Algoritmy I.

Jiří Dvorský

Pracovní verze skript

Verze ze dne 28. února 2007

V průběhu semestru by mělo vzniknout nové, přepracované vydání těchto skript (studijní opory). Aktuální verzi najdete vždy na mých webových stránkách, `http://www.cs.vsb.cz/dvorsky/`

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tato učební opora je určena studentům prvního ročníku (prezenční formy), kteří studují na Fakultě elektrotechniky a informatiky. Skriptum mohou také využít posluchači kombinované nebo dálkové formy studia, případně studenti z jiných fakult.

Vzhledem k tomu, že text je určen pro první ročníky, předpokládá se u čtenáře pouze znalost středoškolské matematiky. Matematický aparát nad tento rámec je probrán v úvodu skripta. Dále se předpokládá základní znalost jazyka *C++* . K tomuto jazyku existuje na našem trhu dostatek kvalitních knih.

Čtenáři se seznámí se základními algoritmy a datovými strukturami, které se v různých variantách objevují při řešení většiny problémů a se kterými se programátor ve své praxi setká. Volba programovacího jazyka, pomocí kterého jsou prezentovány ukázkové příklady (zdrojové kódy) je subjektivní záležitostí autorů. Na rozdíl od většiny českých publikací, jsme zvolili jazyk *C++* . Tento programovací jazyk umožňuje jednoduchý zápis většiny programových konstrukcí a není pouze firemním produktem, jako jsou jazyky Delphi, Visual Basic apod..

Programové ukázky jsou až na několik výjimek uvedeny formou kompletního kódu. V případě, že je uveden pouze popis algoritmu lze kompletní programovou realizaci nalézt na Internetu. Tyto ukázky by měly sloužit k experimentům, které by měl čtenář provést v případě, že se chce seznámit s určitým problémem. Při psaní opory jsme se motivovali představou, že pro to abychom se naučili jezdit na kole nestačí přečíst několik odborných knih o tom, jak se na kole jezdí, ale musím si na kolo sednout a vyzkoušet si to.

Opora je rozdělena na několik částí. V úvodní části se čtenář seznámí se základním aparátem, který je dále využíván při popisu jednotlivých algoritmů. Nejdůležitější pojem, se kterým se v této úvodní části pracuje, je složitost algoritmu a asymptotická notace, která je využívána pro popis chování algoritmů. Složitost algoritmů bývá často podceňována a mnozí pro-

gramátoři si neuvědomují, že ne vždy lze čas potřebný pro výpočet výrazně snížit využitím rychlejšího počítače.

Ve další části jsou rozebrány základní třídicí algoritmy a je zde uvedena klasifikace těchto algoritmů. Pro snadnější pochopení těchto algoritmů je tato část vybavena grafickou reprezentací chování jednotlivých algoritmů a jejich flashovými animacemi. Z této části by si čtenář měl odnést poznatek, že volba a efektivita algoritmu může do značné míry záviset na struktuře a rozsahu vstupních dat.

Následující část je věnována vyhledávání. Vedle základních vyhledávacích algoritmů jsou zde uvedeny i poměrně nové výsledky, které zatím nebyly v české literatuře publikovány. Jedná se hlavně o Red-Black stromy, ternární stromy a splay stromy.

V další kapitole se diskutuje hashování. Práce s těmito datovými strukturami vyžaduje větší míru abstrakce než u lineárních datových struktur, a proto zde platí snad ještě ve větší míře než v předcházejících částech, nutnost vlastních experimentů s realizací jednotlivých algoritmů.

Následující kapitola se věnuje problematice vyhledávání řetězců (pattern matching).

Členění jednotlivých kapitol:

– Každá kapitola se skládá z několika podkapitol a začíná obsahem této kapitoly.

– Na začátku podkapitoly je uvedena předpokládaná časová náročnost kapitoly v minutách spolu s ikonou, která na tento údaj upozorňuje:

– Dále je na začátku spolu s navigační ikonou uvedeno, co je cílem této podkapitoly:

– Pak následuje výklad s obrázky, který navíc obsahuje zdrojové kódy programů, na které upozorňuje tato ikona.

– Příklady pro objasnění problematiky jsou označeny touto ikonou.

– Na konci kapitoly se pak nachází cvičení s kontrolními otázkami a úkoly.

– Výklad doplňují flashové animace nebo Java applety, které se spustí kliknutím na příslušný odkaz označený ikonou

Pro spuštění flashových animací je třeba mít na svém PC stažen plug-in viz `www.macromedia.com/downloads/`

– Vlastní text obsahuje hypertextové odkazy na definované pojmy. Čtenář se jednoduchým kliknutím na modře vysvícený pojem dostane na jeho definici, nebo kapitolu, tabulku, ...

– Na začátku a konci každé kapitoly se nachází navigační lišta, pomocí které se čtenář dostane přímo na předcházející nebo následující kapitolu či obsah.

Autoři budou vděčni za připomínky k textu a programům. Celá opora, je přístupná na `www.cs.vsb.cz/~ochodkova/elearn/algor.pdf`

Ostrava, prosinec 2002

Autoři: Daniela Ďuráková, Jiří Dvorský, Eliška Ochodková

Kapitola 2

Základní matematické pojmy

2.1 Označení

V následujícím textu budeme značit

- $\mathbb{N}$ — množina přirozených čísel;
- $\mathbb{Z}$ — množina celých čísel;
- $\mathbb{Q}$ — množina racionálních čísel;
- $\mathbb{R}$ — množina reálných čísel.

Harmonická čísla

V následujícím textu se setkáme s **harmonickými čísly** H_n, které jsou částečnými součty **harmonické řady**

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \cdots$$

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Harmonická čísla je možné vyjádřit vztahem (viz např. [11])

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \cdots$$

kde $\gamma = 0,577215665$ je Eulerova konstanta.

Logaritmy

V dalším textu budeme používat logaritmy. Symbolem log budeme značit, pokud nebude uvedeno jinak, logaritmus o základu 2. Symbol ln bude znamenat, jak bývá obvyklé, přirozený logaritmus (o základu $e = 2,7182818$).

2.2 Množiny, univerzum

Pojem **množiny** budeme chápat intuitivně jako souhrn některých objektů, myšlených jako celek. Množinu budeme považovat za určenou, je-li možno o každém objektu rozhodnout, zda do souhrnu patří či nikoliv, tj. zda je či není jejím *prvkem*. Důvodem k takovému přístupu je skutečnost, že pojem množiny nelze definovat jednoduchým způsobem pomocí pojmů jednodušších. Množiny budeme považovat za podmnožiny jisté větší množiny tzv. **univerza**.

2.2.1 Uspořádané množiny

Množina A se nazývá **uspořádaná množina**, jestliže je na ní definována binární relace $\leq$ taková, že platí následující podmínky:

- $x \leq x$
- je-li $x \leq y$ a $y \leq x$ potom $x = y$
- $x \leq y$ a $y \leq z$ potom $x \leq z$

Jestliže pro každé $x, y \in A$ platí $x \leq y$ nebo $y \leq x$ potom uspořádání $\leq$ nazýváme **lineární**.

2.3 Permutace

Definice 2.1 ***Permutací*** *n-prvkové množiny X rozumíme libovolnou bijekci (vzájemně jednoznačné zobrazení) $f : X \to X$.*

Tuto bijekci budeme většinou zadávat pomocí tabulky se dvěma řádky tak, že každý řádek bude obsahovat všechny prvky množiny X, přičemž prvek $f(x)$ bude umístěn pod prvkem x.

Například je-li $X = \{a, b, c, d\}$ a permutace $f : X \to X$ je zadaná následovně $f(a) = c$, $f(b) = b$, $f(c) = d$, $f(d) = a$, potom

$$f = \begin{pmatrix} a & b & c & d \\ c & b & d & a \end{pmatrix}$$

Počet všech permutací n prvkové množiny je roven číslu $n!$. Rychlost růstu počtu permutací ukazuje tabulka 2.1.

V dalších našich úvahách budeme bez újmy na obecnosti uvažovat permutace množiny $X = \{1, \ldots, n\}$. Označíme S_n množinu všech permutací množiny $\{1, \ldots, n\}$. Libovolnou permutaci $f \in S_n$ budeme obyčejně ztotožňovat s posloupností $\langle a_1, \ldots, a_n \rangle$, kde $f(i) = a_i$. Součinem dvou permutací

n!	n
1	0
1	1
2	2
6	3
24	4
120	5
720	6
5 040	7
40 320	8
362 880	9
3 628 800	10
39 916 800	11
479 001 600	12
6 227 020 000	13
87 178 291 200	14
1 307 674 368 000	15
20 922 789 888 000	16
355 687 428 096 000	17
6 402 373 705 728 000	18
121 645 100 408 832 000	19
2 432 902 004 176 640 000	20

Tabulka 2.1: Růst počtu permutací

$f, g \in S_n$ budeme rozumět permutaci $f \circ g$ definovanou jako superpozice zobrazení f a g. To znamená, že $(f \circ g)(i) = f(g(i))$. Například jsou-li $f, g \in S_7$ takové, že

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 1 & 3 & 6 & 2 & 4 & 5 \end{pmatrix}$$

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 4 & 5 & 7 & 2 & 6 \end{pmatrix}$$

potom

$$fg = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 6 & 2 & 5 & 1 & 4 \end{pmatrix}$$

.

Permutaci

$$id = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}$$

nazveme **identickou permutací**. Je zřejmé, že platí $id \circ f = f \circ id = f$. Snadno se ukáže, že ke každé permutaci $f \in S_n$ existuje permutace $f^{-1} \in S_n$ taková, že $f \circ f^{-1} = f^{-1} \circ f = id$. Tuto permutaci budeme nazývat **inverzní permutací** k permutaci f. Inverzní permutaci k permutaci f dostaneme tak, že vyměníme řádky v zápisu permutace f.

Například pro

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 1 & 3 & 6 & 2 & 4 & 5 \end{pmatrix}$$

dostaneme

$$f^{-1} = \begin{pmatrix} 7 & 1 & 3 & 6 & 2 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 5 & 3 & 6 & 7 & 4 & 1 \end{pmatrix}$$

.

Dále je zřejmé, že pro libovolné permutace $f, g, h \in S_n$ platí následující identity:

- $(f \circ g) \circ h = f \circ (g \circ h)$
- $id \circ f = f \circ id = f$
- $f^{-1} \circ f = f \circ f^{-1} = id$.

To znamená, že S_n je grupa vzhledem k operaci součinu permutací. Tuto grupu nazýváme **symetrickou grupou** stupně n.

Definice 2.2 *Nechť $i_1, \ldots, i_k$ je posloupnost různých prvků množiny $X = \{1, \ldots, n\}$. Permutaci $f \in S_n$ takovou, že $f(i_1) = i_2$, $f(i_2) = i_3, \ldots, f(i_{k-1}) = i_k$, $f(i_k) = i_1$ a $f(i) = i$ pro každé $i \in X - \{i_1, \ldots, i_k\}$, nazýváme **cyklem** délky k a značíme $(i_1, \ldots, i_k)$. Cykly délky dvě se nazývají **transpozice**.*

Věta 2.1 *Každou permutaci lze rozložit na součin transpozic sousedních prvků.*

Definice 2.3 *Nechť f je permutace množiny $X = \{1, \ldots, n\}$. Řekneme, že dvojice různých prvků (i, j) představuje **inverzi permutace** f, jestliže $(i - j)(f(i) - f(j)) < 0$. Permutace se nazývá **sudá** nebo **lichá** podle toho, má-li sudý nebo lichý počet inverzí. Značí-li* $\mathrm{inv}(f)$ *počet inverzí permutace f, pak definujeme* $\mathrm{sgn}(f) = (-1)^{\mathrm{inv}(f)}$. *Tedy je-li f sudá permutace dostaneme* $\mathrm{sgn}(f) = 1$, *kdežto* $\mathrm{sgn}(f) = -1$ *pro lichou permutaci.* $\mathrm{sgn}(f)$ *nazýváme **znaménko permutace**.*

Věta 2.2 *Pro libovolné permutace $f, g \in S_n$ platí* $\mathrm{sgn}(f \circ g) = \mathrm{sgn}(f)\,\mathrm{sgn}(g)$ *a* $\mathrm{sgn}(f) = \mathrm{sgn}(f^{-1})$

Věta 2.3 *Každá transpozice (i, j) je lichá permutace a obecněji znaménko libovolného cyklu délky k se rovná $(-1)^{k-1}$.*

Poznamenejme, že o počtu inverzí můžeme mluvit pouze v případě, že na množině X je dáno lineární uspořádání, ale znaménko permutace závisí pouze na jejím typu.

2.3.1 Zobrazení permutací

V dalším textu budeme používat tři typy zobrazení permutací: bodový, sloupcový a obloukový graf. Ve všech třech případech grafy znázorňují shodné fáze třídění.

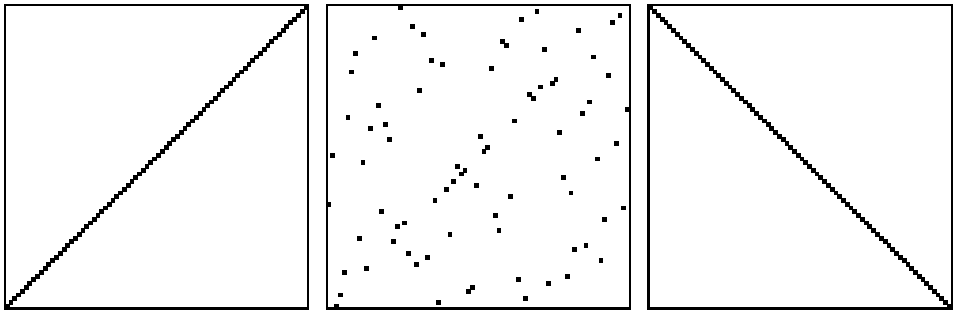
Bodový graf

Toto zobrazení přiřadí permutaci $\langle a_1, \ldots, a_n \rangle$ graf obsahující body se souřadnicemi $\langle 1, a_1 \rangle$, $\langle 2, a_2 \rangle$ až $\langle n, a_n \rangle$. Setříděná posloupnost bude reprezentována grafem obsahujícím body na diagonále (viz obrázky 2.1).

V dalším textu jsou jednotlivé fáze třídění, znázorněné bodovými grafy, rozmístěny na stránce v pořadí, které je uvedeno v tabulce 2.1(a).

Sloupcový graf

Toto zobrazení přiřadí permutaci $\langle a_1, \ldots, a_n \rangle$ graf obsahující sloupce na ose x. Sloupce budou zadány svojí souřadnicí na ose x a výškou, tj. $\langle 1, a_1 \rangle$, $\langle 2, a_2 \rangle$ až $\langle n, a_n \rangle$, kde první složka udává polohu sloupce na ose x a druhá



(a) Identická (b) Náhodná (c) Opačná

Obrázek 2.1: Zobrazení permutací v bodovém grafu

(a) Bodový

1	2	3
4	5	6
7	8	9
10	11	12

(b) Obloukový

1	2	3
4	5	6
7	8	9
10	11	12

Tabulka 2.2: Rozmístění jednotlivých fází třídění v grafech

složka udává výšku sloupce. Šířka sloupců je konstantní. Setříděná posloupnost bude reprezentována grafem obsahujícím sloupce seřazené od nejnižšího k nejvyššímu (viz obrázky 2.2). Jednotlivé fáze jsou řazeny shora dolů.

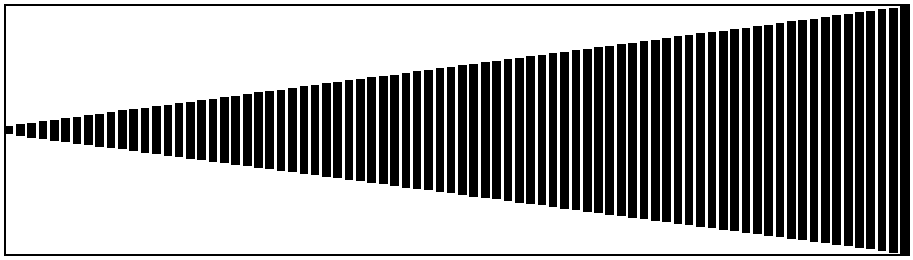
Obloukový graf

Toto zobrazení přiřadí permutaci $\langle a_1, \ldots, a_n \rangle$ graf skládající se z úseček mezi body na dvou kružnicích. Obě kružnice rozdělíme na n dílů. Bod na kružnici vnitřní je určen indexem i, pro $i = 1, \ldots, n$. Index 1 se nachází na kladné části osy x, index n na záporné části osy x. Bod na vnější kružnici je určen hodnotou a_i. Identická permutace se zobrazí jako „vějíř“ (viz obrázek 2.3). V dalším textu jsou jednotlivé fáze třídění, znázorněné obloukovými grafy, rozmístěny na stránce v pořadí, které je uvedeno v tabulce 2.1(b).

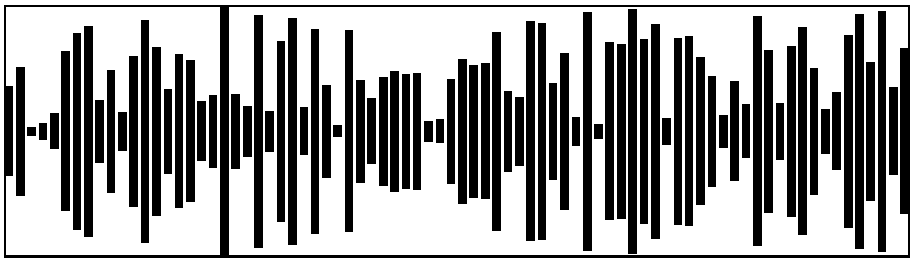
2.4 Algebraické struktury

Definice 2.4 *Nechť G je množina, $G \neq \emptyset$, na které je definována operace $\circ$, s následujícími vlastnostmi: v G existuje prvek n tak, že*

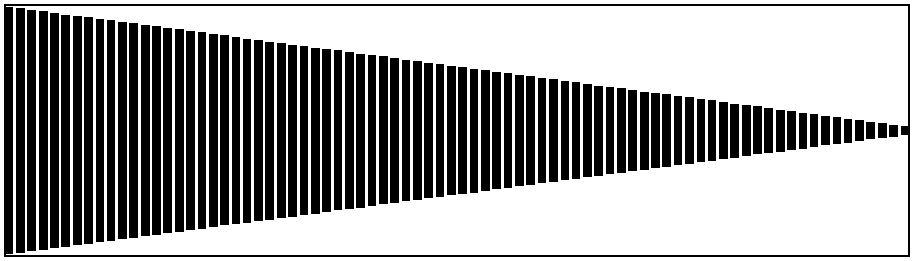
- $\forall g \in G$ je $n \circ g = g$,
- $\forall g \in G\ \exists g^* \in G$ tak, že $g^* \circ g = n$



(a) Identická

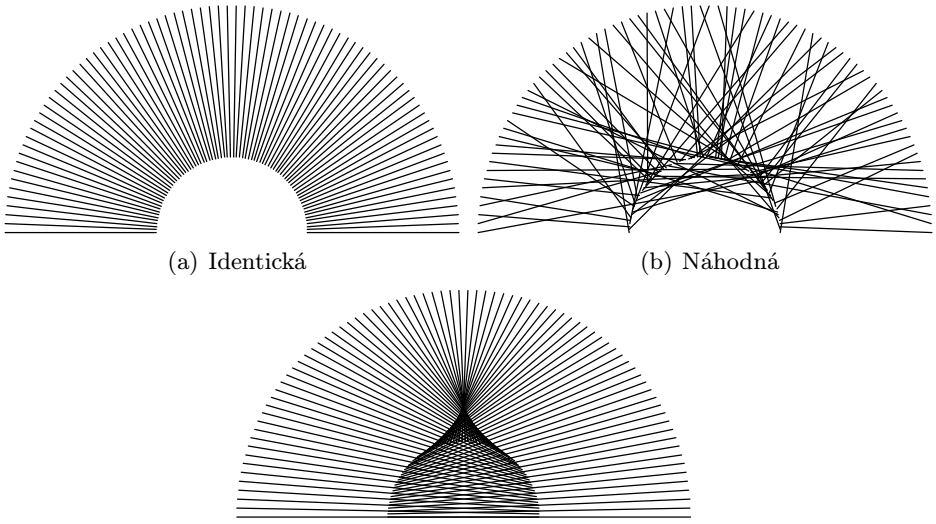


(b) Náhodná



(c) Opačná

Obrázek 2.2: Zobrazení permutací ve sloupcovém grafu



(a) Identická (b) Náhodná

(c) Opačná

Obrázek 2.3: Zobrazení permutací v obloukovém grafu

- $\forall g, f, h \in G$ platí $f \circ (g \circ h) = (f \circ g) \circ h$

Pak říkáme, že $(G, \circ)$ *je* ***grupa*** *a množina* G *je* ***nosičem*** $(G, \circ)$

V grupě platí axiomy:

(G_0) $\forall a, b \in G : \ a \circ b \in G$,

(G_1) $\forall a, b, c \in G$: je-li $a = b \Rightarrow (a \circ c = b \circ c \wedge c \circ a \ = c \circ b)$

(G_2) $\forall a, b, c \in G : \ a \circ (b \circ c) = (a \circ b) \circ c$ (asociativita)

(G_3) $\exists n \in G$ tak, že $\forall a \ \in G : \ n \circ a \ = a$ (neutrální prvek)

(G_4) $\forall g \in G \ \exists g^* \in G$ tak, že $g^* \circ g = n$ (inverzní prvek)

(G_5) $\forall a, b \in G : \ a \circ b = b \circ a$.
Pokud navíc platí tento axiom – komutativita – nazývá se **grupa komutativní**.

Definice 2.5 *Jestliže ve struktuře* $(G, \circ)$ *platí axiomy* (G_0), (G_1) *a* (G_2), *pak se* $(G, \circ)$ *nazývá* ***pologrupa****.*

Příklad 2.1
Množiny všech celých, racionálních, reálných a komplexních čísel s operací sčítání tj. $(\mathbb{Z}, +)$, $(\mathbb{Q}, +)$, $(\mathbb{R}, +)$ a $(\mathbb{C}, +)$ jsou komutativní grupy. Množina všech čtvercových regulárních matic řádu dva tvoří nekomutativní grupu

vzhledem k násobení matic. Množina všech permutací n prvků tvoří komutativní grupu vzhledem k součinu permutací. Naproti tomu struktura $(\mathbb{Z}, \cdot)$ není grupa.

Definice 2.6 ***Okruhem*** *nazýváme uspořádanou trojici $(A, +, \cdot)$, kde A je neprázdná množina $a + a \cdot$ jsou dvě binární operace a přitom platí:*

1. *$(A, +)$ je komutativní grupa*
2. *$(A, \cdot)$ je pologrupa.*
3. *pro každou trojici prvků $a, b, c \in A$ platí*
   $a \cdot (b + c) = a \cdot b + a \cdot c$ (levý distributivní zákon)
   $(b + c) \cdot a = b \cdot a + b \cdot c$ (pravý distributivní zákon)

Grupu $(A, +)$ nazýváme **aditivní grupou** okruhu A. Její neutrální prvek nazýváme nulový prvek okruhu A a značíme jej o. Pologrupu $(A, \cdot)$ nazýváme **multiplikativní pologrupou** okruhu A.

Definice 2.7 *Okruh $(A, +, \cdot)$ se nazývá* ***komutativní****, jestliže $\forall a, b \in A$ platí $a \cdot b = b \cdot a$ (komutativní zákon).*

Definice 2.8 *Okruh $(A, +, \cdot)$ se nazývá* ***okruh s jednotkovým prvkem*** *jestliže existuje prvek $e \in A, e \neq o$ takový, že $\forall a \in A$ platí $a \cdot e = e \cdot a = a$ (zákon jednotkového prvku).*

Definice 2.9 *Nechť $(A, +, \cdot)$ je okruh. Prvky $a, b \in A$ pro než platí $a \cdot b = o$, přičemž $a \neq o$ a současně $b \neq o$ se nazývají* ***dělitelé nuly****.*

Příklad 2.2
V okruhu $(\mathbb{Z}_6, +, \cdot)$ existují dělitelé nuly. Například $\bar{2} \cdot \bar{3} = \bar{0}$, přičemž $\bar{2} \neq \bar{0}$ a $\bar{3} \neq \bar{0}$.
Ale okruh $(\mathbb{Z}_p, +, \cdot)$, kde p je prvočíslo dělitele nuly neobsahuje.

Definice 2.10 *Komutativní okruh s jednotkovým prvkem bez dělitelů nuly se nazývá* ***obor integrity****.*

Definice 2.11 ***Tělesem*** *nazýváme okruh $(T, +, \cdot)$ s jednotkovým prvkem e, ve kterém pro každý prvek $a \in T, a \neq o$ existuje prvek $a^{-1} \in T$ takový, že $a \cdot a^{-1} = a^{-1} \cdot a = e$ (zákon inverzních prvků). Prvek a^{-1} se nazývá* ***inverzní prvek****.*

Věta 2.4 *Každé komutativní těleso je obor integrity*

Věta 2.5 *Každý konečný obor integrity je komutativní těleso.*

Zájemce o další algebraické struktury a jejich vlastnosti odkazujeme na knihu [5].

2.5 Grafy

Definice 2.12 ***Neorientovaným grafem*** *nazýváme dvojici $G = (V, E)$, kde V je množina uzlů, E je množina jedno- nebo dvouprvkových podmnožin V. Prvky množiny E se nazývají* ***hrany*** *grafu a prvky množiny V se nazývají* ***uzly****.*

Mějme hranu $e \in E$, kde $e = \{u, v\}$. Uzlům u a v říkáme **krajní uzly hrany** e. Říkáme také, že jsou **incidentní** (nebo že *incidují*) s hranou e. O hraně e pak říkáme, že je **incidentní** s těmito uzly nebo také že *spojuje* tyto uzly.

Definice 2.13 *Hranu spojující uzel se sebou samým nazýváme* ***smyčkou****.*

Obecně může být množina uzlů grafu nekonečná, my však budeme uvažovat pouze **konečné** grafy, tedy grafy s konečnou množinou uzlů V. Vzhledem k tomu, že jiné než neorientované grafy nebudeme definovat, budeme označení neorientovaný vynechávat.

Definice 2.14 ***Stupeň uzlu*** *je počet hran s uzlem incidentních, tj.*

$$s(v) = |\{e \in E \mid v \in e\}|.$$

Věta 2.6 *Součet stupňů uzlů libovolného grafu $G = (V, E)$ je roven dvojnásobku počtu jeho hran.*

$$\sum_{v \in V} s(v) = 2|E|$$

Důkaz. Zřejmý (v sumě se každá hrana počítá dvakrát). ∎

Definice 2.15 *Graf $G' = (V', E')$ se nazývá* ***podgrafem*** *grafu $G = (V, E)$, je-li $V' \subset V$ a zároveň $E' \subset E$.*

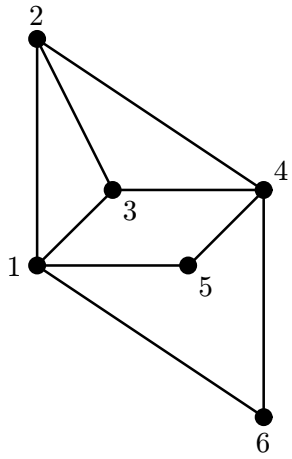
Definice 2.16 *Posloupnost navazujících uzlů a hran $v_1, e_1, v_2, \ldots, v_n, e_n, v_{n+1}$, kde $e_i = \{v_i, v_{i+1}\}$ pro $1 \leq i \leq n$ nazýváme (neorientovaným)* ***sledem****.*

Definice 2.17 *Sled, v němž se neopakuje žádný uzel nazýváme* ***cestou****. Tedy $v_i \neq v_j, \forall\ 1 \leq i \leq j \leq n$. Číslo n pak nazýváme* ***délkou cesty****.*

Z faktu, že se v cestě neopakují uzly, vyplývá, že se v ní neopakují ani hrany. Každá cesta je tedy zároveň i sledem.

Definice 2.18 *Sled, který má alespoň jednu hranu a jehož počáteční a koncový uzel splývají, nazýváme* ***uzavřeným sledem****.*

Definice 2.19 ***Uzavřená cesta*** *je uzavřený sled, v němž se neopakují uzly ani hrany. Uzavřená cesta se nazývá také* ***kružnice****.*



Obrázek 2.4: Graf

V definici kružnice jsme museli zakázat kromě opakování uzlů i opakování hran proto, aby posloupnost v_1, e_1, v_2, e_1, v_1 nemohla být považována za kružnici.

Definice 2.20 *Graf se nazývá* ***acyklický****, jestliže neobsahuje kružnici.*

Definice 2.21 *Graf se nazývá* ***souvislý****, jestliže mezi každými dvěma uzly existuje cesta.*

Definice 2.22 ***Komponentou souvislosti*** *grafu G nazýváme každý podgraf H grafu G, který je souvislý a je maximální s takovou vlastností.*

Věta 2.7 *Nechť $G = (V, E)$ je souvislý graf. Pak platí $|E| \geq |V| - 1$.*

Důkaz. Zřejmý. ∎

Příklad 2.3
Na obrázku 2.4 je znázorněn graf $G = (V, E)$, kde $V = \{1, 2, 3, 4, 5, 6\}$ a $E = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{5, 6\}\}$. Uzly $\{1, 2, 3, 4\}$ spolu s hranami $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$ tvoří cestu. Hrany $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{4, 5\}$, $\{5, 1\}$ pak tvoří kružnici.

Graf je možné zadat grafickou formou (obrázkem) nebo maticí (tabulkou), a to hned několika způsoby. Zmíníme se pouze o **matici sousednosti**. Matice sousednosti pro graf z obrázku 2.4 má následující tvar:

	1	2	3	4	5	6
1	0	1	1	1	1	1
2	1	0	1	1	0	0
3	1	1	0	1	0	0
4	1	1	1	0	1	1
5	1	0	0	1	0	0
6	1	0	0	1	0	0

Matici sousednosti grafu G značíme A_G. Každá symetrická matice, jejíž prvky jsou pouze 0 a 1, s nulovou hlavní diagonálou je maticí sousednosti nějakého neorientovaného grafu.

S grafy a jejich aplikacemi je možné se podrobně seznámit například v knize [8].

Kapitola 3

Algoritmus, jeho vlastnosti

Název „algoritmus“ pochází ze začátku devátého století z Arábie. V letech 800 až 825 napsal arabský matematik Muhammad ibn Músá al Chwárizmí dvě knihy, z nichž jedna se v latinském překladu jmenovala „Algoritmi dicit“, česky „Tak praví al Chwárizmí“. Byla to kniha postupů pro počítání s čísly.

Algoritmu můžeme rozumět jako předpisu pro řešení „nějakého“ problému. Jako příklad lze uvést předpis pro konstrukci trojúhelníka pomocí kružítka a pravítka ze tří daných prvků. Pokud rozebereme řešení takové úlohy do důsledku, musí obsahovat tři věci:

1. hodnoty vstupních dat (tři prvky trojúhelníka),
2. předpis pro řešení,
3. požadovaný výsledek, tj. výstupní data (výsledný trojúhelník).

Na tomto místě je důležité upozornit na fakt, že ne pro každé tři prvky existuje konstrukce trojúhelníka pomocí kružítka a pravítka. Zájemce o tuto problematiku lze odkázat na knihu [24].

Pro zpřesnění pojmu algoritmus tedy dodejme: **Algoritmus** je předpis, který se skládá z *kroků* a který zabezpečí, že na základě vstupních dat jsou poskytnuta požadovaná data výstupní. Navíc každý algoritmus musí mít následující vlastnosti:

[**Konečnost**.] Požadovaný výsledek musí být poskytnut v „rozumném“ čase (pokud by výpočet trval na nejrychlejším počítači např. jeden milion let, těžko bychom mohli hovořit o algoritmu řešení, nemluvě o výpočtu, který by neskončil vůbec). Za rozumný lze považovat čas, kdy nám výsledek výpočtu k něčemu bude.

[**Hromadnost**.] Vstupní data nejsou v popisu algoritmu reprezentována konkrétními hodnotami, ale spíše množinami, ze kterých lze data vybrat (např. při řešení trojúhelníka mohou být velikosti stran desetinná

čísla). Při popisu algoritmu v programovacím jazyce se to projeví tím, že vstupy do algoritmu jsou označeny symbolickými jmény.

[**Jednoznačnost**.] Každý předpis je složen z kroků, které na sebe navazují. Každý krok můžeme charakterizovat jako přechod z jednoho stavu algoritmu do jiného, přičemž každý stav je určen zpracovávanými daty. Tím, jak data v jednotlivých stavech algoritmu vypadají, musí být jednoznačně určeno, který krok následuje (např: V řešení trojúhelníka může nastat situace, kdy vychází na základě vstupních dat jedno nebo dvě řešení. Situace je tedy nejednoznačná, řešení musí být jednoznačné, tzn. v předpisu se s touto možností musí počítat a musí v něm být návod, jak ji řešit.).

[**Opakovatelnost**.] Při použití stejných vstupních údajů musí algoritmus dospět vždy k témuž výsledku.

[**Rezultativnost**.] Algoritmus vede ke správnému výsledku.

Algoritmus můžeme chápat i jako „mlýnek na data“. Nasypeme-li do něj správná data a zameleme, obdržíme požadovaný výsledek. V tomto okamžiku si uvědomme, že kvalita mlýnku může být různá, nás prozatím budou zajímat především vstupní ingredience a správný výstup. Pro začátek je mnohem důležitější vědět to, co chceme, než to, jak toho dosáhneme.

Algoritmus můžeme chápat jako jistý návod pro konstrukci řešení daného problému. V matematice se můžeme setkat i s nekonstruktivními řešeními. Na závěr tohoto odstavce si uveďme příklad nekonstruktivního řešení problému.

