Obsah

1 Úvod

1.1 Algoritmus

Neformálne možno povedať, že ***algoritmus*** je presne definovaná procedúra, ktorá pre nejaké množstvo hodnôt vráti nejaký výstup. Na algoritmus možno hľadieť ako na nástroj pre riešenie presne špecifikovaného výpočtového problému.

Hovoríme, že algoritmus je ***správny***, ak pre každý zmysluplný vstup sa jeho vykonávanie zastaví v konečnom čase, pričom vráti správny výsledok. Nekorektný algoritmus sa nemusí zastaviť na všetky zmysluplné vstupy (vtedy hovoríme o čiastočnej správnosti) alebo sa zastaví a poskytne inú ako očakávanú hodnotu [1]. Algoritmus možno špecifikovať ako počítačový program, formulovať ho v nejakom prirodzenom jazyku alebo hoci aj ako nejaký hardwareový design. Jediná požiadavka je, aby špecifikácia poskytovala presný popis výpočtovej procedúry, podľa ktorej sa bude vykonávanie riadiť. V ďalšom budeme algoritmy zväčša popisovať programami v pseudokóde, ktorý sa podobá bežným programovacím jazykom, ale je možné, že sa vyskytne aj popis v prirodzenom jazyku.

1.1.1 Insert Sort

Triedenie vkladaním je efektívny algoritmus pre triedenie malého počtu prvkov. Na vstupe máme ľubovoľnú postupnosť prvkov, vezmeme prvok z tejto postupnosti a zaradíme ho do spočiatku prázdnej výstupnej postupnosti na také miesto, aby zostala usporiadanou a túto činnosť budeme vykonávať až dokým nevyčerpáme všetky prvky vstupnej postupnosti.

Pseudokód pre triedenie vkladaním je prezentovaný ako procedúra INSERT-SORT, ktorá vezme ako parameter pole $A[1..n]$ obsahujúce postupnosť dĺžky n, ktorú treba utriediť (v kóde je počet n prvkov A stanovený pomocou $length[A]$). Vstupné čísla sú triedené ***in situ*** [2]. Po ukončení vykonávania procedúry INSERT-SORT bude vstupné pole A obsahovať utriedenú výstupnú postupnosť.

INSERT-SORT(A)

```
1  for j ← 2 to length[A] do
2     key ← A[j]
3     ▷ Vlož A[j] do usporiadanej postupnosti A[1..j − 1].
4     i ← j − 1
5     while (i > 0) and (A[i] > key) do
6        A[i + 1] ← A[i]
7        i ← i − 1
8     A[i + 1] ← key
```

1.2 Analýza algoritmov

Analyzovať algoritmus znamená predpokladať množstvo zdrojov, ktoré algoritmus bude vyžadovať pre svoj beh. Zvyčajne sa zaoberáme efektívnosťou algoritmov na základe veľkosti spotrebovávaných zdrojov (nároky na pamäť, šírka komunikácie, rozmery logických obvodov, čas výpočtu – časová zložitosť $T(n)$).

1.2.1 Analýza triedenia insert sort

Čas spotrebovaný procedúrou INSERT-SORT závisí na vstupe: triedenie tisíc čísel trvá dlhšie ako triedenie desiatich čísel. Vykonávanie INSERT-SORT môže trvať rôzne dlho pre postupnosti rovnakej dĺžky v závislosti od toho, nakoľko usporiadané už tieto postupnosti sú. Vo všeobecnosti čas

[1] Na tomto mieste treba poznamenať, že sa používajú aj nekorektné algoritmy, napr. vtedy ak sme schopní určovať chybu výsledku.

[2] Čísla sú preusporiadavané vnútri poľa, pričom mimo neho je uchovávaný nanajvýš konštantný počet týchto čísel.

potrebný na beh algoritmu rastie s veľkosťou vstupu, teda zvykneme popisovať čas behu programu ako funkciu veľkosti jeho vstupu. Preto zadefinujeme pojmy „čas behu“ a „veľkosť vstupu“ trochu presnejšie.

Začneme prezentovaním procedúry INSERT-SORT s nárokmi na čas každého príkazu a počtom vykonaní každého z príkazov. Pre každé $j = 2, 3, \ldots, n$, kde $n = \mathit{length}[A]$, položíme t_j rovné počtu vykonaní testov v cykle `while` na riadku 5 pre danú hodnotu j. Predpokladáme, že komentáre nie sú vykonateľné príkazy, a preto nespotrebovávajú žiaden čas.

INSERT-SORT(A)	*trvanie*	*počet vykonaní*
1 **for** $j \leftarrow 2$ **to** $\mathit{length}[A]$ **do**	c_1	n
2 $\quad \mathit{key} \leftarrow A[j]$	c_2	$n-1$
3 $\quad \triangleright$ Vlož $A[j]$ do usporiadanej postupnosti $A[1\,..\,j-1]$.	0	
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **while** $(i > 0)$ **and** $(A[i] > \mathit{key})$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad\quad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

Čas behu algoritmu je súčet časov behu pre každý vykonaný príkaz; príkaz, ktorý potrebuje c_i časových jednotiek pre vykonanie a je vykonaný n krát, bude prispievať $c_i n$ do celkového času behu [3]. Aby sme vypočítali celkový čas behu $T(n)$ triedenia insert sort, sčítame súčiny hodnôt v stĺpcoch *trvanie* a *počet vykonaní*:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \; .$$

Hoci pre vstupy rovnakej veľkosti, čas behu algoritmu môže závisieť od toho aké vlastnosti má daný vstup. Napríklad, pre insert sort nastáva najlepší prípad keď je vstupné pole už utriedené. Pre každé $j = 2, 3, \ldots, n$ potom zistíme, že $A[i] \leq \mathit{key}$ na riadku 5 keď i má počiatočnú hodnotu $j-1$. Takto $j = 1$ pre $j = 2, 3, \ldots, n$ a najlepší čas behu je

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned}$$

Tento čas možno vyjadriť ako $an + b$ pre konštanty a a b, ktoré závisia od trvaní príkazov c_i; je to teda *lineárna funkcia* n.

Ak je vstupné pole usporiadané v klesajúcom poradí, nastáva najhorší prípad. Vtedy musíme porovnať každý prvok $A[j]$ s každým prvkom v celom podpoli $A[1\,..\,j-1U$, a teda $t_j = j$ pre $j = 2, 3, \ldots, n$.

Zisťujeme, že v najhoršom prípade čas behu insert sortu je

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{3} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned}$$

Tento najhorší čas behu možno vyjadriť ako $an^2 + bn + c$ pre konštanty a, b a c, ktoré opäť závisia na trvaní príkazov c_i; je to teda *kvadratická funkcia* n.

1.3 Návrh algoritmov

Existuje mnoho spôsobov ako navrhovať algoritmy. Insert sort využíva inkrementálny prístup: Nech máme utriedené podpole $A[1\,..\,j-1]$, vložením prvku $A[j]$ na správne miesto dostávame utriedené podpole $A[1\,..\,j]$.

[3] Táto charakteristika nemusí platiť pre zdroje ako napríklad pamäť. Príkaz, ktorý pristupuje ku m slovám v pamäti a je vykonaný n krát nie nutne pristúpi ku mn slovám.

V tejto sekcii sa budeme zaoberať alternatívnou metódou známou ako „*divide & conquer*“. Túto metódu použijeme na návrh triedenia, ktorého najhorší čas behu je pre dostatočne veľké vstupy omnoho kratší ako insert sortu. Výhodou divide & conquer algoritmov je, že ich časy behov je možno často veľmi ľahko stanoviť.

1.3.1 Metóda divide & conquer

Mnoho problémov je rekurzívnych, čo sa týka štruktúry, t.j. ich možno rozdeliť na podproblémy, ktoré je možné riešiť obdobným spôsobom. Riešenie takýchto problémov metódou *divide & conquer* pozostáva z troch krokov na každej úrovni rekurzie:

- **Rozdelenie** problému na podproblémy (*divide*).
- Rekurzívne **vyriešenie** každého z podproblémov (*conquer*). Ak je podproblém dostatočne malý, vyriešime ho nerekurzívne.
- **Spojenie** riešení podproblémov do riešenia pôvodného problému (*combine*).

Jedným z algoritmov, ktoré vznikli riešením problému touto metódou, je algoritmus ***merge sort***, ktorý pracuje približne takto:

- Rozdelenie n-prvkovej postupnosti, ktorú treba utriediť, na dve podpostupnosti, každá dĺžky približne $n/2$.
- Rekurzívne utriedenie podpostupností použitím merge sortu.
- Zlúčenie dvoch utriedených podpostupností do výslednej utriedenej postupnosti.

Treba poznamenať, že rekurziu treba ukončiť v elementárnom prípade, t.j. keď triedená postupnosť má dĺžku 1, vtedy už je totiž automaticky utriedená.

Kľúčovou operáciou algoritmu merge sort je zľučovanie dvoch utriedených postupností. Pre zľučovanie utriedených postupností použijeme pomocnú procedúru MERGE(A, p, q, r), kde A je pole a p, q a r sú indexy prvkov tohto poľa také, že $p \leq q < r$. Procedúra predpokladá, že podpolia $A[p\,..\,q]$ a $A[q+1\,..\,r]$ sú utriedené. Zlúči ich do jedného utriedeného podpoľa, ktoré nahradí súčasné podpole $A[p\,..\,r]$ [4].

Teraz môžeme použiť procedúru MERGE ako subrutinu v našom algoritme merge sort. Procedúra MERGE-SORT(A, p, r) utriedi prvky v podpoli $A[p\,..\,r]$. Ak $p \geq r$, podpole má nanajvýš jeden prvok, a teda je už utriedené. Inak, krok delenia jednoducho vypočíta index q, ktorý rozdelí $A[p\,..\,r]$ na dve podpolia: $A[p\,..\,q]$, obsahujúce $\lceil n/2 \rceil$ prvkov a $A[q+1\,..\,r]$, obsahujúce $\lfloor n/2 \rfloor$ prvkov.

MERGE-SORT(A, p, r)

```
1  if p < r then
2     q ← ⌊(p + r)/2⌋
3     MERGE-SORT(A, p, q)
4     MERGE-SORT(A, q + 1, r)
5     MERGE(A, p, q, r)
```

Pre utriedenie celej postupnosti $A = \langle A[1], A[2], \ldots, A[n] \rangle$ vyvoláme procedúru MERGE-SORT s parametrami $(A, 1, \mathit{length}[A])$, kde opäť $\mathit{length}[A] = n$.

1.3.2 Analýza divide & conquer algoritmov

Ak algoritmus obsahuje rekurzívne volanie samého seba, čas jeho vykonávania možno často popísať ***rekurentným vzťahom*** alebo ***rekurenciou***, ktorá charakterizuje celkové trvanie behu na vstupe dĺžky n pomocou časov behu na kratších vstupoch.

Rekurencia pre čas vykonávania divide & conquer algoritmu je založená na troch krokoch. Nech $T(n)$ je čas behu na probléme veľkosti n. Ak veľkosť problému je dostatočne malá, povedzme $n \leq c$

[4] Hoci prenechávame presný popis tejto procedúry na čitateľa, je ľahké si predstaviť, že jej čas vykonávania je lineárnou funkciou $n = r - p + 1$, t.j. počtu práve zlučovaných prvkov.

pre nejakú konštantu c, priamočiare riešenie trvá konštantný čas, čo píšeme ako $\Theta(1)$. Predpokladajme, že problém rozdelíme na a podproblémov, z ktorých každý má veľkosť $1/b$ pôvodného. Ak vezmeme $D(n)$ ako čas potrebný na rozdelenie problému na podproblémy a $C(n)$ ako čas potrebný na skombinovanie riešení podproblémov do riešenia pôvodného problému, dostaneme rekurenciu

$$T(n) = \begin{cases} \Theta(1) & \text{ak } n \leq c\,, \\ aT(n/b) + D(n) + C(n) & \text{inak.} \end{cases}$$

Neskôr si ukážeme ako riešiť rekurencie tohto tvaru.

Analýza triedenia merge sort. Pokúsme sa stanoviť rekurenciu pre najhorší čas behu merge sortu pre n čísel. Merge sort pre jeden prvok trvá konštantný čas. Ak máme $n > 1$ prvkov, čas vykonávania stanovíme nasledujúcim spôsobom:

- Krok delenia iba vypočítava index prostredného prvku podpoľa, čo trvá konštantný čas. Teda $D(n) = \Theta(1)$.
- Rekurzívne riešime dva podproblémy, každý o veľkosti $n/2$, čo prispieva časom $2T(n/2)$ do celkového času vykonávania.
- Skombinovanie riešení podproblémov trvá $\Theta(n)$, čo je čas vykonávania procedúry MERGE na n-prvkovom podpoli, teda $C(n) = \Theta(n)$.

Ak sčítame funkcie $D(n)$ a $C(n)$ dostaneme lineárnu funkciu n, teda $D(n) + C(n)$ je $\Theta(n)$. Pripočítaním ku výrazu $2T(n/2)$ dostávame rekurenciu pre najhorší čas behu $T(n)$ triedenia merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{ak } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{ak } n > 1\,. \end{cases}$$

Neskôr ukážeme, že $T(n)$ je $\Theta(n \lg n)$, kde $\lg n$ je iný zápis pre $\log_2 n$. Pre dostatočne veľké vstupy je teda triedenie merge sort rýchlejšie ako insert sort.

2 Matematické základy

2.1 Rast funkcií

Stupeň rastu času behu algoritmu jednoducho charakterizuje účinnosť algoritmu a taktiež nám umožňuje navzájom porovnávať alternatívne algoritmy. Niekedy sme síce schopní presne určiť dobu vykonávania algoritmu, ale zvyčajne to nemá veľký praktický význam. Pre dostatočne veľké vstupy je možné zanedbať multiplikatívne konštanty ako aj výrazy nižšieho rádu. Preto sa zvyčajne zaoberáme ***asymptotickou*** účinnosťou algoritmov. Algoritmus, ktorý je asymptoticky efektívnejší, bude zrejme najrýchlejší pre všetky dostatočne veľké vstupy.

2.1.1 Asymptotická notácia

Θ-notácia (Asymptoticky tesné ohraničenie)
Pre danú funkciu $g(n)$ označujeme $\Theta(g(n))$ množinu funkcií

$$\Theta(g(n)) = \{f(n) \ : \text{existujú kladné konštanty } c_1, c_2 \text{ a } n_0 \text{ také, že} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ pre všetky } n \geq n_0\}\,.$$

Funkcia $f(n)$ teda patrí do množiny $\Theta(g(n))$ ak existujú kladné konštanty c_1 a c_2 také, že $f(n)$ môže byť vtesnaná medzi $c_1 g(n)$ a $c_2 g(n)$ pre dostatočne veľké n. Túto skutočnosť zapisujeme ako $f(n) = \Theta(g(n))$ [5].

Definícia $\Theta(g(n))$ vyžaduje, aby každá funkcia patriaca do $\Theta(g(n))$ bola asymptoticky nezáporná. Zrejme $g(n)$ by mala byť tiež asymptoticky nezáporná, lebo inak by množina $\Theta(g(n))$ bola prázdna. Preto budeme predpokladať, že každá funkcia vo Θ-notácii je asymptoticky nezáporná.

[5] Tento zápis sa môže spočiatku zdať zavádzajúci, no neskôr uvidíme, že prináša jednoznačné výhody.

O-notácia (Asymptotické ohraničenie zhora)
Ak máme iba asymptotické ohraničenie zhora, používame O-notáciu. Pre danú funkciu $g(n)$ označujeme $O(g(n))$ množinu funkcií

$$O(g(n)) = \{f(n) : \text{existujú kladné konštanty } c \text{ a } n_0 \text{ také, že } 0 \leq f(n) \leq c\,g(n) \text{ pre všetky } n \geq n_0\} .$$

Skutočnosť, že funkcia $f(n)$ je prvkom množiny $O(g(n))$ označujeme, podobne ako v predchádzajúcom prípade, $f(n) = O(g(n))$. Treba poznamenať, že z $f(n) = \Theta(g(n))$ vyplýva $f(n) = O(g(n))$, nakoľko Θ-notácia je silnejšia ako O-notácia, t.j. $\Theta(g(n)) \subseteq O(g(n))$.

Ω-notácia (Asymptotické ohraničenie zdola)
Ako O-notácia poskytuje asymptotické ohraničenie zhora, tak Ω-notácia poskytuje asymptotické ohraničenie zdola. Pre danú funkciu $g(n)$ označujeme $\Omega(g(n))$ množinu funkcií

$$\Omega(g(n)) = \{f(n) : \text{existujú kladné konštanty } c \text{ a } n_0 \text{ také, že } 0 \leq c\,g(n) \leq f(n) \text{ pre všetky } n \geq n_0\} .$$

Pre všetky hodnoty n, väčšie alebo rovné n_0, je $f(n)$ na alebo nad $g(n)$.

Veta 2.1 *Pre ľubovoľné dve funkcie $f(n)$ a $g(n)$ platí $f(n) = \Theta(g(n))$ vtedy a len vtedy, ak $f(n) = O(g(n))$ a $f(n) = \Omega(g(n))$.*

Asymptotická notácia v rovnostiach

Zavedením O-notácie píšeme napríklad $n = O(n^2)$. Taktiež je možné písať $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Ako máme interpretovať takéto formuly?

Keď sa asymptotická notácia nachádza osamote na pravej strane, ako v $n = O(n^2)$, už sme definovali, že znamienko rovnosti znamená príslušnosť do množiny, t.j. $n \in O(n^2)$. Vo všeobecnosti, keď sa asymptotická notácia nachádza vo formule, interpretujeme ju ako nejakú anonymnú funkciu. Napríklad formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ znamená, že $2n^2 + 3n + 1 = 2n^2 + f(n)$, kde $f(n)$ je nejaká funkcia z množiny $\Theta(n)$.

Používanie asymptotickej notácie týmto spôsobom môže pomôcť eliminovať zo vzťahov nadbytočné detaily. Napríklad, v predchádzajúcom texte sme vyjadrili najhorší čas behu merge sortu ako rekurenciu

$$T(n) = 2T(n/2) + \Theta(n) .$$

Ak za zaoberáme iba asymptotickým správaním $T(n)$, nemá zmysel špecifikovať všetky členy nižšieho rádu.

Počet anonymných funkcií vo výraze sa chápe ako počet výskytov asymptotickej notácie. Napríklad výraz

$$\sum_{i=1}^{n} O(i) ,$$

kde sa nachádza iba jedna anonymná funkcia, nie je to isté ako $O(1) + O(2) + \cdots + O(n)$, čo nemá jasnú interpretáciu.

V niektorých prípadoch, asymptotická notácia sa nachádza na ľavej strane, ako napríklad v

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

Takéto rovnosti interpretujeme pomocou nasledujúceho pravidla: *Nezáleží na tom, ako sú anonymné funkcie zvolené naľavo od znamienka rovnosti, vždy existuje spôsob ako zvoliť anonymné funkcie na pravo od rovnítka, aby rovnosť platila.*

o-notácia
Asymptoticky horné ohraničenie poskytované O-notáciou môže, ale aj nemusí byť asymptoticky

tesné. Ohraničenie $2n^2 = O(n^2)$ je asymptoticky tesné, ale ohraničenie $2n = O(n^2)$ nie je. o-notáciu používame na označenie horného ohraničenia, ktoré nie je asymptoticky tesné. Formálne definujeme $o(g(n))$ ako

$$o(g(n)) = \{f(n) \ : \text{pre každú kladnú konštantu } c > 0, \text{ existuje konštanta } n_0 > 0 \text{ taká, že } 0 \leq f(n) < c\,g(n) \text{ pre všetky } n \geq n_0\} \ .$$

Napríklad, $2n = o(n^2)$, ale $2n^2 \neq o(n^2)$.

Definície O-notácie a o-notácie sú podobné. Hlavný rozdiel je však v tom, že ak $f(n) = O(g(n))$, ohraničenie $0 \leq f(n) \leq cg(n)$ platí pre *nejakú* konštantu $c > 0$, zatiaľčo v $f(n) = o(g(n))$, ohraničenie $0 \leq f(n) < cg(n)$ platí pre *všetky* konštanty $c > 0$. Intuitívne, v o-notácii, funkcia $f(n)$ sa stáva zanedbateľnou v porovnaní s $g(n)$ čím viac sa n blíži nekonečnu, t.j.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \ .$$

ω-notácia

Analogicky, ω-notácia je v podobnom vzťahu ku Ω-notácii ako o-notácia ku O-notácii. ω-notáciu používame na popis dolného ohraničenia, ktoré nie je asymptoticky tesné. Jedným zo spôsobov, ako ju zadefinovať, je

$$f(n) \in \omega(g(n)) \text{ vtedy a len vtedy, ak } g(n) \in o(f(n)) \ .$$

Formálne však $\omega(g(n))$ definujeme ako množinu

$$\omega(g(n)) = \{f(n) \ : \text{pre každú kladnú konštantu } c > 0, \text{ existuje konštanta } n_0 > 0 \text{ taká, že } 0 \leq c\,g(n) < f(n) \text{ pre všetky } n \geq n_0\} \ .$$

Napríklad $n^2/2 = \omega(n)$, ale $n^2/2 \neq \omega(n^2)$. Zo vzťahu $f(n) = \omega(g(n))$ vyplýva, že

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \ ,$$

za predpokladu, že limita existuje. Teda $f(n)$ neohraničene rastie v porovnaní s $g(n)$ čím viac sa n blíži nekonečnu.

Porovnanie funkcií

Mnoho relačných vlastností reálnych čísel sa vzťahuje aj na asymptotické porovnávania. Predpokladajme teraz, že $f(n)$ a $g(n)$ sú asymptoticky kladné.

Tranzitívnosť:

$$\begin{array}{llllll}
f(n) = \Theta(g(n)) & \text{a} & g(n) = \Theta(h(n)) & \text{implikuje} & f(n) = \Theta(h(n)) \ , \\
f(n) = O(g(n)) & \text{a} & g(n) = O(h(n)) & \text{implikuje} & f(n) = O(h(n)) \ , \\
f(n) = \Omega(g(n)) & \text{a} & g(n) = \Omega(h(n)) & \text{implikuje} & f(n) = \Omega(h(n)) \ , \\
f(n) = o(g(n)) & \text{a} & g(n) = o(h(n)) & \text{implikuje} & f(n) = o(h(n)) \ , \\
f(n) = \omega(g(n)) & \text{a} & g(n) = \omega(h(n)) & \text{implikuje} & f(n) = \omega(h(n)) \ .
\end{array}$$

Reflexivita:

$$\begin{array}{l}
f(n) = \Theta(f(n)) \ , \\
f(n) = O(f(n)) \ , \\
f(n) = \Omega(f(n)) \ .
\end{array}$$

Symetria:

$$f(n) = \Theta(g(n)) \quad \text{vtedy a len vtedy, ak} \quad g(n) = \Theta(f(n)) \ .$$

Obrátená symetria:

$$f(n) = O(g(n)) \quad \text{vtedy a len vtedy, ak} \quad g(n) = \Omega(f(n)) \ ,$$
$$f(n) = o(g(n)) \quad \text{vtedy a len vtedy, ak} \quad g(n) = \omega(f(n)) \ .$$

Možno si všimnúť analógiu medzi asymptotickým porovnaním dvoch funkcií f a g a porovnaním dvoch reálnych čísel a a b:

$$\begin{aligned} f(n) = O(g(n)) &\approx a \leq b \ , \\ f(n) = \Omega(g(n)) &\approx a \geq b \ , \\ f(n) = \Theta(g(n)) &\approx a = b \ , \\ f(n) = o(g(n)) &\approx a < b \ , \\ f(n) = \omega(g(n)) &\approx a > b \ . \end{aligned}$$

Toto nás však nesmie viesť ku zovšeobecneniu. Napríklad trichotómia [6], ktorá platí pre reálne čísla, neplatí pre porovnávanie v asymptotickej notácii. Taktiež, všetky reálne čísla možno navzájom porovnať, no napríklad funkcie n a $n^{1+\sin n}$ sú navzájom neporovnateľné.

2.1.2 Štandardné zápisy a funkcie

V nasledujúcom bude podaný prehľad štandardných matematických funkcií a zápisov plus vysvetlenie vzťahov medzi nimi.

Monotónnosť

Funkcia $f(n)$ je ***rastúca***, ak z $m \leq n$ vyplýva $f(m) \leq f(n)$. Podobne, $f(n)$ je ***klesajúca***, ak z $m \leq n$ vyplýva $f(m) \geq f(n)$. Funkcia $f(n)$ je ***rýdzo rastúca*** ak z $m < n$ vyplýva $f(m) < f(n)$ a ***rýdzo klesajúca***, ak z $m < n$ vyplýva $f(m) > f(n)$.

Celé časti

Pre každé reálne číslo x označíme najväčšie celé číslo menšie alebo rovné x symbolom $\lfloor x \rfloor$ a najmenšie celé číslo väčšie alebo rovné x symbolom $\lceil x \rceil$. Pre všetky reálne čísla x máme

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 \ .$$

Pre každé celé číslo n platí

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

a pre každé celé číslo n a celé čísla $a \neq 0$ a $b \neq 0$ je

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \qquad \text{a} \qquad \lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor \ . \tag{1}$$

Polynómy

Nech d je prirodzené číslo. ***Polynóm stupňa d premennej n*** je funkcia $p(n)$ tvaru

$$p(n) = \sum_{i=0}^{d} a_i n^i \ ,$$

kde konštanty $a_0, a_1, \ldots, a_d$ sú ***koeficienty*** polynómu, pričom $a_d \neq 0$. Polynóm je ***asymptoticky kladný***, ak $a_d > 0$. Pre asymptoticky kladný polynóm $p(n)$ stupňa d máme $p(n) = \Theta(n^d)$. Pre ľubovoľnú reálnu konštantu $a \geq 0$ je funkcia n^a rastúca a pre ľubovoľnú reálnu konštantu $a \leq 0$ je funkcia n^a klesajúca. Hovoríme, že funkcia $f(n)$ je ***polynomiálne ohraničená***, ak $f(n) = n^{O(1)}$, čo je ekvivalentné tvrdeniu, že $f(n) = O(n^k)$ pre nejakú konštantu k.

[6] Pre ľubovoľné dve reálne čísla a a b platí práve jedno z nasledujúcich: $a < b$, $a = b$ alebo $a > b$.

Exponenciálne funkcie

Pre všetky reálne čísla $a \neq 0$, m a n platia nasledujúce identity:

$$\begin{aligned} a^0 &= 1\,, & a^1 &= a\,, \\ a^{-1} &= 1/a\,, & (a^m)^n &= a^{mn}\,, \\ (a^m)^n &= (a^n)^m\,, & a^m a^n &= a^{m+n}\,. \end{aligned}$$

Pre každé $a \geq 1$ je funkcia a^n v premennej n rastúca.

Rýchlosť rastu polynómov a exponenciálnych funkcií porovnáva nasledujúce tvrdenie. Pre všetky reálne konštanty $a > 1$ a b platí

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0\,, \tag{2}$$

z čoho možno vyvodiť, že $n^b = o(a^n)$. Teda každá exponenciálna funkcia rastie rýchlejšie ako ľubovoľný polynóm.

Pre každé reálne číslo x máme

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}\,,$$

kde „!“ označuje funkciu faktoriál, ktorá bude definovaná ďalej. Pre každé reálne číslo x platí nerovnosť

$$e^x \geq 1 + x\,,$$

pričom rovnosť nastáva iba v prípade, keď $x = 0$. Ak $|x| \leq 1$ platí nasledujúce ohraničenie:

$$1 + x \leq e^x \leq 1 + x + x^2\,.$$

Ak $x \to 0$, aproximácia e^x výrazom $1 + x$ je celkom dobrá:

$$e^x = 1 + x + \Theta(x^2)\,.$$

(V tejto rovnosti bola asymptotická notácia použitá pre správanie v limite $x \to 0$ namiesto $x \to \infty$.) Pre každé x platí:

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x\,.$$

Logaritmy

Budeme používať nasledujúce zápisy:

$$\begin{aligned} \lg n &= \log_2 n && \text{(dvojkový logaritmus)}, \\ \ln n &= \log_e n && \text{(prirodzený logaritmus)}, \\ \lg^k n &= (\lg n)^k && \text{(umocňovanie)}, \\ \lg\lg n &= \lg(\lg n) && \text{(kompozícia)}. \end{aligned}$$

Dôležitou notačnou konvenciou, ktorú si treba osvojiť je, že *logaritmické funkcie sa budú aplikovať len na nasledujúci výraz vo formule*, takže $\lg n + k$ bude znamenať $(\lg n) + k$ a nie $\lg(n + k)$. Pre $n > 0$ a $b > 1$, funkcia $\log_b n$ je rýdzo rastúca.

Pre všetky reálne čísla $a > 0$, $b > 0$, $c > 0$ a n platí

$$\begin{aligned} a &= b^{\log_b a}\,, & \log_c(ab) &= \log_c a + \log_c b\,, \\ \log_b a^n &= n \log_b a\,, & \log_b a &= \frac{\log_c a}{\log_c b}\,, \\ \log_b(1/a) &= -\log_b a\,, & \log_b a &= 1/\log_a b\,, \\ a^{\log_b n} &= n^{\log_b a}\,. \end{aligned}$$

Nakoľko zmena základu logaritmu ovplyvní hodnotu logaritmu iba o nejakú multiplikatívnu konštantu, budeme používať zápis $\lg n$ vždy, keď nás nebudú zaujímať konštantné faktory.

Jednoduchý rozvoj $\ln(1+x)$ do radu za predpokladu, že $|x|<1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

Taktiež platia nasledujúce nerovnosti pre $x > -1$:

$$\frac{x}{1+x} \le \ln(1+x) \le x ,$$

pričom rovnosť nastáva iba v prípade, že $x = 0$.

Hovoríme, že funkcia $f(n)$ je ***polylogaritmicky ohraničená***, ak $f(n) = \lg^{O(1)} n$. Rast polynómov a polylogaritmov môžeme porovnať vykonaním substitúcie $\lg n$ za n a 2^a za a vo vzťahu (2), čo dáva

$$\lim_{n\to\infty} \frac{\lg^b n}{2^{a\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .$$

Z tejto limity možno vyvodiť, že $\lg^b n = o(n^a)$ pre ľubovoľnú konštantu $a > 0$. Teda ľubovoľná polynomiálna funkcia rastie rýchlejšie ako akákoľvek polylogaritmická funkcia.

Faktoriály

Hodnota výrazu $n!$ je definovaná pre celé čísla $n \ge 0$ ako

$$n! = \begin{cases} 1 & \text{ak } n = 0 , \\ n \cdot (n-1)! & \text{ak } n > 0 . \end{cases}$$

Slabé horné ohraničenie faktoriálu je zrejme $n! \le n^n$. ***Stirlingova aproximácia***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n)) ,$$

nám poskytuje užšie horné i dolné ohraničenie. Použitím Stirlingovej aproximácie možno dokázať

$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) . \end{aligned}$$

Pre všetky n taktiež platí nasledujúce ohraničenie :

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \le n! \le \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+(1/12n)} .$$

Iterovaná logaritmická funkcia

Pre iterovaný logaritmus používame označenie $\lg^* n$. Nech funkcia $\lg^{(i)} n$ je definovaná rekurzívne pre prirodzené čísla i ako

$$\lg^{(i)} n = \begin{cases} n & \text{ak } i = 0 , \\ \lg(\lg^{(i-1)} n) & \text{ak } i > 0 \text{ a } \lg^{(i-1)} n > 0 , \\ \text{nedefinovaná} & \text{ak } i > 0 \text{ a } \lg^{(i-1)} n \le 0 \text{ alebo } \lg^{(i-1)} n \text{ je nedefinovaná.} \end{cases}$$

Pozor na odlíšenie práve definovanej funkcie $\lg^{(i)} n$ (logaritmus aplikovaný následne i krát s počiatočným argumentom n) od $\lg^i n$ (logaritmus n umocnený na i). Iterovaná logaritmická funkcia je definovaná ako

$$\lg^* = \min\left\{ i \ge 0 \ : \ \lg^{(i)} n \le 1 \right\} .$$

Iterovaný logaritmus je *veľmi* pomaly rastúca funkcia:
$\lg^* 2 = 1$, $\lg^* 4 = 2$, $\lg^* 16 = 3$, $\lg^* 65536 = 4$, $\lg^* (2^{65536}) = 5$.

Fibonacciho čísla

Fibonacciho čísla sú definované nasledujúcou rekurenciou:

$$\begin{aligned} F_0 &= 0 \,, \\ F_1 &= 1 \,, \\ F_i &= F_{i-1} + F_{i-2} \quad \text{pre } i \geq 2 \,. \end{aligned}$$

Teda každé Fibonacciho číslo je súčtom dvoch predchádzajúcich, čo dáva postupnosť

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$.

Fibonacciho čísla sú úzko späté so ***zlatým rezom*** ϕ a k nemu združeným číslom $\widehat{\phi}$, ktoré sú dané vzťahmi

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} & \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= 1{,}61803\ldots \,, & &= -0{,}61803\ldots \,. \end{aligned}$$

Špeciálne máme

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} \,,$$

čo možno dokázať indukciou. Nakoľko je $|\widehat{\phi}| < 1$, máme $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, takže i-te Fibonacciho číslo F_i je rovné $\phi^i/\sqrt{5}$ po zaokrúhlení na najbližšie celé číslo. To ale znamená, že Fibonacciho čísla rastú exponenciálne.

2.2 Rekurentné vzťahy

Ak algoritmus obsahuje rekurzívne volanie samého seba, často možno čas jeho behu popísať rekurenciou. ***Rekurencia*** je rovnosť alebo nerovnosť, ktorá popisuje funkciu na základe jej hodnôt pre menšie argumenty. V tejto časti poskytneme tri metódy riešenia rekurencií pre určovanie asymptotických „Θ“ alebo „O“ ohraničení riešenia. V ***substitučnej metóde*** odhadujeme hranicu, a potom používame matematickú indukciu na dôkaz, že náš odhad bol správny. ***Iteračná metóda*** konvertuje rekurenciu na sumáciu, a potom sa spolieha na techniky pre ohraničovanie sumácií na vyriešenie rekurencie. ***Master metóda*** poskytuje ohraničenia pre rekurencie tvaru

$$T(n) = a\,T(n/b) + f(n) \,,$$

kde $a \geq 1$, $b > 1$ a $f(n)$ je daná funkcia.

Technické záležitosti

V praxi zvykneme zanedbávať určité technické detaily keď konštruujeme alebo riešime rekurencie. Vhodným príkladom je napríklad predpoklad, že funkcie majú iba celočíselné argumenty. Obyčajne je čas behu $T(n)$ algoritmu definovaný iba ak n je celé číslo. Napríklad rekurencia popisujúca najhorší čas behu procedúry MERGE-SORT je v skutočnosti

$$T(n) = \begin{cases} \Theta(1) & \text{ak } n = 1 \,, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{ak } n > 1 \,. \end{cases} \tag{3}$$

Hraničné podmienky reprezentujú ďalšiu triedu detailov, ktoré obyčajne ignorujeme. Nakoľko čas behu algoritmu na vstupe konštantnej dĺžky je konštanta, takto vzniknuté rekurencie vo všeobecnosti obsahujú rovnosť $T(n) = \Theta(1)$ pre dostatočne malé n. Preto v rekurenciách nebudeme túto skutočnosť explicitne podávať. Hoci sa zmena hodnoty $T(1)$ prejaví na riešení rekurencie, ale toto riešenie sa zvyčajne zmení iba o konštantný faktor, čo je väčšinou nepodstatné.

2.2.1 Substitučná metóda

Pri riešení rekurencií substitučnou metódou musíme najprv odhadnúť tvar riešenia, a potom použitím matematickej indukcie nájsť chýbajúce konštanty, pričom ukážeme, že naše riešenie je správne. Názov tejto metódy pochádza zo substitúcie odhadovaného riešenia za funkciu keď indukčnú hypotézu overujeme pre menšie hodnoty. Možno ju však aplikovať len v prípadoch, keď je ľahké (sme schopní) odhadnúť tvar riešenia.

Substitučnú metódu možno použiť na stanovenie ako horného, tak aj dolného ohraničenia. Ako príklad vezmime horné ohraničenie pre rekurenciu

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ , \tag{4}$$

čo sa podobá napríklad rekurencii (3). Nech teda naše odhadované riešenie je $T(n) = O(n \lg n)$. Teraz je naším cieľom ukázať, že $T(n) \leq cn \lg n$ pre nejakú vhodne zvolenú konštantu $c > 0$. Začneme predpokladom, že ohraničenie platí pre $\lfloor n/2 \rfloor$, t.j., že $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substitúcia do rekurencie dáva:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \ , \end{aligned}$$

pričom posledný krok platí za predpokladu, že $c \geq 1$.

Teraz treba ukázať, že naše riešenie platí aj pre hraničné podmienky, t.j. musíme ukázať, že existuje dostatočne veľká konštanta c taká, že pre medzné podmienky platí ohraničenie $T(n) \leq cn \lg n$.

Zmena premenných

Niekedy je možné, použitím nepatrnej algebraickej manipulácie, prerobiť neznámu rekurenciu na rekurenciu, ktorá nám už je známa. Ako príklad uvažujme rekurenciu

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n \ ,$$

ktorá sa zdá byť zložitá. Môžeme ju zjednodušiť zmenou premenných. Predpokladajme teraz, že hodnoty, s ktorými pracujeme sú celočíselné. Substitúcia $m = \lg n$ dáva

$$T(2^m) = 2T(2^{m/2}) + m \ .$$

Nech teraz $S(m) = T(2^m)$, čím získame rekurenciu

$$S(m) = 2S(m/2) + m \ ,$$

ktorá nápadne pripomína rekurenciu (4) a má to isté riešenie: $S(m) = O(m \lg m)$. Spätnou substitúciou máme $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

2.2.2 Iteračná metóda

Metóda iterovania rekurencie nevyžaduje odhadnúť riešenie, ale môže byť treba vykonať viac operácií ako pri substitučnej metóde. Myšlienkou je expandovanie (iterácia) rekurencie a jej vyjadrenie v tvare sumy výrazov závislých iba od n a počiatočných podmienok.

Ako príklad uvažujme rekurenciu

$$T(n) = 3T(\lfloor n/4 \rfloor) + n \ .$$

Iterujme ju nasledovným spôsobom:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \ , \end{aligned}$$

kde $\lfloor\lfloor n/4\rfloor/4\rfloor = \lfloor n/16\rfloor$ a $\lfloor\lfloor n/16\rfloor/4\rfloor = \lfloor n/64\rfloor$ vyplývajú z (1).

Dokedy musíme iterovať rekurenciu pokým narazíme na hraničnú podmienku? i-ty výraz v rozvoji je $3^i\lfloor n/4^i\rfloor$. Iterácia narazí na $n = 1$ keď $\lfloor n/4^i\rfloor = 1$, t.j. keď i prekročí $\log_4 n$. Pokračovaním v iterácii po tento bod a použitím nerovnosti $\lfloor n/4^i\rfloor \leq n/4^i$ zistíme, že

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n)\ . \end{aligned}$$

Použili sme tu skutočnosť, že $\log_4 3 < 1$, aby sme mohli tvrdiť, že $\Theta(n^{\log_4 3}) = o(n)$.

Iteračná metóda zvyčajne vedie ku množstvu operácií, preto kľúčom k úspešnému zvládnutiu tejto techniky je zameranie sa na dva parametre: počet, koľkokrát treba iterovať rekurenciu na dosiahnutie hraničnej podmienky a súčet výrazov vznikajúcich na každej úrovni iterácie. Niekedy je možné už počas iterovania odhadnúť tvar riešenia – vtedy použijeme substitučnú metódu.

Ak rekurencia obsahuje dolné a horné celé časti, môže sa výpočet veľmi skomplikovať. Často pomôže predpoklad, že rekurencia je definovaná iba pre celočíselné mocniny nejakého čísla.

Rekurzívne stromy

Rekurzívny strom je výhodným nástrojom pre vizualizáciu toho, čo sa deje, keď iterujeme rekurenciu. Je zvlášť vhodný ak rekurencia popisuje divide & conquer algoritmus. Obrázok 1 znázorňuje odvodenie rekurzívneho stromu pre

$$T(n) = 2T(n/2) + n^2\ .$$

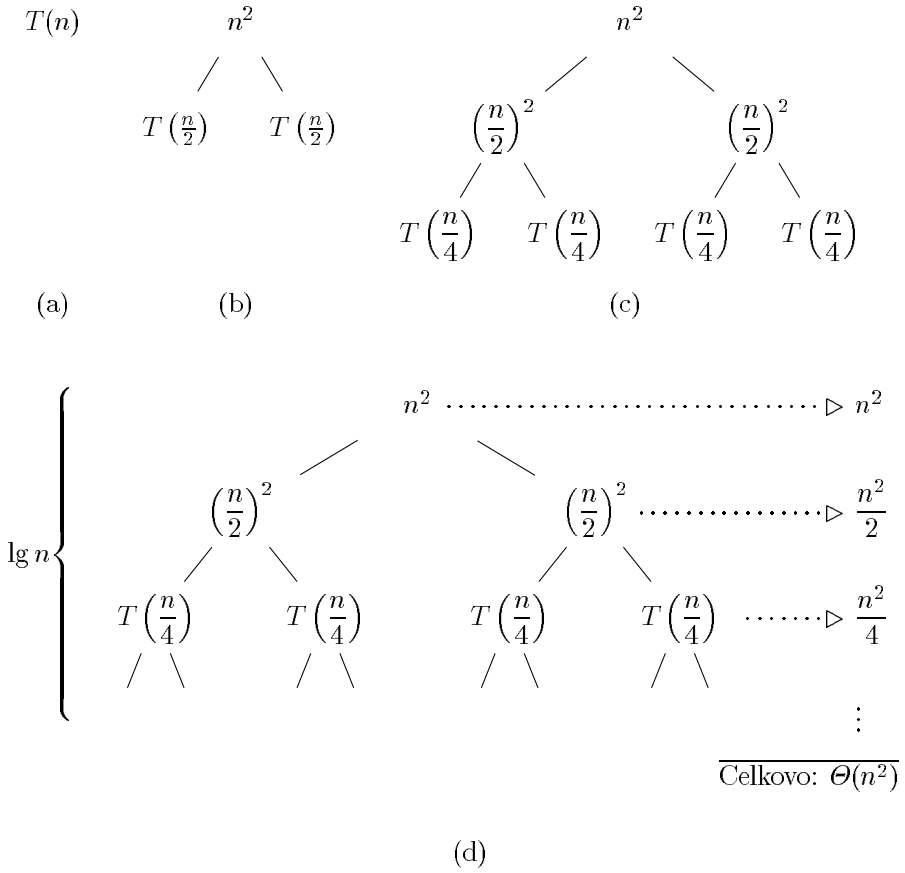
Kvôli jednoduchosti predpokladáme, že n je celočíselná mocnina 2. Časť (a) obrázku znázorňuje $T(n)$, čo v časti (b) už je rozvinuté do ekvivalentného stromu reprezentujúceho rekurenciu. Výraz n^2 je koreň (nároky prvej úrovne rekurzie) a dva podstromy koreňa sú dve menšie rekurencie $T(n/2)$. Časť (c) znázorňuje tento proces ešte o jeden krok ďalej po rozvinutí $T(n/2)$. Nároky oboch poduzlov na druhej úrovni rekurzie sú $(n/2)^2$. Pokračujeme rozvíjaním každého uzla stromu rozdeľovaním podľa rekurencie, pokým nedosiahneme hraničnú podmienku. Časť (d) znázorňuje plne rozvinutý strom.

