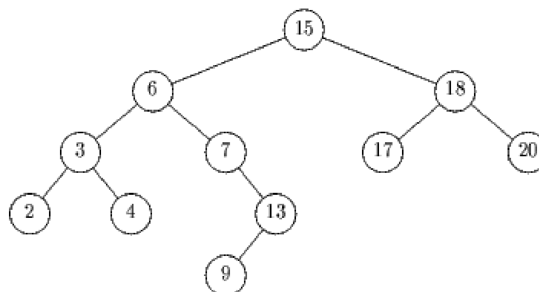
prepletený strom (threaded tree). Ak využijeme iba smerník na pravý podstrom a *left*(v) nechávame Λ, ide o tzv. *right-threaded tree*.

Táto informácia nám pomôže pri traverzovaní; následníka v inorder usporiadaní nájdeme jednoducho: Ak je pravý podstrom prázdny, smerník ukazuje priamo na následníka; ak je neprázdny, nájdeme „najľavejší vrchol" pravého podstromu, t.j. ideme raz pravým smerníkom a potom, kým sa dá, ľavým. Výhodou je rýchlosť, jednoduchosť a hlavne, nepotrebujeme rekurziu alebo zásobník na to, aby sme prešli v inorder usporiadaní celý strom.

Poznamenajme, že následníka vieme hľadať aj vtedy, keď máme smerník na otca: ak má vrchol pravého syna, následník je najľavejší vrchol pravého podstromu, ak ho nemá, následník je najbližší taký predok, že daný vrchol sa nachádza v jeho pravom podstrome, teda budeme nasledovať smerníky na otca, kým sa nestane, že vrchol, z ktorého sme prišli je ľavým synom.

2.3 Binárny vyhľadávací strom.

§ **Definícia.** Binárny strom voláme *binárny vyhľadávací strom* (*binary search tree*, skrátene BST), ak pre každý jeho vrchol platí, že všetky kľúče v ľavom podstrome sú menšie ako kľúč v jeho koreni a ten je menší ako všetky kľúče v pravom podstrome.



Obr. 2.3: Binárny vyhľadávací strom

Minimum je „najľavejší" a maximum „najpravejší" vrchol stromu – dostaneme sa k nemu tak, že pôjdeme z koreňa a budeme sledovať iba ľavé, resp. pravé smerníky. Inorder výpis takéhoto stromu je utriedený zoznam; pojmy „následník" a „následník v inorderi" (analogicky s predchodcom)

splývajú, $predecessor(v) = pred(v)$ a $successor(v) = succ(v)$.

Program 2.4 Minimum a maximum podstromu v

```
int min(int v) { while (L[v]) v ← L[v]; return v; }
int max(int v) { while (R[v]) v ← R[v]; return v; }
```

§ **Vyhľadávanie.** Vyhľadávanie vo vyhľadávacom strome je jednoduché: Začneme v koreni stromu; Ak je kľúč v danom vrchole, našli sme ho. Ak je hľadaný kľúč menší ako kľúč v danom vrchole, musí sa nachádzať v ľavom podstrome (ak je ľavý podstrom prázdny, hľadanie končí neúspešne). Naopak, ak je hľadaný kľúč väčší, musí sa nachádzať v pravom podstrome (ak je ten prázdny, hľadanie končí neúspešne).

Vyhľadávanie vieme implementovať jednoducho pomocou rekurzie:

Program 2.5 Vyhľadávanie v BST

```
int find(v, k) {
    if (v ≡ Λ ∨ k ≡ K[v]) return v;
    if (k < K[v]) find(L[v], k); else find(R[v], k);
}
#define FIND(k)    find(ROOT, k)
```

Ukážeme si ešte šikovnejšiu implementáciu. Najskôr si zadefinujme veľmi mocné a užitočné makro C (napríklad od slov Choose Child):

```
#define C(v, k)    c[(k) > K[v]][v]
```

$\mathtt{C}(v, k)$ je práve ten syn vrcholu v, do ktorého sa máme vydať z v, ak hľadáme kľúč k. Druhým trikom je použitie zarážky: Na pozíciu Λ uložíme kľúč k – takto nemusíme kontrolovať, či je daný podstrom prázdny, hľadanie vždy skončí úspešne: ak sa v strome kľúč nenachádza, hľadanie skončí v Λ. Vďaka tomu, že testovaná podmienka je jednoduchšia je tento prístup aj rýchlejší.

Program 2.6 Vyhľadávanie so zarážkou

```
int find(int k) { int v ← ROOT; K[Λ] ← k; while (k ≢ K[v]) v ← C(v, k); return v; }
```

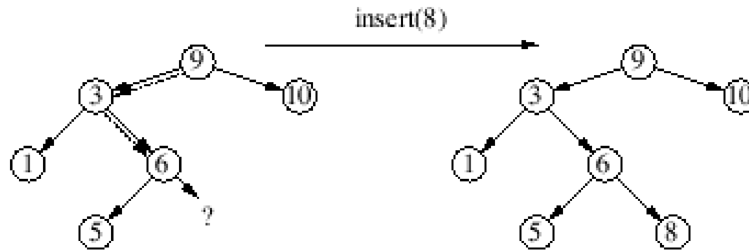
Ešte iný spôsob vyhľadávania v binárnom strome vymyslel Andersson [And91], popíšeme si ho v paragrafe o vymazávaní.

§ **Vkladanie.** Vkladanie je rovnako jednoduché ako vyhľadávanie: Zjavne musíme prvok vložiť tam, kde ho neskôr funkcia *find* nájde. Začneme preto v koreni a podľa toho, či je kľúč menší alebo väčší vkladáme nový vrchol rekurzívne do ľavého alebo pravého podstromu. Ak natrafíme na smerník Λ, presmerujeme ho na novovytvorený vrchol a skončíme.

Program 2.7 Vkladanie do BST

```
void insert(int k) { int v ← ROOT; while (C(v, k)) v ← C(v, k); C(v, k) ← newv(k); }
```

§ **Vymazávanie.** Ak je vrchol list, ostránime ho ľahko – iba prepíšeme príslušný smerník jeho otca na Λ (a uvoľníme pamäť). Ak má vrchol len jedného syna, tiež vieme vrchol ľahko odstrániť: stačí prepojiť jeho otca s týmto synom. Najkomplikovanejší je tretí prípad: ak má vrchol v oboch synov, nájdeme $succ(v)$. Platí, že následník je najľavejší vrchol v pravom podstrome, teda nemá ľavého syna. Následníka v vieme teda ľahko odstrániť; vrchol v nahradíme týmto vrcholom. Keďže následník v bol zároveň najmenším väčším prvkom v strome, usporiadanie sa neporuší – vľavo budú naďalej menšie, vpravo väčšie kľúče.



Obr. 2.4: Vkladanie do BST

V skutočnosti sa dá vymazávanie redukovať na dva prípady: Buď je pravý podstrom prázdny, vtedy stačí napojiť ľavý podstrom na otca vrcholu, alebo neprázdny, vtedy ho nahradíme jeho nasledníkom.

Ak máme smerníky na otca, môžeme vymazávanie implementovať napríklad takto:

Program 2.8 Vymazávanie z BST so smerníkom na otca

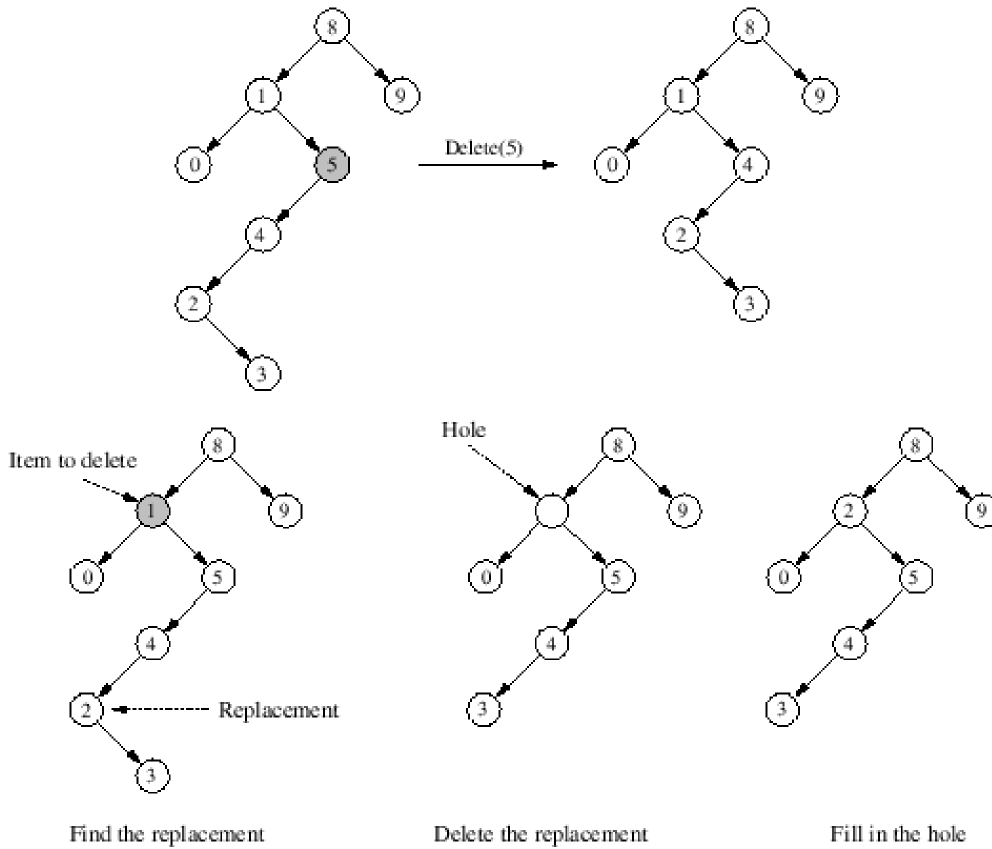
```
void delete(int v) {
   if (¬v) return;
   if (R[v]) { int w ← min(R[v]); K[v] ← K[w]; delete(w); }
   else { C(p[v], K[v]) ← L[v]; delv(w); }
}
#define DELETE(k)   delete(find(k))
```

Iný spôsob, pomocou dvojhodnotového porovnania, objavil Andersson. Kľúč porovnáme s kľúčom vo vrchole. Ak je hľadaný kľúč menší, ideme doľava, inak si zapamätáme (v ℓ) daný vrchol (ako kandidáta na rovnosť) a posunieme sa doprava. Takto pokračujeme, kým nenarazíme na Λ. Odkedy sa prvýkrát pohneme doprava platí, že $key(\ell) \leq k$. Na konci teda bude platiť $key(\ell) \leq k$, navyše ℓ bude najväčší taký vrchol. V nasledujúcom programe si navyše zistíme b – posledný vrchol na tejto ceste (pred Λ) a jeho otca p. Potom vieme aj ľahko vymazávať – ak je totiž $key(\ell) = k$, potom b je jeho následník.

Program 2.9 Operácie BST pomocou dvojhodnotového porovnania

```
int ℓ, b, p;
void findm(int k) { b ← ROOT; ℓ ← p ← Λ;
   while (v) { p ← b; if (k < K[b ← v]) v ← L[v]; else { ℓ ← v; v ← R[v]; } }
}
#define FIND(k)   (findm(k), K[ℓ] ≡ k ? ℓ : Λ)
#define INSERT(k)   { findm(k); C(b, k) ← newv(k); }
void delete(int k) {
   findm(k);
   if (K[ℓ] ≢ k) return;
   K[ℓ] ← K[b]; C(p, k) ← R[b]; delv(b);
}
```

§ **Časová zložitosť.** Popísali sme si slovníkové operácie na BST. Aká je ich časová zložitosť? Prehľadávanie stromu trvá lineárny čas – po každej hrane prejdeme najviac dvakrát (tam a späť), resp. pri right-threaded tree implementácií prejdeme každým smerníkom práve raz a tých je lineárne veľa. Preto, hoci jednotlivé volania funkcie hľadajúcej predchodcu alebo následníka môžu trvať dlhšie, priemerná časová zložitosť je konštantná.



Obr. 2.5: Vymazávanie z BST

Čo sa týka zvyšných operácií: *insert*, *find*, *delete*, *minimum*, *maximum*, ich časovú zložitosť vieme ohraničiť výškou stromu, t.j. $O(h(T))$. Spomínali sme, že pri n vrcholoch je najmenšia možná výška $\lceil \lg |T| \rceil$. V najhoršom prípade to je však $n-1$, čo nie je lepšie ako obyčajný spájaný zoznam. Ako sa však ukazuje, v priemernom prípade je strom vyvážený.

§ Výška náhodne generovaného binárneho vyhľadávacieho stromu. Ak by sme skúmali náhodné binárne stromy v tom zmysle, že každý z C_n (Katalanovo číslo) binárnych stromov o n vrcholoch má rovnakú pravdepodobnosť výskytu, potom výška priemerného stromu by bola rádovo $\sqrt{n}$.

Nás však zaujíma trochu iný pojem. Je ním *náhodne generovaný* BST. Náhodne generovaný BST je taký BST, ktorý vznikne postupným vkladaním (popísanou procedúrou *insert*) *náhodnej permutácie* prvkov, pričom každá z $n!$ permutácií má rovnakú pravdepodobnosť.

Tvrdíme, že očakávaná výška vrcholu v náhodne generovanom strome je $O(\log n)$. Bez ujmy na všeobecnosti predpokladajme, že máme strom s kľúčmi z množiny $\{1, 2, \ldots, n\}$ a chceme vedieť hĺbku m. Očakávaná hĺbka je priemerná hĺbka cez všetky permutácie π množiny $\{1, 2, \ldots, n\}$. Rozdeľme všetky prvky do dvoch množín: $m_{\leq} = \{1, 2, \ldots, m\}$ a $m_{\geq} = \{m, \ldots, n\}$. Nech A je množina všetkých predchodcov m (vrátane m). Množiny $m_{\leq}$ a $m_{\geq}$ nezávisia od permutácie π, ale A áno. Našou úlohou je vypočítať očakávanú dĺžku cesty k m, čo je $|A|-1 = |A \cap m_{\leq}| + |A \cap m_{\geq}| - 2$. Vďaka symetrií stačí vypočítať $|A \cap m_{\leq}|$. Zoraďme prvky $m_{\leq}$ podľa toho, v akom poradí sme ich vkladali do stromu (toto poradie vlastne dostaneme tak, že z π vyškrtáme čísla, ktoré nepatria do $m_{\leq}$). Všimnime si, že každá permutácia $m_{\leq}$ je rovnako pravdepodobná a prvok z $m_{\leq}$ patrí do A práve vtedy, keď je väčší ako všetky prvky v permutácií vľavo od neho.

Dostávame problém sekretárok z [CLRS01]: Máme kanceláriu a v náhodnom poradí k nám chodia sekretárky. Vždy, keď k nám príde lepšia sekretárka, ako máme, najmeme ju namiesto predošlej. Koľko sekretárok takýmto spôsobom najmeme? Prvú najmeme vždy, druhú s pravdepodobnosťou 1/2 (ak bola lepšia ako prvá), tretiu s pravdepodobnosťou 1/3 (ak bola lepšia ako prvá aj druhá), atď. Všeobecne i-tu najmeme s pravdepodobnosťou $1/i$. Očakávaný počet najatí je teda H_n, kde n je počet sekretárok a $H_n = \sum_{i=1}^{n} 1/i$ je n-té harmonické číslo.