Naším úkolem je najít dvě iracionální čísla x a y taková aby platilo, že x^y je číslo racionální. Zvolíme $x = \sqrt[2]{2}$ a $y = \sqrt[2]{2}$ je-li x^y číslo racionální jsme hotovi, není-li x^y číslo racionální zvolíme $x = \sqrt[2]{2}^{\sqrt[2]{2}}$ a $y = \sqrt[2]{2}$. Potom dostaneme

$$x^y = \sqrt[2]{2}^{\sqrt[2]{2}^{\sqrt[2]{2}}} = \sqrt[2]{2}^{\sqrt[2]{2}\sqrt[2]{2}} = \sqrt[2]{2}^2 = 2$$

Je zřejmé, že řešením je buď dvojice čísel $x = \sqrt[2]{2}$ a $y = \sqrt[2]{2}$ nebo dvojice čísel $x = \sqrt[2]{2}^{\sqrt[2]{2}}$ a $y = \sqrt[2]{2}$, přičemž nejsme z uvedeného řešení schopni říci která dvojice je vlastně řešením našeho problému.[1]

3.1 Programování a programovací jazyk

Programováním budeme rozumět následující činnosti (které ovšem nebudeme navzájem oddělovat):

[1] Problém spočívá v tom, že není jasné zda je $\sqrt[2]{2}^{\sqrt[2]{2}}$ číslo iracionální nebo ne.

1. Správné pochopení zadání úlohy, které vyústí v přesný popis možných situací a návrh vstupních a výstupních dat.

2. Sestavení algoritmu řešení.

3. Detekování úseků, které budou řešeny samostatně.

4. Zápis zdrojového textu úlohy v programovacím jazyce odladění.

5. Přemýšlení nad hotovým dílem, vylepšování (ovšem bez změn v návrhu vstupu a výstupu).

Programovací jazyk neslouží pouze pro zápis našich požadavků pro počítač. Je určen také jako prostředek pro vyjádření našich představ o tom, jak má probíhat výpočet (a také k tomu, aby tyto představy byl schopen vnímat jiný člověk). Prakticky to znamená, že ze zdrojového textu programu zapsaného v „nějakém“ programovacím jazyce by mělo být zřetelně vidět, jak se kombinací jednoduchých myšlenek dosáhlo řešení komplexnějšího problému (**program** je algoritmus zapsaný v některém programovacím jazyce).

K tomu každý vyšší programovací jazyk poskytuje uživateli tři nástroje (viz A.1):

1. Primitivní výrazy, tj. data (čísla, znaky, apod.) a procedury (sčítání, násobení, logické operátory apod.).

2. Mechanismus pro sestavování složitějších výrazů z jednodušších.

3. Mechanismus pro pojmenování složitějších výrazů a tím zprostředkování možnosti pracovat s nimi stejně jako s primitivními výrazy (definování proměnných a nových procedur).

Data reprezentují objekty se kterými pracujeme, procedury (viz A.1) reprezentují pravidla pro manipulaci s daty (procedury jsou tedy algoritmy).

Podstatnou vlastností programovacího jazyka je asociování jmen a hodnot. Např. jméno `486` je svázáno s hodnotou čísla 486 v desítkové soustavě, jméno + je svázáno s procedurou pro sčítání (hodnotou jména + je procedura). To znamená, že uživatel při psaní zdrojového textu pracuje ve výrazech se jmény, interpret (překladač) jazyka text zpracuje a počítá s hodnotami.

3.2 Složitost

Pojem složitosti, kterým se budeme zabývat, je blízký jeho významu v běžném jazyce. Dalo by se říci, že zkoumáme matematizaci tohoto pojmu. Na rozdíl od některých jiných pojmů, které byly matematizovány, neexistuje

jen jeden matematický model složitosti, ale celá řada možných definic vystihujících různé aspekty. Intuitivní pojem složitosti je svázán s představou množství informace obsažené v daném jevu. Není však zřejmé, jakým způsobem by se měla informace s jevem nebo objektem spojovat. Různé způsoby spojování dávají různé míry složitosti, tím je dána nejednoznačnost tohoto pojmu.

Při praktické realizaci každé výpočetní metody jsme omezeni prostředky, které máme k dispozici – čas, paměť, počet registrů atd. Důležitým parametrem každé výpočetní metody je její **složitost**, kterou můžeme chápat jako vztah dané metody k daným prostředkům. Takovou výpočetní metodou je například třídění. Ačkoliv je zvolena adekvátní metoda třídění a metoda je odladěna na vzorových datech, pořád je ještě možné, že pro určitá konkrétní data se výpočet protáhne na hranici únosnosti, nebo dokonce do té míry, že se výsledků nedočkáme. Podobně se může stát, že výpočet ztroskotá na přeplnění operační paměti počítače. Zkušený programátor proto bere v úvahu, že jeho program bude pracovat s omezenými prostředky.

Složitost dělíme na složitost časovou (časovou složitostí rozumíme funkci, která každé množině vstupních dat přiřazuje počet operací vykonaných při výpočtu podle daného algoritmu.) a složitost paměťovou (**paměťovou složitost** definujeme jako závislost paměťových nároků algoritmu na vstupních datech). Časová složitost výpočetních metod zpravidla vzbuzuje menší respekt než složitost prostorová. Ne každý narazil ve své praxi na problémy s vysokou časovou složitostí a pokud ano, čelil jim možná poukazem na pomalý počítač v dobré víře, že použití několikanásobně rychlejšího počítače by jeho potíže vyřešilo. A jelikož takový počítač neměl k dispozici, snažil se zrychlit dosavadní program drobnými úpravami, přepsáním některých částí do assembleru apod.

Takový postup je někdy úspěšný, jindy je již předem odsouzen k neúspěchu a to, co následuje, je jen zbytečné trápení plynoucí z neznalosti základních vlastností výpočetní složitosti algoritmů.

Libovolnému programu P přiřadíme funkci t, která udává jeho **časovou složitost**. To znamená, jestliže program P zpracuje data D a vydá výsledek $P(D)$, udává $t(D)$ počet elementárních operací, které program P nad daty D vykoná. Tyto operace můžeme ztotožnit s časovými jednotkami, takže na $t(D)$ můžeme pohlížet jako na čas, který program P potřebuje ke zpracování dat D.

Časovou složitost t je často možné přirozeným způsobem stanovit nejen v závislosti na konkrétních datech D, ale už na základě znalosti jejich rozsahu $|D|$ (stanoveném například v bitech). Potom $t(n) = m$ znamená, že program P na data D rozsahu $n = |D|$ spotřebuje m časových jednotek.

Předpokládejme nyní, že pět různých programů P_1, P_2, P_3, P_4, P_5 má časovou složitost danou funkcemi

$$t_1(n) \quad = \quad n$$

$$\begin{aligned} t_2(n) &= n \log n \\ t_3(n) &= n^2 \\ t_4(n) &= n^3 \\ t_5(n) &= 2^n \end{aligned}$$

Předpokládejme dále, že elementární operace vykonávané programy trvají $1ms$ a spočítejme, jak rozsáhlá data mohou jednotlivé programy zpracovat za sekundu, za minutu a za hodinu.

program	složitost	1s	1min	1hod
$t_1(n)$	n	1000	$6 \cdot 10^4$	$3,6 \cdot 10^6$
$t_2(n)$	$n \log n$	140	4895	$2,0 \cdot 10^5$
$t_3(n)$	n^2	31	244	1897
$t_4(n)$	n^3	10	39	153
$t_5(n)$	2^n	9	15	21

I zběžný pohled na tabulku nás přesvědčí, že u programů, jejichž složitost je dána rychle rostoucí funkcí, se při prodlužování doby výpočtu jen pomaleji dosahuje zpracování dat většího rozsahu.

U výpočetních metod s lineární složitostí se například 10-násobné zrychlení (nebo zvětšení doby) výpočtu projeví 10-násobným zvětšením rozsahu zpracovávaných dat, u metod s kvadratickou složitostí se toto zrychlení projeví zhruba 3-násobným zvětšením rozsahu zpracovávaných dat atd. až u programu P s exponenciální složitostí 2^n se 10-násobné zrychlení projeví zvětšením rozsahu dat zhruba o $3,3$. Dosažení rozsahu dat například $n = 50$ u programu P_5 zrychlováním (nebo prodlužováním) výpočtu už vůbec nepřichází v úvahu.

Jedinou schůdnou cestou je nalezení algoritmu s menší časovou složitostí. Jestliže se například podaří nahradit program složitosti 2^n programem složitosti n^3, otvírá se tím cesta ke zvládnutí většího rozsahu dat v míře, kterou nelze zrychlováním výpočtů suplovat.

Úvodní poznámky zakončíme stručnou zmínkou o taxonomii časové složitosti výpočetních problémů. Základním kritériem pro určování časové složitosti výpočetních problémů je jejich algoritmická zvládnutelnost. Předně je si třeba uvědomit, že existují algoritmicky neřešitelné problémy, pro které nemá smysl zkoušet algoritmy konstruovat. Příkladem je problém sestrojení algoritmu, který by o každém algoritmu měl rozhodnout, zda jeho činnost skončí po konečném počtu kroků či nikoliv. Dále existují problémy, pro které byl nalezen exponenciální dolní odhad časové složitosti. Je to například problém rozhodnutí, zda dva regulární výrazy (ve kterých můžeme navíc jako operaci používat druhou mocninu) jsou ekvivalentní.

Definice 3.1 *Nechť f je libovolná funkce v oboru přirozených čísel. Říkáme, že problém T má časovou složitost nejvýše f, jestliže existuje algoritmus A*

pro T takový, že složitost všech ostatních algoritmů je menší nebo rovna složitosti algoritmu A. Funkce f se nazývá horním odhadem časové složitosti problému T.

Definice 3.2 *Říkáme, že problém T má časovou složitost alespoň f, jestliže existuje program P pro T takový, že $t_P(n) \geq f(n)$ pro všechna n. V tomto případě je f* ***dolním odhadem časové složitosti*** *problému T.*

Nalézt horní odhad f složitosti problému T tedy znamená najít nějaký program P pro T se složitostí nejvýše f. Stanovit dolní odhad g složitosti problému T je svou povahou úkol mnohem těžší, neboť je třeba ukázat, že všechny programy P pro T mají složitost aspoň g.

3.3 Složitostní míry

Podaří-li se vyjádřit časovou či paměťovou složitost algoritmu jako funkci rozsahu vstupních dat, pak pro hodnocení efektivity algoritmu je důležité zejména to, jak roste složitost v závislosti na růstu rozsahu vstupních dat. Jinak řečeno, zajímá nás limitní chování složitosti tzv. **asymptotická složitost**.

Tedy: při zkoumání složitosti problémů jsme často nuceni spokojit se s přesností až na multiplikativní konstantu. To se v příslušném žargonu zpravidla vyjadřuje tím, že mluvíme o složitosti „řádově“ f. Formálně se pro asymptotické chování funkcí zavádějí následující značení (notace):

Θ-Značení

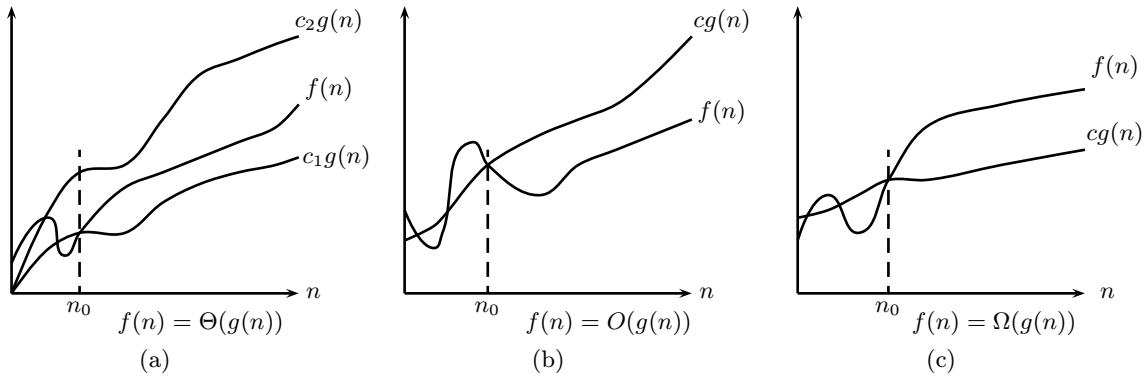
Pro každou funkci $g(n)$, označíme zápisem $\Theta(g(n))$ množinu funkcí $\Theta(g(n)) = \{f(n)$: takových, že existují kladné konstanty c_1, c_2 a n_0 tak, že $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ pro všechna $n \geq n_0\}$.

Funkce $f(n)$ patří do množiny $\Theta(g(n))$ jestliže existují kladné konstanty c_1 a c_2 takové, že tato funkce nabývá hodnot mezi $c_1 g(n)$ a $c_2 g(n)$. Skutečnost, že $f(n)$ splňuje předcházející vlastnost zapisujeme „$f(n) = \Theta(g(n))$“. Tento zápis znamená $f(n) \in \Theta(g(n))$.

O-Značení

Θ-značení omezuje asymptoticky funkci zdola a shora. Jestliže budeme chtít omezit funkci jen shora použijeme O-značení.

Pro každou funkci $g(n)$, označíme zápisem $O(g(n))$ množinu funkcí $O(g(n)) = \{f(n)$: takových, že existují kladné konstanty c a n_0 tak, že $0 \leq f(n) \leq cg(n)$ pro všechna $n \geq n_0\}$.



Obrázek 3.1: Grafické vyjádření Θ, O a Ω značení

n_0 představuje nejmenší možnou hodnotu vyhovující kladeným požadavkům; každá vyšší hodnota samozřejmě také vyhovuje. **(a)** Θ notace ohraničuje funkci mezi dva konstantní faktory. Píšeme, že $f(n) = \Theta(g(n))$, jestliže existují kladné konstanty n_0, c_1 a c_2 takové, že počínaje n_0, hodnoty funkce $f(n)$ vždy leží mezi $c_1g(n)$ a $c_1g(n)$ včetně. **(b)** O značení shora ohraničuje funkci nějakým konstantním faktorem. Píšeme, že $f(n) = O(g(n))$, jestliže existují kladné konstanty n_0 a c takové, že počínaje n_0, hodnoty funkce $f(n)$ jsou vždy menší nebo rovny hodnotě $cg(n)$. **(c)** Ω notace určuje dolní hranici funkce $f(n)$. Píšeme, že $f(n) = \Omega(g(n))$, jestliže existují kladné konstanty n_0 a c takové, že počínaje n_0, hodnoty funkce $f(n)$ jsou vždy větší nebo rovny hodnotě $cg(n)$.

Ω-Značení

Pro každou funkci $g(n)$, označíme zápisem $\Omega(g(n))$ množinu funkcí $\Omega(g(n)) = \{f(n)$: takových, že existují kladné konstanty c a n_0 tak, že $0 \leq cg(n) \leq f(n)$ pro všechna $n \geq n_0\}$.

o-Značení

Pro každou funkci $g(n)$, označíme zápisem $o(g(n))$ množinu funkcí $o(g(n)) = \{f(n)$: takových, že pro každou kladnou konstantu $c0$ existuje konstanta n_0 taková, že $0 \leq f(n) < cg(n)$ pro všechna $n \geq n_0\}$.

ω-Značení

Pro každou funkci $g(n)$, označíme zápisem $\omega(g(n))$ množinu funkcí $\omega(g(n)) = \{f(n)$: takových, že pro každou kladnou konstantu $c0$ existuje konstanta n_0 taková, že $0 \leq cg(n) < f(n)$ pro všechna $n \geq n_0\}$.

Mnoho vlastností relací mezi reálnými čísly se velmi dobře přenáší na asymptotické porovnání funkcí.

Tranzitivita

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

Reflexivita

$f(n) = \Theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

Symetrie

$f(n) = \Theta(g(n))$ tehdy a jen tehdy, když $g(n) = \Theta(f(n))$

Transponovaná symetrie

$$\begin{array}{lll} f(n) = O(g(n)) & \text{tehdy a jen tehdy, když} & g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) & \text{tehdy a jen tehdy, když} & g(n) = \omega(f(n)) \end{array}$$

Z předcházejících vlastností je zřejmé, že asymptotické značení pro porovnávání funkcí se velmi podobá porovnávání reálných čísel. Tato podobnost se dá vyjádřit následovně:

$$\begin{array}{lll} f(n) = \Theta(g(n)) & \approx & x = y \\ f(n) = O(g(n)) & \approx & x \leq y \\ f(n) = \Omega(g(n)) & \approx & x \geq y \\ f(n) = o(g(n)) & \approx & x < y \\ f(n) = \omega(g(n)) & \approx & xy \end{array}$$

Trichotomie

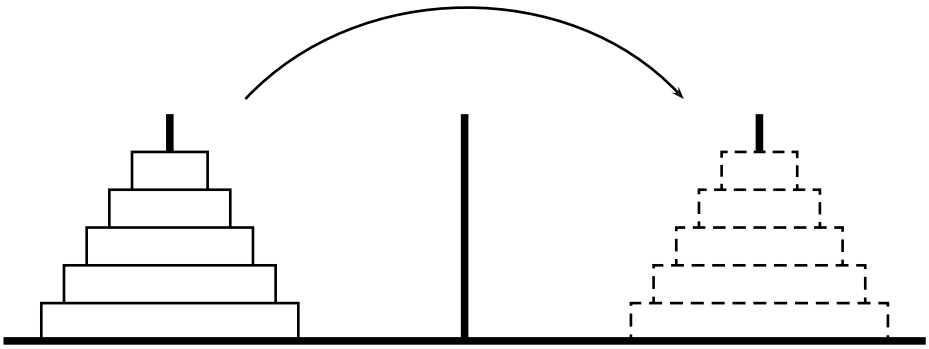
Pro každé dvě reálná čísla x a y platí právě jeden z následujících vztahů: $x < y$, $x = y$ nebo xy. To znamená, že každé dvě reálná čísla jsou porovnatelná, ale tato vlastnost neplatí pro asymptotické porovnávání funkcí. To znamená, že existují funkce $f(n)$ a $g(n)$ takové, že neplatí ani jeden ze vztahů $f(n) = O(g(n))$ nebo $f(n) = \Omega(g(n))$. Například funkce n a $n^{1+\sin(n)}$ nemůžeme porovnat pomocí asymptotické notace.

Příklad

Na závěr této kapitoly si ukážeme kompletní příklad určení složitosti problému. Tento příklad je použit z V. Snášel, M. Kudělka [20].

Ve věži v Hanoji v brahmánském chrámu byly při stvoření světa postaveny na zlaté desce tři diamantové jehly. Na jedné z nich bylo navlečeno 64

různě velkých kotoučů tak, že vždy menší ležel na větším. Brahmánští kněží každou sekundu vezmou jeden kotouč a přemístí ho na jinou jehlu, přitom však nikdy nesmí položit větší kotouč na menší. V okamžiku, kdy všech 64 kotoučů bude ležet na jiné diamantové jehle než na začátku, nastane prý konec světa.



Tuto legendu si vymyslel francouzský matematik Edouard Lucas v roce 1833 a cílem bylo nalézt co nejmenší počet přesunů pro 5 kotoučů.

Mějme tři tyče označené A B C a na první z nich navlečeno n disků, které se zdola nahoru zmenšují. Hledejme nejmenší počet přesunů, kterými přeneseme všechny disky na tyč B. Každým přesunem rozumíme přenesení jednoho disku na jinou tyč, nikdy přitom nesmíme položit větší disk na menší.

Označme nejmenší počet přesunů n disků jako funkci jedné proměnné $T(n)$, pro $n \in \mathbb{N}$.

Začněme úlohu řešit od jednoduchých případů. Zřejmě platí

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 3. \end{aligned}$$

Popišme nyní algoritmus pro realizaci přenosu n disků, $n \in \mathbb{N}$:

1. Přeneseme $n-1$ disků na volnou tyč C.
2. Přesuneme největší (spodní) disk na tyč B.
3. Přeneseme $n-1$ disků z tyče C na B.

Je možné, že existuje více algoritmů, řešících daný problém, proto si uvedený algoritmus označme **P**.

Algoritmus **P** redukuje úlohu s n disky na úlohu s $n-1$ disky. Zopakujeme-li tento postup $n-1$ krát, budeme podle popisu algoritmu pouze přesouvat jeden disk.

Programová realizace algoritmu P

```
#include <stdio.h>

int count;

void MoveDisk(int x, int y)
{
    printf ("Tah %3d: Presun horni disk z tyce %d na tyc %d\n", count++, x, y)
        ;
}

void Hanoi(const int n, const int a, const int b, const int c)
{
    if (n 0)
    {
        Hanoi(n−1,a,c,b);
        MoveDisk(a,b);
        Hanoi(n−1,c,b,a);
    }
}

void main()
{
    count = 0;
    Hanoi(4, 1, 2, 3);
    printf ("Celkem bylo potreba: %d tahu\n", count);
}
```

Z popisu algoritmu **P** plyne:

$$\begin{aligned} T_P(1) &= 1, \\ T_P(n) &= 2 \cdot T_P(n-1) + 1, \end{aligned} \tag{3.1}$$

kde $T_P(n)$ je počet přesunů n disků užitím algoritmu **P**.

Je algoritmus P nejlepší?

Zatím víme, že užitím algoritmu **P** je problém řešitelný, nicméně z našich předchozích úvah nevyplývá, zda je tento algoritmus optimální, tedy je-li $T(n) = T_P(n)$. Dokažme tuto rovnost.

Zřejmě platí

$$T(n) \leq T_P(n), \ pro\ n \in \mathbb{N}. \tag{3.2}$$

Musíme dokázat nerovnost

$$T(n) \geq T_P(n), \ pro\ n \in \mathbb{N}. \tag{3.3}$$

Matematická indukce

1. Zřejmě platí $T(1) \geq T_P(1)$.

2. Předpokládejme, že (3.3) platí pro $m \in \mathbb{N}$.

3. Dokažme, že (3.3) platí pro $m+1 \in \mathbb{N}$.

   Předpokládejme, že existuje algoritmus **X**, kterým přeneseme $m+1$ disků pomocí menšího počtu přesunů, než je $T_P(m+1)$, tedy

   $$T_X(m+1) < T_P(m+1). \tag{3.4}$$

   Tento algoritmus musí přenést největší (spodní) disk z tyče A na B (1 přesun). V tomto okamžiku musí být m disků na tyči C. Nejmenší počet přesunů pro přenos m disků je podle indukčního předpokladu větší nebo roven $T_P(m)$. Po přenesení největšího disku musíme přenést m disků z tyče C na B. K tomu podle předpokladu potřebujeme opět nejméně $T_P(m)$ přesunů.

   Dostáváme:

   $$T_X(m+1) \geq 2 \cdot T_P(m) + 1 \overset{\text{podle } (3.1)}{=} T_P(m+1),$$

   což je spor s předpokladem (3.4).

   Pro libovolný algoritmus X musí platit

   $$\begin{aligned} T_X(n) &\geq T_P(n), \\ \text{tedy i } T(n) &\geq T_P(n). \end{aligned} \tag{3.5}$$

Z nerovností (3.2) a (3.5) pak plyne

$$T(n) = T_P(n),$$

proto algoritmus **P** je optimální a dostáváme:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 \cdot T(n-1) + 1 \text{ pro } n \in \mathbb{N},\ n > 1. \end{aligned} \tag{3.6}$$

Doplňme tedy funkci T do našeho programu:

```
int T(int n)
{
   if (n == 1)
      return 1;
   else
      return 2 * T(n-1) + 1;
}
```

Použijme program pro sestavení následující tabulky:

n	1	2	3	4	5	6	7	8
$T(n)$	1	3	7	15	31	63	127	255

Je zřejmé, že pro $n \leq 8$ platí

$$T(n) = 2^n - 1. \tag{3.7}$$

Dokažme si tuto rovnost pro $n \in \mathbb{N}$ (*Matematická indukce*):

1. Zřejmě platí $T(1) = 1$.
2. Předpokládejme, že rovnost (3.7) platí pro $m \in \mathbb{N}$.
3. Dokažme, tuto rovnost pro $m + 1$:
   podle (3.6) $\quad T(m+1) = 2 \cdot T(m) + 1 = 2 \cdot (2^m - 1) + 1 = 2^{m+1} - 1.$

Nyní můžeme odpovědět na otázku z úvodu. Za jakou dobu nastane konec světa?

$$\text{Za } 2^{64} - 1 \text{ sekund, což je více než za 584 miliard let.}$$

Uvedený příklad by neměl svádět k představě, že určení složitosti problému je jednoduchou záležitostí. V mnoha praktických případech je velmi komplikované určit pouze odhad složitosti problému. O těchto problémech se zmíníme v následujících kapitolách.

Cvičení

1. Platí $2^{n+1} = O(2^n)$? Platí $2^{2n} = O(2^n)$?
2. Jak dlouho trvá napočítání do 100000. Vyzkoušejte na vašem počítači program

```
j=0;
for(i=1; i < 100000; i++)
  j++;
```

3. Odpovězte na předcházející otázku s použitím *repeat* a *while*.
4. Pro každou funkci $f(n)$ a čas t v následující tabulce určete největší n pro které je problém řešitelný v čase t. Předpokládáme, že doba trvání problému o rozsahu n je $f(n)$ mikrosekund.

$f(n)$	1s	1min	1hod	1den	1rok	1století
$\log n$						
$\sqrt{n}$						
n						
$n \cdot \log n$						
n^2						
n^3						
2^n						
$n!$						

5. Pro řešení daného problému máme k dispozici dva programy P_1 a P_2 s časovou složitostí danou funkcemi

$$\begin{aligned} t_1(n) &= n^2 \\ t_2(n) &= n \log n + 10^{10} \end{aligned}$$

Určete pro které rozsahy dat je lepší program P_1.

3.4 Rekurze

Vtipný úvod...

Rekurze je dnes považována za jednu ze základních technik používaných v programování. Přesto je často považována za něco tajemného, neboť v případě nevhodného použití může vést k velmi špatně hledaným chybám.

Co si máme představit pod pojmem rekurze? Rekurzí rozumíme techniku, kdy dochází k opakovanému použití programové konstrukce při řešení téže úlohy. Taková definice by se ovšem příliš nelišila od již známé konstrukce cyklu. Ovšem na rozdíl od cyklu u rekurze použití téže konstrukce je zahrnuto uvnitř konstukce samotné. Používá se všude tam, kdy je efektivní původní úlohu rozdělit na menší podúlohy a poté použít tentýž postup řešení pro každou podúlohu.

V programování je **rekurze** představována funkcí nebo procedurou, která uvnitř těla funkce nebo procedury obsahuje volání téže funkce nebo procedury. Říkáme, že funkce nebo procedura „volá samu sebe“.

Formy rekurze

Rekurzivní algoritmus je možné vyjádřit jako konstrukci $\mathcal{K}$, která se skládá ze základních příkazů P_i a samotného K

$$K \equiv \mathcal{K}[P_i, K]$$

Hovoříme o dvou typech rekurentních konstrukcí:

- o **přímou rekurzi** se jedná v případě, že konstrukce K obsahuje přímé volání sama sebe,
- konstrukce K je **nepřímo rekurzivní**, obsahuje-li volání jiné konstrukce, označme ji P, která opět volá konstrukci K.

Při použití rekurze je nutné stanovit podmínku A pro ukončení rekurentního volání

$$K \equiv if\ A\ then\ \mathcal{K}[P_i, K]$$

Zajištění podmínky je diskutováno v následujících částech textu.

V oblasti programování se s rekurzí setkáme při výpočtu faktoriálu či Fibonacciho čísel, při generování anagramů, při rekurentním prohledávání binárních stromů, při skládání Hanojských věží.. Setkáváme se s ní i běžně v životě - při zkoumání přírodních závislostí (fraktály ve stavbě rostlin) i při použití techniky (snímání kamery a zobrazení sebe sama).

3.4.1 Charakteristika rekurze

Princip rekurze si ukážeme na výpočtu faktoriálu čísla n. Funkce faktoriál je definována

$$f(n) = \begin{cases} 1 & pro\ n = 0 \\ f(n * f(n-1)) & pro\ n \geq 1 \end{cases}$$

po rozepsání dostáváme

$$\begin{aligned} f(n) &= f(n * f(n-1)) = f(n * f((n-1) * f((n-2)))) = \ldots = \\ &= f(n * f((n-1) * f((n-2) * \ldots * f(1 * f(0)) \ldots)) \end{aligned}$$

což vede ke známému vyjádření

$$n! = n * (n-1)! = n * (n-1)(n-2)! = \ldots = n * (n-1) * (n-2) * \ldots * 1 * 0!$$

Jak probíhá výpočet faktoriálu pro $n = 10$ ukazuje tabulka 3.1.

Všimněme si, jak rychle narůstá rozsah výsledné hodnoty, což obecně při nevhodném použití rekurze může způsobit přetečení a ve výsledku se mohou zobrazit naprosto nesmyslné hodnoty. Pro rekurzi je tedy před jejím použitím nutná dobrá analýza a stanovení správné podmínky pro její ukončení.

Jednoduchý přepis výpočtu faktoriálu:

```
int  factorial (int n)
  {
   if (n == 0)
     return 1;
```

Vstupní hodnota	Výpočet hodnoty	Hodnota faktoriálu
0	podle definice	1
1	$1 * 1$	1
2	$2 * 1$	2
3	$3 * 2$	6
4	$4 * 6$	24
5	$5 * 24$	120
6	$6 * 120$	720
7	$7 * 720$	5 040
8	$8 * 5\,040$	40 320
9	$9 * 40\,320$	362 880
10	$10 * 362\,880$	3 628 800

Tabulka 3.1: Výpočet hodnot faktoriálu

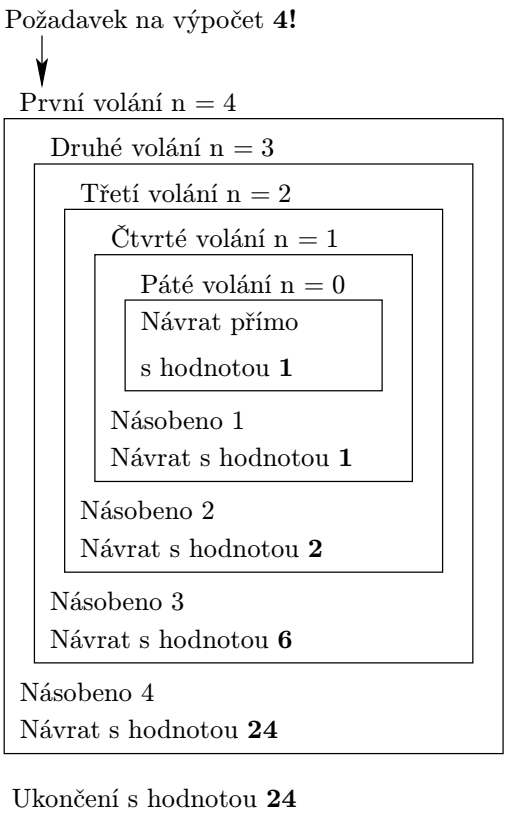
```
  else
    return (n * factorial (n − 1));
}
```

Co se děje při provádění rekurentního volání?

Jak je zajištěno, aby se uchovaly všechny hodnoty proměnných v okamžiku rekurentního volání dílčí úlohy a při jejím ukončení, tj.návratu z poslední rekurentní funkce či procedury byly předány správné hodnoty?

Můžeme si představit, že jednotlivé úlohy seřadíme do řady, kdy právě řešená úloha je na posledním místě. V okamžiku dokončení výpočtu poslední úlohy platnost hodnot s ní spojených končí a proto dojde k jejímu uvolnění z konce pomyslné řady. Řazení prvků do řady a přístup k pouze poslednímu z nich je typické pro zásobník. Realizace rekurze je tedy podporována paměťovým zásobníkem, ve kterém jsou uchovávány všechny aktuální hodnoty proměnných a parametů v okamžiku volání rekurentních funkcí či procedur, a k jejich uvolňování dochází při ukončení každé z nich v opačném pořadí, jak je naznačeno na obrázku 3.2.

Každým rekurzivním voláním funkce či procedury vzniká nová množina všech parametrů a lokálních proměnných. Mají sice stejné identifikátory jako při prvním volání, ale jejich hodnoty jsou jiné. Tento problém se řeší uchováním patřičných hodnot právě v zásobníkové paměti. Zde se pro každou úroveň volání vyčlení dostatečný prostor, ve kterém jsou po dobu trvání dané rekurentní funkce či procedury uchovány všechny potřebné hodnoty. Při návratu z rekurzivního volání požadujeme, aby se výpočet dokončil s těmi hodnotami, které odpovídají patřičné úrovni. Návraty z rekurentních volání



Obrázek 3.2: Princip vnořování rekurze

probíhají v opačném pořadí a tím je zajištěno ...výběr... odpovídajících hodnot pro danou úroveň rekurze.

Na obrázku 3.2 je patrné použití vhodné podmínky pro ukončení rekurze. Při pátém volání je předávána hodnota $n = 0$, což znamená okamžité dosazení (jak vyplývá z definice faktoriálu) a návrat do vyšší úrovně s hodnotou 1. Stanovení vhodné podmínky většinou vyplyne z analýzy řešené úlohy nebo rozborem navrhovaného algoritmu. Pokud ovšem dojde k situaci, že podmínka není dobře formulována, hrozí nekonečné opakování rekurentního volání. Problematika stanovování vhodných podmínek byla zmíněna u konstrukce cyklů.

Nevhodné použití rekurze nemusí být zaviněno pouze špatně stanovenou podmínkou jejího ukončení, ale v některých případech vyplývá i ze samotného použití rekurze. Pak už je na programátorovi, aby analýzou algoritmu dokázal předejít takovým případům – v další části je předvedeno na příkladu Fibonacciho čísel.