Teraz vyhodnotíme rekurenciu sčítaním hodnôt na každej úrovni stromu. Vrchná úroveň má celkovú hodnotu n^2, druhá úroveň má $(n/2)^2 + (n/2)^{=} n^2/2$, tretia má hodnotu $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$, atď. Nakoľko tieto hodnoty klesajú geometricky, ich súčet bude väčší ako najväčší (prvý) výraz nanajvýš o nejakú multiplikatívnu konštantu, teda riešenie je $\Theta(n^2)$.

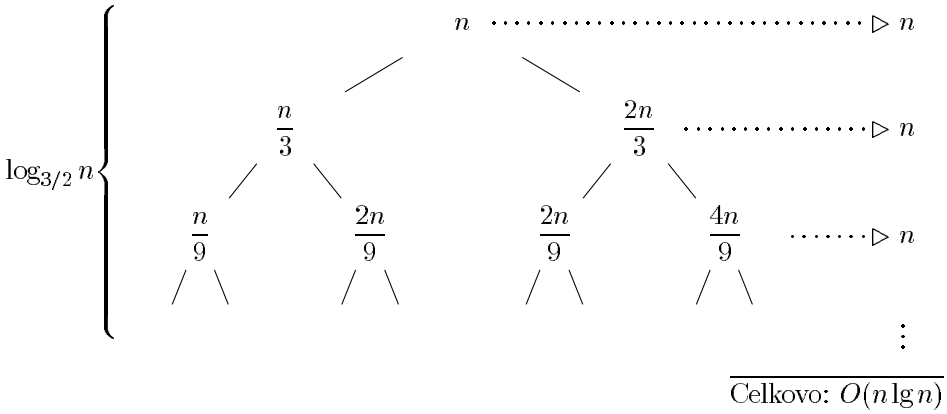
Iným, trochu zložitejším príkladom môže byť na obrázku 2 rekurzívny strom pre

$$T(n) = T(n/3) + T(2n/3) + n\ ,$$

kde opäť pre jednoduchosť vynechávame operácie dolnej a hornej celej časti. Ak sčítame hodnoty na jednotlivých úrovniach rekurzívneho stromu, dostaneme hodnotu n pre každú úroveň. Najdlhšia cesta od koreňa k listu je $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Keďže $(2/3)^k n = 1$ ak $k = \log_{3/2} n$, výška stromu je $\log_{3/2} n$. Teda horným ohraničením riešenia rekurencie je $n\log_{3/2} n = O(n \lg n)$.



Obr. 1: Konštrukcia rekurzívneho stromu pre rekurenciu $T(n) = 2T(n/2) + n^2$. Časť (**a**) znázorňuje funkciu $T(n)$, ktorá je expandovaná v (**b**)–(**d**) pre sformovanie rekurzívneho stromu. Plne rozvinutý strom v časti (**d**) má výšku $\lg n$ (má $\lg n + 1$ úrovní).



Obr. 2: Rekurzívny strom pre rekurenciu $T(n) = T(n/3) + T(2n/3) + n$.

2.2.3 Master metóda

Master metóda poskytuje jednoduchý návod ako riešiť rekurencie tvaru

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)\ , \tag{5}$$

kde $a \geq 1$ a $b > 1$ sú konštanty a $f(n)$ je asymptoticky kladná funkcia. Hoci Master metóda vyžaduje zapamätanie 3 prípadov, jej použitím sa zjednoduší riešenie množstva rekurencií.

Rekurencia (5) popisuje čas behu algoritmu, ktorý rozdeľuje problém veľkosti n na a podproblémov, každý o veľkosti n/b, kde a a b sú nejaké kladné konštanty. Podproblémy sú riešené rekurzívne, každý v čase $T(n/b)$. Nároky na delenie problému a skombinovávanie výsledkov vzniknutých podproblémov sú popisované funkciou $f(n)$.

Veta 2.2 (Master Theorem) *Nech $a \geq 1$, $b > 1$ sú konštanty, nech $f(n)$ je (asymptoticky kladná) funkcia a $T(n)$ je definovaná rekurentne ako nezáporná funkcia*

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)\ ,$$

kde n/b berieme buď ako $\lfloor n/b \rfloor$ alebo ako $\lceil n/b \rceil$. Potom $T(n)$ možno asymptoticky ohraničiť nasledovne:

1. *Ak $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ pre nejakú konštantu $\varepsilon > 0$, potom $T(n) = \Theta(n^{\log_b a})$.*
2. *Ak $f(n) = \Theta\left(n^{\log_b a}\right)$, potom $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.*
3. *Ak $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$, $\varepsilon > 0$ a ak $af(n/b) \leq cf(n)$ pre nejakú konštantu $c < 1$ a pre všetky dostatočne veľké n, potom $T(n) = \Theta(f(n))$.*

3 Triedenie a hľadanie k-teho najmenšieho prvku

Táto časť prezentuje niekoľko algoritmov na riešenie nasledujúceho ***problému triedenia***:

Vstup: Postupnosť n čísel $\langle a_1, a_2, \ldots, a_n \rangle$.

Výstup: Permutácia $\langle a'_1, a'_2, \ldots, a'_n \rangle$ vstupnej postupnosti taká, že $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Vstupná postupnosť je zvyčajne n-prvkové pole, hoci môže byť reprezentovaná aj iným spôsobom, napríklad ako spájaný zoznam.

Štruktúra dát

V praxi sú čísla, ktoré majú byť utriedené, zriedkakedy izolovanými hodnotami. Zvyčajne sú časťami množín dát, ktoré sa nazývajú ***záznamy***. Každý takýto záznam obsahuje ***kľúč*** – hodnotu, podľa ktorej triedime – a zvyšok záznamu pozostávajúci zo ***satelitných dát***. Ak každý záznam obsahuje veľké množstvo satelitných dát, často namiesto premiestňovania celých záznamov permutujeme iba ukazovatele na tieto záznamy.

Tieto implementačné detaily sú tým, čím sa líši algoritmus od programu. Či triedime individuálne čísla alebo veľké záznamy, je to irelevantné pre metódu, ktorou triediaca procedúra určuje utriedené poradie. Teda, ak sa zameriavame na problém triedenia, zvyčajne predpokladáme, že vstup pozostáva iba z čísel.

Triediace algoritmy

Už sme uviedli dva algoritmy, ktoré triedia postupnosti n čísel. Insert sort potrebuje v najhoršom prípade na utriedenie čas $\Theta(n^2)$, hoci je rýchly pre vstupy malých veľkostí. Merge sort mal lepší asymptotický čas behu, $\Theta(n \lg n)$, ale procedúra MERGE, ktorú používa nepracuje in situ. V tejto časti uvedieme ďalšie dva algoritmy, ktoré triedia postupnosti ľubovoľných reálnych čísel.

3.1 Heapsort

3.1.1 Halda

Dátová štruktúra ***(binárna) halda*** je pole, na ktoré možno hľadieť ako na úplný (až na poslednú úroveň, kde je úplný zľava) binárny strom (viď obr. 3). Každý uzol stromu korešponduje s prvkom poľa, ktorý uchováva hodnotu tohto uzla. Pole A, ktoré reprezentuje haldu, je objekt s dvoma atribútami: *length*$[A]$, čo je celkový počet prvkov v poli, a *heap-size*$[A]$, čo je počet prvkov haldy uloženej v tomto poli. T.j., hoci celé pole $A[1\,..\,\mathit{length}[A]]$ môže obsahovať platné hodnoty, žiadny prvok za $A[\mathit{heap\text{-}size}[A]]$, kde $\mathit{heap\text{-}size}[A] \leq \mathit{length}[A]$, nie je prvkom haldy. Koreňom stromu je $A[1]$, otcom i-teho uzla je prvok s indexom PARENT(i), ľavým synom je prvok s indexom LEFT(i) a pravým synom je prvok s indexom RIGHT(i). Tieto funkcie možno vypočítať jednoducho:

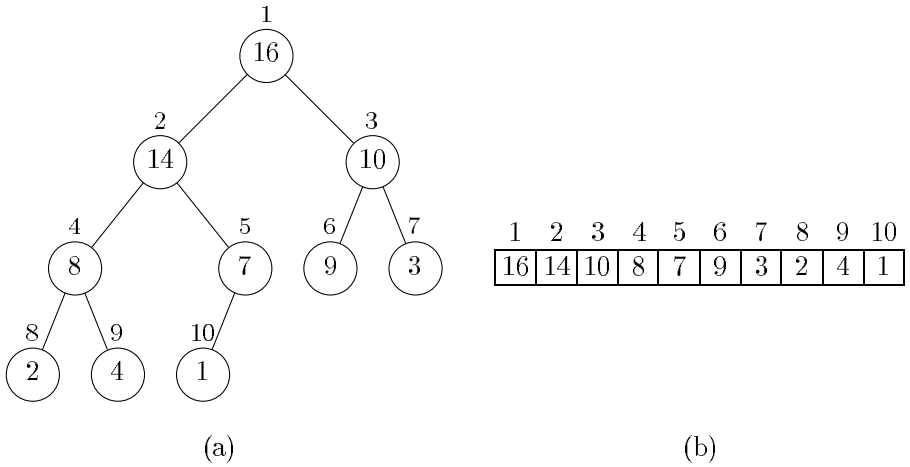
```
PARENT(i)              LEFT(i)              RIGHT(i)
1  return ⌊i/2⌋        1  return 2i         1  return 2i + 1
```

Halda musí spĺňať tzv. ***vlastnosť haldy***: pre každý uzol i rôzny od koreňa platí:

$$A[\text{PARENT}(i)] \geq A[i]\ , \tag{6}$$

teda hodnota v uzle je menšia alebo rovná hodnote v jeho otcovi. To znamená, že najväčšia hodnota je v koreni, a podstromy ľubovoľného uzla neobsahujú väčšie hodnoty ako hodnota tohto uzla.

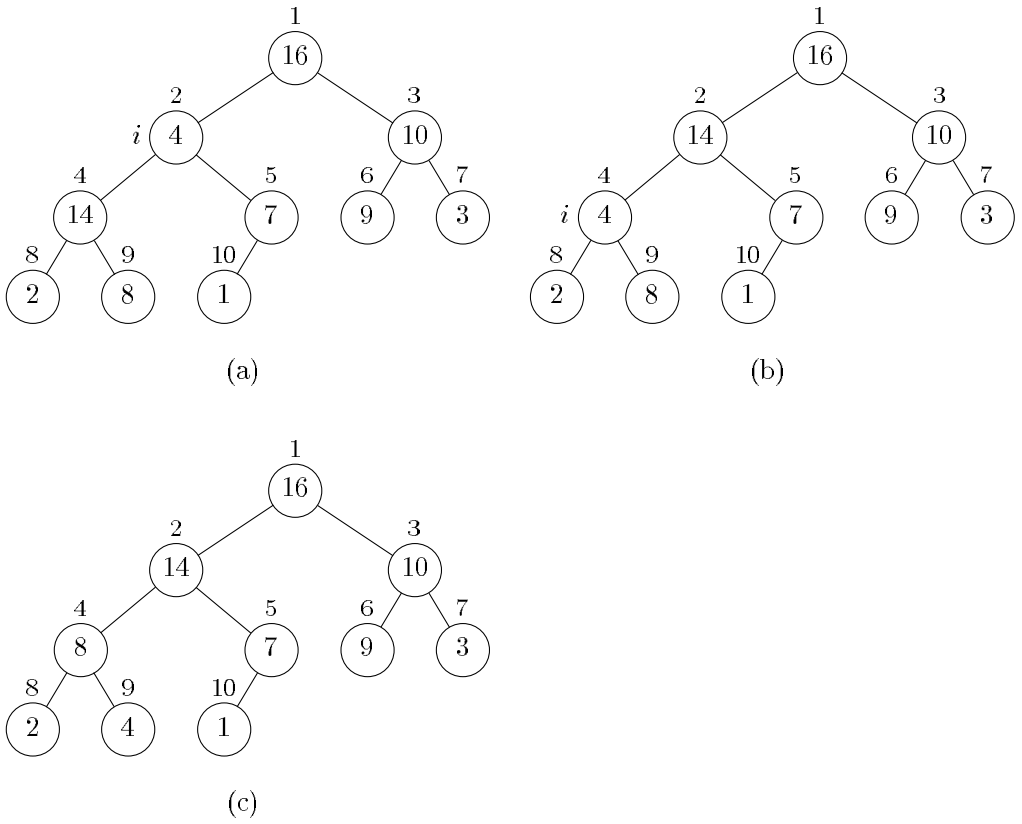
Pod ***výškou*** uzla v strome rozumieme počet hrán na najdlhšej jednoduchej ceste od uzla k listu a výšku stromu definujeme ako výšku jeho koreňa. Výška stromu reprezentujúceho haldu je teda $\Theta(\lg n)$.



Obr. 3: Halda zobrazená ako (**a**) binárny strom, (**b**) ako pole. Čísla v kružniciach sú hodnoty uložené v uzloch stromu a čísla mimo kružníc sú ich prislúchajúce indexy do poľa.

3.1.2 Udržiavanie haldy

HEAPIFY je dôležitá procedúra pre manipuláciu s haldou. Jej vstupom je pole A a index i do tohto poľa. Ak je táto rutina vyvolaná, predpokladá, že binárne stromy s koreňmi LEFT(i) a RIGHT(i) sú haldy, ale $A[i]$ môže mať menšiu hodnotu ako jeho deti, čo porušuje vlastnosť haldy (6). Úlohou HEAPIFY je premiestniť hodnotu v $A[i]$ na také miesto, aby sa podstrom s koreňom i stal haldou.



Obr. 4: Činnosť HEAPIFY$(A, 2)$, kde *heap-size*$[A] = 10$. (**a**) Počiatočná konfigurácia haldy, s hodnotou $A[2]$ porušujúcou vlastnosť haldy. Táto vlastnosť je obnovená pre uzol 2 v (**b**) výmenou $A[2]$ za $A[4]$, čo zároveň narušuje haldu v uzle 4. Rekurzívne vyvolanie HEAPIFY$(A, 4)$ teraz nastaví $i = 4$. Po zámene $A[4]$ s $A[9]$, ako je znázornené v (**c**) je uzol 4 už vporiadku, a rekurzívne vyvolanie HEAPIFY$(A, 9)$ nevyvolá žiadnu zmenu tejto dátovej štruktúry.

HEAPIFY(A, i)
1 $l \leftarrow$ LEFT(i)
2 $r \leftarrow$ RIGHT(i)
3 **if** $(l \leq$ *heap-size*$[A])$ **and** $(A[l] > A[i])$
4 **then** *largest* $\leftarrow l$
5 **else** *largest* $\leftarrow i$
6 **if** $(r \leq$ *heap-size*$[A])$ **and** $(A[r] > A[$*largest*$])$
7 **then** *largest* $\leftarrow r$
8 **if** *largest* $\neq i$ **then**
9 **exchange** $A[i] \leftrightarrow A[$*largest*$]$
10 HEAPIFY$(A,$ *largest*$)$

Obrázok 4 znázorňuje činnosť HEAPIFY. V každom kroku je určený najväčší prvok z $A[i]$, $A[$LEFT$(i)]$ a $A[$RIGHT$(i)]$, a jeho index je uložený do premennej *largest*. Ak hodnota $A[i]$ je najväčšia, potom podstrom s koreňom v uzle i je halda a procedúra končí. Inak najväčšiu hodnotu má jeden zo synov, a preto je $A[i]$ vymenená za $A[$*largest*$]$, čo spôsobí, že uzol i spolu s jeho synmi budú spĺňať vlastnosť haldy. Uzol *largest* má teraz pôvodnú hodnotu $A[i]$, a teda podstrom s koreňom v tomto uzle môže porušovať vlastnosť haldy. Následne musí byť pre tento podstrom rekurzívne vyvolaná procedúra HEAPIFY.

Čas behu procedúry HEAPIFY na podstrome veľkosti n s koreňom v danom uzle i pozostáva z času $\Theta(1)$ potrebného na vzájomnú výmenu prvkov $A[i]$, $A[$LEFT$(i)]$ a $A[$RIGHT$(i)]$ a z času pre

beh HEAPIFY na podstrome s koreňom v jednom zo synov uzla i. Takýto podstrom môže mať veľkosť nanajvýš $2n/3$ – najhorší prípad nastáva, keď posledná úroveň stromu je zaplnená presne do polovice – teda čas behu HEAPIFY možno popísať rekurenciou

$$T(n) \leq T(2n/3) + \Theta(1) \ .$$

Riešenie tejto rekurencie, použitím Master Theoremu, je $T(n) = O(\lg n)$. Tento čas môžeme poprípade charakterizovať ako $O(h)$, kde h je výška príslušného uzla.

3.1.3 Vytváranie haldy

Pre vytvorenie haldy z poľa $A[1\,..\,n]$, kde $n = \mathit{length}[A]$, môžeme použiť procedúru HEAPIFY postupným prechádzaním stromu zdola nahor. Nakoľko všetky prvky v podpoli $A[(\lfloor n/2 \rfloor + 1)\,..\,n]$ sú listy, každý z nich tvorí jednoprvkovú haldu. Procedúra BUILD-HEAP prechádza postupne zvyšné uzly stromu a na každom vykonáva operáciu HEAPIFY. Poradie, v ktorom tieto uzly prechádza, zabezpečuje, aby podstromy s koreňmi v synoch uzla i boli haldy predtým, ako sa na tomto uzle začne vykonávať operácia HEAPIFY.

BUILD-HEAP(A)

```
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1 do
3     HEAPIFY(A, i)
```

Obr. 5 znázorňuje činnosť procedúry BUILD-HEAP.

Jednoduché horné ohraničenie času behu BUILD-HEAP môžeme vypočítať nasledovným spôsobom. Každé volanie HEAPIFY trvá čas $O(\lg n)$ a táto rutina je vyvolaná $O(n)$-krát. Preto celkový čas behu bude nanajvýš $O(n \lg n)$. Táto hranica je síce korektná, ale nie asymptoticky tesná.

Tesnejšie ohraničenie môžeme odvodiť ak si uvedomíme, že čas behu HEAPIFY na danom uzle závisí od jeho výšky v strome, a že výšky väčšiny uzlov sú relatívne malé. Využijeme skutočnosť, že v n-prvkovej halde je nanajvýš $\lceil n/2^{h+1} \rceil$ uzlov výšky h.

Čas potrebný pre beh HEAPIFY na uzle výšky h je $O(h)$, takže celkové nároky procedúry BUILD-HEAP môžeme vyjadriť ako

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Poslednú sumu možno vyhodnotiť s využitím nasledujúcej rovnosti

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \ .$$

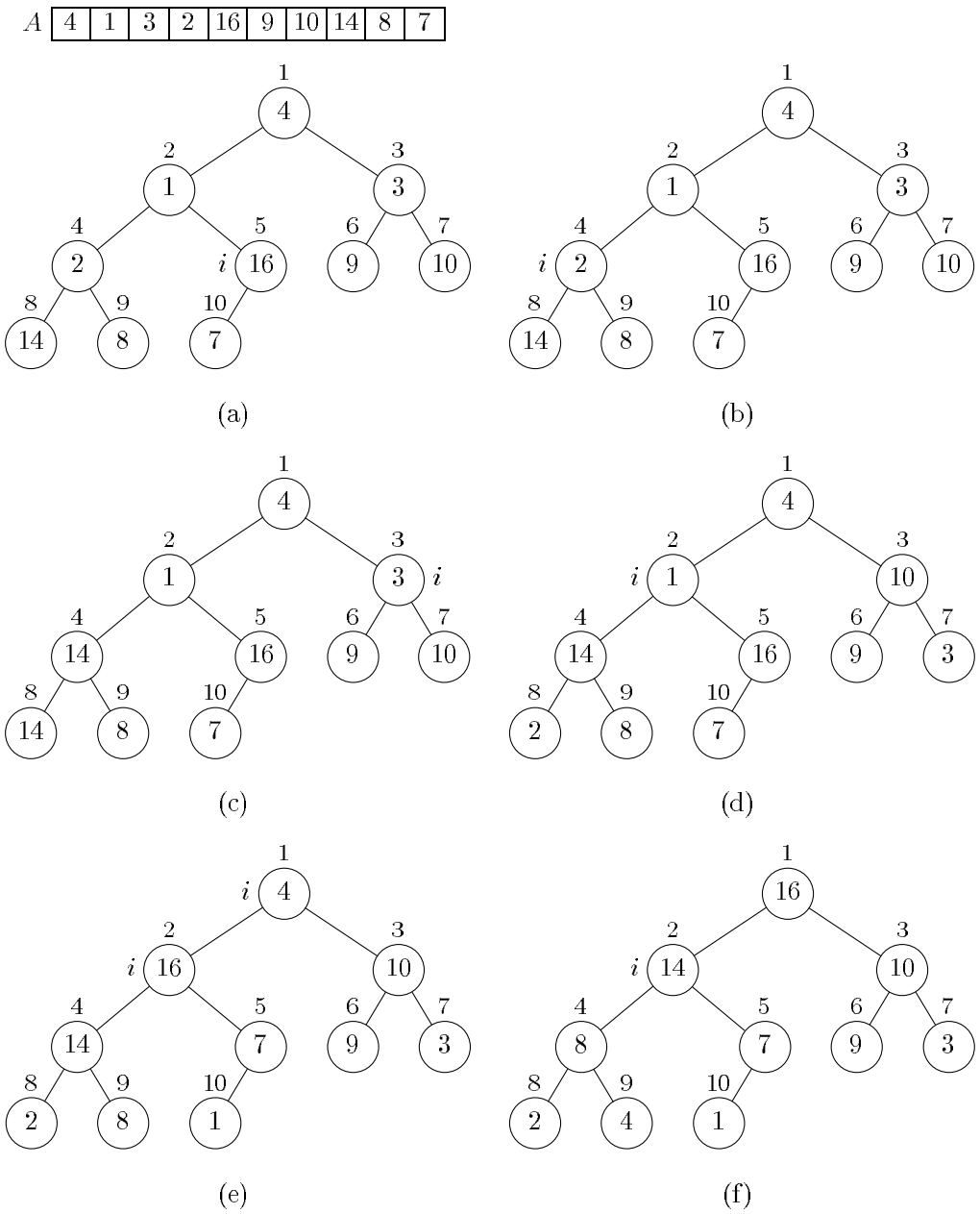
Teda

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n) \ ,$$

čo znamená, že z neusporiadaného poľa sme schopní vytvoriť haldu v *lineárnom* čase.

3.1.4 Algoritmus triedenia heapsort

Algoritmus heapsort najprv pomocou BUILD-HEAP vytvorí zo vstupného poľa $A[1\,..\,n]$ haldu. Nakoľko najväčší prvok poľa je uložený v $A[1]$, možno ho okamžite premiestniť na jeho správnu pozíciu zámenou za $A[n]$. Ak teraz vylúčime uzol n z haldy (znížením hodnoty *heap-size*$[A]$), zistíme, že z $A[1\,..\,(n-1)]$ možno ľahko opäť vytvoriť haldu. Vlastnosť haldy totiž môže porušovať len nový koreň stromu. Všetko, čo treba vykonať na obnovenie vlastnosti haldy, je zavolať HEAPIFY($A, 1$),



Obr. 5: Činnosť BUILD-HEAP. Znázornenie dátovej štruktúry pred vyvolaním HEAPIFY na riadku 3 procedúry BUILD-HEAP. **(a)** 10-prvkové vstupné pole A a binárny strom, ktorý toto pole reprezentuje. Obrázok znázorňuje, že index i ukazuje na uzol 5 pred zavolaním HEAPIFY(A, i). **(b)** Výsledná dátová štruktúra. Index i v ďalšej iterácii ukazuje na uzol 4. **(c)–(e)** Následné iterácie cyklu `for` v BUILD-HEAP. **(f)** Halda po ukončení vykonávania BUILD-HEAP.

čo vytvorí z $A[1..(n-1)]$ opäť haldu. Algoritmus heapsort potom tento proces opakuje až po haldu veľkosti 2.

HEAPSORT(A)
1 BUILD-HEAP(A)
2 **for** $i \leftarrow$ *length*[A] **downto** 2 **do**
3 **exchange** $A[1] \leftrightarrow A[i]$
4 *heap-size*[A] $\leftarrow$ *heap-size*[A] -1
5 HEAPIFY($A, 1$)

Čas behu procedúry HEAPSORT je $O(n \lg n)$, nakoľko BUILD-HEAP trvá $O(n)$ a každé z $n-1$ vyvolaní HEAPIFY trvá $O(\lg n)$.

3.1.5 Prioritná fronta

Prioritná fronta je dátová štruktúra pre uchovávanie množiny S prvkov, z ktorý každých má priradenú hodnotu nazývanú ***kľúč***. Prioritná fronta podporuje nasledujúce operácie:

- INSERT(S, x) vloží prvok x do množiny S ($S \leftarrow S \cup \{x\}$).
- MAXIMUM(S) vráti prvok S s najväčšou hodnotou kľúča ($\max\{x \mid x \in S\}$).
- EXTRACT-MAX(S) vráti prvok S s najväčšou hodnotou kľúča, pričom ho z množiny S odstráni.

Jednou z aplikácií prioritnej fronty je plánovanie činností na zdieľanom počítači. Prioritná fronta udržiava prehľad o činnostiach, ktoré sa majú vykonať, a ich prioritách. Ak nejaká činnosť skončí alebo je prerušená, vyberie sa ďalšia činnosť s najvyššou prioritou použitím EXTRACT-MAX. Novú činnosť možno zaradiť do fronty kedykoľvek použitím INSERT.

Na implementáciu prioritnej fronty možno použiť haldu. Operácia HEAP-MAXIMUM vracia najväčší prvok haldy v čase $\Theta(1)$ jednoducho vrátením hodnoty $A[1]$ z haldy. Kód procedúry HEAP-EXTRACT-MAX sa podobá telu cyklu **for** procedúry HEAPSORT:

HEAP-EXTRACT-MAX(A)
1 **if** *heap-size*[A] < 1 **then**
2 **error** "heap underflow"
3 *max* $\leftarrow A[1]$
4 $A[1] \leftarrow A$[*heap-size*[A]]
5 *heap-size*[A] $\leftarrow$ *heap-size*[A] -1
6 HEAPIFY($A, 1$)
7 **return** *max*

Čas behu HEAP-EXTRACT-MAX je zrejme $O(\lg n)$, nakoľko vykonáva iba o konštantné množstvo práce viac ako HEAPIFY.

Procedúra HEAP-INSERT vkladá do haldy A nový uzol. Najprv haldu rozšíri pridaním nového listu do stromu. Potom od tohto listu až ku koreňu postupne prechádza strom dovtedy, pokým nenájde pre nový prvok správne miesto.

HEAP-INSERT(A, *key*)
1 *heap-size*[A] $\leftarrow$ *heap-size*[A] $+1$
2 $i \leftarrow$ *heap-size*[A]
3 **while** ($i > 1$) **and** (A[PARENT(i)] $<$ *key*) **do**
4 $A[i] \leftarrow A$[PARENT(i)]
5 $i \leftarrow$ PARENT(i)
6 $A[i] \leftarrow$ *key*

Čas behu HEAP-INSERT na n-prvkovej halde je $O(\lg n)$, nakoľko dĺžka cesty, ktorú treba prejsť od nového listu, je $O(\lg n)$.

Teda halda umožňuje implementovať prioritnú frontu takým spôsobom, že čas behu každej z horeuvedených operácií na množine veľkosti n bude $O(\lg n)$.

3.2 Quicksort

Quicksort je triediaci algoritmus, ktorého najhorší čas behu na vstupnom poli n čísel je $\Theta(n^2)$. Napriek tomu je výhodné z hľadiska účinnosti v priemernom prípade: očakávaný čas behu je totiž $\Theta(n \lg n)$, pričom konštantné faktory skryté v Θ-notácii sú v porovnaní s merge sortom veľmi malé.

3.2.1 Popis algoritmu Quicksort

Quicksort, podobne ako merge sort, sa zakladá na paradigme divide & conquer. Divide & conquer proces pre utriedenie podpoľa $A[p..r]$ pozostáva z troch krokov:

Divide: Pole $A[p..r]$ je prerozdelené do dvoch neprázdnych podpolí $A[p..q]$ a $A[q+1..r]$ tak, že každý prvok $A[p..q]$ je menší alebo rovný ako ľubovoľný prvok $A[q+1..r]$. Index q je vypočítavaný počas tejto procedúry.

Conquer: Dve podpolia $A[p..q]$ a $A[q+1..r]$ sú utriedené rekurzívnymi volaniami procedúry QUICKSORT.

Combine: Keďže podpolia sú utriedené na mieste, netreba už nič robiť, celé pole $A[p..r]$ je teraz utriedené.

Nasledujúca procedúra implementuje quicksort.

```
QUICKSORT(A, p, r)
1  if p < r then
2     q ← PARTITION(A, p, r)
3     QUICKSORT(A, p, q)
4     QUICKSORT(A, q + 1, r)
```

Pre utriedenie celého poľa A treba vyvolať QUICKSORT$(A, 1, \mathit{length}[A])$.

Prerozdeľovanie poľa

Kľúčovou časťou algoritmu quicksort je procedúra PARTITION, ktorá preusporiadava podpole $A[p..r]$ in situ.

```
PARTITION(A, p, r)
1  x ← A[p]
2  i ← p − 1
3  j ← r + 1
4  while TRUE do
5     repeat j ← j − 1 until A[j] ≤ x
6     repeat i ← i + 1 until A[i] ≥ x
7     if i < j
8       then exchange A[i] ↔ A[j]
9       else return j
```

Procedúra PARTITION pracuje nasledujúcim spôsobom. Najprv si ako prvok, okolo ktorého bude preusporiadavať pole $A[p..r]$, zvolí $x = A[p]$ [7]. Tento prvok sa nazýva pivot. Potom rozširuje oblasť $A[p..i]$ smerom nahor, resp. oblasť $A[j..r]$ smerom nadol tak, aby každý prvok v $A[p..i]$ bol menší alebo rovný x a každý prvok v $A[j..r]$ bol väčší alebo rovný x. Na začiatku tejto činnosti sú $i = p - 1$ a $j = r + 1$, takže tieto oblasti sú prázdne.

Vnútri tela cyklu `while` na riadkoch 5 a 6 je index j dekrementovaný a index i inkrementovaný pokým nie je $A[i] \geq x \geq A[j]$ [8]. Predpokladajúc, že tieto nerovnosti sú ostré, hodnota $A[i]$ je

[7] Ak by sme namiesto $A[p]$ použili ako pivota $A[r]$, ktorý by bol náhodou zároveň aj najväčším prvkom podpoľa $A[p..r]$, potom by PARTITION vrátila QUICKSORT-u hodnotu $q = r$ a QUICKSORT by sa zacyklil.

[8] Indexy i a j nikdy neprekročia hranice podpoľa $A[p..r]$.

príliš veľká na to, aby patrila do dolnej oblasti a $A[j]$ je príliš malá na to, aby mohla patriť do hornej oblasti. Výmenou $A[i]$ za $A[j]$ (riadok 8) možno tieto oblasti rozšíriť. (Ak nerovnosti nie sú ostré, výmenu možno uskutočniť aj tak.)

Telo cyklu `while` je vykonávané až pokým nie je $i \geq j$, keď celé pole $A[p..r]$ už je rozdelené na dve podpolia $A[p..q]$ a $A[q+1..r]$, kde $p \leq q < r$, tak, že žiadny prvok $A[p..q]$ nie je väčší ako ľubovoľný prvok $A[q+1..r]$. Hodnota $q = j$ je vrátená na konci procedúry.

Čas behu PARTITION na poli $A[p..r]$ je $\Theta(n)$, kde $n = r - p + 1$.

3.2.2 Efektívnosť Quicksortu

Čas behu quicksortu závisí od toho, či je rozdeľovanie vyvážené alebo nevyvážené a od toho, ktorý prvok vyberieme za pivot. Ak je rozdeľovanie vyvážené, algoritmus je asymptoticky taký rýchly ako merge sort. Naopak, ak je rozdeľovanie nevyvážené, môže byť pomalý ako insert sort.

Najhoršie možné rozdeľovanie

Najhoršie správanie quicksortu zrejme nastáva v prípade, ak rozdeľovacia rutina produkuje jeden región pozostávajúci z $n-1$ prvkov a druhý obsahujúci len jeden prvok. Predpokladajme teraz, že nevyvážené rozdeľovanie nastáva na každej úrovni rekurzie algoritmu. Keďže rozdeľovanie trvá čas $\Theta(n)$ a $T(1) = \Theta(1)$, rekurencia pre čas behu je

$$T(n) = T(n-1) + \Theta(n) ,$$

z čoho je vidieť, že $T(n) = \Theta(n^2)$. T.j. v prípade maximálnej nevyváženosti na každej úrovni je čas behu rovný $\Theta(n^2)$, teda nie je lepší ako čas insert sortu. Naviac, tento čas nastáva paradoxne v prípade, keď vstupné pole je už utriedené, čo je bežná situácia, v ktorej čas insert sortu je $O(n)$.

Najlepšie možné rozdeľovanie

Ak rozdeľovacia procedúra vytvára regióny veľkosti $n/2$, quicksort beží omnoho rýchlejšie. Príslušná rekurencia má potom tvar

$$T(n) = 2T(n/2) + \Theta(n) ,$$

čo podľa Master Theoremu má riešenie $T(n) = \Theta(n \lg n)$.

Vyvážené rozdeľovanie

Priemerný čas behu quicksortu je omnoho bližšie ku najlepšiemu prípadu ako ku najhoršiemu.

Predpokladajme napríklad, že rozdeľovacia rutina vždy rozdelí pole na dve časti s pomerom veľkostí 9:1, čo na prvý pohľad vyzerá veľmi nevyvážené. Získame tak rekurenciu

$$T(n) = T(9n/10) + T(n/10) + n$$

popisujúcu čas behu quicksortu. Jej riešením je však opäť čas $\Theta(n \lg n)$. Teda rozdeľovanie s pomerom 9:1 na každej úrovni rekurzie spôsobí, že quicksort je asymptoticky rovnako rýchly ako v najlepšom prípade. Dôvodom je, že ľubovoľné rozdelenie s konštantným pomerom veľkostí vzniknutých regiónov vedie ku rekurzívnemu stromu hĺbky $\Theta(\lg n)$, kde nároky na čas v každej úrovni rekurzie sú $O(n)$.

3.2.3 Znáhodnenie Quicksortu

Pri analýze algoritmov triedenia zvyčajne predpokladáme, že všetky permutácie vstupných čísel môžu nastať s rovnakou pravdepodobnosťou. V skutočnosti však nanešťastie nemôžeme očakávať, že tomu tak bude. Namiesto predpokladania určitej distribúcie vstupov je možné problém riešiť umelou zmenou tejto distribúcie. Predpokladajme napríklad, že predtým ako quicksort začne triediť vstupné dáta, mu ich vždy ***náhodne preusporiadame*** (náhodné preusporiadanie možno uskutočniť v čase $O(n)$). Táto modifikácia nezmení jeho najhorší čas behu, ale spôsobí, že čas behu

bude nezávislý od usporiadania vstupných hodnôt. Takto znáhodnenej verzii quicksortu nemožno predložiť žiaden vstup, ktorý by vždy spôsobil najhoršie správanie tohto algoritmu.

Znáhodnená stratégia je zvyčajne použiteľná v prípadoch keď je mnoho spôsobov, ako môže algoritmus pokračovať, ale je ťažké určiť, ktorý je ten správny. Ak je dobrých mnoho alternatív, náhodný výber alternatívy môže viesť ku dobrej stratégii.

Modifikáciou procedúry PARTITION môžeme získať ďalšiu verziu quicksortu, ktorá používa stratégiu náhodného výberu. Na každej úrovni rekurzie algoritmu, predtým ako je pole rozdelené, ***vymeníme prvok*** $A[p]$ za prvok náhodne zvolený z $A[p..r]$. Takto môžeme očakávať, že vstupné pole bude v priemernom prípade celkom dobre rozdeľované (z hľadiska vyváženosti). Algoritmus náhodne permutujúci vstupné pole pracuje tiež celkom dobre, ale jeho analýza je náročnejšia ako analýza tejto verzie.

Zmeny uskutočnené na procedúrach PARTITION a QUICKSORT sú malé. V novej rozdeľovacej procedúre jednoducho implementujeme výmenu prvkov:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[p] ↔ A[i]
3  return PARTITION(A, p, r)
```

Nový quicksort prerobíme tak, aby volal procedúru RANDOMIZED-PARTITION namiesto PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r then
2     q ← RANDOMIZED-PARTITION(A, p, r)
3     RANDOMIZED-QUICKSORT(A, p, q)
4     RANDOMIZED-QUICKSORT(A, q + 1, r)
```

3.2.4 Analýza Quicksortu

V tejto časti sa budeme hlbšie zaoberať správaním quicksortu. Začneme analýzou najhoršieho prípadu pre beh procedúr QUICKSORT a RANDOMIZED-QUICKSORT, a rozoberieme priemerný prípad pre beh procedúry RANDOMIZED-QUICKSORT.

Analýza najhoršieho prípadu

Vieme, že najhoršie rozdelenie na každej úrovni rekurzie spôsobí, že čas behu quicksortu bude $\Theta(n^2)$, čo je intuitívne aj najhorší čas behu tohto algoritmu.

Použitím substitučnej metódy môžeme ukázať, že čas behu quicksortu je $O(n^2)$. Nech $T(n)$ je najhorší čas behu procedúry QUICKSORT na vstupe veľkosti n. Máme teda rekurenciu

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) \,, \tag{7}$$

kde parameter q sa pohybuje od 1 po $n-1$, keďže procedúra PARTITION produkuje dve oblasti, každá o veľkosti minimálne 1. Odhadujeme, že $T(n) \le cn^2$ pre nejakú konštantu c. Substitúciou tohto odhadu do (7) získame

$$\begin{aligned} T(n) &\le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) \\ &= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n) \,. \end{aligned}$$

Výraz $q^2 + (n-q)^2$ dosahuje maximum pre $q = 1$ alebo $q = n-1$. Z toho teda máme ohraničenie $\max_{1 \le q \le n-1}(q^2 + (n-q)^2) \le 1^2 + (n-1)^2 = n^2 - 2(n-1)$. Pokračovaním v ohraničovaní $T(n)$ získavame

$$\begin{aligned} T(n) &\le cn^2 - 2c(n-1) + \Theta(n) \\ &\le cn^2 \,, \end{aligned}$$

keďže môžeme zvoliť konštantu c dostatočne veľkú, aby rozdiel $\Theta(n) - 2c(n-1)$ bol záporný. Teda najhorší čas behu quicksortu je $\Theta(n^2)$.

Analýza priemerného prípadu

Už sme podali intuitívny argument, prečo priemerný čas behu procedúry RANDOMIZED-QUICKSORT je $\Theta(n \lg n)$. Na to, aby sme mohli analyzovať očakávaný čas behu tohto algoritmu, musíme najprv dobre pochopiť prácu rozdeľujúcej procedúry. Až potom môžeme zostaviť rekurenciu pre priemerný čas potrebný na utriedenie n-prvkového poľa a následne stanoviť ohraničenia pre očakávaný čas behu.

Analýza rozdeľovania

Keď sa vyvoláva PARTITION na riadku 3 procedúry RANDOMIZED-PARTITION, prvok $A[p]$ už je vymenený s náhodným prvkom z $A[p..r]$. Pre zjednodušenie analýzy predpokladajme, že všetky vstupné čísla sú navzájom rôzne. Ak by v triedenom poli bolo viacero navzájom rovnakých prvkov, priemerný čas behu quicksortu by bol aj tak $O(n \lg n)$, ale analýza by bola náročnejšia.

Najprv si všimnime, že hodnota q vrátená procedúrou PARTITION závisí iba na ranku čísla $x = A[p]$ medzi prvkami $A[p..r]$. (***Rank*** čísla x v množine je počet prvkov tejto množiny, ktoré sú menšie alebo rovné x.) Ak položíme $n = r - p + 1$ rovné počtu prvkov $A[p..r]$, výmena $A[p]$ za náhodný prvok z $A[p..r]$ vedie ku pravdepodobnosti $1/n$, že $\text{rank}(x) = i$ pre $i = 1, 2, \ldots, n$, kde $x = A[p]$.

Teraz vypočítame pravdepodobnosti rozličných výsledkov prerozdeľovania. Ak $\text{rank}(x) = 1$, potom v prvom prechode cyklu `while` na riadkoch 4 až 9 procedúry PARTITION sa index i zastaví na $i = p$ a index j sa zastaví taktiež na $j = p$. Teda, keď je vrátené $q = j$, dolná časť obsahuje jediný prvok $A[p]$. Tento prípad nastáva s pravdepodobnosťou $1/n$, čo je pravdepodobnosť, že $\text{rank}(x) = 1$.