Teda očakávaná veľkosť $|m_{\leq} \cap A|$ je H_m a veľkosť $|m_{\geq} \cap A|$ je H_{n-m}. Platí, že $\ln(n+1) \leq H_n \leq \ln n + 1$, teda hĺbka bude menšia ako $\ln m + \ln(n-m) < 2\ln n \approx 1.386 \lg n$. Dôkaz silnejšieho výsledku, že očakávaná výška je $O(\log n)$ nájdeme je napríklad v [CLRS01] (tu sme dokázali, že očakávaná hĺbka vrcholu je $O(\log n)$; výška je hĺbka najhlbšieho externého vrcholu). Nájsť presnú hodnotu očakávanej výšky stromu je ťažký problém. V roku 1986 Devroye [Dev86] dokázal, že pre očakávanú výšku $E(H_n)$ stromu s n vrcholmi platí $E(H_n) \sim c \ln n$, kde $c \approx 4.31107$ je najväčší koreň rovnice $c \ln(2e/c) = 1$. V [Dev87] potom dokázal, že chyba $H_n - c\ln n = O(\sqrt{\ln n \ln\ln n})$. Až v 1995 sa mu podarilo dokázať spolu s Reedom [DR95], že $E(H_n) = c \ln n + O(\ln\ln n)$ a $\mathrm{D}(H_n) = O((\ln\ln n)^2)$. Najlepší výsledok pochádza od Reeda [Ree03], ktorý v roku 2003 dokázal, že

$$E(H_n) = c \ln n - \frac{3c}{2(c-1)} \ln\ln n + O(1)$$

a disperzia je $O(1)$. To znamená, že $E(H_n) \approx 2.9882 \lg n - 1.3537 \lg\lg n$ a podľa Čebyševovej nerovnosti $P(|h(T) - H_n| \geq k \cdot a) \leq 1/a^2$, kde T je strom o n vrcholoch a k je konštanta.

Čo sa týka vymazávania, v 1962 Hibbard dokázal, že po vymazaní náhodného prvku z náhodného stromu (popísaným algorimom) dostaneme strom, ktorý je opäť náhodný. Presnejšie, nech T je strom o n vrcholoch a $P(T)$ pravdepodobnosť, že náhodne generovaný strom o n vrcholoch bude práve T a nech $Q(T)$ je pravdepodobnosť, že náhodne generovaný strom o $n+1$ vrcholoch, z ktorého vymažeme náhodný prvok bude práve T (pri $P(T)$ berieme do úvahý všetkých $n!$ permutácií, pri $Q(T)$ berieme všetkých $(n+1) \cdot (n+1)!$ možností). Potom $P(T) = Q(T)$ [Knu78].

Poznamenajme ešte, že tento problém úzko súvisí s triediacim algoritmom quicksort (kde za pivota volíme prvý prvok) na náhodnej postupnosti. V quicksorte sa každý prvok porovná s pivotom a tak sa pole rozdelí na menšie a väčši prvky a algoritmus sa aplikuje rekurzívne na obe časti. Pri BST koreň implicitne rozdelí prvky na menšie a väčšie a vždy, keď vkladáme nový prvok, porovnáme ho s koreňom. Vieme, že triedenie algoritmom quicksort má v priemernom prípade zložitosť $O(n \log n)$, a teda aj počet porovnaní pri vytváraní náhodného stromu je $O(n \log n)$. Preto priemerná výška musí byť $O(\log n)$.

2.4 Vyvažovanie

§ **Vyvažovacie techniky.** Videli sme teda, že v priemernom prípade je výška BST veľmi dobrá. Ak však do BST začneme vkladať napríklad utriedenú postupnosť, dostaneme strom výšky $n-1$, čo je veľmi nepríjemné. Preto sa informatici snažili nájsť algoritmy, ktoré by udržali stromy „košaté“ – aby garantovali výšku $O(\log n)$.

Viacero takýchto algoritmov si ukážeme v nasledujúcich kapitolách. Väčšinou sa zvolí nejaká lokálna podmienka pre všetky vrcholy, ktorá garantuje výšku $O(\log n)$. Pri vložení nového vrcholu alebo vymazávaní sa môže stať, že sa podmienka poruší – vtedy sa pristúpi k opravám, tzv. vyvažovaniu stromu.

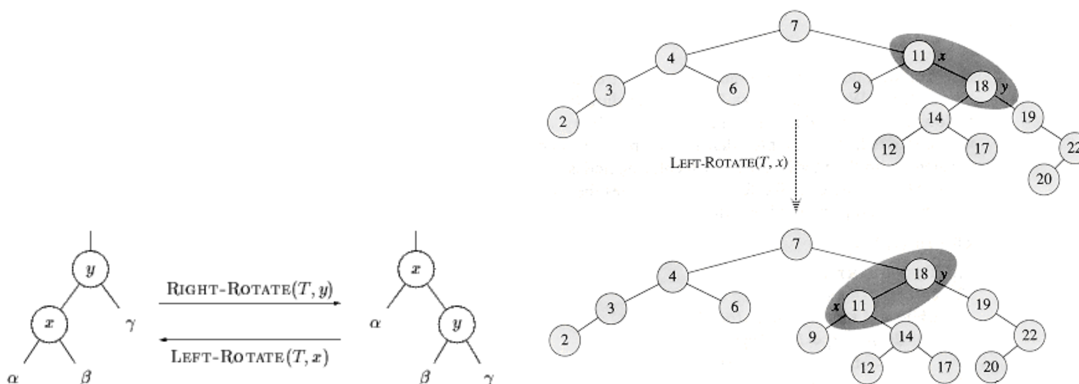
Vyvažovanie stromov sa väčšinou deje jedným z troch spôsobov:

- pomocou rotácií,
- spájaním a delením vrcholov,
- úplným prestavaním stromu.

§ Rotácie. Rotácia je veľmi lokálna operácia, ktorú vieme vykonať v konštantnom čase prepísaním pár smerníkov (pozri obr. ??a). Stačí zmeniť hranu y–x a x–β. Rotácia je jednoznačne určená dvoma vrcholmi, prípadne, ak poznáme smerníky na otcov, tým spodným z dvoch vrcholov. Prechod od obrázku vľavo k tomu vpravo budeme označovať $rotation(y, x)$, prechod doľava naopak $rotation(x, y)$. Ak budú dané smerníky na otcov, stačí $rotation(x)$ a $rotation(y)$ (smer rotácie je potom daný tým, či je vrchol ľavé alebo pravé dieťa svojho otca). Budeme tiež hovoriť, že rotujeme x, resp. y.

Rotácia má tri dôležité vlastnosti:

- zachováva symetrické usporiadanie; na obrázku sú v podstrome α prvky menšie ako x a prvky γ väčšie ako y a prvky β sú medzi x a y; pred aj po rotácií je α vľavo od x, γ vpravo od y a β medzi x a y,
- ľavá a pravá rotácia sú navzájom inverzné operácie,
- rotácia mení hĺbky vrcholov; pri pravej rotácií klesla hĺbka x a podstromu α a stúpla hĺbka y a podstromu γ, pri ľavej rotácií je tomu naopak.



Obr. 2.6: a) Schéma rotácie b) Rotácia v strome

Rotácia vyzerá ako veľmi nevinná operácia. Kombinovaním rotácií však vieme získať zložitejšie transformácie. Často sa napríklad používa tzv. dvojitá rotácia, kedy dvakrát zrotujeme ten istý vrchol1.

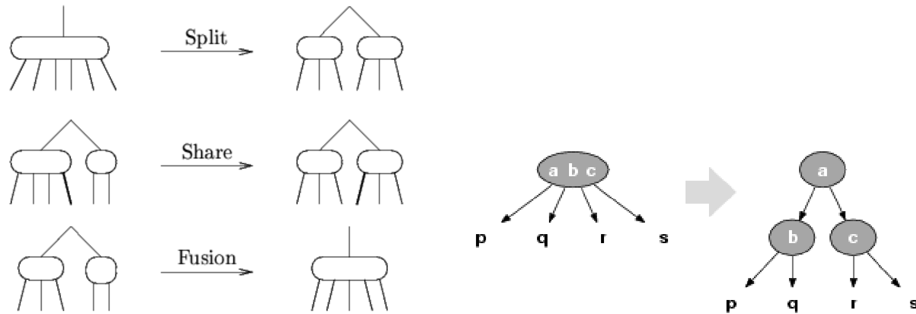
Každý strom sa dá prerobiť na ľubovoľný iný iba pomocou $O(n)$ rotácii! Jeden spôsob môže vyzerať takto: prevedieme strom na pravú reťaz vrcholov (strom, kde každý vrchol okrem koreňa je pravým synom svojho otca) a odtiaľ na nový strom. Ukážeme, že toto sa dá pomocou $O(n)$ pravých rotácií – odtiaľ už vyplýva uvedený výsledok, keďže pravú reťaz vieme prerobiť na ľubovoľný strom presne opačným spôsobom: v opačnom poradí použijeme na rovnakých miestach ľavé rotácie.

Strom prerobíme na pravú reťaz jednoducho: začneme v koreni a budeme aplikovať pravú rotáciu, kým sa to dá; potom prejdeme na vrchol vpravo a postup opakujeme. Keďže každou rotáciou do pravej reťaze pribudne jeden vrchol, stačí najviac $n - 1$ pravých rotácií. Každý strom vieme teda prerobiť na ľubovoľný iný pomocou $2n - 2$ rotácií.

Takto sa dá na množine stromov o n vrcholoch zaviesť vzdialenosť (tzv. *rotačná vzdialenosť*). Sleator, Tarjan a Thurston [STT88] dokázali, že v skutočnosti pre $n \geq 11$ sa ľubovoľný strom o n vrcholoch dá transformovať na iný iba pomocou $2n - 6$ rotácií. Navyše existuje nekonečná trieda dvojíc stromov, ktoré toľko rotácií naozaj potrebujú.

§ Spájanie a delenie vrcholov. Druhý spôsob, ako udržiavať stromy s malou výškou je nechať malú voľnosť v stupni vrcholu. Ak je vrchol „malý“, pridáme nový kľúč priamo doňho. Ak je „veľký“, môžeme ho rozdeliť na dva menšie. Podobne pri vymazávaní, ak sú dva vrcholy vedľa

seba „malé“, môžeme ich spojiť do jedného. Na obr. 2.7a máme zhrnuté operácie, ktoré prichádzajú do úvahy, keď povolíme vrcholy s rôznymi stupňami: rozdelenie, prerozdelenie a spájanie vrcholov. Na obr. 2.7b je detailnejšie rozkreslené rozdelenie 4-vrcholu na 3 vrcholy. (Operácia proti smeru šípky by bolo spojenie).



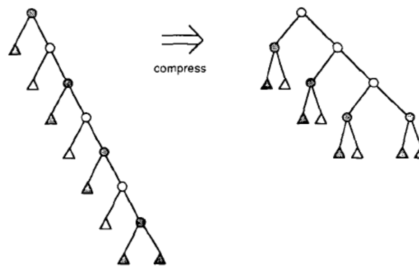
Obr. 2.7: a) Rozdelenie, prerozdelenie a spojenie vrcholu b) Rozdelenie vrcholu a, b, c

§ Prestavanie stromu. Tretí spôsob je prestavanie stromu. Môže to byť úplné prestavanie, alebo len čiastočné, kedy prestaviame iba nejaký vybraný podstrom. Pod prestavaním máme na mysli, že daný podstrom prerobíme na optimálny alebo dokonale vyvážený podstrom.

Toto sa dá jednoducho v lineárnom čase a pamäti: Najskôr do pomocného poľa vypíšeme všetky prvky v inorderi, t.j. utriedené (pritom vrcholy vymažeme). Následne pole prvkov spracúvame rekurzívne: vytvoríme strom z ľavej a pravej polovice, stredný prvok na záver pripojíme ako koreň.

Day, Stout a Warren [SW86] však navrhli spôsob, akým to urobiť použitím iba konštantnej pamäte pomocou $O(n)$ rotácií.

Jemne načrtneme ich myšlienku: Najskôr transformujeme pravými rotáciami strom na pravú reťaz, následne opakovane každý druhý vrchol na reťazi otáčame doľava. Ak totiž máme dva po sebe idúce vrcholy v reťazi, ktorých ľavý podstrom je úplný o $2^n - 1$ vrcholoch (v pravej reťazi máme $n = 0$), ľavou rotáciou na druhom vrchole dostaneme jeden vrchol na reťazi, ktorého ľavý podstrom je úplný strom o $2^{n+1} - 1$ vrcholoch. Keď teda prejdeme celú reťaz a každý druhý prvok zrotujeme doľava, dostaneme reťaz polovičnej dĺžky s dvojnásobne veľkými úplnymi stromami vľavo. Takémuto prechodu hovoríme kompresia (pozri obr. 2.8). Na obrázku stačí spraviť 3 ľavé rotácie v „svetlých“ vrcholoch a dostaneme strom vpravo. Tam by sme spravili ešte jednu ľavú rotáciu pravého podstromu a mali by sme optimálny strom.



Obr. 2.8: Kompresia v algoritme DSW

Kapitola 3

Vyvážené vyhľadávacie stromy

Action is at bottom a swinging and flailing of the arms
to regain one's balance and keep afloat.
Eric Hoffer

3.1 AVL-strom

§ **Definícia.** AVL-strom je historicky prvým stromom, ktorý garantuje logaritmickú výšku. Názov „AVL" sú iniciály autorov: G. M. Adelson-Velskij, E. M. Landis. Definujme $bal(v) = h(right(v)) - h(left(v))$. Budeme hovoriť, že strom je AVL-stromom, keď pre každý vrchol $|\,bal(v)| \leq 1$, čiže $bal(v) \in \{-1, 0, +1\}$. Inými slovami, výška pravého a ľavého podstromu každého vrcholu sa líši najviac o 1. AVL-strom sa dá implementovať tak, že ku každému vrcholu pridáme 2 bity, kam vložíme $bal(v)$, jednoduchšie sa však programuje, ak si pre každý vrchol udržujeme hodnotu $h(v)$.

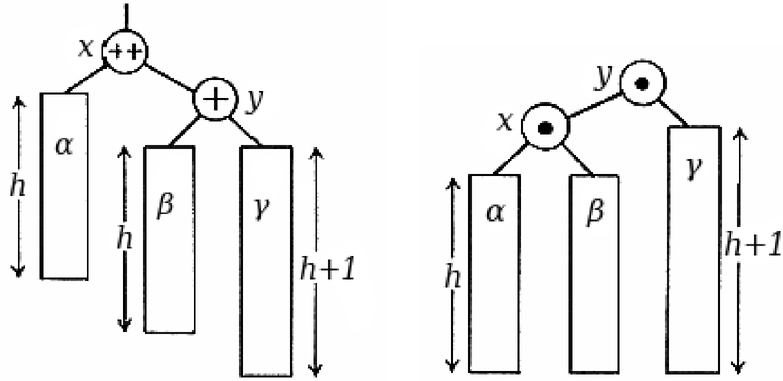
§ **Výška AVL-stromu.** Čo nám zabezpečí takáto definícia? Nech T_h je AVL-strom výšky h, ktorý má najmenší možný počet vrcholov. Potom jeden zo synov má výšku $h-1$ a druhý $h-1$ alebo $h-2$. Keďže chceme minimalizovať $size(T_h)$, môžeme predpokladať, že ľavý podstrom je T_{h-1} a pravý T_{h-2}. Zjavne T_0 je prázdny strom a T_1 tvorí jeden vrchol. Takto rekurzívne definované stromy majú názov *Fibonacciho stromy* (pre zjavnú podobnosť s Fibonacciho číslami). Riešením rekurencie dostaneme, že $n \geq F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2$. Teda výška AVL-stromu je vždy medzi $\lg n + 1$ a $1.4404 \lg(n+2) - 0.328$.