Vlastnosti rekurze

Předchozí příklad ukazuje základní typické vlastnosti každé rekurentní konstrukce.

- volání sama sebe,
- volání téhož algoritmu vede k řešení "menšího" problému,
- je-li řešen nejjednodušší případ problému, je aktivována podmínka pro ukončení rekurze a dochází k návratu z konstrukce bez volání sebe sama.

Při předávání argumentu s minimální povolenou hodnotou podmínka ukončení zajistí návrat bez použití rekurentního volání.

Použití rekurze vede k rozkladu řešené úlohy na dílčí úlohy, které se pak řeší analogicky jako původní úloha. Důležité je, aby dílčí úlohy měly nižší složitost než úloha původní a daly se jednoduše spojit do výsledného řešení.

Rekurzivní algoritmy mají většinou exponenciální časovou složitost, neboť rozklad úlohy rozměru n vede na n úloh rozměrů $n-1$. Proto se pokoušíme algoritmus upravit tak, abychom snížili jeho časovou náročnost. Je-li n rozměr úlohy a součet rozměrů částečných úloh je $a * n$ pro $a > 1$, má algoritmus polynomiální složitost . Tento princip označovaný jako *divide-and-conquer* (rozděl a panuj) je užíván při takových úlohách, kde je možné rozdělováním na menší podúlohy dojít k základní jednoduše řešitelné úloze (ta je současně podmínkou pro ukončení rekurze) a nezhorší se časová složitost použitého řešení.

3.4.2 Efektivita rekurze

Rekurzi je možné chápat jako zobecněnou iteraci, uvědomíme-li si, že se jedná o opakování určitého bloku příkazů a podmínku ukončení opakování umístěnou uvnitř tohoto bloku[2].

Kdy je výhodné použít rekurentní postup a kdy se použití rekurze ukáže jako neefektivní? Předvedeme si, jak se vyhnout neúčinnému rekurentnímu výpočtu na příkladu výpočtu Fibonacciho čísel[3].

Úloha byla poprvé publikována roku 1202 Leonardem Pisano (známým též pod jménem Leonardo Fibonacci) v knize *Liber Abacci* s následujícím zněním:

Kolik potomků – párů králíků bude mít po roce jeden původní králičí pár?

K řešení problému se uvádí, že každý pár králíků plodí nový pár králíků dvakrát – po měsíci a jestě jednou po dvou měsících. Poté se přestane rozmnožovat. Chceme vědět kolik bude nových párů v jednotlivých generacích.

Výpočet každé generace, tj. n - tého čísla Fibonacciho posloupnosti přirozeně vede k použití rekurze. V první generaci je to jeden pár, v druhé generaci dva páry, ve třetí tři páry, ve čtvrté pět párů, v páté generaci osm párů, ... Z těchto několika uvedených hodnot je vidět, že každé nové **Fibonacciho číslo** se dá vypočítat jako součet dvou předchozích.

Předpis Fibonacciho funkce

$$fib(n) = \begin{cases} 0 & pro\ n = 0 \\ 1 & pro\ n = 1 \\ fib(n-1) + fib(n-2) & pro\ n > 1 \end{cases}$$

Řada Fibonacciho čísel vypadá následovně

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

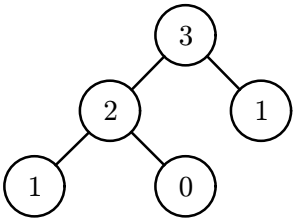
Pro výpočet hodnot je možné použít jednoduchý přepis definice funkce do algortimu využívajícího rekurentní vztah.

```
int fib(int n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fib(n - 1) + fib(n - 2);
}
```

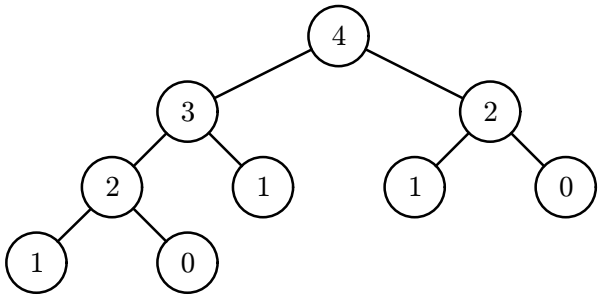
Provedeme-li ale podrobnější rozbor úlohy, ukáže se neustále nárůst dílčích úloh – roste množství výpočtů již známých členů posloupnosti.

[2] Umístění podmínky není možné na začátku opakovacího bloku, neboť takový případ by vedl k nekonečnému cyklu.

[3] Zlomky tvořené po sobě následujícími Fibonacciho čísly představují poměry známé z botaniky a limitně vedou k takzvanému *zlatému řezu*



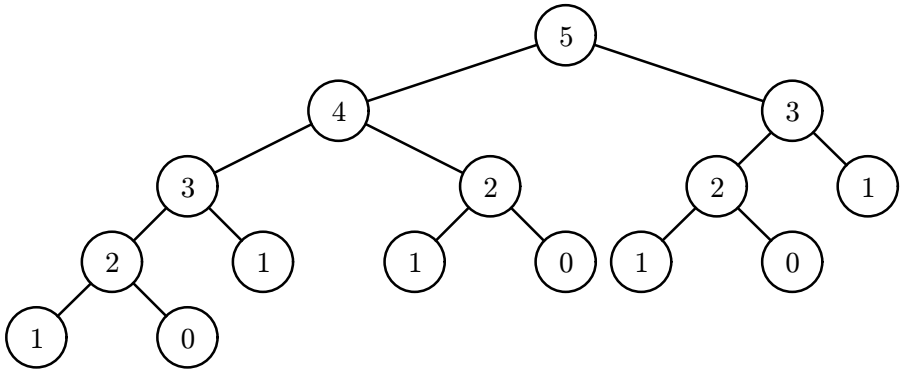
Obrázek 3.3: F3



Obrázek 3.4: F4

Naznačíme-li výpočet ve formě binárního stromu, kdy každý uzel představuje jedno volání funkce pro výpočet Fin(n). Kořenem je hledané Fibonacciho číslo a potomci každého uzlu na jednotlivých úrovních představují dvě předchozí Fibonacciho čísla nutné pro výpočet. Z uvedených obrázků (3.3, 3.4, 3.5) je patrné, kolikrát je nutno opakovaně počítat již známé hodnoty. U každé hladiny stromu dojde ke zdvojnásobení počtu uzlů, dvakrát se zvětší počet dílčích úloh. Je jasné, že složitost výše uvedeného algortimu bude exponenciální. Takto použitá rekurze je neefektivní, navíc nároky na paměť jsou zbytečně přehnané. Stačí si zapamatovat již vypočítané hodnoty Fibonacciho funkce a tím se sníží jak časová složitost, tak i velikost použité paměti.
Použijeme-li pomocné pole F, do kterého si budeme ukládat již jednou vypočítaná Fibonacciho čísla, snížíme exponenciální složitost algoritmu na lineární.

```
int  fibi (int n)
{
   int f[3], i;

   for (i = 0; i <= n; i++)
      f[i % 3] = f[(i - 1) % 3] + f[(i - 2) % 3];

   return f[(i - 1) % 3];
}
```



Obrázek 3.5: F5

Ukázka časového srovnání obou kódů pro čísla 42 a 45. Výpočet probíhal na stejném počítači a z níže uvedených hodnot je patrné, jak použití rekurze vede k nárůstu spotřeby strojového času.

```
42
real    0m12.753s
user    0m12.600s
sys     0m0.000s

45
real    0m53.436s
user    0m53.060s
sys     0m0.020s

42
real    0m0.004s
user    0m0.000s
sys     0m0.000s

45
real    0m0.005s
user    0m0.000s
sys     0m0.000s
```

Z algoritmu je patrné, že pouhou změnou přístupu k řešení — místo rekurzivního přístupu shora dolů jsme použili iterační postup (takzvanou Birdovu tabelační metodu [4]) — dojde k významnému snížení jak časových, tak i paměťových nároků.

Ovšem v případě, že potřebujeme jen jednu hodnotu z řady Fibonacciho čísel, nám tento postup nepomůže a pak je vhodné eliminovat rekurzi na základě rozboru dané úlohy a použít například kombinaci rekurentího a iteračního řešení.

Jak je tedy vhodné postupovat v případě, že chceme rekurzi odstranit? Provedeme analýzu úlohy s použitím rekurze. Zjistíme, zda existuje řešení

úlohy, které přímo vede k algoritmu bez oužití rekurze (jako tomu bylo v případě Fibonacciho čísel). Nebo rekurzivní program rozdělíme na disjunktní podprogramy, vyhledáme rekurzivní i vzájemná volání a upravíme jej do iteračního tvaru.

Kapitola 4

Lineární datové struktury

Množiny jsou jak pro matematiku, tak pro informatiku základní strukturou. Zatímco matematické množiny se většinou nemění, množiny používané v algoritmech se mění – množiny se mohou zvětšovat, zmenšovat nebo jinak měnit. Takové množiny označujeme jako **Datové struktury** a říkáme, že mají dynamický charakter. Tyto datové struktury se dělí na **datové struktury lineární** (data jsou uložena lineárním způsobem) a **datové struktury nelineární**.

Algoritmy pracují s množinami pomocí různých operací. Některým algoritmům postačuje vložení prvku, smazání prvku a test přítomnosti v množině. Jiné používají komplikovanější operace např. vyjmutí nejmenšího prvku. Z toho plyne, že nejlepší implementace množiny silně závisí na používaných operacích.

Prvky množin

Implementace prvků množin je různá. Od jednoduchých typů až po třídy s komplikovanou vnitřní strukturou. Tato struktura pro nás ale není zajímavá. Některé implementace množin kladou na prvky různé nároky. Předpokládá se například, že prvky lze nějakým způsobem identifikovat (navzájem rozlišit). Dále je možno požadovat, aby prvky náležely do úplně uspořádané množiny (platí trichotomie). Úplné uspořádání nám dovoluje mluvit o minimu resp. maximu nebo mluvit o dalším prvku větším než daný prvek množiny.

Typické operace nad množinami

Operace nad množinami můžeme rozdělit do dvou skupin: **dotazy**, které vrací informaci o množině a **modifikující operace**, které mění množinu. Mezi nejčastější operace patří:

[**Search(S,k)**] nalezení prvku k v množině S (dotaz).

[**Insert(S,x)**] vložení prvku x do množiny S (modif. operace).

[**Delete(S,x)**] vymazání prvku x z množiny S (modif. operace).

[**Minimum(S)**] nalezení minima úplně uspořádané množiny S (dotaz).

[**Maximum(S)**] nalezení maxima úplně uspořádané množiny S (dotaz).

[**Successor(S,x)**] nalezení dalšího prvku z množiny S většího než x. Vyžaduje úplně uspořádanou množinu. (dotaz)

[**Predecessor(S,x)**] nalezení předchozího prvku z množiny S menšího než x. Vyžaduje úplně uspořádanou množinu. (dotaz)

4.1 Pole

Pole (angl. array) patří k nejjednodušším datovým strukturám. Přístup k prvkům pole je určen udáním hodnoty **indexu** a není závislý na přístupu k jinému prvku. Proto říkáme, že pole je strukturou s **přímým** nebo náhodným přístupem.

Počet prvků pole může být určen pevně nebo se může měnit v době zpracování. V prvním případě nazýváme pole **statickým** a ve druhém **dynamickým**.

Ukážeme si, jak lze realizovat základní množinové operace pomocí pole.

Nesetříděné pole je nejjednodušší možností reprezentace n-prvkové podmnožiny S univerza U. V tomto poli je každé pozici pole $a[0, \ldots, n-1]$ přiřazen právě jeden prvek množiny S v libovolném pořadí. Všechny operace nad touto strukturou využívají **sekvenční vyhledávání**, které má lineární složitost v očekávaném i nejhorším případě.

Ukážeme si nyní několik možností realizace vyhledávání v poli.

```
template<class T> int find(T& x, const int n)
{
    for(int i = 0; i < n; i++)
    {
        if (x == a[i])
            return i
    }; // for
    return -1;
}
```

Vyhledávání pomocí zarážky je jednoduchou modifikací základního algoritmu. Na začátek pole vložíme hledanou hodnotu. Cyklus se zjednoduší o test podmínky překročení hranic pole.

```
template<class T> int find(T& x, const int n)
{
    int i = n;
    a[0] = x;
    while(x != a[i--]);
```

```
    return (i != 0 ? i: −1);
}
```

Tyto algoritmy potřebují v nejhorším případě n porovnání. To je případ, kdy hledaný prvek do množiny nepatří.

Průměrný počet porovnání spočteme tak, že sečteme počet porovnání, který je potřeba pro nalezení i-tého prvku, a ten vydělíme počtem prvků v poli. Vycházíme zde z předpokladu, že pravděpodobnost výskytu libovolného prvku množiny S je stejná. Za těchto předpokladů dostáváme

$$\begin{aligned} C_{avg} &= (1 + 2 + \cdots + n)\frac{1}{n} \\ &= \frac{1}{2}(1 + n)n\frac{1}{n} \\ &= \frac{1}{2}(1 + n) \end{aligned}$$

Setříděné pole

Využijeme-li uspořádání nad univerzem, můžeme reprezentovat n-prvkovou podmnožinu S univerza prostřednictvím **setříděného pole** $a[0, \ldots, n-1]$, v němž je každé pozici přiřazen právě jeden prvek z S. Platí: $a[i] \leq a[i+1]$, pro $i = 0, 1, \ldots, n-2$. V setříděném poli se vyhledává pomocí tzv. **binárního vyhledávání** (tento algoritmus bývá také nazýván „vyhledávání půlením intervalu“). Tuto metodu reprezentuje následující program.

```
template<class T> int find(T& x, const int n)
{
   int i = 0;
   int j = n;
   while (j != i + 1)
   {
     k = (i + j) / 2;
     if (a[k] x)
        j = k;
     else
        i = k;
   }; // while
   return (a[i] == x ? i: −1);
}
```

Program reprezentuje Dijkstrovo řešení binárního vyhledávání za předpokladu, že pro hledaný prvek platí vstupní podmínka $a[0] \leq x < a[n-1]$. Časová složitost binárního vyhledávání je $O(\log n)$, neboť každé porovnání zmenšuje vyhledávací prostor na polovinu – a to lze přibližně $\log n$ - krát.

Vvyhledávání interpolační

Implementace dynamického pole

V následující části si ukážeme implementaci dynamického pole. Další prvky do pole lze vkládat pomocí metody `Insert`, která při zaplnění stávajícího prostoru pole přealokuje. Dále je v tomto příkladu ukázka přetížení operátoru `[]` pro přístup k jednotlivým prvkům pole.

```
enum ErrorType = {invalidArraySize,
  memoryAllocationError,
  indexOutOfRange};

char *errorMsg[] = {"Invalid array size",
  "Memory allocation error",
  " Invalid index: " };


template <class T> class CArray
{
public:
  CArray(int sz = 50);
  ~CArray();
  T&   operator[](const int index);
  void  Insert (const T Item);
  int   Size(void) const;

private:
  void  Error(ErrorType error, int badIndex=0) const;
  void  Resize(int sz);

  T*   m_list;
  int  m_size;
  int  m_count;
};  // CArray

template <class T> CArray<T>::CArray(int sz): m_count(0)
{
  if (sz <= 0)
    Error( invalidArraySize );
  m_size = sz;
  m_list = new T[m_size];
  if (m_list == NULL)
    Error(memoryAllocationError);
}  // CArray::CArray

template <class T> CArray<T>::~CArray()
{   delete [] m_list; }

template <class T> T& CArray<T>::operator[] (const int index)
{
  if (n < 0 || index m_count-1)
    Error(indexOutOfRange, index);
  return m_list[index];
}  // CArray::operator[]
```

```
template <class T> void CArray<T>::Insert(const T Item)
{
  if (m_count= m_size)
    Resize(m_size + 10);
  m_list [m_count++] = Item;
} // CArray::Insert

template <class T> int CArray<T>::Size(void) const
{ return m_size; }

template <class T> void CArray<T>::Error(ErrorType error,
  int badIndex) const
{
  cerr << errorMsg[error];
  if (error == indexOutOfRange)
    cerr << badIndex;
  cerr << endl;
  exit(1);
} // CArray::Error

template <class T> void CArray<T>::Resize(int sz)
{
  if (sz <= 0)
    Error( invalidArraySize );
  if (sz == m_size)
    return;
  T* newlist = new T[sz];
  if (newlist == NULL)
    Error(memoryAllocationError);
  n = (m_count <= m_size) ? m_count : m_size;
  while (n--)
  {
    newlist[n] = m_list[n];
  }
  delete[] m_list;
  m_list = newlist;
  m_size = sz;
} // CArray::Resize
```

4.2 Zásobník

Zásobník (angl. stack) představuje jednoduchý typ množiny, u které je přesně určen způsob vkládání a mazání prvků. U zásobníku je uplatněn princip **last-in, first out – LIFO** tj. prvek, který byl poslední vložen, je jako první ze zásobníku vyzvednut. Zásobník lze přirovnat k zásobníku nábojů v pistoli[1]. Náboje jsou přesouvány do nábojové komory v opačném pořadí, než byly do zásobníku vloženy. V jednom okamžiku máme k dispozici pouze horní náboj nebo je zásobník prázdný. Ke spodním nábojům se lze dostat jen vyjmutím předchozích nábojů.

[1] Je myšlena pistole se zásobníkem v pažbě, nikoliv bubínkový revolver!

Obrázek nejde přeložit.

Obrázek 4.1: Zásobník

Ukazatel na aktuální prvek v zásobníku (posledně vložený) se nazývá **vrchol zásobníku** (angl. stack pointer). Opakem je **dno zásobníku**. Operace vložení do zásobníku se tradičně nazývá **Push** a vyjmutí se nazývá **Pop**. Jako třetí se u zásobníku implementuje dotaz **Empty**, který indikuje prázdnost zásobníku. Navíc se někdy přidává dotaz **Top**, který vrací prvek na vrcholu zásobníku, aniž by ho vyjmul (nedestruktivní varianta Pop). Pokud provedeme operaci Pop na prázdném zásobníku nastává chyba tzv. **podtečení** (angl. underflow). Zásobník má teoreticky neomezenou kapacitu. Pokud ji omezíme např. velikostí přidělené paměti, a nelze již přidat další prvek nastává opět chyba tzv. **přetečení** (angl. overflow).

Všechny zmiňované operace lze provést v konstantním čase, nezávisí tedy na velikosti zásobníku.

Zásobník lze implementovat jednak pomocí statických proměnných (v poli), jednak pomocí dynamických proměnných (dynamicky alokované záznamy a ukazatele na ně).

Implementace pomocí pole

Zásobník lze velice triviálním způsobem implementovat v poli.

```
template<class T>class CStack
{
public:
  CStack(int max = 100)
  {
    m_items = new T[max];
    m_sp = 0;
  }

  ~CStack()
  {  delete m_items; }

  void Push(T v)
  {  m_items[m_sp++] = v; }

  T Pop()
  { return m_items[--m_sp]; }

  T Top()
  { return m_items[m_sp]; }

  bool Empty()
  { return !m_sp; }

protected:
  T*  m_items;      // položky v zásobníku
```

```
  int  m_sp;          // stack pointer
};  // CStack
```

Implementace pomocí dynamických struktur

V této implementaci je zásobník realizován pomocí dynamicky alokovaných záznamů. Datová položka `m_z` představuje ukazatel na neexistující záznam zastupující standardní *NULL* z *C++* . Tento způsob reprezentace lze s úspěchem použít v implementaci mnoha datových struktur (viz například část 6.8). V následujícím příkladu je tento postup použit jen jako ukázka.

```
template<class T>class CStack
{
public:
  CStack()
  {
    m_sp = m_z = new CItem;
    m_z->next = m_z;
  }

  ~CStack()
  {
    CItem* aux = m_sp;
    while (m_sp != m_z)
    {
      aux = m_sp;
      m_sp = m_sp->next;
      delete aux;
    };  // while
    delete m_sp;
  }

  void Push(T v)
  {
    CItem* n = new CItem;
    n->data = v;
    n->next = m_sp;
    m_sp = n;
  }

  T Pop()
  {
    T      x  = m_sp->data;
    CItem* aux = m_sp;
    m_sp = m_sp->next;
    delete aux;
    return x;
  }

  T Top()
  { return m_sp->data; }
```

Obrázek 4.2: Fronta

```
  bool Empty()
  { return m_sp == m_z; }

protected:
  struct CItem
  {
    T       data;      // datová složka záznamu
    CItem* next;       // pointer na další záznam
  };  // CItem
  CItem* m_sp;          // stack pointer
  CItem* m_z;           // dno zásobníku
};  // CStack
```

4.3 Fronta

Dalším základním typem množiny s přesně určenými operacemi pro vkládání je **fronta** (angl. queue). Fronta uplatňuje mechanismus přístupu **FIFO** – **first in, first out** – jako první je z fronty odebrán prvek, který byl do fronty první vložen. Jde tudíž o obdobu fronty, jak ji známe z každodenního života. (V tomto okamžiku neuvažujeme prvky, které se mohou „předbíhat“. Potom bychom hovořili o frontě s prioritou).

Operace vložení prvku se tradičně nazývá **Put**, operace odebrání potom **Get**. Obdobně jako u zásobníku je definován dotaz **Empty**, který indikuje prázdnost fronty. Pokud provedeme operaci Get nad prázdnou frontou, nastane chyba **podtečení**. U velikostně omezené fronty může nastat i **přetečení**, překročíme-li při vkládání přidělený prostor.

Pro implementaci fronty jsou již potřeba dva ukazatele. Jeden ukazatel určuje **hlavu** (začátek) fronty (angl. head) tj. ukazuje na prvek, který je na řadě pro odebrání, druhým ukazatelem je **ocas** (konec) fronty (angl. tail). Tento ukazatel ukazuje na poslední prvek ve frontě.

Implementace pomocí dynamických struktur

Fronta se dá snadno realizovat pomocí dynamických struktur. Pomocí pole je implementace o něco obtížnější.

```
template<class T>class CQueue
{
public:
  CQueue()
  {
    m_head = m_tail = NULL;
  }  // CQueue

  ~CQueue()
```

```
  {
    while (m_head != NULL)
    {
      m_tail = m_head;
      m_head = m_head->next;
      delete m_tail;
    }; // while
  } // ~CQueue

  void Put(T x)
  {
    CItem* n;
    n = new CItem;
    n->data = x;
    n->next = NULL;
    if (Empty())
      m_head = n;
    else
      m_tail->next = n;
    m_tail = n;
  } // Put

  T Get()
  {
    CItem* aux;
    T      result;
    if (!Empty())
    {
      aux = m_head;
      m_head = m_head->next;
      result = aux->data;
      delete aux;
    }; // if
    return result;
  } // Get

  bool Empty()
  { return m_head == NULL; }

protected:
  struct CItem
  {
    T      data;    // data
    CItem* next;    // další prvek fronty
  }; // CItem

  CItem* m_head;    // hlava fronty
  CItem* m_tail;    // ocas fronty
}; // CQueue
```

4.4 Seznam

Spojový seznam (angl. linked list) je datová struktura, ve které jsou data uložena lineárním způsobem. Na rozdíl od pole, kde lineární uspořádání je určeno indexem pole, pořadí prvku v seznamu je určeno ukazateli mezi prvky seznamu. Spojový seznam umožňuje jednoduchou, pružnou reprezentaci (ovšem ne nutně efektivní) všech typických operací s dynamickými množinami.

Obousměrný spojový seznam (angl. doubly linked list) je tvořen objekty (daty, prvky, záznamy) a dvěma ukazateli **prev** a **next**. Každý objekt pochopitelně může obsahovat další data specifická pro danou aplikaci. Ukazatel `prev` ukazuje na předchůdce daného prvku seznamu, ukazatel `next` ukazuje na následníka daného prvku seznamu. Jestliže ukazatel `prev` prvku x je roven hodnotě NULL, prvek x nemá tudíž předchůdce, je prvním prvkem seznamu a tvoří **hlavu** seznamu. Jestliže ukazatel `next` prvku x je roven NULL, daný prvek nemá následníka, je tedy poslední v seznamu a tvoří **ocas** seznamu. Položka `m_head` ukazuje na první prvek seznamu. Jestliže je `m_head` rovna NULL, seznam je prázdný.

Spojové seznamy se vyskytují v mnoha variantách. Mohou být jednosměrné nebo obousměrné, setříděné nebo nesetříděné, cyklické (kruhové) nebo acyklické. Jestliže v prvcích seznamu vynecháme ukazatel `prev`, dostaneme **jednosměrný seznam**. Seznam nazýváme **setříděný**, jestliže prvky seznamu jsou seřazeny. V opačném případě se seznam nazývá **nesetříděný**. V **cyklickém seznamu** ukazuje ukazatel `prev` hlavy seznamu na ocas seznamu a ukazatel `next` zase na hlavu seznamu. Seznam si lze představit jako prstenec z prvků. V dalším výkladu budeme uvažovat nesetříděný obousměrný spojový seznam.

Ukázka implementace

Seznam lze realizovat například následující třídou:

```
template<class T>class CList
{
public:
  CList();
  ~CList();

  bool Search(T a);
  void  InsertFirst (T a);
  void Delete(T a);

protected:
  struct CListItem
  {
    T           data;        // data prvku
    CListItem*  prev;        // předcházející prvek
    CListItem*  next;        // následující prvek
```

```
  };  // CListItem

  CListItem*    m_head;        // hlava seznamu
};  // CList
```

Konstruktor a destruktor

Úkolem **konstruktoru** je, stručně řečeno, vytvořit novou instanci třídy a inicializovat členské proměnné třídy. V našem případě musíme nastavit proměnnou `m_head` na nějakou zvolenou hodnotu. Protože vytváříme na začátku seznam prázdný nastavíme `m_head` na NULL.

```
template<class T> CList<T>::CList()
{
  m_head = NULL;
}  // CList<T>::CList
```

Destruktor má za úkol naopak regulérně uvolnit veškerou paměť alokovanou danou instancí. V případě seznamu se musíme postarat o uvolnění všech alokovaných prvků.

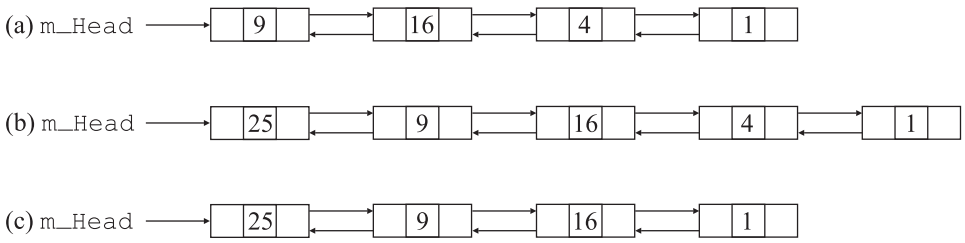
```
template<class T> CList<T>::~CList()
{
  CListItem* p;
  while(m_head != NULL)
  {
    p = m_head;
    m_head = m_head->next;
    delete p;
  };  // while
}  // CList<T>::~CList
```

Vyhledávání v seznamu

Vyhledávání v seznamu realizujeme metodou `bool CList<T>::Search(T a)`. Metoda vrací $true$, jestliže je prvek a nalezen, jinak vrací $false$.

```
template<class T>bool CList<T>::Search(T a)
{
  CListItem*  x;
  for(x = m_head; x != NULL; x = x->next)
    if (x->data == a)
      return true;
  return false;
}  // CList<T>::Search
```

Jak je vidět z kódu metody, je nutno při hledání probrat postupně všechny prvky seznamu. Složitost vyhledávání je proto $\Theta(n)$ uvažujeme-li seznam s n prvky.



Obrázek 4.3: Seznam
(a) Obousměrný spojový seznam představující (dynamickou) množinu $\{1, 4, 9, 16\}$. **(b)** Seznam po provedení operace InsertFirst(25). **(c)** Seznam po smazání prvku 4

Vložení prvku

Mějme například metodu `void CList<T>::InsertFirst(T a)`, která vloží prvek na začátek seznamu (viz obrázek 4.3(b)). Zde je nutno zvlášť ošetřit případ, kdy vkládáme do prázdného seznamu.

```
template<class T>void CList<T>::InsertFirst(T a)
{
  CListItem* x;

  x = new CListItem;
  x->data = a;
  x->next = m_head;
  if (m_head != NULL)
    m_head->prev = x;
  m_head = x;
  x->prev = NULL;
}  // CList<T>::InsertFirst
```

Složitost této metody je $\Theta(1)$.

Smazání prvku

Metoda `void CList<T>::Delete(T a)` smaže prvek a ze seznamu. Nejprve je nutno prvek a nalézt a potom jej vhodnou záměnou ukazatelů okolních prvků vyjmout ze seznamu.

```
template<class T>void CList<T>::Delete(T a)
{
  CListItem* x;
// nalezení prvku
  for(x = m_head; x != NULL; x = x->next)
    if (x->data == a)
      break;
  if (x == NULL)
    return;    // nenalezeno
// vyjmutí ze seznamu
  if (x->prev != NULL)
```

```
    x->prev->next = x->next;
  else
    m_head = x->next;
  if (x->next != NULL)
    x->next->prev = x->prev;
  delete x;
} // CList<T>::Delete
```

Vlastní vyjmutí ze seznamu lze provést v konstantním čase, ale je tu opět nutnost vyhledat prvek v seznamu, což se lze provést v nejhorším případě v čase $O(n)$.

Zarážky

Kód metody `Delete` by šel výrazně zjednodušit, kdybychom mohli ignorovat hraniční podmínky na začátku a konci seznamu. Potom by kód `Delete` mohl vypadat následovně (mimo hledání v seznamu):

```
x->prev->next = x->next;
x->next->prev = x->prev;
```

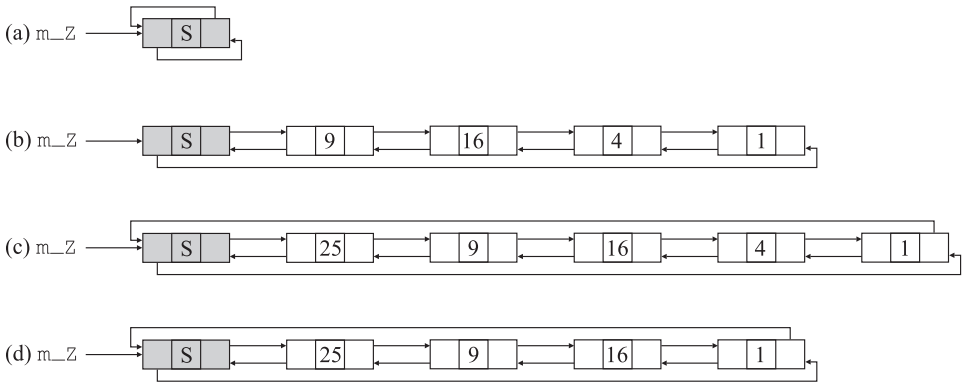
Zarážka (angl. sentinel) je pomocný prvek, který umožňuje zjednodušit hraniční podmínky při práci se seznamem. Většinou se zarážka realizuje jako normální prvek seznamu, který nenese žádná data. Označme jej například `m_z`. Všechny odkazy na NULL v metodách seznamu zaměníme za `m_z`. Obrázek 4.4 ukazuje, jak se použitím zarážky změnil obousměrný seznam na cyklický seznam tím, že jsme zarážku umístili mezi hlavu a ocas seznamu. Z toho plyne, že `m_z->next` ukazuje na hlavu seznamu (nyní lze ukazatel `m_head` vynechat) a `m_z->prev` ukazuje na ocas seznamu. Prázdný seznam obsahuje jen ukazatel `m_z` a ukazatele `prev` a `next` jsou nastaveny samy na sebe.

Kód metody Search se změní velice málo

```
template<class T>bool CList<T>::Search(T a)
{
  CListItem* x;
  for(x = m_head; x != m_z; x = x->next)
    if (x->data == a)
      return true;
  return false;
} // CList<T>::Search
```

Výrazně se však zjednoduší metoda InsertFirst

```
template<class T>void CList<T>::InsertFirst(T a)
{
  CListItem* x;
  x = new CListItem;
  x->data = a;
  x->next = m_z->next;
  m_z->next->prev = x;
  m_z->next = x;
```



Obrázek 4.4: Sentinely

Obousměrný spojový seznam se zarážkami m_z se změní na kruhový seznam, kde m_z je vložen mezi hlavu a ocas seznamu. **(a)** Prázdný seznam. **(b)** Seznam z obrázku 4.3, s prvkem 9 na začátku a prvkem 1 na konci. **(c)** Seznam po vložení 25. **(d)** Seznam po smazání prvku 4.

```
  x->prev = m_z;
}  // CList<T>::InsertFirst
```

Zarážky neovlivní složitost samotných operací nad seznamy. Slouží spíše k přehlednějšímu zápisu kódu operací. Mohou však urychlit běh programu jako celku, pokud například provádíme operaci se seznamem v cyklu s velkým počtem opakování.

Zarážky pochopitelně spotřebují paměť odpovídající jednomu prvku v seznamu navíc. Tento nárůst lze považovat za bezvýznamný, pokud zpracováváme seznamy s velkým počtem prvků, ale pokud jsme nuceni zpracovávat velké množství seznamů s málo prvky je využití paměti značně nehospodárné. Například 10 seznamů o 1000 prvcích potřebuje 10 zarážek (10 zarážek : 10000 prvků). Naproti tomu 1000 seznamů o 10 prvcích potřebuje 1000 zarážek (1000 zarážek : 10000 prvků).