Ak $\text{rank}(x) \geq 2$, potom existuje aspoň jeden prvok menší ako $x = A[p]$. Následne, v prvom priechode cyklu `while`, sa index i zastaví na $i = p$, ale j sa zastaví pred dosiahnutím p. Výmena sa potom uskutoční, aby sa $A[p]$ umiestnilo v hornej časti. Až sa vykonávanie PARTITION skončí, každý z $\text{rank}(x) - 1$ prvkov v dolnej časti je ostro menší ako x. Teda, ak $\text{rank}(x) \geq 2$, pre každé $i = 1, 2, \ldots, n-1$ je pravdepodobnosť, že dolná časť obsahuje i prvkov, rovná $1/n$.

Skombinovaním týchto dvoch prípadov dostávame, že veľkosť $q - p + 1$ dolnej časti rozdelenia je 1 s pravdepodobnosťou $2/n$ a že veľkosť je i s pravdepodobnosťou $1/n$ pre $i = 2, 3, \ldots, n-1$.

Rekurencia pre priemerný prípad

Nech $T(n)$ je priemerný čas potrebný na utriedenie n-prvkového vstupného poľa. Vyvolanie RANDOMIZED-QUICKSORT na 1-prvkovom poli trvá konštantný čas, teda máme $T(1) = \Theta(1)$. Na rozdelenie poľa $A[1..n]$ je potrebný čas $\Theta(n)$. Ak procedúra PARTITION vráti index q, potom je RANDOMIZED-QUICKSORT vyvolaný pre podpolia dĺžok q a $n - q$. Priemerný čas pre utriedenie poľa dĺžky n možno vyjadriť ako

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) \; . \tag{8}$$

Hodnota q má približne uniformné rozdelenie s výnimkou toho, že $q = 1$ je dvakrát tak pravdepodobné ako ostatné prípady. Použitím skutočnosti, že $T(1) = \Theta(1)$ a $T(n-1) = O(n^2)$, z analýzy najhoršieho prípadu máme

$$\frac{1}{n}(T(1) + T(n-1)) = \frac{1}{n}(\Theta(1) + O(n^2)) = O(n) \; ,$$

preto výraz $\Theta(n)$ vo vzťahu (8) môže absorbovať výraz $\frac{1}{n}(T(1) + T(n-1))$. Rekurencia (8) teraz nadobúda tvar

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) \; . \tag{9}$$

Možno si všimnúť, že pre $k = 1, 2, \ldots, n-1$, každý výraz $T(k)$ sumy sa objaví raz ako $T(q)$ a druhý raz ako $T(n-q)$. Zlúčením týchto dvoch výrazov dostávame

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \,. \tag{10}$$

Riešenie rekurencie

Rekurenciu (10) možno vyriešiť použitím substitučnej metódy. Induktívne predpokladajme, že $T(n) \leq an\lg n + b$ pre nejaké konštanty $a > 0$ a $b > 0$, ktoré treba určiť. Sme schopní vybrať a a b dostatočne veľké tak, aby $an\lg n + b$ bolo väčšie ako $T(1)$. Potom pre $n > 1$ použitím substitúcie máme

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\leq \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n) \,. \end{aligned}$$

Neskôr ukážeme, že sumu v poslednom riadku možno ohraničiť

$$\sum_{k=1}^{n-1} k\lg k \leq \frac{1}{2}n^2\lg n - \frac{1}{8}n^2 \,. \tag{11}$$

Použitím tohto ohraničenia dostávame

$$\begin{aligned} T(n) &\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\leq an\lg n - \frac{a}{4}n + 2b + \Theta(n) \\ &= an\lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\ &\leq an\lg n + b \,, \end{aligned}$$

keďže môžeme zvoliť a dostatočne veľké tak, že $\frac{a}{4}n$ bude dominovať nad $\Theta(n) + b$. Z uvedeného vyplýva, že priemerný čas behu quicksortu je $O(n\lg n)$.

Tesné ohraničenie sumy

Zostáva dokázať platnosť ohraničenia (11) sumy

$$\sum_{k=1}^{n-1} k\lg k \,.$$

Keďže každý výraz je nanajvýš rovný $n\lg n$, máme triviálne ohraničenie

$$\sum_{k=1}^{n-1} k\lg k \leq n^2\lg n \,.$$

Toto ohraničenie však nepostačuje na vyriešenie horeuvedenej rekurencie ako $T(n) = O(n\lg n)$, potrebovali by sme totiž ohraničenie $\frac{1}{2}n^2\lg n - \Omega(n^2)$.

Rozdeľme sumu na dve časti

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \ .$$

Výraz $\lg k$ v prvej sume na pravej strane je možné ohraničiť $\lg(n/2) = \lg n - 1$. Ten istý výraz, ale v druhej sume možno ohraničiť $\lg n$. Teda

$$\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &\leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leq \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \ ,
\end{aligned}$$

ak $n \geq 2$. Toto je ohraničenie (11).

3.3 Triedenie v lineárnom čase

Už sme uviedli niekoľko algoritmov, ktoré triedia n čísel v čase $O(n \lg n)$. Každému z týchto algoritmov sme schopní predložiť takú postupnosť n vstupných čísel, že čas jeho behu bude $\Omega(n \lg n)$.

Všetky tieto algoritmy majú jedno spoločné: *poradie utriedených prvkov určujú iba na základe vzájomného porovnávania vstupných prvkov.* Takéto algoritmy triedenia nazývame ***triedenia porovnávaním***.

V ďalšom ukážeme, že každé triedenie porovnávaním musí v najhoršom prípade na utriedenie postupnosti n čísel vykonať $\Omega(n \lg n)$ porovnaní. Teda merge sort a heapsort sú asymptoticky optimálne, pričom neexistuje žiadne triedenie porovnávaním, ktoré by bolo rýchlejšie viac ako o konštantný faktor.

V nasledujúcom uvedieme tri triediace algoritmy – counting sort, radix sort a bucket sort – ktoré bežia v lineárnom čase. Netreba pripomínať, že tieto algoritmy používajú iné operácie ako porovnávanie na stanovenie utriedeného poradia.

3.3.1 Dolné ohraničenie pre triedenia porovnávaním

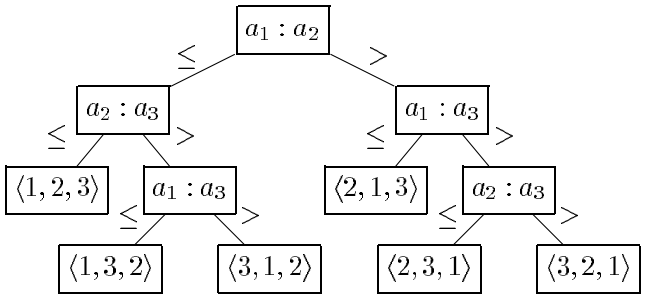
V triedeniach porovnávaním používame na určenie utriedeného poradia vstupnej postupnosti $\langle a_1, a_2, \ldots, a_n \rangle$ *iba* porovnávanie prvkov. T.j. pre dané dva prvky a_i a a_j pre určenie ich relatívneho poradia vykonáme jeden z testov $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ alebo $a_i > a_j$.

Bez újmy na všeobecnosti môžeme predpokladať, že vstupné prvky sú navzájom rôzne. Tým sa porovnávania tvaru $a_i = a_j$ stávajú bezpredmetnými. Taktiež je vhodné poznamenať, že porovnania $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, $a_i < a_j$ sú z hľadiska poskytovanej informácie o relatívnom poradí a_i a a_j ekvivalentné. Preto budeme predpokladať, že všetky porovnania majú tvar $a_i \leq a_j$.

Model rozhodovacieho stromu

Na triedenia porovnávaním možno hľadieť ako na ***rozhodovacie stromy***. Rozhodovací strom reprezentuje porovnania vykonané triediacim algoritmom operujúcim na vstupe danej veľkosti. Riadenie, presun dát a všetky ďalšie aspekty algoritmu sú ignorované. Obrázok 6 znázorňuje rozhodovací strom prislúchajúci insert sortu pracujúcemu na vstupnej postupnosti pozostávajúcej z troch prvkov.

V rozhodovacom strome je každý vnútorný uzol označený $a_i : a_j$ pre nejaké i a j z rozsahu $1 < i, j \leq n$, kde n je počet prvkov vo vstupnej postupnosti. Každý list je označený permutáciou



Obr. 6: Rozhodovací strom pre insert sort operujúci na troch prvkoch. Možných permutácií vstupných prvkov je $3! = 6$, teda rozhodovací strom musí mať aspoň 6 listov.

$\langle \pi(1), \pi(2), \ldots, \pi(n)\rangle$. Vykonávanie triediaceho algoritmu prislúcha prechádzaniu cesty od koreňa rozhodovacieho stromu k jeho listu. Na každom vnútornom uzle je vykonané porovnanie $a_i \leq a_j$. Keď sa dostaneme na list, triediaci algoritmus ukončil usporiadavanie utriedenou postupnosťou $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Každá z $n!$ permutácií n prvkov sa musí objaviť ako jeden z listov rozhodovacieho stromu triediaceho algoritmu.

Dolné ohraničenie najhoršieho prípadu

Dĺžka najdlhšej cesty od koreňa stromu k ľubovoľnému jeho listu reprezentuje najväčší počet porovnaní, ktoré triediaci algoritmus vykonáva na jednom vstupe. Dolné ohraničenie výšok rozhodovacích stromov je teda dolným ohraničením času behu ľubovoľného algoritmu triedenia porovnávaním.

Veta 3.1 *Každý rozhodovací strom pre triedenie n prvkov má výšku $\Omega(n \lg n)$.*

Dôkaz. Uvažujme rozhodovací strom výšky h na triedenie n prvkov. Keďže z n prvkov možno vytvoriť $n!$ permutácií, z ktorých každá reprezentuje jedno preusporiadanie, strom musí obsahovať $n!$ listov. Nakoľko strom výšky h nemá viac ako 2^h listov, máme

$$n! \leq 2^h \ ,$$

resp. po zlogaritmovaní

$$h \geq \lg(n!) \ .$$

Využitím Stirlingovej aproximácie dostávame

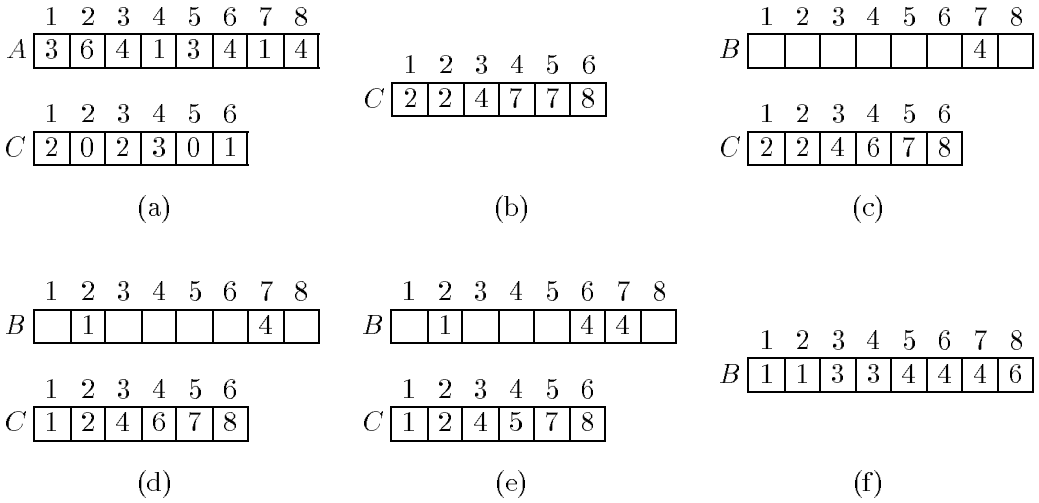
$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n = n\lg n - n\lg e \\ &= \Omega(n \lg n) \ . \quad \square \end{aligned}$$

Dôsledok 3.1 *Heapsort a merge sort sú asymptoticky optimálne triedenia.*

3.3.2 Counting sort

Counting sort predpokladá, že každý z n vstupných prvkov je celé číslo z intervalu $1..k$ pre nejakú celočíselnú konštantu k. Ak $k = O(n)$, triedenie beží v čase $O(n)$.

Základnou myšlienkou tohto triedenia je pre každý vstupný prvok x určiť počet prvkov menších ako x. Táto informácia môže byť použitá pre priame umiestnenie prvku x na správnu pozíciu vo výstupnom poli. Napríklad, ak je ako x menších 17 prvkov, potom x patrí na výstupnú pozíciu 18. Túto schému treba jemne modifikovať pre prípad, že niekoľko prvkov má tú istú hodnotu, keďže ich nechceme všetky umiestniť na to isté miesto.



Obr. 7: Činnosť counting sortu na vstupnom poli $A[1..8]$, kde všetky prvky A sú kladné a menšie alebo rovné $k = 6$. (**a**) Pole A a pomocné pole C po vykonaní riadku 4. (**b**) Pole C po riadku 7. (**c**)–(**e**) Výstupné pole B a pomocné pole C po jednej, dvoch, resp. troch iteráciách cyklu na riadkoch 9–11. (**f**) Utriedené výstupné pole B.

V kóde counting sortu predpokladáme, že vstupom je pole $A[1..n]$, a teda $\mathit{length}[A] = n$. Potrebujeme ďalšie dve polia: pole $B[1..n]$ pre utriedený výstup a pole $C[1..k]$ ako dočasný pracovný priestor.

COUNTING-SORT(A, B, k)

```
1  for i ← 1 to k do
2     C[i] ← 0
3  for j ← 1 to length[A] do
4     C[A[j]] ← C[A[j]] + 1
5  ▷ C[i] teraz obsahuje počet výskytov prvku i v A.
6  for i ← 2 to k do
7     C[i] ← C[i] + C[i − 1]
8  ▷ C[i] teraz obsahuje počet prvkov menších alebo rovných i.
9  for j ← length[A] downto 1 do
10    B[C[A[j]]] ← A[j]
11    C[A[j]] ← C[A[j]] − 1
```

Činnosť counting sortu je znázornená na obrázku 7. Na riadkoch 3–4 po inicializácii vyšetríme každý prvok. Ak hodnota vstupného prvku je i, inkrementujeme $C[i]$. Po vykonaní kódu na riadkoch 3–4 $C[i]$ obsahuje počet vstupných prvkov rovných i pre $i = 1, 2, \ldots, k$. Na riadkoch 6–7 pre každé $i = 1, 2, \ldots, k$ určíme, koľko vstupných prvkov je menších alebo rovných i. Nakoniec, na riadkoch 9–11 umiestnime každý prvok $A[j]$ na jeho správnu pozíciu vo výstupnom poli B. Ak je všetkých n prvkov navzájom rôznych, potom pri prvom vstupe na riadok 9 je pre každé $A[j]$ hodnota $C[A[j]]$ správnou pozíciou $A[j]$ vo výstupnom poli, nakoľko $C[A[j]]$ prvkov je menších alebo rovných $A[j]$. Keďže ale prvky nemusia byť navzájom rôzne, hodnotu $C[A[j]]$ dekrementujeme vždy, keď umiestnime hodnotu $A[j]$ do poľa B, čo zabezpečí, aby ďalší vstupný prvok s hodnotou rovnou $A[j]$, ak taký existuje, bol umiestnený na pozíciu pred $A[j]$ vo výstupnom poli.

Cykly `for` na riadkoch 1–2 a 6–7 trvajú $O(k)$ a cykly na riadkoch 3–4 a 9–11 trvajú $O(n)$, teda celkový čas behu counting sortu je $O(k + n)$. V praxi zvyčajne counting sort používame ak $k = O(n)$, keď tento čas je $O(n)$. Counting sort teda prekonáva dolnú hranicu $\Omega(n \lg n)$ triedení porovnávaním.

Dôležitou vlastnosťou counting sortu je, že je to ***stabilné triedenie***: čísla s rovnakou hodnotou

sa objavia vo výstupnom poli v rovnakom poradí ako vo vstupnom poli. Stabilita je dôležitá ak spolu s kľúčovými hodnotami preusporiadavame aj satelitné dáta.

3.3.3 Radix sort

Radix sort rieši problém triedenia čísel možno trochu kontraintuitívne, utriedením najprv podľa poslednej číslice. Vstupná postupnosť sa takto rozdelí na desať podpostupností, v ktorých všetky čísla končia rovnakou číslicou. Každú z takto vzniknutých postupností opäť utriedime, ale už podľa predposlednej číslice. Tento proces bude pokračovať až pokým čísla takto neutriedime podľa všetkých číslic. (Samozrejme, ak čísla majú rôzne dlhé zápisy, podľa potreby ich zľava doplníme nulami.) Je veľmi dôležité, aby triediaci algoritmus, ktorý používame na triedenie podľa jednotlivých cifier, bol *stabilný*, inak by totiž algoritmus radix sort nesprávne triedil. Činnosť radix sortu je znázornená na obrázku 8.

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

Obr. 8: Činnosť radix sortu na zozname siedmych trojciferných čísel. Prvý stĺpec je vstup. Zvyšné stĺpce znázorňujú postupné utrieďovanie podľa jednotlivých pozícií číslic od cifry najnižšieho rádu po cifru najvyššieho rádu, čo indikujú vertikálne šípky.

Radix sort je občas používaný na triedenie záznamov, ktoré ako kľúč obsahujú viacero položiek. Napríklad môžeme chcieť utriediť dátumy podľa troch kľúčových položiek: roku, mesiaca a dňa. Mohli by sme použiť triediaci algoritmus s porovnávacou funkciou, ktorá najprv porovná roky, ak nastane zhoda, porovná mesiace a ak nastane ďalšia zhoda, porovná dni. Alternatívne môžeme utriediť tieto dátumy na 3 priechody použitím stabilného triedenia: najprv utriedením podľa dňa, potom mesiaca a napokon roku.

Kód radix sortu je priamočiary. Nasledujúca procedúra predpokladá, že každý prvok poľa A veľkosti n má d cifier, kde cifra 1 je číslica najnižšieho rádu a cifra d je najvyššieho rádu.

RADIX-SORT(A, d)

```
1  for i ← 1 to d do
2      pole A utrieď stabilným triedením podľa cifry i.
```

Správnosť radix sortu možno jednoducho dokázať indukciou podľa čísla triedeného stĺpca. Analýza času behu závisí na stabilnom triedení, ktoré je použité ako pomocný triediaci algoritmus. Ak je každá číslica z rozsahu $1..k$ a k nie je príliš veľké, zvyčajne vhodnou voľbou je counting sort. Každý priechod cez n d-ciferných čísel trvá čas $\Theta(n+k)$. Keďže je celkovo d týchto priechodov, celkový čas radix sortu je teda $\Theta(nd+kd)$. Ak d je konštanta a $k = O(n)$, radix sort beží v lineárnom čase.

3.3.4 Bucket sort

Bucket sort je triedenie, ktoré v priemernom prípade beží v lineárnom čase. Podobne ako counting sort, aj bucket sort je rýchle, keďže niečo predpokladá o vstupe. Zatiaľčo counting sort predpokladá, že vstup pozostáva z celých čísel z malého rozsahu, bucket sort predpokladá, že vstup je generovaný náhodným procesom, ktorý prvky distribuuje uniformne na intervale $\langle 0, 1)$.

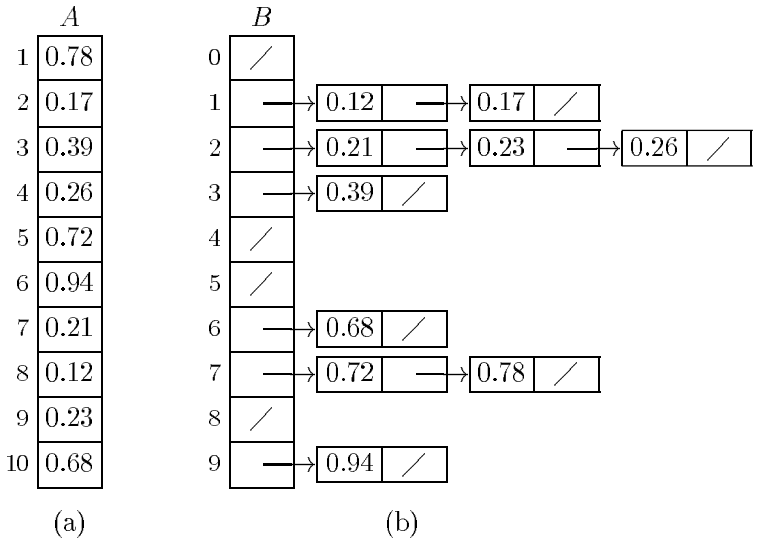
Myšlienkou bucket sortu je rozdeliť interval $\langle 0, 1)$ na n rovnako veľkých disjunktných podintervalov alebo ***bucketov***, a potom do nich rozmiestniť n vstupných čísel. Keďže vstupy sú uniformne

distribuované na $\langle 0, 1)$, očakávame, že do jedného bucketu nepadne príliš mnoho čísel. Aby sme vytvorili výstupnú postupnosť, jednoducho utriedime čísla vo všetkých bucketoch, ktorými potom budeme postupne prechádzať.

Kód bucket sortu predpokladá, že vstupom je n-prvkové pole A, a že každý jeho prvok $A[i]$ spĺňa podmienku $0 \leq A[i] < 1$. Kód potrebuje aj pomocné pole $B[0\,..\,n-1]$ spájaných zoznamov (bucketov) a predpokladá, že je k dispozícii mechanizmus spravujúci takéto zoznamy.

BUCKET-SORT(A)
1 $n \leftarrow \mathit{length}[A]$
2 **for** $i \leftarrow 1$ **to** n **do**
3 vlož $A[i]$ do zoznamu $B[\lfloor nA[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n-1$ **do**
5 utrieď zoznam $B[i]$ použitím insert sortu
6 zreťaz zoznamy $B[0], B[1], \ldots, B[n-1]$ v tomto poradí

Obrázok 9 znázorňuje činnosť bucket sortu na vstupnom poli 10 čísel.



Obr. 9: Činnosť algoritmu bucket sort. (**a**) Vstupné hodnoty v poli $A[1\,..\,10]$. (**b**) Pole $B[0\,..\,9]$ utriedených zoznamov (bucketov) po vykonaní riadku 5 algoritmu. Bucket i obsahuje hodnoty z intervalu $\langle i/10, (i+1)/10)$. Utriedený výstup je zreťazením zoznamov $B[0], B[1], \ldots, B[9]$.

Aby sme videli, že algoritmus pracuje správne, uvažujme dva prvky $A[i]$ a $A[j]$. Ak padnú do toho istého bucketu, vo výstupnej postupnosti sa objavia v správnom poradí lebo obsah každého bucketu je utriedený insert sortom. Predpokladajme teraz, že padnú do rôznych bucketov. Nech sú nimi $B[i']$ a $B[j']$, a nech bez újmy na všeobecnosti $i' < j'$. Keď na riadku 5 zreťazíme zoznamy v B, prvky bucketu $B[i']$ sa budú nachádzať pred prvkami $B[j']$, a teda $A[i]$ sa bude nachádzať pred $A[j]$ vo výstupnej postupnosti. Musíme preto ukázať, že $A[i] \leq A[j]$. Predpokladajme, že by $A[i] > A[j]$. Dostávame

$$i' = \lfloor nA[i] \rfloor \geq \lfloor nA[j] \rfloor = j' \,,$$

čo je v spore s predpokladom $i' < j'$. Teda bucket sort pracuje správne.

Pre analýzu času behu si všimnime, že vykonávanie všetkých riadkov okrem riadku 5 trvá v najhoršom prípade čas $O(n)$. Jediným zaujímavým je preto celkový čas behu spotrebovaný insert sortom.

Nech n_i je náhodná premenná popisujúca počet prvkov umiestnených v buckete $B[i]$. Keďže insert sort beží v kvadratickom čase, očakávaný čas potrebný na utriedenie prvkov v buckete $B[i]$

teda je $\mathrm{E}[O(n_i^2)] = O(\mathrm{E}[n_i^2])$. Celkový očakávaný čas triedenia všetkých prvkov je preto

$$\sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) = O\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right) . \tag{12}$$

Aby sme mohli vyhodnotiť túto sumu, musíme určiť distribúciu každej náhodnej premennej n_i. Máme n prvkov a n bucketov. Pravdepodobnosť, že daný prvok padne do bucketu $B[i]$ je $1/n$. Teda pravdepodobnosť, že $n_i = k$ sleduje binomické rozdelenie $b(k; n, p)$, ktorého stredná hodnota je $\mathrm{E}[n_i] = np = 1$ a variancia $\mathrm{Var}[n_i] = np(1-p) = 1 - 1/n$. Pre ľubovoľnú náhodnú premennú, a teda aj pre n_i^2 máme

$$\mathrm{E}[n_i^2] = \mathrm{Var}[n_i] + \mathrm{E}^2[n^i] = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n} = \Theta(1) .$$

Použitím tohto ohraničenia vo vzťahu (12) môžeme skonštatovať, že očakávaný čas behu insert sortu použitého v bucket sorte je $O(n)$. Teda očakávaný čas behu celého algoritmu bucket sort je $O(n)$.

3.4 Hľadanie mediána[9] a k-teho najmenšieho prvku.

Táto časť sa zaoberá problémom výberu i-teho najmenšieho prvku z množiny n navzájom rôznych prvkov. Pre jednoduchosť predpokladáme, že množina obsahuje navzájom rôzne čísla, hoci v ďalšom budeme uvažovať, že množina je reprezentovaná poľom, a teda sa v nej prvky môžu opakovať. ***Problém výberu*** (i-teho najmenšieho prvku) možno formálne špecifikovať nasledovne:

Vstup: Množina A obsahujúca n (navzájom rôznych) čísel a číslo i také, že $1 \leq i \leq n$.
Výstup: Prvok $x \in A$, ktorý je väčší ako práve $i-1$ prvkov A.

Problém výberu možno jednoducho vyriešiť v čase $O(n \lg n)$ utriedením čísel použitím heap sortu alebo merge sortu a následným výberom i-teho prvku z utriedenej postupnosti. Treba však poznamenať, že existujú aj rýchlejšie algoritmy (konkrétne $O(n)$).

3.4.1 Minimum a maximum

Koľko porovnaní treba vykonať, aby bolo možné určiť minimum množiny n prvkov? Jednoducho môžeme získať horné ohraničenie $n-1$ porovnaní: postupne prechádzame všetky prvky množiny a uchovávame doteraz najmenší nájdený prvok. V nasledujúcej procedúre predpokladáme, že množina je uložená v poli A, kde $length[A] = n$.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A] do
3    if min > A[i] then
4      min ← A[i]
5  return min
```

Je toto najlepšie, čo môžeme urobiť? Áno, pretože dolnou hranicou počtu porovnaní pre problém hľadania minima, ako aj maxima, je $n-1$. Uvažujme napríklad riešenie tohto problému metódou divide & conquer – opäť máme dolné ohraničenie $n-1$. Možno stojí za zmienku, že očakávaný počet vykonaní riadku 4 je $\Theta(\lg n)$.

Poznamenajme ešte, že minimum, resp. maximum dvoch reálnych čísel možno určiť bez potreby ich vzájomného porovnania: [10]

$$\min\{x_1, x_2\} = \frac{x_1 + x_2 - |x_1 - x_2|}{2}, \quad \text{resp.} \quad \max\{x_1, x_2\} = \frac{x_1 + x_2 + |x_1 - x_2|}{2} .$$

[9] Medián postupnosti n prvkov je definovaný ako prvok, ktorý je menší alebo rovný jednej polovici všetkých prvkov a súčasne väčší alebo rovný druhej polovici všetkých prvkov tejto postupnosti.

[10] Funkcia $|x|$ v sebe skrýva porovnanie prvku x s 0.

Je evidentné, že takýto spôsob výpočtu minima, resp. maxima nemá veľký praktický význam z dôvodu jeho relatívne veľkej výpočtovej náročnosti, nehovoriac o možnom vzniku výpočtových chýb.

Súčasný výpočet minima a maxima

V niektorých aplikáciách treba nájsť súčasne minimum aj maximum prvkov množiny n prvkov. Nie je ťažké ukázať, že algoritmus riešiaci tento problém vykoná asymptoticky optimálny počet $\Omega(n)$ operácií porovnania.

V skutočnosti je treba iba $3\lceil n/2 \rceil$ operácií na súčasné nájdenie minima aj maxima. Počas ich hľadania v pamäti udržiavame doteraz nájdený najmenší a najväčší prvok. Namiesto porovnávania každého vstupného prvku so súčasným minimom a maximom (dve porovnania pripadajúce na každý prvok), budeme spracovávať vstupné prvky v pároch. Najprv v týchto pároch porovnáme prvky navzájom, a potom menší z nich porovnáme so súčasným minimom a väčší z páru so súčasným maximom (tri porovnania pripadajúce na dvojicu prvkov).

3.4.2 Výber v lineárnom očakávanom čase

Všeobecný problém výberu sa zdá byť výpočtovo náročnejší ako obyčajné hľadanie minima alebo maxima. Možno je prekvapujúce, že pre oba problémy existujú algoritmy s časom behu $\Theta(n)$. Teraz uvedieme divide & conquer algoritmus pre problém výberu vytvorený na základe quicksortu. Myšlienkou je opäť rekurzívne rozdeľovanie poľa, ale s tým rozdielom, že rekurzívne spracovávame iba jednu takto vzniknutú oblasť. Aj preto je očakávaný čas behu tohto algoritmu iba $\Theta(n)$. Nasledujúci kód pre RANDOMIZED-SELECT vracia i-ty najmenší prvok poľa $A[p\,..\,r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r then
2    return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i ≤ k
6    then return RANDOMIZED-SELECT(A, p, q, i)
7    else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Potom ako je na riadku 3 algoritmu vykonaná procedúra RANDOMIZED-PARTITION, pole $A[p\,..\,r]$ je rozdelené na dve neprázdne podpolia $A[p\,..\,q]$ a $A[q+1\,..\,r]$ také, že každý prvok $A[p\,..\,q]$ je menší ako ľubovoľný prvok poľa $A[q+1\,..\,r]$. Na riadku 4 algoritmu je vypočítaný počet prvkov k poľa $A[p\,..\,q]$. Algoritmus teraz na základe čísla k stanoví, v ktorom z podpolí sa i-ty najmenší prvok nachádza. Ak $i \leq k$, hľadaný prvok sa nachádza v dolnej oblasti, a je rekurzívne vybraný z podpoľa na riadku 6. Ak $i > k$, prvok leží v hornej oblasti. Keďže vieme, že vo vstupnom poli existuje aspoň k hodnôt menších ako hodnota i-teho najmenšieho prvku – konkrétne to sú prvky poľa $A[p\,..\,q]$ – hľadaný prvok je $(i-k)$-ty najmenší prvok $A[q+1\,..\,r]$, ktorý je rekurzívne nájdený na riadku 7.

Najhorší čas behu RANDOMIZED-SELECT je $\Theta(n^2)$ hoci aj pre nájdenie minima, v prípade, že pole vždy rozdelíme podľa najväčšieho zostávajúceho prvku.

3.4.3 Výber v lineárnom čase

Teraz sa budeme zaoberať algoritmom, ktorého čas behu je $O(n)$ *v najhoršom prípade*. Podobne ako RANDOMIZED-SELECT, aj algoritmus SELECT nájde požadovaný prvok rekurzívnym rozdeľovaním vstupného poľa. Myšlienkou tohto algoritmu je *zaručiť* dobré rozdeľovanie poľa. SELECT používa deterministický algoritmus rozdeľovania PARTITION z quicksortu modifikovaný tak, aby prvok, podľa ktorého rozdeľuje vstupné pole, bol jeho parametrom.

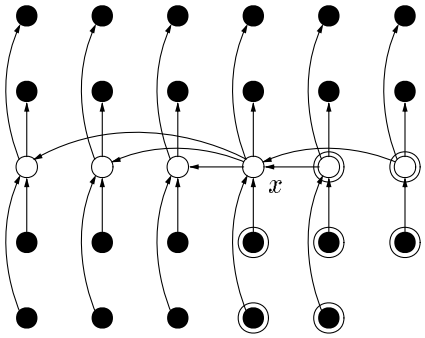
Algoritmus SELECT nájde i-ty najmenší prvok vstupného poľa n prvkov vykonaním nasledujúcich krokov:

1. Rozdeľ n prvkov vstupného poľa na $\lfloor n/5 \rfloor$ skupín po 5 prvkoch a nanajvýš jednu skupinu vytvorenú zo zvyšných $n \bmod 5$ prvkov.
2. Každú z $\lceil n/5 \rceil$ takto vzniknutých skupín použitím insert sortu utrieď a následne vyber mediány týchto skupín (v prípade nepárneho počtu prvkov skupiny vyber jej prostredný prvok, v prípade párneho prvku vyber väčší z dvoch prostredných prvkov).
3. Rekurzívne použi SELECT na nájdenie mediána x z $\lceil n/5 \rceil$ mediánov nájdených v predchádzajúcom kroku.
4. Rozdeľ vstupné pole podľa mediána mediánov x použitím modifikovanej verzie PARTITION. Nech k je počet prvkov v dolnej oblasti, potom $n-k$ je počet prvkov v hornej oblasti.
5. Rekurzívne použi SELECT na nájdenie i-teho najmenšieho prvku v dolnej oblasti, ak $i \leq k$, alebo $(i-k)$-teho najmenšieho prvku v hornej oblasti, ak $i > k$.

Aby sme mohli analyzovať čas behu procedúry SELECT, najprv určíme dolné ohraničenie počtu prvkov, ktoré sú väčšie ako rozdeľujúci prvok x. Obrázok 10 môže napomôcť lepšej predstave o činnosti tejto procedúry. Najmenej polovica mediánov nájdených v kroku 2 je väčšia alebo rovná mediánu mediánov x. To znamená, že aspoň polovica z $\lceil n/5 \rceil$ skupín prispieva troma prvkami, ktoré sú väčšie ako x, s výnimkou skupiny, ktorá má menej ako 5 prvkov ak n nie je deliteľné 5 bez zvyšku. Naviac, jedna zo skupín obsahuje, ako jeden z jej prvkov, samotnú hodnotu x. Ak neberieme do úvahy tieto dve skupiny, dostávame, že počet prvkov väčších ako x je aspoň

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6\,.$$

Podobne, aj počet prvkov, ktoré sú menšie ako x je aspoň $3n/10-6$. To znamená, že v najhoršom prípade je v 5. kroku SELECT rekurzívne vyvolaný nanajvýš pre $7n/10+6$ prvkov.



Obr. 10: Analýza algoritmu SELECT. Malé kruhy reprezentujú n prvkov, pričom sú podľa príslušných skupín zoradené do stĺpcov. Mediány skupín sú označené bielymi kružnicami a medián týchto mediánov je označený symbolom x. Šípky vedú od väčších prvkov k menším, z čoho možno vidieť, že 3 z každej skupiny piatich prvkov naľavo od x sú väčšie ako x a podobne, ale s výnimkou poslednej skupiny obsahujúcej menej ako 5 prvkov, je to tak aj napravo od x. Prvky, ktoré sú určite väčšie ako x sú naviac zakrúžkované.

Teraz môžeme zostaviť rekurenciu pre najhorší čas behu $T(n)$ algoritmu SELECT. Kroky 1, 2 a 4 trvajú čas $O(n)$. (Krok 2 pozostáva z $O(n)$ volaní insert sortu na množinách veľkosti $O(1)$.) Krok 3 trvá $T(\lceil n/5 \rceil)$ a krok 5 trvá nanajvýš $T(7n/10+6)$ za predpokladu, že T je rastúca funkcia. Všimnime si, že $7n/10+6 < n$ pre $n > 20$, a že každý vstup pozostávajúci z nanajvýš 80 prvkov vyžaduje čas $O(1)$. Dostávame preto rekurentný vzťah

$$T(n) \leq \begin{cases} \Theta(1) & \text{ak } n \leq 80\,, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{ak } n > 80\,. \end{cases}$$

Pomocou substitučnej metódy ukážeme, že čas behu algoritmu SELECT je lineárny. Predpokladajme teraz, že $T(n) \leq cn$ pre nejakú konštantu c a pre všetky $n \leq 80$. Substituovanie tejto induktívnej hypotézy do pravej strany rekurencie dáva

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10+6) + O(n) \\ &\leq cn/5 + c + 7cn/10 + 6c + O(n) \\ &\leq 9cn/10 + 7c + O(n) \\ &\leq cn \ , \end{aligned}$$

keďže môžeme nájsť dostatočne veľké c, aby $c(n/10-7)$ bolo väčšie ako funkcia popísaná výrazom $O(n)$ pre všetky $n > 80$. Najhorší čas behu procedúry SELECT je teda lineárny.

4 Dátové štruktúry

Množiny majú pre informatiku podobný význam ako pre matematiku. Ale zatiaľčo sú matematické množiny nemenné, množiny, s ktorými narábajú algoritmy, sa môžu zväčšovať, zmenšovať, alebo inak časom meniť. Takéto množiny nazývame ***dynamické***.

Mnoho algoritmov potrebuje *vkladať* prvky do množiny, *odstraňovať* ich z nej a *testovať príslušnosť* prvku do množiny. Dynamická množina, ktorá podporuje tieto operácie sa nazýva ***slovník***.

Prvky dynamickej množiny

Typická implementácia dynamickej množiny predpokladá, že každý prvok je reprezentovaný objektom, s ktorého položkami je možné manipulovať za predpokladu, že máme ukazovateľ na tento objekt. Niekedy je možné naviac predpokladať, že jednou z položiek objektu je indentifikujúca ***kľúčová*** položka. Ak sú všetky kľúče navzájom rôzne, možno dynamickú množinu chápať ako množinu kľúčových hodnôt. Objekt môže obsahovať ***satelitné dáta***, ktoré sú uchovávané vo zvyšných položkách, ale inak sú nepoužité v implementácii množiny. Taktiež môže obsahovať položky, s ktorými sa manipuluje pri vykonávaní množinových operácií. Tieto položky môžu napríklad obsahovať ukazovatele na ďalšie objekty v množine.

Operácie na dynamických množinách

Operácie na dynamických množinách možno rozdeliť do dvoch kategórií: na ***dotazy***, ktoré poskytujú nejakú informáciu o množine a ***modifikujúce operácie***, ktoré menia obsah množiny. Špecifická aplikácia bude zvyčajne využívať len niekoľko operácií z nasledujúceho zoznamu.

SEARCH(S,k)	Dotaz, ktorý pre danú množinu S a kľúčovú hodnotu k vráti ukazovateľ x na prvok množiny S taký, že $key[x] = k$, alebo NIL ak žiadny taký prvok do S nepatrí.
INSERT(S,x)	Modifikujúca operácia, ktorá rozšíri množinu S o prvok, na ktorý ukazuje x.
DELETE(S,x)	Modifikujúca operácia, ktorá vymaže prvok množiny S určený ukazovateľom x.
MINIMUM(S)	Dotaz na totálne usporiadanú množinu S, ktorý vráti prvok x patriaci S s najmenšou hodnotou kľúča.
MAXIMUM(S)	Dotaz na totálne usporiadanú množinu S, ktorý vráti prvok x patriaci S s najväčšou hodnotou kľúča.
SUCCESSOR(S,x)	Dotaz, ktorý pre daný prvok x, ktorého kľúč sa nachádza v totálne usporiadanej množine S, vráti nasledujúci väčší prvok S alebo NIL, ak x je maximum.
PREDECESSOR(S,x)	Dotaz, ktorý pre daný prvok x, ktorého kľúč sa nachádza v totálne usporiadanej množine S, vráti nasledujúci menší prvok S alebo NIL, ak x je minimum.

Dotazy SUCCESSOR a PREDECESSOR sú často rozširované na množiny obsahujúce nie nutne navzájom rôzne kľúče. Pre množinu n kľúčov je obvyklé predpokladať, že dotaz MINIMUM nasledovaný $n-1$ volaniami SUCCESSOR enumeruje prvky množiny v usporiadanom poradí.

Čas potrebný na vykonanie množinovej operácie je zvyčajne vyjadrovaný na základe veľkosti danej množiny ako jedného z parametrov.

4.1 Elementárne dátové štruktúry

4.1.1 Zásobníky a fronty

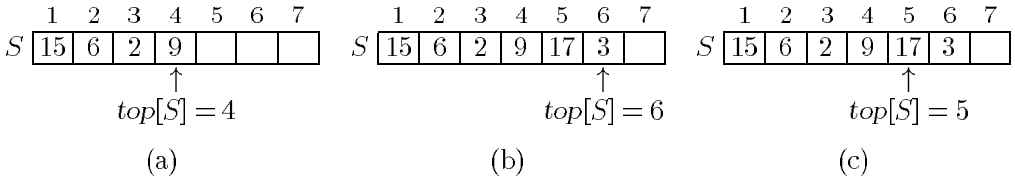
Zásobníky a fronty sú dynamické množiny, pri ktorých prvok odstránený z množiny operáciou DELETE, je už vopred určený. Pri ***zásobníku*** je to prvok, ktorý bol vložený do množiny naposledy (*LIFO – last-in, first-out*), a pri fronte je to prvok, ktorý sa v množine nachádza najdlhší čas (*FIFO – first-in, first-out*).

Zásobník

Operácia INSERT na zásobníku sa zvyčajne nazýva PUSH a bezparametrová operácia DELETE sa nazýva POP.

Ako možno vidieť na obrázku 11, zásobník obsahujúci nanajvýš n prvkov sa dá implementovať pomocou poľa $S[1..n]$. Toto pole má atribút $top[S]$, ktorý je indexom naposledy vloženého prvku. Zásobník pozostáva z prvkov $S[1..top[S]]$, kde $S[1]$ je prvok na dne a $S[top[S]]$ je prvok na vrchole zásobníka.

Ak $top[S] = 0$, zásobník neobsahuje žiadne prvky – je ***prázdny.*** Zásobník možno testovať na prázdnosť operáciou STACK-EMPTY. Ak je na prázdnom zásobníku vykonaná operácia POP hovoríme, že zásobník ***podtiekol***, čo je obyčajne chyba. Ak $top[S]$ prekročí n, zásobník ***pretiekol.*** (V pseudokóde sa nezaoberáme prípadom pretečenia.)



Obr. 11: Implementácia zásobníka S pomocou poľa. Prvky zásobníka sa nachádzajú iba na pozíciách 1 až $top[S]$. (**a**) Zásobník S má 4 prvky. Najvrchnejší prvok je 9. (**b**) Zásobník S po vykonaní PUSH$(S, 17)$ a PUSH$(S, 3)$. (**c**) Zásobník S po vyvolaní POP(S) vrátil hodnotu 3, ktorá bola naposledy vložená. Hoci sa táto hodnota ešte nachádza v poli, už nie je na zásobníku a na vrchole zásobníka je teraz 17.

Zásobníkové operácie možno implementovať niekoľkými riadkami kódu.