§ **Vyvažovanie.** Algoritmy AVL-stromu a vôbec algoritmy takmer všetkých vyvažovaných stromov najskôr vložia, prípadne vyberú prvok zo stromu tak, ako sme to robili v BST. Tým sa môže porušiť podmienka vyváženia a strom treba vyvažovať.

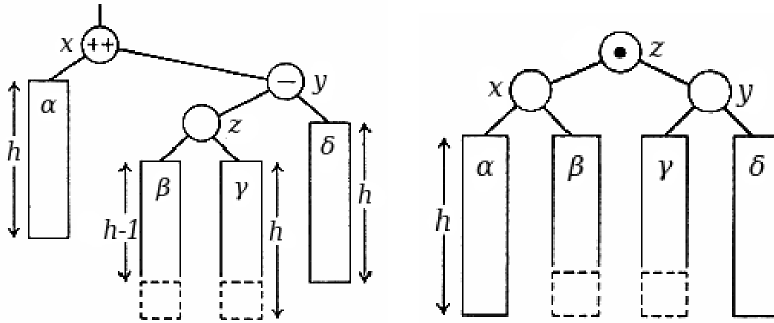
Po vložení nového vrcholu prechádzame tú istú cestu ešte raz, smerom od nového vrcholu ku koreňu. Na tejto ceste opravujeme výšky vrcholov tak, aby platilo: $h(x) = \max(h(left(x), h(right(x)) + 1$. Problém nastane vtedy, keď narazíme na vrchol s $bal(x) = +1$, pričom sme vrchol vložili do jeho pravého podstromu a rozdiel výšok tak narástol na $+2$ (podobne ak vrcholu s $bal(x) = -1$ narástla výška ľavého podstromu). Nie je ťažké presvedčiť sa, že (ak nerátame 2 symetrické prípady s $bal(x) = -1$) existujú iba dva prípady, ktoré podtrebujeme špeciálne riešiť (nech $y = right(x)$ a $z = left(y)$):

- $bal(x) = +2$, $bal(y) = 0$ alebo 1 (obr. 3.1) a
- $bal(x) = +2$, $bal(y) = -1$ (obr. 3.2).

Prvý prípad riešime rotáciou x, druhý prípad dvojitou rotáciou z (raz doprava, raz doľava). Prípad $bal(x) = -1$ je symetrický.



Obr. 3.1: Prípad 1



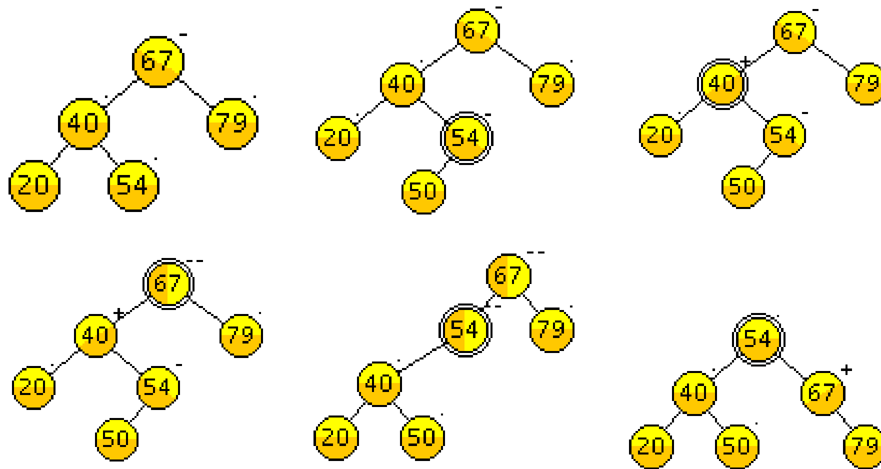
Obr. 3.2: Prípad 2

Rovnaký postup je pri vymazávaní: vymažeme vrchol a následne sledujeme cestu k vrcholu, upravujeme výšky a v prípade potreby rotujeme. Zjavne pri vkladaní, či vymazávaní maximálne dvakrát prejdeme cestu medzi koreňom a listom a už sme dokázali, že tá má aj v najhoršom prípade logaritmickú dĺžku. Všetky slovníkové operácie teda vieme vykonať v najhoršom prípade v čase $O(\log n)$. Na vyváženie po vkladaní stačia 2 rotácie.

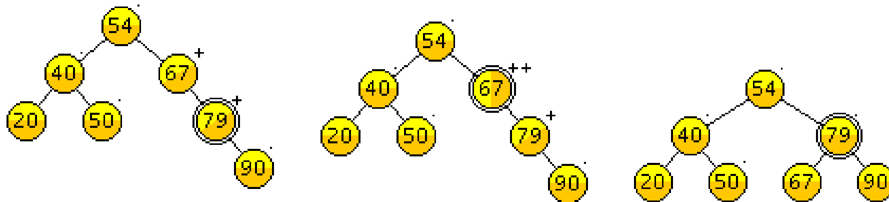
§ **Spájanie.** Clark A. Crane [Knu78] dokonca vymyslel efektívne algoritmy pre operácie $join(T_1, T_2)$, kde všetky prvky T_1 sú menšie ako T_2 a $split(T, k)$. Tu si ukážeme *join*, *split* sa dá implementovať tiež v čase $O(\log n)$ pomocou série *join*-ov. Bez ujmy na všeobecnosti predpokladajme, že $h(T_1) \geq h(T_2)$. Vymažeme najmenší vrchol J z T_2. Nech T_2' je T_2 bez vrcholu J. Nasledujme teraz pravé smerníky T_1, kým nenatrafíme na vrchol P taký, že $h(P) - h(T_2') = 0$ alebo 1; vždy keď ideme o 1 doprava, výška sa zmenší o 1 alebo o 2, takže také P vždy existuje. Teraz podstrom P nahradíme koreňom J, ktorého ľavý podstrom je P a pravý podstrom je T_2' a pokračujeme, ako keby sme práve J pridali do stromu (cestujeme nahor, upravujeme výšky, prípadne rotujeme). Všimnime si, že nahradením nám výška podstromu vzrástla iba o jedna, takže nemáme žiadny problém.

3.2 2-3-strom

§ **Definícia.** Iný druh stromu, tzv. *2-3-strom* vymyslel Hopcroft (nepublikoval). V 2-3-strome má každý vrchol jeden alebo dva kľúče. Všetky vrcholy okrem listov, ktoré majú jeden kľúč majú



Obr. 3.3: Vyvažovanie AVL-stromu po pridaní kľúča 50



Obr. 3.4: Vyvažovanie AVL-stromu po pridaní kľúča 90

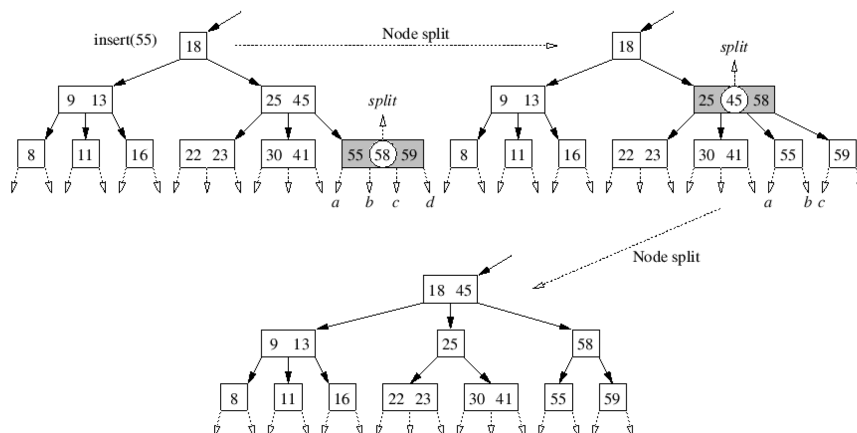
dvoch synov (pričom prvky v ľavom podstrome sú menšie ako prvky v pravom podstrome) a vrcholy, ktoré majú dva kľúče majú troch synov (prvky v ľavom podstrome sú menšie ako prvý kľúč, prvky v pravom podstrome sú väčšie ako druhý kľúč a prvky v strednom podstrome majú hodnoty väčšie ako prvý kľúč, ale menšie ako druhý kľúč). Od 2-3-stromu požadujeme, aby mal všetky listy na tej istej úrovni. Odtiaľ dostávame, že výška stromu je rádovo medzi $\log_3 n$ a $\log_2 n$.

§ **Slovníkové operácie.** Prevedenie slovníkových operácií je konceptuálne veľmi jednoduché, hoci nie najpríjemnejšie na programovanie. Vyhľadávanie je zovšeobecnením vyhľadávania v binárnom strome. Na binárnych vrcholoch sa správame rovnako ako v BST, na ternárnych vrcholoch porovnáme hľadaný kľúč k s prvým aj druhým kľúčom k_1 a k_2. Ak $k < k_1$, pokračujeme ľavým podstromom; ak $k_1 < k < k_2$, pokračujeme stredným podstromom a ak $k_2 < k$, pokračujeme pravým podstromom.

Poznamenajme, že hoci 2-3-strom je väčšinou nižší ako AVL-strom, počet porovnaní pri hľadaní v 2-3-strome nemusí byť menší: Najhorší prípad je asi keď do 2-3-stromu pridávame postupne rastúcu postupnosť čísel. Ak do 2-3-stromu vložíme čísla $1, 2, \ldots, 2^h - 1$ bude mať 2-3-strom výšku h a každý vrchol bude binárny. Ak pridáme ďalších $2^h - 1$ čísel, budú najpravejšie vrcholy na každej úrovni ternárne. Ak budeme vyhľadávať ∞, pôjdeme práve po tejto najpravejšej ceste, takže v každom vrchole urobíme dve porovnania. Existuje teda 2-3-strom, ktorý obsahuje $2^{h+1} - 2$ kľúčov a potrebuje na vyhľadanie $2h$ porovnaní (v optimálnom prípade stačí $h + 1$).

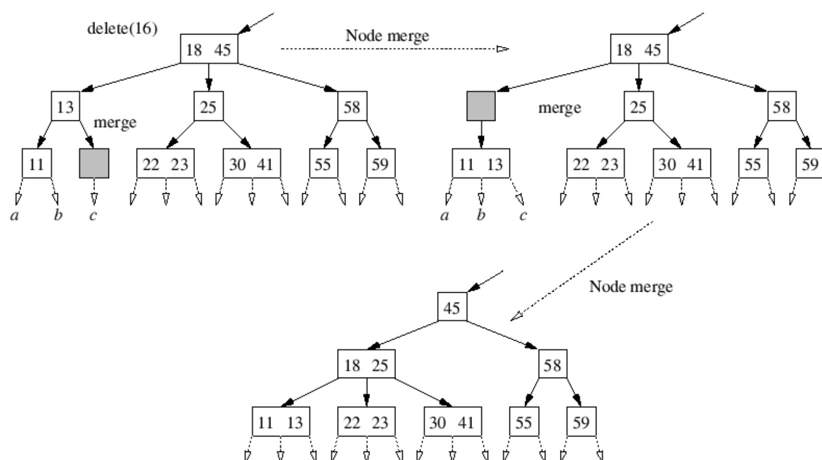
Pri vkladaní vložíme nový kľúč do listu. Môže sa nám stať, že list už obsahoval 2 kľúče, teraz je teda preplnený. Vrchol s tromi kľúčmi rozdelíme na tri vrcholy s jedným kľúčom (spojené v tvare „čerešničiek“) a stredný vrchol vložíme do rodičovského vrcholu. Môže sa nám stať, že sa preplní aj ten – vtedy zase vrchol rozdelíme na tri a stredný vložíme o úroveň vyššie (pozri obr. 3.5).

Takto pokračujeme až ku koreňu. Jediný spôsob, ako 2-3-strom rastie je, že sa preplní aj koreň a vznikne tak nový koreň o úroveň vyššie.



Obr. 3.5: Úprava 2-3-stromu po pridaní kľúča 55

Vymazávanie je o čosi zložitejšie: Vymazávame tak, ako v BST: ak vrchol nie je list, kľúč nahradíme jeho nasledníkom (ktorý je list). Ak boli pred vymazaním v liste dva kľúče, môžeme skončiť. Ak však bol iba jeden, list je teraz prázdny. Sú dve možnosti: ak má brat dva kľúče, môžeme si od neho jeden požičať (preusporiadame kľúče tohto vrcholu, jeho brata a ich otca). Ak má brat iba jeden kľúč, vyberieme z ich spoločného otca jeden kľúč a spojíme ich. Takto sa nám problém posunie o úroveň vyššie (pozri obr. 3.6). 2-3-stromu sa zmenší výška, len keď sa vymaže až vrchol.



Obr. 3.6: Úprava 2-3-stromu po vymazaní kľúča 16

3.3 B-stromy

§ **Definícia.** B-stromy objavili v roku 1970 Bayer a McCreight (nazávisle na nich aj Kaufman). Prezentácia B-stromov je z [Knu78]. Viac-cestný strom nazývame *B-strom rádu m*, ak

- Každý vrchol má $\leq m$ synov.
- Každý vrchol okrem koreňa má aspoň $\geq m/2$ synov.
- Koreň má aspoň dvoch synov.
- Všetky externé vrcholy sú na rovnakej úrovni a nenesú žiadnu informáciu.
- Každý vrchol, ktorý nie je list a má k synov, má $k-1$ kľúčov.

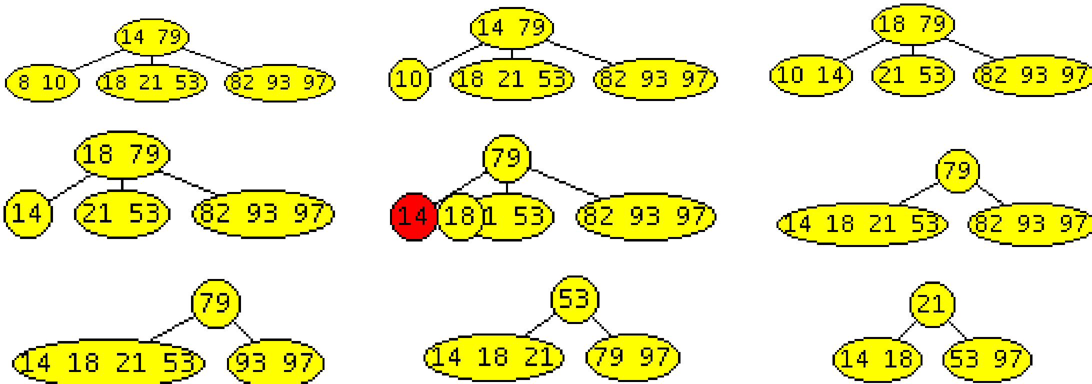
Nás budú zaujímať stromy rádu $m \geq 3$. B-strom rádu 3 je práve Hopcroftov 2-3-strom. B-strom rádu 4 sa volá tiež *2-3-4-strom*.

Vrchol, ktorý obsahuje j kľúčov a $j+1$ smerníkov môžeme reprezentovať postupnosťou $p_0, k_1, p_1, k_2, p_2, \ldots, p_{j-1}, k_j, p_j$, kde $k_1 < k_2 < \cdots < k_j$ sú kľúče a p_i je smerník na podstrom, ktorého všetky kľúče sú medzi k_1 a k_2.

§ **Slovníkové operácie.** Slovníkové operácie budú zovšeobecnením tých, ktoré sme popísali pre 2-3-strom. Pri hľadaní v každom vrchole nájdeme buď $k_i = k$ a končíme, alebo také i, že $k_i < k < k_{i+1}$ (pre jednoduchosť predpokladajme, že $k_0 = -\infty$ a $k_{j+1} = \infty$). Vtedy treba zvoliť podstrom p_i a pokračovať v hľadaní v ňom.