Cvičení

1. Jaké výhody a nevýhody mají datové struktury, se kterými jste se právě seznámili?
2. Co je to indexace a k čemu je používána?
3. Máme zásobník, ve kterém je uloženo 9 prvků. Jak zjistíme hodnotu prvku, který byl do zásobníku uložen jako první? Liší se operace pro zjištění hodnoty prvního prvku u fronty od této operace u zásobníku?

4. Mohou být do pole, zásobníku nebo fronty ukládány prvky různých datových typů?

5. Máme 10 prvků, které jsou postupně ukládány do pole a stejné prvky jsou postupně uloženy v obousměrném spojovém seznamu. Jak zjistíme hodnotu prvku, který byl ukládán jako třetí v pořadí v poli a v seznamu?

6. Je složitost operace vyhledání prvku v poli a v obousměrném spojovém seznamu stejná?

7. Realizujte pole pomocí fronty.

8. Realizujte dvourozměrné pole pomocí fronty.

9. Realizujte pole pomocí zásobníku.

10. Realizujte zásobník pomocí fronty.

11. Realizujte frontu pomocí zásobníku.

12. Realizujte frontu pomocí pole.

13. Realizujte zásobník pomocí pole.

14. Pomocí zásobníku realizujte algoritmus, který zjistí zda posloupnost znaků má tvar xCy, kde x je posloupnost ze písmen A, B a y je opačná posloupnost k x. Např. $x = AAABAB, y = BABAAA$. Při čtení posloupnosti můžeme číst pouze následující symbol posloupnosti.

15. Realizujte dvojrozměrné pole pomocí dvou zásobníků.

Kapitola 5

Třídění

5.1 Úvod

Motivace

Třídění je činnost, která je vlastní lidskému rodu. Třídíme např. předměty každodenní potřeby (v bytě), nákupy (v tašce), listiny (v úřadě), peníze (v bance), knihy (v knihovně), lidi (ve společenských vědách), data (na počítači) atd.

V běžném životě se problém třídění chápe šířeji než v informatice, a označuje tzv. „škatulkování“ – tj. činnost, při které rozkládáme konečné množiny objektů na disjunktní podmnožiny objektů v nějakém smyslu ekvivalentních. Přitom vlastní pořadí těchto tříd nás z počátku nezajímá. Protože však cílem třídění je umožnit pozdější efektivní vyhledávání jednotlivých objektů (anebo celých tříd objektů), nadejde chvíle, kdy nás začne zajímat i pořadí jednotlivých tříd (pro dostatečně velký počet tříd). Zde se již blížíme třídění chápanému ve smyslu informatiky.

V dalším se bude pod pojmem **třídění** rozumět proces přeuspořádání prvků určité množiny reprezentované posloupností, podle určitého uspořádání (v matematickém smyslu).

Historie

Historicky pochází první stopa z 2. století před n.l. Je to hliněná babylónská tabulka, obsahující 800 lexikograficky setříděných až jedenáctimístných čísel – zapsaných v šedesátkové soustavě. Je zajímavé, že i když lidstvo uspořádávalo znaky své „abecedy“ do určitého pořadí prakticky od objevení písma, lexikografické uspořádání slov pochází z 13. století.

Z algoritmického hlediska se začala věnovat pozornost třídění až koncem 19. století, kdy byly v souvislosti se sčítáním obyvatelstva USA v roce 1890 vynalezeny první třídící stroje (H. Hollerith). Tyto třídící stroje se postupně zdokonalovaly, a co do výkonnosti je předčily až první počítače ve 40.

letech tohoto století. Není bez zajímavosti, že první program, který měl ověřit adekvátnost strojových instrukcí jednoho z prvních počítačů (EDVAC), byl třídící program, který sestrojil John von Neumann v roce 1945 (byl to algoritmus třídění slučováním).

Od těch dob nastal spolu s rozvojem počítačů bouřlivý rozvoj třídících algoritmů. V průběhu následujících asi 30 roků byly objeveny všechny základní třídící techniky zhruba v té formě, jak je popsal D. E. Knuth [11]. Z tohoto díla čerpají i všechny pozdější práce K. Mehlhorn [13], Wiedermann [22], N. Wirth [23].

5.2 Třídící problém

Třídící problém budeme definovat následovně: je daná množina $A = \{a_1, a_2, ..., a_n\}$. Je potřebné najít permutaci π těchto n prvků, která zobrazuje danou posloupnost do neklesající posloupnosti $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ tak, že $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$ Množinu U, ze které vybíráme prvky tříděné množiny, nazýváme univerzum.

Třídící problém, tak jak jsme jej definovali v předcházející části, je stále ještě jistou abstrakcí reálné situace, kdy obyčejně máme s každým prvkem z U vázanou nějakou další informaci, která však nemá na definici uspořádání žádný vliv. Prvky množiny U se nazývají **klíče** a informace vázaná na klíč spolu s klíčem nazýváme **záznam**. Jestliže je velikost vázané informace příliš „velká“, je výhodnější setřídit jen klíče s patřičnými odkazy na vázané informace, které se v tomto případě nehýbou. Bez újmy na obecnosti budeme dále předpokládat, že třídíme pouze klíče.

Třídící metoda se nazývá **stabilní**, když zachovává relativní uspořádání záznamů se stejným klíčem. To znamená, že pro třídící permutaci platí:

$$\pi(i) < \pi(j) \text{ právě tehdy, když } a_{\pi(i)} = a_{\pi(j)} \text{ pro } 1 \leq i < j \leq n.$$

Stabilita třídění je často důležitá tehdy, když jsou prvky již uspořádané podle určitých **sekundárních klíčů**, to znamená vlastností, které samotný **(primární) klíč** neodráží. Dále, třídící algoritmus nazýváme **přirozeným**, jestliže jeho složitost roste resp. klesá v závislosti na míře setříděnosti vstupní posloupnosti (viz strana **??**). Třídění se nazývá **in situ** (neboli na původním místě), jestliže třídící algoritmus vyžaduje, kromě vlastního tříděného pole, pomocnou paměť pouze konstantního rozsahu. Jinými slovy algoritmus nepoužívá žádnou další paměť, jejíž velikost by byla závislá na rozsahu tříděných hodnot (např. další pole rozsahu n).

5.2.1 Klasifikace třídících algoritmů

Podle toho jak třídící algoritmy pracují, můžeme je rozdělit do několika skupin:

- algoritmy **adresního třídění**, které využívají jednoznačný vztah mezi absolutními hodnotami prvků z U a jejich pozicí v uspořádané množině. Pro výpočet tohoto vztahu můžeme použít libovolné operace mimo porovnání tříděných prvků.

- algoritmy **asociativního třídění**, které používají pro určení pozice prvku v S jen relativní hodnoty prvků, které určují vzájemným porovnáváním těchto prvků.

- algoritmy **hybridního třídění**, které jsou kombinací předcházejících metod. Jinak řečeno tyto algoritmy nejsou nijak omezovány při zjišťování pozice daného prvku.

Jestliže cílový datový typ, nad kterým implementujeme třídicí algoritmus, připouští paralelní operace, mluvíme o **paralelním třídění**; jinak mluvíme o **sériovém třídění**.

Víme, že výběr vhodné datové struktury v podstatné míře ovlivňuje efektivitu algoritmu. Toto tvrzení je dvojnásobně pravdivé právě v případě třídících algoritmů.

Z definice problému třídění vidíme, že vstupem pro třídící algoritmus je tříděná množina S, reprezentovaná posloupností. Tuto posloupnost můžeme v počítači reprezentovat dvěma základními způsoby polem nebo seznamem. Polem můžeme množinu S reprezentovat v případě, že máme k dispozici dostatečně velkou paměť s přímým přístupem. Naopak, když počet údajů (prvků posloupnosti) převyšuje kapacitu paměti s přímým přístupem, musíme údaje reprezentovat seznamem.

Volba reprezentace už jednoznačně určuje jaké datové struktury můžeme při třídění používat. Tato volba vede k rozdělení třídících algoritmů na dvě fundamentálně odlišné třídy:

- algoritmy vnitřního (interního) třídění,

- algoritmy vnějšího (externího) třídění.

Všeobecně můžeme říci, že při **vnitřním třídění** máme větší volnost při výběru vhodných datových struktur (pole, seznamy, stromy...) a operací nad nimi. Naopak při **vnějším třídění** musíme vystačit jen se sekvenčním přístupem k prvkům tříděné množiny, a tedy třída přípustných algoritmů je oproti předcházejícímu případu značně omezená.

Z čistě teoretického hlediska odpovídají algoritmům vnitřního třídění algoritmy implementované na počítači RAM[1], a algoritmům vnějšího třídění algoritmy implementované na vícepáskovém Turingově stroji.

[1] RAM je v tomto případě zkratka ***R**andom **A**ccess **M**achine* —RAM. Jedná se o abstraktní model počítače spadající do oblasti teoretické informatiky

5.3 Adresní třídící algoritmy

Z definice adresních algoritmů v předcházející části vidíme, že tyto algoritmy nepoužívají při své činnosti žádnou preferovanou operaci. Proto za míru časové efektivnosti zvolíme celkový počet operací vykonaných po dobu třídění. Adresní třídění se podobá uklízení např. rozházených dětských hraček: když o každé hračce víme, na které místo patří, stačí ji jen vzít a dát na své místo.

5.3.1 Přihrádkové třídění

Přihrádkové třídění je základním algoritmem adresního třídění. Většinou slouží jako základ pro konstrukci složitějších algoritmů adresního třídění.

Algoritmus

Nechť $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ je posloupnost celých čísel z ohraničeného univerza $U = \{0, \ldots, m-1\}$ (některá z čísel a_i mohou být stejná). Když m není příliš veliké, potom je možné posloupnost efektivně setřídit touto metodou:

1. krok — inicializace: inicializuj m prázdných seznamů („přihrádek“), pro každé číslo $i \in \{0, \ldots, m-1\}$ jeden seznam;

2. krok — distribuce: čti posloupnost zleva doprava, a prvek a_i umísti do a_i-tého seznamu, na jeho konec (tj. seznamy se chovají jako fronty);

3. krok — zřetězení: zřetězit všechny seznamy tak, že začátek $(i-1)$-ního seznamu se připojí na konec i-tého.

Tak vznikne jediný seznam obsahující prvky v utříděném pořadí. Protože jeden prvek je možné zařadit do i-tého seznamu v konstantním čase, n prvků v čase $O(n)$. Zřetězení m seznamů si vyžaduje čas $O(m)$, takže celková složitost přihrádkového třídění je $O(m+n)$. Tento druh třídění se používá zejména tehdy, když $m \ll n$. Potom je jeho složitost lineární. Třídění sice není in situ, ale je stabilní.

Příklad adresního třídění

Mějme například realizovat funkci, která načte jednotlivé znaky ze souboru `InputName` a vypíše je setříděné do souboru `OutputName`. Celý problém lze naprogramovat velice jednoduše využitím zjednodušené verze přihrádkového třídění. Vytvoříme pole 256 počítadel (`counters`) – pro každý znak jedno – na počátku je nastavíme na 0. Potom čteme vstupní soubor a příslušné počítadlo inkrementujeme. Nakonec vypíšeme do výstupního souboru tolik znaků na kolik jsou jednotlivá počítadla nastavena, to znamená, že nejprve vypíšeme například 20 znaků „a“, potom 10 znaků „c“, 1 znak „f“ atd.

IKONA

Příklad 5.1
Mějme dánu posloupnost písmen
`s o r t i n g e x a m p l e`
Po setřídění dostaneme posloupnost
`a e e g i l m n o p r s t x`

```
void CharSort(char* InputName, char* OutputName)
{
  FILE* input = fopen(InputName, "rb");
  FILE* output = fopen(OutputName, "wb");
  int    counters[256];
  int    i, j;
  for(i = 0; i < 256; i++)
    counters[i] = 0;
  while ((i = getc(input)) != EOF)
    counters[i] += 1;
  fclose(input);
  for(i = 0; i < 256; i++)
    for(j = 1; j <= counters[i]; j++)
      putc(i, output);
  fclose(output);
} // CharSort
```

5.3.2 Lexikografické třídění

Přihrádkové třídění se dá rozšířit na lexikografické třídění posloupnosti řetězců celých čísel.

Definice 5.1 *Nechť $\leq$ je uspořádání na množině U. Relace $\leq$ rozšířená na řetězce s komponentami z U je **lexikografickým uspořádáním**, jestliže $(s_1, \ldots, s_p) \leq (t_1, \ldots, t_q)$ právě tehdy, když:*

1. *buď existuje číslo j takové, že $s_i = t_i$ pro $1 \leq i < j$ a $s_j < t_j$,*
2. *nebo $p \leq q$ a $s_i = t_i$ pro $1 \leq i \leq p$.*

Například když uvažujeme řetězce písmen, tak slova přirozeného jazyka ve slovníku jsou lexikograficky setříděná. Uvažujme nejdříve problém **lexikografického třídění** stejně dlouhých řetězců délky k, pro $k > 1$, s komponentami z intervalu $0 \ldots m-1$. Posloupnost potom můžeme setřídit následujícím postupem, ve kterém se v k průchodech střídají fáze distribuce do přihrádek a zřetězení přihrádek. Přitom v i-tém průchodu distribuujeme posloupnost, kterou jsme dostali v přecházejícím průchodu, do přihrádek podle $(k-i-1)$-ní komponenty, potom všechny přihrádky zřetězíme a postup opakujeme pro $i = 1, 2, \ldots, k$.

Vidíme, že jednotlivé k-tice prohlížíme odzadu, zprava doleva. I-tá distribuce třídí k-tice podle i-té komponenty odzadu, a když se dvě k-tice dostanou do té samé přihrádky, tak první z nich je lexikograficky menší (vzhledem

na poslední $i-1$, resp. i komponenty) než druhá, protože v tomto pořadí je zanechal předcházející přechod; přihrádkové třídění v i-tém přechodu jejich pořadí v důsledku stability nezmění.

Při praktické realizaci tohoto algoritmu je důležité k-tice při distribuci skutečně fyzicky nepřesouvat do přihrádek, ale místo nich se přesouvá jen ukazatel na příslušnou k-tici. Tím se dosáhne, že k-tice se může „přidat“ do přihrádky v čase $O(1)$ a ne v čase $O(k)$. Jeden přechod má potom složitost $O(m+n)$, a proto celková složitost algoritmu bude $T(n) = O(k(m+n))$. Pro $m \leq n$ má algoritmus lineární složitost, vzhledem k celkovému počtu komponent.

5.3.3 Třídění řetězců různé délky

Řetězce různé délky se dají doplnit na stejnou délku nějakým speciálním symbolem, a potom setřídit předcházejícím algoritmem. Když však musíme třídit jen několik dlouhých a hodně krátkých řetězců, tak je tento postup neefektivní, protože:

- v každém přechodu při distribuci se zkoumá hodně doplňujících symbolů;
- v každém přechodu zůstává hodně prázdných přihrádek, které se při zřetězování musí vynechat.

Neefektivnost algoritmu se projevuje i v odhadu jeho složitosti, který je $O((m+n)l_{max})$, kde l_{max} je délka nejdelšího řetězce. Algoritmus je totiž lineární vzhledem k počtu všech – i doplňujících komponent – ale nemusí být lineární vzhledem na počet původních komponent. První nedostatek odstraníme tak, že v předvýpočtu setřídíme řetězce podle jejich délky – od nejdelších po nejkratší. Potom aplikujeme předcházející algoritmus s l_{max} přechody, ale s tím rozdílem, že první přechod třídí jen řetězce délky l_{max}, druhý přechod třídí jen řetězce délky aspoň $l_{max}-1$,... atd. Tak dosáhneme, že celková složitost distribuce ve všech přechodech je úměrná celkové délce vstupu.

Druhý nedostatek odstraníme, když si v předvýpočtu sestrojíme postupným čtením všech řetězců tabulku, ve které si pro každou přihrádku poznamenáme, v kterém přechodu bude použita. K této tabulce potom sestrojíme znovu pomocí přihrádkového třídění inverzní tabulku, ve které bude zaznamenáno pro každý přechod, které přihrádky v něm budou neprázdné. Tuto tabulku potom využíváme v každém přechodu ve fázi zřetězování tak, že zřetězujeme jen neprázdné přihrádky. Tak dosáhneme, že celková složitost zřetězování ve všech přechodech je úměrná počtu neprázdných přihrádek, které se vyskytly po dobu celého třídění, a tento počet zřejmě není větší než byla délka vstupu.

Podrobnější analýzou tohoto algoritmu se dá dokázat, že jeho složitost bude: $T(n) = O(l_{total} + m)$, kde l_{total} je součet délek všech řetězců. To znamená, že pro fixní m jsme dostali skutečně algoritmus lineární složitosti.

5.3.4 Radix sort

Třídění **RadixSort** je postaveno na následujícím principu. Klíče užívané k definování pořadí záznamů jsou v mnoho případech velice komplikované. Například klíče užívané pro třídění v katalozích knihoven. Proto je vhodné postavit třídicí metody na porovnání dvou klíčů a výměně dvou záznamů. Pro mnoho aplikací je možno využít faktu, že klíč lze považovat za číslo jistého rozsahu. Třídící metoda založená na této myšlence se nazývá RadixSort. Tento algoritmus neporovnává dva klíče, nýbrž zpracovává a porovnává části klíčů.

RadixSort považuje klíče za čísla zapsaná v číselné soustavě o základu M (radix) a pracuje s jednotlivými číslicemi. Představme si úředníka, který má setřídit hromadu karet, přičemž na každé kartě je natištěno tříciferné číslo. Jeden z rozumných postupů je asi tento: vytvořit deset hromádek, na první dávat karty s čísly menšími než 100, na druhou karty s čísly 100 až 199 atd. Každou z těchto deseti hromádek pak znovu roztřídit stejným způsobem nebo pokud je na hromádce karet málo, setřídit je jednoduchým způsobem. Metoda, kterou jsem popsali je jednoduchou ukázkou RadixSortu o základě $M = 10$. Pochopitelně pro počítač je vhodnější pracovat s číselnou soustavou o základě $M = 2$.

Předpokládejme, že přeskupujeme záznamy v poli tak, že záznamy jejichž klíče začínají bitem 0 předcházejí záznamy s klíči začínajícími bitem 1. Tato úvaha vede na rekurzivní třídící algoritmus typu QuickSort: jestliže jsou dvě části pole setříděny, potom je setříděno i celé pole. Při výměnách záznamů v poli postupujeme zleva a hledáme klíč začínající na 1, stejně tak zprava hledáme klíč začínající na 0. Tyto záznamy vyměníme a pokračujeme dokud se indexy testovaných záznamů uprostřed pole nepřekříží.

Tato implementace je velice podobná QuickSortu. Rozdělování pole na dvě části je obdobné, s tím rozdílem, že jako pivot je použito číslo 2^b místo nějakého prvku z pole. Protože však číslo 2^b nemusí být obsaženo v daném poli, nelze zaručit, že se prvky dostanou na svá místa ihned při dělení pole. Stejně tak číslo 2^b nelze použít jako zarážku pro prohledávací cykly, tudíž je do těchto cyklů přidána podmínka $i < j$. Viz obrázky 5.1, 5.2, 5.3 a 5.4. Další podrobnosti lze nalézt v [18].

```
int bit(int a, int b)
{
  return (a >> b) & 1;
} // bit

void RadixExchange(int l, int r, int b)
{
```

```
  int  t, i, j;
  if ( (r > l) && (b >= 0))
  {
    i = l;
    j = r;
    do
    {
      while ((bit(a[i], b) == 0) && (i < j))
        i++;
      while ((bit(a[j], b) == 1) && (i < j))
        j--;
      t = a[i]; a[i] = a[j]; a[j] = t;
    } while (j != i);
    if (bit(a[r], b) == 0)
      j++;
    RadixExchange(l,j - 1,b - 1);
    RadixExchange(j,r,b - 1);
  }; // if
} // RadixExchange
```

ANIMACE

5.4 Asociativní třídicí algoritmy

Asociativní třídicí algoritmy jsou nejvšeobecnější skupinou třídících algoritmů, protože nepředpokládají o prvcích tříděné množiny nic víc než že jsou vybrané z uspořádaného univerza.

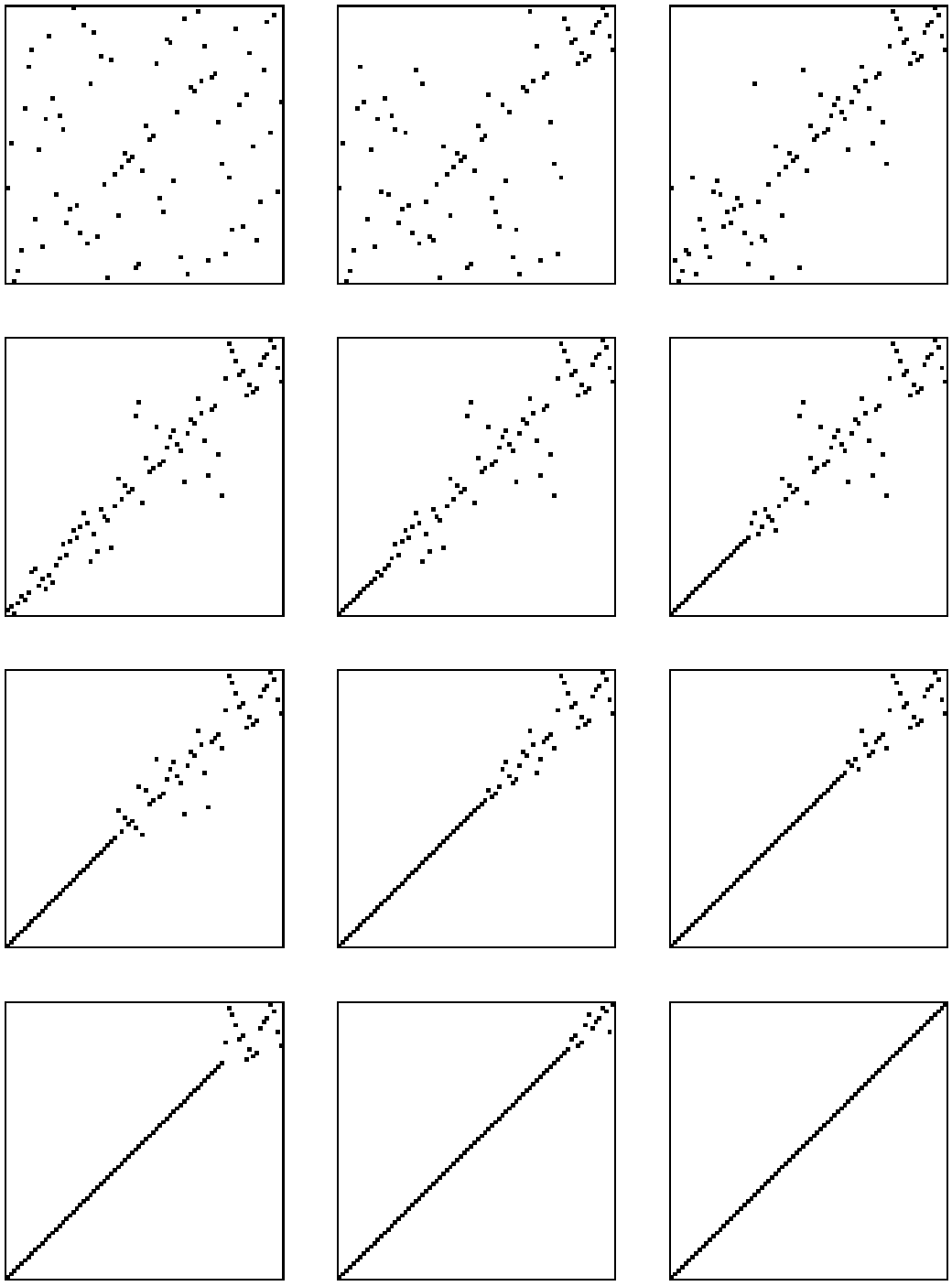
Praktické zkušenosti z analýzy asociativních třídicích algoritmů ukazují, a v dalším to i uvidíme, že složitost těchto algoritmů podstatně ovlivňují jen operace, které potřebují jako svoje operandy (argumenty) prvky tříděné množiny. Jsou to operace dvou typů: porovnávání, které porovnává tříděné prvky, a přesuny, které prvky v rámci paměti určitým způsobem přesouvají.

Proto složitost asociativních třídících algoritmů budeme měřit jednak **počtem porovnání**, a jednak **počtem přesunů** tříděných prvků.

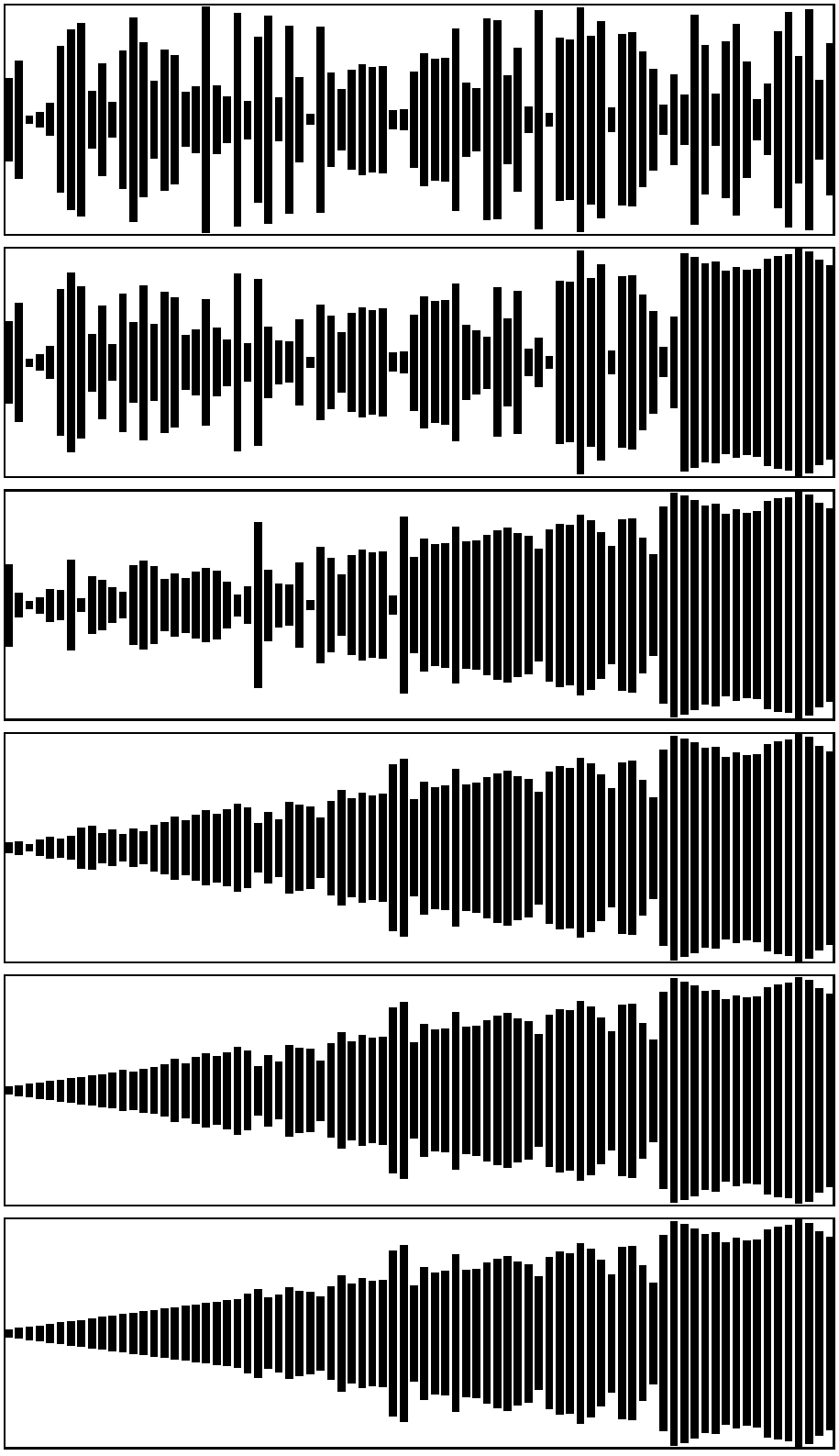
Všimněme si ještě, že porovnání se týká jenom samotných klíčů, přesuny se mohou týkat celých záznamů. Markantní rozdíl mezi cenou porovnání a cenou přesunu se projeví zvláště tehdy, když je délka záznamu podstatně větší než délka klíče.

Efektivnost některých asociativních třídících algoritmů závisí od toho, v jakém pořadí jsou uspořádané prvky ve vstupní posloupnosti. Když je například vstupní posloupnost setříděná, potom třídicí algoritmus, který to odhalí, nemusí dále nic dělat.

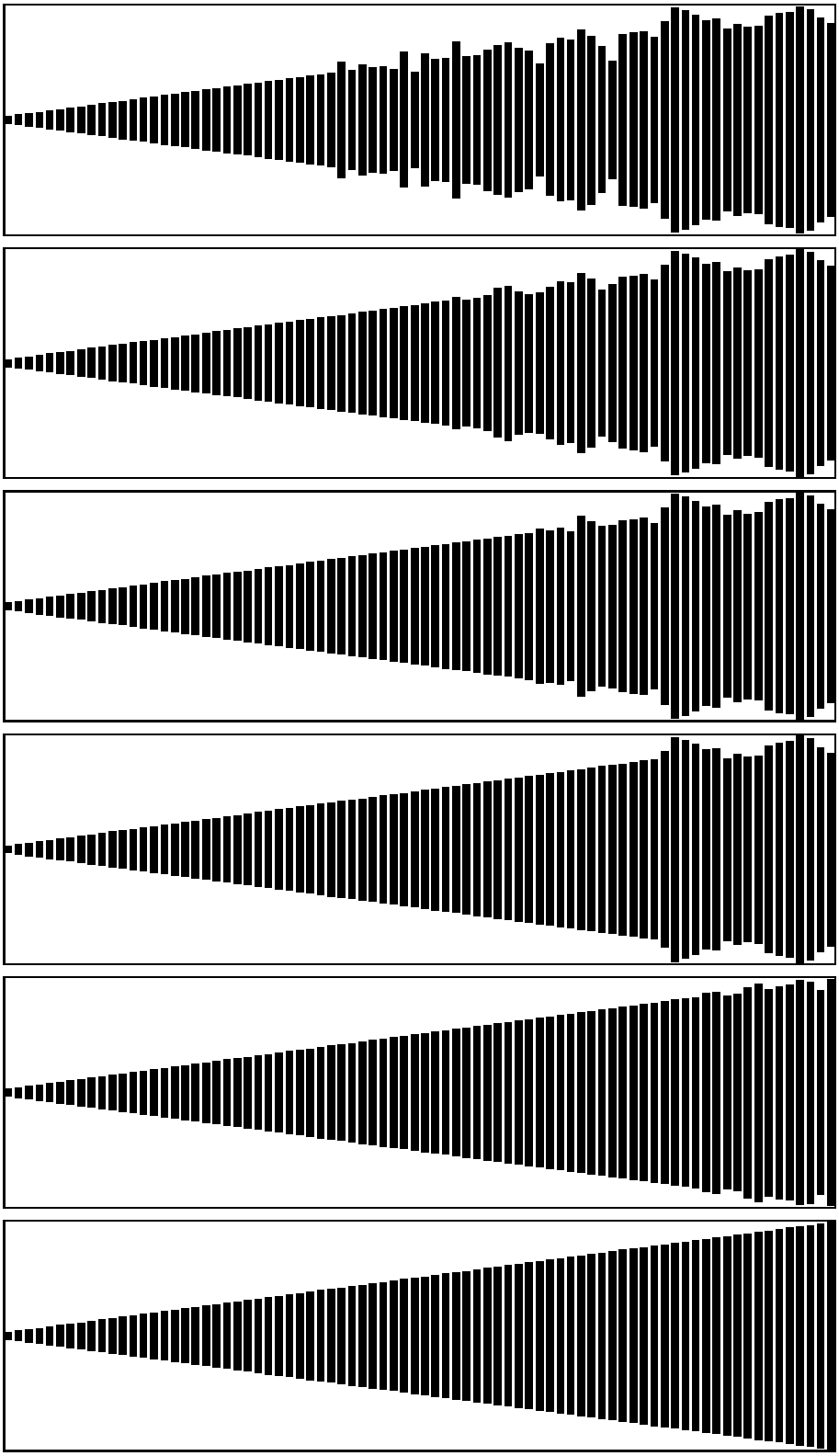
V dalším budeme za **míru setříděnosti** posloupnosti $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ považovat počet inverzí. Lehce zjistíme, že počet inverzí setříděné posloupnosti je 0, a maximální počet inverzí, rovný $\frac{n(n-1)}{2}$, obsahuje posloupnost setříděná v opačném pořadí.