```
STACK-EMPTY(S)
1  if top[S] = 0
2    then return TRUE
3    else return FALSE
```

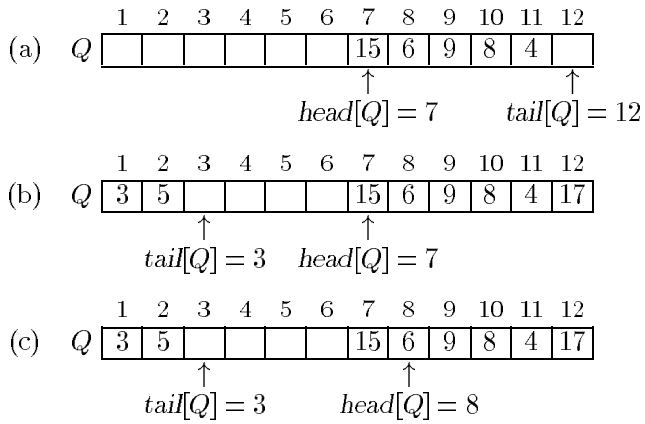
```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2    then error "underflow"
3    else
4      top[S] ← top[S] − 1
5      return S[top[S] + 1]
```

Obrázok 11 znázorňuje účinky modifikujúcich operácií PUSH a POP. Všetky tri zásobníkové operácie trvajú čas $O(1)$.

Fronta

Operáciu vkladania prvku do fronty nazývame ENQUEUE a operáciu odstránenia prvku z fronty nazývame DEQUEUE. Podobne ako zásobníková operácia POP, aj DEQUEUE odstraňuje implicitne



Obr. 12: Fronta implementovaná pomocou poľa $Q[1..12]$. Prvky fronty sa nachádzajú od $head[Q]$ doprava (cyklicky) až po $tail[Q]$. (**a**) Fronta má 5 prvkov na pozíciách $Q[7..11]$. (**b**) Stav fronty po vykonaní ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$ a ENQUEUE$(Q, 5)$. (**c**) Stav fronty po vykonaní operácie DEQUEUE(Q), ktorej výsledkom bola hodnota 15 nachádzajúca sa predtým na hlave fronty. Nová hlava má teraz kľúčovú hodnotu 6.

zadaný prvok. Fronta má ***hlavu*** a ***chvost***. Prvok vkladaný do fronty je umiestnený na jej koniec. Prvok, ktorý sa má vybrať nasledujúcou operáciou DEQUEUE, sa nachádza na začiatku fronty.

Obrázok 12 znázorňuje jeden spôsob implementácie fronty obsahujúcej nanajvýš $n-1$ prvkov použitím poľa $Q[1..n]$. Fronta má atribút $head[Q]$, ktorý ukazuje na jej hlavu. Atribút $tail[Q]$ ukazuje na pozíciu, na ktorú bude umiestnený ďalší vložený prvok. Prvky fronty sú na pozíciách $head[Q], head[Q]+1, \ldots, tail[Q]-1$, pričom predpokladáme, že pozícia 1 priamo nasleduje za pozíciou n. Ak $head[Q] = tail[Q]$, fronta je prázdna. Na začiatku máme $head[Q] = tail[Q] = 1$. Ak je fronta prázdna, pokus o výber prvku z tejto fronty spôsobí jej podtečenie. Ak $head[Q] = tail[Q]+1$, fronta je plná, a pokus o vloženie prvku spôsobí jej pretečenie.

V našich procedúrach ENQUEUE a DEQUEUE je testovanie prípadného podtečenia alebo pretečenia vynechané.

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3    then tail[Q] ← 1
4    else tail[Q] ← tail[Q] + 1
```

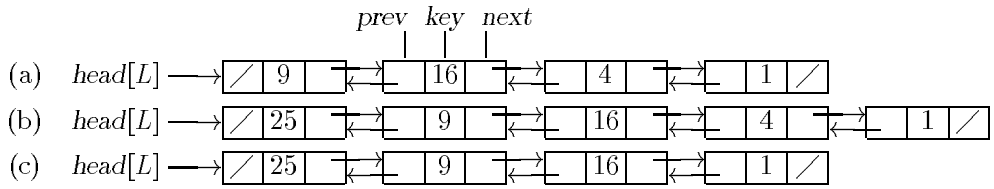
```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3    then head[Q] ← 1
4    else head[Q] ← head[Q] + 1
5  return x
```

Obrázok 12 znázorňuje činnosť operácií ENQUEUE a DEQUEUE. Obe trvajú čas $O(1)$.

4.1.2 Spájané zoznamy

Spájaný zoznam je dátová štruktúra, v ktorej sú objekty zoradené lineárne. Narozdiel od poľa, v ktorom je poradie stanovené indexami, poradie v spájanom zozname je určované ukazovateľmi vo všetkých objektoch. Spájané zoznamy poskytujú jednoduchú a flexibilnú reprezentáciu pre dynamické množiny podporujúce všetky operácie uvedené na strane 35.

Ako možno vidieť na obrázku 13, každý prvok ***zoznamu s dvojitou väzbou*** L je objekt s položkou key a dvoma ďalšími položkami (typu ukazovateľ): $next$ a $prev$. Objekt môže samozrejme obsahovať aj satelitné dáta. Ak x je prvok zoznamu, $next[x]$ ukazuje na jeho nasledovníka v zozname a $prev[x]$ ukazuje na jeho predchodcu. Ak $prev[x] =$ NIL, prvok x nemá predchodcu, a preto je prvým prvkom (***head***) zoznamu. Ak $next[x] =$ NIL, prvok x nemá nasledovníka, a je teda posledným prvkom (***tail***) zoznamu. Atribút $head[L]$ ukazuje na prvý prvok zoznamu. Ak $head[L] =$ NIL, zoznam je prázdny.



Obr. 13: (**a**) Zoznam L s dvojitou väzbou reprezentujúci dynamickú množinu $\{1, 4, 9, 16\}$. Každý prvok zoznamu je objekt s položkami pre kľúče a ukazovatele (znázornené šípkami) na nasledujúce alebo predchádzajúce objekty. Položky *next* posledného prvku a *prev* hlavy obsahujú hodnotu NIL. (**b**) Po vykonaní LIST-INSERT(L, x), kde $key[x] = 25$, zoznam obsahuje nový objekt s kľúčom 25 ako prvý prvok. Tento nový objekt ukazuje na pôvodnú hlavu s kľúčom 9. (**c**) Výsledok vyvolania LIST-DELETE(L, x), kde x ukazuje na objekt s kľúčom 4.

Zoznam môže nadobúdať jednu z viacerých podôb. Môže mať buď jednoduchú alebo dvojitú väzbou, môže byť utriedený, poprípade môže byť kruhový a pod. Pri zoznamoch ***s jednoduchou väzbou*** v každom prvku vynechávame ukazovateľ *prev*. Ak je zoznam ***utriedený***, poradie prvkov v zozname zodpovedá usporiadaniu kľúčov jednotlivých prvkov. V ***kruhovom zozname*** smerník *prev* hlavy ukazuje na posledný prvok a smerník *next* ukazuje na hlavu zoznamu. V nasledujúcom budeme predpokladať, že zoznamy majú dvojitú väzbu a nie sú usporiadané.

Vyhľadávanie v spájanom zozname

Procedúra LIST-SEARCH(L, k) nájde v zozname L prvý prvok s kľúčom k jednoduchým lineárnym prehľadávaním, pričom vráti ukazovateľ na tento prvok. Ak sa v danom zozname taký prvok nenachádza, procedúra vráti hodnotu NIL.

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while (x ≠ NIL) and (key[x] ≠ k) do
3     x ← next[x]
4  return x
```

Je zrejmé, že na vyhľadanie prvku v zozname pozostávajúcom z n prvkov, procedúra LIST-SEARCH potrebuje v najhoršom prípade čas $\Theta(n)$.

Vkladanie do spájaného zoznamu

Procedúra LIST-INSERT umiestni daný prvok x na začiatok zoznamu, čo je zrejmé z obr. 13b.

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL then
3     prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Čas behu procedúry LIST-INSERT na zozname n prvkov je $O(1)$.

Vymazávanie zo spájaného zoznamu

Procedúra LIST-DELETE odstráni prvok x zo spájaného zoznamu L. Aby mohol byť prvok zo zoznamu odstránený, procedúra musí naň dostať ukazovateľ. Prvok bude potom zo zoznamu vyňatý jednoduchou zmenou ukazovateľov. Ak teda chceme vymazať prvok s daným kľúčom, musíme najprv vyvolať procedúru LIST-SEARCH.

LIST-DELETE(L, x)

```
1  if prev[x] ≠ NIL
2    then next[prev[x]] ← next[x]
3    else head[L] ← next[x]
4  if next[x] ≠ NIL
5    then prev[next[x]] ← prev[x]
```

Obrázok 13c znázorňuje vymazanie prvku zo zoznamu. Procedúra LIST-DELETE beží v čase $O(1)$, ale ak si želáme odstrániť prvok s daným kľúčom, treba na to v najhoršom prípade čas $\Theta(n)$ z dôvodu nutnosti vyvolania funkcie LIST-SEARCH.

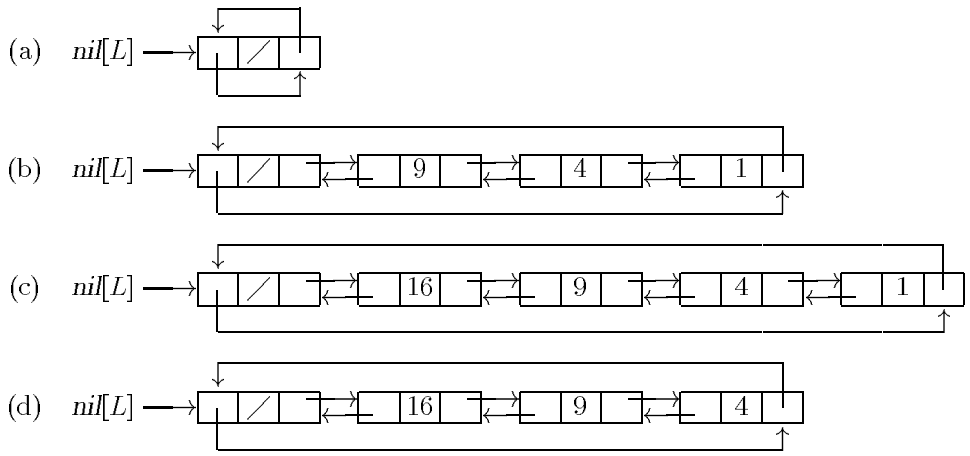
Sentinely

Kód pre LIST-DELETE by mohol byť jednoduchší ak by sme mohli ignorovať okrajové podmienky na začiatku a konci zoznamu.

LIST-DELETE$'(L, x)$

```
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

Sentinel je pomocný objekt neobsahujúci dáta, ktorý nám umožňuje zjednodušiť okrajové podmienky. Predpokladajme, že spolu so zoznamom L budeme poskytovať aj objekt $nil[L]$, ktorý síce reprezentuje NIL, ale obsahuje všetky položky podobne ako ostatné prvky zoznamu. Kdekoľvek v kóde pre manipuláciu so zoznamom sa bude vyskytovať NIL, nahradíme ho odkazom na sentinel $nil[L]$. Ako možno vidieť na obr. 14, toto spôsobí, že z nášho zoznamu sa stane kruhový zoznam so sentinelom $nil[L]$ umiestneným medzi hlavu a posledný prvok zoznamu; položka $next[nil[L]]$ ukazuje na jeho hlavu a $prev[nil[L]]$ ukazuje na jeho posledný prvok. Keďže $next[nil[L]]$ ukazuje na hlavu, môžeme eliminovať atribút $head[L]$. Prázdny zoznam pozostáva iba zo sentinelu, pričom položky $next[nil[L]]$ a $prev[nil[L]]$ ukazujú na $nil[L]$.



Obr. 14: Spájaný zoznam L, ktorý používa sentinel $nil[L]$ (najviac vľavo). Atribút $head[L]$ už nie je potrebný, pretože ku hlave môžeme pristupovať pomocou $next[nil[L]]$. **(a)** Prázdny zoznam. **(b)** Spájaný zoznam reprezentujúci množinu $\{1, 4, 9\}$ s kľúčom 9 v hlave a kľúčom 1 v poslednom prvku. **(c)** Ten istý zoznam po vykonaní LIST-INSERT$'(L, x)$, kde $key[x] = 16$. Nový objekt je teraz hlavou tohto zoznamu. **(d)** Zoznam po odstránení objektu s kľúčom 1. Posledným prvkom je teraz objekt s kľúčom 4.

Kód procedúry LIST-SEARCH zostáva v podstate nezmenený, ale odkazy na NIL, resp. $head[L]$ sú nahradené $nil[L]$, resp. $next[nil[L]]$.

LIST-SEARCH$'(L, k)$

```
1  x ← next[nil[L]]
2  while (x ≠ nil[L]) and (key[x] ≠ k) do
3     x ← next[x]
4  return x
```

Na vloženie prvku do zoznamu možno použiť nasledujúcu procedúru.

LIST-INSERT$'(L, x)$

```
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

Obr. 14 znázorňuje činnosť LIST-INSERT$'$ a LIST-DELETE$'$ na spájanom zozname.

Sentinely zriedkakedy redukujú asymptotické ohraničenia časov behu operácií na dátových štruktúrach, ale môžu znížiť aspoň konštantné faktory. Ak však narábame s väčším počtom malých zoznamov, množstvo pamäti spotrebovanej ich sentinelmi už nemusí byť zanedbateľné.

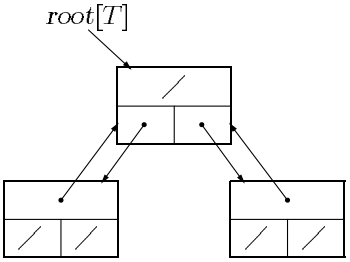
4.1.3 Reprezentácia stromov

V tejto časti sa budeme zaoberať reprezentáciou stromov pomocou spájaných dátových štruktúr. Najprv to budú binárne stromy, a potom to bude metóda pre stromy s neobmedzeným počtom potomkov.

Každý uzol stromu reprezentujeme objektom, ktorý podobne ako pri spájaných zoznamoch, obsahuje kľúčovú položku. Počet položiek obsahujúcich ukazovatele na ďalšie objekty závisí od typu reprezentovaného stromu.

Binárne stromy

Ako možno vidieť na obr. 15, položky p, *left* a *right* používame pre uchovanie ukazovateľov na rodiča, pravého a ľavého potomka každého uzla binárneho stromu T. Ak $p[x] = \text{NIL}$, potom x je koreň stromu. Ak uzol x nemá ľavého potomka, potom $left[x] = \text{NIL}$ (podobne aj pre pravého potomka). Na koreň celého stromu T ukazuje atribút $root[T]$. Ak $root[T] = \text{NIL}$, potom je strom prázdny.



Obr. 15: Reprezentácia binárneho stromu T. Každý uzol x má položky $p[x]$ (hore), $left[x]$ (vľavo dole) a $right[x]$ (vpravo dole). Kľúčové položky nie sú znázornené.

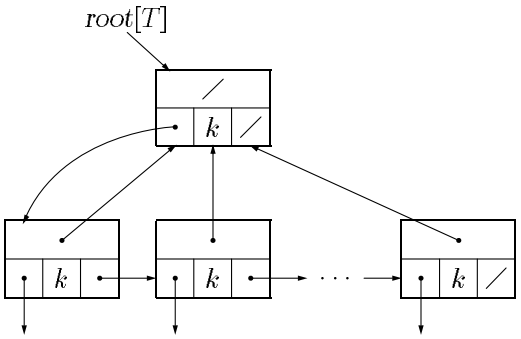
Stromy s neohraničeným počtom potomkov

Schému pre reprezentáciu binárnych stromov možno rozšíriť na ľubovoľnú triedu stromov, v ktorej je počet potomkov každého uzla zhora ohraničený nejakou konštantou k: položky *left* a *right* jednoducho nahradíme položkami $child_1, child_2, \ldots, child_k$. Táto schéma však nie je vhodná pre reprezentáciu stromov s neohraničeným počtom potomkov. Naviac, ak hodnota k je príliš veľká a väčšina uzlov má malý počet potomkov, výrazne plýtvame pamäťou.

Našťastie existuje spôsob, ako pomocou binárnych stromov reprezentovať stromy s ľubovoľným počtom potomkov, pričom spotrebovaná pamäť má veľkosť $O(n)$, kde n je počet uzlov reprezentovaného stromu. Reprezentácia ***left-child, right-sibling*** je znázornená na obr. 16. Podobne ako predtým, aj teraz každý uzol obsahuje ukazovateľ na rodiča p, a $root[T]$ ukazuje na koreň stromu T. Namiesto toho, aby sme pre každého potomka mali jeden ukazovateľ, máme iba dva ukazovatele:

1. *left-child*$[x]$ ukazuje na najľavejšieho potomka uzla x,
2. *right-sibling*$[x]$ ukazuje na súrodenca x hneď vpravo od neho.

Ak uzol x nemá potomkov, potom *left-child*$[x] =$ NIL a ak uzol x je najpravejším potomkom svojho rodiča, potom *right-sibling*$[x] =$ NIL.



Obr. 16: Left-child, right-sibling reprezentácia stromu T. Každý uzol x má položky $p[x]$ (hore), *left-child*$[x]$ (vľavo dole) a *right-sibling*$[x]$ (vpravo dole).

4.2 Hašovanie

Mnoho aplikácií potrebuje pre svoju činnosť dynamické množiny podporujúce iba základné slovníkové operácie INSERT, SEARCH a DELETE. Efektívnymi dátovými štruktúrami pre implementáciu slovníkov sú hašovacie tabuľky. Hoci vyhľadávanie prvku v hašovacej tabuľke môže trvať rovnako dlho ako v spájanom zozname – čas $\Theta(n)$ v najhoršom prípade – v praxi si napriek tomu hašovanie počína veľmi dobre. Za vhodných okolností je očakávaný čas vyhľadávania prvku v hašovacej tabuľke rovný $O(1)$.

Hašovacia tabuľka je zovšeobecnením klasického poľa. Priama adresácia do poľa nám umožňuje vyšetriť obsah ľubovoľnej z jeho položiek v čase $O(1)$. Priamu adresáciu používame ak si môžeme dovoliť alokovať pole, ktoré má voľné pozície pre všetky možné kľúče.

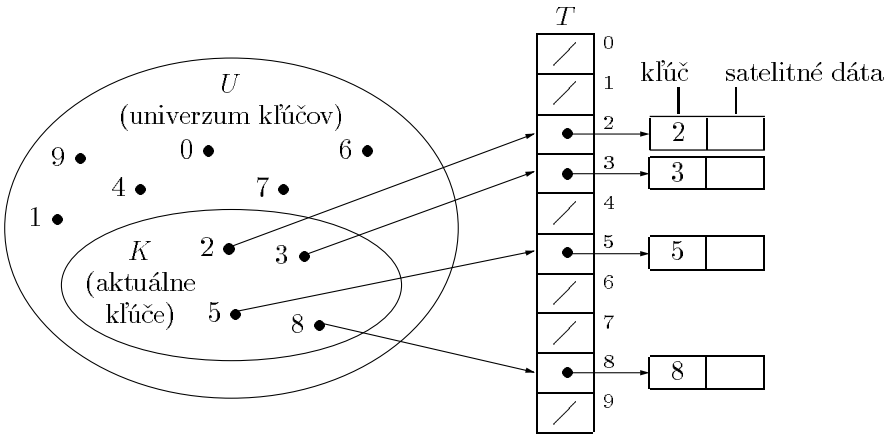
Ak je počet uchovávaných kľúčov relatívne malý v porovnaní s celkovým počtom možných kľúčov, potom sa hašovacie tabuľky stávajú vhodnou alternatívou priameho adresovania, nakoľko hašovacia tabuľka zvyčajne používa pole veľkosti úmernej počtu uchovávaných kľúčov. Namiesto priameho použitia kľúča ako indexu do poľa, tento index je z kľúča *vypočítavaný*.

4.2.1 Priama adresácia

Priama adresácia je jednoduchá technika vhodná v prípade, že univerzum U kľúčov [11] nie je príliš veľké. Uvažujme aplikáciu, ktorá potrebuje dynamickú množinu, v ktorej každý prvok má kľúč z univerza $U = \{0, 1, \ldots, m-1\}$, pričom m nie je príliš veľké. Naviac predpokladajme, že žiadne dva prvky nemajú rovnakú hodnotu kľúča.

Na reprezentáciu dynamickej množiny použijeme pole (ako tabuľku s priamou adresáciou) $T[0\,..\,m-1]$, v ktorom každá pozícia (slot) prislúcha nejakému kľúču z univerza U. Obr. 17 znázorňuje princíp tejto techniky: slot k ukazuje na prvok množiny, ktorý má kľúč k. Ak množina neobsahuje prvok s kľúčom k, potom $T[k] =$ NIL.

[11] Množina všetkých možných kľúčov.



Obr. 17: Implementácia dynamickej množiny pomocou tabuľky T s priamou adresáciou. Každému kľúču z univerza $U = \{0, 1, \ldots, 9\}$ prislúcha jedna položka tejto tabuľky. Množina $K = \{2, 3, 5, 8\}$ aktuálnych kľúčov určuje pozície tabuľky, ktoré obsahujú ukazovatele na jednotlivé prvky. Ostatné pozície obsahujú ukazovatele NIL.

Implementácia základných slovníkových operácií je skutočne triviálna.

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[key[x]] \leftarrow$ NIL

Všetky tieto operácie sú veľmi rýchle – vyžadujú čas iba $O(1)$.

V niektorých aplikáciách je možné uložiť prvky dynamickej množiny priamo do tabuľky, t.j. namiesto ukladania kľúča prvku a satelitných dát do externého objektu, uložíme daný objekt (bez kľúča) na príslušnú pozíciu tabuľky. Treba však mať na pamäti, že musíme byť nejakým spôsobom schopní rozlíšiť prázdne pozície tabuľky od neprázdnych.

4.2.2 Hašovacie tabuľky

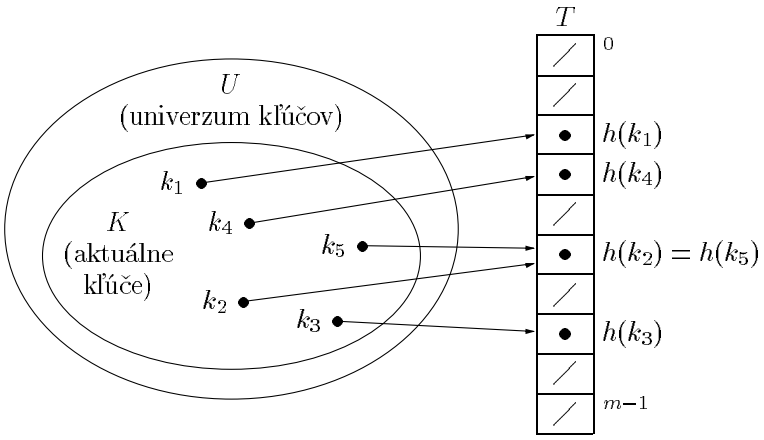
Ak je univerzum U príliš veľké, použitie tabuľky T veľkosti $|U|$ môže byť nepraktické alebo dokonca i nemožné. Naviac množina K práve uložených kľúčov môže byť relatívne malá v porovnaní s U, teda väčšina pamäti alokovaná pre T môže zostať nevyužitá.

Ak množina K kľúčov uložených v slovníku je značne menšia ako univerzum U všetkých možných kľúčov, hašovacia tabuľka potrebuje omnoho menej pamäti ako tabuľka s priamou adresáciou. Konkrétne, požiadavky na pamäť možno redukovať na $\Theta(|K|)$, hoci vyhľadávanie prvku v hašovacej tabuľke bude v priemernom prípade trvať čas iba $O(1)$.

Pri priamej adresácii je prvok s kľúčom k uložený na pozícii k. Pri hašovaní je tento prvok uložený na pozícii $h(k)$; teda na výpočet pozície z kľúča k je použitá ***hašovacia funkcia*** h. Táto funkcia zobrazuje univerzum U kľúčov na pozície ***hašovacej tabuľky*** $T[0\,..\,m-1]$:

$$h \;:\; U \to \{0, 1, \ldots, m-1\}\,.$$

Hovoríme, že prvok s kľúčom k sa ***hašuje*** na pozíciu $h(k)$. Obrázok 18 znázorňuje základnú myšlienku hašovania. Úlohou hašovacej funkcie je zmenšiť rozsah indexov, ktoré treba ošetriť – namiesto $|U|$ hodnôt potrebujeme teraz ošetriť iba m hodnôt.

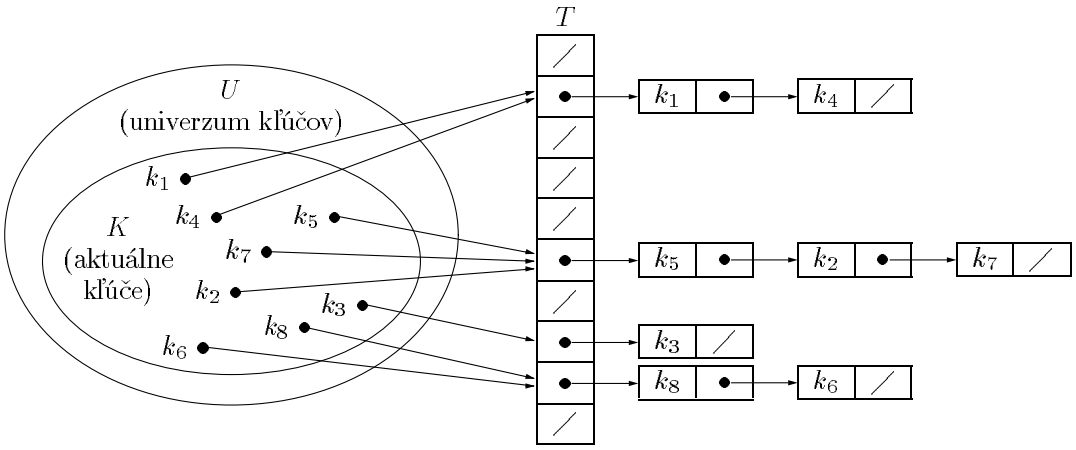


Obr. 18: Použitie hašovacej funkcie h pre zobrazenie kľúčov na pozície hašovacej tabuľky. Kľúče k_2 a k_5 sú zobrazené na tú istú pozíciu, teda kolidujú.

Zádrheľom tejto myšlienky je, že dva rôzne kľúče sa môžu zobraziť na tú istú pozíciu, t.j. kolidovať. Našťastie však existujú techniky riešiace prípadné kolízie.

Hašovanie s reťazením

Pri zreťazovaní umiestňujeme všetky prvky, ktoré sa zobrazia na tú istú pozíciu, do spájaného zoznamu, ako možno vidieť na obr. 19. Slot j obsahuje ukazovateľ na hlavu zoznamu uložených prvkov, ktoré sa hašujú na pozíciu j. Ak takéto prvky neexistujú, slot j obsahuje NIL.



Obr. 19: Riešenie kolízií zreťazením. Každý slot $T[j]$ hašovacej tabuľky obsahuje spájaný zoznam všetkých kľúčov, ktorých kľúče sa hašujú na pozíciu j. Napríklad $h(k_1) = h(k_4)$ a $h(k_5) = h(k_2) = h(k_7)$.

Slovníkové operácie na hašovacej tabuľke T možno ľahko implementovať, ak sú kolízie riešené zreťazením.

CHAINED-HASH-INSERT(T, x)

1 vlož x do čela zoznamu $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)

1 vyhľadaj v zozname $T[h(k)]$ prvok s kľúčom k

CHAINED-HASH-DELETE(T, x)

1 odstráň x zo zoznamu $T[h(key[x])]$

Najhorší čas behu vkladania je $O(1)$. Pre vyhľadávanie je najhorší čas behu úmerný dĺžke zoznamu. Vymazávanie prvku x možno vykonať v čase $O(1)$ ak použijeme spájané zoznamy s dvojitou väzbou.

Analýza hašovania so zreťazením

Nech T je hašovacia tabuľka o m slotoch, v ktorej je uložených n prvkov. Pre T definujeme ***ukladací koeficient*** α ako n/m, čo je priemerný počet prvkov uložených v zozname.

V najhoršom prípade je čas potrebný na vyhľadanie kľúča v hašovacej tabuľke rovný $\Theta(n)$ (čo sa týka konštantných faktorov, je dokonca horší ako čas vyhľadávania v obyčajnom zozname). Účinnosť hašovania v priemernom prípade závisí najmä od toho ako hašovacia funkcia h distribuuje množinu kľúčov medzi m slotov. Teraz budeme predpokladať, že pravdepodobnosť, že ktorýkoľvek prvok padne do ľubovoľného z m slotov je $1/m$ nezávisle od toho, kam padne hociktorý iný. Tento predpoklad nazývame ***jednoduché uniformné hašovanie***.

Ďalej predpokladáme, že hodnotu $h(k)$ možno vypočítať v čase $O(1)$, takže čas potrebný na nájdenie prvku s kľúčom k závisí od dĺžky zoznamu $T[h(k)]$ lineárne. Odhliadnuc od času $O(1)$ potrebného na výpočet hašovacej funkcie a prístup ku slotu $h(k)$, je počet prvkov, ktoré treba preskúmať vyhľadávacím algoritmom nanajvýš rovný počtu prvkov zoznamu $T[h(k)]$. Budeme uvažovať dva prípady. V prvom prípade je vyhľadávanie neúspešné, t.j. žiaden prvok tabuľky nemá hodnotu kľúča rovnú k, a v druhom je úspešné, t.j. prvok s danou hodnotou kľúča je v tabuľke nájdený.

Veta 4.1 (O neúspešnom hľadaní) *V hašovacej tabuľke, v ktorej sú kolízie riešené zreťazením, neúspešné vyhľadávanie trvá v priemernom prípade čas $\Theta(1+\alpha)$ za predpokladu jednoduchého uniformného hašovania.*

Dôkaz. V prípade jednoduchého uniformného hašovania sa každý kľúč sa zobrazí do jedného z m slotov s rovnakou pravdepodobnosťou. Priemerný čas neúspešného vyhľadávania kľúča k je vlastne priemerný čas hľadania až po koniec jedného z m zoznamov. Priemerná dĺžka takéhoto zoznamu je $\alpha = n/m$ (ukladací koeficient) a celkový potrebný čas, vrátane času na výpočet $h(k)$, je teda $\Theta(1+\alpha)$. □

Veta 4.2 (O úspešnom hľadaní) *V hašovacej tabuľke, v ktorej sú kolízie riešené zreťazením, úspešné vyhľadávanie trvá v priemernom prípade čas $\Theta(1+\alpha)$ za predpokladu jednoduchého uniformného hašovania.*

Dôkaz. Predpokladáme, že hľadaný kľúč je hociktorý z n kľúčov uložených v tabuľke s rovnakou pravdepodobnosťou. Predpokladajme teraz, bez újmy na všeobecnosti, že procedúra CHAINED-HASH-INSERT vkladá nový prvok na koniec zoznamu. Očakávaný počet prvkov vyšetrených počas úspešného hľadania je o 1 väčší ako počet prvkov vyšetrených v okamihu keď hľadaný prvok bol do zoznamu vložený. Aby sme našli očakávaný počet preskúmaných prvkov, vezmeme hodnotu o 1 väčšiu ako je priemerná hodnota očakávanej dĺžky zoznamu, do ktorého je i-ty prvok pridávaný. Táto hodnota je $(i-1)/m$, a teda očakávaný počet vyšetrených prvkov je

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\,.\end{aligned}$$

Teda celkový čas potrebný pre úspešné vyhľadávanie (vrátane času pre výpočet hašovacej funkcie) je $\Theta(2+\alpha/2-1/2m) = \Theta(1+\alpha)$. □

Ak je počet slotov hašovacej tabuľky úmerný aspoň počtu prvkov v tabuľke, máme $n = O(m)$, a následne $\alpha = n/m = O(m)/m = O(1)$. Teda vyhľadávanie trvá v priemernom prípade konštantný čas. Naviac vieme, že vkladanie aj vymazávanie z hašovacej tabuľky možno realizovať v čase $O(1)$. Z uvedeného vyplýva, že všetky slovníkové operácie môžu v priemernom prípade trvať čas $O(1)$.

4.2.3 Hašovacie funkcie

V tejto časti sa budeme zaoberať niektorými otázkami týkajúcimi sa návrhu dobrých hašovacích funkcií, a potom prezentujeme tri schémy pre ich tvorbu: metódu delenia, metódu násobenia a univerzálne hašovanie.

Dobré hašovacie funkcie

Dobrá hašovacia funkcia spĺňa (približne) predpoklad jednoduchého uniformného hašovania: pravdepodobnosť, že ktorýkoľvek prvok padne do ľubovoľného z m slotov je rovnaká pre všetky prvky. Formálnejšie, predpokladajme, že každý kľúč vyberáme nazávisle z U podľa distribúcie pravdepodobnosti P, t.j. $P(k)$ je pravdepodobnosť, že je vybraný kľúč k. Potom predpoklad jednoduchého uniformného hašovania je

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{pre } j = 0, 1, \ldots, m-1 \; . \tag{13}$$

Nanešťastie vo všeobecnosti nie je možné túto podmienku overiť, nakoľko P zvyčajne nepoznáme.

Niekedy (zriedkavo) poznáme distribúciu P. Napríklad, predpokladajme, že o kľúčoch vieme, že sú to náhodné reálne čísla k nezávisle a uniformne rozdelené na rozsahu $0 \leq k < 1$. V tomto prípade možno ukázať, že hašovacia funkcia

$$h(k) = \lfloor km \rfloor$$

spĺňa podmienku (13).

Kľúče ako prirodzené čísla

Mnoho hašovacích funkcií predpokladá, že univerzum kľúčov je množina prirodzených čísel. Teda, ak kľúče nie sú prirodzenými číslami, treba nájsť spôsob, ako ich interpretovať ako prirodzené čísla. V nasledujúcom budeme predpokladať, že kľúče sú z množiny prirodzených čísel.

Metóda delenia

V ***metóde delenia*** pre vytváranie hašovacích funkcií zobrazujeme kľúč k na jednu z m pozícií určenú hodnotu zvyšku po delení čísla k číslom m. Teda hašovacia funkcia má tvar

$$h(k) = k \bmod m \; .$$

Napríklad, ak hašovacia tabuľka má veľkosť $m = 12$ a kľúč má hodnotu $k = 100$, potom $h(k) = 4$. Keďže je potrebné vykonať iba jednu operáciu delenia, hašovanie delením je dosť rýchle.

Ak používame túto metódu, zvyčajne sa vyhýbame určitým hodnotám m. Napríklad m by sa nemalo rovnať celočíselnej mocnine 2, keďže by tak $h(k)$ nadobúdala hodnoty niekoľkých bitov k najnižšieho rádu. Mocninám 10 by sme sa mali vyhnúť ak aplikácia narába s číslami v desiatkovej sústave, inak by hašovacia funkcia nezávisela na všetkých čísliciach k. Vhodnými hodnotami pre m sú prvočísla nie príliš blízko celočíselným mocninám 2.

Metóda násobenia

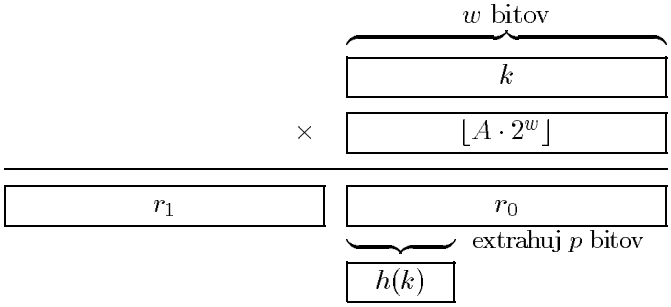
Metóda násobenia pre vytváranie hašovacích funkcií pozostáva z dvoch krokov. Najprv vynásobíme kľúč k konštantou A z rozsahu $0 < A < 1$ a extrahujeme desatinnú časť kA. Potom túto

hodnotu vynásobíme číslom m pričom ako výsledok vezmeme dolnú celú časť. Hašovacia funkcia má teda tvar

$$h(k) = \lfloor m(kA \bmod 1) \rfloor ,$$

kde výraz „$kA \bmod 1$“ označuje desatinnú časť kA, t.j. $kA - \lfloor kA \rfloor$.

Výhodou metódy násobenia je, že hodnota m nie je kritická. Zvyčajne ju pre nenáročnosť realizácie volíme rovnú mocnine 2. Predpokladajme, že šírka slova počítača je w bitov, a že k sa zmestí do jedného slova. Najprv hodnotu k vynásobíme w-bitovým číslom $\lfloor A \cdot 2^w \rfloor$. Výsledkom je $2w$-bitová hodnota $r_1 2^w + r_0$, kde r_1 je slovo vyššieho rádu a r_0 je slovo nižšieho rádu. Požadovaná p-bitová hodnota hašovacej funkcie pozostáva z p najvýznamnejších bitov r_0. Hoci táto metóda funguje pre ľubovoľnú hodnotu konštanty A, možno odporúčať použitie hodnoty napr. $A = (\sqrt{5} - 1)/2$.



Obr. 20: Metóda násobenia. w-bitové číslo reprezentujúce kľúč k je vynásobené w-bitovou hodnotou $\lfloor A \cdot 2^w \rfloor$, kde $0 < A < 1$ je vhodná konštanta. Hodnotu hašovacej funkcie $h(k)$ tvorí p najvyšších bitov dolného slova súčinu.

Univerzálne hašovanie

Pri doterajších metódach hašovania je v najhoršom prípade možné, že všetkých n kľúčov padne do toho istého slotu, čo znamená, že priemerný čas vyhľadávania prvku s daným kľúčom bude $\Theta(n)$. Voči takejto situácii nie je odolná žiadna pevná hašovacia funkcia. Jediným účinným spôsobom ako zlepšiť tento stav je vyberať hašovaciu funkciu *náhodne* bez ohľadu na hodnoty ukladaných kľúčov. Túto metódu nazývame ***univerzálne hašovanie***. V priemere dáva dobré výsledky nezávisle od hodnôt ukladaných kľúčov.

Hlavnou myšlienkou univerzálneho hašovania je zakaždým počas behu programu náhodne vybrať hašovaciu funkciu zo špeciálnej triedy funkcií. Podobne ako v prípade quicksortu, znáhodnenie zaručuje, že neexistuje žiaden vstup, ktorý by vždy spôsobil najhoršie možné správanie algoritmu, a naviac je zabezpečená jeho dobrá efektívnosť v priemernom prípade.

Nech $\mathcal{H}$ je konečná množina hašovacích funkcií, ktoré zobrazujú dané univerzum U kľúčov do množiny $\{0, 1, \ldots, m-1\}$. Hovoríme, že $\mathcal{H}$ je ***univerzálna***, ak pre každú dvojicu navzájom rôznych kľúčov $x, y \in U$ je počet funkcií $h \in \mathcal{H}$, pre ktoré $h(x) = h(y)$, práve $|\mathcal{H}|/m$ [12].

Nasledujúca veta hovorí, že univerzálna trieda hašovacích funkcií dáva v priemernom prípade dobré výsledky.

Veta 4.3 *Ak funkciu h z univerzálnej množiny hašovacích funkcií použijeme na hašovanie n kľúčov do tabuľky veľkosti m, kde $n \leq m$, očakávaný počet kolízií s daným kľúčom x bude menší ako 1.*

Dôkaz. Pre každú dvojicu y, z navzájom rôznych kľúčov označme c_{yz} náhodnú premennú, ktorá nadobúda hodnotu 1 ak $h(y) = h(z)$ (t.j. y a z kolidujú) a hodnotu 0 inak. Keďže dvojica kľúčov koliduje s pravdepodobnosťou $1/m$, máme

$$\mathrm{E}[c_{yz}] = 1/m .$$

[12] Inými slovami, ak hašovaciu funkciu vyberieme náhodne z $\mathcal{H}$, pravdepodobnosť kolízie medzi x a y, pričom $x \neq y$, je presne $|\mathcal{H}|/m$, čo je práve pravdepodobnosť kolízie, ak by boli $h(x)$ a $h(y)$ náhodne vybrané z množiny $\{0, 1, \ldots, m-1\}$.

Nech C_x je celkový počet kolízií s prvkom x v hašovacej tabuľke T veľkosti m obsahujúcej n kľúčov. Dostávame tak

$$\mathrm{E}[C_x] = \sum_{\substack{y \in T \\ y \neq x}} \mathrm{E}[c_{xy}] = \frac{n-1}{m} ,$$

z čoho, za predpokladu, že $n \leq m$, máme $\mathrm{E}[C_x] < 1$. □

Teraz uvedieme spôsob ako navrhnúť univerzálnu množinu hašovacích funkcií. Nech veľkosť m tabuľky je prvočíslo (podobne ako pri metóde delenia). Kľúč x rozložíme na $r+1$ bajtov tak, aby $x = \langle x_0, x_1, \ldots, x_r \rangle$, pričom $x_i < m$. Nech $a = \langle a_0, a_1, \ldots, a_r \rangle$ je postupnosť $r+1$ prvkov vybraných náhodne z množiny $\{0, 1, \ldots, m-1\}$. Zadefinujeme príslušnú hašovaciu funkciu $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m . \tag{14}$$

Podľa tejto definície má množina

$$\mathcal{H} = \bigcup_a \{h_a\} \tag{15}$$

m^{r+1} prvkov.

Veta 4.4 *Množina $\mathcal{H}$ definovaná vzťahmi (14) a (15) je univerzálnou triedou hašovacích funkcií.*

Dôkaz. Uvažujme dvojicu x, y navzájom rôznych kľúčov. Predpokladajme, bez újmy na všeobecnosti, že $x_0 \neq y_0$. Pre ľubovoľné pevné hodnoty $a_1, a_2, \ldots, a_r$ existuje práve jedna hodnota a_0 taká, že $h(x) = h(y)$:

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod m .$$

Keďže m je prvočíslo, uvedená rovnosť má práve jedno riešenie pre a_0 modulo m. Preto každá dvojica kľúčov x a y koliduje práve pre m^r hodnôt a, nakoľko x a y kolidujú práve raz pre ľubovoľné hodnoty $\langle a_1, a_2, \ldots, a_r \rangle$ (pre jedinú hodnotu a_0 uvedenú vyššie). Keďže existuje m^{r+1} možných postupností a, kľúče x a y kolidujú s pravdepodobnosťou $m^r/m^{r+1} = 1/m$, čo znamená, že množina $\mathcal{H}$ je univerzálna. □

4.2.4 Otvorená adresácia

Pri ***otvorenej adresácii*** sú všetky prvky uložené v samotnej hašovacej tabuľke, t.j. každá položka tejto tabuľky buď obsahuje prvok dynamickej množiny alebo NIL. Ak hľadáme daný prvok, systematicky prechádzame pozície tabuľky pokým ho nenájdeme alebo nie je zrejmé, že v sa tabuľke nenachádza. Keďže nepoužívame žiadne zoznamy ani nemáme žiadne prvky uložené mimo tabuľky, hašovacia tabuľka sa môže zaplniť tak, že už nebude možné do nej vkladať ďalšie prvky (ukladací faktor α je teda vždy menší alebo rovný 1).

Výhodou otvorenej adresácie je, že sa pri nej úplne predchádza použitiu ukazovateľov. Namiesto toho *vypočítavame* postupnosť pozícií tabuľky, ktoré sa majú preskúmať.

Pre vloženie prvku do tabuľky pomocou otvorenej adresácie, postupne vyšetrujeme alebo ***sondujeme*** hašovaciu tabuľku až pokým nenájdeme voľnú pozíciu, do ktorej je možné tento prvok uložiť. Namiesto postupného prechádzania pozícií tabuľky, ich poradie závisí od hodnoty vkladaného kľúča. Aby sme mohli určiť, ktoré sloty treba otestovať, rozšírime doménu hašovacej funkcie o číslo testu. Teda hašovacia funkcia má teraz tvar

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} .$$

Pri otvorenej adresácii požadujeme, aby pre každý kľúč k bola ***sondovacia postupnosť***

$$\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$$

permutáciou $\langle 0, 1, \ldots, m-1 \rangle$, čo má za úlohu zabezpečiť, aby bola každá pozícia hašovacej tabuľky prístupná pre ľubovoľný vkladaný kľúč. V nasledujúcom kóde predpokladáme, že prvky hašovacej tabuľky T sú kľúče bez satelitných informácií. Každý slot teda obsahuje buď kľúč alebo hodnotu NIL (ak je slot prázdny).

HASH-INSERT(T, k)