Pri vkladaní, tak ako v 2-3-strome, vložíme nový kľúč do vhodného listu. Ak sa stane, že sa list preplní, t.j. bude obsahovať m kľúčov, rozdelíme ho na dva: $p_0, k_1, p_1, \ldots, k_{\lceil m/2 \rceil -1}, p_{\lceil m/2 \rceil -1}$ a $p_{\lceil m/2 \rceil}, k_{\lceil m/2 \rceil +1}, \ldots, p_m$ a kľúč $k_{\lceil m/2 \rceil}$ vložíme do vyššej úrovne. Tento proces buď skončí na ceste ku koreňu, alebo, ak sú všetky vrcholy na ceste ku koreňu preplnené, vytvorí sa nový koreň.

Pri vymazávaní kľúč vo vrchole, ktorý nie je listom nahradíme jeho následníkom. Z listu kľúč jednoducho odstránime. Ak sa stane, že nám ostane menej ako $m/2$ kľúčov, buď existuje brat, ktorý má viac ako $\lceil m/2 \rceil$ a prerozdelením kľúčov problém vyriešime. Ak sú obaja bratia (v prípade krajných vrcholov, iba jeden) „malý“, t.j. obsahujú práve $\lceil m/2 \rceil$ kľúčov, odstránime z rodiča kľúč, ktorý je medzi nimi a tieto kľúče spojíme do jedného vrcholu s m kľúčmi. Môže sa stať že sa takto stane otec príliš malým, potom pokračujeme v riešení tejto situácie. V najhoršom prípade sa problém dostane až ku koreňu a buď koreňu ostane aspoň jeden kľúč, alebo výška stromu klesne.



Obr. 3.7: Z B-stromu postupne vymazávame čísla 8, 10, 82, 93, 79

§ **Časová zložitosť.** Predpokladajme, že ℓ je počet úrovní a strom má $n+1$ externých vrcholov v hĺbke ℓ. Potom, keďže koreň má aspoň 2 a ostatné vrcholy aspoň $\lceil m/2 \rceil$ detí, počet vrcholov na úrovniach $1, 2, 3, \ldots$ je aspoň $2, 2\lceil m/2 \rceil, 2\lceil m/2 \rceil^2, \ldots$, teda $n+1 \geq 2\lceil m/2 \rceil^{\ell-1}$ a výška stromu je

$$\ell \geq 1 + \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right).$$

Časová zložitosť všetkých operácií je $O(m \log_m n)$, čo je $O(\log n)$ pre malé konštantné m – napríklad 2-3-, 2-3-4-stromy. B-stromy však boli vyvýjané s úplne inou motiváciou: Pre externé vyhľadávanie. Pevný disk je oveľa pomalší ako operačná pamäť a funguje takým spôsobom, že prečítanie bloku údajov trvá približne rovnako ako prečítanie konkrétneho kľúča v bloku. Preto je snaha mať m dostatočne veľké, aby bol počet prístupov na externý disk malý.

§ **Variácie.** Existuje veľa dátových štruktúr príbuzných B-stromom. B^+-strom je B-strom, kde všetky dáta sú v listoch, vnútorné vrcholy slúžia iba na orientáciu v strome. To je výhodné z dvoch dôvodov: po prvé vnútorné vrcholy môžu obsahovať viac kľúčov (máme väčšie vetvenie) a po druhé kľúče chceme často prechádzať aj sekvenčne; to ide ľahko, ak ich pospájame do zoznamu a to sa robí ľahko, ak sú dáta iba v listoch.

Druhou dôležitou variáciou je B^*-strom. Zatiaľ čo B-strom môže byť len do polovice plný, v B^*-strome kľúče prerozdeľujeme tak, aby bol každý vrchol zaplnený aspoň z dvoch tretín.

Ďalším zovšeobecnením B-stromov sú (a, b)-stromy, kde je počet synov ohraničený konštantami a a b.

Dôležitá obmena je aplikovať *split* pri vkladaní (resp. *join* pri vymazávaní) už pri ceste zhora nadol, keď je vrchol takmer plný (takmer prázdny) – takto si vieme ušetriť cestu nahor.

3.4 Červeno-čierny strom

§ **Definícia.** S nápadom reprezentovať 2-3-4-stromy pomocou binárnych stromov prišiel Bayer [Bay72], uviedol tzv. *SBB-stromy (symmetric binary B-tree)*. Červeno-čierne prevedenie dali SBB-stromom Guibas a Sedgewick článkom [GS78]. Vymazávanie pomocou iba konštantného počtu rotácií vymyslel Tarjan [Tar83b].

Binárny vyhľadávací strom je *červeno-čierny*, ak

- Každý vrchol je buď červený alebo čierny.
- Každý externý vrchol je čierny.
- Ak je vrchol červený, obaja jeho synovia sú čierny.
- Všetky cesty od koreňa k listom obsahujú rovnaký počet čiernych vrcholov.

Ako súvisia červeno-čierne stromy s 2-3-4-stromami? Predstavme si, že čierny vrchol spolu s tými deťmi, ktoré sú červené tvorí jeden „pseudovrchol“. Čierny vrchol je koreň pseudovrcholu. Keďže každý červený vrchol musí mať čiernych synov, sú tri možnosti:

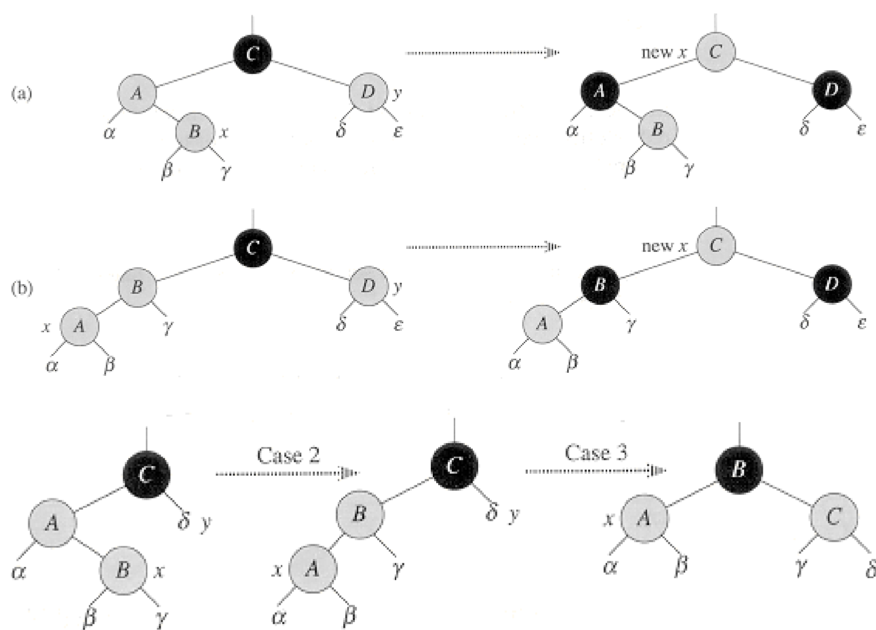
- pseudovrchol sa skladá z jedneho čierneho vrcholu (ktorý má čierne deti) – predstavuje (binárny) 2-vrchol
- čierny vrchol má jedno čierne a jedno červené dieťa – predstavuje 3-vrchol
- čierny vrchol má obe deti červené – predstavuje 4-vrchol

Keďže všetky cesty od koreňa k listom obsahujú rovnaký počet čiernych vrcholov, ak by sme zostrojili strom z pseudovrcholov, boli by všetky listy na jednej úrovni, dostali by sme 2-3-4-strom.

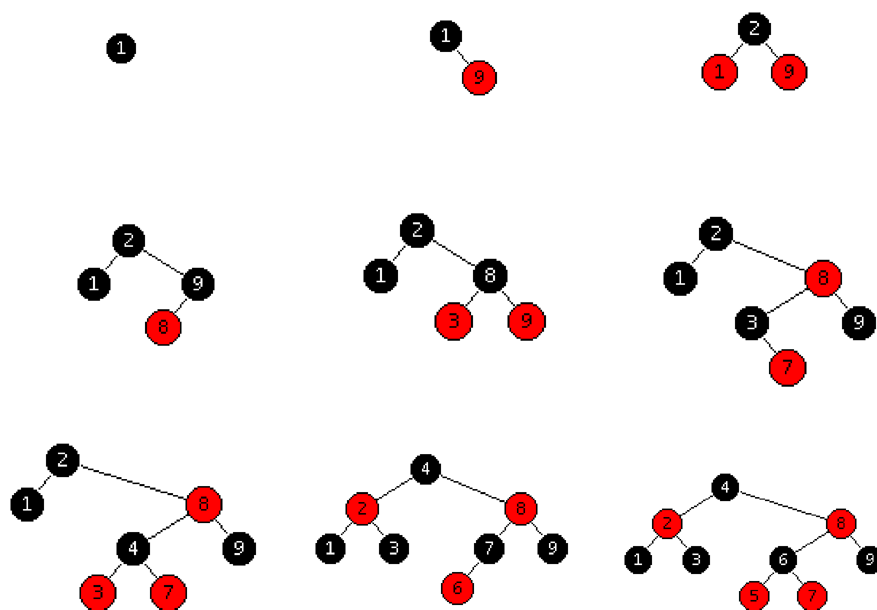
§ **Slovníkové operácie.** Do červeno-čierneho stromu vkladáme rovnako ako do BST. Vložený vrchol zafarbíme na červeno. Ak sa poruší vlastnosť červeno-čierneho stromu (vrchol má červeného otca), aplikujú sa transformácie. Podobne je to pri vymazávaní.

Na obr. 3.8 sú transformácie pri vkladaní. Svetlo-šedý vrchol je červený. Korene všetkých podstromov sú čierne. Všimnime si, že v prvom a druhom prípade máme pseudovrchol zo štyroch vrcholov; v interpretácií 2-3-4-stromov máme teda vrchol preplnený. Prefarbením dostávame jeden 3-vrchol a jeden 2-vrchol a jeden červený vrchol sa dostal do vyššieho pseudovrcholu. V treťom riadku máme 4-vrchol, akurát zle usporiadaný, čo vyriešime dvoma rotáciami.

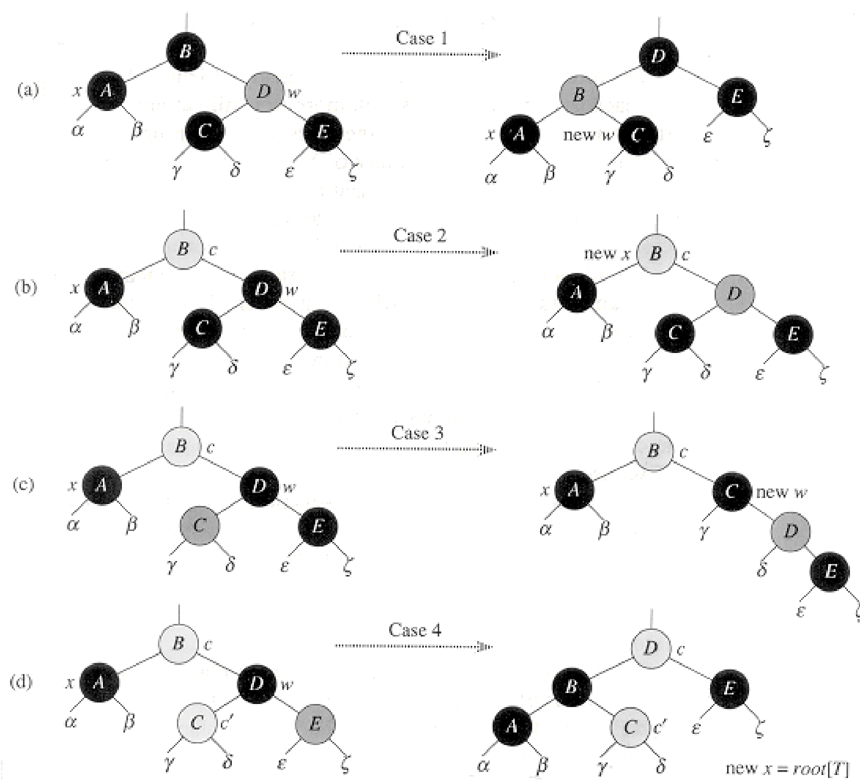
Na obr. 3.10 sú transformácie pri vymazávaní. Tmavo-šedé vrcholy sú červené, svetlo-šedé majú ľubovoľnú (červenú alebo čiernu) farbu.



Obr. 3.8: Transformácie po vložení prvku do červeno-čierneho stromu



Obr. 3.9: Do červeno-čierneho stromu vkladáme kľúče $1, 9, 2, 8, 3, 7, 4, 6, 5$



Obr. 3.10: Transformácie po vymazaní prvku z červeno-čierneho stromu

3.5 AA-strom

Zaujímavý strom vymyslel v roku 1993 Arne Andersson [And93] (odtiaľ iniciály AA v mene stromu). Jeho hlavnou motiváciou bolo zjednodušiť červeno-čierne stromy, respektíve vymyslieť strom, ktorý by mal na jednej strane garantovanú zložitosť operácií $O(\log n)$, na druhej strane bol jednoduchší ako väčšina vyvážených stromov.

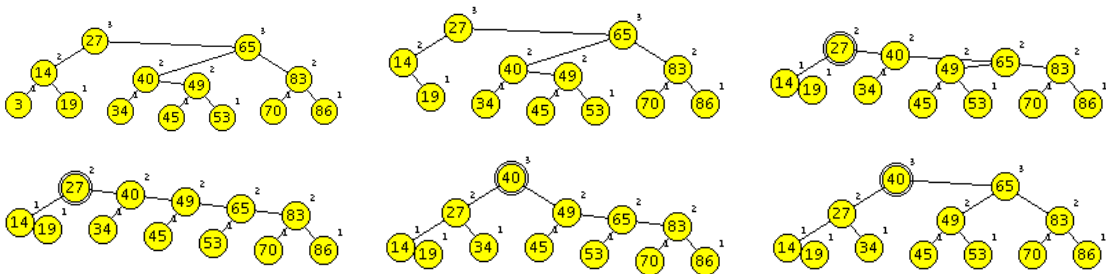
Andersson nevyšiel z 2-3-4-stromov, na ktorých sú založené červeno-čierne stromy, ale z 2-3-stromov. Pseudovrchol pozostáva z jedného alebo dvoch vrcholov. Hrany spájajúce vrcholy pseudovrcholu voláme horizontálne, zvyšné sú vertikálne. Každý vrchol si pamätá svoj *rank*, ktorý predstavuje vertikálnu výšku vrcholu.

Na 2-3-stromoch bol založený už Bayerov BB-strom (binary B-tree), binárna reprezentácia 2-3-stromu (BB-strom je predchodca SBB-stromu, z ktorého vznikli červeno-čierne stromy). Bayer však nedostal veľmi jednoduché algoritmy, musel rozoberať viacero prípadov. Andersson navrhol takýto spôsob, ako počet prípadov zredukovať: Predtým ako skontrolujeme veľkosť pseudovrcholu sa uistíme, že iba pravé hrany sú horizontálne (tak dokázal 5 prípadov zredukovať iba na dva).

Andersson zaviedol dve základné operácie na AA-stromoch:

- $skew(v)$ – eliminuje ľavú horizontálnu hranu z vrcholu v; ak je ľavá hrana horizontálna, vykonáme pravú rotáciu;
- $split(v)$ – ak je pseudovrchol v príliš veľký, rozdelíme ho tak, že zvýšime *rank* druhého vrcholu a zrotujeme ho doľava.