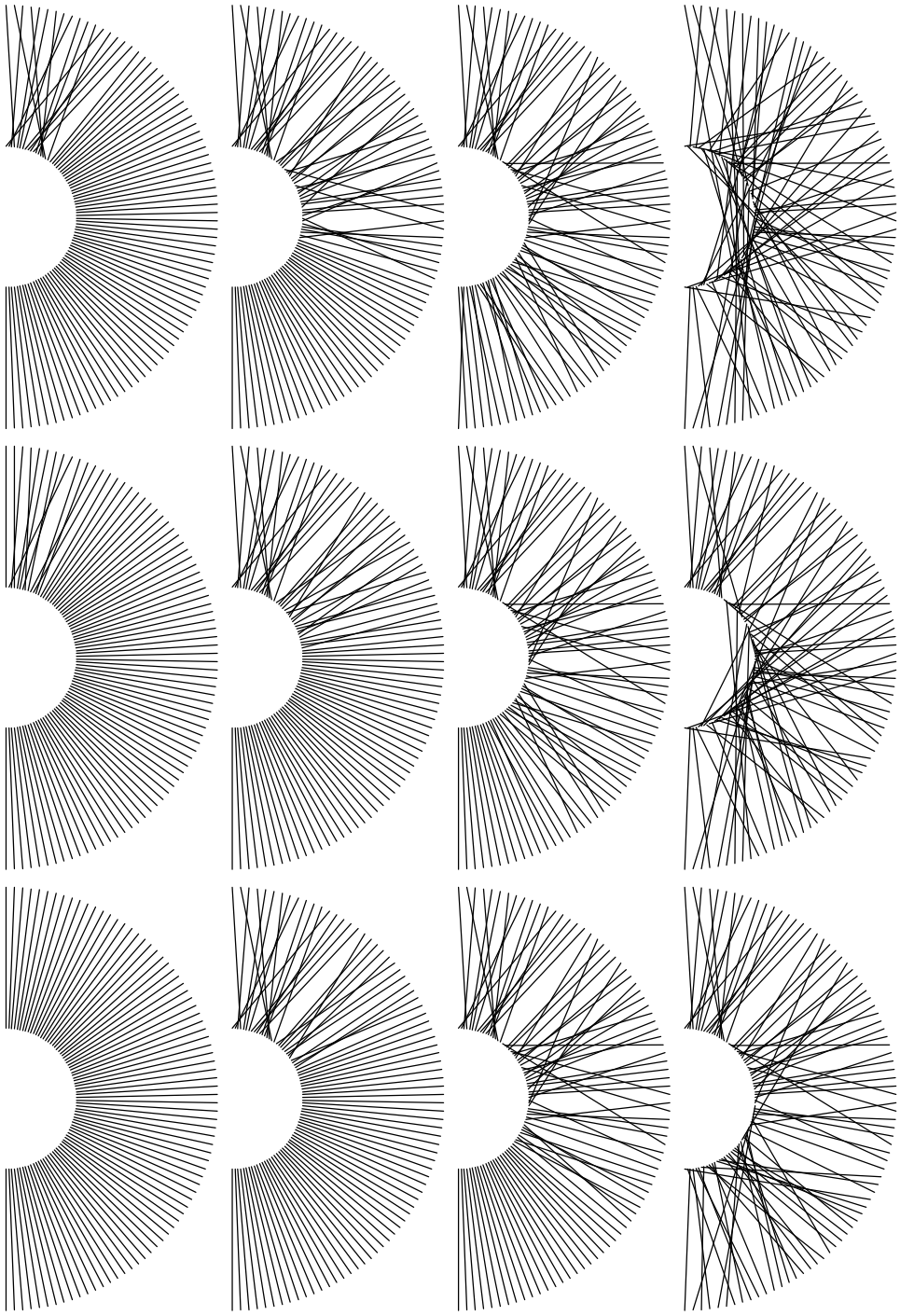
Obrázek 5.1: RadixSort — průběh třídění I



Obrázek 5.2: RadixSort — průběh třídění IIa



Obrázek 5.3: RadixSort — průběh třídění IIb



Obrázek 5.4: RadixSort – průběh třídění III

Průměrný počet inverzí v posloupnosti je přibližně $\frac{1}{4}n(n-1)$, za předpokladu, že každá permutace posloupnosti je stejně pravděpodobná.

Uvažujme všechny permutace n prvků $\{1, 2, \ldots, n\}$. Permutace n prvků vytvoříme přidáním n k $(n-1)!$ permutacím $n-1$ prvků $\{1, 2, \ldots, n-1\}$. Toto n můžeme přidat na poslední místo (pak se počet inverzí nezvýší), na předposlední místo (pak přibude jedna inverze) a tak dále, až na první místo (pak přibude $n-1$ inverzí). Pro průměrný počet inverzí I_n tedy bude platit

$$\begin{aligned} n!I_n &= (n-1)!I_{n-1} + 0 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + 1 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + 2 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + (n-1) \cdot (n-1)! \end{aligned}$$

Tedy

$$\begin{aligned} n!I_n &= n(n-1)!I_{n-1} + (0 + 1 + 2 + \cdots + (n-1)) \cdot (n-1)! \\ &= n!I_{n-1} + \frac{1}{2}n(n-1)(n-1)! \\ &= n!I_{n-1} + \frac{1}{2}(n-1)n! \end{aligned}$$

odkud

$$\begin{aligned} I_n &= I_{n-1} + \frac{1}{2}(n-1) \\ &\vdots \\ &= \frac{1}{2}(n-1) + \frac{1}{2}(n-2) + \frac{1}{2}(n-3) + \frac{1}{2} \cdot 1 \\ &= \frac{1}{4}n(n-1) \end{aligned}$$

Na algoritmy asociativního třídění se tedy můžeme dívat jako na algoritmy odstraňující postupně inverze ze vstupní posloupnosti pomocí vzájemných výměn vhodných prvků, až do té doby dokud posloupnost neobsahuje žádné inverze. Tento pohled bývá často užitečný při analýze a syntéze třídicích algoritmů.

Zkušenost ukazuje, že základní principy třídění porovnáváním je možno odvodit aplikací metody divide-et-impera na základní třídicí problém. Předpokladem využití této metody je možnost efektivního vykonání 1. a 3. kroku následujícího postupu:

1. krok - analýza: je-li rozsah problému konstantní, tak ho vyřeš přímo, jinak se redukuje na několik problémů stejného typu, ale menšího rozsahu;

2. krok - rekurze: problémy menšího rozsahu se řeší rekurzivně;

3. krok - syntéza: z řešení menších problémů se syntetizuje řešení původního problému.

V konkrétních případech, při využití této metody, máme na výběr následující extrémní možnosti, za předpokladu že problém rozkládáme vždy jen na dva podproblémy, a to buď:

a) vyváženě - oba menší problémy jsou přibližně stejného rozsahu;

b) nevyváženě - jeden z problémů má konstantní rozsah - nebo věnujeme více úsilí 1. kroku – rozkladu – tak, že krok – syntéza bude triviální, a nebo opačně.

5.4.1 Třídění vkládáním

Princip třídění přímým vkládáním se podobá metodě, jakou hráč karet obvykle seřazuje karty v ruce, když je po rozdání postupně bere ze stolu a vkládá je mezi již uspořádané karty, které má v ruce.

Nechť je pole tříděných položek rozděleno na část setříděnou (od indexu 0 do $i-1$) a na část nesetříděnou (od indexu i do $n-1$). Z nesetříděné části vybereme libovolný prvek a ten zařadíme do setříděné části tak, aby tato část zůstala setříděná. Tento postup opakujeme dokud není nesetříděná část prázdná.

Nalezne-li se index k ($0 \leq k \leq i$), pro nějž platí $a[k-1] \leq a[i] \leq a[k]$, pak se část od indexu k do $i-1$ posune o jednu pozici doprava a na uvolněnou pozici k se vloží zařazovaná položka s indexem i. Inicializace spočívá v rozdělení pole na dvě části: Prvek $a[0]$ tvoří setříděnou část a zbytek pole nesetříděnou část. Cyklus zařazování končí zařazením n-tého prvku.

Příklad:

```
44  55  12  42  94  18   6  67
44  55  12  42  94  18   6  67
12  44  55  42  94  18   6  67
12  42  44  55  94  18   6  67
12  42  44  55  94  18   6  67
12  18  42  44  55  94   6  67
 6  12  18  42  44  55  94  67
 6  12  18  42  44  55  67  94
```

```
void InsertSort (int a [], int n)
{
  int i, j, v;
  for(i = 0; i < n; i++)
  {
    v = a[i];
    j = i;
```

```
    while ((a[j−1] > v) && (j > 0))
    {
      a[j] = a[j−1];
      j−−;
    } // while
    a[j] = v;
  }; // for
} // InsertSort
```

Všimněme si, že pro $i \geq 2$ vnitřní cyklus algoritmu InsertSort proběhne právě tolikrát, kolik má prvek $a[i]$ inverzí. To znamená, že celková složitost tohoto algoritmu je úměrná počtu inverzí vstupní permutace. Algoritmus se chová přirozeně v tom smyslu, že jeho složitost plynule roste s mírou neutříděnosti vstupní permutace a to je jeho velká výhoda.

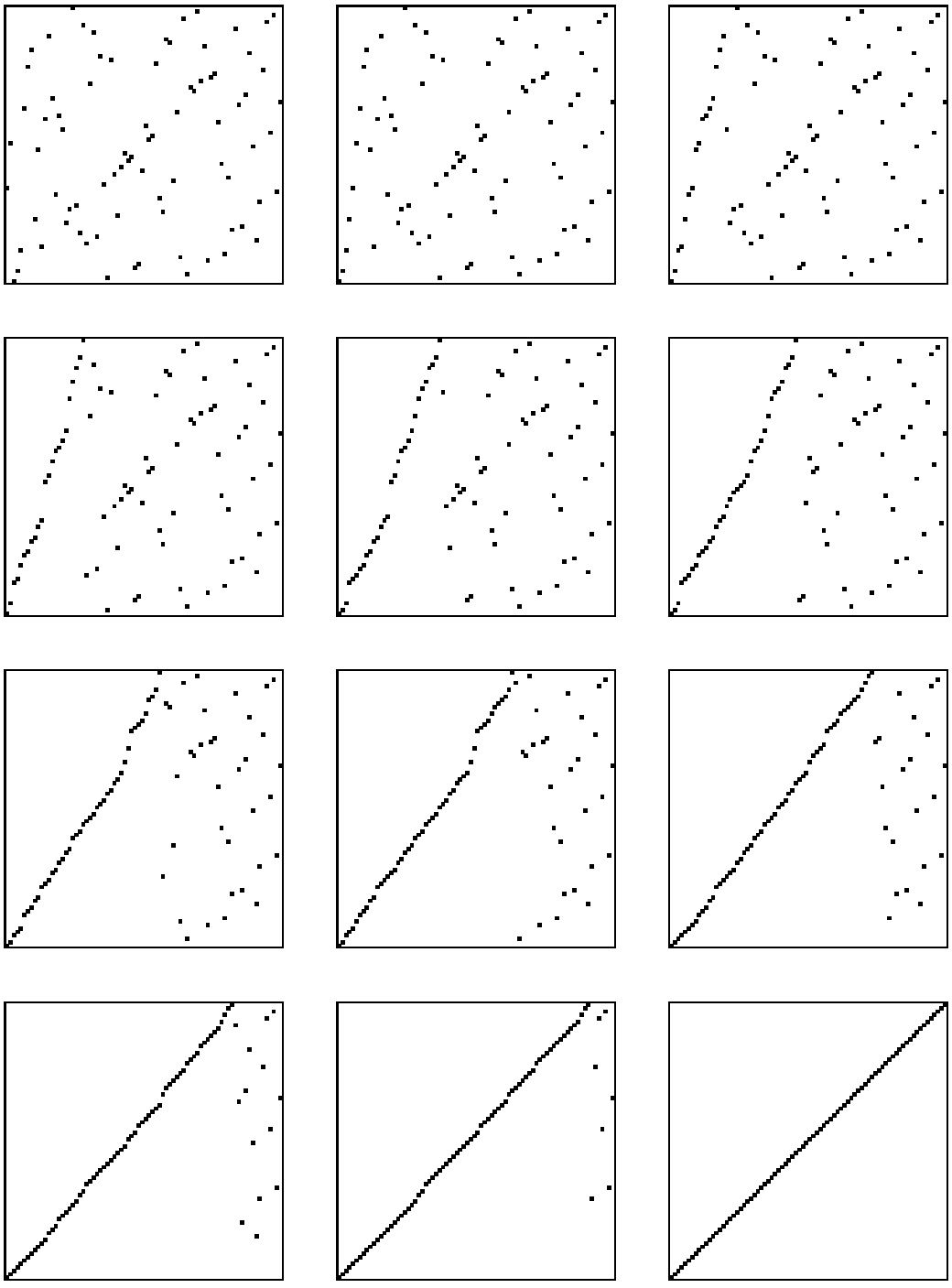
V předchozí části byla zmíněna možnost třídit pouze ukazatele na záznamy, pokud je velikost záznamu mnohem větší než velikost klíče. V tomto případě by výměna (kopírování) záznamů zabralo nejvíce času z celého algoritmu. V následující ukázce kódu je použito pole ukazatelů p na prvky v poli a jsou indexovány prostřednictvím tohoto pole.

```
void InsertSort (int a [], int n)
{
  int i, j, v;
  for(i = 0; i < n; i++)
    p[i] = i;
  for(i = 0; i < n; i++)
  {
    v = p[i];
    j = i;
    while ((a[p[j−1]] > a[v]) && (j > 0))
    {
      p[j] = p[j−1];
      j−−;
    }; // while
    p[j] = v;
  }; // for
} // InsertSort
```

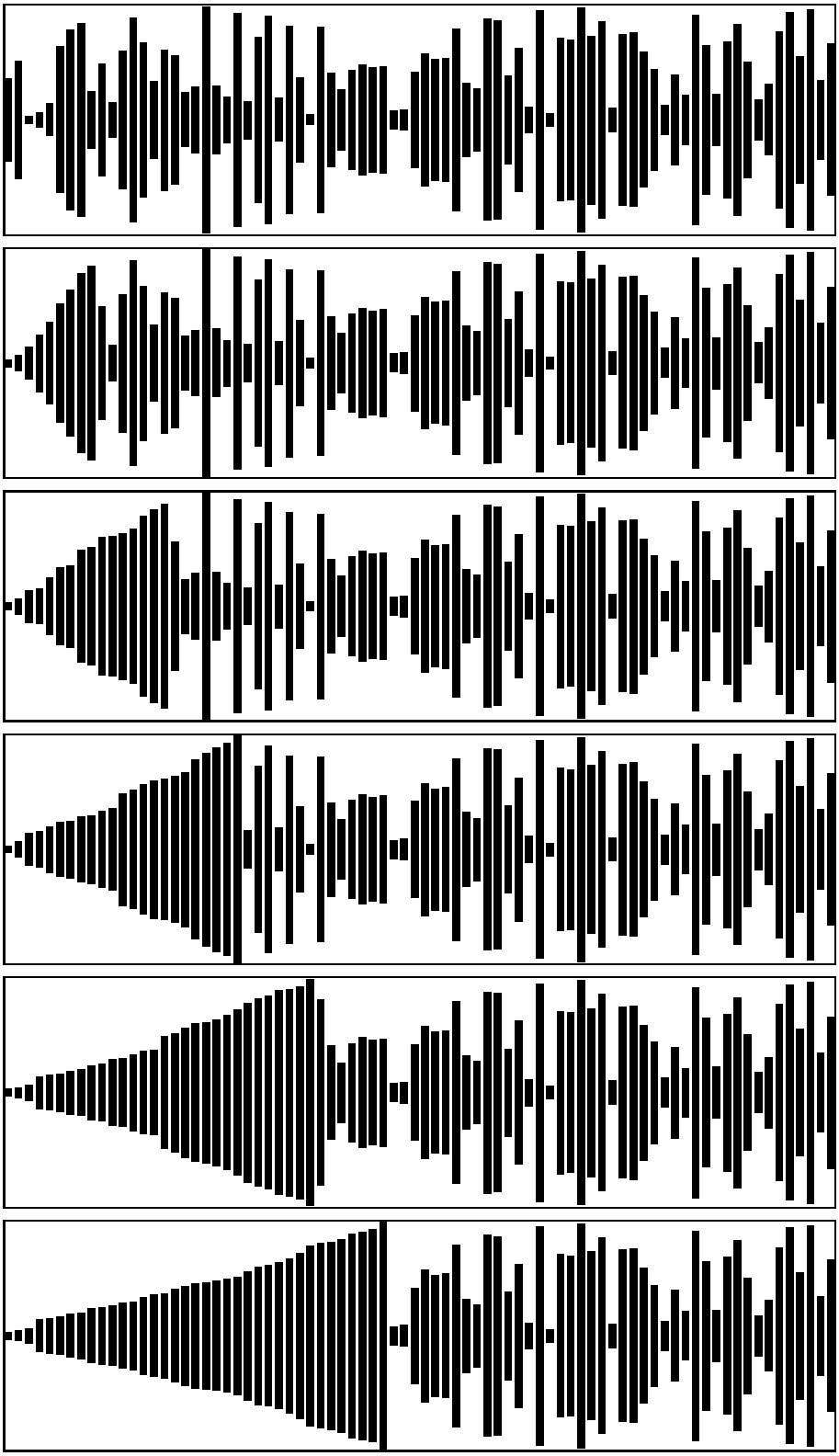
Analýza

Počet porovnání klíčů C_i v i-tém průchodu je nejvíce $i - 1$ a nejméně 1. Za předpokladu, že všechny permutace n klíčů jsou stejně pravděpodobné, můžeme C_i v průměru pokládat $i/2$. Počet přesunů M_i je roven C_i. Celkový počet porovnání a přesunů potom bude:

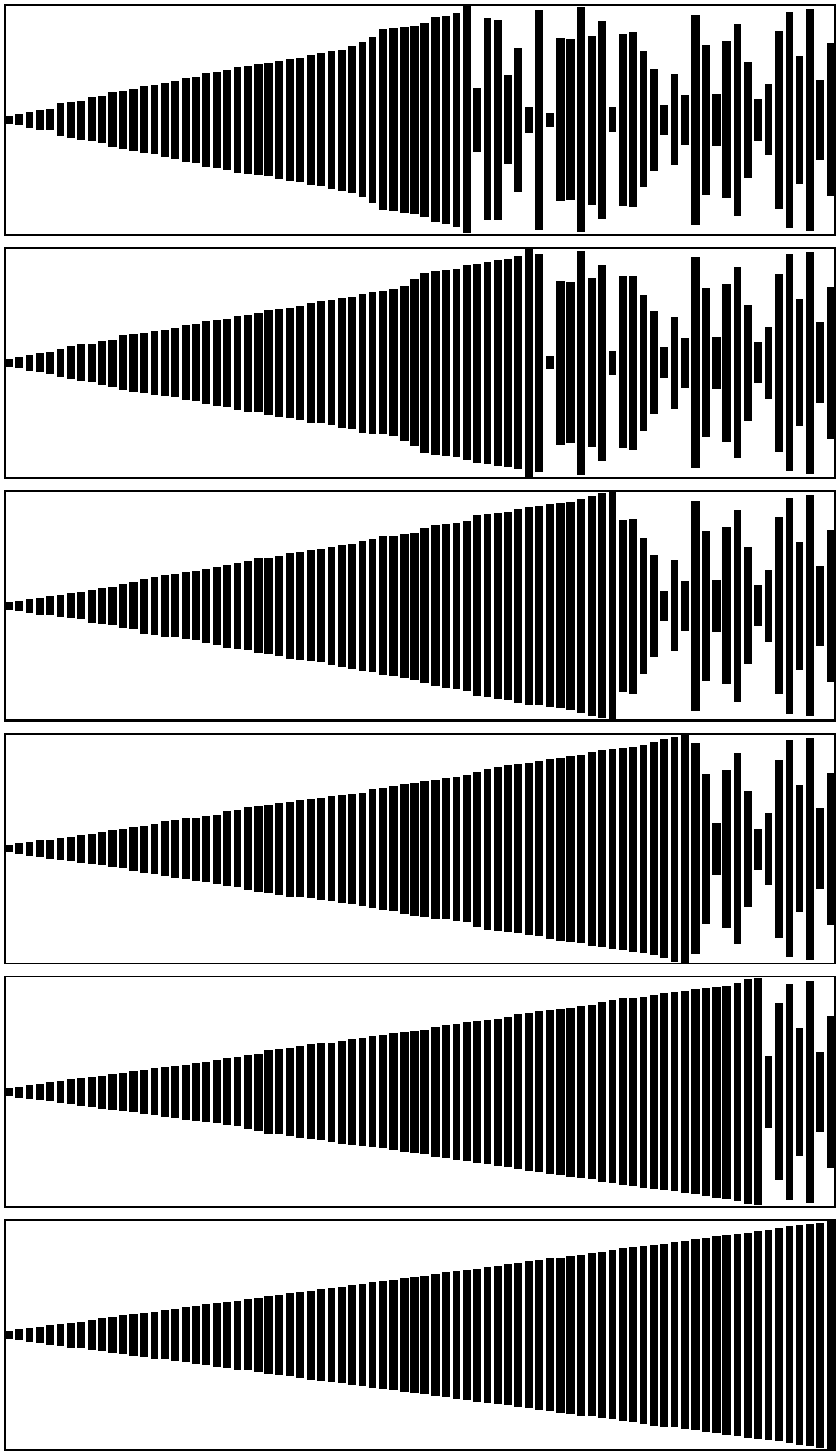
$$\begin{aligned} C_{min} &= n - 1 \\ C_{avg} &= \frac{1}{4}(n^2 + n + 2) \\ C_{max} &= \frac{1}{2}(n^2 + n) - 1 \end{aligned}$$



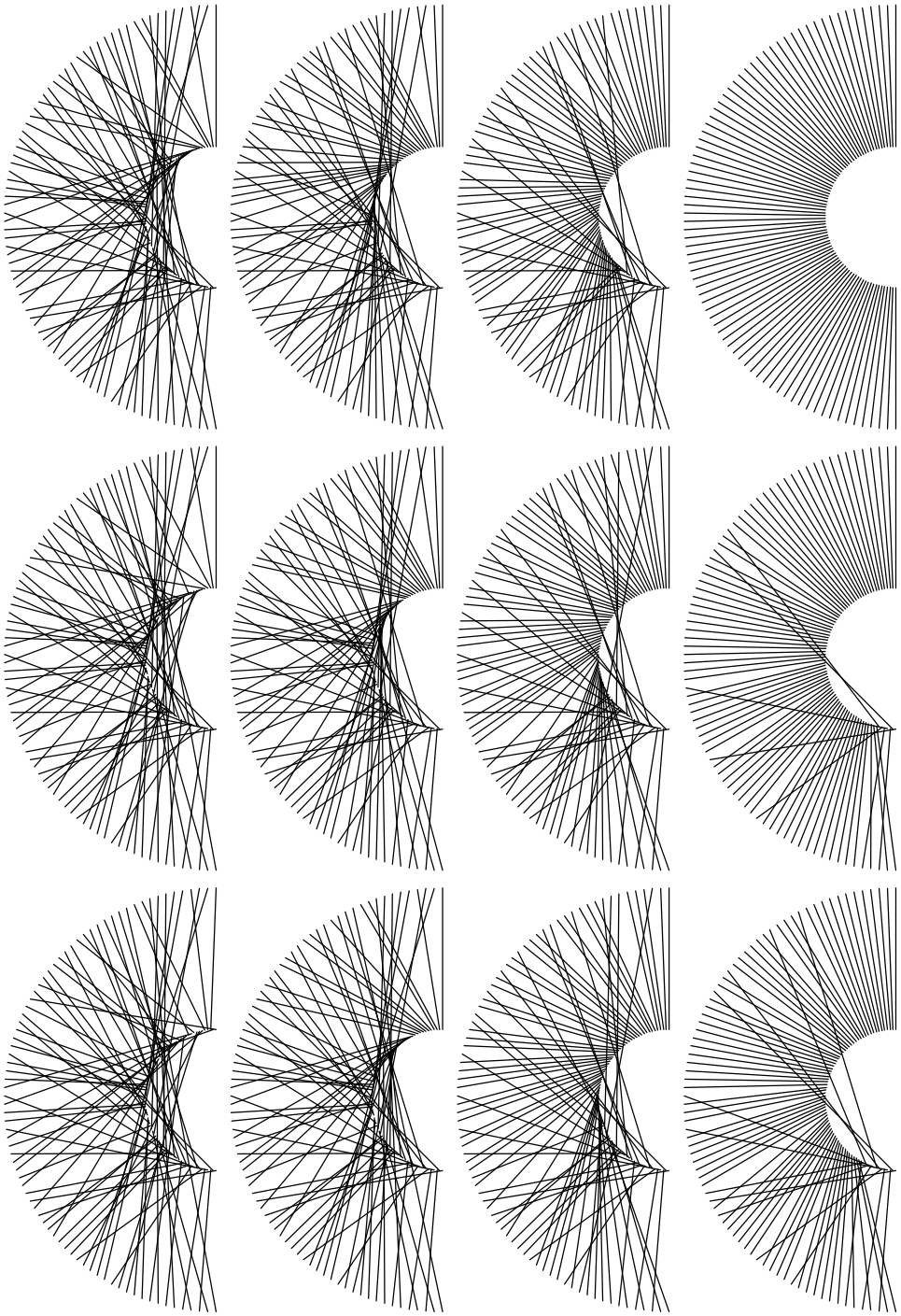
Obrázek 5.5: InsertSort – průběh třídění I



Obrázek 5.6: InsertSort – průběh třídění IIa



Obrázek 5.7: InsertSort – průběh třídění IIb



Obrázek 5.8: InsertSort – průběh třídění III

$$\begin{aligned} M_{min} &= 2(n-1) \\ M_{avg} &= \frac{1}{4}(n^2+9n-10) \\ M_{max} &= \frac{1}{2}(n^2+3n-4) \end{aligned}$$

Nejmenší hodnoty C a M nastávají v případě, že zdrojová posloupnost je setříděna. Tento případ nazýváme nejlepší. Opakem je nejhorší případ, který nastane v okamžiku, kdy zdrojová posloupnost je setříděna v obráceném pořadí. Z tohoto plyne, že třídění vkládáním je přirozené. Dále je jasné, že toto třídění je i stabilní.

5.4.2 Třídění vkládáním s ubývajícím krokem

ShellSort Z popisu algoritmu třídění vkládáním a z definice inverze je jasné, že každou výměnou sousedních prvků se sníží celkový počet inverzí přesně o 1. To je také důvod, proč je složitost těchto algoritmů v nejhorším a průměrném případě kvadratická. Je jasné, že výměnou prvků ležících dále od sebe, by počet inverzí rychleji klesal k nule.

Jednoduchý, ale přitom geniální algoritmus popsal D. L. Shell v roce 1959, když navrhl využít třídění vkládáním ve více chodech. V $i-$tém chodu se třídí prvky ležící ve vzdálenosti h_i , pro $i = t, t-1, \ldots, 0$, $h_{i+1} > h_i, h_1 = 0$, $t > 0$. Číslo h_i se nazývá i-tý krok metody. Takto dostaneme v závislosti od volby kroků celou třídu třídících algoritmů. Tyto algoritmy fungují efektivně, protože v počátečních chodech se třídí relativně krátké posloupnosti a v dalších chodech se třídí delší, ale utříděnější posloupnosti.

Příklad:

```
44  18  12  42  94  55   6  67
44  18   6  42  94  55  12  67
18  44   6  42  94  55  12  67
 6  18  44  42  94  55  12  67
 6  18  42  44  94  55  12  67
 6  18  42  44  55  94  12  67
 6  12  18  42  44  55  94  67
 6  12  18  42  44  55  67  94
```

```
void ShellSort(int a[], int n)
{
  int i, j, h, v;
  for(h = 1; h <= n; h = 3*h+1);
  do
  {
    h = h / 3;
    for(i = h; i < n; i++)
    {
      v = a[i];
      j = i;
```

```
      while ((a[j - h] > v) && (j >= h))
      {
        a[j] = a[j-h];
        j -= h;
      }; // while
      a[j] = v;
    }; // for
  } while (h != 0);
} // ShellSort
```

Intuitivně je zřejmé, že složitost Shellova algoritmu bude záviset na volbě posloupnosti kroků. Mezi nejznámější návrhy patří tyto posloupnosti kroků:

A: $h_1 = 1$, $h_{i+1} = 2 * h_i + 1$

B: $h_1 = 1$, $h_2 = 3$, $h_{i+1} = 2 * h_i - 1$ (pro $i > 2$)

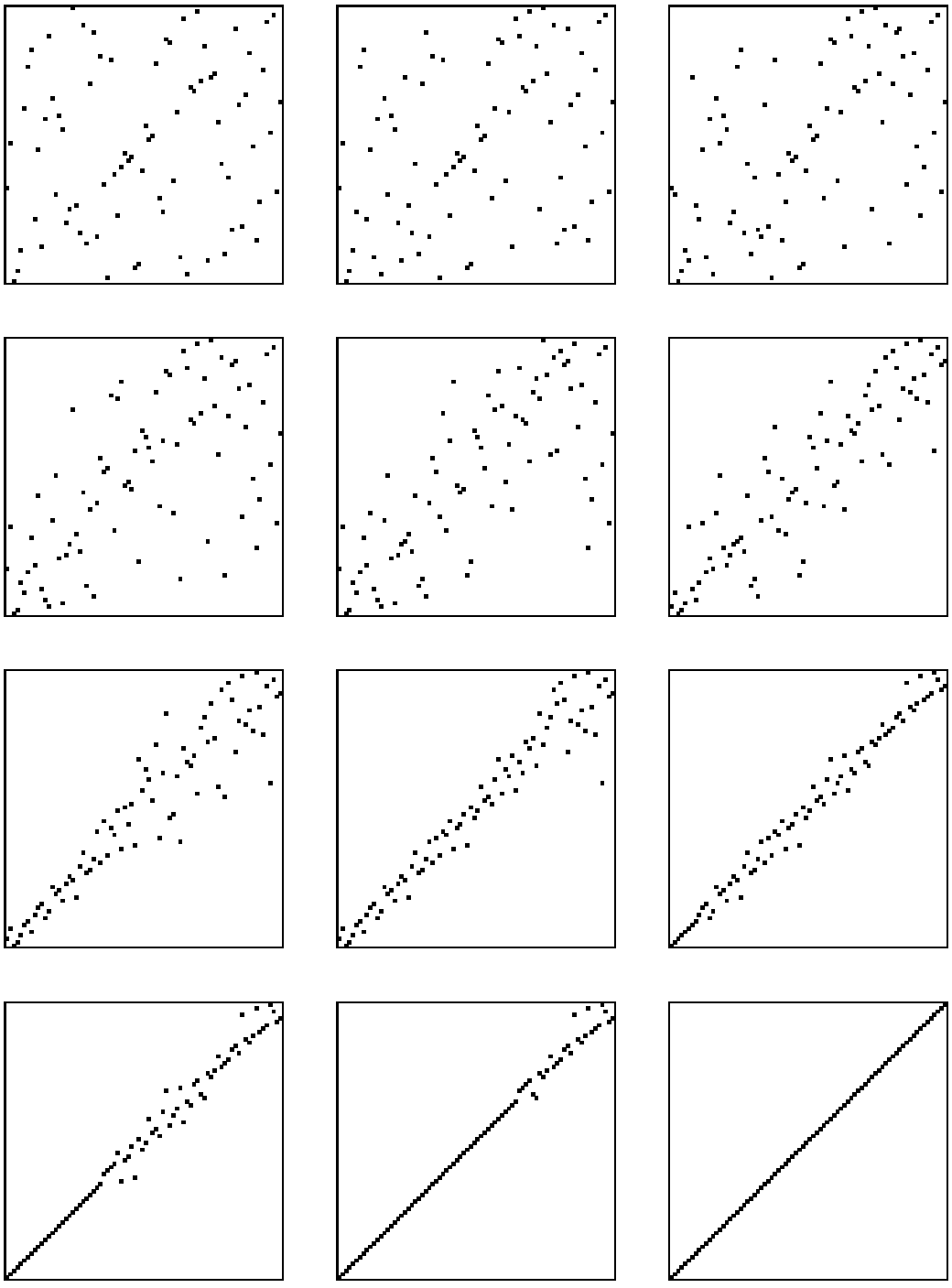
C: $h_1 = 1$, $h_{i+1} = 3 * h_i + 1$

V našem příkladu byla použita posloupnost kroků podle schématu C. Podrobná matematická analýza volby optimální posloupnosti však patří k nevyřešeným problémům. Jsou známé jen některé částečné výsledky.

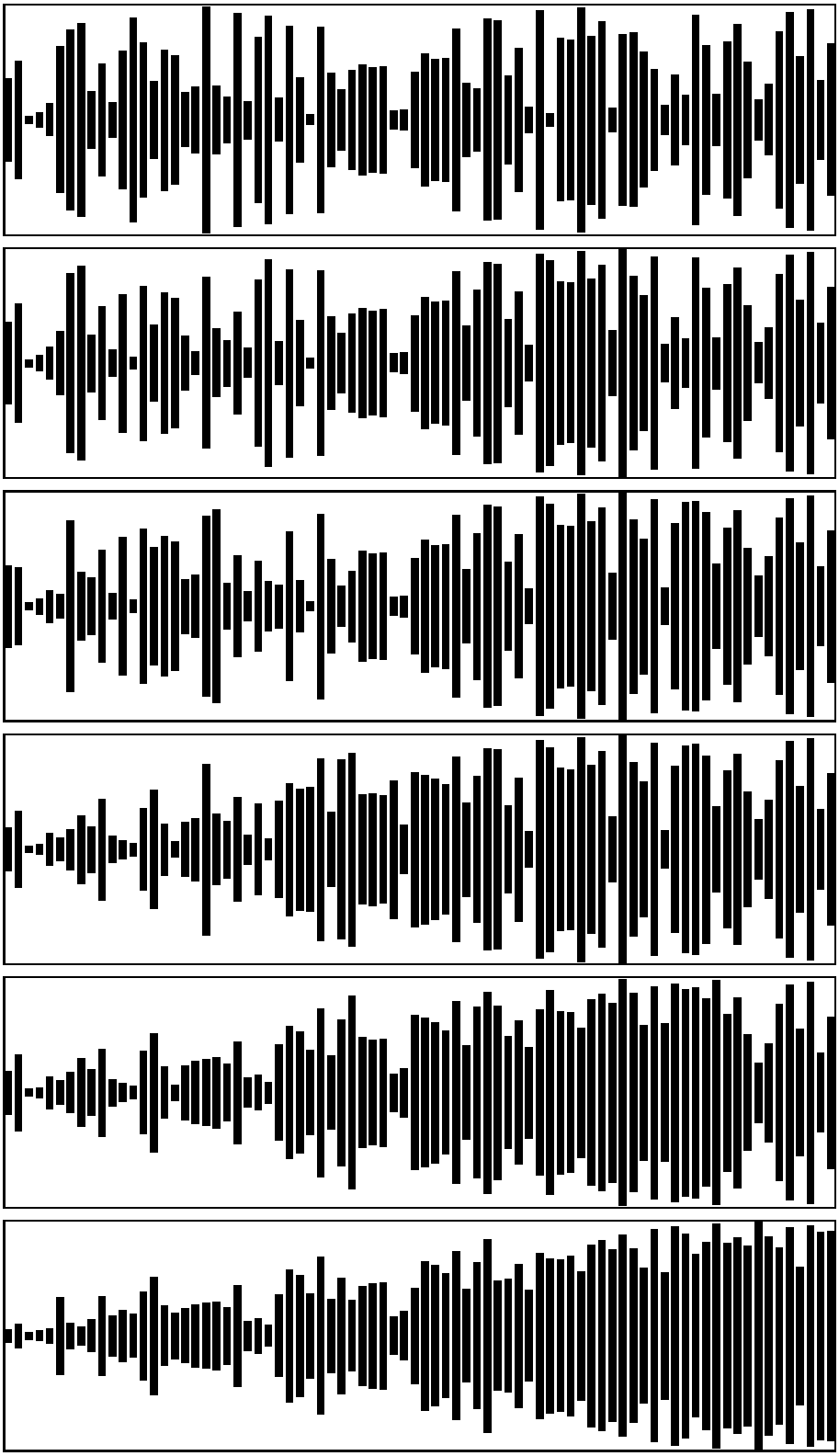
5.4.3 Třídění binárním vkládáním

Vrátíme se ještě k jednoduchému třídění vkládáním. Každého asi ihned napadne, že pozice prvku $a[i]$ v poli $a[1 \ldots i-1]$ se dá efektivně určit pomocí binárního vyhledávání, čímž dostaneme metodu **třídění binárním vkládáním**.