```
1  i ← 0
2  repeat
3     j ← h(k, i)
4     if T[j] = NIL
5       then T[j] ← k
6            return j
7       else i ← i + 1
8  until i = m
9  error "hash table overflow"
```

Algoritmus pre vyhľadávanie kľúča k prechádza tie isté pozície tabuľky, ktoré vyšetroval algoritmus vkladania počas pridávania kľúča k. Preto sa vyhľadávanie môže ukončiť (neúspešne), ak sa nájde prázdna pozícia. Procedúra HASH-SEARCH vezme ako vstup hašovaciu tabuľku T a kľúč k, pričom vráti hodnotu j, ak nájde pozíciu j obsahujúcu kľúč k, alebo NIL, ak sa kľúč k v tabuľke T nenachádza.

HASH-SEARCH(T, k)

```
1  i ← 0
2  repeat
3     j ← h(k, i)
4     if T[j] = k then return j
5     i ← i + 1
6  until (T[j] = NIL) or (i = m)
7  return NIL
```

Vymazávanie z hašovacej tabuľky s otvorenou adresáciou je náročnejšie. Keď vymazávame kľúč zo slotu i, nemôžeme ho jednoducho označiť ako prázdny jednoduchým uložením hodnoty NIL. Spôsobilo by to totiž, že by sa mohli zneprístupniť aj ďalšie kľúče tabuľky. Jedným z riešení je označenie slotu uložením špeciálnej hodnoty DELETED namiesto NIL. Potom je však treba pozmeniť kód procedúry HASH-SEARCH tak, aby pokračovala ďalej ak nájde hodnotu DELETED, a kód HASH-INSERT, aby takto označené pozície uvažovala ako voľné, do ktorých možno nový kľúč vložiť. Ak budeme z tabuľky vymazávať prvky týmto spôsobom, časy vyhľadávania už nebudú závisieť od ukladacieho faktoru α. Z tohto dôvodu sa preto častejšie používa hašovanie s reťazením.

V našej analýze uvažujeme ***uniformné hašovanie***: predpokladáme, že pre každý kľúč je rovnako pravdepodobná ľubovoľná permutácia $\{0, 1, \ldots, m-1\}$ ako sondovacia postupnosť. Uniformné hašovanie je zovšeobecnením jednoduchého uniformného hašovania definovaného vyššie. Skutočné uniformné hašovanie je ťažké implementovať, hoci v praxi používame isté priblíženia (napr. dvojité hašovanie definované ďalej).

Ná výpočet sondovacích postupností sa používajú tri techniky: lineárne sondovanie, kvadratické sondovanie a dvojité hašovanie. Tieto techniky zaručujú, aby pre každý kľúč k bola $\langle h(k,1), h(k,2), \ldots, h(k,m) \rangle$ permutáciou $\langle 0, 1, \ldots, m-1 \rangle$, ale žiadna z nich nespĺňa podmienku uniformného hašovania (ani jedna nie je schopná vygenerovať viac ako m^2 navzájom rôznych sondovacích postupností, namiesto potrebných $m!$).

Lineárne sondovanie

Nech $h' : U \to \{0, 1, \ldots, m-1\}$ je obyčajná hašovacia funkcia. Metóda ***lineárneho sondovania*** používa hašovaciu funkciu

$$h(k, i) = (h'(k) + i) \bmod m$$

pre $i = 0, 1, \ldots, m-1$. Pre daný kľúč k je $T[h'(k)]$ prvou sondovanou pozíciou. Ďalšou pozíciou je $T[h'(k)+1]$, atď. až po $T[m-1]$. Ďalej pokračuje pozíciami $T[0], T[1], \ldots$, až nakoniec je to $T[h'(k)-1]$. Keďže počiatočná pozícia určuje celú sondovaciu postupnosť, celkový počet sondovacích postupností je preto iba m.

Lineárne sondovanie sa ľahko implementuje, ale vzniká pri ňom tzv. ***primárne clustrovanie*** (prvky majú sklon zhlukovať sa okolo primálnych kľúčov, t.j. kľúčov, ktoré pri ich vložení nespôsobili kolíziu). Pri použití tejto metódy sa vytvárajú dlhé úseky obsadených pozícií, čím sa zvyšuje priemerný čas vyhľadávania. Z tohto dôvodu lineárne sondovanie nie je vhodnou aproximáciou uniformného hašovania.

Kvadratické sondovanie

Kvadratické sondovanie používa hašovaciu funkciu tvaru

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \,,$$

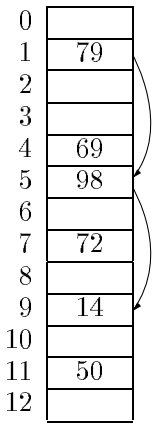
kde (podobne ako pri lineárnom sondovaní) h' je pomocná hašovacia funkcia, c_1 a $c_2 \neq 0$ sú konštanty a $i = 0, 1, \ldots, m-1$. Počiatočná sondovaná pozícia je $T[h'(k)]$, pričom ďalšie pozície závisia kvadraticky od poradia sondovania i. Hoci táto metóda pracuje omnoho lepšie ako lineárna, vzniká miernejšia forma clustrovania, tzv. ***sekundárne clustrovanie***. Podobne ako pri lineárnom sondovaní, počiatočná sondovaná pozícia určuje celú postupnosť, teda celkový počet navzájom rôznych sondovacích postupností je opäť iba m.

Dvojité hašovanie

Dvojité hašovanie je jednou z najlepších metód pre otvorenú adresáciu pretože vytvárané permutácie sa javia veľmi náhodné. ***Dvojité hašovanie*** používa hašovaciu funkciu tvaru

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m \,,$$

kde h_1 a h_2 sú pomocné hašovacie funkcie. Počiatočná sondovaná pozícia je $T[h_1(k)]$, pričom ďalšie pozície sú posunuté o $h_2(k)$ modulo m. Teda sondovacia postupnosť závisí od k dvoma spôsobmi: od počiatočnej pozície a od posunu. Obr. 21 znázorňuje vkladanie prvku do tabuľky použitím dvojitého hašovania.



Obr. 21: Vkladanie pomocou dvojitého hašovania. Máme tabuľku pozostávajúcu z 13 prvkov, pričom $h_1(x) = k \bmod 13$ a $h_2(k) = 1 + (k \bmod 11)$. Nakoľko $14 \equiv 1 \pmod{13}$ a $14 \equiv 3 \pmod{11}$, kľúč 14 bude vložený na prázdnu pozíciu 9, potom ako boli preskúmané sloty 1 a 5.

Aby bolo možné vždy prehľadať celú tabuľku, musí byť hodnota $h_2(k)$ nesúdeliteľná s veľkosťou tabuľky m. Ak by totiž m a $h_2(k)$ mali najväčšieho spoločného deliteľa $d > 1$ pre nejaký kľúč k, potom by vyhľadávanie kľúča k otestovalo iba $1/d$ celej hašovacej tabuľky. Jednoduchým spôsobom, ako zabezpečiť túto podmienku, je položiť m rovné nejakej mocnine 2 a zostrojiť funkciu h_2

tak, aby vždy vracala nepárnu hodnotu. Ďalším spôsobom je položiť m rovné nejakému prvočíslu a navrhnúť h_2 tak, aby vždy vracala celočíselnú hodnotu menšiu ako m.

Dvojité hašovanie je výrazným zlepšením v porovnaní s lineárnym alebo kvadratickým sondovaním najmä v tom, že je použitých $\Theta(m^2)$ sondovacích postupností namiesto $\Theta(m)$. Výsledkom je, že dvojité hašovanie má vlastnosti blížiace sa ideálnemu uniformnému hašovaniu.

Analýza otvorenej adresácie

Naša analýza otvorenej adresácie sa zakladá, podobne ako analýza hašovania s reťazením, na hodnote ukladacieho koeficientu α hašovacej tabuľky, pričom počet n uložených prvkov a veľkosť tabuľky m sa blížia nekonečnu. Keďže pri otvorenej adresácii pripadá na jednu pozíciu nanajvýš jeden prvok, je $n \leq m$, z čoho vyplýva $\alpha \leq 1$.

Predpokladáme, že je použité uniformné hašovanie, t.j. že sondovacou postupnosťou $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ pre ľubovoľný kľúč k je s rovnakou pravdepodobnosťou hociktorá z možných permutácií $\langle 0,1,\ldots,m-1\rangle$.

Veta 4.5 *Nech je daná hašovacia tabuľka s otvorenou adresáciou, pričom jej ukladací koeficient je $\alpha = n/m < 1$. Očakávaný počet pokusov neúspešného vyhľadávania je nanajvýš $1/(1-\alpha)$ za predpokladu uniformného hašovania.*

Dôkaz. Nech p_i je pravdepodobnosť, že pri neúspešnom hľadaní bolo obsadených práve i pozícií, kde $i = 0,1,2,\ldots$. Pre $i > n$ máme zrejme $p_i = 0$, teda očakávaný počet pokusov je

$$1 + \sum_{i=0}^{\infty} i\,p_i \ .$$

Aby sme mohli vyhodnotiť tento výraz položme q_i rovné pravdepodobnosti, že pri neúspešnom hľadaní bolo obsadených aspoň i pozícií. Dostávame tak

$$\sum_{i=0}^{\infty} i\,p_i = \sum_{i=1}^{\infty} q_i \ .$$

Pravdepodobnosť, že prvý pokus pristúpi na obsadenú pozíciu je n/m, teda

$$q_1 = \frac{n}{m} \ .$$

Pri uniformnom hašovaní druhý pokus, ak je potrebný, zasiahne jednu zo zvyšných $m-1$ pozícií, z ktorých je $n-1$ obsadených. Druhý pokus vykonáme iba v prípade, že prvý bol neúspešný, teda

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \ .$$

Vo všeobecnosti je i-ty pokus vykonaný iba ak prvých $i-1$ pokusov bolo neúspešných a je rovnako pravdepodobné, že pristúpi ku ktorejkoľvek zo zvyšných $m-i+1$ pozícií, z ktorých je $n-i+1$ obsadených. Máme teda

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\leq \left(\frac{n}{m}\right)^i = \alpha^i \end{aligned}$$

pre $i = 1,2,\ldots,n$, nakoľko $(n-j)/(m-j) \leq n/m$ ak $n \leq m$ a $j \geq 0$.

Za predpokladu, že $\alpha < 1$ je priemerný počet pokusov pri neúspešnom vyhľadávaní

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i\,p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\ &= \frac{1}{1-\alpha} \ . \end{aligned}$$

Tento vzťah možno intuitívne interpretovať nasledovne: jeden pokus je uskutočnený vždy, druhý pokus treba vykonať s pravdepodobnosťou približne α, tretí s pravdepodobnosťou α^2, atď. □

Dôsledok 4.1 *Vloženie prvku do hašovacej tabuľky s otvorenou adresáciou a ukladacím faktorom α vyžaduje v priemernom prípade nanajvýš $1/(1-\alpha)$ pokusov za predpokladu uniformného hašovania.*

Veta 4.6 *Majme hašovaciu tabuľku s otvorenou adresáciou a ukladacím koeficientom $\alpha < 1$. Očakávaný počet pokusov pri úspešnom vyhľadávaní je nanajvýš*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha} ,$$

za predpokladu, že každý kľúč je hľadaný s rovnakou pravdepodobnosťou, pričom sa opäť uvažuje uniformné hašovanie.

Dôkaz. Vyhľadávanie kľúča k sleduje tú istú sondovaciu postupnosť ako keď bol kľúč k vkladaný. Podľa dôsledku 4.1, ak k bol $(i+1)$-vý kľúč vložený do hašovacej tabuľky, očakávaný počet pokusov uskutočnených počas hľadania k je nanajvýš $1/(1-i/m) = m/(m-i)$. Spriemerovanie cez všetkých n kľúčov v tabuľke nám dáva priemerný počet pokusov pri úspešnom vyhľadávaní:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

kde $H_n = \sum_{i=1}^{n} 1/i$ je n-te harmonické číslo. Použitím ohraničenia $\ln n \leq H_n \leq \ln n + 1$ dostávame

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &\leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} + \frac{1}{\alpha} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha} \end{aligned}$$

pre ohraničenie očakávaného počtu pokusov v prípade úspešného vyhľadávania. □

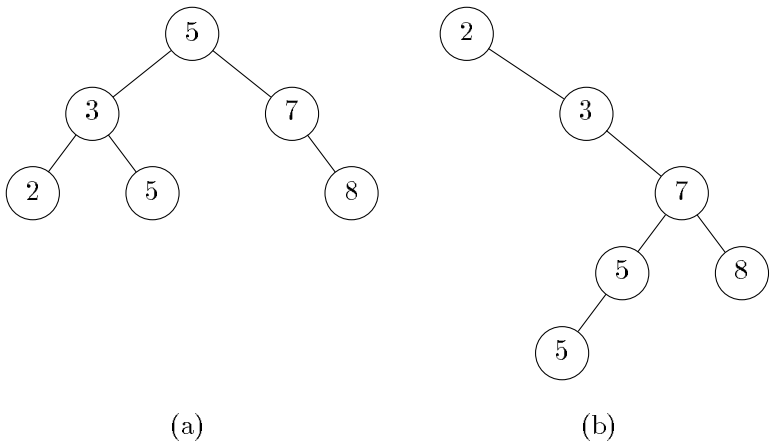
4.3 Binárne prehľadávacie stromy

Binárne prehľadávacie stromy sú dátové štruktúry podporujúce viaceré operácie na dynamických množinách vrátane SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT a DELETE. Možno ich preto použiť aj ako slovník alebo ako prioritnú frontu.

Základné operácie na binárnom prehľadávacom strome trvajú čas úmerný výške stromu. V prípade úplného binárneho stromu s n uzlami tieto operácie trvajú v najhoršom prípade čas $\Theta(\lg n)$. Ak každý prvok v strome má nanajvýš jedného potomka (strom je lineárny), tie isté operácie trvajú v najhoršom prípade čas $\Theta(n)$.

4.3.1 Vlastnosti binárnych prehľadávacích stromov

Binárny prehľadávací strom je usporiadaný, ako naznačuje jeho názov, binárny strom. Takýto strom možno reprezentovať pomocou spájanej dátovej štruktúry, ktorá v každom svojom uzle obsahuje dáta. Okrem položky *key* každý uzol obsahuje položky *left*, *right* a p, ktoré ukazujú na jeho ľavého potomka, pravého potomka, resp. jeho rodiča. Ak potomok alebo rodič chýba, príslušná položka obsahuje hodnotu NIL. Koreň je jediným uzlom, ktorého položka ukazujúca na rodiča obsahuje NIL.



Obr. 22: Binárne prehľadávacie stromy. Pre každý uzol x sú kľúče v jeho ľavom podstrome menšie alebo rovné $key[x]$ a kľúče v jeho pravom podstrome väčšie alebo rovné $key[x]$. Rôzne binárne prehľadávacie stromy môžu reprezentovať rovnaké množiny hodnôt. Najhorší čas behu väčšiny operácií na prehľadávacom strome je úmerný jeho výške. (**a**) Binárny prehľadávací strom pozostávajúci zo 6 uzlov, ktorý má výšku 2. (**b**) Menej efektívny binárny prehľadávací strom výšky 4, ktorý obsahuje tie isté kľúče.

Kľúče binárneho prehľadávacieho stromu sú vždy uložené takým spôsobom, aby spĺňali nasledujúcu vlastnosť:

> Nech x je uzol binárneho prehľadávacieho stromu. Ak y je uzol v ľavom podstrome x, potom $key[y] \leq key[x]$. Ak y je uzol v pravom podstrome uzla x, potom $key[y] \geq key[x]$.

Vlastnosť binárnych prehľadávacích stromov nám umožňuje vypísať všetky kľúče stromu v utriedenom poradí jednoduchým rekurzívnym algoritmom ***inorder tree walk***. Názov tohto algoritmu je odvodený z toho, že hodnota kľúča rodiča podstromu je vypísaná medzi hodnotami v ľavom podstrome a hodnotami v pravom podstrome. (Analogicky, ***preorder tree walk*** vypíše hodnotu koreňa pred hodnotami v oboch podstromoch a ***postorder tree walk*** ju vypíše po hodnotách v podstromoch.) Pre výpis všetkých prvkov binárneho prehľadávacieho stromu v utriedenom poradí treba vyvolať nasledujúcu procedúru s parametrom *root[T]*.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL then
2     INORDER-TREE-WALK(left[x])
3     print key[x]
4     INORDER-TREE-WALK(right[x])
```

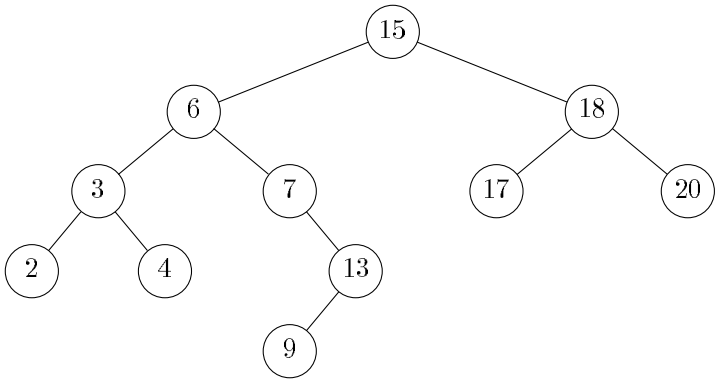
Napríklad pre obidva stromy na obrázku 22 procedúra INORDER-TREE-WALK vypíše kľúče v poradí $2, 3, 5, 5, 7, 8$. Čas behu tejto procedúry pre strom obsahujúci n uzlov je $\Theta(n)$.

4.3.2 Dotazy na binárne prehľadávacie stromy

Najbežnejšou operáciou vykonávanou na binárnych prehľadávacích stromoch je vyhľadávanie kľúča uloženého v strome. Okrem operácie SEARCH tieto stromy môžu podporovať také dotazy ako MINIMUM, MAXIMUM, SUCCESSOR a PREDECESSOR. V tejto časti sa budeme zaoberať týmito operáciami a ukážeme, že každú možno implementovať tak, aby čas jej behu bol $O(h)$ na strome výšky h.

Vyhľadávanie

Na vyhľadávanie uzla s daným kľúčom používame nasledujúcu procedúru. Za predpokladu, že



Obr. 23: Dotazy na binárny prehľadávací strom. Aby sa v strome našiel kľúč 13, treba od koreňa prejsť po ceste $15 \to 6 \to 7 \to 13$. Najmenšia hodnota kľúča v strome je 2. Nájsť ju je možné tak, že od koreňa sa vždy budeme vyberať smerom vľavo. Najväčšia hodnota je 20, a možno ju nájsť analogicky cestou smerom vpravo. Nasledovník uzla s kľúčom 15 je uzol s kľúčom 17, keďže je to minimálny kľúč v pravom podstrome 15. Uzol s kľúčom 14 nemá pravý podstrom, a teda jeho nasledovník je jeho najnižší predok, ktorého ľavý potomok je tiež predok. V tomto prípade, je jeho nasledovníkom uzol s kľúčom 15.

máme kľúč k a ukazovateľ na koreň stromu, TREE-SEARCH vráti ukazovateľ na uzol s kľúčom k, ak existuje, inak vráti NIL.

TREE-SEARCH(x, k)

```
1  if (x = NIL) or (k = key[x]) then
2    return x
3  if k < key[x]
4    then return TREE-SEARCH(left[x], k)
5    else return TREE-SEARCH(right[x], k)
```

Procedúra začína hľadať v koreni a postupne prechádza strom smerom dolu, ako možno vidieť na obr. 23. Pre každý uzol x, na ktorý narazí, porovná kľúč k s kľúčom $key[x]$. Ak sú rovnaké, vyhľadávanie sa ukončí. Ak je k menšie ako $key[x]$, vyhľadávanie pokračuje v ľavom podstrome x, nakoľko vlastnosť binárneho prehľadávacieho stromu zaručuje, že k nemôže byť v pravom podstrome x. Symetricky, ak k je väčšie ako $key[x]$, vyhľadávanie pokračuje v pravom podstrome x. Uzly navštívené počas rekurzie tvoria cestu od koreňa smerom dolu, a teda čas behu TREE-SEARCH je $O(h)$, kde h je výška stromu.

Túto istú procedúru možno zapísať iteratívne náhradou rekurzie za cyklus `while`. Na väčšine počítačov je táto verzia efektívnejšia.

ITERATIVE-TREE-SEARCH(x, k)

```
1  while (x ≠ NIL) and (k ≠ key[x]) do
2    if k < key[x]
3      then x ← left[x]
4      else x ← right[x]
5  return x
```

Minimum a maximum

Prvok, ktorého hodnota kľúča je minimum, možno vždy nájsť sledovaním ukazovateľov na ľavých potomkov, až pokým nenarazíme na NIL. Nasledujúca procedúra vracia ukazovateľ na najmenší prvok v podstrome s koreňom v uzle x.

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL do
2     x ← left[x]
3  return x
```

Vlastnosť binárnych prehľadávacích stromov zaručuje, že TREE-MINIMUM je korektná. Ak uzol x nemá ľavý podstrom, potom minimum podstromu s koreňom x je $key[x]$. Ak naopak x má ľavý podstrom, potom keďže žiaden prvok napravo od neho nie je menší ako $key[x]$ a každý prvok naľavo nie je väčší ako $key[x]$, potom najmenší kľúč možno nájsť v podstrome s koreňom v uzle $left[x]$. Pseudokód pre TREE-MAXIMUM je analogický. Obe tieto procedúry bežia na strome výšky h v čase $O(h)$.

Nasledovník a predchodca

Pre daný uzol binárneho prehľadávacieho stromu je niekedy potrebné nájsť jeho nasledovníka v utriedenom poradí. Štruktúra binárneho prehľadávacieho stromu nám umožňuje nájsť nasledovníka uzla bez akéhokoľvek porovnávania kľúčov. Nasledujúca procedúra vracia nasledovníka uzla x strome, ak existuje, alebo NIL, ak x má najväčšiu hodnotu kľúča v strome.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL then
2     return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while (y ≠ NIL) and (x = right[y]) do
5     x ← y
6     y ← p[y]
7  return y
```

Kód procedúry TREE-SUCCESSOR je rozdelený na dva prípady. Ak pravý podstrom uzla x je neprázdny, potom nasledovník x je najľavejší uzol v pravom podstrome, ktorý možno nájsť vyvolaním TREE-MINIMUM($right[x]$). Ak, na druhej strane, je pravý podstrom x prázdny a x má nasledovníka y, potom je y najnižší predok x, ktorého ľavý syn je tiež predok x. Aby sme našli y, jednoducho ideme smerom hore od x až pokým nenarazíme na uzol, ktorý je ľavým synom svojho otca.

Čas behu TREE-SUCCESSOR na strome výšky h je $O(h)$, nakoľko buď prechádzame strom smerom iba nahor alebo iba nadol. Procedúra TREE-PREDECESSOR, ktorá je symetrická ku procedúre TREE-SUCCESSOR tiež beží v čase $O(h)$.

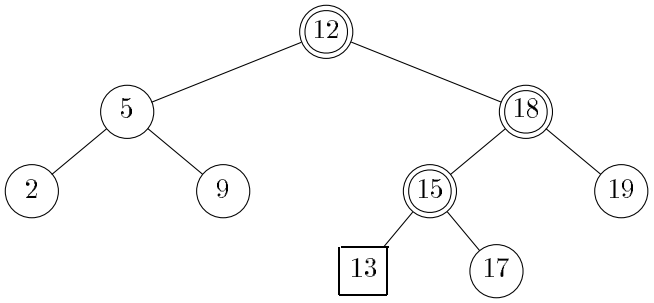
Veta 4.7 *Operácie* SEARCH, MINIMUM, MAXIMUM, SUCCESSOR *a* PREDECESSOR *možno implementovať tak, aby čas ich behu na binárnom prehľadávacom strome výšky h bol $O(h)$.*

4.3.3 Vkladanie a vymazávanie

Operácie vkladania a vymazávania spôsobujú, že dynamická množina reprezentovaná binárnym prehľadávacím stromom sa mení. Dátová štruktúra musí byť však modifikovaná takým spôsobom, aby sa zachovala vlastnosť binárneho prehľadávacieho stromu.

Vkladanie

Pre vloženie novej hodnoty v do binárneho prehľadávacieho stromu T použijeme procedúru TREE-INSERT. Procedúre je predaný uzol z, pre ktorý $key[z] = v$, $left[z] =$ NIL a $right[z] =$ NIL. Táto procedúra potom modifikuje T a niektoré položky z takým spôsobom, aby z mohol byť vložený na príslušnú pozíciu v strome.



Obr. 24: Vkladanie prvku s kľúčom 13 do binárneho prehľadávacieho stromu. Uzly označené dvojitými kružnicami označujú cestu od koreňa smerom dole ku pozícii, kam je položka vkladaná. Pri vkladaní bolo vytvorené spojenie medzi uzlami 15 a 13.

TREE-INSERT(T, x)

```
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL do
4     y ← x
5     if key[z] < key[x]
6       then x ← left[x]
7       else x ← right[x]
8  p[z] ← y
9  if y = NIL
10    then root[T] ← z
11    else if key[z] < key[y]
12            then left[y] ← z
13            else right[y] ← z
```

TREE-DELETE(T, z)

```
1  if (left[z] = NIL) or (right[z] = NIL)
2    then y ← z
3    else y ← TREE-SUCCESSOR(z)
4  if left[y] ≠ NIL
5    then x ← left[y]
6    else x ← right[y]
7  if x ≠ NIL
8    then p[x] ← p[y]
9  if p[y] = NIL
10   then root[T] ← x
11   else if y = left[p[y]]
12           then left[p[y]] ← x
13           else right[p[y]] ← x
14 if y ≠ z then
15   key[z] ← key[y]
16   ▷ Ak má y iné položky,
17   ▷ treba ich tiež skopírovať.
18 return y
```

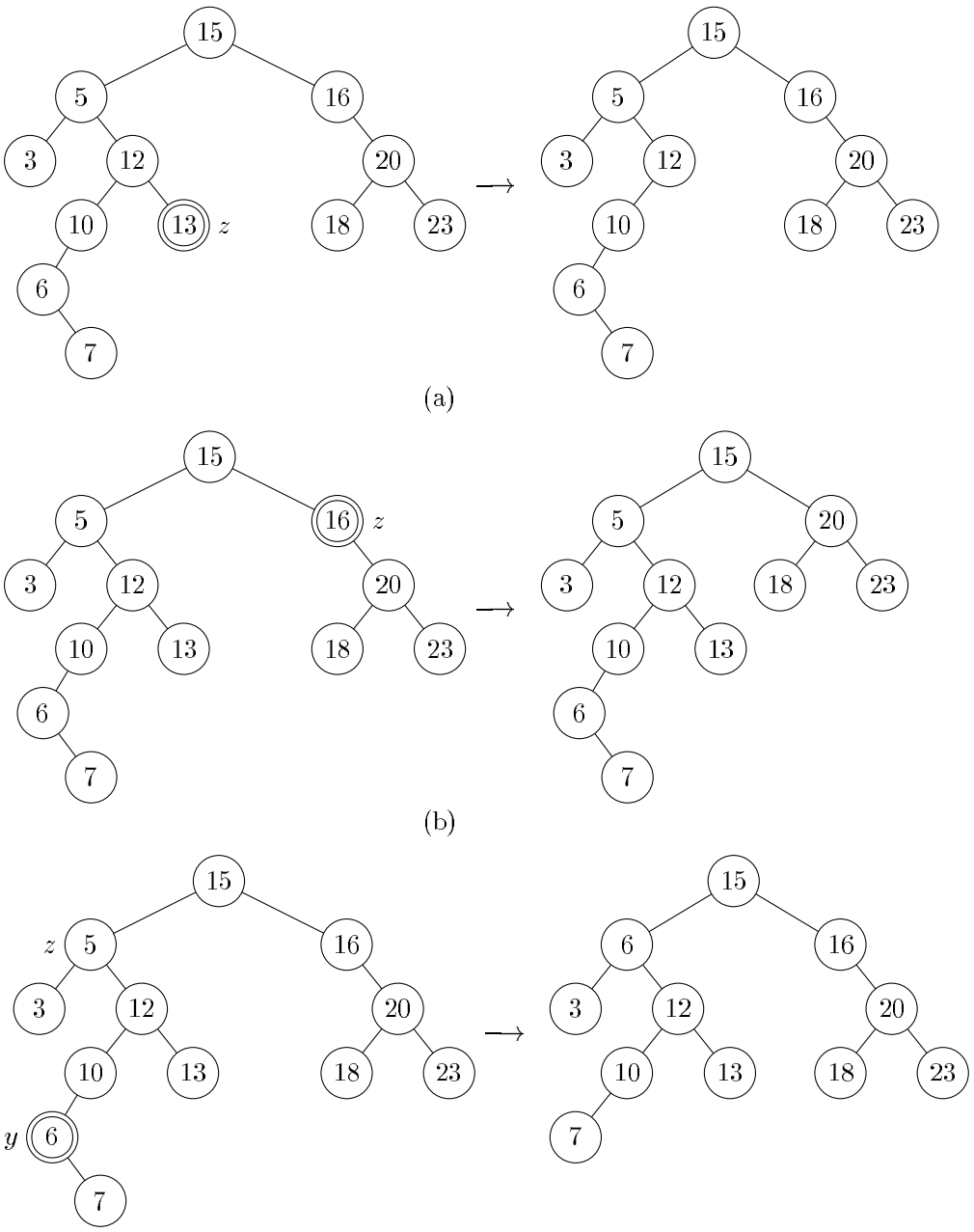
Obrázok 24 znázorňuje činnosť procedúry TREE-INSERT. Podobne ako procedúra TREE-SEARCH, TREE-INSERT začína v koreni a sleduje cestu smerom nadol. Ukazovateľ x postupne prechádza túto cestu, pričom y je stále nastavený na jeho otca. Po inicializácii, cyklus `while` na riadkoch 3–7 bude pohybovať ukazovateľmi x a y smerom nadol až pokým x nebude mať hodnotu NIL, pričom pokračovanie v ľavom alebo pravom synovi bude závisieť od výsledku porovnania kľúčov $key[z]$ a $key[x]$. Na riadkoch 8–13 je prvok z vložený do stromu jednoduchým nastavením ukazovateľov.

Rovnako ako ostatné jednoduché operácie na prehľadávacích stromoch, aj procedúra TREE-INSERT beží v čase $O(h)$, kde h je výška daného stromu.

Vymazávanie

Procedúra pre vymazávanie daného uzla z z binárneho prehľadávacieho stromu má ako parameter ukazovateľ na z. Táto procedúra uvažuje tri prípady, ako možno vidieť na obr. 25. Ak z nemá potomkov, modifikujeme jeho otca $p[z]$ náhradou ukazovateľa na z hodnotou NIL. Ak má vymazávaný uzol iba jedného potomka, odstránime ho vytvorením spojenia medzi jeho rodičom a jeho synom. Nakoniec, ak má uzol dvoch potomkov, odstránime nasledovníka y uzla z, ktorý určite nemá ľavého syna, a nahradíme obsah z obsahom uzla y.

Kód procedúry TREE-DELETE (uvedená už vyššie) organizuje tieto tri prípady trochu odlišným spôsobom. Na riadkoch 1–3, algoritmus určí uzol y, ktorý treba zo stromu odstrániť. Týmto uzlom



Obr. 25: Vymazávanie uzla z z binárneho prehľadávacieho stromu. V každom z prípadov je vymazávaný uzol označený dvojitou kružnicou. (**a**) Ak z nemá potomkov, jednoducho ho vymažeme. (**b**) Ak z má jediného potomka, odstránime ho tak, že jeho syn bude teraz synom jeho otca. (**c**) Ak má z dvoch potomkov, zo stromu odstránime jeho nasledovníka y, ktorý má nanajvýš jedného potomka, a potom obsah z nahradíme pôvodným obsahom uzla y.

je buď vstupný uzol z (ak z má nanajvýš jedného potomka) alebo nasledovník z (ak z má dvoch potomkov). Potom, na riadkoch 4–6, je x nastavený na potomka uzla y alebo na NIL, ak y nemá potomkov. Uzol y je na riadkoch 7–13 odstránený zo stromu zmenou ukazovateľov $p[y]$ a x. Je to však trochu komplikované, v prípade okrajových podmienok, ktoré nastávajú keď $x = \text{NIL}$ alebo keď y je koreň. Nakoniec, na riadkoch 14–17, ak bol odstránený nasledovník uzla z, je jeho obsahom prepísaný pôvodný obsah z. Uzol y je vrátený na riadku 18, teda volajúca procedúra

ho môže opäť použiť (napr. zaradiť do zoznamu voľných prvkov). Procedúra TREE-DELETE beží v čase $O(h)$ na strome výšky h.

Veta 4.8 *Operácie* INSERT *a* DELETE *na dynamickej množine možno implementovať pomocou binárneho prehľadávacieho stromu výšky h tak, že ich čas behu je $O(h)$.*

4.4 Červeno-čierne stromy

Už sme ukázali, že pomocou binárneho prehľadávacieho stromu výšky h je možné implementovať základné operácie na dynamických množinách s časom behu $O(h)$. Teda, množinové operácie sú rýchle ak výška prehľadávacieho stromu je malá, ale ak je táto výška veľká (relatívne ku počtu prvkov stromu), ich účinnosť nemusí byť vôbec väčšia ako pri použití obyčajného spájaného zoznamu. Výhodou červeno-čiernych stromov je ich vyváženosť, ktorá zaručuje, aby boli časy behu týchto operácií nanajvýš rovné $O(\lg n)$ v najhoršom prípade, kde n je počet uzlov červeno-čierneho stromu.

4.4.1 Vlastnosti červeno-čiernych stromov

Červeno-čierny strom (ďalej RB-strom) je binárny prehľadávací strom, v ktorom každý uzol je rozšírený naviac o jeden bit informácie reprezentujúci ***farbu*** tohto uzla, ktorá môže byť buď červená (RED) alebo čierna (BLACK). Zavedením určitých pravidiel, podľa ktorých možno uzly na ľubovoľnej z ciest od koreňa k listom ofarbiť iba určitým spôsobom, je možné zaistiť, aby žiadna takáto cesta nebola viac ako dvakrát tak dlhá ako ktorákoľvek iná, t.j. aby bol strom aspoň približne ***vyvážený***.

Každý uzol teraz obsahuje položky *color*, *key*, *left*, *right* a p. Ak potomok alebo rodič uzla neexistuje, príslušná položka obsahuje hodnotu NIL. Tieto hodnoty NIL budeme chápať ako ukazovatele na listy binárneho prehľadávacieho stromu, ktoré však neobsahujú žiadne údaje.

Binárny prehľadávací strom je červeno-čierny, ak spĺňa nasledujúce podmienky:

1. Každý uzol je buď červený alebo čierny.
2. Každý list (NIL) je čierny.
3. Ak je uzol červený, obaja jeho synovia sú čierni.
4. Všetky jednoduché cesty vedúce od daného uzla k listu obsahujú rovnaký počet čiernych uzlov.

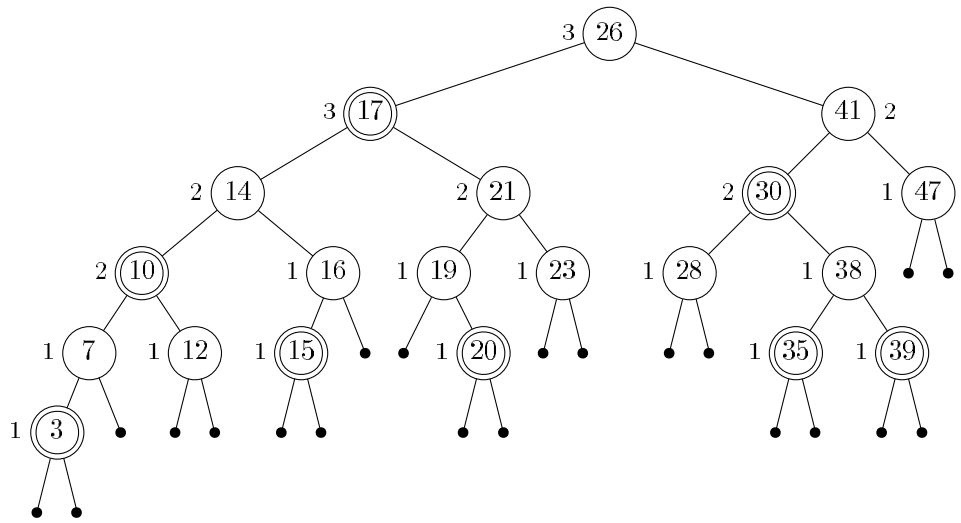
Príklad RB-stromu je uvedený na obr. 26.

Počet čiernych uzlov na ľubovoľnej z ciest od uzla x k listu (uzol x nezarátavame) nazývame ***čiernou výškou*** uzla. Označujeme ju $\mathrm{bh}(x)$. Čiernu výšku RB-stromu definujeme ako čiernu výšku jeho koreňa.

Lema 4.1 *Červeno-čierny strom s n vnútornými uzlami má výšku nanajvýš $2\lg(n+1)$.*

Dôkaz. Najprv ukážeme, že podstrom s koreňom v ľubovoľnom uzle x obsahuje najmenej $2^{\mathrm{bh}(x)}-1$ vnútorných uzlov. Toto pomocné tvrdenie dokážeme indukciou na výšku x. Ak je táto výška 0, potom x musí byť list (NIL). Strom pozostávajúci z jediného listu samozrejme obsahuje aspoň $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ vnútorných uzlov. Uvažujme teraz uzol x, ktorý má kladnú výšku, a je vnútorným uzlom s dvoma potomkami. Obaja synovia majú čiernu výšku buď $\mathrm{bh}(x)$ alebo $\mathrm{bh}(x)-1$, v závislosti od ich farby. Ak je červená, čierna výška syna je $\mathrm{bh}(x)$, ak je čierna, čierna výška syna je $\mathrm{bh}(x)-1$. Nakoľko výška syna uzla x je menšia ako výška x samotného, môžeme aplikovať induktívnu hypotézu, aby sme mohli tvrdiť, že obaja synovia majú aspoň $2^{\mathrm{bh}(x)-1}-1$ vnútorných uzlov. Teda podstrom s koreňom v uzle x obsahuje aspoň $2^{\mathrm{bh}(x)-1}-1)+(2^{\mathrm{bh}(x)-1}-1)+1 = 2^{\mathrm{bh}(x)}-1$ vnútorných uzlov, čím sme dokázali pomocné tvrdenie.

Nech h je teraz výška stromu. Z vlastnosti 3 vyplýva, že aspoň polovica uzlov na ľubovoľnej jednoduchej ceste od koreňa k listu musí byť čierna (samozrejme s výnimkou koreňa). Preto je čierna výška aspoň $h/2$, teda $n \geq 2^{h/2}-1$, čo ďalej dáva $h \leq 2\lg(n+1)$. □

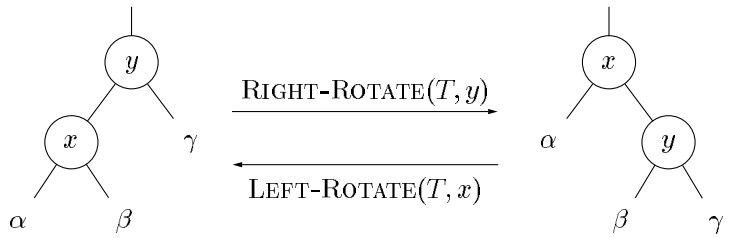


Obr. 26: Červeno-čierny strom, ktorého čierne uzly sú označené obyčajnými kružnicami a červené uzly sú označené dvojitými kružnicami. Každý uzol červeno-čierneho stromu je buď červený alebo čierny, každý list NIL je čierny, synovia červeného uzla sú čierni a všetky jednoduché cesty od daného uzla ku listu obsahujú rovnaký počet čiernych uzlov. Každý vnútorný uzol je označený svojou čiernou výškou. Listy majú čiernu výšku 0.