Jednoduchú implementáciu dostaneme, ak použijeme techniku zarážky (takto nemusíme kontrolovať, či daný vrchol skutočne existuje). Nový vrchol vkladáme tak, že na vhodné miesto (ako pri BST) pripojíme vrchol s *rank*-om 1. Následne sa vraciame smerom ku koreňu a na každom vrchole v vykonáme $skew(v)$ a $split(v)$. Pri vymazávaní vymažeme vrchol tak, ako pri BST a na ceste ku koreňu v každom vrchole v: Ak chýba pseudovrchol pod v (jedno z detí má o 2 menší *rank*), znížime *rank* o 1. Ak pritom pravé dieťa patrilo do toho istého pseudovrcholu, znížime jeho *rank* tiež. Zavoláme $skew(v)$, $skew(right(v))$ a $skew(right(right(v)))$, aby sme sa uistili, že pseudovrchol má všetky horizontálne hrany doprava. Následne zavoláme $split(v)$ a $split(right(v))$.



Obr. 3.11: Z AA-stromu vymazávame najmenší prvok

Kapitola 4

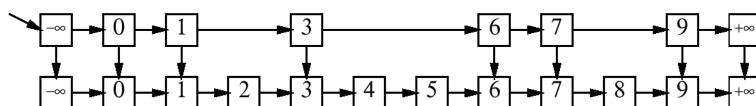
Znáhodnené vyhľadávacie stromy

Without the capacity to provide its own information,
the mind drifts into randomness.
Mihaly Csikszentmihalyi

4.1 Skip list

§ **Definícia.** *Skip list* je dátová štruktúra s očakávanou logaritmickou zložitosťou, s ktorou v roku 1988 prišiel Pugh. Skip list vlastne ani nie je binárnym stromom, ale, ako hovorí Pughov článok [Pug90], skôr alternatívou k vyváženým stromom.

Hlavná myšlienka skip listu je zobrať obyčajný utriedený spájaný zoznam a trochu ho vylepšiť. Problém so spájaným zoznamom je, že hoci je utriedený, nevieme v ňom efektívne vyhľadávať (napríklad binárnym vyhľadávaním), pretože k vrcholom vieme pristupovať iba sekvenčne. Vytvorme teraz nový zoznam s približne polovicou vrcholov, pomocou ktorého sa dostaneme približne do každého druhého vrcholu pôvodného zoznamu (pozri obr. 4.1). Takto nám na prístup k ľubovoľnému vrcholu bude stačiť približne $\lceil n/2 \rceil + 1$ krokov. Ak pridáme ďalší takýto zoznam, ktorý bude mať približne polovicu vrcholov kratšieho zoznamu a budeme ním zrýchlene pristupovať k tomuto druhému zoznamu, bude nám stačiť približne $\lceil n/4 \rceil + 2$ krokov. Ak by sme takto pokračovali ďalej, môžeme vytvoriť približne $\lg n$ zoznamov, každý asi dvakrát kratší ako predošlý.

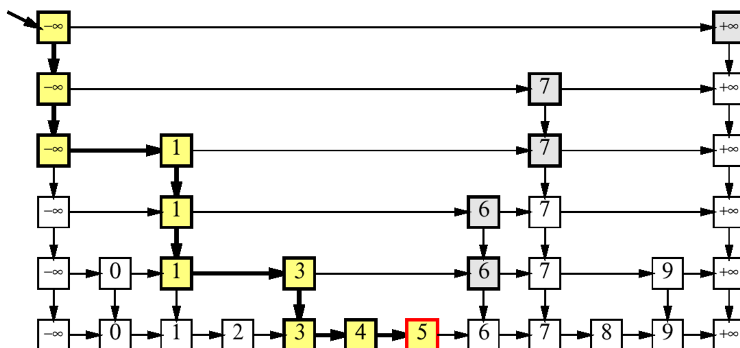


Obr. 4.1: Vylepšenie spájaného zoznamu

Zámerne používame slovo približne. Ideálne by bolo, keby nás prvý smerník hodil presne do polovice a smerníky ďalšieho zoznamu do príslušnej štvrtiny, atď. Takáto štruktúra by sa však strašne ťažko udržiavala. My necháme na náhodu, aby nám vytvorila dostatočne dobrý skip list.

§ **Slovníkové operácie.** Pri vyhľadávaní začneme na poslednej úrovni (hore). Kým je nasledujúci prvok menší alebo rovný hľadanému kľúču, nasledujeme smerníky zoznamu (robíme „veľké skoky“ v pôvodnom zozname). Akonáhle je nasledujúci prvok väčší, prejdeme do nižšej úrovne („zmenšíme krok“; pozri obr. 4.2).

Pri vkladaní najskôr nájdeme v najnižšom zozname miesto, kam prvok vložiť, následne sa rozhodneme, či ho uložíme aj do skráteného zoznamu. V tom má byť približne polovica vrcholov, hodíme si teda mincou a ak padne hlava, postupujeme, ak znak, algoritmus končí. Ak postupujeme, opäť si hodíme mincou a s pravdepodobnosťou 1/2 postupujeme ešte ďalej, s pravdepodobnosťou



Obr. 4.2: Hľadanie 5 v skip liste

1/2 končíme. Takto vieme bez akéhokoľvek prepočítavania a neustáleho upravovania zabezpečiť, aby bolo na i-tej úrovni s vysokou pravdepodobnosťou rádovo $n/2^i$ vrcholov (pôvodný zoznam tvorí nultú úroveň). Pri vymazávaní nájdeme príslušný prvok a vymažeme ho z každej úrovne.

§ **Analýza.** Označme $L(x)$ počet úrovní, ktoré obsahujú prvok x. Keďže prvok postupuje o úroveň vyššie s pravdepodobnosťou 1/2, platí: $E(L(x)) = 1/2{\cdot}0 + 1/2(1+E(L(x)))$, teda $E(L(x)) = 1$. Teda priemerný vrchol postúpi o 1 úroveň. Na to, aby bolo $L(x) \geq k$ potrebujeme k-krát hodiť hlavu. Teda $P(L(x) \geq k) = 2^{-k}$. Špeciálne $P(L(x) \geq \lg n) = 1/n$, $P(L(x) \geq 2\lg n) = 1/n^2$, $P(L(x) \geq 3\lg n) = 1/n^3$, atď. Označme $L = \max_x L(x)$. Pravdepodobnosť, že výška skip listu je aspoň $c\lg n$ sa rovná pravdepodobnosti, že aspoň jeden vrchol má výšku aspoň $c\lg n$. Keďže platí $P(A \vee B) \leq P(A) + P(B)$, platí: $P(L \geq c\lg n) \leq n \cdot P(L(x) \geq c\lg n) = 1/n^{c-1}$. Teda napríklad pravdepodobnosť, že skip list bude mať výšku aspoň dvojnásobok optimálnej výšky je len $1/n$, trojnásobok len $1/n^2$. Skip list bude mať s vysokou pravdepodobnosťou logaritmickú výšku. Dá sa dokázať horný odhad $(2 + 1/(n-1))\lg n$ na očakávanú výšku.

Aká je zložitosť vyhľadávania? Šikovný spôsob, ako situáciu analyzovať je pozrieť sa na ňu spätne v čase: Začnime s nájdeným vrcholom a pozrime sa, ako sme sem prišli. Ak je rovnaký vrchol aj o úroveň vyššie, podľa popisu algoritmu sme pri hľadaní museli dôjsť zhora a navyše na danom mieste nám musela padnúť hlava, aby vrchol postúpil. Na druhej strane, ak rovnaký vrchol o úroveň vyššie už nie je, museli sme prísť zľava a pri vkladaní nám musel padnúť znak. Takto pokračujeme smerom späť v čase, kým sa nedostaneme na poslednú úroveň vľavo (ľavý horný roh). Vieme, že s vysokou pravdepodobnosťou má skip list výšku $O(\log n)$, takže smerom nahor sme sa hýbali $O(\log n)$-krát. Na to potrebujeme hodiť hlavu $O(\log n)$-krát, čo je aj dĺžka cesty, ktorú sme urazili. Hľadanie má teda očakávanú zložitosť $O(\log n)$ rovnako je to s ostatnými operáciami.

Čo sa týka pamäťovej zložitosti, keďže na prvej úrovni je iba zhruba polovica vrcholov, na ďalšej iba zhruba štvrtina, atď., spolu je navyše približne $n/2 + n/4 + n/8 + \cdots = n$ vrcholov.

4.2 Treap

§ **Definícia.** *Treap* je elegantná dátová štruktúra, ktorá vznikne kombináciou vyhľadávacieho stromu a haldy – odtiaľ pochádza aj názov treap = tree + heap. V roku 1980 ju vynašiel Vuillemin pod názvom Karteziánsky strom, pretože každý vrchol mal dve súradnice, x a y [Vui80]. Známa sa však stala až pod menom treap po roku 1989, keď ju študovali Aragon a Seidel [AS89].

Každý vrchol treapu obsahuje dve hodnoty: *kľúč* a *prioritu*. Ak sa pozrieme iba na kľúče, uvidíme vyhľadávací strom, t.j. $key(u) < key(v) < key(w)$, kde u je v ľavom a w v pravom podstrome v. Ak sa pozrieme iba na priority, uvidíme haldu, t.j. $priority(p(v)) > priority(v)$ (kvôli analýze budeme predpokladať, že všetky priority sú rôzne; v praxi stačí, aby boli rovnaké len s malou pravdepodobnosťou). Ak za priority zvolíme klesajúcu postupnosť čísel, dostaneme

obyčajný BST. Ak za priority zvolíme rôzne váhy (pravdepodobnosť alebo frekvenciu výskytu), dostaneme strom, v ktorom budú prvky s väčšou prioritou bližšie ku koreňu a tak ich skôr nájdeme. My budeme uvažovať náhodný treap, kde sa priority volia ako navzájom nezávislé náhodné čísla rovnomerne distribuované na nejakom intervale, dajme tomu $\langle 0, 1)$ (v praxi, samozrejme, postačí generátor pseudonáhodných čísel, môžu sa voliť náhodné čísla z rozsahu $0, \ldots, 2^{16} - 1$).

Na prvý pohľad nie je zrejmé, či existuje treap pre ľubovoľné hodnoty kľúčov a priorít. Ľahko sa ukáže že je tomu tak a ak navyše predpokladáme, že kľúče a priority sú rôzne, existuje práve jeden taký strom. Dôkaz je indukciou: Ak máme jeden prvok, zjavne existuje jedinečný treap. Ak je vrcholov viac, zoberieme ten, ktorý má najväčšiu prioritu. Zjavne tento vrchol musí byť koreň. Koreň nám všetky ostatné prvky rozdelí na väčšie a menšie. Podľa indukčného predpokladu existujú jedinečné treapy pre tieto množiny. Tie tvoria ľavý a pravý podstrom nášho treapu. Poznamenajme, že tento dôkaz tiež ukazuje, že treap s náhodnými prioritami je vlastne náhodne generovaný BST: výsledný treap je rovnaký ako keby sme do BST vkladali prvky v klesajúcom poradí podľa priority. Odtiaľ dostávame očakávanú výšku treapu $O(\log n)$.

§ **Slovníkové operácie.** Ak máme naprogramovanú procedúru $\mathit{rotation}(v)$, ktorá zrotuje vrchol v, sú všetky operácie veľmi jednoduché: Nový vrchol vložíme rovnako ako do BST, následne vrchol rotujeme (bubleme smerom nahor), kým nie je priorita otca väčšia. Vymazávanie je po naprogramovaní rotácií ešte jednoduchšie ako v BST. Dá sa naň pozerať ako na inverznú operáciu k vkladaniu: rovnako ako v halde vrchol najskôr prebubleme nadol, kým nie je list a odstránime ho. Operáciu $\mathit{join}(T, T')$, kde prvky T sú menšie alebo rovné k a prvky T' sú väčšie alebo rovné k vykonáme tak, že vytvoríme vrchol s kľúčom k, ľavý podstrom bude T, pravý T' a následne k vymažeme. Ak chceme treap rozdeliť podľa kľúča k, stačí doň vložiť kľúč k s nekonečnou prioritou – vďaka haldovému usporiadaniu priorít sa dostane do koreňa a ľavý a pravý podstrom budú tvoriť výsledok.

Očakávaná časová zložitosť všetkých týchto operácií je zjavne $O(\log n)$, očakávaný počet rotácií pri vkladaní a vymazávaní je menej ako 2 (presnejšie blíži sa 2 pre veľké n).

Kapitola 5

Amortizované vyhľadávacie stromy

5.1 GB-strom

§ **Definícia.** *Všeobecný vyvážený strom (general balnced tree, GB-tree)*, skrátene GB-strom vynašiel roku 1989 Arne Andersson [And99] a neskôr, v 1993 nezávisle na ňom Igal Galperin a Ronald Rivest [GR93] pod názvom *scapegoat tree*. Pod týmto názvom je strom známy v anglickej literatúre. Avšak pre chýbajúci obstojný slovenský ekvivalent budeme radšej používať názov GB-strom.

GB-strom používa asi to najjednoduchšie kritérium vôbec – strom môže mať akýkoľvek tvar, poženujeme iba, aby mal logaritmickú výšku. T.j. nerobíme nič, kým strom nie je príliš nevyvážený. Vtedy strom čiastočne prestaviame.

Presnejšie, majme danú konštantu $\alpha > 1$. Jediná podmienka, ktorú musí strom spĺňať, aby bol GB(α)-stromom je, aby $h(T) \leq \lceil \alpha \lg |T| \rceil$. Vo vrcholoch nepotrebujeme žiadne pomocné informácie. Stačia nám iba dve *globálne* premenné, ktoré si popíšeme pri vkladaní a vymazávaní.

§ **Vkladanie.** Pri vkladaní najskôr nájdeme miesto, kam nový vrchol vložíme tak, ako v BST. Cestou nadol počítame hĺbku, do ktorej vrchol vkladáme; $|T|$ zvýšime o 1. Ak $h(T) \leq \lceil \alpha \lg |T| \rceil$, strom je korektný GB($\alpha$)-strom a vkladanie sa úspešne končí. Ak sa stane, že $h(T) > \lceil \alpha \lg |T| \rceil$, ideme od vrcholu späť po ceste ku koreňu a zastavíme sa na prvom vrchole, ktorý nie je vyvážený, t.j. na najhlbšom vrchole v takom, že $h(v) > \lceil \alpha \lg |v| \rceil$. Celý podstrom v prestaviame.

Koľko času tento postup potrebuje? Na konci kapitoly 2 sme ukázali, že prestavanie vieme v konštantnej pamäti v čase $O(|v|)$. Navyše, keďže veľkosť vrcholov $|v|$ nepoznáme, pre každý vrchol na ceste nahor ju musíme počítať. Toto vieme tiež v čase $|v|$: dôležité je iba, aby sme nepočítali veľkosť pri každom vrchole odznovu. Ak vieme veľkosť vrcholu w, veľkosť $p(w)$ vypočítame tak, že k už vypočítanému $|w|$ dopočítame veľkosť jeho brata. Takto vieme veľkosti všetkých vrcholov od listu po nájdený vrchol v, ktorý porušuje podmienku GB(α)-stromov, vypočítať v čase $O(|v|)$. Celkovo teda v najhoršom prípade trvá vloženie nového vrcholu $O(n)$.

Ukážme, že procedúra vkladania je korektná: Na vstupe dostaneme korektný GB(α)-strom a vrchol, ktorý doň máme vložiť. Keďže pred vložením platí $h(T) \leq \lceil \alpha \lg |T| \rceil$, jediný spôsob, ako sa môže podmienka porušiť je, že práve novo-vložený vrchol zväčší výšku stromu. Ak je po vložení $h(T) > \lceil \alpha \lg |T| \rceil$, potom určite existuje vrchol na ceste ku koreňu, pre ktorý $h(v) > \lceil \alpha \lg |v| \rceil$ (v najhoršom prípade je to koreň). Vieme tiež, že po prestavaní bude mať podstrom výšku $\lceil \lg |v| \rceil$ (kde $|v|$ je veľkosť podstromu s koreňom v pred prestavaním). Výška sa teda zmenší aspoň o 1 a teda bude nanajvýš taká ako pred vložením. Keďže vieme, že pred pridaním vrcholu bol strom korektný, bude aj po prestavaní.