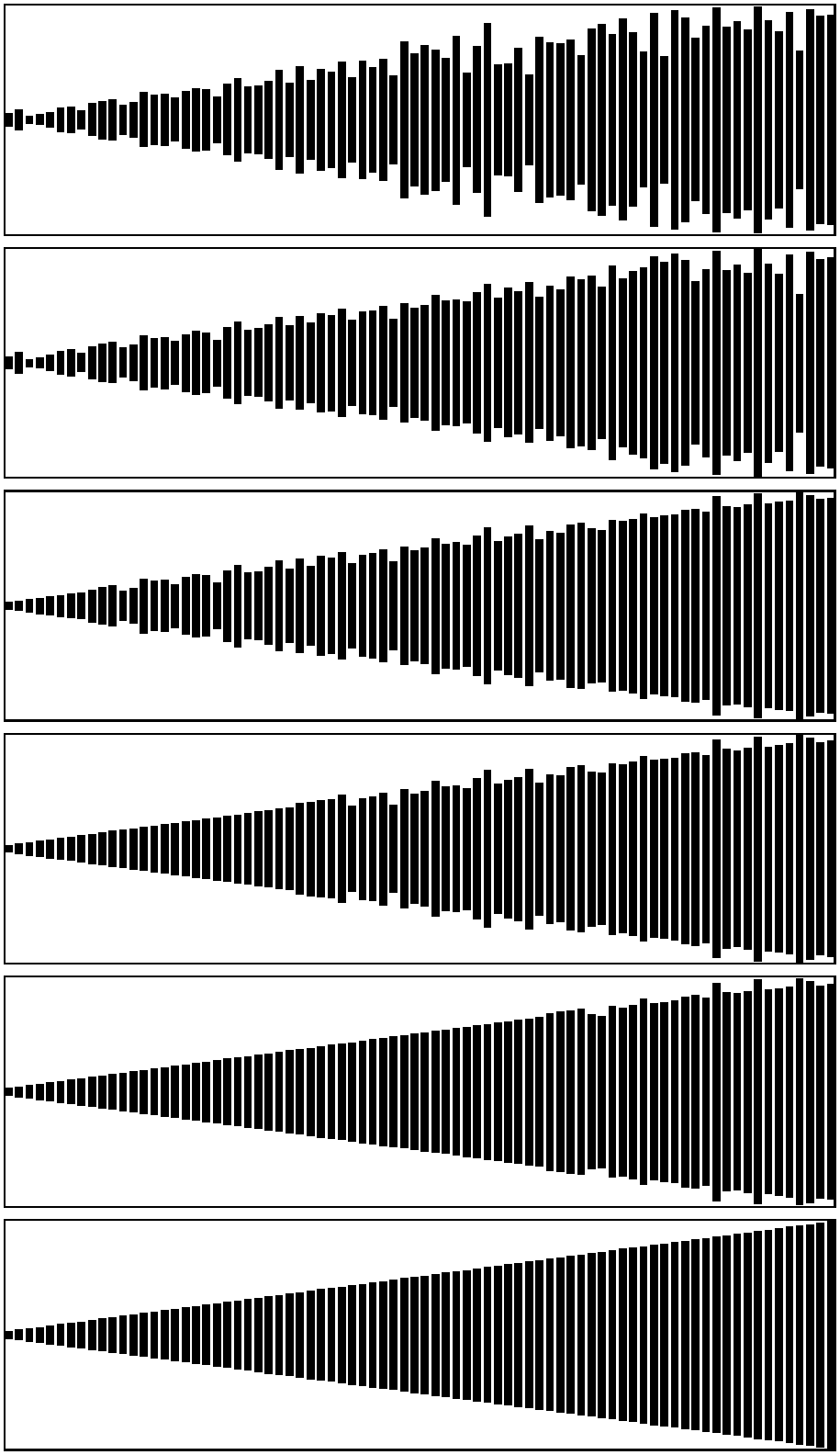
```
void BinaryInsertSort (int a [], int n)
{
  int i, j, v, l, r, m;
  for(i = 1; i < n; i++)
  {
    v = a[i];
    l = 0;
    r = i;
    while (l < r)
    {
      m = (l + r) / 2;
      if (a[m] <= v)
        l = m + 1;
      else
        r = m;
    }; // while
    for(j = i; j > r; j--)
      a[j] = a[j-1];
    a[r] = v;
  }; // for
} // BinaryInsertSort
```



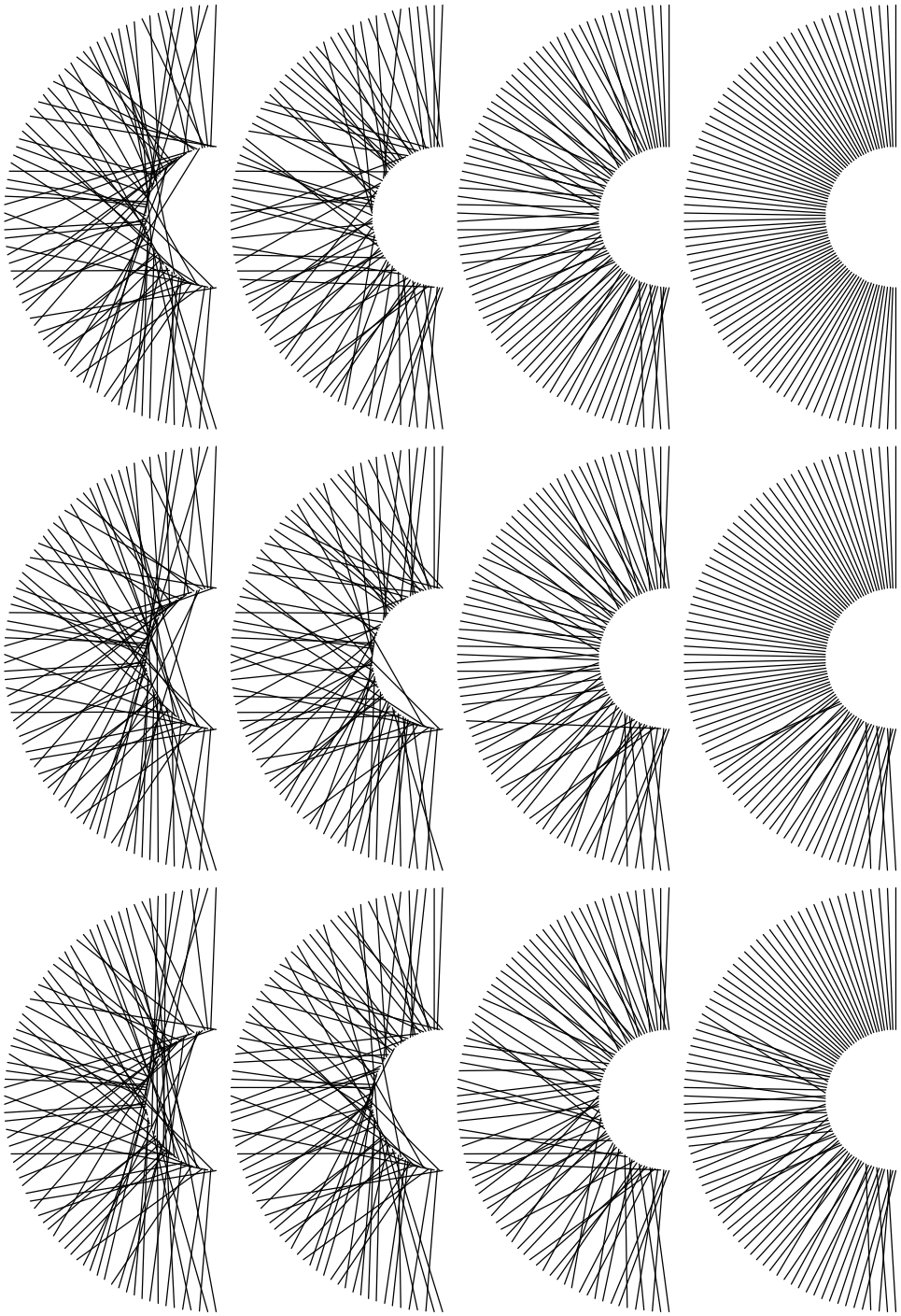
Obrázek 5.9: ShellSort – průběh třídění I



Obrázek 5.10: ShellSort – průběh třídění IIa



Obrázek 5.11: ShellSort – průběh třídění IIb



Obrázek 5.12: ShellSort – průběh třídění III

Analýza

Místo pro uložení prvku se najde tehdy, pokud platí $a_j \leq x < a_{j+1}$ tj. zkoumaný interval má délku 1. Interval skládající se z i klíčů se rozpůlí $\lceil \log i \rceil$ krát. Počet porovnání potom bude

$$C = \sum_{i=1}^{n} \lceil \log i \rceil$$

Aproximací této sumy pomocí integrálu dostáváme:

$$\int_1^n \log x dx = \Big[x(\log x - c)\Big]_1^n = n(\log n - c) + c$$

kde $c = \log e = 1/\ln 2 = 1,44269\ldots$.

Počet porovnání je v podstatě nezávislý na počátečním uspořádání prvků. Bohužel vylepšení algoritmu binárním vyhledáváním se týká pouze počtu porovnání nikoliv počtu potřebných přesunů prvků. Uvedeným vylepšením algoritmu se výrazně nevylepší hodnota M: tato zůstává i nadále řádu n^2. Tento příklad ukazuje, že často může dojít k situaci, kdy přirozené vylepšení algoritmu má nakonec menší efekt, než se původně očekávalo a v některých případech může dojít i ke zhoršení.

5.4.4 Třídění výběrem

Při **třídění výběrem** viděném z pohledu metody divide-et-impera, se podstatná činnost vykoná při dekompozičním kroku algoritmu. Tříděná posloupnost se rozkládá na jednoprvkovou množinu a zbytek tak, že jednoprvková množina obsahuje minimální (nebo maximální) prvek. Zbývající posloupnost se rekurzivně třídí dále. Algoritmus tedy vrací postupně, jako výsledek každého rekurzivního volání, posloupnost setříděných prvků v klesajícím (nebo rostoucím) pořadí.

Příklad:

```
44  55  12  42  94  18   6  67
 6  55  12  42  94  18  44  67
 6  12  55  42  94  18  44  67
 6  12  18  42  94  55  44  67
 6  12  18  42  94  55  44  67
 6  12  18  42  44  55  94  67
 6  12  18  42  44  55  94  67
 6  12  18  42  44  55  67  94
```

```
void SelectSort(int a[], int n)
{
  int i, j, min, t;
  for(i = 0; i < n; i++)
  {
```

```
      min = i;
      for(j = i+1; j < n; j++)
        if (a[j] < a[min])
          min = j;
      t = a[min];
      a[min] = a[i];
      a[i] = t;
    }; // for
  } // SelectSort
```

Analýza

Je zřejmé, že počet porovnání C nezávisí na počátečním uspořádání. V tomto smyslu je tato metoda méně přirozená než třídění vkládáním. Počet porovnání C je

$$C = \frac{1}{2}(n^2 - n)$$

Počet přesunů M je minimálně $M_{min} = 3(n-1)$ jestliže jsou klíče již uspořádané a maximálně

$$M_{max} = \left\lceil \left(\frac{n^2}{4} \right) \right\rceil + 3(n-1)$$

pokud jsou klíče setříděny opačně. Průměrný počet přesunů M_{avg} se dá těžko určit i přes jednoduchost algoritmu. Závisí na tom, kolikrát se najde klíč k_j menší než všechny předcházející klíče $k_1, \ldots, k_n$. Tato hodnota se bere jako průměr všech $n!$ permutací n klíčů a je určena vztahem $H_n - 1$, kde H_n je n-té harmonické číslo (viz kapitola HarmonickaCisla). Pro dostatečně velké n můžeme zanedbat zlomkové části a průměrný počet přiřazení v i-tém průchodu aproximovat jako

$$F_i = \ln i \ + \gamma + 1$$

Průměrný počet přesunů M_{avg} při třídění výběrem je potom dán sumou F_i pro $i = 1 \ldots n$.

$$M_{avg} = \sum_{i=1}^{n} F_i = n(\gamma + 1) + \sum_{i=1}^{n} \ln i$$

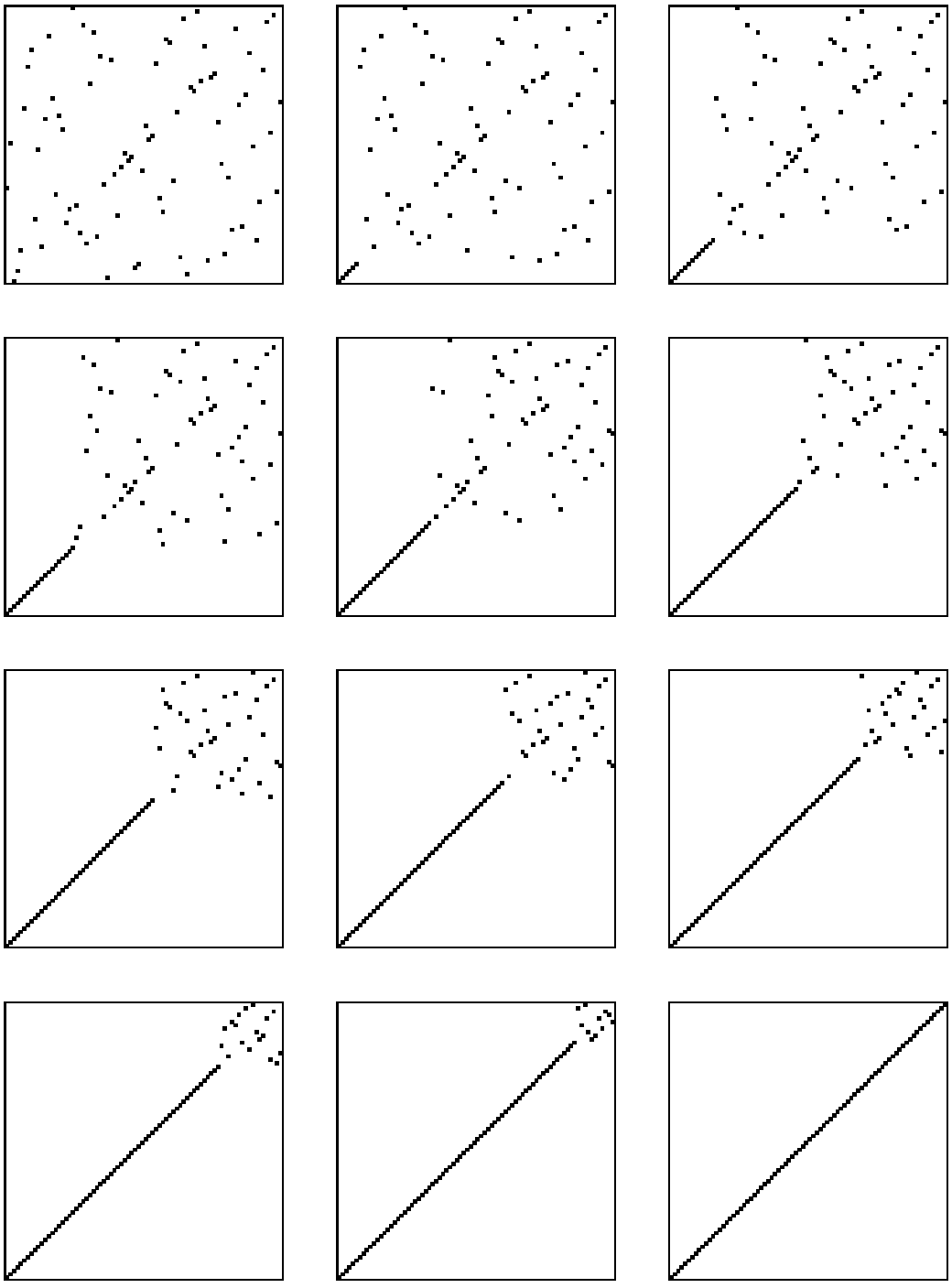
Další aproximací sumy diskrétních výrazů pomocí integrálu

$$\int_1^n \ln x dx = \Big[x(\ln x - 1) \Big]_1^n = n \ln n - n + 1$$

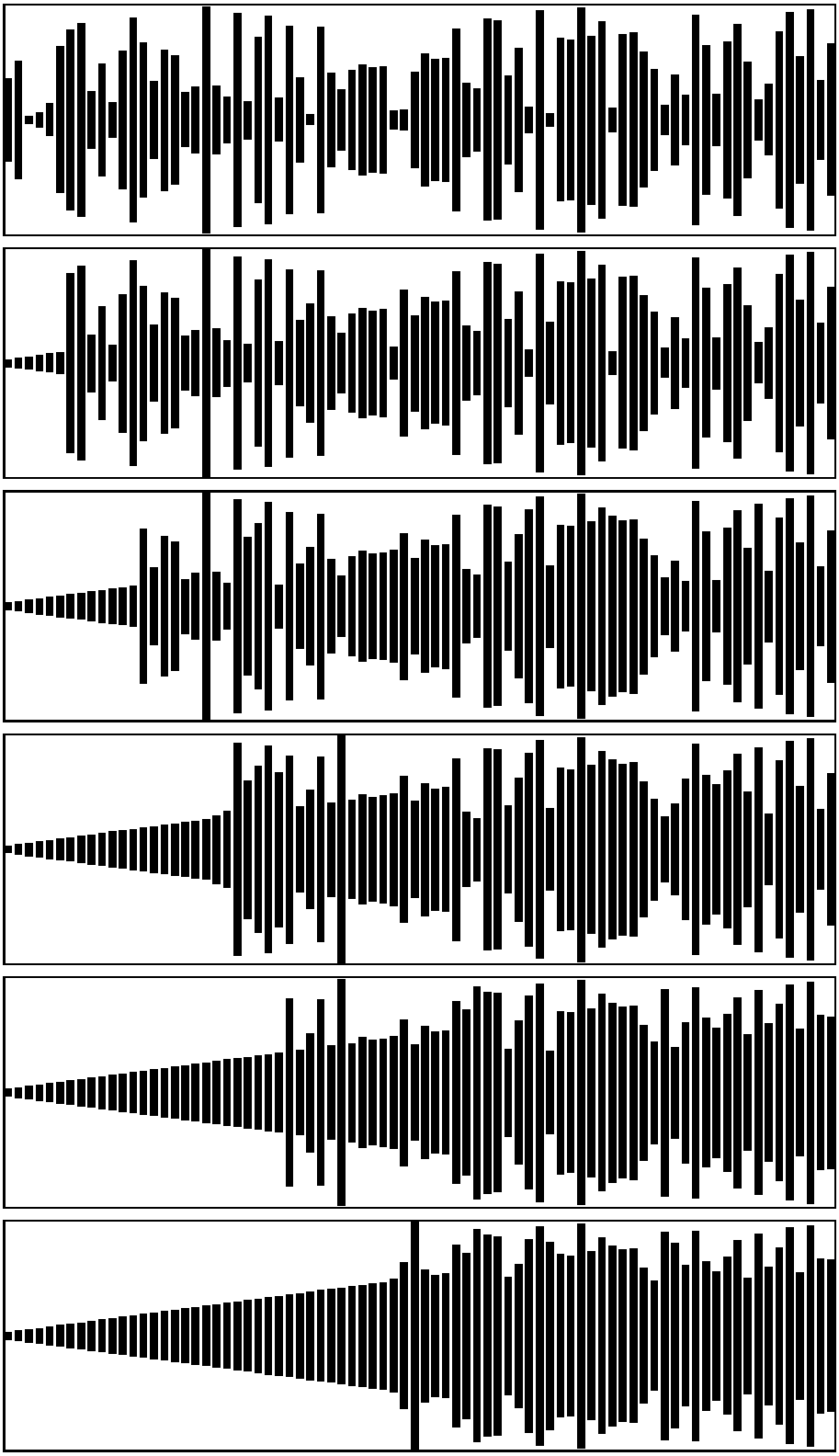
dostáváme přibližnou hodnotu

$$M_{avg} \doteq n(\ln n + \gamma)$$

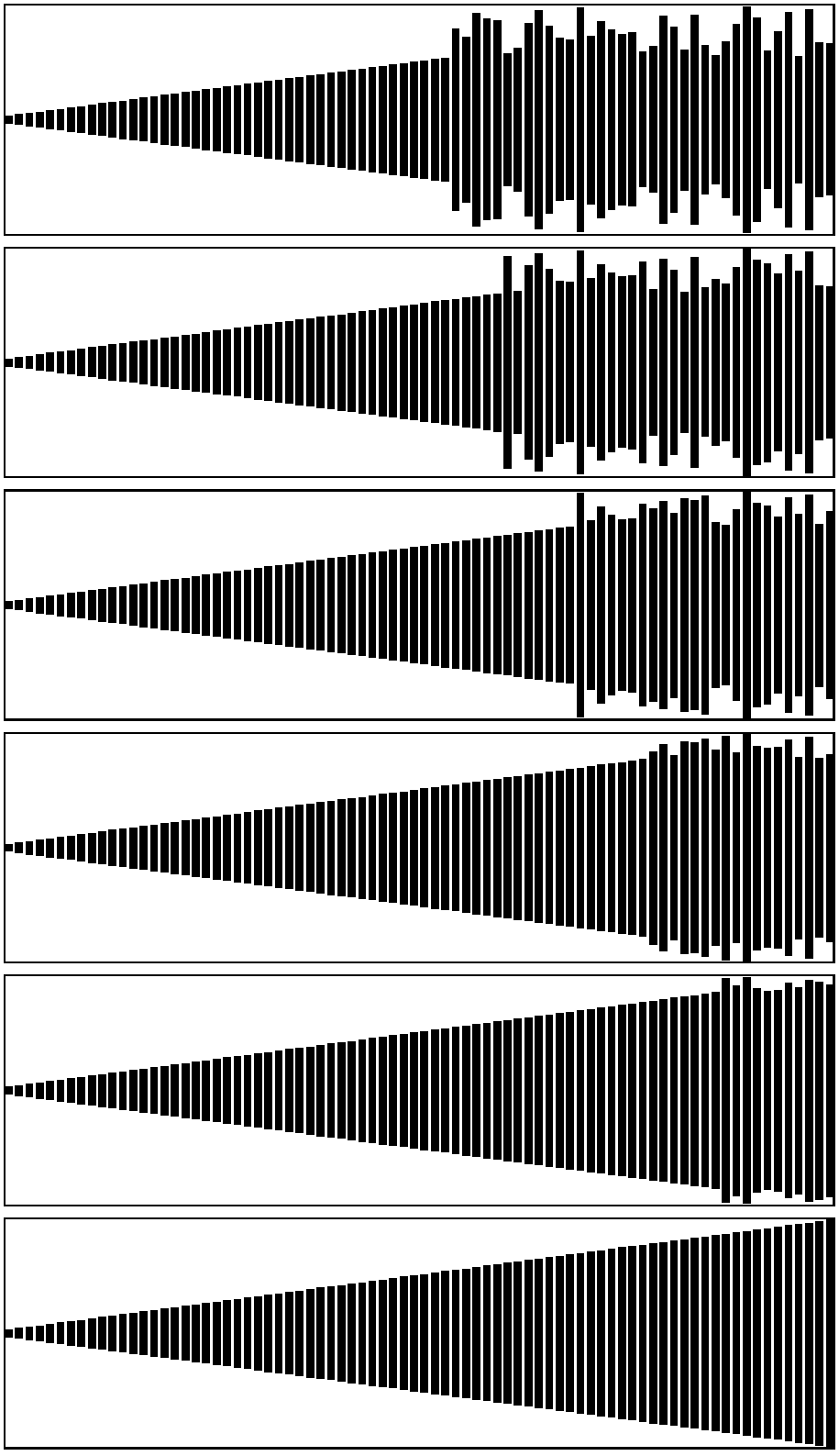
Závěrem můžeme konstatovat, všeobecně je třídění výběrem je efektivnější než třídění vkládáním, kromě případu, že zdrojová posloupnost je setříděná nebo téměř setříděná.



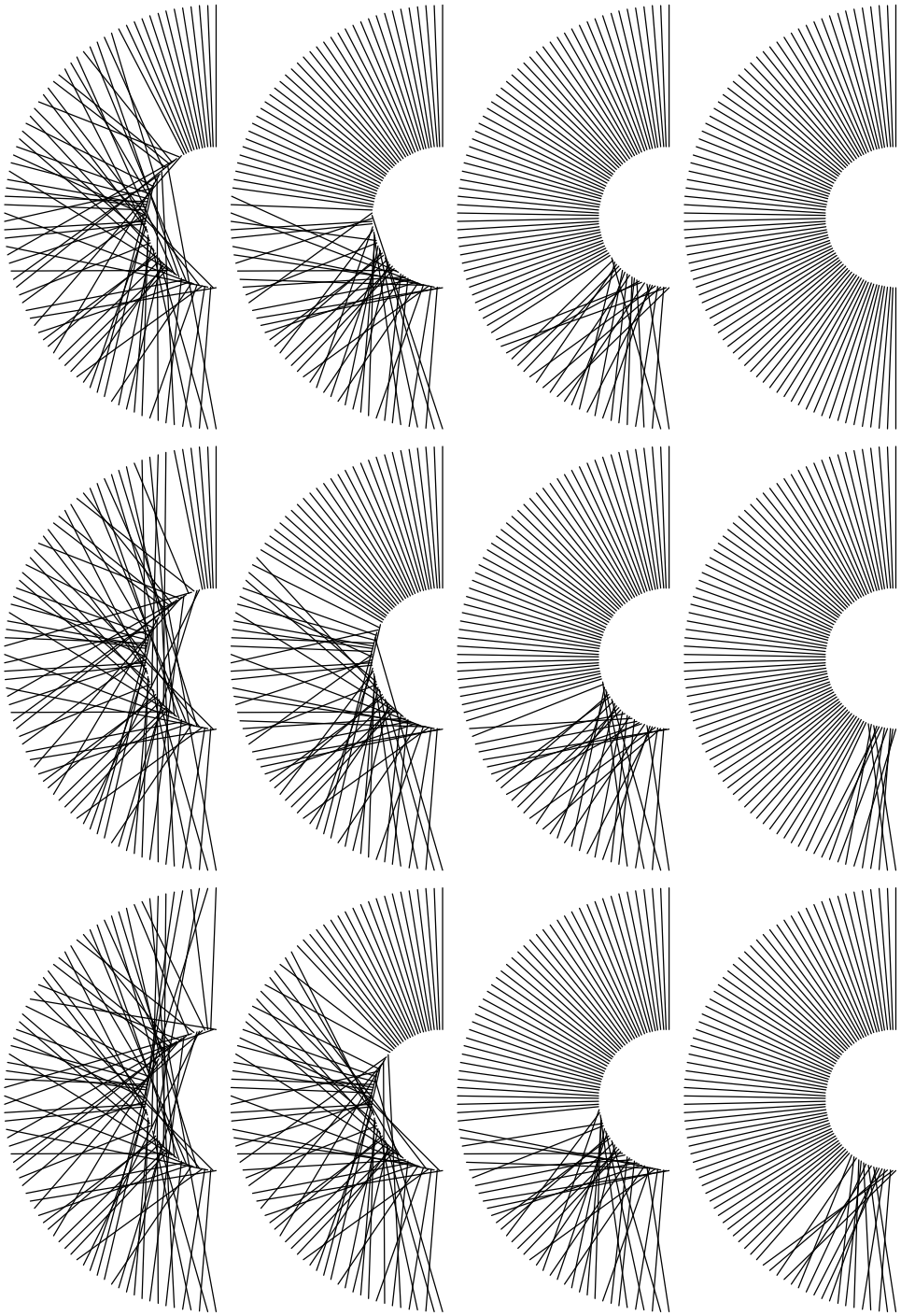
Obrázek 5.13: SelectSort – průběh třídění I



Obrázek 5.14: SelectSort – průběh třídění IIa



Obrázek 5.15: SelectSort – průběh třídění IIb



Obrázek 5.16: SelectSort – průběh třídění III

5.4.5 Bublinkové třídění

V předchozím algoritmu můžeme vnitřní cyklus – výběr minimálního prvku z $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ – naprogramovat také tak, že postupně porovnáváme a_j s a_{j+1}, pro $j = 1, \ldots, i-1$. Jestliže testovaná dvojice není uspořádaná, potom vyměníme pozice testovaných prvků. Tak se stane, že maximální prvek posloupnosti $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ „probublá“ na i-tou pozici. Takový algoritmus je označován pojmem **bublinkové třídění**.