Priamym dôsledkom tejto lemy je, že operácie SEARCH, MINIMUM, MAXIMUM, SUCCESSOR a PREDECESSOR možno implementovať pomocou RB-stromov tak, že čas ich behu je $O(\lg n)$, kde n je počet uzlov stromu, čo vyplýva z toho, že RB-strom pozostávajúci z n uzlov je prehľadávací strom výšky $O(\lg n)$. Hoci algoritmy TREE-INSERT a TREE-DELETE bežia v čase $O(\lg n)$ pre daný RB-strom, nezaručujú, že modifikovaný binárny prehľadávací strom bude opäť červeno-čierny.

4.4.2 Rotácia

Operácie TREE-INSERT a TREE-DELETE bežia na RB-strome o n uzloch v čase $O(\lg n)$. Keďže ale strom modifikujú, výsledok nemusí spĺňať vlastnosti RB-stromu. Aby sme tieto vlastnosti obnovili, musíme zmeniť farby niektorých uzlov a taktiež upraviť štruktúru ukazovateľov.



Obr. 27: Rotujúce operácie na binárnom prehľadávacom strome. Operácia RIGHT-ROTATE(T, x) transformuje konfiguráciu dvoch uzlov naľavo do konfigurácie napravo zmenou konštantného počtu ukazovateľov. Konfiguráciu napravo možno transformovať do konfigurácie naľavo inverznou operáciou LEFT-ROTATE(T, y). Oba uzly sa môžu nachádzať kdekoľvek v strome. Symboly α, β a γ reprezentujú ľubovoľné podstromy. Rotujúce operácie zachovávajú inorder usporiadanie kľúčov.

Štruktúru ukazovateľov meníme pomocou operácie ***rotácia***, ktorá je logickou operáciou na binárnom prehľadávacom strome zachovávajúcou inorder usporiadanie kľúčov. Obr. 27 znázorňuje

dva druhy rotácie: rotáciu vľavo a rotáciu vpravo. Ak na uzle x vykonávame rotáciu vľavo, predpokladáme, že jeho pravý syn nie je NIL. Táto operácia „otáča“ strom okolo spojenia medzi x a y. Uzol y sa stáva novým koreňom podstromu a uzol x sa stáva jeho ľavým synom, pričom pôvodný ľavý syn y je teraz pravým synom x.

```
LEFT-ROTATE(T, x)
1  y ← right[x]            ▷ Nastav y.
2  right[x] ← left[y]      ▷ Ľavý podstrom y je teraz pravým podstromom x.
3  if left[y] ≠ NIL then
4     p[left[y]] ← x
5  p[y] ← p[x]             ▷ Otcom x bude teraz otec y.
6  if p[x] = NIL
7     then root[T] ← y
8     else if x = left[p[x]]
9             then left[p[x]] ← y
10            else right[p[x]] ← y
11 left[y] ← x             ▷ Uzol x je teraz ľavým synom y.
12 p[x] ← y
```

Kód pre RIGHT-ROTATE je podobný. Obe procedúry LEFT-ROTATE a RIGHT-ROTATE bežia v čase $O(1)$. Počas rotácie sa menia iba ukazovatele, všetky ostatné položky uzlov zostávajú nezmenené.

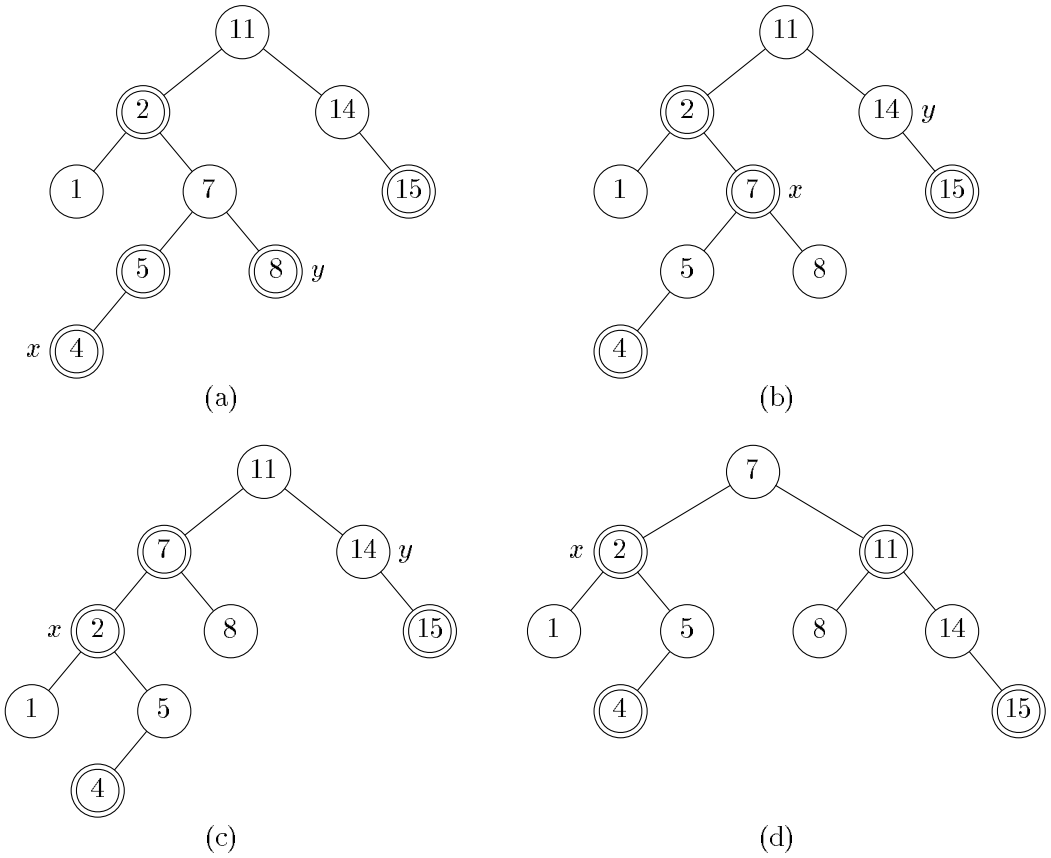
4.4.3 Vkladanie

Vloženie uzla do n-uzlového RB-stromu možno uskutočniť v čase $O(\lg n)$. Na vloženie uzla x do stromu T použijeme procedúru TREE-INSERT, akoby to bol obyčajný binárny prehľadávací strom a následne x označíme červenou farbou. Aby sme zaručili zachovanie vlastností RB-stromu, uzly prefarbíme a na strome vykonáme rotácie. Väčšina kódu RB-INSERT ošetruje rôzne prípady, ktoré nastávajú pri úpravách modifikovaného stromu.

```
RB-INSERT(T, x)
1  TREE-INSERT(T, x)
2  color[x] ← RED
3  while (x ≠ root[T]) and (color[p[x]] = RED) do
4     if p[x] = left[p[p[x]]]
5        then y ← right[p[p[x]]]
6             if color[y] = RED
7                then color[p[x]] ← BLACK              ▷ prípad 1
8                     color[y] ← BLACK                 ▷ prípad 1
9                     color[p[p[x]]] ← RED             ▷ prípad 1
10                    x ← p[p[x]]                      ▷ prípad 1
11               else if x = right[p[x]] then
12                       x ← p[x]                      ▷ prípad 2
13                       LEFT-ROTATE(T, x)             ▷ prípad 2
14                    color[p[x]] ← BLACK              ▷ prípad 3
15                    color[p[p[x]]] ← RED             ▷ prípad 3
16                    RIGHT-ROTATE(T, p[p[x]])         ▷ prípad 3
17       else ▷ rovnako ako vetva then ibaže
18            ▷ treba vymeniť right s left
19 color[root[T]] ← BLACK
```

Kód procedúry RB-INSERT nie je až taký zložitý ako sa môže na prvý pohľad zdať. Analýzu tohto kódu rozdelíme na tri hlavné kroky. Najprv určíme, ktoré vlastnosti RB-stromu možno

narušiť na riadkoch 4.4.3–2 keď je do stromu vkladaný a zafarbovaný na červeno prvok x. Ďalej sa budeme zaoberať úlohou cyklu `while` na riadkoch 3–18, a nakoniec rozoberieme všetky tri prípady, ktoré počas vykonávania tohto cyklu môžu nastať. Obr. 28 znázorňuje činnosť RB-INSERT.



Obr. 28: Činnosť RB-INSERT. (**a**) Strom po vložení uzla x. Keďže x aj jeho otec $p[x]$ sú červení, nastáva porušenie vlastnosti 3. Nakoľko strýko y uzla x je červený, možno aplikovať prípad č. 1. V (**b**) sú už uzly prefarbené a ukazovateľ x je posunutý smerom nahor. Opäť sú x aj jeho otec červení, ale strýko y je teraz čierny. Keďže x je pravým synom $p[x]$, možno aplikovať prípad č. 2. V (**c**) je znázornený strom už po vykonaní rotácie. Teraz je x ľavým synom svojho otca a možno aplikovať prípad č. 3. Rotáciou vpravo dostávame strom (**d**), ktorý už je konečne červeno-čierny.

Ktoré z vlastností RB-stromov môžu byť narušené po vykonaní riadkov 4.4.3–2? Vlastnosť 1 určite zostáva platiť i naďalej. Podobne platí aj vlastnosť 2, keďže vložený červený uzol má ako synov hodnoty NIL, ktoré sú čierne. Vlastnosť 4, ktorá hovorí, že počet čiernych uzlov je rovnaký na každej jednoduchej ceste od daného uzla k listom, je taktiež splnená, lebo červený uzol x s čiernymi potomkami nahrádza čierny NIL. Teda jedinou vlastnosťou, ktorú možno porušiť, je vlastnosť 3 (červený uzol nemôže mať červených synov). Táto vlastnosť je porušená práve vtedy, ak otec uzla x je červený (viď obr. 28a).

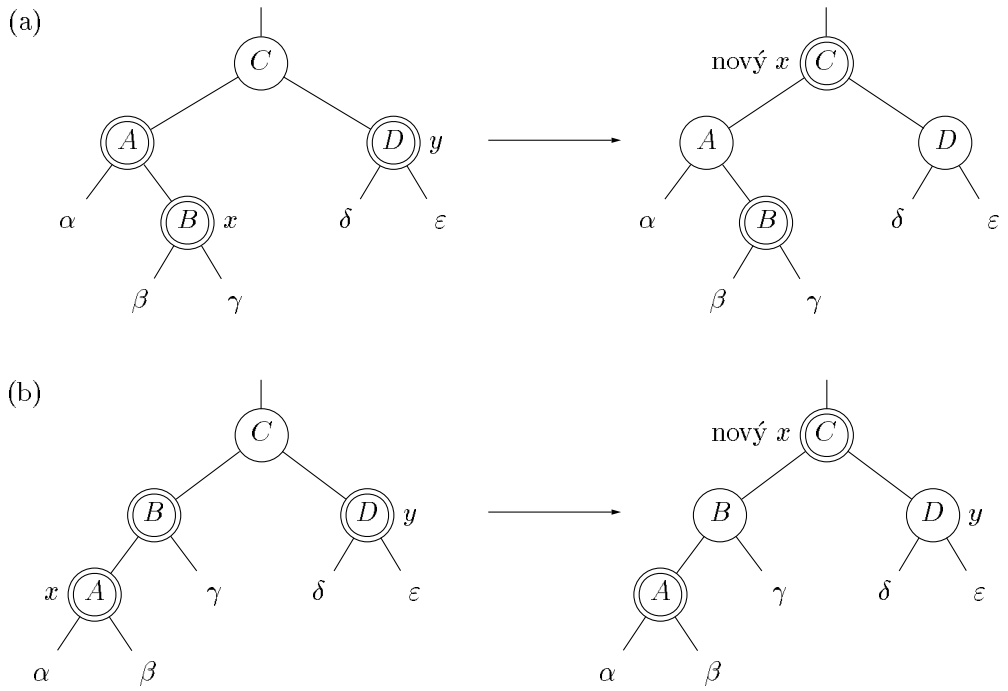
Úlohou cyklu `while` na riadkoch 3–18 je presunúť porušenie vlastnosti 3 v strome smerom nahor za súčasného zachovania vlastnosti 4. Na začiatku každej iterácie cyklu ukazuje x na červený uzol s červeným otcom, čo je jediným porušením vlastností RB-stromu. Túto situáciu možno riešiť dvoma spôsobmi buď posunúť ukazovateľ x smerom nahor alebo vykonaním nejakých rotácií a ukončením cyklu.

V skutočnosti je 6 prípadov, ktoré treba v cykle `while` uvažovať, ale tri z nich sú symetrické ku zvyšným trom, v závislosti od toho, či je otec $p[x]$ uzla x ľavým alebo pravým synom svojho otca $p[p[x]]$. Uviedli sme kód iba pre ten prípad, v ktorom je $p[x]$ ľavým synom. Ďalej sme zaviedli dôležitý predpoklad, že koreň stromu je čierny (viď riadok 19), čo nám teraz umožňuje

predpokladať, že uzol $p[x]$ nie je koreň, a teda, že $p[p[x]]$ vždy existuje.

Prípad 1 možno odlíšiť od prípadov 2 a 3 podľa farby strýka uzla x. Na riadku 5 je y nastavený na strýka $right[p[p[x]]]$ uzla x, pričom test jeho farby je uskutočnený na riadku 6. Ak je y červený, potom je vykonaný prípad 1. Inak je riadenie predané prípadom 2 a 3. Keďže je $p[x]$ červený, vo všetkých troch prípadoch je $p[p[x]]$ čierny, a teda vlastnosť 3 je porušovaná iba medzi uzlami x a $p[x]$.

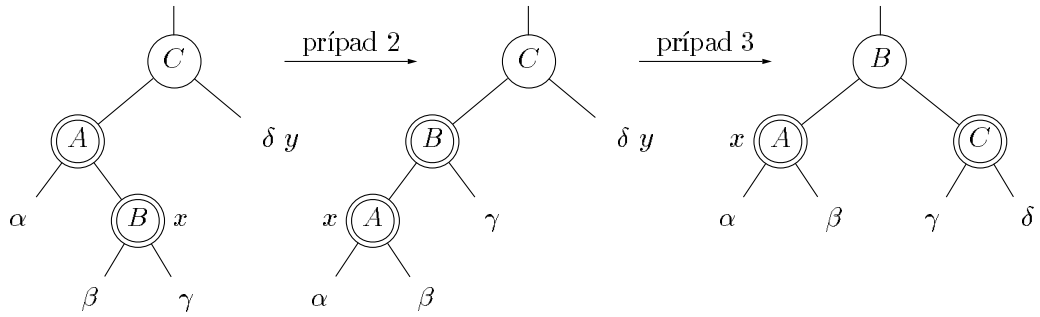
Situácia v prípade 1 (riadky 7–10) je znázornená na obr. 29. Obsluha prvého prípadu je vykonaná ak obidva uzly $p[x]$ a y sú červené. Keďže $p[p[x]]$ je čierny, môžeme ofarbiť obidva tieto uzly na čierno, čím odstránime problém, že x a $p[x]$ sú oba zároveň červené, a uzol $p[p[x]]$ na červeno, aby sme zachovali vlastnosť 4. Jediný problém, ktorý môže nastať, je, že $p[p[x]]$ môže mať červeného otca. Vtedy musíme opakovať cyklus `while` s $p[p[x]]$ ako novým uzlom x.



Obr. 29: Prípad 1 procedúry RB-Insert. Vlastnosť 3 je porušená lebo x a jeho otec $p[x]$ sú obidvaja červení. Či už je x (**a**) pravý syn alebo (**b**) ľavý syn, je vykonaná rovnaká akcia. Každý z podstromov α, β, γ, δ a ε má čierny koreň a rovnakú čiernu výšku. Kód pre prípad 1 prefarbuje niektoré uzly, pričom zachováva vlastnosť 4. Cyklus `while` pokračuje so starým otcom $p[p[x]]$ uzla x ako s novým x. Porušenie vlastnosti 3 môže teraz nastať iba medzi novým uzlom x, ktorý je červený, a jeho otcom, ktorý je tiež červený.

V prípadoch 2 a 3 je strýko y uzla x čierny. Tieto prípady sú rozlíšené podľa toho, či je x ľavým alebo pravým synom $p[x]$. Riadky 12–13 ošetrujú prípad 2, ako je možné vidieť na obr. 30, kde je uvedený aj príklad ošetrenia prípadu 3. V prípade 2 je uzol x ľavým synom svojho otca. Okamžite je použitá rotácia vľavo, aby sme dostali prípad 3 (riadky 14–16), v ktorom je uzol x ľavým synom. Pretože sú oba uzly x a $p[x]$ červené, rotácia neovplyvní ani čierne výšky uzlov ani vlastnosť 4. Či nastane prípad 3 priamo alebo rotáciou, cez prípad 2, je strýko y uzla x čierny, lebo inak by sme ošetrili prípad 1. Niektoré uzly prefarbíme a vykonáme rotáciu vpravo, a potom, keďže už nemáme dva susedné červené uzly, činnosť je ukončená. Keďže uzol $p[x]$ je teraz čierny, telo cyklu `while` už nie je treba znova vykonávať.

Aký je čas behu procedúry RB-Insert? Pretože výška RB-stromu o n uzloch je $O(\lg n)$, vyvolanie Tree-Insert trvá čas $O(\lg n)$. Telo cyklu `while` sa opakuje iba v prípade 1, pričom



Obr. 30: Prípady 2 a 3 procedúry RB-INSERT. Podobne ako v prípade 1, je vlastnosť 3 porušená v oboch prípadoch 2 a 3 pretože x a jeho otec $p[x]$ sú obidvaja červení. Každý z podstromov α, β, γ a δ má čierny koreň a rovnakú čiernu výšku. Prípad 2 je prevedený na prípad 3 rotáciou vľavo, ktorá zachováva vlastnosť 4. Riešenie prípadu 3 spôsobí prefarbenie niektorých uzlov a rotáciu stromu vpravo. Cyklus `while` potom skončí, lebo je splnená podmienka 3.

sa ukazovateľ x pohybuje v strome smerom nahor. Celkový počet vykonaní cyklu `while` je teda nanajvýš $O(\lg n)$. Teda vykonávanie RB-INSERT trvá celkovo čas $O(\lg n)$. Zaujímavé je, že sa nikdy nevykonajú viac ako dve rotácie (cyklus je ukončený hneď po ošetrení prípadu 2 alebo 3).

4.4.4 Vymazávanie

Podobne ako ostatné základné operácie na n-uzlovom RB-strome, aj vymazávanie uzla trvá čas $O(\lg n)$.

Pre zjednodušenie hraničných podmienok v kóde pre strom T, na reprezentáciu NIL-ov použijeme sentinel $nil[T]$, ktorý je objektom s rovnakými položkami ako obyčajný uzol stromu, ale s tým rozdielom, že jeho farba je stále čierna. Namiesto každého NIL-u by sme mohli pridať jeden sentinel, ale príliš by sme tak plýtvali pamäťou. Namiesto toho použijeme jeden sentinel $nil[T]$ na reprezentáciu všetkých NIL-ov. Preto, ak manipulujeme so synom uzla x, musíme najprv nastaviť $p[nil[T]]$ na x.

```
RB-DELETE(T, z)
1  if (left[z] = nil[T]) or (right[z] = nil[T])
2    then y ← z
3    else y ← TREE-SUCCESSOR(z)
4  if left[y] ≠ nil[T]
5    then x ← left[y]
6    else x ← right[y]
7  p[x] ← p[y]
8  if p[y] = nil[T]
9    then root[T] ← x
10   else if y = left[p[y]]
11           then left[p[y]] ← x
12           else right[p[y]] ← x
13 if y ≠ z then
14   key[z] ← key[y]
15   ▷ Ak má y ďalšie položky, treba ich tiež skopírovať.
16 if color[y] = BLACK then
17   RB-DELETE-FIXUP(T, x)
18 return y
```

Procedúra RB-DELETE je mierne modifikovanou procedúrou TREE-DELETE. Po odstránení uzla zavolá pomocnú procedúru RB-DELETE-FIXUP, ktorá prefarbí niektoré uzly a vykoná rotácie, aby obnovila vlastnosti RB-stromu. Medzi procedúrami TREE-DELETE a RB-DELETE sú tri rozdiely.

1. Všetky výskyty NIL v TREE-DELETE boli v RB-DELETE nahradené odkazmi na sentinel $nil[T]$.
2. Test na riadku 7 TREE-DELETE, či má x hodnotu NIL, bol odstránený a priradenie $p[x] \leftarrow p[y]$ je tak vykonávané vždy (riadok 7 RB-DELETE). Teda, ak x je $nil[T]$, jeho ukazovateľ na rodiča ukazuje na rodiča odstráneného uzla y.
3. Ak má y čiernu farbu, je na riadkoch 16–17 vyvolaná procedúra RB-DELETE-FIXUP. Ak je y červený, červeno-čierne vlastnosti nie sú jeho ostránením narušené.

Uzol x predaný ako parameter procedúre RB-DELETE-FIXUP je buď jediným synom y ešte pred vymazaním uzla y, alebo je to sentinel $nil[T]$, ak y nemal potomkov. V druhom prípade priradenie na riadku 7 zaručuje, že otec uzla x je teraz uzlom, ktorý bol predtým otcom y, či už x je vnútorný uzol alebo sentinel $nil[T]$.

Teraz sa môžeme zaoberať tým, ako procedúra RB-DELETE-FIXUP obnovuje červeno-čierne vlastnosti prehľadávacieho stromu.

RB-DELETE-FIXUP(T, x)

```
1  while (x ≠ root[T]) and (color[x] = BLACK) do
2    if x = left[p[x]]
3      then w ← right[p[x]]
4           if color[w] = RED then
5             color[w] ← BLACK                          ▷ prípad 1
6             color[p[x]] ← RED                         ▷ prípad 1
7             LEFT-ROTATE(T, p[x])                      ▷ prípad 1
8             w ← right[p[x]]                           ▷ prípad 1
9           if (color[left[w]] = BLACK) and (color[right[w]] = BLACK)
10            then color[w] ← RED                       ▷ prípad 2
11                 x ← p[x]                             ▷ prípad 2
12            else if color[right[w]] = BLACK then
13                   color[left[w]] ← BLACK             ▷ prípad 3
14                   color[w] ← RED                     ▷ prípad 3
15                   RIGHT-ROTATE(T, w)                 ▷ prípad 3
16                   w ← right[p[x]]                    ▷ prípad 3
17                 color[w] ← color[p[x]]               ▷ prípad 4
18                 color[p[x]] ← BLACK                  ▷ prípad 4
19                 color[right[w]] ← BLACK              ▷ prípad 4
20                 LEFT-ROTATE(T, p[x])                 ▷ prípad 4
21                 x ← root[T]                          ▷ prípad 4
22     else ▷ rovnako ako vetva then, ibaže
23          ▷ treba navzájom zameniť right a left.
24 color[x] ← BLACK
```

Ak uzol y odstránený procedúrou RB-DELETE bol čierny, jeho odstránenie spôsobilo, že každá cesta, ktorá ho predtým obsahovala má teraz o jeden čierny uzol menej. To znamená, že vlastnosť 4 je teraz porušovaná každým predkom y. Tento problém môžeme odstrániť tým, že budeme predpokladať, že uzol x má o jednu čiernu farbu „viac“. Teda, keď odstraňujeme uzol y, jeho čiernosť presúvame na jeho syna. Jediným problémom môže byť, že syn môže byť takto dvojnásobne čierny, čím sa poruší vlastnosť 1.

Procedúra RB-DELETE-FIXUP sa snaží obnoviť vlastnosť 1. Úlohou cyklu `while` na riadkoch 1–23 je presúvať extra čiernu farbu v strome smerom nahor, až pokým nenastane jedna z nasledujúcich podmienok:

1. x ukazuje na červený uzol, ktorý môžeme prefarbiť na čierno,
2. x ukazuje na koreň stromu, kde čierna farba môže byť pohltená,
3. možno vykonať vhodné rotácie alebo prefarbenia uzlov.

Vnútri cyklu `while` x ukazuje vždy na čierny uzol, ktorý má o jednu čiernu farbu viac (tento uzol nie je nikdy koreňom). Na riadku 2 určujeme, či je ľavým alebo pravým synom svojho otca $p[x]$. Ukazovateľ na súrodenca uzla x udržiavame v premennej w. Keďže uzol x je dvojnásobne čierny, je zrejmé, že w nemôže byť $nil[T]$.

Štyri prípady, ktoré kód ošetruje, sú znázornené na obr. 31. Pred tým ako sa nimi budeme hlbšie zaoberať ukážme, že transformácia v každom z prípadov zachováva vlastnosť 4. Hlavnou myšlienkou je, že vo všetkých prípadoch je počet čiernych uzlov od koreňa (vrátane neho) znázorneného podstromu ku každému z podstromov $\alpha, \beta, \ldots, \zeta$ transformáciou zachovávaný. Napríklad na obr. 31a je počet čiernych uzlov od koreňa ku podstromom α alebo β rovný 3 ako pred, tak i po vykonaní transformácie. Podobne, počet čiernych uzlov od koreňa ku ktorémukoľvek zo stromov γ, δ, ε a ζ je 2 pred aj po vykonaní transformácie. Na obr. 31b musíme zarátať aj farbu c, ktorá môže byť buď červená alebo čierna. Ak zadefinujeme black(RED) $= 0$ a black(BLACK) $= 1$, potom počet čiernych uzlov od koreňa ku α bude $2 +$ black(c) pred aj po vykonaní transformácie. Podobne možno overiť aj ďalšie prípady.

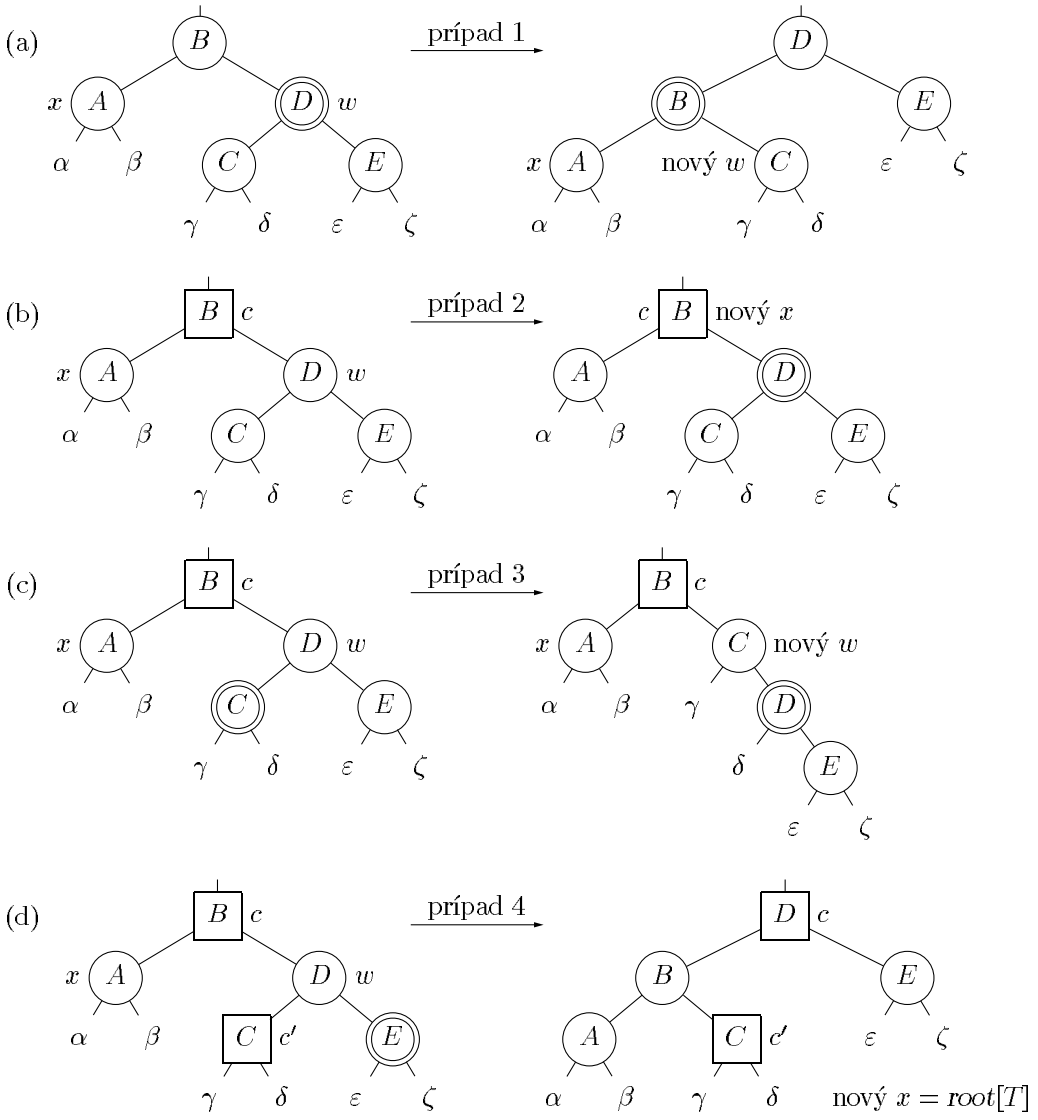
Prípad 1 (riadky 5–8 a obr. 31a) nastáva keď uzol w, t.j. súrodenec x, je červený. Keďže w musí mať čiernych synov, môžeme vymeniť farby w a $p[x]$ a následne na $p[x]$ vykonať rotáciu vľavo bez porušenia ktorejkoľvek z červeno-čiernych vlastností. Nový súrodenec x (jeden zo synov uzla w) je teraz čierny, a teda sme prípad 1 previedli na jeden z prípadov 2, 3 alebo 4.

Prípad 2 (riadky 10–11 a obr. 31b). Prípady 2, 3 a 4 nastávajú ak uzol w je čierny. Rozlíšiť ich možno podľa farieb jeho synov. V prípade 2 sú obaja synovia uzla w čierni. Keďže je w tiež čierny, vezmeme obom uzlom x a w po jednej čiernej farbe, čím x zostane čierny a w červený, pričom túto nadbytočnú čiernu farbu pridáme uzlu $p[x]$. Potom opakujeme cyklus `while` s uzlom $p[x]$ ako novým x. Všimnime si teraz, že ak prípad 2 nastane tesne po prípade 1, farba c nového uzla x je červená, keďže $p[x]$ bol červený, a teda sa vykonávanie cyklu skončí.

Prípad 3 (riadky 13–16 a obr. 31c) nastáva ak uzol w je čierny, jeho ľavý syn je červený a jeho pravý syn je čierny. Vtedy môžeme vymeniť farby uzla w a jeho ľavého syna $left[w]$, a následne okolo w vykonať rotáciu vpravo bez toho, aby sme porušili ktorúkoľvek z červeno-čiernych vlastností. Nový súrodenec w uzla x je teraz čiernym uzlom s červeným pravým synom, teda sme prípad 3 previedli na prípad 4.

Prípad 4 (riadky 17–21 a obr. 31d) nastáva ak súrodenec w uzla x je čierny a jeho pravý syn je červený. Prefarbením niektorých uzlov a vykonaním rotácie vľavo okolo uzla $p[x]$ môžeme z uzla x odstrániť nadbytočnú čiernu farbu bez porušenia ktorejkoľvek z červeno-čiernych vlastností. Ak je uzol x koreňom stromu, vykonávanie cyklu `while` sa okamžite po overení podmienky testu skončí.

Aký je čas behu procedúry RB-DELETE? Pretože výška RB-stromu o n uzloch je $O(\lg n)$, sú aj časové nároky tejto procedúry bez vyvolania RB-DELETE-FIXUP rovné $O(\lg n)$. Ak vnútri procedúry RB-DELETE-FIXUP nastane jeden z prípadov 1, 3 alebo 4, vykoná sa konečný počet prefarbení a jej vykonávanie sa skončí. Prípad 2 je jediný, v ktorom je možné opakovanie cyklu `while`, pričom sa ukazovateľ x pohybuje v strome smerom nahor. Jeho pohyb je však obmedzený



Obr. 31: Prípady v cykle `while` procedúry RB-DELETE. Čierne uzly sú označené obyčajnými kružnicami, červené dvojitými kružnicami a uzly, ktoré môžu byť ako červené, tak aj čierne sú označené štvorcami a znakmi c a c'. Symboly $\alpha, \beta, \ldots, \zeta$ reprezentujú ľubovoľné podstromy. Vo všetkých prípadoch je konfigurácia na ľavej strane prevedená na konfiguráciu vpravo prefarbením uzlov alebo vykonaním rotácie. Uzol, na ktorý ukazuje x, má naviac jednu čiernu farbu. Jediným prípadom, po ktorom sa pokračuje vo vykonávaní cyklu, je prípad 2. (**a**) Prípad 1 je prevedený na prípad 2, 3 alebo 4 výmenou farieb uzlov B a D a vykonaním rotácie vľavo. (**b**) V prípade 2, je nadbytočná čierna farba reprezentovaná ukazovateľom x posunutá smerom nahor ofarbením uzla D na červeno a nastavením x na uzol B. Ak po prípade 1 nastane prípad 2, cyklus sa ukončí, lebo farba c je červená. (**c**) Prípad 3 je prevedený na prípad 4 výmenou farieb uzlov C a D a vykonaním rotácie vpravo. (**d**) V prípade 4, možno nadbytočnú farbu reprezentovanú x odstrániť prefarbením niektorých uzlov a vykonaním rotácie vľavo (bez porušenia červeno-čiernych vlastností), pričom cyklus končí.

výškou RB-stromu, ktorá je rovná $O(\lg n)$. To však znamená, že vykonávanie procedúry RB-DELETE-FIXUP trvá čas $O(\lg n)$, pričom sa uskutočnia nanajvýš 3 rotácie. Preto je celkový čas behu procedúry RB-DELETE taktiež rovný $O(\lg n)$.

5 Pokročilé techniky návrhu a analýzy algoritmov

V tejto časti sa budeme zaoberať dvoma dôležitými technikami pre návrh a analýzu efektívnych algoritmov: dynamickým programovaním a greedy algoritmami.

5.1 Dynamické programovanie

Dynamické programovanie, podobne ako metóda divide & conquer, rieši problémy spájaním riešení podproblémov [13]. Divide & conquer algoritmus rozdeľuje problém na nezávislé podproblémy, rekurzívne tieto podproblémy vyrieši, a následne ich riešenia skombinuje, aby vyriešil pôvodný problém. Naopak, dynamické programovanie je použiteľné vtedy, ak podproblémy nie sú navzájom nezávislé, t.j. keď problémy zdieľajú podproblémy. Z tohto hľadiska vykonáva divide & conquer algoritmus viacej práce ako je treba – opakovane rieši spoločné podproblémy. Algoritmus využívajúci techniku dynamického programovania rieši každý podproblém iba raz, pričom jeho výsledok ukladá do tabuľky, aby predišiel jeho opätovnému, už zbytočnému, riešeniu.

Dynamické programovanie sa zvyčajne používa na riešenie ***optimalizačných problémov.*** Optimalizačné problémy môžu mať viacero riešení, pričom každému z riešení možno nejakým spôsobom priradiť hodnotu. Naším želaním je nájsť aspoň jedno riešenie s optimálnou (minimálnou alebo maximálnou) hodnotou. Takéto riešenie nazývame optimálne.
Návrh DP-algoritmu možno rozdeliť do štyroch krokov.

1. Charakterizuj štruktúru optimálneho riešenia.
2. Rekurzívne definuj hodnotu optimálneho riešenia.
3. Metódou bottom-up vypočítaj ohodnotenie optimálneho riešenia.
4. Z vypočítanej informácie skonštruuj optimálne riešenie.

Kroky 1–3 tvoria základ riešenia problému pomocou dynamického programovania. Ak nám stačí poznať iba ohodnotenie optimálneho riešenia, možene krok 4 vypustiť. Ak však požadujeme aj samotné optimálne riešenie, musíme krok 4 vykonať, pričom je niekedy vhodné si už počas výpočtu v kroku 3 niekam ukladať prídavné informácie, aby sa zjednodušila konštrukcia optimálneho riešenia.

5.1.1 Násobenie matíc

Typickým príkladom pre využitie techník dynamického programovania je násobenie matíc. Daná je postupnosť $\langle A_1, A_2, \ldots, A_n \rangle$ n matíc, pričom treba vypočítať súčin $A_1 A_2 \cdots A_n$. Tento výraz možno vyhodnotiť použitím štandardného algoritmu pre násobenie dvoch matíc hneď po jeho uzátvorkovaní, ktorým sú odstránené všetky nejednoznačnosti vzájomného násobenia matíc. Súčin matíc je ***úplne uzátvorkovaný***, ak pozostáva iba z jednej matice alebo uzátvorkovaného súčinu dvoch plne uzátvorkovaných súčinov matíc.

Násobenie matíc je asociatívne, t.j. všetky možné uzátvorkovania matíc vedú k tej istej hodnote súčinu. Napríklad súčin $A_1 A_2 A_3 A_4$ možno úplne uzátvorkovať piatimi rôznymi spôsobmi:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

Spôsob akým matice uzátvorkujeme výrazne ovplyvňuje časové nároky na vyhodnotenie súčinu. Najprv uvažujme násobenie dvoch matíc. Nasledujúci pseudokód implementuje štandardný algoritmus násobenia dvoch matíc. Atribút matice *rows*, resp. *columns* je počet jej riadkov, resp. stĺpcov.

[13] Programovanie sa v tomto kontexte nevzťahuje na písanie kódu, ale na tabuľkovú metódu.