§ **Vymazávanie.** Na vymazávanie použijeme metódu „lenivého“ vymazávania *(lazy deletion)* a úplného prestavania. Lenivé preto, lebo keď vrchol nájdeme, nevymažeme ho, iba ho označíme na vymazanie (robotu presunieme na neskôr). To bude vyžadovať malú modifikáciu nášho algoritmu na vyhľadávanie: naďalej sa budeme riadiť porovnaniami vo vrcholoch (či sú označené alebo nie), avšak hľadanie skončí úspešne iba vtedy, keď kľúč nájdeme a daný vrchol nie je označený na vymazanie. Jediný problém tejto metódy je, že čas na vyhľadávanie a vkladanie teraz nie je funkciou počtu kľúčov v strome, ale počtu všetkých vrcholov – „naozajstných“, plus tých, ktoré sú označené na vymazanie. Aby sa nám nestalo, že budeme vyhľadávať v strome, v ktorom nie sú žiadne kľúče a všetky vrcholy sú označené na vymazanie, budeme si počítať počet vrcholov označených na vymazanie. Ak ten dosiahne polovicu všetkých vrcholov, celý strom od základu prestaviame (vo všeobecnosti môžeme zobrať ľubovoľnú konštantu $0 < d < 1$ a prestavať strom, ak počet vymazaných vrcholov presiahne $d \cdot |T|$). Takto bude v strome najviac dvakrát (resp. $1/d$-krát) toľko vrcholov ako je treba a vyhľadávanie bude logaritmické: $\lg 2n = \lg n + 1$, teda budeme potrebovať iba jedno porovnanie navyše.

§ **Amortizovaná zložitosť.** Takto navrhnuté operácie majú časovú zložitosť v najhoršom prípade $O(n)$. Ukážeme však, že veľké prestavania sa nedejú často. Vymazávanie trvá $\Theta(n)$, ale iba vtedy, keď celý strom prestaviame. Väčšina vymazávaní trvá iba $O(\log n)$. Strom od základov prestaviame iba vtedy, keď od posledného kompletného prestavania vymažeme aspoň $\Omega(n)$ vrcholov. Amortizovaná zložitosť úplného prestavania je teda $O(1)$. Inými slovami, keby sme si pri každom vymazaní usporili jednu korunu, pri prestavaní sme museli vymazať aspoň polovicu vrcholov, teda máme $\Theta(n)$ korún a vieme „zaplatiť“ prestavanie.

Amortizovaná zložitosť vkladania je tiež $O(\log n)$, i keď dôkaz je trochu komplikovanejší: Definujme rozdiel veľkostí synov v ako $\delta(v) = \max(0, ||\,\mathit{left}(v)| - |\,\mathit{right}(v)|| - 1)$. Alebo jednoduchšie, ak v_1 je väčší a v_2 menší syn v, potom $\delta(v) = \max(0, |v_1| - |v_2| - 1)$. Hodnotu $\delta(v)$ budeme volať *nevyváženosť* vo vrchole v. Indukciou sa dá ľahko ukázať, že dokonale vyvážené stromy sú práve tie stromy, pre ktoré je $\delta(v) = 0$ pre každý vrchol. Na druhej strane, pri vložení jedného vrcholu sa $\delta(v)$ zmení najviac o 1. Ukážeme, že tesne predtým, ako prestaviame podstrom v bude platiť, že $\delta(v) = \Omega(|v|)$: Keďže chceme prestavať podstrom v, musí platiť $h(v) > \lceil \alpha \lg |v| \rceil$ a pre oboch synov v_1 a v_2 musí platiť $h(v_i) \leq \lceil \alpha \lg |v_i| \rceil$. Bez ujmy na všeobecnosti, nech $|v_1| > |v_2|$. Potom platí

$$\lceil \alpha \lg |v| \rceil < h(v) \leq h(v_1) + 1 \leq \lceil \alpha \lg |v_1| \rceil + 1,$$

odkiaľ $|v_1| > 2^{-1/\alpha}|v|$. Navyše z definície $|v| = |v_1| + |v_2|$. Dostávame

$$\delta(v) \geq |v_1| - (|v| - |v_1|) - 1 = 2|v_1| - |v| - 1 > (2^{1-1/\alpha} - 1)|v| - 1.$$

Keďže $\alpha > 1$, výraz v zátvorke je kladná konštanta. Dokázali sme teda, že $\delta(v) = \Omega(|v|)$. Keďže pri každom vkladaní sa hodnota $\delta(v)$ môže zmeniť iba o 1, predtým ako prestaviame podstrom v, musíme do tohto podstromu vložiť rádovo $|v|$ vrcholov a teda amortizovaná zložitosť čiastočného prestavania je $O(1)$. Inými slovami, keby si každý vložený vrchol ušetril jednu korunu na prestavbu časti stromu, do ktorej patrí, potom skôr ako sa strom stane nevyváženým ušetríme dosť korún, aby sme vedeli zaplatiť jeho prestavanie.

5.2 Splay strom

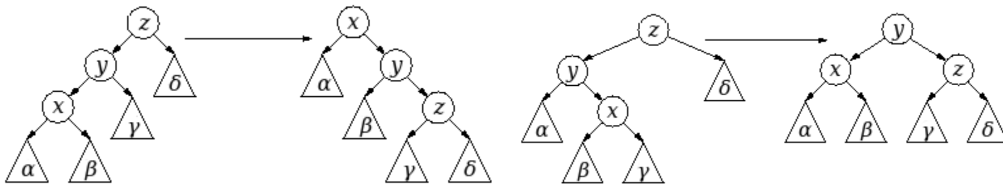
§ **Samočinné vyvažovanie.** *Splay strom (splay tree)* je dátová štruktúra podporujúca slovníkové operácie v amortizovanom logaritmickom čase, ktorú vynašli Sleator a Tarjan v 1983 [ST83, Tar83a]. Hlavnou myšlienkou je pri každom prístupe k vrcholu vykonať nejakú operáciu tak, aby sa strom „sám vyvažoval“ (odtiaľ názov *self-adjusting binary trees*). Myšlienka pochádza od obyčajných spájaných zoznamov, kde sa používa heuristika *move-to-front* („prelož na začiatok“). Ak totiž distribúcia požiadaviek nie je uniformná, vrcholy, ku ktorým pristupujeme častejšie majú tendenciu nachádzať sa pri začiatku a teda vieme ich rýchlejšie nájsť. Tento postup sa dá aplikovať

aj na binárne stromy. Pred Sleatorom a Tarjanom sa tým zaoberali Allen a Munro [AM78] a Bitner [Bit79], ale žiadna z ich metód nedosahovala amortizovanú zložitosť $O(\log n)$. Tú sa podarilo dosiahnuť až spomínaným dvom autorom vďaka operácií *splay*.

§ **Operácia *splay*.** Operácia $splay(S, k)$ dostane ako argument kľúč k a reorganizuje vyhľadávací strom S tak, že v koreni bude vrchol s kľúčom k. Ak sa k v strome nenachádza, bude v koreni vrchol s najmenším väčším alebo najväčším menším kľúčom. Ako taký kľúč nájdeme? Rovnako ako pri jednoduchom vyhľadávaní sa riadime kľúčmi, kým nenarazíme na Λ. Posledný interný vrchol na tejto ceste je hľadaný vrchol.

Aby sme dosiahli logaritmickú zložitosť, nestačí vhodný vrchol nájsť a jednoducho ho „prebublať“ ku koreňu. Bublanie budeme robiť špeciálnym spôsobom:

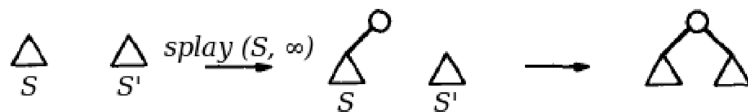
- Ak x má otca, ale nemá starého otca (je synom koreňa), jednoducho zrotujeme x.
- Ak x má otca y a starého otca a aj x aj y sú obaja ľavý alebo obaja pravý synovia svojich otcov, najskôr zrotujeme y, potom x. Tento prípad voláme neformálne „cik-cik“ (obr. 5.1a)
- Ak x má otca y a starého otca, ale jeden z nich je ľavý syn a jeden je pravý syn, potom dvakrát zrotujeme x. Tento prípad voláme neformálne „cik-cak“. (obr. 5.1b)



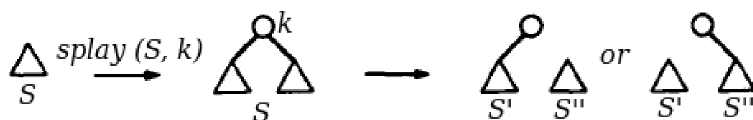
Obr. 5.1: a) Cik-cik b) cik-cak

§ **Slovníkové operácie pomocou *splay*.** Splay strom je veľmi elegantný a ľahko naprogramovateľný strom, pretože všetky slovníkové operácie vieme jednoducho naprogramovať pomocou operácie *splay*:

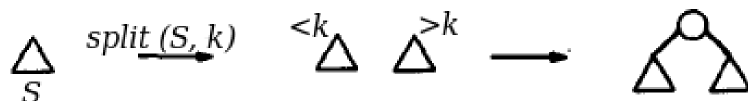
- $find(S, k)$ – vysplayujeme k; ak sa k nachádza v koreni, našli sme ho; ak nie, v strome sa nenachádza;
- $minimum(S, k)$ – vysplayujeme $-\infty$; podľa definície v koreni bude najväčší menší alebo najmenší väčší prvok, keďže však nič menšie ako $-\infty$ nie je, bude tam najmenší prvok;
- $maximum(S, k)$ – vysplayujeme $+\infty$;
- $join(S, S')$ – keď chceme spojiť dva splay stromy do jedného, pričom prvky S sú menšie ako prvky S', stačí v S vysplayovať $+\infty$ – koreň bude maximálny prvok a teda nebude mať pravého syna; S' spravíme novým pravým synom koreňa S (pozri obr. 5.2);
- $split(S, k)$ – ak chceme rozdeliť S na S' a S'' tak, že $x \leq k \leq y$ pre každé x z S' a y z S'', stačí vysplayovať k a zrušiť hranu medzi koreňom a jedným zo synov (podľa toho, či je kľúč v koreni väčší alebo menší ako k) (pozri obr. 5.3);
- $insert(S, k)$ – zavoláme $split(S, k)$, aby sme dostali S' a S''; vytvoríme nový koreň s kľúčom k, pričom S' bude jeho ľavý syn, S'' jeho pravý syn (obr. 5.4);
- $delete(S, k)$ – vysplayujeme k do koreňa a jednoducho ho vymažeme; ľavý a pravý podstrom spojíme pomocou operácie *join* (obr. 5.5).



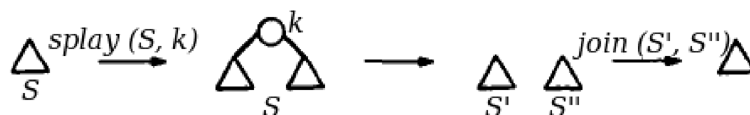
Obr. 5.2: Operácia *join* pomocou *splay*



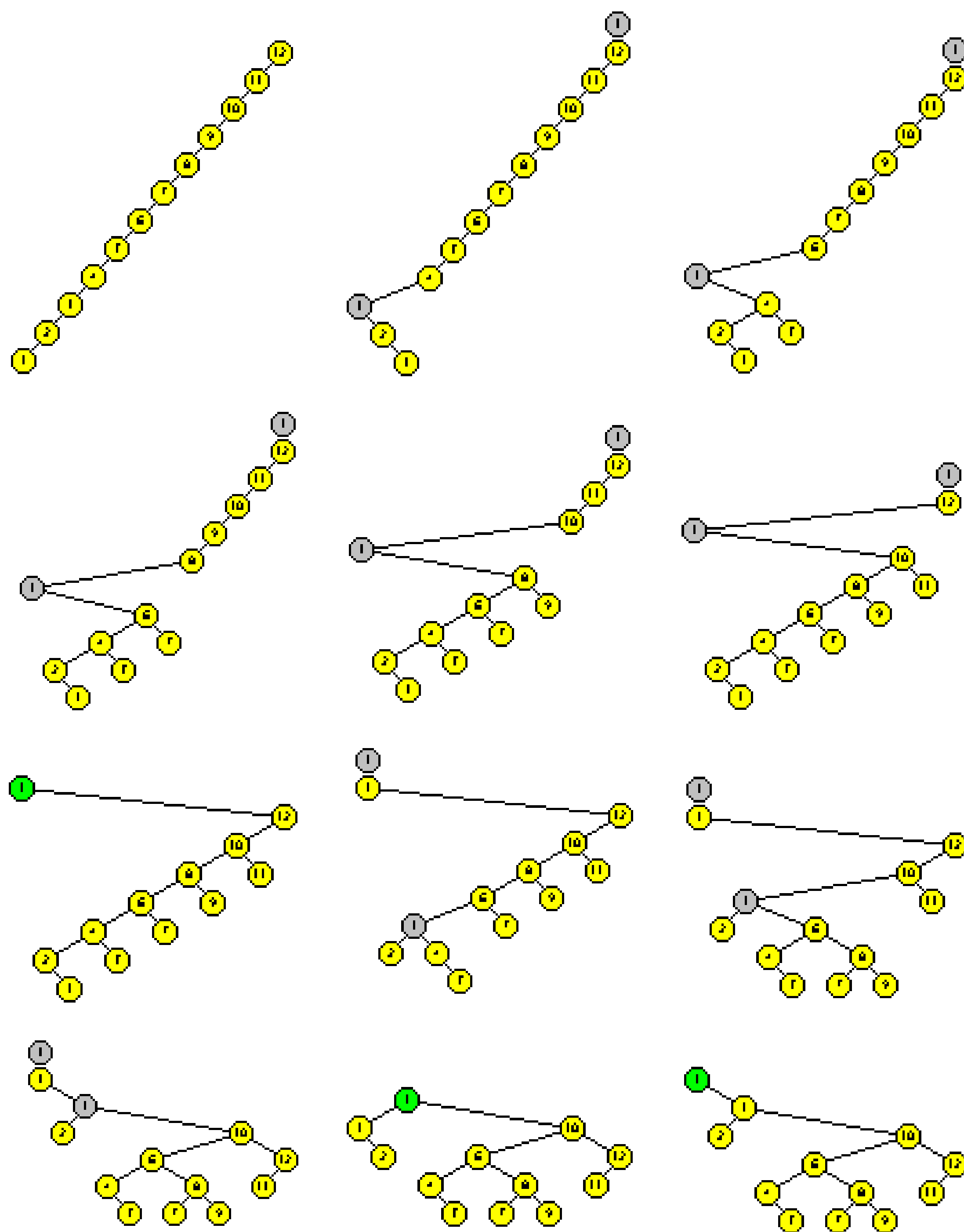
Obr. 5.3: Operácia *split* pomocou *splay*



Obr. 5.4: Operácia *insert* pomocou *splay*



Obr. 5.5: Operácia *delete* pomocou *splay*



Obr. 5.6: Do splay stromu sme vložili najskôr $1, 2, \ldots, 12$, následne sme vysplayovali číslo 1, potom 3 a opäť 1

Časť II

Vizualizácia algoritmov

Kapitola 6

Úvod

Zase sa hráš s tými guličkami?
otec, pár dní pred odovzdaním práce

§ **Cieľ.** V prvej časti sme si popísali rôzne vyhľadávacie stromy. Hlavným cieľom tejto práce je však vyhľadávacie stromy a algoritmy na nich *vizualizovať*, to znamená graficky zobrazovať ich priebeh tak, aby boli ľahko pochopiteľné. Ako sa hovorí, lepšie raz viedieť ako stokrát počuť. Výsledkom práce bude Java applet voľne prístupný na internete.