Příklad:

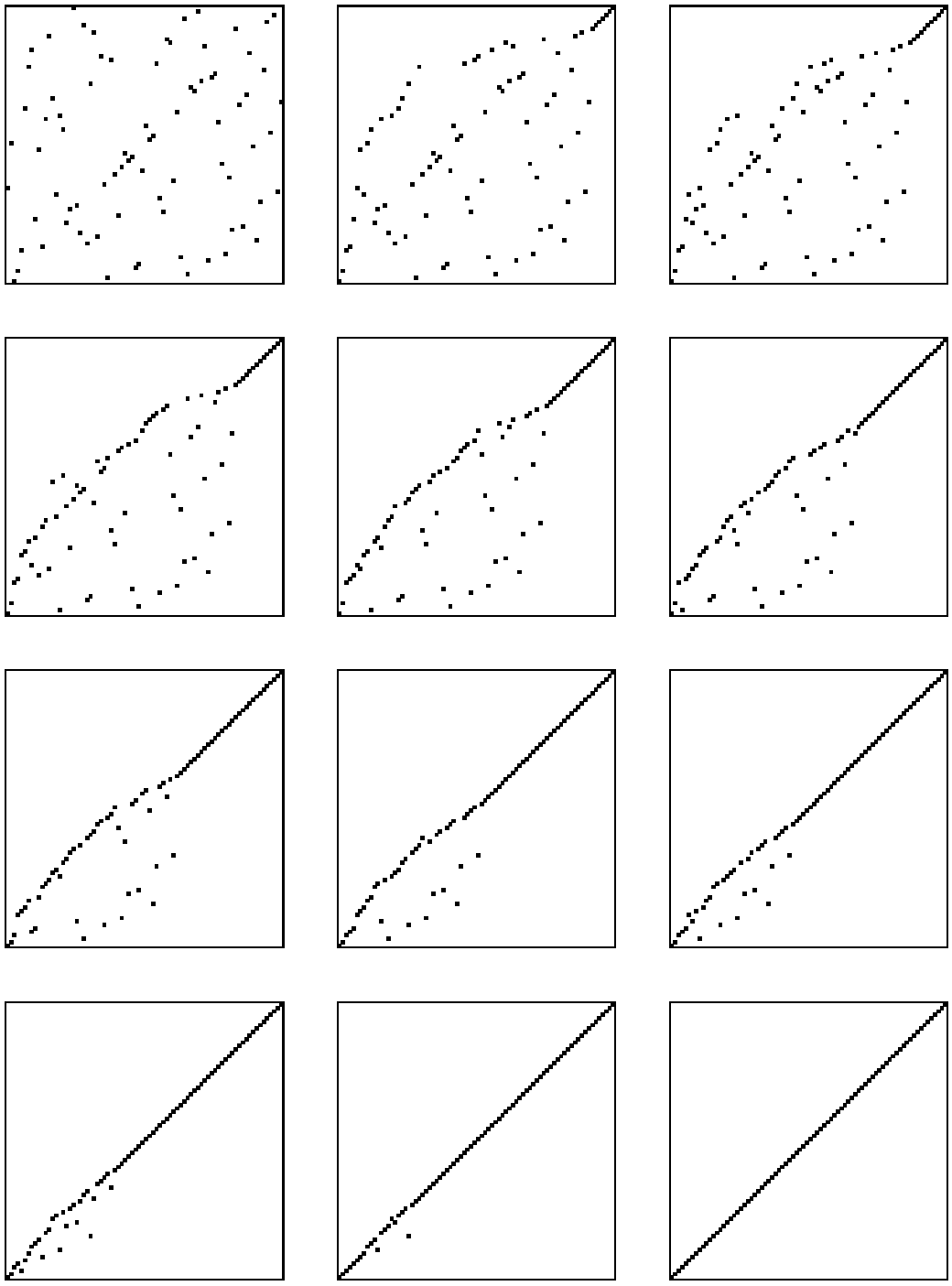
```
44  55  12  42  94  18   6  67
44  12  42  55  18   6  67  94
12  42  44  18   6  55  67  94
12  42  18   6  44  55  67  94
12  18   6  42  44  55  67  94
12   6  18  42  44  55  67  94
 6  12  18  42  44  55  67  94
 6  12  18  42  44  55  67  94
```

```
void BubbleSort(int a [], int n)
{
  int i, j, t;
  for(i = n-1; i >= 0; i--)
    for(j = 1; j <= i; j++)
      if (a[j-1] > a[j])
      {
        t = a[j-1];
        a[j-1] = a[j];
        a[j] = t;
      }; // if
} // BubbleSort
```

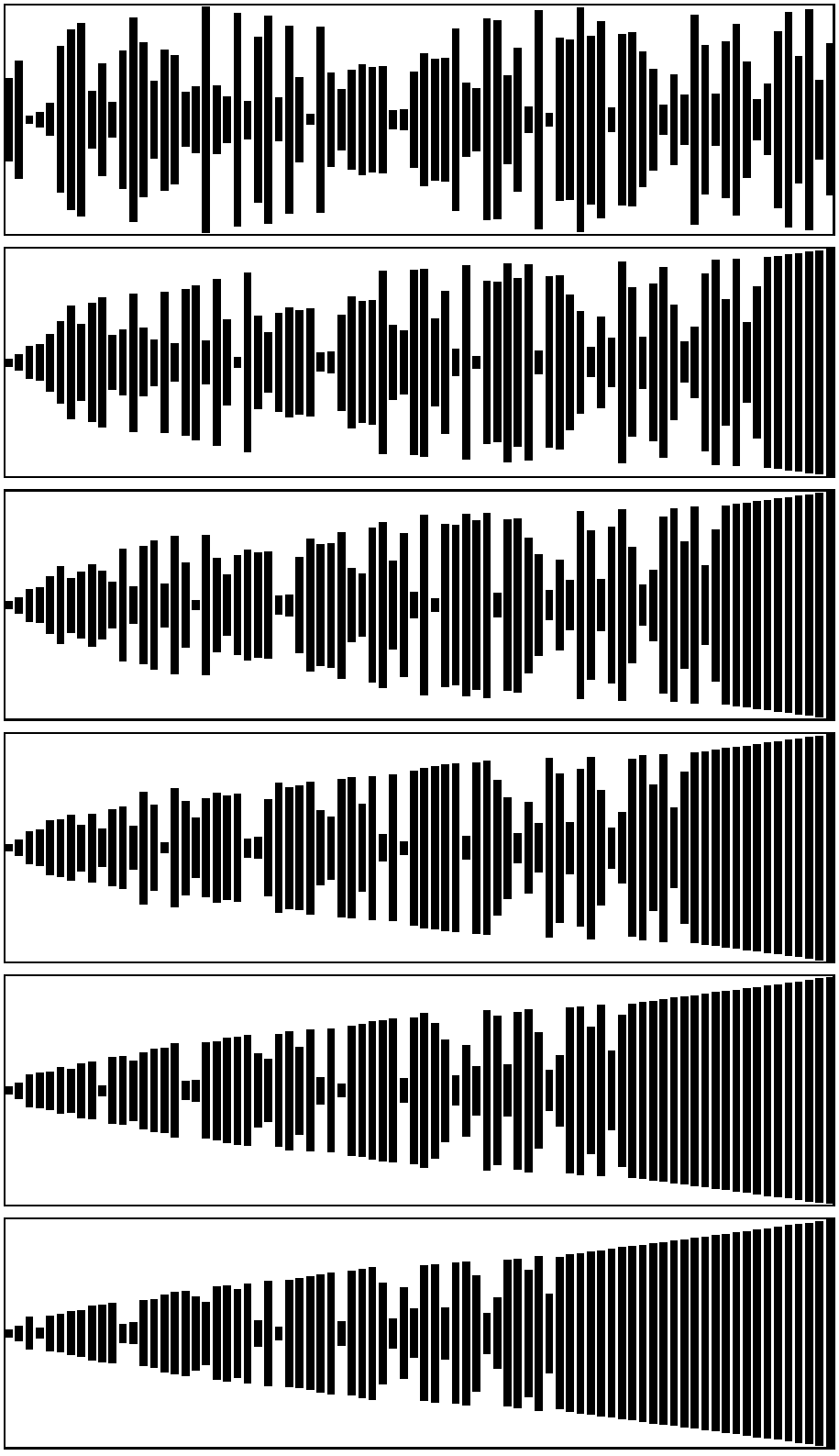
Uvedená metoda má řadu algoritmicky zajímavých variant. Varianta zvaná **RippleSort** si pamatuje pozici první dvojice u které došlo k výměně. V příštím cyklu začíná porovnávat až od předcházející dvojice.

5.4.6 ShakerSort

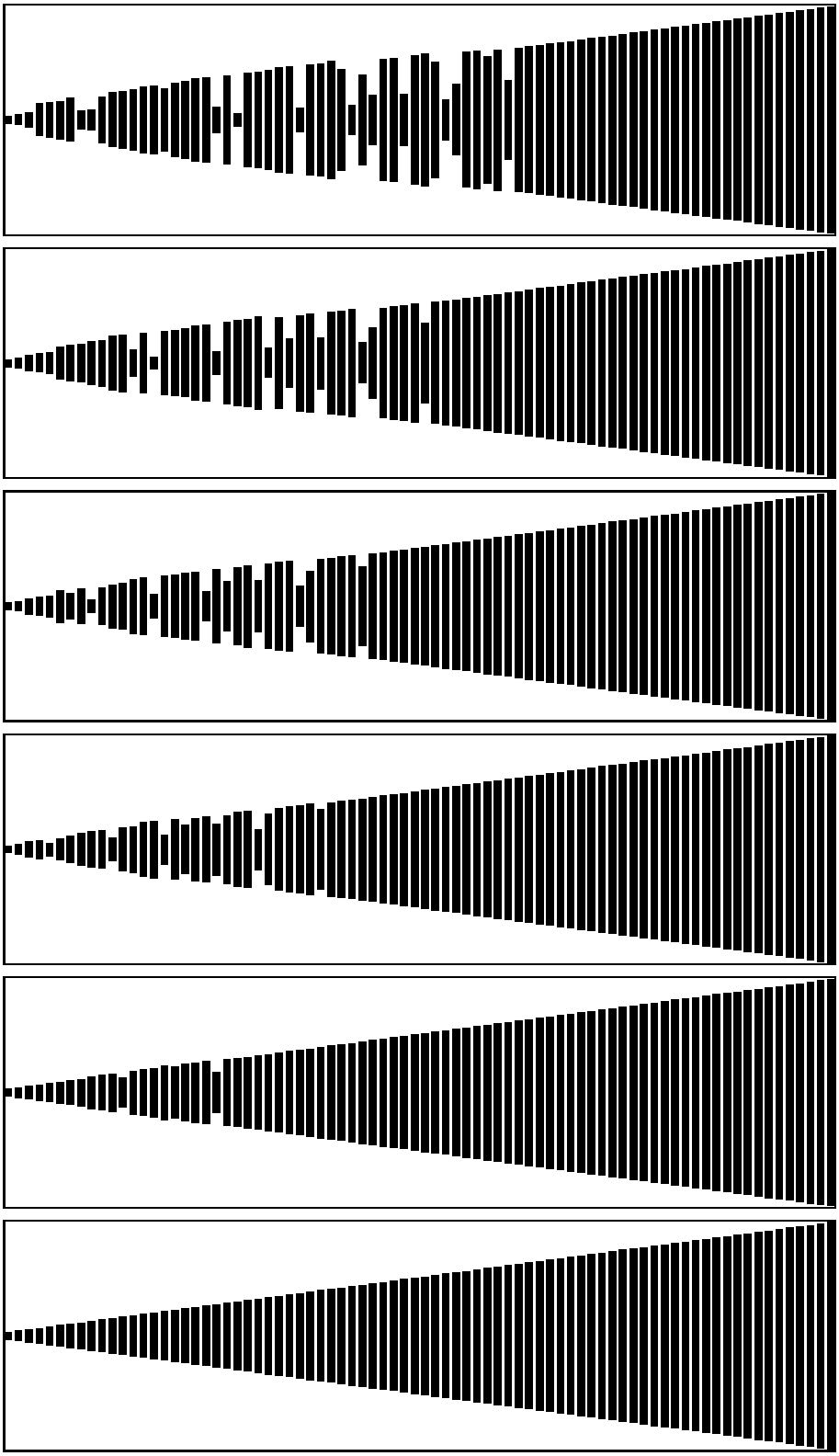
Varianta bublinkového třídění zvaná **ShakerSort** prochází pole střídavě zleva-doprava a zprava-doleva. Seřazené části posloupnosti jsou v průběhu třídění na obou koncích posloupnosti a při ukončení třídění se spojí. Průběh třídění je znázorněn na obrázcích 5.21, 5.22, 5.23 a 5.24.



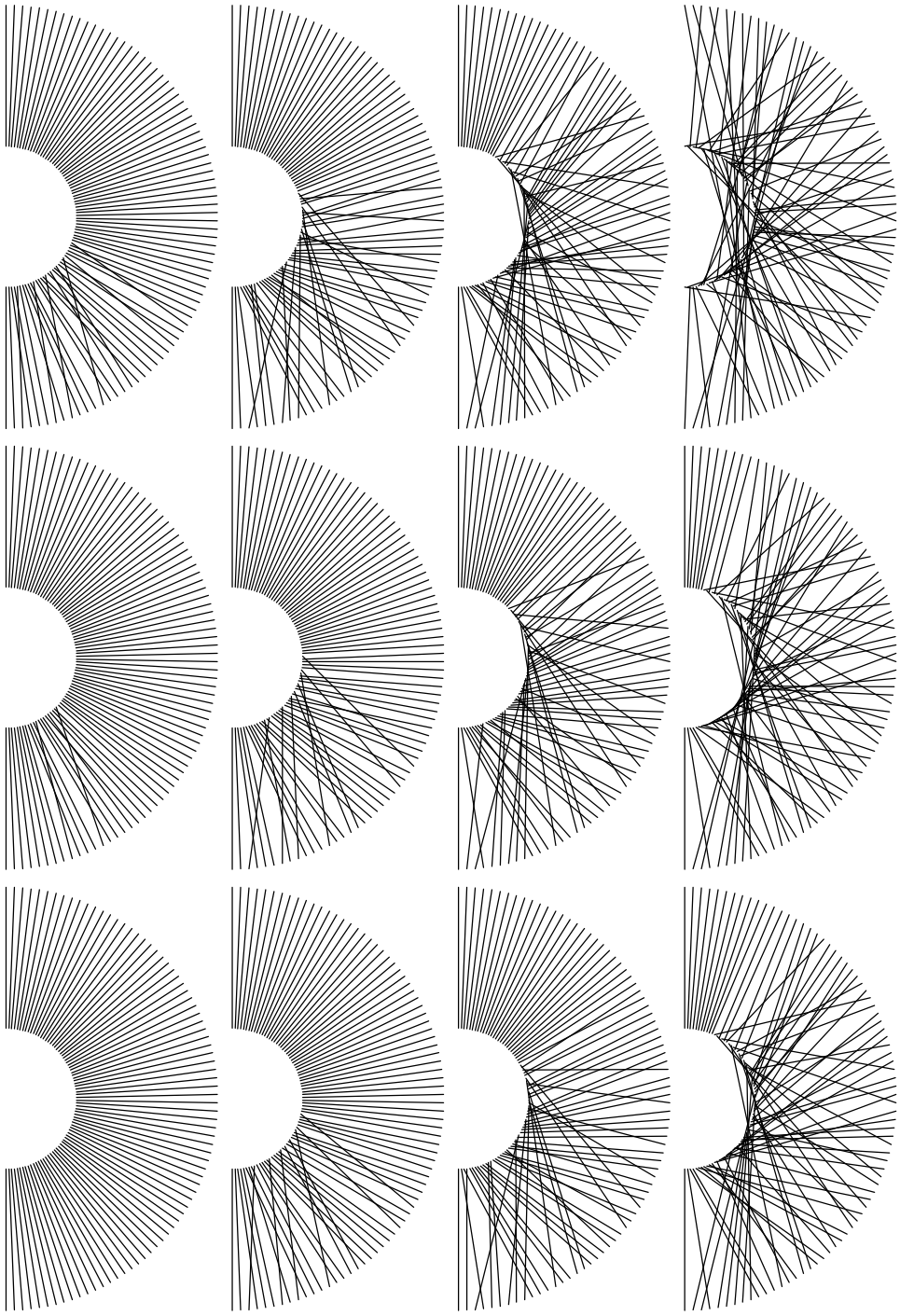
Obrázek 5.17: BubbleSort – průběh třídění I



Obrázek 5.18: BubbleSort – průběh třídění IIa



Obrázek 5.19: BubbleSort – průběh třídění IIb



Obrázek 5.20: BubbleSort – průběh třídění III

Příklad:

```
44  55  12  42  94  18   6  67
44  12  42  55  18   6  67  94
 6  44  12  42  55  18  67  94
 6  12  42  44  18  55  67  94
 6  12  18  42  44  55  67  94
 6  12  18  42  44  55  67  94
 6  12  18  42  44  55  67  94
```

```
void ShakerSort(int a [], int n)
{
  int i, t, k, r, l;
  l = 0;
  k = r = n - 1;
  do
  {
    for(i = r; i > l; i--)
      if (a[i-1] > a[i])
      {
        t = a[i-1];
        a[i-1] = a[i];
        a[i] = t;
        k = i;
      }; // if
    l = k;
    for(i = l; i < r; i++)
      if (a[i] > a[i+1])
      {
        t = a[i+1];
        a[i+1] = a[i];
        a[i] = t;
        k = i;
      }; // if
    r = k;
  } while (l < r);
} // ShakerSort
```

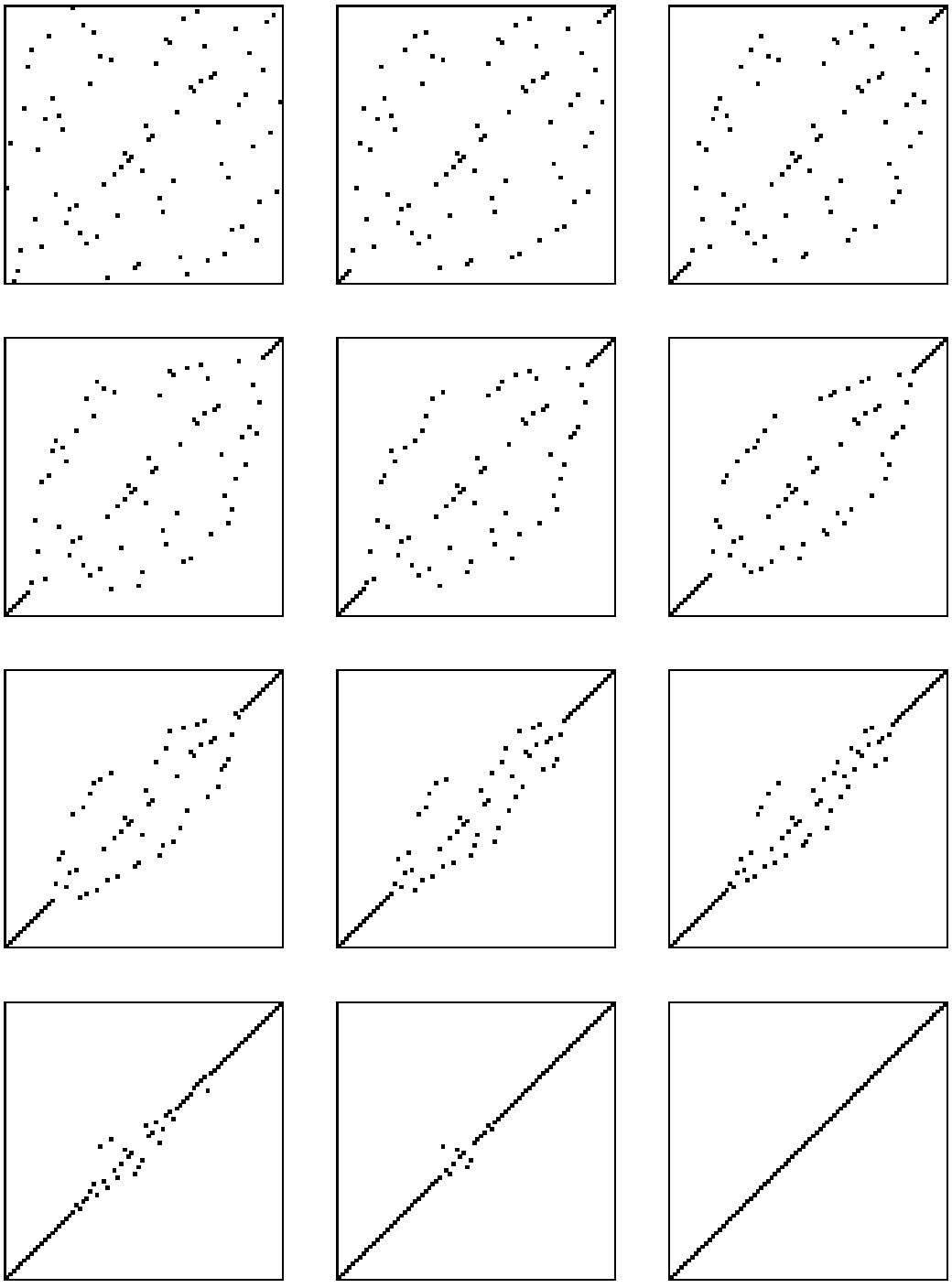
ANIMACE

Varianta zvaná **ShuttleSort** pracuje tak, že dojde-li u dvojice k výměně vrací se algoritmus a posunuje s prvkem tak dlouho dokud dochází k výměně. Pak se vrací do pozice u níž ukončil posun. Metoda končí, porovná-li úspěšně poslední dvojici prvků posloupnosti. Žádná z předcházejících variant však nepřináší kvalitativně lepší výsledky.

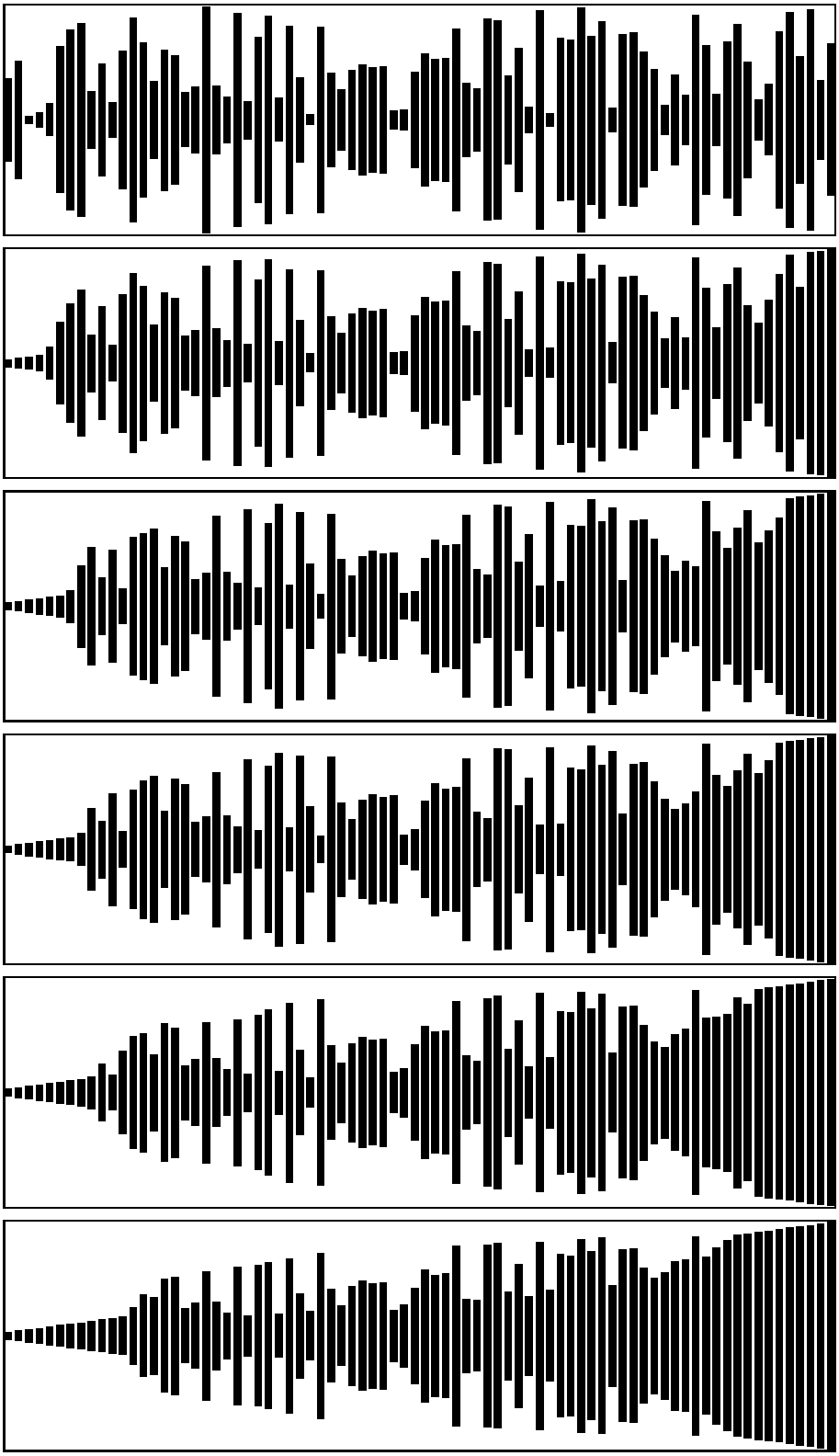
Analýza

Počet porovnání v bublinkovém třídění je $C = \frac{1}{2}(n^2 - n)$, minimální, průměrné a maximální počty přesunů jsou

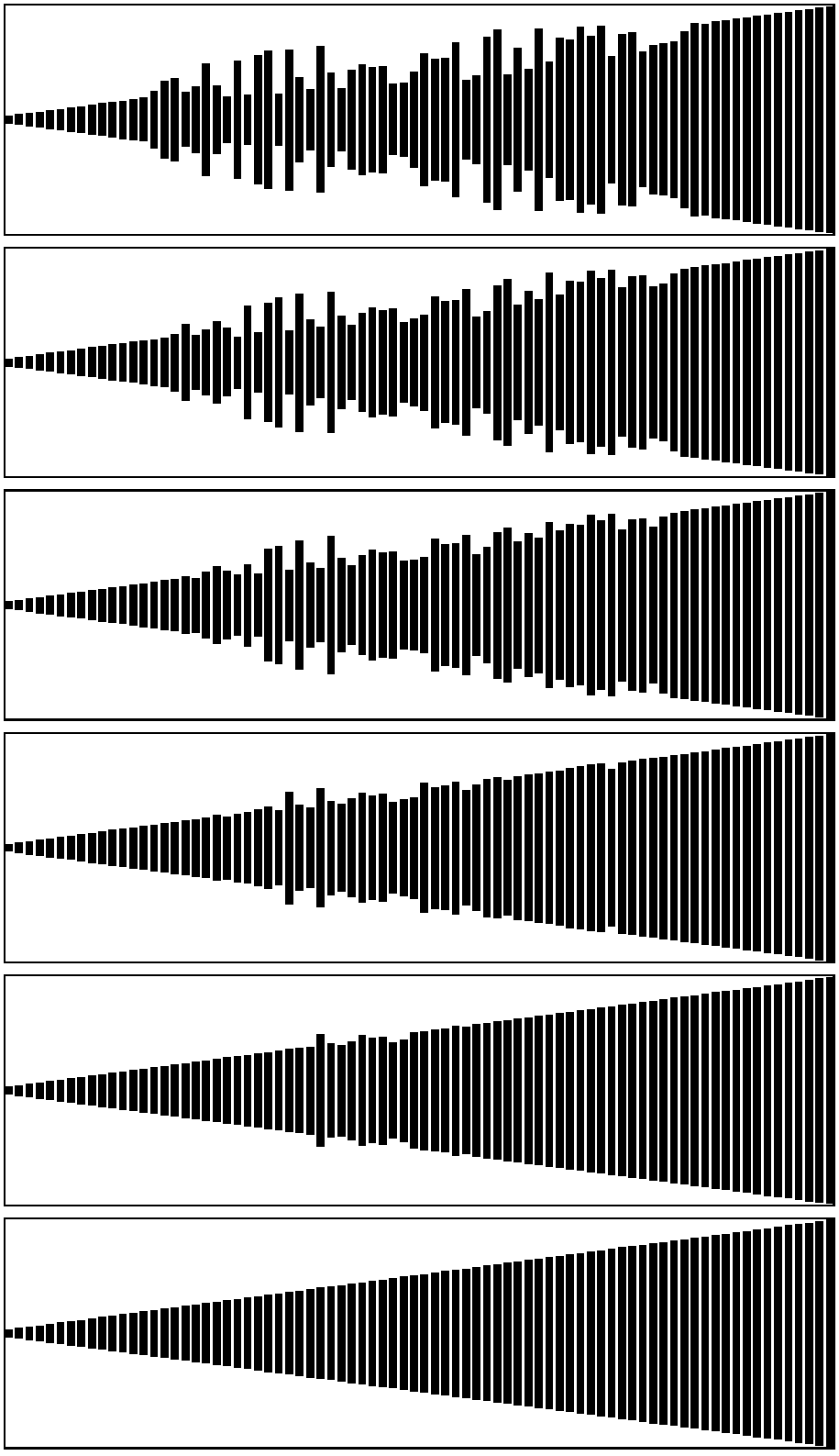
$$M_{min} = 0$$



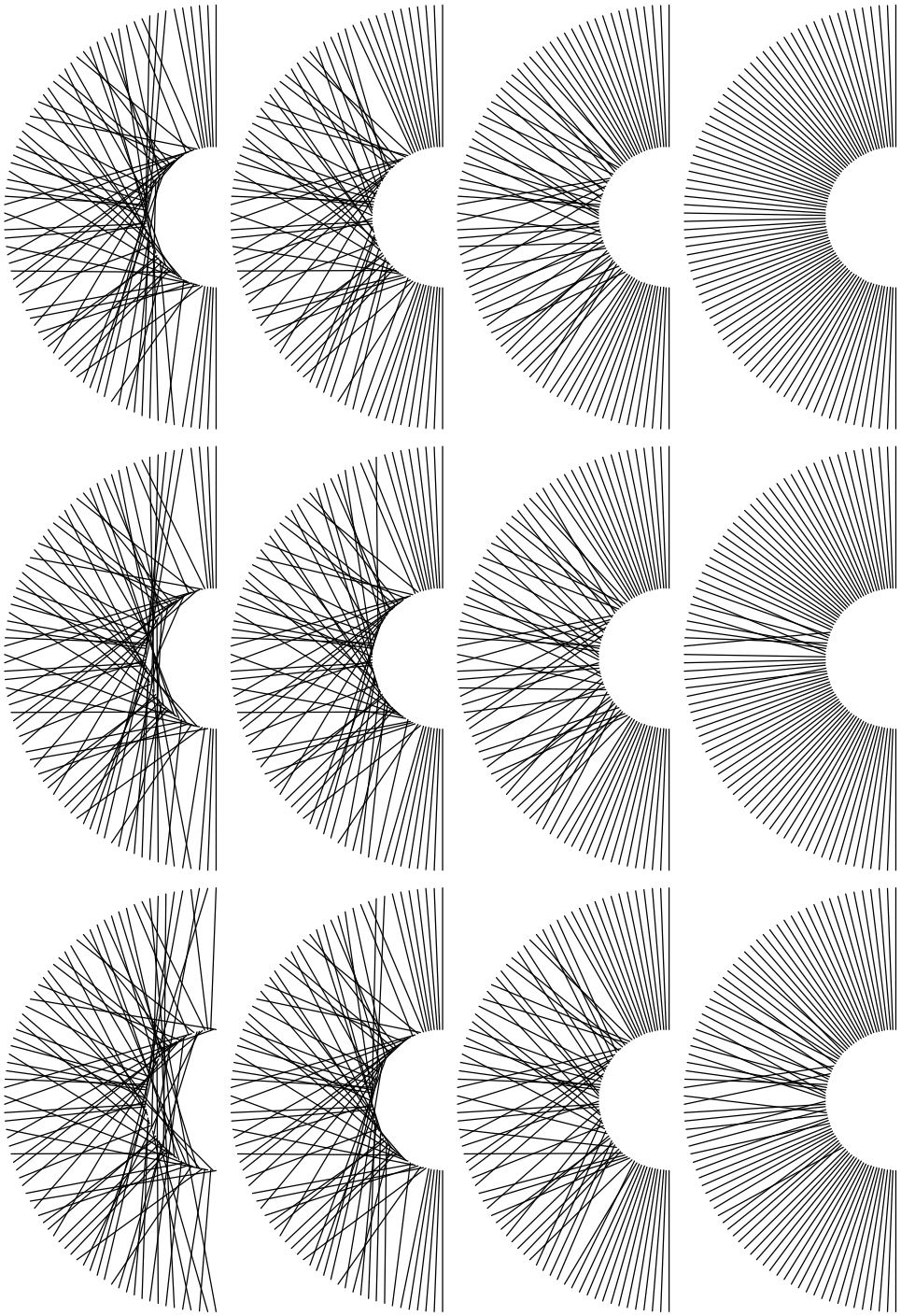
Obrázek 5.21: ShakerSort – průběh třídění I



Obrázek 5.22: ShakerSort – průběh třídění IIa



Obrázek 5.23: ShakerSort – průběh třídění IIb



Obrázek 5.24: ShakerSort – průběh třídění III

$$M_{avg} = \frac{3}{4}(n^2 - n)$$
$$M_{max} = \frac{3}{2}(n^2 - n)$$

Uvedené výsledky zohledňují i vylepšené verze algoritmu (ShakerSort). Minimální počet porovnání je $C_{min} = n - 1$. Co se týče vylepšeného bublinkového třídění, Knuth[11] zjistil, že průměrný počet průchodů je $n - k_1\sqrt{n}$ a průměrný počet porovnání

$$\frac{1}{2}\left[n^2 - n(k_2 + \ln n)\right]$$

Poznamenejme však, že všechna uvedená zlepšení nijak neovlivňují počet výměn, ale pouze zmenšují počet nadbytečných dvojnásobných kontrol. Záměna dvou prvků je však většinou o hodně náročnější činnost než porovnání prvků, tudíž uvedená zlepšení mají menší význam, než by člověk intuitivně očekával.

Tato analýza ukazuje, že bublinkové třídění a jeho malé zlepšení nedosahuje kvalit třídění vkládáním a výběrem. Algoritmus ShakerSort se dá s výhodou použít v případech, že prvky jsou téměř setříděné.

5.4.7 DobSort

V roce 1980 navrhl Dobosiewicz variantu (tzv. **DobSort**) *à*la Shell, která se ukázala být neočekávaně efektivní. Idea je následující: zvolíme posloupnost kroků $i = t, t-1, \ldots, 1, h_{i+1} > h_i, h_1 = 1, t > 1$ délky $t = O(\log n)$. V i-tém chodu třídíme stejně jako při bublinkovém třídění (bez jakýchkoliv vylepšení) prvky posloupnosti ležící ve vzdálenosti h_i , pro $i = t, t-1, \ldots, 2$. Po $i - 1$ chodech se třídění ukončí bublinkovém třídění.

```
void DobSort(int a [], int n)
{
  int i, j, h, t;
  for(h = 1; h <= n; h = 3*h+1);
  do
  {
    h = h / 3;
    j = 1;
    while (j < n-h)
    {
      if (a[j] > a[j+h])
      {
        t = a[j];
        a[j] = a[j + h];
        a[j + h] = t;
      }; // if
      j++;
    }; // while
  } while (h != 1);
```

```
  // následuje upravená verze bublinkového třídění
  j = 1;
  while (j == 1)
  {
    j = 0;
    for(i = 0; i < n; i++)
      if (a[i-1] > a[i])
      {
        t = a[i];
        a[i] = a[i-1];
        a[i-1] = t;
        j = 1;
      }; // if
  }; // while
} // DobSort
```

Výsledky ukázaly, že pro $n \leq 10000$ byl algoritmus prakticky stejně rychlý jako Quicksort, a přibližně dvakrát rychlejší než Shellsort. U tohoto algoritmu však zatím nebyla provedena podrobnější analýza složitosti.