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2    then error "nekompatibilné rozmery matíc"
3    else for i ← 1 to rows[A] do
4           for j ← 1 to columns[B] do
5             C[i, j] ← 0
6             for k ← 1 to columns[A] do
7               C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8         return C
```

Dve matice A a B môžeme vynásobiť iba ak je počet stĺpcov A rovný počtu riadkov B. Ak A je matica typu $p \times q$ a B typu $q \times r$, výsledná matica je typu $p \times r$. Čas potrebný na výpočet C je určený najmä počtom skalárnych násobení na riadku 7, ktorý je pqr. V nasledujúcom budeme časy behu vyjadrovať na základe tohto počtu.

Aby sme ilustrovali rozličné časové nároky spôsobené rôznym uzátvorkovaním súčinu matíc, uvažujme matice $\langle A_1, A_2, A_3 \rangle$. Nech rozmery týchto matíc sú postupne 10×100, 100×5 a 5×50. Ak budeme počítať $((A_1 A_2)A_3)$, vykonáme $10 \cdot 100 \cdot 5 = 5000$ skalárnych násobení na výpočet súčinu $A_1 A_2$ plus ďalších $10 \cdot 5 \cdot 50 = 2500$ skalárnych násobení aby sme túto maticu rozmeru 10×5 vynásobili maticou A_3, teda celkovo vykonáme 7500 skalárnych násobení. Ak by sme namiesto $((A_1 A_2)A_3)$ počítali $(A_1(A_2 A_3))$, museli by sme vykonať $100 \cdot 5 \cdot 50 = 25000$ skalárnych násobení na výpočet súčinu $A_2 A_3$ plus ďalších $10 \cdot 100 \cdot 50 = 50000$ skalárnych násobení pre vynásobenie A_1 touto maticou, teda celkovo by sme vykonali 75000 skalárnych násobení, čo je 10-krát viac ako pri uzátvorkovaní v prvom prípade.

Problém násobenia postupnosti matíc možno formulovať nasledovne: pre danú postupnosť $\langle A_1, A_2, \ldots, A_n \rangle$ n matíc, kde pre $i = 1, 2, \ldots, n$ má matica A_i rozmery $p_{i-1} \times p_i$, úplne uzátvorkuj súčin $A_1 A_2 \cdots A_n$ tak, aby počet potrebných skalárnych násobení bol minimálny.

Výpočet počtu uzátvorkovaní

Predtým, ako začneme problém riešiť pomocou dynamického programovania, by sme sa mali presvedčiť, že preverovanie všetkých možných uzátvorkovaní nevedie ku efektívnemu algoritmu. Označme $P(n)$ počet možných uzátvorkovaní postupnosti n matíc. Keďže postupnosť n matíc môžeme pre ľubovoľné $k = 1, 2, \ldots, n-1$ rozdeliť medzi k-tou a $(k+1)$-vou maticou a následne obe výsledné podpostupnosti nezávisle uzátvorkovať, dostávame rekurenciu

$$P(n) = \begin{cases} 1 & \text{ak } n = 1\,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{ak } n \geq 2\,. \end{cases}$$

Riešením tohto rekurentného vzťahu je postupnosť ***Catalanových čísel***:

$$P(n) = C(n-1)\,,$$

kde

$$\begin{aligned} C(n) &= \frac{1}{n+1}\binom{2n}{n} \\ &= \Omega(4^n/n^{3/2})\,. \end{aligned}$$

Počet možných uzátvorkovaní teda od n závisí exponenciálne, a preto metóda, pri ktorej overíme všetky možné spôsoby uzátvorkovania, je pre určovanie optimálneho uzátvorkovania nevhodná.

Štruktúra optimálneho uzátvorkovania

Prvým krokom paradigmy dynamického programovania je charakterizovanie štruktúry optimálneho riešenia. Pre násobenie postupnosti matíc môžeme tento krok uskutočniť nasledovným

spôsobom. Pre maticu, ktorá vznikne vyhodnotením súčinu $A_iA_{i+1}\cdots A_j$, zavedieme označenie $A_{i\,..\,j}$. Pri optimálnom uzátvorkovaní rozdelíme súčin $A_1A_2\cdots A_n$ na k-tej pozícii, kde k je nejaké celé číslo z rozsahu $1 \leq k < n$, na dve postupnosti matíc $A_1A_2\cdots A_k$ a $A_{k+1}A_{k+2}\cdots A_n$. To znamená, že na výpočet matice $A_{1\,..\,n}$ musíme najprv vyhodnotiť matice $A_{1\,..\,k}$ a $A_{k+1\,..\,n}$, ktoré následne navzájom vynásobíme. Náklady na optimálne uzátvorkovanie teda pozostávajú z nákladov na výpočet matíc $A_{1\,..\,k}$ a $A_{k+1\,..\,n}$ a z nákladov na ich vzájomné vynásobenie.

Všimnime si, že uzátvorkovania podpostupností $A_1A_2\cdots A_k$ a $A_{k+1}A_{k+2}\cdots A_n$ vnútri optimálneho uzátvorkovania pre $A_1A_2\cdots A_n$ musia byť taktiež *optimálne*. Teda optimálne riešenie problému násobenia matíc v sebe obsahuje optimálne riešenia podproblémov. Optimálna podštruktúra optimálneho riešenia je jednou z podmienok použitia dynamického programovania.

Rekurzívne riešenie

Druhým krokom paradigmy dynamického programovania je rekurzívne definovanie ohodnotenia optimálneho riešenia na základe optimálnych riešení podproblémov. Pre problém násobenia matíc si ako podproblémy vyberieme problémy stanovenia minimálneho počtu operácií potrebných na výpočet $A_iA_{i+1}\cdots A_j$, pričom $1 \leq i \leq j \leq n$. Nech $m[i,j]$ je minimálny počet skalárnych násobení potrebných pre výpočet matice $A_{i\,..\,j}$ – $m[1,n]$ teda označuje najmenší počet násobení, ktoré treba vykonať, aby sme vypočítali $A_{1\,..\,n}$.

Hodnoty $m[i,j]$ môžeme definovať rekurzívne nasledovným spôsobom. Ak $i = j$, postupnosť pozostáva iba z jednej matice $A_{i\,..\,i} = A_i$, teda na výpočet súčinu netreba vykonať žiadne operácie skalárneho násobenia. Máme teda $m[i,i] = 0$ pre $i = 1, 2, \ldots, n$. Aby sme mohli vypočítať $m[i,j]$ pre $i < j$, využijeme štruktúru optimálneho riešenia z kroku 1. Predpokladajme, že optimálne uzátvorkovanie rozdelí súčin $A_iA_{i+1}\cdots A_j$ na k-tej pozícii ($i \leq k < j$), t.j. medzi maticami A_k a A_{k+1}. Potom je $m[i,j]$ rovné súčtu minimálnych nákladov na výpočet matíc $A_{i\,..\,k}$ a $A_{k+1\,..\,j}$ a nákladov na vzájomné vynásobenie týchto matíc. Keďže na výpočet súčinu $A_{i\,..\,k}A_{k+1\,..\,j}$ potrebujeme $p_{i-1}p_kp_j$ skalárnych násobení, dostávame rekurenciu

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\ .$$

Tento rekurzívny vzťah však predpokladá, že hodnotu k poznáme pre každé i a j. O k vieme iba toľko, že môže nadobúdať jednu z $j - i$ hodnôt, konkrétne $k = i, i+1, \ldots, j-1$. Teraz môžeme prejsť všetky možnosti – dosadzovaním $i, i+1, \ldots, j-1$ za k do výrazu na pravej strane uvedenej rekurencie. Hľadanou hodnotou k je tá, pre ktorú bude tento výraz minimálny. Teda naša rekurzívna definícia pre minimálne náklady na uzátvorkovanie súčinu $A_iA_{i+1}\cdots A_j$ nadobúda tvar

$$m[i,j] = \begin{cases} 0 & \text{ak } i = j\ , \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{ak } i < j\ . \end{cases} \tag{16}$$

Hodnoty $m[i,j]$ vyjadrujú náklady optimálnych riešení podproblémov. Aby sme mohli vykonštruovať optimálne riešenie, zadefinujme $s[i,j]$ ako pozíciu k, na ktorej môžeme rozdeliť súčin $A_iA_{i+1}\cdots A_j$, aby sme dostali optimálne uzátvorkovanie. Teda $s[i,j]$ sa rovná hodnote k takej, že $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Výpočet optimálnych nákladov

Hoci by sme boli okamžite schopní na základe rekurencie (16) napísať funkčný algoritmus, ktorý počíta minimálne náklady $m[1,n]$ na výpočet $A_1A_2\cdots A_n$, jeho čas behu však bol stále exponenciálny.

Všimnime si teraz, že máme relatívne málo podproblémov: jeden problém pre každý výber hodnôt i a j takých, že $1 \leq i \leq j \leq n$, t.j. celkovo $\binom{n}{2} + n = \Theta(n^2)$. Rekurzívny algoritmus môže na každý podproblém mnohokrát naraziť v rôznych vetvách jeho rekurzívneho stromu. Toto prekrývanie podproblémov je ďalšou z podmienok použitia dynamického programovania.

Namiesto rekurzívneho výpočtu rekurencie (16) vykonáme tretí krok paradigmy dynamického programovania a vypočítame optimálne náklady použitím metódy bottom-up. Nasledujúci pseudokód predpokladá, že matica A_i má rozmery $p_{i-1} \times p_i$ pre $i = 1, 2, \ldots, n$. Vstupom je postupnosť

$\langle p_0, p_1, \ldots, p_n \rangle$, kde $length[p] = n + 1$. Procedúra používa pomocnú tabuľku $m[1..n, 1..n]$ pre uchovávanie nákladov $m[i, j]$ a tabuľku $s[1..n, 1..n]$, zaznamenávajúcu, ktorým indexom k sa dosiahli optimálne náklady na výpočet $m[i, j]$.

MATRIX-CHAIN-ORDER(p)

```
1  n ← length[p] − 1
2  for i ← 1 to n do
3     m[i, i] ← 0
4  for l ← 2 to n do
5     for i ← 1 to n − l + 1 do
6        j ← i + l − 1
7        m[i, j] ← ∞
8        for k ← i to j − 1 do
9           q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10          if q < m[i, j] then
11             m[i, j] ← q
12             s[i, j] ← k
13 return m and s
```

Algoritmus vyplní tabuľku m takým spôsobom, ktorý prislúcha riešeniu uzátvorkovávacieho problému na postupnostiach matíc rastúcej dĺžky. Vzťah (16) ukazuje, že náklady $m[i, j]$ na výpočet súčinu postupnosti $j - i + 1$ matíc závisia iba od nákladov na výpočet súčinov postupností matíc dĺžok menších ako $j - i + 1$. T.j. pre $k = i, i + 1, \ldots, j - 1$ je matica $A_{i..k}$ súčinom $k - i + 1 < j - i + 1$ matíc a matica $A_{k+1..j}$ súčinom $j - k < j - i + 1$ matíc.

Algoritmus najprv spočíta $m[i, i] \leftarrow 0$ pre $i = 1, 2, \ldots, n$ (minimálne náklady pre postupnosti dĺžky 1) na riadkoch 2–3. Potom použije rekurenciu (16) na výpočet $m[i, i+1]$ pre $i = 1, 2, \ldots, n-1$ (minimálne náklady pre postupnosti dĺžok 2) počas prvého vykonania cyklu na riadkoch 4–12. Ďalším priechodom cyklu sa spočíta $m[i, i + 2]$ pre $i = 1, 2, \ldots, n - 2$, atď. V každom kroku, náklady $m[i, j]$ vypočítané na riadkoch 9–12 závisia iba na už vypočítaných položkách tabuľky $m[i, k]$ a $m[k + 1, j]$.

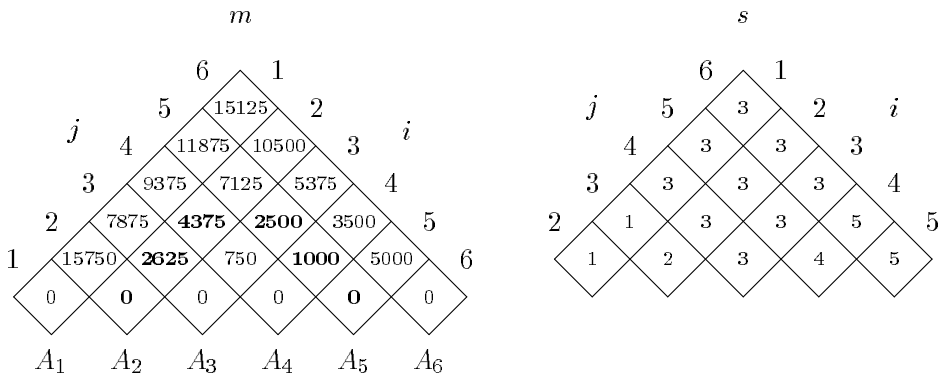
Obr. 32 znázorňuje túto procedúru na postupnosti $n = 6$ matíc. Keďže máme definované $m[i, j]$ iba pre $i \leq j$, je použitá iba časť tabuľky m nad hlavnou diagonálou. Obrázok znázorňuje tabuľku otočenú tak, aby hlavná diagonála bola umiestnená horizontálne. Postupnosť matíc je uvedená dole. Minimálne náklady na vynásobenie podpostupnosti $A_i A_{i+1} \cdots A_j$ matíc možno v tejto tabuľke nájsť na pozícii $[i, j]$. Každý horizontálny riadok v tabuľke obsahuje položky pre postupnosti matíc rovnakej dĺžky. MATRIX-CHAIN-ORDER počíta riadky odspodu nahor a zľava doprava vnútri každého riadku. Položka $m[i, j]$ je vypočítaná použitím súčinov $p_{i-1} p_k p_j$ pre $k = i, i + 1, \ldots, j - 1$ a všetkých položiek juhozápadne a juhovýchodne od $m[i, j]$.

Jednoduchá inšpekcia štruktúry vnorených cyklov v procedúre MATRIX-CHAIN-ORDER dáva čas behu algoritmu $O(n^3)$. Možno ukázať, že tento čas je aj $\Omega(n^3)$. To znamená, že MATRIX-CHAIN-ORDER je omnoho efektívnejšia ako metóda enumerujúca a testujúca všetky možné uzátvorkovania, ktorej čas behu je exponenciálny.

Konštrukcia optimálneho riešenia

Hoci MATRIX-CHAIN-ORDER stanovuje optimálny počet skalárnych násobení potrebných na výpočet súčinu matíc, neukazuje priamo, ako matice vynásobiť. Krokom 4 paradigmy dynamického programovania je skonštruovanie optimálneho riešenia na základe vypočítanej informácie.

V našom prípade používame tabuľku $s[1..n, 1..n]$, aby sme určili najlepší spôsob uzátvorkovania matíc. Každá položka $s[i, j]$ zaznamenáva hodnotu k takú, že optimálne uzátvorkovanie $A_i A_{i+1} \cdots A_j$ rozdelí súčin medzi A_k a A_{k+1}. Teda vieme, že finálne násobenie matíc vo výpočte $A_{1..n}$ optimálne je $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Skoršie násobenia matíc možno rekurzívne vypočítať, nakoľko $s[1, s[1, n]]$ určuje posledné násobenie matíc vo výpočte $A_{1..s[1,n]}$ a $s[s[1, n] + 1, n]$ určuje posledné násobenie matíc vo výpočte $A_{s[1,n]+1,n}$. Nasledujúca rekurzívna procedúra počíta súčin



Obr. 32: Tabuľky m a s vypočítané procedúrou MATRIX-CHAIN-ORDER pre $n = 6$ a rozmery matíc:

matica	rozmery
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Tabuľky sú otočené, aby bola hlavná diagonála umiestnená horizontálne. V tabuľke m je použitá iba hlavná diagonála a prvky ležiace nad ňou, pričom v tabuľke s sú použité len prvky nad hlavnou diagonálou. Minimálny počet skalárnych násobení pre vynásobenie 6 matíc je $m[1,6] = 15125$. Pri výpočte $m[2,5]$ sú použité iba zvýraznené položky.

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

matíc $A_{i..j}$ pre dané matice $A = \langle A_1, A_2, \ldots, A_n \rangle$, tabuľku s vypočítanú pomocou MATRIX-CHAIN-ORDER a indexy i, j. Počiatočné volanie je MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.

MATRIX-CHAIN-MULTIPLY(A, s, i, j)

```
1  if j > i
2    then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
3         Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
4         return MATRIX-MULTIPLY(X, Y)
5    else return A_i
```

Na príklade obrázku 32 volanie MATRIX-CHAIN-MULTIPLY$(A, s, 1, 6)$ vypočíta súčin matíc prislúchajúci uzátvorkovaniu

$$((A_1(A_2A_3))((A_4A_5)A_6))\,.$$

5.1.2 Najdlhšia spoločná podpostupnosť

Nasledujúcim problémom, ktorým sa budeme teraz zaoberať, je hľadanie najdlhšej spoločnej podpostupnosti. Postupnosť $Z = \langle z_1, z_2, \ldots, z_k \rangle$ je ***podpostupnosťou*** postupnosti $X = \langle x_1, x_2, \ldots, x_m \rangle$, ak existuje rýdzo rastúca postupnosť $\langle i_1, i_2, \ldots, i_k \rangle$ indexov X taká, že pre všetky $j = 1, 2, \ldots, k$ platí $x_{i_j} = z_j$. Pre dané dve postupnosti X a Y hovoríme, že postupnosť Z je ***spoločnou podpostupnosťou*** X a Y, ak je podpostupnosťou X aj Y.

Problém ***najdlhšej spoločnej podpostupnosti (LCS)*** možno formulovať nasledovne: sú

dané dve postupnosti $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $\langle y_1, y_2, \ldots, y_n \rangle$, treba nájsť najdlhšiu spoločnú podpostupnosť postupností X a Y.

Charakterizácia najdlhšej spoločnej podpostupnosti

Jedným z možných riešení tohto problému je opäť enumerácia všetkých podpostupností X. Pre každú podpostupnosť X budeme overovať, či je tiež podpostupnosťou Y, pričom si budeme uchovávať najdlhšiu nájdenú podpostupnosť. Každá podpostupnosť X prislúcha nejakej podmnožine indexov $\{1, 2, \ldots, m\}$ postupnosti X. Celkový počet podpostupností X je teda 2^m, z čoho vyplýva, že toto riešenie vyžaduje exponenciálny čas, a teda je nevhodné pre dlhšie postupnosti.

Problém LCS sa vyznačuje vlastnosťou optimálnej podštruktúry (viď nasledujúcu vetu). Pre danú postupnosť $X = \langle x_1, x_2, \ldots, x_m \rangle$ a index $i = 0, 1, \ldots, m$ definujeme i-ty ***prefix*** X ako postupnosť $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

Veta 5.1 *Nech $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$ sú postupnosti a nech $Z = \langle z_1, z_2, \ldots, z_k \rangle$ je ľubovoľná LCS postupností X a Y. Platia nasledujúce tvrdenia:*

1. *Ak $x_m = y_n$, potom $z_k = x_m = y_n$ a Z_{k-1} je LCS pre X_{m-1} a Y_{n-1}.*
2. *Ak $x_m \neq y_n$, potom z $z_k \neq x_m$ vyplýva, že Z je LCS pre X_{m-1} a Y.*
3. *Ak $x_m \neq y_n$, potom z $z_k \neq y_n$ vyplýva, že Z je LCS pre X a Y_{n-1}.*

Dôkaz.

(1) Ak by bolo $z_k \neq x_m$, potom by sme ku Z mohli pridať $x_m = y_n$, aby sme dostali najdlhšiu spoločnú podpostupnosť X a Y dĺžky $k + 1$, čo by bolo v spore s predpokladom, že Z je *najdlhšia* spoločná podpostupnosť X a Y. Preto musí platiť, že $z_k = x_m = y_n$. Teraz, prefix Z_{k-1} je spoločnou podpostupnosťou X_{m-1} a Y_{n-1}. Chceme ukázať, že je LCS. Predpokladajme teraz, že existuje spoločná podpostupnosť W postupností X_{m-1} a Y_{n-1}, ktorej dĺžka je väčšia ako $k-1$. Potom pridaním $x_m = y_n$ do W dostaneme spoločnú podpostupnosť pre X a Y, ktorej dĺžka je väčšia ako k, čo je spor.

(2) Ak $z_k \neq x_m$, potom je Z spoločnou podpostupnosťou X_{m-1} a Y. Ak by existovala spoločná podpostupnosť W postupností X_{m-1} a Y, ktorej dĺžka by bola väčšia ako k, potom by W bola taktiež spoločnou podpostupnosťou X_m a Y, čo by bolo v spore s predpokladom, že Z je LCS pre X a Y.

(3) Podobne ako prípad (2). □

Veta 5.1 hovorí, že LCS dvoch postupností obsahuje LCS prefixov týchto postupností. To znamená, že problém LCS sa vyznačuje vlastnosťou optimálnej podštruktúry.

Rekurzívne riešenie podproblémov

Z vety 5.1 vyplýva, že treba vyšetriť jeden alebo dva podproblémy keď hľadáme LCS postupností $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Ak $x_m = y_n$, musíme nájsť LCS pre X_{m-1} a Y_{n-1}. Pridanie $x_m = y_n$ k tejto LCS dáva LCS pre X a Y. Ak $x_m \neq y_n$, potom musíme vyriešiť dva podproblémy: nájsť LCS pre X_{m-1} a Y a LCS pre X a Y_{n-1}. Najdlhšou spoločnou podpostupnosťou pre X a Y je tá dlhšia z nich.

Ľahko si možno v probléme LCS všimnúť vlastnosť prekrývania podproblémov. Aby sme našli LCS pre X a Y, môže byť treba nájsť LCS pre X a Y_{n-1} a pre X_{m-1} a Y. Oba tieto problémy môžu mať spoločný podproblém nájdenia LCS pre X_{m-1} a Y_{n-1}.

Podobne ako pri probléme násobenia matíc, naše rekurzívne riešenie vyžaduje nájsť rekurenciu pre náklady optimálneho riešenia. Definujme $c[i, j]$ ako dĺžku LCS postupností X_i a Y_j. Ak $i = 0$ alebo $j = 0$, jedna z postupností má nulovú dĺžku, teda LCS má taktiež dĺžku 0. Na základe optimálnej podštruktúry problému nájdenia LCS máme rekurentný vzťah

$$c[i,j] = \begin{cases} 0 & \text{ak } i = 0 \text{ alebo } j = 0\,, \\ c[i-1, j-1] + 1 & \text{ak } i, j > 0 \text{ a } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{ak } i, j > 0 \text{ a } x_i \neq y_j\,. \end{cases} \tag{17}$$

Výpočet dĺžky LCS

Na základe vzťahu 17 môžeme ľahko napísať rekurzívny algoritmus s exponenciálnym časom behu, ktorý počíta dĺžku LCS dvoch postupností. Keďže pre postupnosti dĺžok m a n máme iba $\Theta(mn)$ rôznych podproblémov, na výpočet riešenia môžeme použiť dynamické programovanie.

Vstupom pre procedúru LCS-LENGTH sú dve postupnosti $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Hodnoty $c[i,j]$ sú uchovávané v poli $c[0\,..\,m, 0\,..\,n]$, a vypĺňané sú postupne po riadkoch zľava doprava. Pre zjednodušenie konštrukcie optimálneho riešenia je použité pole $b[1\,..\,m, 1\,..\,n]$. Položka $b[i,j]$ ukazuje na položku prislúchajúcu optimálnemu riešeniu zvolenému pri výpočte $c[i,j]$. Procedúra vracia polia b a c ($c[m,n]$ obsahuje dĺžku LCS pre X a Y).

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m do
 4    c[i, 0] ← 0
 5  for j ← 0 to n do
 6    c[0, j] ← 0
 7  for i ← 1 to m do
 8    for j ← 1 to n do
 9      if x_i = y_i
10        then c[i, j] ← c[i − 1, j − 1] + 1
11             b[i, j] ← "↖"
12        else if c[i − 1, j] ≥ c[i, j − 1]
13               then c[i, j] ← c[i − 1, j]
14                    b[i, j] ← "↑"
15               else c[i, j] ← c[i, j − 1]
16                    b[i, j] ← "←"
17 return c and b
```

Obrázok 33 znázorňuje tabuľku vytvorenú procedúrou LCS-LENGTH pre postupnosti $X = \langle A, B, C, B, D, A, B \rangle$ a $Y = \langle B, D, C, A, B, A \rangle$. Keďže na výpočet každej položky tejto tabuľky treba čas $O(1)$, celkový čas behu tejto procedúry je preto $O(mn)$.

Konštrukcia LCS

Pole b vrátené procedúrou LCS-LENGTH môže byť použité pre skonštruovanie LCS pre $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Jednoducho začneme v $b[m,n]$ a budeme prechádzať po políčkach v smere šípiek. Vždy keď narazíme na šípku "↖" v políčku $b[i,j]$ je $x_i = y_j$ prvkom LCS. Prvky LCS touto metódou prechádzame v opačnom poradí. Nasledujúca procedúra vypíše LCS postupností X a Y v správnom poradí. Počiatočné volanie tejto procedúry je PRINT-LCS($b, X,$ *length*$[X]$, *length*$[Y]$).

```
PRINT-LCS(b, X, i, j)
1  if (i = 0) or (j = 0) then return
2  if b[i, j] = "↖"
3    then PRINT-LCS(b, X, i − 1, j − 1)
4         print x_i
5  else if b[i, j] = "↑"
6         then PRINT-LCS(b, X, i − 1, j)
7         else PRINT-LCS(b, X, i, j − 1)
```

Pre tabuľku b na obr. 33 táto procedúra vypíše "BCBA". Jej čas behu je $O(m+n)$, keďže v jednom kroku rekurzie je dekrementovaný aspoň jeden index z i, j. Azda je ešte vhodné podotknúť, že je možné redukovať pamäťové nároky procedúry LCS-LENGTH, keďže v danom okamihu potrebuje

	j	0	1	2	3	4	5	6
i		y_j	***B***	D	***C***	A	***B***	***A***
0	x_i	0	0	0	0	0	0	0
1	A	**0**	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	***B***	0	↖ **1**	←**1**	←1	↑ 1	↖ 2	←2
3	***C***	0	↑ 1	↑ 1	↖ **2**	←**2**	↑ 2	↑ 2
4	***B***	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ **3**	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ **3**	↑ 3
6	***A***	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ **4**
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ **4**

Obr. 33: Tabuľka (polia c a b) vypočítaná procedúrou LCS-Length pre postupnosti $X = \langle A, B, C, B, D, A, B\rangle$ a $Y = \langle B, D, C, A, B, A\rangle$. Políčko na i-tom riadku a j-tom stĺpci obsahuje hodnotu $c[i, j]$ a príslušnú šípku pre hodnotu $b[i, j]$. Hodnota 4 v $c[7, 6]$ je dĺžkou LCS pre X a Y. Pre $i, j > 0$ závisí hodnota $c[i, j]$ iba na tom, či $x_i = y_i$ a hodnotách $c[i-1, j]$, $c[i, j-1]$ a $c[i-1, j-1]$, ktoré sú vypočítané pred $c[i, j]$. Pre rekonštrukciu prvkov LCS treba sledovať cestu tvorenú šípkami $b[i, j]$ od pravého dolného rohu – každá šípka "↖" na tejto ceste prislúcha položke (vyznačenej hrubšie), pre ktorú $x_i = y_j$ je prvkom LCS.

pre svoju činnosť iba dva riadky poľa c: počítaný riadok a predchádzajúci riadok. Táto úprava je však použiteľná len v prípade, že potrebujeme poznať iba dĺžku hľadanej LCS.

5.2 Greedy algoritmy

Algoritmy pre optimalizačné problémy zvyčajne vykonávajú postupnosť krokov, pričom v každom z týchto krokov si vyberajú z viacerých možností ako pokračovať. V mnohých prípadoch optimalizačných problémov môže byť použitie dynamického programovania nevhodnou voľbou, nakoľko môžu existovať ešte efektívnejšie algoritmy. ***Greedy algoritmus*** si v každom kroku vždy vyberá tú možnosť, ktorá vyzerá najlepšie v danom momente [14]. T.j. vyberá lokálne najlepšiu možnosť dúfajúc, že bude viesť ku globálne najlepšiemu riešeniu. Je zrejmé, že greedy algoritmy neposkytujú vždy optimálne riešenia, ale v mnohých prípadoch problémov k optimálnym riešeniam vedú.

5.2.1 Problém výberu aktivít

Predpokladajme, že máme množinu $S = \{1, 2, \ldots, n\}$ n ***aktivít***. Každá aktivita i má ***počiatočný čas*** s_i a ***koncový čas*** f_i, kde $s_i \leq f_i$, a časový interval, ktorý jej prislúcha je $\langle s_i, f_i)$. Hovoríme, že aktivity i a j sú ***kompatibilné***, ak intervaly $\langle s_i, f_i)$ a $\langle s_j, f_j)$ sa neprekrývajú. ***Problémom výberu aktivít*** je z danej množiny aktivít vybrať najväčší počet navzájom kompatibilných aktivít.

Greedy algoritmus pre problém výberu aktivít je daný nasledujúcim pseudokódom. Predpokladáme, že vstupné aktivity sú usporiadané podľa ich koncových časov vzostupne:

$$f_1 \leq f_2 \leq \cdots \leq f_n \ .$$

V pseudokóde sa predpokladá, že vstupy s a f sú reprezentované pomocou polí.

[14] Greedy = pažravý.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {1}
3  j ← 1
4  for i ← 2 to n do
5    if s_i ≥ f_j
6      then A ← A ∪ {i}
7           j ← i
8  return A
```

Množina A zhromažďuje vybrané aktivity. Premenná j obsahuje číslo aktivity naposledy pridanej do tejto množiny. Keďže aktivity sú prechádzané podľa koncových časov v neklesajúcom poradí, f_j je vždy maximálnym koncovým časom aktivít v A. T.j.

$$f_j = \max\{f_k \ : \ k \in A\} \ .$$

Riadky 2–3 vyberú aktivitu 1, inicializujú A, aby obsahovala iba túto aktivitu a nastavia na ňu j. Cyklus na riadkoch 4–7 bude prechádzať postupne všetky aktivity, pričom aktivitu i zaradí (riadky 6–7) do A, vždy keď je kompatibilná so všetkými zatiaľ vybranými aktivitami v množine A (stačí porovnať jej začiatočný čas s_i s koncovým časom f_j naposledy vybranej aktivity). Procedúra GREEDY-ACTIVITY-SELECTOR je celkom efektívna – n aktivít spracuje v čase $\Theta(n)$ za predpokladu, že sú už vopred utriedené podľa ich koncových časov.

Aktivita vybraná v danom okamihu procedúrou GREEDY-ACTIVITY-SELECTOR je vždy tá, ktorá má najskorší koncový čas a neprekrýva sa s doteraz vybranými aktivitami. Je to lokálne najlepší výber z hľadiska času ponechaného pre ostatné aktivity, ktorý je takto maximalizovaný.

Dôkaz správnosti greedy algoritmu

Keďže greedy algoritmy neposkytujú vždy optimálne riešenia, musíme dokázať, že algoritmus GREEDY-ACTIVITY-SELECTOR vždy nájde najlepšie riešenie pre daný problém výberu aktivít.

Veta 5.2 *Algoritmus* GREEDY-ACTIVITY-SELECTOR *dáva pre problém výberu riešenie maximálnej veľkosti.*

Dôkaz. Nech $S = \{1, 2, \ldots, n\}$ je množina aktivít. Keďže predpokladáme, že aktivity sú utriedené podľa koncového času, aktivita č. 1 má najskorší počiatočný čas. Chceme ukázať, že existuje optimálne riešenie, ktoré začína greedy výberom, t.j. aktivitou č. 1.

Predpokladajme, že $A \subseteq S$ je optimálne riešenie pre daný problém výberu aktivít. Aktivity v A usporiadajme podľa ich koncových časov. Predpokladajme naviac, že prvá aktivita v A je aktivita k. Ak $k = 1$, potom A začína greedy výberom. Ak $k \neq 1$, chceme ukázať, že existuje iné optimálne riešenie B ku S, ktoré začína aktivitou č. 1. Nech $B = A \setminus \{k\} \cup \{1\}$. Keďže $f_1 \leq f_k$, aktivity v B sa neprekrývajú a B ich obsahuje rovnaký počet ako A, preto je B taktiež optimálnym riešením pre S, ktoré naviac začína greedy výberom. Týmto sme ukázali, že ku problému výberu aktivít vždy existuje riešenie začínajúce greedy výberom.

Naviac, po uskutočnení výberu aktivity č. 1 sa problém redukuje na nájdenie optimálneho riešenia pre aktivity v S kompatibilné s aktivitou č. 1. To znamená, že ak A je optimálnym riešením pôvodného problému S, potom $A' = A \setminus \{1\}$ je optimálnym riešením problému $S' = \{i \in S \ : \ s_i \geq f_1\}$. Ak by sme totiž boli schopní nájsť riešenie B' pre S' obsahujúce viac aktivít ako A', pridanie aktivity č. 1 ku B' by viedlo ku riešeniu B pre S s viac aktivitami ako A, čo však nie je možné. Indukciou na počet uskutočnených výberov možno ukázať, že greedy výber v každom kroku vedie ku optimálnemu riešeniu. □

Greedy algoritmus získava optimálne riešenie problému tak, že v každom okamihu keď sa treba rozhodnúť si volí cestu, ktorá vyzerá v danom momente najlepšie. Táto heuristická stratégia nevedie vždy ku optimálnemu riešeniu, ale ako bolo možné vidieť pri probléme výberu aktivít,

niekedy k nemu vedie – vtedy hovoríme, že problém má vlastnosť greedy výberu. Vo všeobecnosti nie je možné jednoznačne rozhodnúť, či greedy stratégia bude pre daný problém úspešná.

0-1 knapsack problém

Majme n predmetov. Nech v_i je hodnota i-teho predmetu a w_i jeho väzba. Úlohou je vybrať predmety takým spôsobom $S \subseteq \{1, \ldots, n\}$, aby súčet $\sum_{i \in S} v_i$ bol maximálny, pričom $\sum_{i \in S} w_i \leq W$, kde W je maximálna väzba.

Zlomkový knapsack problém

V zlomkovom knapsack probléme sa predpokladá, že predmety možno deliť. Majme opäť n predmetov. Nech v_i je hodnota i-teho predmetu a w_i jeho väzba. Úlohou je vybrať predmety takým spôsobom $S = \{s_i\}_{i=1}^{n}$, kde $s_i \in \langle 0, 1 \rangle$, aby súčet $\sum_i s_i v_i$ bol maximálny, pričom $\sum_i s_i w_i \leq W$, kde W je maximálna väzba.

Hoci sa oba knapsack problémy sa vyznačujú vlastnosťou optimálnej podštruktúry, zlomkový knapsack problém je riešiteľný použitím greedy stratégie narozdiel od 0-1 knapsack problému, pri ktorom greedy stratégia zlyháva [15]. Na vyriešenie zlomkového knapsack problému najprv pre každú položku vypočítame jej jednotkovú cenu v_i/w_i. Potom budeme vyberať najväčšie možné množstvo tej položky spomedzi zvyšných položiek, ktorá má najväčšiu jednotkovú cenu. Túto činnosť budeme vykonávať až pokým celková väzba nebude rovná W alebo nevyčerpáme všetky položky. Čas behu tohto algoritmu spolu s utriedením položiek podľa jednotkovej ceny je $O(n \lg n)$.

5.2.2 Huffmanov kód

Huffmanov kód je účinná technika pre kompresiu dát prideľujúca kratší kód znakom vstupnej postupnosti, ktoré majú väčšiu početnosť výskytov, ako znakom, ktoré sa vo vstupnej postupnosti vyskytujú menej často.

	`a`	`b`	`c`	`d`	`e`	`f`
Početnosť	45	13	12	16	9	5
Pôvodný kód	000	001	010	011	100	101
Nový kód	0	101	100	111	1101	1100

Obr. 34: Problém kódovania znakov. Postupnosť 100 znakov z množiny `a`–`f` možno zakódovať do postupnosti 300 bitov, ak každému znaku priradíme 3-bitové kódové slovo. Použitím uvedeného kódu premenlivej dĺžky možno tú istú postupnosť zakódovať do postupnosti 224 bitov.

Existuje mnoho spôsobov ako zakódovať postupnosť znakov. Budeme uvažovať problém vytvorenia ***binárneho kódu***, v ktorom je každý znak reprezentovaný jedinečným binárnym reťazcom. Ak použijeme ***kód pevnej dĺžky***, na reprezentáciu 6 znakov potrebujeme 3 bity:[16] `a` = 000, `b` = 001, ..., `f` = 101.

Použitím ***kódu premenlivej dĺžky*** môžeme dosiahnuť skrátenie výstupnej postupnosti priradením kratšieho kódu znakom s častejším výskytom a dlhšieho kódu znakom, ktoré majú menšiu početnosť výskytov. Na obr. 34 je znázornený príklad takéhoto kódu.

Prefixové kódy

Kódy, ktoré spĺňajú tzv. ***prefixovú vlastnosť***, t.j. žiadne kódové slovo nie je prefixom iného, sa nazývajú ***prefixové kódy***. Keďže je možné ukázať, že pomocou prefixového kódu možno dosiahnuť optimálnu kompresiu priraďujúcu každému znaku jeho kód vo výstupnej postupnosti, môžeme v nasledujúcom bez újmy na všeobecnosti uvažovať iba prefixové kódy.

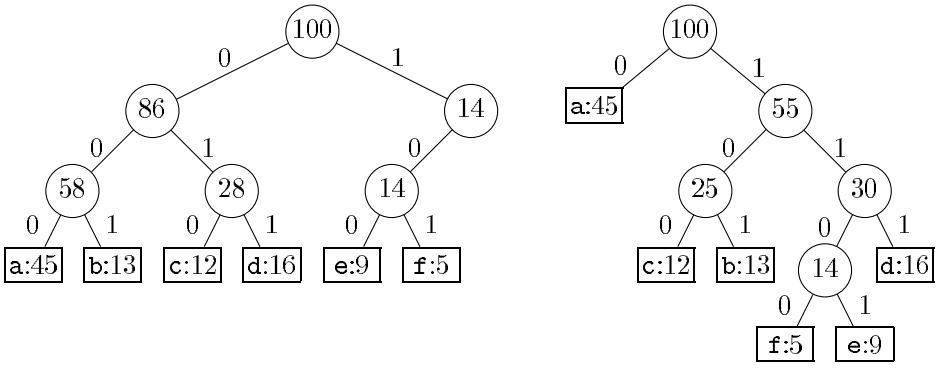
Prefixové kódy sú vhodné z hľadiska zjednodušenia procesu dekódovania (dekódovanie bude určite deterministické). Použitím kódu premenlivej dĺžky z obr. 34 zakódujeme postupnosť znakov

[15] Pre 0-1 knapsack problém však možno úspešne použiť dynamické programovanie.

[16] Ak by sme použili číselnú sústavu so základom 6, bolo by možné znak zakódovať do $\lg 6 \approx 2{,}6$ bitu.

`aabc` ako $0 \cdot 0 \cdot 101 \cdot 100 = 00101100$, kde „$\cdot$“ označuje operáciu zreťazenia. Keďže žiadne kódové slovo nie je prefixom iného, môžeme jednoducho identifikovať počiatočné kódové slovo, preložiť ho naspäť na pôvodný znak, odstrániť ho zo vstupnej postupnosti a tento proces opakovať na zvyšku tejto postupnosti. V našom príklade možno reťazec 00101100 rozložiť jednoznačne na $0 \cdot 0 \cdot 101 \cdot 100$, čo po dekódovaní dáva opäť `aabc`.

Aby kódové slová mohli byť jednoducho rozpoznané, pri procese dekódovania potrebujeme vhodnú reprezentáciu pre prefixový kód. Vhodným riešením je binárny strom, ktorého listy sú dané znaky. Binárne kódové slovo znaku interpretujeme ako cestu od koreňa ku tomuto znaku tak, že 0 znamená „choď doľava“ a 1 znamená „choď doprava“. Obr. 35 znázorňuje stromy pre kódy uvedené vyššie.



(a) (b)

Obr. 35: Stromy prisluchajúce kódom na obr. 34. Každý list je označený znakom a početnosťou jeho výskytu. Každý vnútorný uzol je označený súčtom hodnôt listov jeho podstromu. (**a**) Strom prislúchajúci kódu pevnej dĺžky `a` $= 100, \ldots,$ `f` $= 101$. (**b**) Strom prislúchajúci optimálnemu prefixovému kódu `a` $= 0,$ `b` $= 101, \ldots,$ `f` $= 1100$.

Optimálny kód pre postupnosť znakov je vždy reprezentovaný *úplným* binárnym stromom, v ktorom každý vnútorný uzol má dvoch synov. Kód pevnej dĺžky z obr. 34 nie je optimálny, nakoľko jeho strom na obr. 35a nie je úplným binárnym stromom (je možné vytvoriť kódové slová začínajúce $10\ldots$, ale nie $11\ldots$). Keďže sa teraz budeme zaoberať úplnými binárnymi stromami, môžeme povedať, že ak C je abeceda vstupných znakov, potom strom pre optimálny prefixový kód má presne $|C|$ listov, jeden pre každý znak tejto abecedy, a presne $|C| - 1$ vnútorných uzlov.

Nech T je strom prislúchajúci prefixovému kódu. Pre každý znak c abecedy C označme $f(c)$ početnosť jeho výskytov vo vstupnej postupnosti a $d_T(c)$ hĺbku listu prislúchajúceho znaku c. Počet bitov potrebných na zakódovanie vstupnej postupnosti je rovný

$$B(T) = \sum_{c \in C} f(c) d_T(c) \,, \tag{18}$$

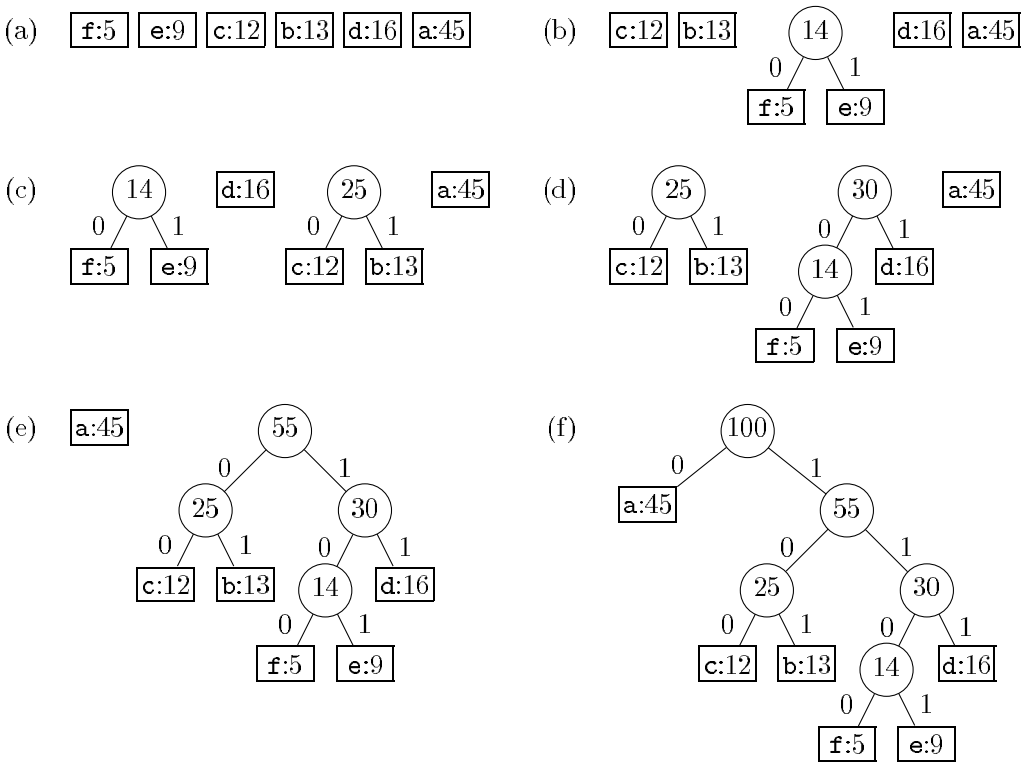
čo definujeme ako ***cenu*** stromu T.

Konštrukcia Huffmanovho kódu

Teraz popíšeme greedy algoritmus, ktorý vytvorí optimálny prefixový kód nazývaný ***Huffmanov kód***. Algoritmus zostrojí strom T prislúchajúci optimálnemu kódu spôsobom zdola nahor. Začína množinou $|C|$ listov a vykonáva postupnosť $|C|-1$ spájajúcich operácií, aby vytvoril finálny strom.

V nasledujúcom pseudokóde predpokladáme, že C je množina n znakov, a že každý znak $c \in C$ je objekt s definovanou početnosťou výskytu $f[c]$. Prioritná fronta Q kľúčovaná na f je použitá na identifikáciu dvoch objektov s najmenšou frekvenciou výskytu, ktoré budú zlúčené do jedného. Výsledkom zlúčenia dvoch objektov je objekt, ktorého frekvencia výskytu je súčtom frekvencií výskytov týchto objektov.

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1 do
4     z ← ALLOCATE-NODE()
5     x ← left[z] ← EXTRACT-MIN(Q)
6     y ← right[z] ← EXTRACT-MIN(Q)
7     f[z] ← f[x] + f[y]
8     INSERT(Q, z)
9  return EXTRACT-MIN(Q)
```



Obr. 36: Kroky Huffmanovho algoritmu pre frekvencie výskytov uvedené na obr. 34. V každej časti obrázku je znázornený obsah prioritnej fronty utriedený vzostupne podľa frekvencie. V každom kroku sú zlúčené dva stromy s najmenšími frekvecniami výskytov. Listy sú znázornené ako obdĺžniky obsahujúce znak a jeho početnosť. Vnútorné uzly sú znázornené kružnicami obsahujúcimi súčty početností ich potomkov. Hrana spájajúca vnútorný uzol s ľavým potomkom je označená 0, s pravým potomkom je označená 1. Kódové slovo pre daný znak je postupnosť núl a jednotiek na hranách spájajúcich koreň s príslušným listom. (**a**) Počiatočná množina $n = 6$ uzlov. (**b**)–(**e**) Ďalšie štádiá prioritnej fornty. (**f**) Finálny strom.

Pre postupnosť znakov s frekvenciami výskytov uvedenými na obr. 34 je činnosť Huffmanovho algoritmu znázornená na obr. 36. Keďže v abecede je 6 znakov, počiatočná veľkosť fronty je $n = 6$ a na vytvorenie stromu je potrebných 5 zlúčení. Finálny strom reprezentuje optimálny prefixový kód. Kódové slovo pre daný znak je postupnosť núl a jednotiek na hranách cesty od koreňa ku tomuto znaku.