§ **Využitie.** Applet je vytváraný pre edukačné účely. Ľuďom, ktorí sa zaujímajú o dátové štruktúry (či už z potešenia alebo kvôli skúške) by mal pomôcť s preberanou látkou (BST a červenočierne stromy) a navyše utvoriť prehľad o vyvažovacích algoritmoch.

Užívateľ si môže pomocou appletu sám skontrolovať, či algoritmu naozaj rozumie tak, že bude dopredu predpovedať, čo sa bude diať.

Nakoľko autor je členom Korešpondenčného seminára z programovania (KSP), určite túto prácu využije pri prednáškach o efektívnych algoritmoch a dátových štruktúrach na sústredeniach organizovaných KSP.

„Cieľovou skupinou“ tejto práce sú teda študenti.

§ **Požiadavky.** Na vyvýjaný applet máme niekoľko požiadaviek:

- *predpovedateľnosť a konzistentnosť* – aby sa užívateľ mohol sám testovať
- *grafické spracovanie* – musí byť prehľadné a vysvetľujúce; užívateľovi by mali byť prezentované potrebné údaje v zrozumiteľnej forme, no nemal by byť príliš zahltený nepotrebnými údajmi; malo by byť tiež *jednotné* a *real-time*,
- *robustnosť* – program nesmie predpokladať nič o vstupe a riešiť každú situáciu,
- *interaktívnosť* – cieľom práce nie je animovať niekoľko prípadov; uživateľ musí mať kontrolu nad programom a možnosť výberu,
- *variabilita* – mala by byť možnosť nastaviť parametre animácie, najme *rýchlosť*,
- *rozšíriteľnosť* – program by sa mal dať ľahko rozšíriť o nové algoritmy a dátové štruktúry a preložiť do iných jazykov.

Kapitola 7

Popis

'To Start Press Any Key.'
Where's the ANY key?
Homer Simpson

§ **Spustenie.** Applet spolu s doprovodnou html-stránkou je na priloženom CD. Stačí v obľúbenom prehliadači (alebo inom programe, napríklad `appletviewer`) spustiť stránku `index.html` (potrebná je Java Virtual Machine).

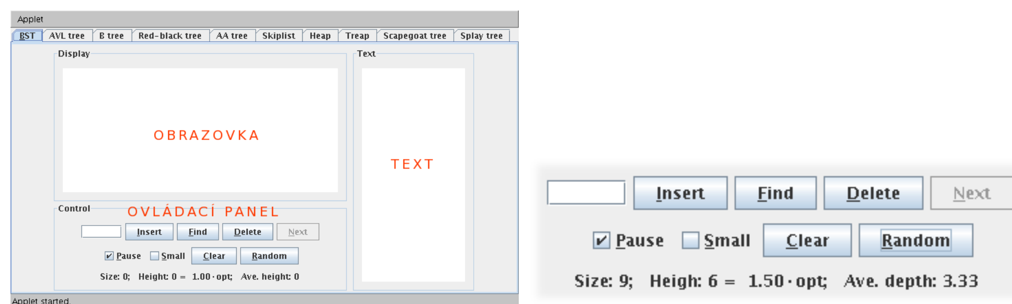
Applet je prístupný aj cez internet na adrese

`http://people.ksp.sk/~kuko/bak/`

§ **Hlavné časti.** V hornej časti appletu sa nachádza menu, z ktorého je možné vybrať si strom, ktorý chceme vizualizovať. Na výber je BST, AVL-strom, B-strom, červeno-čierny strom, AA-strom, skip list, halda, treap, GB-strom a splay strom.

Samotný panel pre vizualizáciu sa skladá z troch hlavných častí (pozri obr. 7.1a):

- v strede je obrazovka, ktorá zobrazuje danú dátovú štruktúru a priebeh algoritmu,
- dolu je ovládací panel (obr. 7.1a), pomocou ktorého užívateľ volí nastavenia a zadáva operácie a vstupné hodnoty, ktoré sa majú animovať,
- vpravo je doprovodný text, ktorý odôvodňuje kroky algoritmu.



Obr. 7.1: a) Applet b) Ovládací panel

§ Ovládanie. Základné ovládanie je jednoduché: do voľného políčka napíšeme vstupnú hodnotu, argument, kľúč, ktorý chceme vložiť, vyhľadať, alebo odstrániť. Vstupná hodnota je číslo od 0 do 99 (čísla menšie ako nula sa považujú za 0, väčšie ako 99 za 99 a vstupy, ktoré nie sú čísla sa ignorujú).

Následne klikneme na príslušnú operáciu – `Insert`/`Find`/`Delete` – a animácia sa začne. Algoritmus sa vždy zastavuje na podstatných miestach algoritmu a čaká, kým užívateľ stlačí `Next`. Potom pokračuje. Takto má užívateľ čas všetko sledovať a možnosť voliť si vlastné tempo.

Ak chceme spraviť niektoré úpravy rýchlo a nechceme sa preklikávať cez `Next`, stačí odkliknúť `Pause` a algoritmus nebude zastavovať, zobrazí sa priamo výsledok.

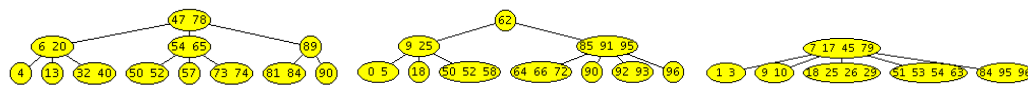
Tlačítko `Clear` celý strom vymaže, takže užívateľ môže začať odznovu. Tlačítko `Random` pridá náhodných 10 prvkov a slúži na rýchle vygenerovanie náhodného stromu.

Pomocou tlačítka `Small` sa dá zobrazenie zmenšiť, avšak vzhľadom na ciele a využitie programu predpokladáme, že užívateľ sa bude učiť skôr na malých príkladoch, ktoré sa zmestia na obrazovku.

§ Pokročilé ovládanie. Dosiaľ sme popísali ovládanie, ktoré je podľa názoru autora intuitívne a užívateľ naň príde bez toho, aby si potreboval čítať nejaký manuál. V skutočnosti do vstupného políčka netreba písať práve jeden argument. Ak pri vkladaní necháme vstupné políčko prázdne, vygeneruje sa náhodný kľúč a začne sa vkladať ten. Ak do vstupného políčka zapíšeme viacero čísel oddelených medzerami, spustí sa daná operácia viackrát, postupne s každým argumentom. Napríklad ak chceme vytvoriť binárny strom z kľúčov $1, 2, 3, 4, 5$ a poukázať na najhorší prípad binárneho stromu, stačí do vstupného políčka napísať `1 2 3 4 5`, odškrtnúť `Pause` a stlačiť `Insert` – v momente sa vytvorí žiadaný strom.

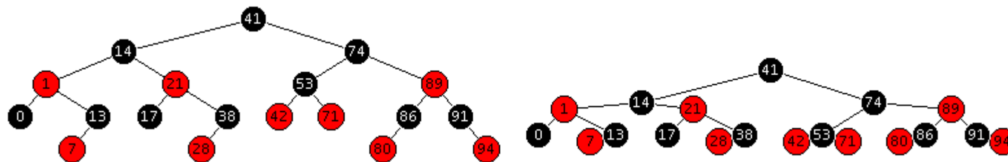
Ak do vstupného políčka zadáme n a stlačíme `Random`, pridá sa n náhodných prvkov.

§ Špeciálne nastavenia. Pod špeciálnymi nastaveniami máme na mysli nastavenia, ktoré sú príznačné len pre konkrétne druhy stromov.



Obr. 7.2: B-stromy rádu 3, 4 a 5.

B-stromy majú svoj rád, t.j. koľko detí môže mať jeden vrchol (pozri obr. 7.2). Napríklad B-stromy rádu 3 sú známe ako 2-3-stromy, rádu 4 sú 2-3-4-stromy. V applete sa dá tento parameter nastaviť, povolený je rád od 3 do 20 (podľa názoru autora je vyšší rád pre pedagogické účely zbytočný). Pri zmene tohto parametru sa celý strom z pochopiteľných dôvodov vymaže (napríklad strom rádu 5 nie je korektným stromom rádu 3 – niektoré vrcholy môžu byť preplnené).



Obr. 7.3: a) Červeno-čierny strom b) ten istý strom v 2-3-4 móde

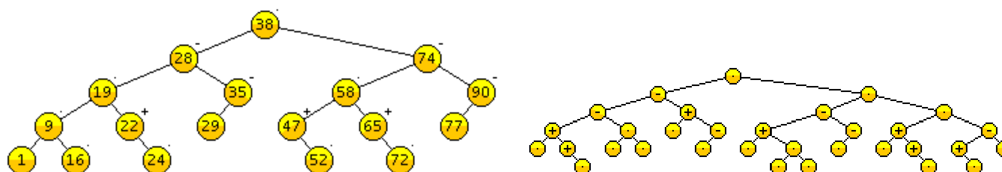
Červeno-čierne a AA-stromy majú svoj pôvod v 2-3-4- a 2-3-stromoch. Preto je prítomné políčko s názvom `2-3-4 mode` (resp. `2-3 mode`). Po jeho zaškrtnutí sa strom vykresľuje trochu inak – vrcholy patriace do toho istého pseudovrcholu majú tú istú úroveň (pozri obr. 7.3). Takto úžívateľ ľahšie uvidí vzťah medzi červeno-čiernymi a 2-3-4-stromami a medzi AA- a 2-3-stromami.

GB-stromy majú parameter α, ktorý určuje, ako veľmi nevyvážené môžu byť skôr ako sa celé prestavajú; α je v tomto programe desatinné číslo väčšie ako 1 a menšie alebo rovné 5.

§ **Štatistiky.** V poslednom riadku ovládacieho panelu sa nachádzajú štatistiky. Medzi štatistikami sú:

- `Size` – počet vrcholov stromu,
- `Height` – výška stromu vyjadrená tiež ako násobok perfektne vyváženého, teda optimálneho stromu (napríklad ak má AVL-strom s 54 vrcholmi výšku 7, optimálny strom má výšku 6, čo je približne 1.17-násobok),
- `Ave. depth` – priemerná hĺbka vrcholov, teda $\sum_v d(v)/\,size(T)$ je to o 1 menšie číslo ako je priemerný počet porovnaní potrebných na nájdenie nejakého vrcholu,
- `#Nodes` a `#Keys` – pri B-stromoch rozlišujeme počet vrcholov a počet vložených kľúčov, pretože jeden vrchol väčšinou obsahuje viacero kľúčov,
- `#Deleted` – v GB-strome počítame počet vymazaných vrcholov – ak je tento väčší ako polovica počtu vrcholov, nastáva prestavanie stromu.
- `#Excess nodes` – v skip liste je to počet vrcholov, ktoré sú „navyše" (okrem zarážiek a spodnej úrovne)

§ **Zobrazenie informácií.** Strom zobrazujeme tak, ako je zaužívané, s koreňom nahor. Vrcholy stromu sú kruhy, kľúč je číslo vnútri kruhu. Hrana, ktorou je spojený otec so synom je zobrazená ako úsečka.



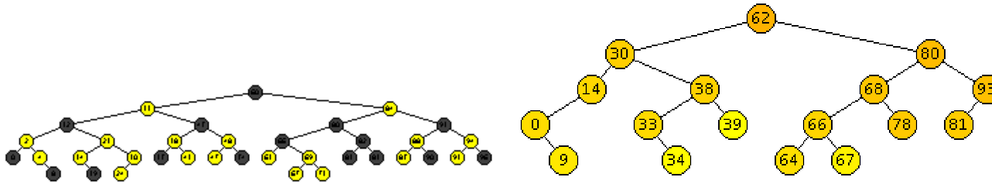
Obr. 7.4: Zobrazenie AVL-stromov

V prípade AVL-stromu je pri každom vrchole navyše malé +, · alebo − podľa toho, či je vrchol naklonený doprava, vyvážený, alebo doľava. Táto informácie je tiež zobrazená farebne oranžovým polkruhom v smere naklonenia (pozri obr. 7.4).

Ak chceme nejaký vrchol zvýrazniť, používame dve možnosti: vrchol zakrúžkujeme alebo mu zmeníme farbu. Napríklad vrchol, ktorý vkladáme, je modrý. Vrchol, ktorý mažeme, červený. Vrchol, ktorý hľadáme, šedý a ak nájdeme, zelený, ak nenájdeme, červený.

Farba je vôbec dôležitý nosič informácie. Neodškriepiteľné je to v prípade červeno-čiernych stromov. V GB-strome vrcholy označené na vymazávanie sú tmavo šedé (pozri obr. 7.5a). Farbu sme však využili aj v halde a treape. V treape potrebujeme zobraziť dve informácie: kľúč a prioritu. Bolo by neprehľadné zobrazovať obe informácie naraz. Z hľadiska jednotnosti by mal v krúžku byť kľúč a priorita by mohla byť ako ďalšia informácia vedľa vrcholu. Oveľa prehľadnejšie je však zobraziť prioritu pomocou farby. Zvolili sme odtiene žltej a oranžovej – čím má vrchol väčšiu prioritu, tým oranžovejšiu (viac do červeno-hneda) farbu má. Takto zachováme jednotnosť spracovania a užívateľ ľahko skontroluje, že priority majú haldovité usporiadanie jedným pohľadom (zdola nahor by mali farby tmavnúť; pozri obr. 7.5b).

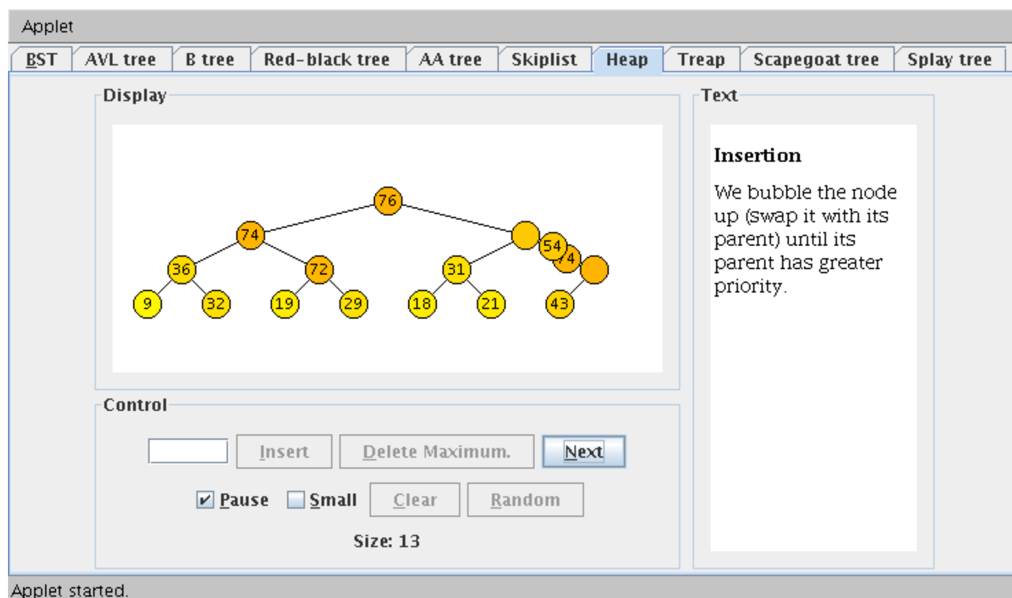
§ **Applet v akcií.** Applet v akcií je zobrazený na obr. 7.6. Vidieť tu vkladanie do BST. Vkladáme nový vrchol s kľúčom 50. Algoritmus práve čaká, kým stlačíme `Next`, aby mohol pokračovať ďalej. Text vpravo vysvetľuje, čo sa práve udialo: keďže 50 $>$ 23, vkladáme nový vrchol do pravého podstromu. Užívateľ, ktorý si testuje svoje vedomosti môže tipovať, že 50 $<$ 79 a preto by sme mali pokračovať vľavo. Po stlačení `Next` mu dá program za pravdu.



Obr. 7.5: Zobrazenie a) GB-stromu b) treapu

Na obr. 7.7 je zachytené vkladanie do haldy. Vrchol 74 sme pridali na koniec a teraz ho prebublávame nahor. Na obrázku sa práve vymieňa s 54. (Poznámka: štatistika `size` = 14 sa opraví až na koniec.)

Obr. 7.6: Applet v akcií: Vkladanie do BST



Obr. 7.7: Applet v akcií: Vkladanie do haldy

Kapitola 8

Implementácia

> Programmer = an organism that turns
> coffee into software.

§ **Základné triedy.** Celý applet sa skladá z panelov pre jednotlivé dátové štruktúry. Ako sme spomenuli, panel sa skladá z obrazovky, ovládacieho panelu a textu. Na obrazovke sa vykreslujú dátové štruktúry a bežia algoritmy. Tomu zodpovedajú základné triedy:

- Trieda `Screen` vytvára miesto pre obrazovku – rozširuje triedu `JPanel`, pričom poskytuje vlasnú implementáciu procedúry `paint`. Pri zavolaní procedúry `paint` sa vykreslí dátová štruktúra. Trieda tiež implementuje rozhranie `Runnable`. Obrazovka teda beží v samostatnom vlákne a periodicky sa prekresľuje (tak je zabezpečená plynulá animácia). Poznamenajme ešte, že toto vlákno vieme pozastaviť, ak danú obrazovku momentálne nepotrebujeme.
- Trieda `Buttons` zodpovedá za tlačítka (spodnú časť obrazovky). Implementuje rozhranie `ActionListener`. Keď užívateľ stlačí tlačítko, je na tejto triede, aby dotaz spracovala. Musí parsovať vstupný text a zavolať príslušnú funkciu dátovej štruktúry.
- Trieda `Commentary` je podtriedou `JEditorPane` a implementuje jednoduché procedúry na vypísanie textu.
- Trieda `DataStructure` je abstraktná trieda. Každá dátová štruktúra, ktorú vykresľujeme implementuje práve túto triedu. Všetky tieto štruktúry totiž majú nejaké spoločné vlastnosti, ktoré od nich navyše vyžadujeme. Napríklad každá dátová štruktúra musí vedieť povedať, aké potrebuje tlačítka. Musí sa vedieť vykresliť, zväčšiť/zmenšiť, či referovať o sebe nejaké štatistiky.
- Trieda `Algorithm` je zase trieda, ktorú rozširujú všetky algoritmy. `Algorithm` je vlastne vlákno, ktoré sa však navyše vie bezpečne zastaviť, aby sa mohlo opäť spustiť po stlačení `Next`.

Triedu `DataStructure` ešte rozširujú dve abstraktné triedy: `Dictionary` a `PriorityQueue`, ktoré špecifikujú, aké tlačítka dátová štruktúra potrebuje a aké operácie vykonáva. Konkrétne v prípade `Dictionary` sú to operácie *insert*, *find* a *delete*, v prípade `PriorityQueue` je to *insert* a *delete maximum*. Halda rozširuje `PriorityQueue`, ostatné dátové štruktúry rozširujú `Dictionary` – tým je venovaná táto práca.

§ **Kreslenie stromu.** Problém kreslenia stromu je pre každý vrchol v daného stromu vypočítať súradnice $x(v)$, $y(v)$, aby bol výsledný obrázok čo možno „najkrajší“. Toto zjavne nie je exaktná úloha, jednotlivé riešenia však vieme porovnávať. Určite si neželáme, aby sa nám krížili hrany. V našom prípade by sme chceli, aby vrcholy navyše išli zhora nadol. Konkrétne nám pôjde o tzv.

úrovňové kreslenie, kde $y(v) = -d(v)$, kde $d(v)$ je hĺbka vrcholu v (ďalej sa budeme zaoberať iba x-ovými súradnicami). Od dobrého kresliaceho algoritmu by sme ďalej mohli požadovať napríklad to, aby bolo nakreslenie istým spôsobom symetrické, ak je symetrická daná stromová štruktúra a aby bol priestor, ktorý nakreslenie zaberá, pokiaľ možno malý, teda aby strom nebol príliš rozťahaný.

Jednoduchý kresliaci algoritmus môže vyzerať takto: koreň dáme do stredu obrazovky; jeho synov do 1. a 3. štvrtiny, ich synovia postupne do +., 3., 5. a 7. osminy, atď. Inými slovami, keby sme kreslili úplný strom, koreň by sme dali do stredu; tým sa nám obrazovka pomyselne rozdelí na pravú a ľavú polovicu synov vždy dáme do stredov týchto polovíc. Vrcholy neúplného stromu budú na tých istých miestach ako v úplnom. Tento algoritmus je vhodný pre úplné stromy, ale nie je veľmi vhodný pre „deravé“ stromy.

Iná možnosť je každému vrcholu priradiť $rank(v)$, poradie vrcholu v inorderi a položiť $x(v) = rank(v)$. Výhodou daných algoritmov okrem jednoduchosti je fakt, že vrcholy skôr v inorderi, v prípade vyhľadávacích stromov to znamená s menším kľúčom, sú vľavo od tých s väčším kľúčom. Ľahko sa teda kontroluje, či je strom vyhľadávací. Ľahko sa hľadá následník a v smere zľava doprava vidíme kľúče utriedené. Nevýhodou je že nakreslenia sú širšie ako je nutné a v druhom prípade rodič nie je centrovaný vzhľadom na deti.

Lepší algoritmus navrhli Reingold a Tilford [RT81]: Ak T obsahuje jediný vrchol, kreslenie je jednoduché. Inak sa rekurzívne zavoláme na ľavý a pravý podstrom a nakreslíme najskôr tie. Následne tieto podstromy priblížime tak, aby ich vzdialenosť bola 2 a vložíme vrchol do stredu medzi korene podstromov. Ak má vrchol iba jedného syna, napríklad ľavého, nakreslíme ho o 1 vpravo. Myšlienka tohto algoritmu je jednoduchá, problémom je efektívna implementácia. Reingold a Tilford však ukázali, že sa to dá v lineárnom čase.

V našom programe používame prakticky druhý prístup. Implementácia je rekurzívna: list zabalíme do pomyselnej krabičky; pri ostatných vrcholoch najskôr zabalíme do krabičky ľavý a pravý podstrom, krabičky položíme vedľa seba, na ich rozhranie položíme koreň a celé to zabalíme do krabičky. Tento algoritmus sa dá ľahko zovšeobecniť pre viaccestné stromy.

§ **Kreslenie vrcholov.** Trieda `Node`, teda vrchol, tvorí základnú grafickú jednotku. `Node` má svoj kľúč, súradnice (x, y), kde sa práve nachádza, súradnice (tox, toy), kam smeruje a počet krokov *steps*, na koľko to musí stihnúť. Ďalej sa dá nastaviť farba pozadia a popredia a zakrúžkovať.

Vrchol sa vie vykresliť a pohnúť. Hýbanie vrcholmi zabezpečuje trieda `Screen` a dátová štruktúra. `Screen` sa periodicky prekresľuje, a dátová štruktúra vždy keď sa vykreslí, pohne všetkými vrcholmi. Ďalej vrchol obsahuje pomocné procedúry ako „choď na toto miesto“, „choď tam, kde je vrchol v“, alebo „kam ide vrchol v“, „choď vedľa vrcholu v“ (pre viacero hodnôt slova „vedľa“ a podobne).

Rozšírením `Node` je `BSTNode`, teda vrchol binárneho stromu. Väčšina iných vrcholov potom rozširuje práve `BSTNode`. Zatiaľčo `Node` je iba „grafický prvok“, `BSTNode` už reprezentuje vrchol stromu a jeho nakreslenie. Obsahuje teda navyše smerník na ľavého a pravého syna a rodiča.

Zatiaľčo `Node` sa vie nakresliť, `BSTNode` vie nakresliť seba a celý svoj podstrom. Má tiež definované niektoré stromové operácie ako prilinkovanie vrcholu, prepočítanie súradníc stromu či prepočítanie štatistík stromu.

Kapitola 9

Záver

§ **Existujúce riešenia.** Asi najlepšou vizualizáciou na internete je v súčasnosti

`http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm`

Vizualizovaný je AVL-strom, červeno-čierny strom a splay strom. Umožňuje vkladanie, hľadanie, vyberanie a traverzovanie, dokonca jednotlivé rotácie. Tento applet má pútavú grafiku a spracovanie.

Asi najlepšou vizualizáciou B-stromov je

`http://slady.cz/java/bt/`

Dobrým a celkom bohatým zdrojom sú

`http://dbs.mathematik.uni-marburg.de/research/projects/dsn/`

s BST, AVL-stromom, červeno-čiernym stromom, splay stromom a treapom (tiež spájanými zoznamami a triedeniami) a

`http://www.cs.usfca.edu/galles/visualization/`

(kde je z vyhľadávacích stromov iba BST, AVL-strom a B-strom, na druhej strane je tam veľa iných dátových štruktúr).

Na internete sme však nenašli uspokojivú vizualizáciu skip listov a žiadnu vizualizáciu AA-stromov a GB-stromov.

§ **Výsledok.** Implementovali sme applet s deviatimi dátovými štruktúrami: BST, AVL-stromom, B-stromom, červeno-čiernym stromom, AA-stromom, skip listom, haldou, treapom, GB-stromom a Splay stromom. Aplikácia je napísaná spôsobom, ktorý počítal s rozšírením v budúcnosti. Ako ukazuje skip list, vytvorené prostredie sa nemusí špecializovať výlučne na stromy a ako ukazuje halda, vizualizovaná dátová štruktúra nemusí byť ani slovník.

Hlavný prínos appletu je, že nielen animuje dátovú štruktúru, ale tiež vysvetľuje (vďaka textovému oknu). Podstatné je tiež, že v aplikácií sú vizualizované aj stromy, ktoré zatiaľ nie sú dostupné na internete, napríklad AA-strom, treap, GB-strom.

§ **Budúca práca na projekte.** Dátových štruktúr je nespočetné množstvo, ani užšieho výberu dátových štruktúr (vyhľadávacie stromy, implementácie slovníka) sme nemohli naprogramovať všetky. Vôbec sme neimplementovali napríklad BB[α]-stromy, αBB-stromy, nehovoriac o neighbour tree, son tree, brother tree. Aj z implementovaných dátových štruktúr je implementovaný vždy len jeden variant: nie sú implementované napríklad B^+ či B^* stromy, zaujímavý variant splayovania je semisplayovanie. Taktiež nie sú implementované algoritmy pracujúce zhora nadol (napríklad pre B-stromy a splay stromy také existujú).

Skôr ako zaoberať sa kurióznymi dátovými štruktúrami by som však chcel vizualizovať praktické veci: vizualizáciu jednoduchých operácií ako rotácie, delenie vrcholu, možno prestavanie. Užívateľ si ich tak rýchlejšie osvojí, ako keď ich musí sledovať spolu s ďalším množstvom informácií v kontexte nejakého algoritmu (ktorý už rotácie používa ako základnú operáciu). Ďalšou potrebnou operáciou je traverzovanie, zobrazenie/hľadanie predchodcu, následníka.

Z dátových štruktúr by som v najbližšom čase chcel implementovať union-find a perzistentný BST. Aplikáciu tiež treba preložiť do slovenčiny.

Literatúra

[AM78] Brian Allen and Ian Munro, *Self-organizing binary search trees*, Journal of the ACM **25** (1978), 526–535.

[And91] Arne Andersson, *A note on searching in a binary search tree*, Software - Practice and Experience **21** (1991), no. 10, 1125–1128.

[And93] ———, *Balanced search trees made simple*, Algorithms and Data Structures, Third Workshop (Montréal, Canada) (Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, eds.), Lecture Notes in Computer Science, vol. 709, Springer, 11–13 August 1993, pp. 60–71.

[And99] ———, *General balanced trees*, J. Algorithms **30** (1999), no. 1, 1–18.

[Aoe90] Jun-Ichi Aoe, *A compendium of key search references*, SIGIR Forum **24** (1990), no. 3, 26–42.

[AS89] C. Aragon and R. Seidel, *Randomized search trees*, Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci., 1989, pp. 540–545.

[Bay72] Rudolf Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica **1** (1972), 290–306.

[Bit79] James R. Bitner, *Heuristics that dynamically organize data structures*, SIAM Journal on Computing **8** (1979), no. 1, 82–110.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, Cambridge, MA, 2001, Second Edition.

[Dev86] Luc Devroye, *A note on the height of binary search trees*, Journal of the ACM **33** (1986), no. 3, 489–498.

[Dev87] ———, *Branching processes in the analysis of the heights of trees*, Acta Informatica **24** (1987), no. 3, 277–298.

[DR95] Devroye and Reed, *On the variance of the height of random binary search trees*, SICOMP: SIAM Journal on Computing **24** (1995).

[GR93] Igal Galperin and Ronald L. Rivest, *Scapegoat trees*, SODA, 1993, pp. 165–174.

[GS78] Guibas and Sedgewick, *A dichromatic framework for balanced trees*, FOCS: IEEE Symposium on Foundations of Computer Science (FOCS), 1978.

[Knu73] D. Knuth, *The art of computer programming, volume 1: Fundamental algorithms*, Addison-Wesley, Reading, MA., 1973.

[Knu78] ———, *The art of computer programming, volume 3: Sorting and searching*, Addison-Wesley, Reading, MA., 1978.

[Koz92] Dexter C. Kozen, *The design and analysis of algorithms*, Texts and Monographs in Computer Science, Springer-Verlag, New York, 1992.

[Pug90] William Pugh, *Skip lists: A probabilistic alternative to balanced trees*, Commun. ACM **33** (1990), no. 6, 668–676.

[Ree03] Bruce A. Reed, *The height of a random binary search tree*, J. ACM **50** (2003), no. 3, 306–332.

[Rol02] Rolfe, *One-time binary search tree balancing: The day/stout/warren (DSW) algorithm*, SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) **34** (2002).

[RT81] E. Reingold and J. S. Tilford, *Tidier drawing of trees.*, IEEE Transactions On Software Engineering **SE-7** (1981), no. 2, 223–228.

[ST83] Sleator and Tarjan, *Self-adjusting binary trees*, STOC: ACM Symposium on Theory of Computing (STOC), 1983.

[STT88] Sleator, Tarjan, and Thurston, *Rotation distance, triangulations, and hyperbolic geometry*, JAMS: Journal of the American Mathematical Society **1** (1988).

[SW86] Stout and Warren, *Tree rebalancing in optimal time and space*, CACM: Communications of the ACM **29** (1986).

[Tar83a] Robert E. Tarjan, *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, 1983.

[Tar83b] Robert Endre Tarjan, *Updating a balanced search tree in $O(1)$ rotations*, Information Processing Letters **16** (1983), no. 5, 253–257.

[Vui80] Vuillemin, *A unifying look at data structures*, CACM: Communications of the ACM **23** (1980).