Prioritná fronta Q je inicializovaná na riadku 2 vložením všetkých znakov množiny C. Cyklus `for` na riadkoch 3–8 opakovane z fronty vyberá dva uzly x a y, ktoré majú najmenšiu frekvenciu výskytu, a nahradzuje ich novým uzlom z, ktorý vznikol spojením x a y (v ľubovoľnom poradí).

Frekvencia z je rovná súčtu frekvencií x a y (riadok 7). Po $n-1$ zlúčeniach vo fronte zostane iba jeden uzol – koreň stromu reprezentujúceho optimálny kód.

Analýza času behu Huffmanovho algoritmu predpokladá, že Q je implementovaná ako binárna halda. Pre množinu C obsahujúcu n prvkov možno inicializáciu Q použitím procedúry BUILD-HEAP uskutočniť v čase $O(n)$. Telo cyklu `for` na riadkoch 3–8 je vykonaný presne $(|n|-1)$-krát a keďže každá operácia na halde vyžaduje čas $O(\lg n)$, tento cyklus prispieva časom $O(n \lg n)$ ku celkovému času behu. Teda celkový čas behu procedúry HUFFMAN na množine n znakov je $O(n \lg n)$.

Korektnosť Huffmanovho algoritmu

Aby sme mohli dokázať, že greedy algoritmus HUFFMAN je správny, najprv ukážeme, že problém určenia optimálneho prefixového kódu sa vyznačuje vlastnosťou greedy výberu a optimálnou podštruktúrou.

Lema 5.1 *Nech C je abeceda a každý jej znak $c \in C$ má frekvenciu výskytu $f[c]$. Nech x, y sú dva znaky v C, ktoré majú najnižšiu frekvenciu. Potom existuje optimálny prefixový kód, pri ktorom kódové slová pre x a y majú rovnakú dĺžku a líšia sa len v poslednom bite.*

Dôkaz. Myšlienkou dôkazu je vziať strom T reprezentujúci ľubovoľný optimálny prefixový kód a modifikovať ho takým spôsobom, aby naďalej reprezentoval optimálny prefixový kód, ale aby znaky x a y boli v novom strome súrodencami maximálnej hĺbky. Ak toto môžeme uskutočniť, potom ich kódové slová sa budú mať rovnakú dĺžku, pričom sa budú líšiť len v poslednom bite.

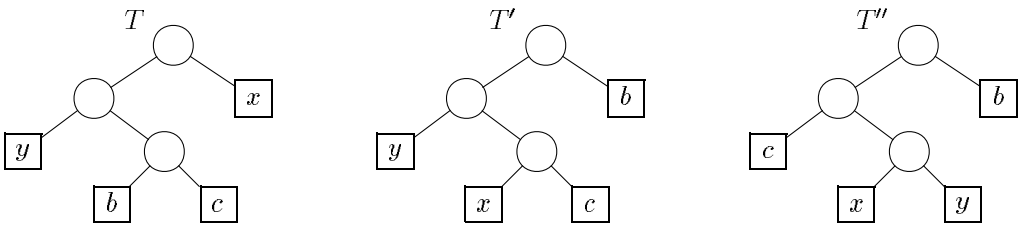
Nech b a c sú dva znaky, ktoré sú susednými listami maximálnej hĺbky v strome T. Bez újmy na všeobecnosti predpokladajme, že $f[b] \leq f[c]$ a $f[x] \leq f[y]$. Keďže $f[x]$ a $f[y]$ sú najnižšie frekvencie a $f[b]$ a $f[c]$ sú ľubovoľné frekvencie, máme $f[x] \leq f[b]$ a $f[y] \leq f[c]$. Ako možno vidieť na obr. 37, aby sme získali strom T' vymeníme v strome T pozície b a x. Následne vymeníme c a y, aby sme dostali strom T''. Podľa vzťahu 18 je rozdiel cien T a T' rovný

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f[c] d_T(c) - \sum_{c \in C} f[c] d_{T'}(c) \\ &= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\ &= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\ &= (f[b] - f[x])(d_T(b) - d_T(x)) \\ &\geq 0 \,, \end{aligned}$$

keďže oba rozdiely $f[b] - f[x]$ a $d_T[b] - d_T[x]$ sú nezáporné. Presnejšie, $f[b] - f[x]$ je nezáporné lebo x je list s minimálnou frekvenciou a $d_T[b] - d_T[x]$ je nezáporné lebo b je list maximálnej hĺbky v T. Z podobných dôvodov, keďže výmena y a c nespôsobí nárast ceny stromu, je rozdiel $B(T') - B(T'')$ taktiež nezáporný. To znamená, že $B(T'') \leq B(T)$. Keďže T je optimálny strom máme naviac nerovnosť $B(T) \leq B(T'')$, z čoho vyplýva, že $B(T'') = B(T)$, a teda strom T'' je optimálny strom, v ktorom sa x a y nachádzajú na najnižšej úrovni, pričom sú potomkami toho istého uzla, z čoho už priamo vyplýva tvrdenie lemy. □

Z lemy 5.1 vyplýva, že proces vytvárania optimálneho stromu môže začínať greedy výberom, t.j. zlúčením dvoch znakov s najmenšou frekvenciou výskytu. Nasledujúca lema ukazuje, že problém konštrukcie optimálneho prefixového kódu má vlastnosť optimálnej podštruktúry.

Lema 5.2 *Nech T je úplný binárny strom, ktorý reprezentuje optimálny prefixový kód nad abecedou C. Uvažujme ľubovoľné dva znaky x a y, ktoré sú súrodencami v T, a nech z je ich rodič. Potom za predpokladu, že $f[z] = f[x] + f[y]$, strom $T' = T \setminus \{x, y\}$ reprezentuje optimálny prefixový kód pre abecedu $C' = (C \setminus \{x, y\}) \cup \{z\}$.*



Obr. 37: Ilustrácia kľúčového kroku v dôkaze lemy 5.1. V optimálnom strome T, listy b a c sú dva z najhlbších listov, pričom sú súrodencami. Listy x a y sú tie, ktoré Huffmanov algoritmus zlúči ako prvé; nachádzajú sa na ľubovoľných pozíciách v T. Listy b a x sú vymenené, aby sme získali strom T'. Následne sú vymenené listy c a y, aby sme získali strom T''. Keďže žiadna zámena nespôsobí nárast ceny stromu, výsledný strom T'' je preto optimálny.

Dôkaz. Najprv ukážeme, že cenu $B(T)$ stromu T možno vyjadriť na základe ceny $B(T')$ stromu T'. Pre každé $c \in C \setminus \{x, y\}$ máme $d_T(c) = d_{T'}(c)$, a preto $f[c]d_T(c) = f[c]d_{T'}(c)$. Keďže $d_T(x) = d_T(y) = d_{T'}(z) + 1$, máme

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) \ , \end{aligned}$$

z čoho ďalej

$$B(T) = B(T') + f[x] + f[y] \ .$$

Ak by T' reprezentoval neoptimálny prefixový kód pre abecedu C', potom by existoval strom T'', ktorého listami sú znaky C', taký, že $B(T'') < B(T')$. Keďže z je znak z C', nachádza sa ako list v strome T''. Ak by sme pridali x a y ako synov uzla z v T'', potom by sme získali prefixový kód pre C s cenou $B(T'') + f[x] + f[y] < B(T)$, čo by bol spor s optimálnosťou stromu T. Teda T' musí byť optimálny pre abecedu C'. □

Veta 5.3 *Procedúra* HUFFMAN *produkuje optimálny prefixový kód.*

Dôkaz. Vyplýva priamo z liem 5.1 a 5.2. □

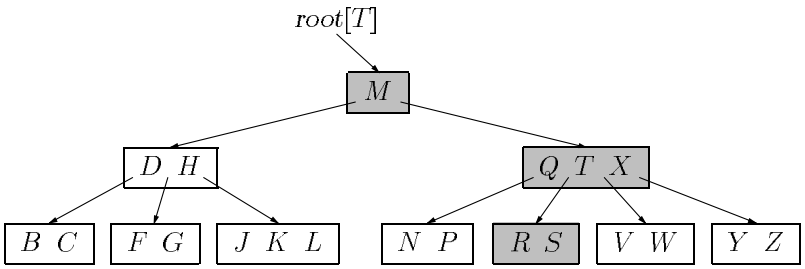
6 Pokročilé dátové štruktúry

V nasledujúcom sa budeme zaoberať B-stromami. B-strom je vyvážený vyhľadávací strom navrhnutý špeciálne pre uloženie na magnetickom disku. Keďže magnetické disky sú značne pomalšie ako operačná pamäť, efektívnosť B-stromov určujeme nielen na základe výpočtového času potrebného pre operácie na dynamických množinách, ale aj na základe počtu vykonaných prístupov k disku. Pre každú operáciu na B-strome sa počet prístupov k disku zväčšuje s výškou tohto B-stromu, preto operácie na B-stromoch sú navrhnuté takým spôsobom, aby bola čo najmenšia.

6.1 B-stromy

B-stromy sú vyvážené prehľadávacie stromy navrhnuté pre uloženie na magnetických diskoch. Istým spôsobom sa podobajú RB-stromom (výška každého uzla v n-uzlovom B-strome je $O(\lg n)$), ale sú výhodnejšie z hľadiska minimalizácie diskových operácií. Uzly B-stromov môžu mať mnoho potomkov od niekoľkých až po tisíce, preto ich výška môže byť značne menšia ako výška RB-stromov s rovnakým počtom uzlov.

B-stromy sú zovšeobecnením binárnych prehľadávacích stromov. Príklad B-stromu je znázornený na obr. 38. Ak uzol x B-stromu obsahuje $n[x]$ kľúčov, potom má $n[x] + 1$ potomkov. Kľúče uzla x sú použité ako deliace body rozdeľujúce množinu kľúčov spadajúcich pod x na $n[x] + 1$ častí. Ak hľadáme kľúč v B-strome, z daného uzla x môžeme pokračovať v jednom z $n[x] + 1$ synov uzla x. Rozhodnúť sa možno na základe hodnôt kľúčov uložených v uzle x.



Obr. 38: Príklad B-stromu, ktorého kľúčami sú spoluhlásky. Vnútorný uzol x obahuje $n[x]$ kľúčov a má $n[x]+1$ potomkov. Všetky listy majú rovnakú hĺbku. Pri hľadaní písmena R sú navštívené uzly označené šedou farbou.

6.1.1 Definícia B-stromov

Pre jednoduchosť budeme predpokladať, že podobne ako pri binárnych prehľadávacích stromoch, sú všetky satelitné dáta uložené v tom istom uzle ako kľúč. V praxi by sme samozrejme spolu s kľúčom uložili iba ukazovateľ na stránku obsahujúcu satelitné dáta pre tento kľúč. Ďalšia bežne používaná organizácia B-stromu ukladá všetky satelitné dáta v listoch a vo vnútorných uzloch ukladá iba kľúče a ukazovatele na potomkov.
B-strom T je strom, ktorý spĺňa nasledujúce vlastnosti.

1. Každý uzol x má nasledujúce položky:
   a. $n[x]$ – počet kľúčov práve uložených v uzle x,
   b. samotné kľúče uložené v neklesajúcom poradí: $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$,
   c. $leaf[x]$ – booleovská hodnota, ktorá je TRUE ak x je list a FALSE, ak x je vnútorný uzol.
2. Ak x je vnútorný uzol, obsahuje taktiež $n[x]+1$ ukazovateľov $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ na svojich potomkov. Listy nemajú potomkov, preto sú hodnoty ich položiek c_i nedefinované.
3. Kľúče $key_i[x]$ oddeľujú rozsahy kľúčov uložených v každom podstrome: ak k_i je ľubovoľný kľúč uložený v podstrome s koreňom $c_i[x]$, potom
$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \cdots \leq key_{n[x]}[x] \leq k_{n[x]+1} .$$
4. Všetky listy majú rovnakú hĺbku, ktorá sa rovná výške stromu h.
5. Počet potomkov uzla je ohraničený zhora aj zdola, pričom ho možno vyjadriť na základe pevne zvoleného celého čísla $t \geq 2$ nazývaného ***minimálny stupeň*** B-stromu:
   a. Každý uzol s výnimkou koreňa musí mať aspoň $t-1$ kľúčov. Každý vnútorný uzol okrem koreňa má teda aspoň t potomkov. Ak je strom neprázdny, koreň musí obsahovať aspoň jeden kľúč.
   b. Každý uzol môže obsahovať nanajvýš $2t-1$ kľúčov. Vnútorné uzly môžu preto mať nanajvýš $2t$ potomkov. Hovoríme, že uzol je ***plný***, ak obsahuje presne $2t-1$ kľúčov.

Najjednoduchší prípad B-stromu (***2-3-4 strom***) nastáva, ak $t = 2$. Každý vnútorný uzol potom má buď 2, 3 alebo 4 potomkov. V praxi sa používajú B-stromy pre omnoho väčšie hodnoty t.

Výška B-stromu

Počet prístupov k disku potrebný pre väčšinu operácií na B-strome je priamo úmerný výške tohto B-stromu.

Veta 6.1 *Ak $n \geq 1$, potom pre každý n-kľúčový B-strom T výšky h s minimálnym stupňom $t \geq 2$ platí*
$$h \leq \log_t \frac{n+1}{2} .$$

Dôkaz. Ak B-strom má výšku h, potom počet jeho uzlov je najmenší, ak koreň obsahuje jeden kľúč a všetky ostatné uzly obsahujú $t-1$ kľúčov. V tomto prípade na prvej úrovni sa nachádzajú 2 uzly, na druhej úrovni $2t$ uzlov, na tretej $2t^2$ uzlov, atď. až po úroveň h, kde je $2t^{h-1}$ uzlov. Teda počet n kľúčov spĺňa nerovnosť

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \,, \end{aligned}$$

z čoho priamo vyplýva tvrdenie vety. □

Hoci je výška B-stromov aj RB-stromov rovná $O(\lg n)$, pre B-stromy môže byť základ logaritmu omnoho väčší. Počet uzlov, ktoré treba načítať z disku pri väčšine stromových operácií, je teda približne $(\lg t)$-krát menší ako pre RB-stromy.

6.1.2 Základné operácie na B-stromoch

V tejto časti prezentujeme operácie B-Tree-Search, B-Tree-Create a B-Tree-Insert. Pri týchto procedúrach si osvojíme dve konvencie:

- Koreň B-stromu je vždy v operačnej pamäti, takže ho nikdy netreba z disku čítať (Disk-Read). Zapísať (Disk-Write) ho treba vždy po jeho modifikácii.
- Na všetkých uzloch predávaných ako parametre musí byť už vopred vykonaná operácia Disk-Read.

Vyhľadávanie v B-strome

Vyhľadávanie v B-strome sa podobá vyhľadávaniu v binárnom prehľadávacom strome, s tým rozdielom, že namiesto výberu jednej z dvoch možných ciest máme viac možností v závislosti od počtu potomkov uzla.

B-Tree-Search je priamočiarym zovšeobecnením procedúry Tree-Search definovanej pre binárne prehľadávacie stromy. Vstupom procedúry B-Tree-Search je ukazovateľ na koreň x podstromu a kľúč k, ktorý v tomto podstrome hľadáme. Volanie na najvyššej úrovni je teda B-Tree-Search($root[T], k$). Ak sa k nachádza v B-strome, B-Tree-Search vráti usporiadanú dvojicu (y, i) pozostávajúcu z uzla y a indexu i takého, že $key_i[y] = k$. Inak je vrátená hodnota NIL.

B-Tree-Search(x, k)

```
1  i ← 1
2  while (i ≤ n[x]) and (k > key_i[x]) do
3     i ← i + 1
4  if (i ≤ n[x]) and (k = key_i[x])
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else Disk-Read(c_i[x])
9          return B-Tree-Search(c_i[x], k)
```

Na riadkoch 1–3 je nájdená najmenšia hodnota i taká, že $k \leq key_i[x]$, alebo ak neexistuje, tak premennej i je priradená hodnota $n[x] + 1$. Riadky 4–5 overujú, či sme našli hľadaný kľúč. Ak áno, beh procedúry je ukončený. Riadky 6–9 buď ukončia hľadanie, pričom indikujú neúspech hľadania (ak x je list), alebo rekurzívne pokračujú v hľadaní v príslušnom podstrome x po vykonaní potrebnej operácie Disk-Read. Obr. 38 znázorňuje činnosť procedúry B-Tree-Search.

Podobne ako pri procedúre TREE-SEARCH uzly, ktoré treba prejsť pri hľadaní kľúča, sa nachádzajú na ceste od koreňa smerom nadol. Preto je počet diskových stránok načítaných procedúrou B-TREE-SEARCH rovný $\Theta(h) = \Theta(\log_t n)$, kde h je výška a n je počet kľúčov B-stromu. Keďže $n[x] < 2t$, čas behu cyklu `while` na riadkoch 2–3 pre každý uzol je $O(t)$, a teda celkový výpočtový čas je rovný $O(th) = O(t \log_t n)$.

Vytvorenie prázdneho B-stromu

Na vytvorenie B-stromu T najprv použijeme B-TREE-CREATE, aby sme vytvorili prázdny koreň, a potom budeme vyvolávať B-TREE-INSERT na vloženie nových kľúčov. Obe tieto procedúry používajú pomocnú procedúru ALLOCATE-NODE (s časom behu $O(1)$), ktorá alokuje jednu diskovú stránku pre nový uzol.

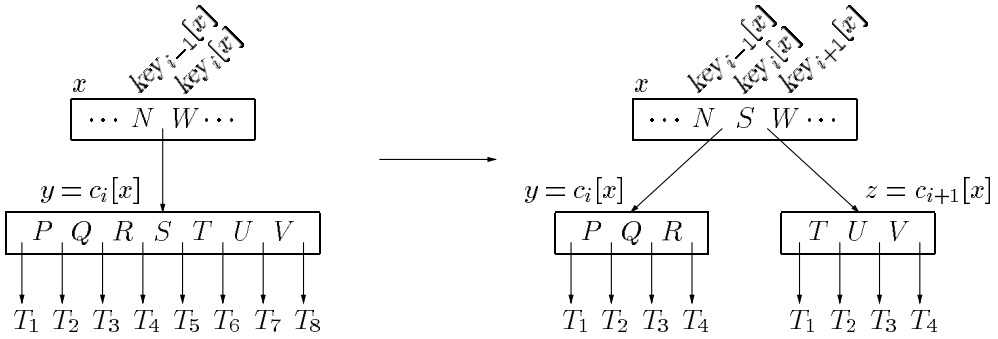
B-TREE-CREATE(T)
1 $x \leftarrow$ ALLOCATE-NODE()
2 $leaf[x] \leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 DISK-WRITE(x)
5 $root[T] \leftarrow x$

Procedúra B-TREE-CREATE vyžaduje $O(1)$ diskových operácií a $O(1)$ času procesora.

Rozdeľovanie uzla v B-strome

Vkladanie kľúča do B-stromu je značne komplikovanejšie ako vkladanie do binárneho prehľadávacieho stromu. Základnou operáciou použitou počas vkladania je ***rozdelenie*** plného uzla y (obsahujúceho $2t-1$ kľúčov) okolo ***prostredného kľúča*** $key_t[y]$ na dva uzly po $t-1$ kľúčoch. Prostredný kľúč je presunutý nahor do otca uzla y, ktorý nesmie byť plný. Tento kľúč je potom použitý ako deliaci bod medzi novými stromami. Ak y nemá otca, potom výška stromu narastie o 1.

Procedúra B-TREE-SPLIT-CHILD vezme ako vstup vnútorný uzol, ktorý *nie je plný* (predpokladáme, že sa už nachádza v pamäti), index i a uzol y taký, že $y = c_i[x]$ je *plným* synom uzla x. Procedúra potom tohto syna rozdelí na dva uzly, pričom príslušne modifikuje uzol x.

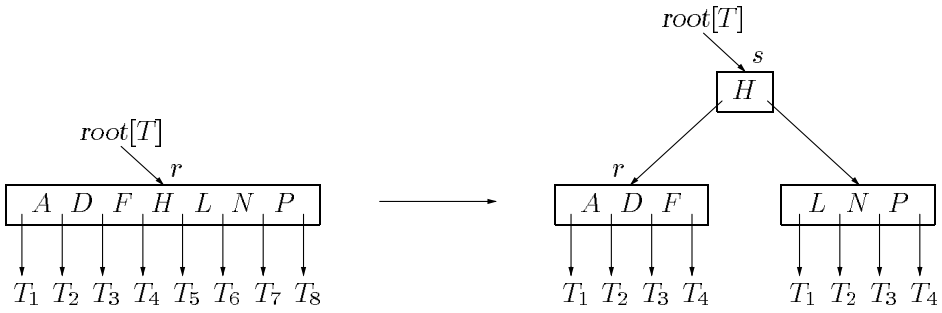


Obr. 39: Rozdelenie uzla pre $t = 4$. Uzol y je rozdelený na dva uzly y a z, pričom jeho prostredný prvok je presunutý do jeho otca.

Tento proces je znázornený na obr. 39. Plný uzol y je rozdelený okolo prostredného prvku S, ktorý je následne presunutý do uzla x, ktorý je rodičom y. Kľúče, ktoré sú väčšie ako tento prostredný prvok, sú premiestnené do nového uzla z. Tento uzol sa stáva novým synom x.

```
B-Tree-Insert(T, k)
1  r ← root[T]
2  if n[r] = 2t − 1
3    then s ← Allocate-Node()
4         root[T] ← s
5         n[s] ← 0
6         c_1[s] ← r
7         B-Tree-Split-Child(s, 1, r)
8         B-Tree-Insert-Nonfull(s, k)
9    else B-Tree-Insert-Nonfull(r, k)
```

Riadky 3–8 ošetrujú prípad, v ktorom je koreň plný. Koreňom sa stáva nový uzol s. Rozdelenie koreňa je jediný spôsob, ktorým možno zväčšiť výšky B-stromu. Tento prípad je znázornený na obr. 40. Procedúra končí vyvolaním B-Tree-Insert-Nonfull, ktoré vloží kľúč k do stromu



Obr. 40: Rozdelenie koreňa pre $t = 4$. Koreň r je rozdelený na dve časti, pričom je vytvorený nový uzol s, ktorý sa stáva koreňom. Nový koreň obsahuje prostredný prvok r, pričom jeho synmi sú dve polovice pôvodného koreňa. Pri tejto operácii narastá výška B-stromu.

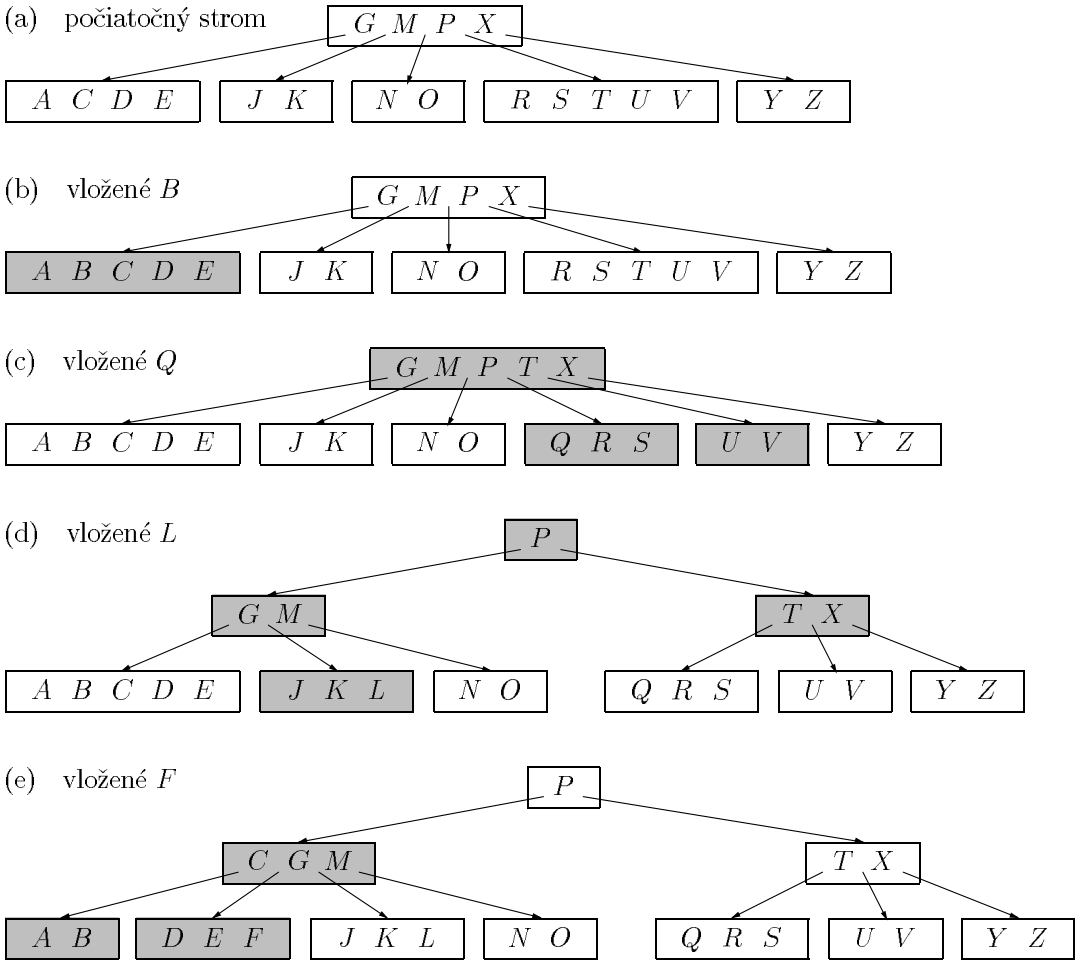
s koreňom, ktorý nie je plný. B-Tree-Insert-Nonfull prechádza strom postupne nadol, pričom každý plný uzol, ktorý ide navštíviť, rozdelí vyvolaním B-Tree-Split-Child.

Pomocná rekurzívna procedúra B-Tree-Insert-Nonfull vloží kľúč k do uzla x, o ktorom predpokladá, že nie je plný v okamihu jej vyvolania.

```
B-Tree-Split-Child(x, i, y)
1  z ← Allocate-Node()
2  leaf[z] ← leaf[y]
3  n[z] ← t − 1
4  for j ← 1 to t − 1 do
5    key_j[z] ← key_{j+t}[y]
6  if not leaf[y] then
7    for j ← 1 to t do
8      c_j[z] ← c_{j+t}[y]
9  n[y] ← t − 1
10 for j ← n[x] + 1 downto i + 1 do
11   c_{j+1}[x] ← c_j[x]
12 c_{i+1}[x] ← z
13 for j ← n[x] downto i do
14   key_{j+1}[x] ← key_j[x]
15 key_i[x] ← key_t[x]
16 n[x] ← n[x] + 1
17 Disk-Write(y)
18 Disk-Write(z)
19 Disk-Write(x)
```

```
B-Tree-Insert-Nonfull(x, k)
1  i ← n[x]
2  if leaf[x]
3    then while (i ≥ 1) and (k < key_i[x]) do
4           key_{i+1}[x] ← key_i[x]
5           i ← i − 1
6         key_{i+1}[x] ← k
7         n[x] ← n[x] + 1
8         Disk-Write(x)
9    else while (i ≥ 1) and k < key_i[x] do
10          i ← i − 1
11        i ← i + 1
12        Disk-Read(c_i[x])
13        if n[c_i[x]] = 2t − 1 then
14          B-Tree-Split-Child(x, i, c_i[x])
15          if k > key_i[x] then
16            i ← i + 1
17        B-Tree-Insert-Nonfull(c_i[x], k)
```

Procedúra B-Tree-Split-Child pracuje priamočiaro metódou „cut & paste“. Rozdeľovaný uzol y je i-tym potomkom uzla x. Počet jeho potomkov je vykonaním tejto operácie znížený z $2t-1$ na $t-1$. Najväčších $t-1$ synov uzla y je premiestnených do uzla z (riadky 1–9), ktorý sa stáva novým potomkom x. Tento uzol je umiestnený do tabuľky potomkov uzla x hneď za y (riadky 10–12). Prostredný kľúč y je presunutý nahor do uzla x na pozíciu medzi y a z (riadky 13–16). Modifikované diskové stránky sú zapísané na riadkoch 17–19. Výpočtový čas spotrebovaný procedúrou B-Tree-Split-Child je rovný $\Theta(t)$.



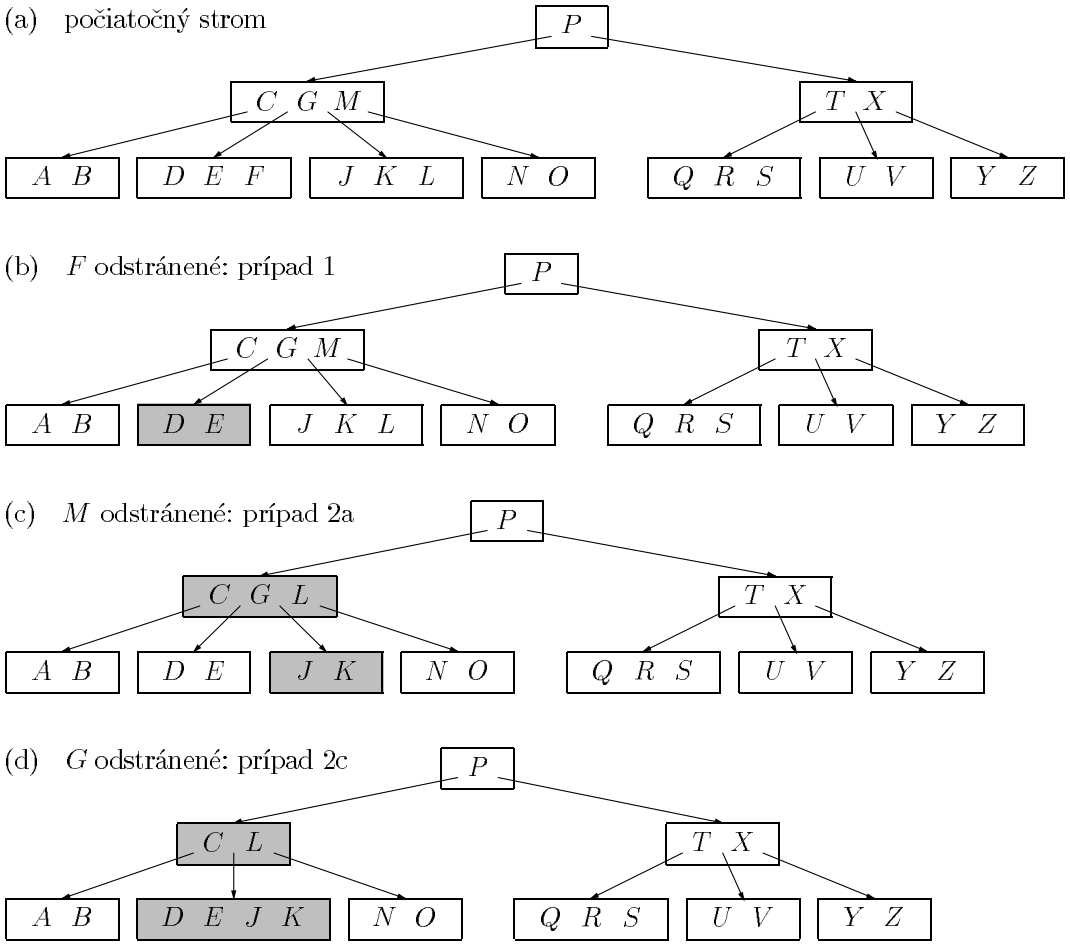
Obr. 41: Vkladanie kľúčov do B-stromu. Minimálny stupeň t tohto B-stromu je 3, teda uzol môže obsahovať nanajvýš 5 kľúčov. Uzly, ktoré sú modifikované operáciou vkladania sú vyznačené šedou farbou. (**a**) Počiatočný strom. (**b**) Výsledok vloženia B do počiatočného stromu. Bolo vykonané jednoduché vloženie do listu. (**c**) Výsledok vloženia Q do predchádzajúceho stromu. Uzol $RSTUV$ je rozdelený na dva uzly obsahujúce RS a UV, kľúč T je premiestnený do koreňa a Q je vložený do uzla RS. (**d**) Výsledok vloženia L do predchádzajúceho stromu. Koreň je rozdelený, čím výška stromu narastie o 1. Potom je L vložené do listu obsahujúceho JK. (**e**) Výsledok vloženia F do predchádzajúceho stromu. Uzol $ABCDE$ je rozdelený predtým ako je F vložené do jeho pravej polovice (uzol DE).

Vkladanie kľúča do B-stromu

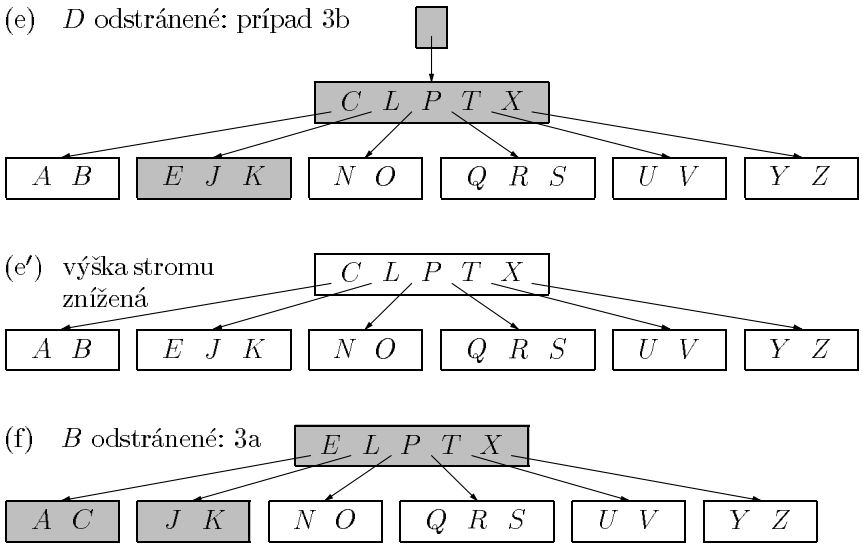
Vkladanie kľúča k do B-stromu T výšky h možno vykonať na jeden priechod stromom smerom nadol, čo vyžaduje $O(h)$ prístupov k disku. Potrebný výpočtový čas je rovný $O(th) = O(t \log_t n)$. Procedúra B-Tree-Insert používa B-Tree-Split, aby bolo zaručené, že počas rekurzie nenarazíme na plný uzol.

Procedúra B-Tree-Insert-Nonfull pracuje nasledovným spôsobom. Na riadkoch 3–8 je ošetrený prípad, keď x je list. Vtedy je kľúč k jednoducho do uzla x vložený. Ak x nie je list, potom musíme k vložiť do príslušného listu nachádzajúceho sa v podstrome s koreňom v x. V tomto prípade, riadky 9–11 určia syna x, v ktorom bude rekurzia pokračovať. Kód na riadku 13 zistí, či sa jedná o plný uzol. Ak áno, tento uzol bude na riadku 14 rozdelený použitím B-Tree-Split-Child na dva uzly, ktoré nie sú plné. Následne, na riadkoch 15–16 sa určí, v ktorom z týchto uzlov sa bude pokračovať. Účelom kódu na riadkoch 13–16 je teda zabezpečiť, aby procedúra na riadku 17 nenarazila na plný uzol. Obr. 41 znázorňuje rôzne prípady vkladania do B-stromu.

Počet prístupov k disku vykonaných procedúrou B-Tree-Insert je $O(h)$ pre B-strom výšky h nakoľko medzi volaniami B-Tree-Insert-Nonfull je vykonaných iba $O(1)$ operácií Disk-Read a Disk-Write. Celkový použitý výpočtový čas je $O(th) = O(t \log_t n)$. Keďže je procedúra B-Tree-Insert-Nonfull chvostovo-rekurzívna, možno ju implementovať pomocou cyklu `while`. To znamená, že počet stránok, ktoré sa musia naraz nachádzať v operačnej pamäti, je ohraničený $O(1)$.



Obr. 42: Vymazávanie kľúčov z B-stromu. Minimálny stupeň t tohto B-stromu je 3, teda uzol (s výnimkou koreňa) musí obsahovať minimálne 2 kľúče. Uzly, ktoré sú modifikované operáciou vymazávania sú vyznačené šedou farbou. (**a**) B-strom z obr. 41e. (**b**) Odstránenie F. Prípad 1 – jednoduché odstránenie z listu. (**c**) Odstránenie M. Prípad 2a – predchodca L kľúča M je presunutý nahor na pôvodnú pozíciu M. (**d**) Odstránenie G. Prípad 2c – G je premiestnený nadol, aby bol vytvorený uzol $DEGJK$, z ktorého je následne G vymazaný.



Obr. 43: Pokračovanie obr. 42. (**e**) Odstránenie D. Prípad 3b – rekurzia nemôže pokračovať v uzle CL lebo má iba 2 kľúče. Z tohto dôvodu je P presunutý nadol, zlúčený s CL a TX, čím je vytvorený uzol $CLPTX$, ktorý už obsahuje dostatočný počet kľúčov na to, aby mohol byť D z neho odstránený (prípad 1). (**e′**)Prázdny koreň, ktorý vznikol v (d) je odstránený, čím výška stromu klesne. (**f**) Odstránenie B. Prípad 3a – C je presunutý na pozíciu B a E je premiestnený na pôvodnú pozíciu C.

6.1.3 Vymazávanie kľúča z B-stromu

Vymazávanie z B-stromu je trochu komplikovanejšie ako vkladanie, preto nebudeme uvádzať celý pseudokód, ale len načrtneme spôsob, akým vymazať daný kľúč z B-stromu. Predpokladajme, že procedúra B-Tree-Delete má vymazať kľúč k z podstromu s koreňom x. Táto procedúra je zostavená tak, aby bolo vždy zaručené, že vždy keď je B-Tree-Delete vyvolaná na uzle x, počet kľúčov x je aspoň t (t.j. o 1 kľúč viac ako minimálny počet). Preto niekedy treba vykonať presun kľúča do potomka ešte predtým ako naň procedúra rekurzívne narazí. Táto zosilnená podmienka nám umožňuje vymazať kľúč zo stromu na jeden priechod smerom nadol bez potreby návratu (s jednou výnimkou, ktorú vysvetlíme). Ak niekedy nastane situácia, že sa koreň x stane vnútorným uzlom bez kľúčov, potom ho zo stromu vymažeme, pričom jeho jediný potomok $c_1[x]$ sa stane novým koreňom stromu, čím sa zníži celková výška stromu (vlastnosť, že koreň stromu obsahuje aspoň jeden kľúč, ak je tento strom neprázdny, zostáva po tejto operácii zachovaná). Obr. 42 znázorňuje rôzne prípady, ktoré môžu nastať pri vymazávaní kľúča z B-stromu.

1. Ak sa kľúč k nachádza v uzle x, pričom x je list, jednoducho k z uzla x vymažeme.
2. Ak sa kľúč k nachádza v uzle x, pričom x je vnútorný uzol, treba spraviť nasledujúce:

   a. Ak potomok y, ktorý predchádza k v uzle x má aspoň t kľúčov, potom nájdi predchodcu k' uzla k v podstrome s koreňom v uzle y. Rekurzívne vymaž k' a uzol k nahraď k' v x. (Nájdenie a následné vymazanie uzla k' možno uskutočniť na jeden priechod stromu smerom nadol.)
   b. Podobne, ak potomok z, ktorý nasleduje k v uzle x má aspoň t kľúčov, potom nájdi následovníka k' uzla k v podstrome s koreňom v uzle z. Rekurzívne vymaž k' a uzol k nahraď k' v x. (Nájdenie a následné vymazanie uzla k' možno uskutočniť na jeden priechod stromu smerom nadol.)
   c. Inak, ak oba uzly y a z majú iba $t-1$ kľúčov, pridaj kľúč k spolu so všetkými kľúčmi z do y, pričom z x odstránime kľúč k aj ukazovateľ na z. Uzol y teraz obsahuje $2t-1$ kľúčov. Potom uvoľni z a rekurzívne vymaž k z y.

3. Ak sa kľúč k nenachádza vo vnútornom uzle x, nájdi koreň $c_i[x]$ príslušného podstromu, ktorý musí obsahovať k za predpokladu, že sa k v strome nachádza. Ak $c_i[x]$ má iba $t-1$ kľúčov, vykonaj krok 3a alebo 3b, aby bolo zaručené, že zostúpime na uzol obsahujúci aspoň t kľúčov. Potom rekurzívne pokračuj v príslušnom potomkovi uzla x.

   a. Ak $c_i[x]$ obsahuje iba $t-1$ kľúčov, ale má súrodenca, ktorý obsahuje aspoň t kľúčov, do $c_i[x]$ presuň extra kľúč z uzla x. Následne presuň kľúča zo suseda uzla $c_i[x]$ (pravého alebo ľavého) nahor do x. (Treba presunúť aj príslušného potomka.)
   b. Ak $c_i[x]$ a obaja jeho susedia (za predpokladu, že existujú) obsahujú $t-1$ kľúčov, zlúč c_i s jedným z nich, čo umožní presunutie kľúča z x nadol do stredu novovzniknutého uzla.

Keďže prevažná časť kľúčov B-stromov sa nachádza v listoch, môžeme očakávať, že v praxi sú operácie vymazávania najčastejšie použité na odstránenie kľúčov z listov. Procedúra B-Tree-Delete potom pracuje na jeden priechod stromu smerom nadol, bez potreby návratu. Keď vymazávame kľúč nachádzajúci sa vo vnútornom uzle, procedúra prejde strom smerom nadol, ale môže sa potrebovať vrátiť do uzla, z ktorého bol tento kľúč odstránený, aby ho nahradila jeho predchodcom alebo nasledovníkom (prípady 2a, 2b).

Hoci sa táto procedúra zdá byť komplikovaná, potrebuje iba $O(h)$ diskových operácií pre B-strom výšky h, keďže medzi rekurzívnymi vyvolaniami tejto procedúry je vykonaných iba $O(1)$ volaní Disk-Read a Disk-Write. Potrebný výpočtový čas je $O(th) = O(t \log_t n)$.