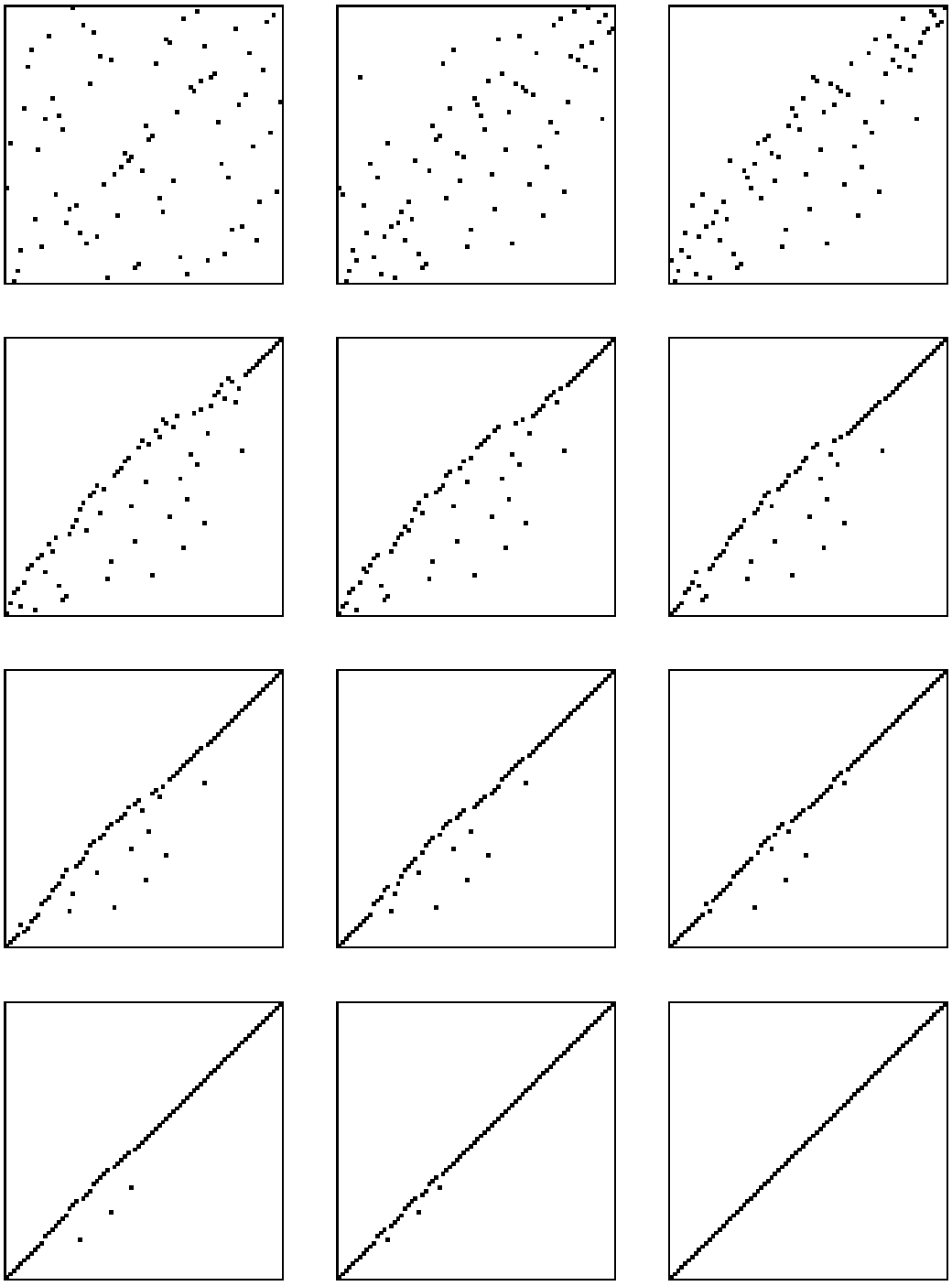
5.4.8 Třídění haldou

HeapSort je další způsob, jak vylepšit jednoduchý algoritmus třídění výběrem. Spočívá v nalezení efektivnějšího způsobu výběru i-tého největšího prvku. Je zřejmé, že pro $i = 1$ (pro nalezení maximálního prvku) vždy potřebujeme $n-1$ porovnání. Pro další kroky $i > 1$ je však možné si zapamatovat si jistým způsobem výsledky předcházejících porovnání, a později je využít. Tomuto účelu nejlépe vyhovuje datová struktura halda.

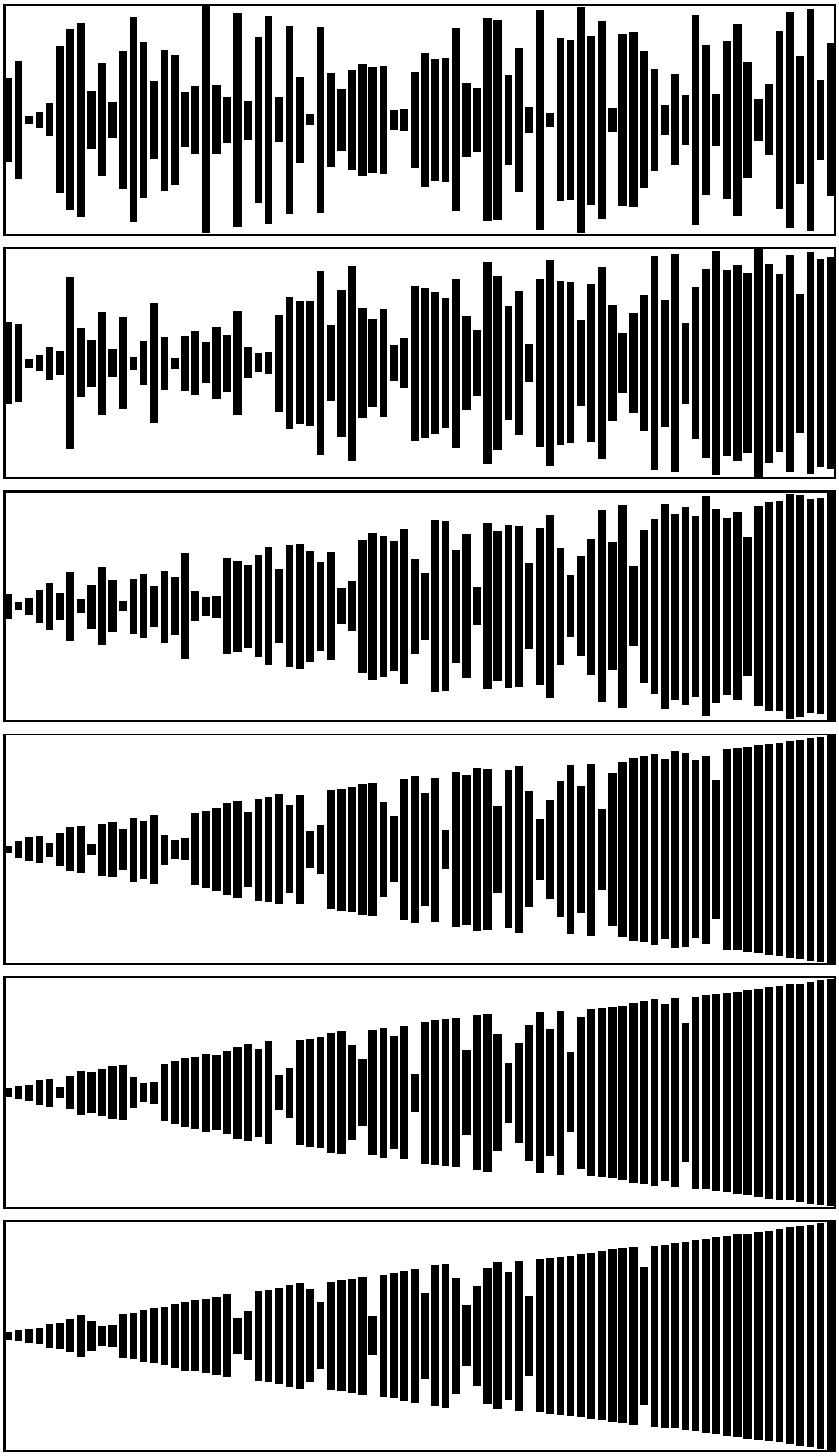
Halda reprezentující posloupnost S mohutnosti n, je úplný binární strom výšky $h \geq 1$ s n vrcholy, a následujícími vlastnostmi:

- všechny listy se nachází ve vzdálenosti h nebo $h-1$ od kořene;
- všechny listy na úrovni h jsou vlevo od listů na úrovni $h-1$;
- každému vrcholu tohoto stromu je přiřazen jeden prvek posloupnosti S tak, že všem jeho potomkům jsou přiřazeny menší prvky.

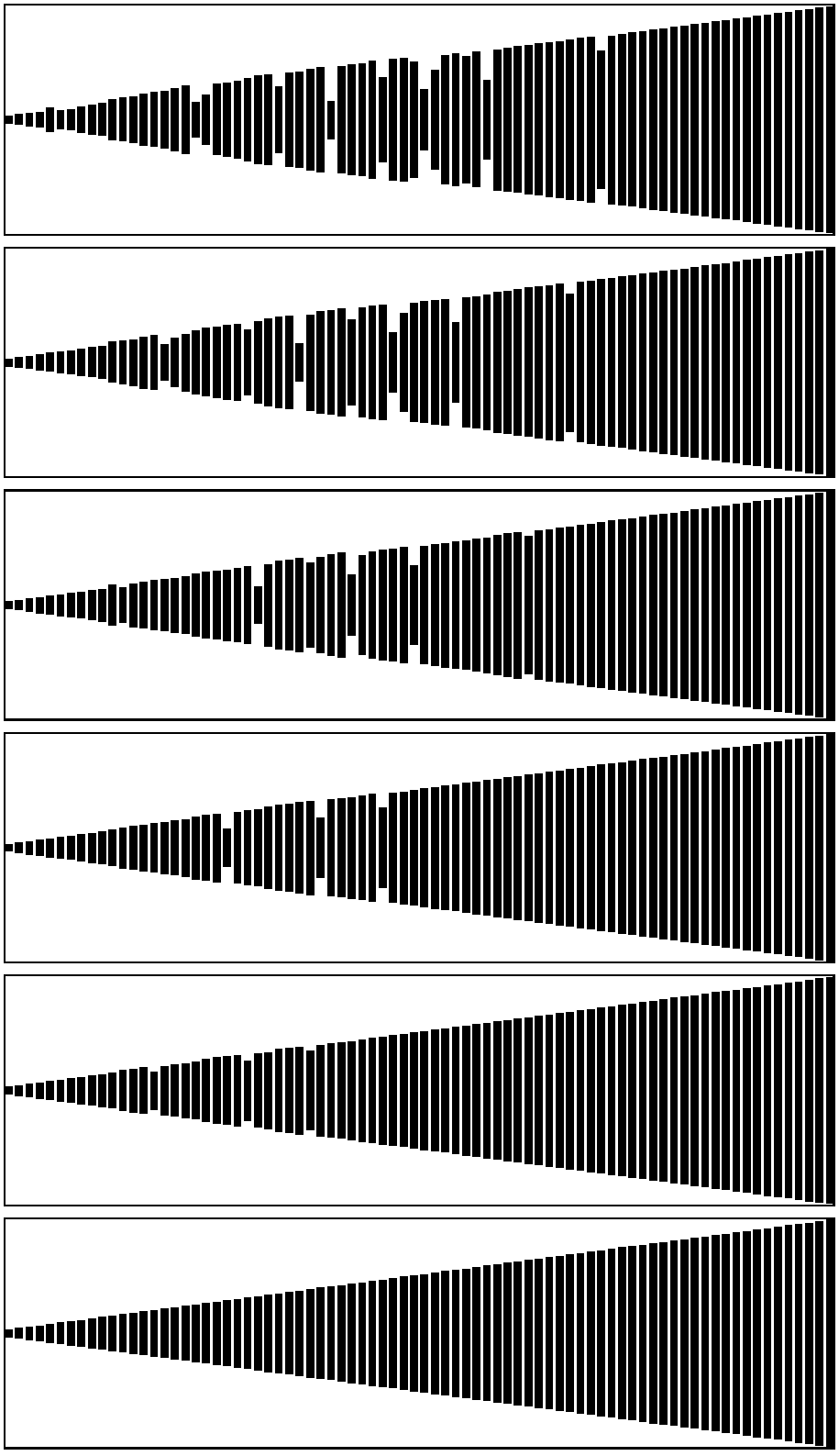
Halda se dá výhodně reprezentovat v poli tak, že do pole postupně zapíšeme všechny prvky přiřazené jednotlivým poschodím, zleva doprava, od kořene směrem k listům. Při této reprezentaci zřejmě platí, že prvek ležící na i-té pozici má levého potomka na pozici $2i$, pravého potomka na pozici $2i+1$. Z podmínek haldy potom plyne, že maximální prvek je vždy v kořenu haldy to znamená na první pozici pole.



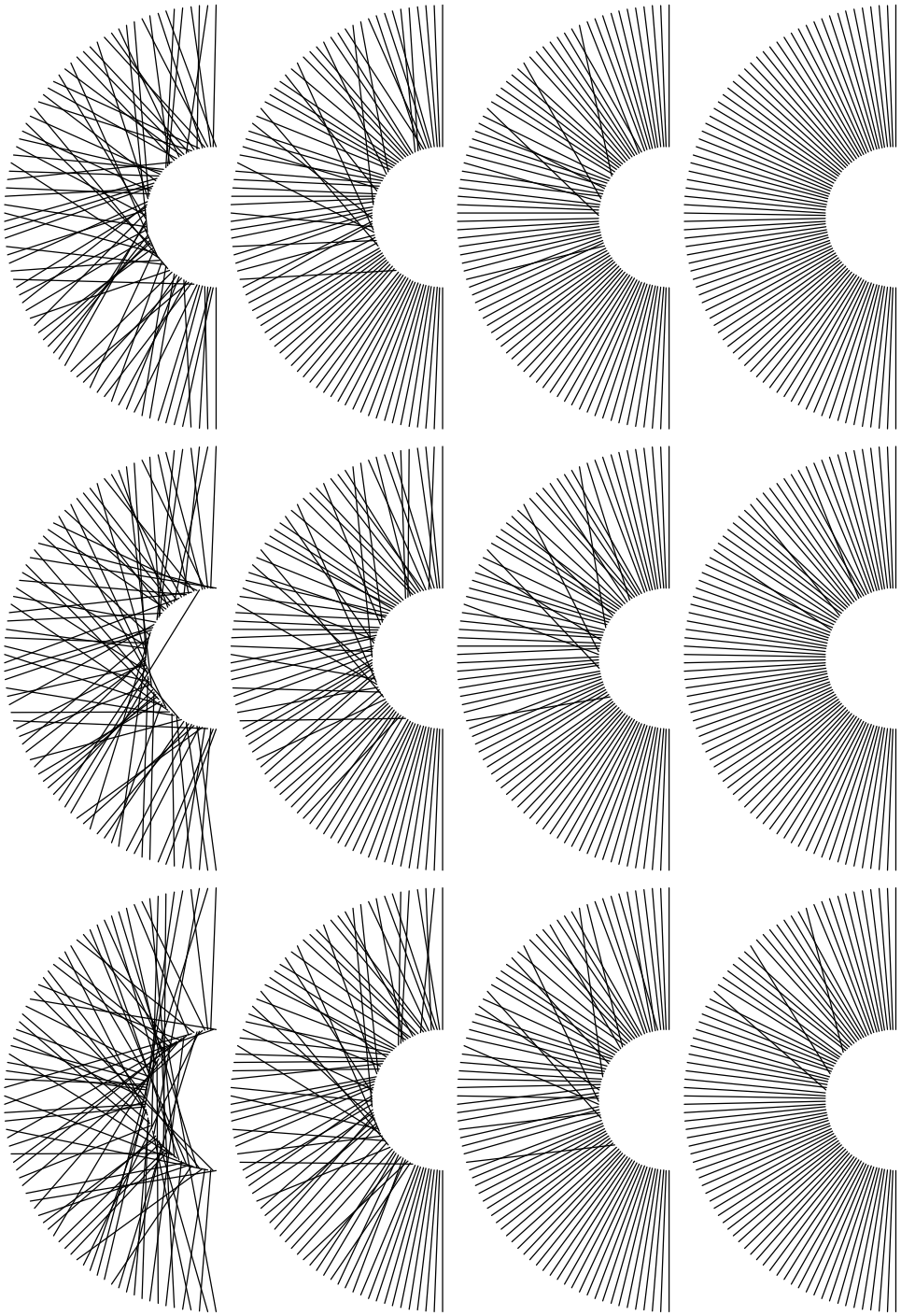
Obrázek 5.25: DobSort – průběh třídění I



Obrázek 5.26: DobSort – průběh třídění IIa



Obrázek 5.27: DobSort – průběh třídění IIb

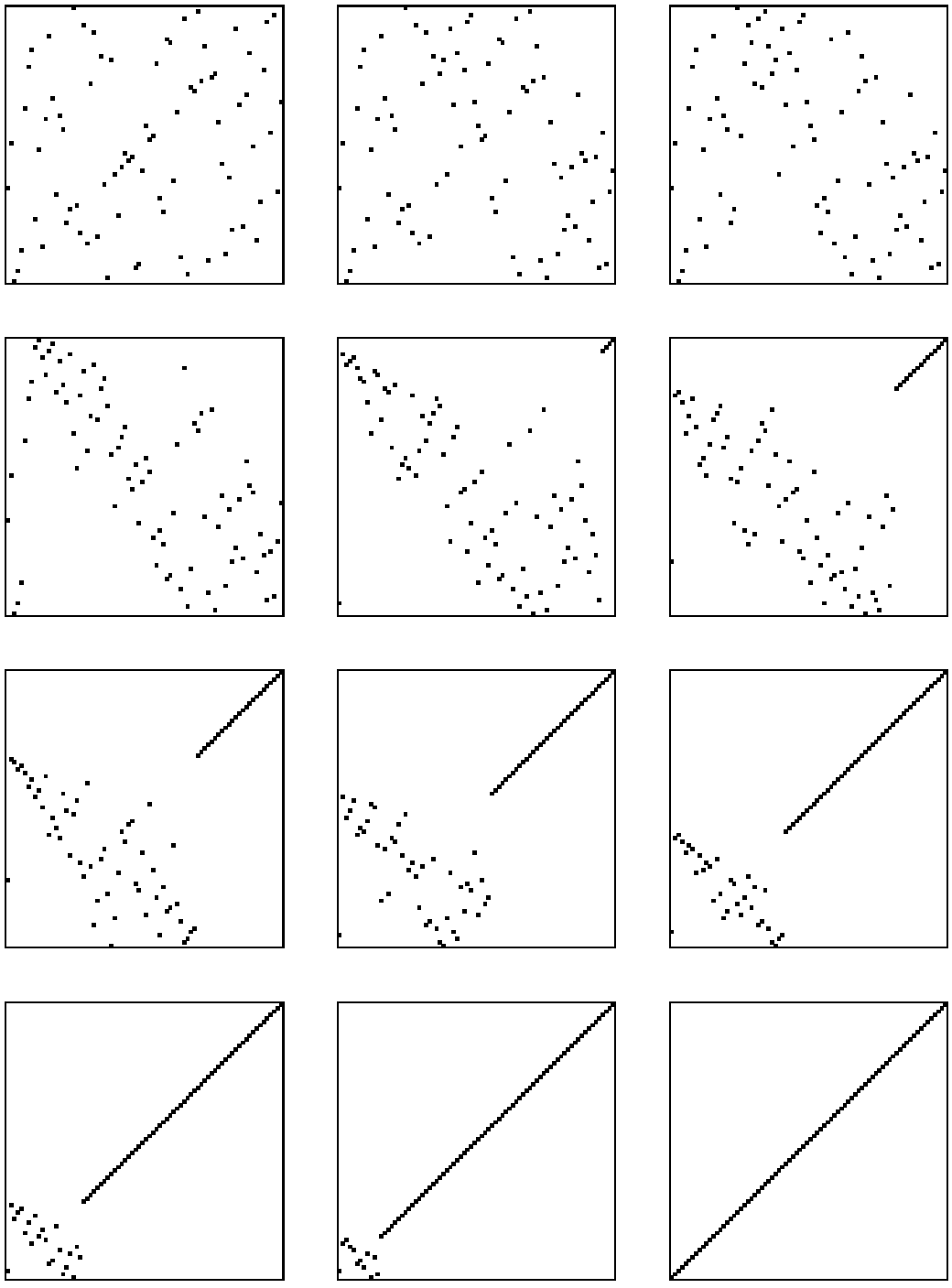


Obrázek 5.28: DobSort – průběh třídění III

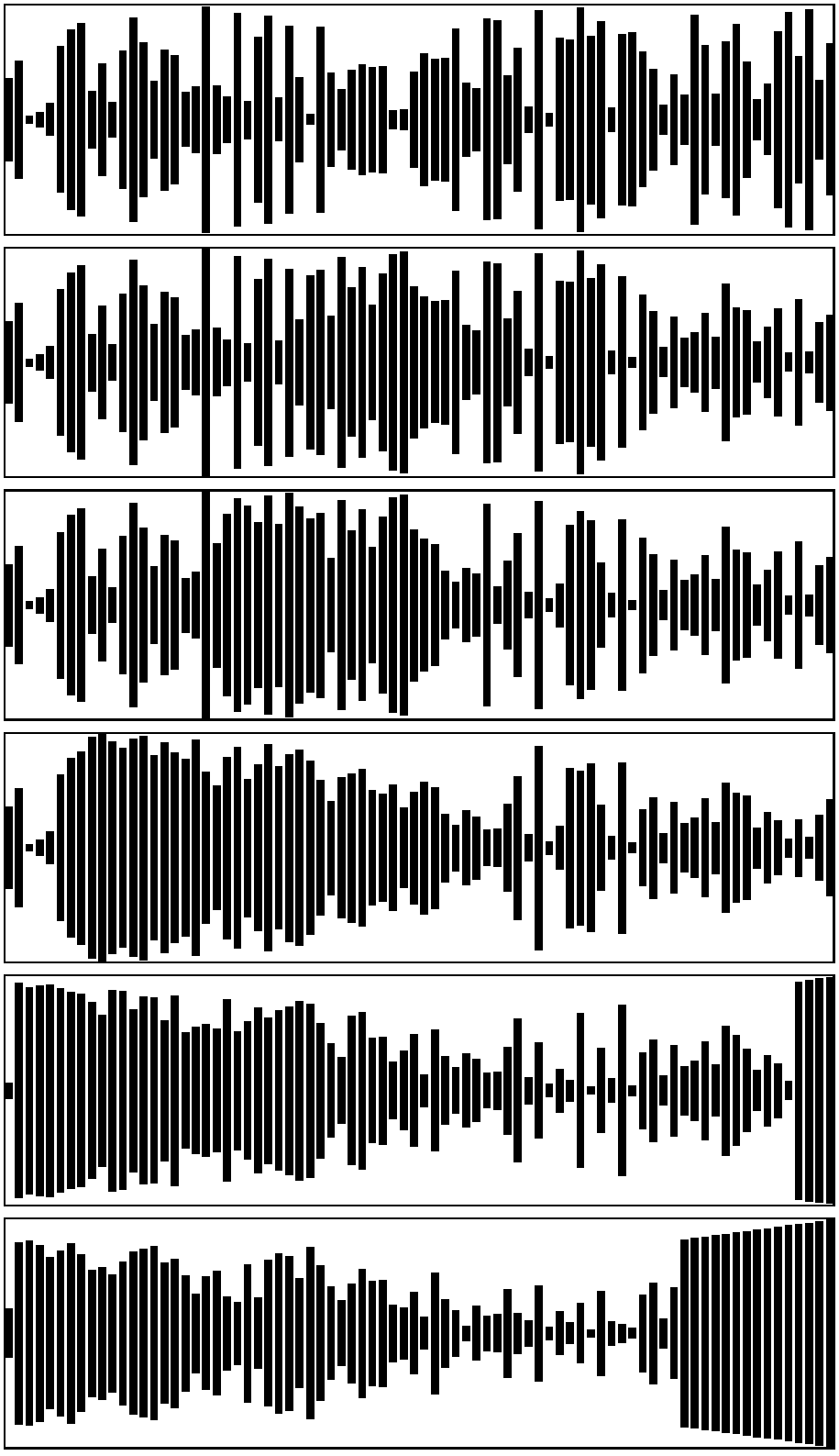
Příklad:

```
44  55  12  42  94  18   6  67
44  55  12  67  94  18   6  42
44  55  94  67  12  18   6  42
44  94  55  67  12  18   6  42
94  67  55  44  12  18   6  42
67  55  42  44  12  18   6  94
55  44  42   6  12  18  67  94
44  42  18   6  12  55  67  94
42  18  12   6  44  55  67  94
18   6  12  42  44  55  67  94
12   6  18  42  44  55  67  94
 6  12  18  42  44  55  67  94
```

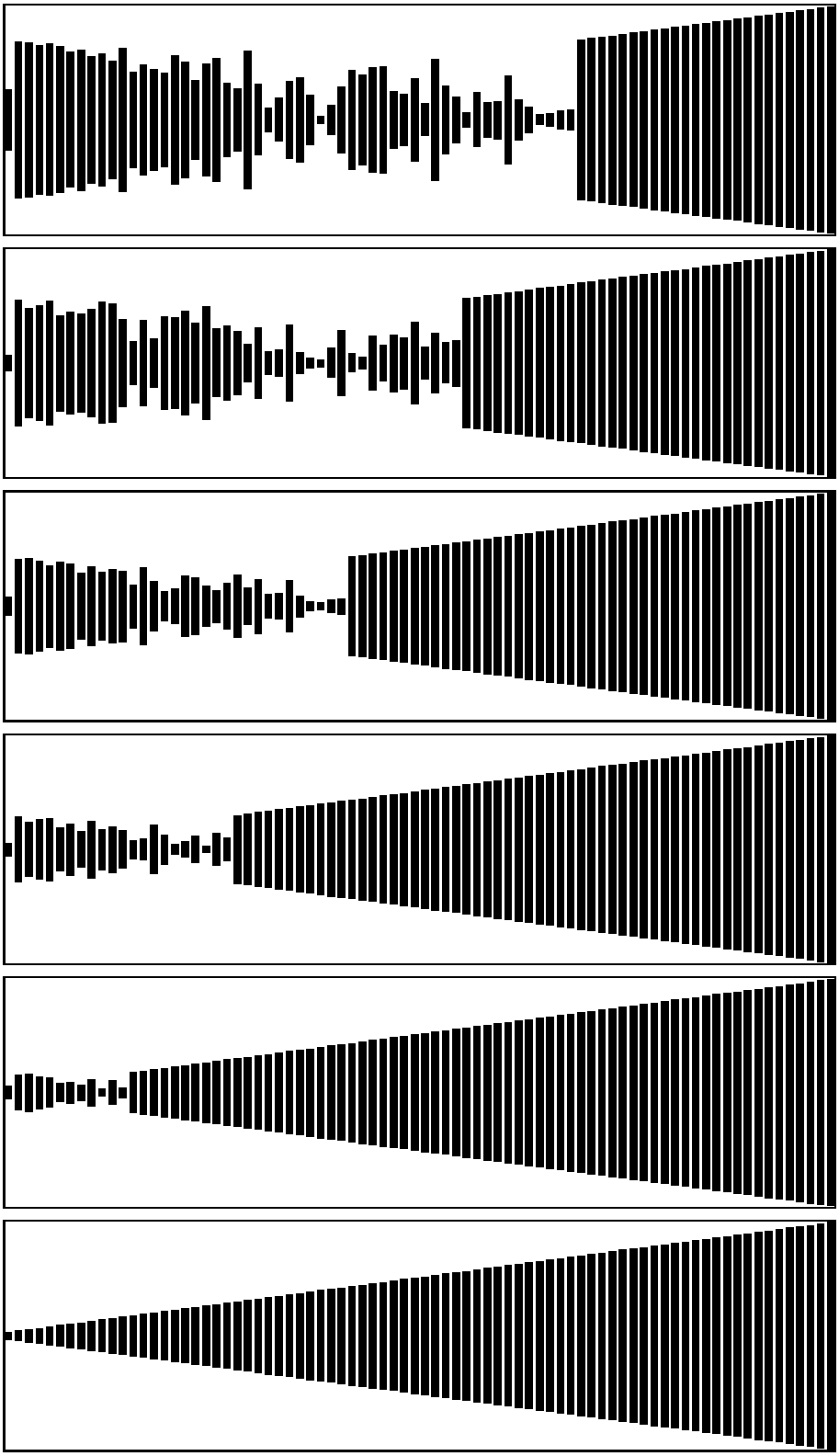
```
void DownHeap(int a[], int k, int l)
{
  int j, v;
  v = a[k];
  while (k < l/2)
  {
    j = k + k;
    if (j < (l-1))
      if (a[j] < a[j+1])
        j += 1;
    if (v >= a[j])
      break;
    a[k] = a[j];
    k = j;
  }; // while
  a[k] = v;
} // DownHeap


void HeapSort(int a [], int n)
{
  int i, t;
  for(i = n/2; i >= 0; i--)
    DownHeap(a, i, n);
  i = n-1;
  do
  {
    t = a[0]; a[0] = a[i]; a[i] = t;
    i -= 1;
    DownHeap(a, 0, i+1);
  } while (i > 0);
} // HeapSort
```



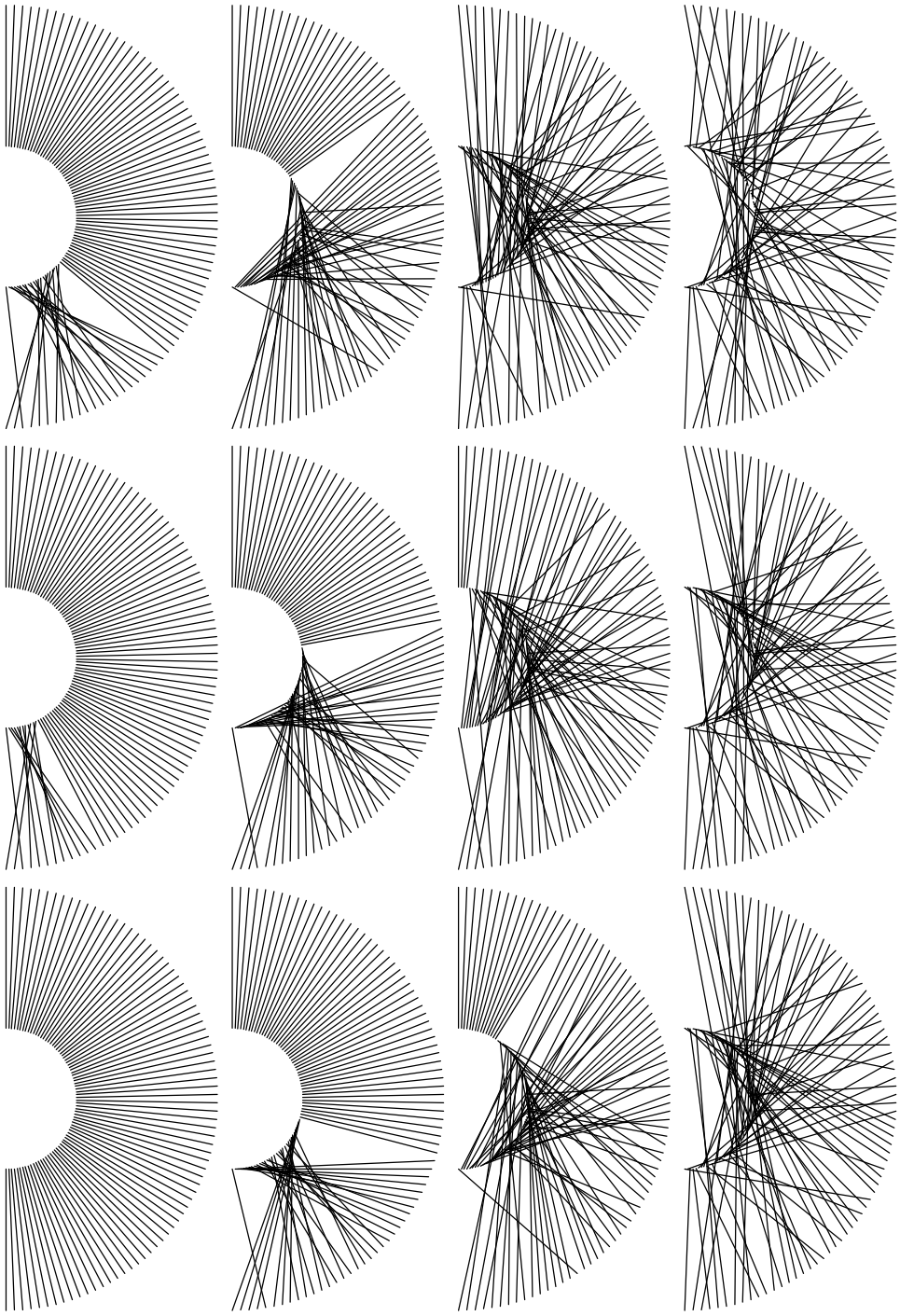
Obrázek 5.29: HeapSort – průběh třídění I



Obrázek 5.30: HeapSort – průběh třídění IIa



Obrázek 5.31: HeapSort – průběh třídění IIb



Obrázek 5.32: HeapSort – průběh třídění III

Analýza

Věta 5.1 *Všechny základní operace s haldou – vložení, zrušení, výměna (naše pomocná funkce `DownHeap`) prvků – vyžadují méně než* $2 \log n$ *porovnání, za předpokladu, že halda má* n *prvků.*

Důkaz. Všechny tyto operace vyžadují průchod haldou od jejího kořene k listům, což představuje ne více než $\log n$ uzlů pro haldu s n prvky. Násobící faktor 2 pochází právě od funkce `DownHeap`, která ve svém cyklu provádí dvě porovnání. ∎

Věta 5.2 *Konstrukci haldy zdola nahoru lze provést v lineárním čase.*

Důkaz. Ke tvrzení nás oprávňuje fakt, že většina zpracovávaných hald je velice malá. Například k vybudování haldy ze 127 prvků, je funkce `DownHeap` vyvolána 64 krát pro haldu velikosti 1, 32 krát pro haldu velikosti 3, 16 krát pro haldu velikosti 7, 8 krát po 15 prvcích, 4 krát pro haldu o 31 prvcích, dvakrát pro haldu velikosti 63 prvků a naposledy pro haldu o 127 prvcích. Dohromady $64 \cdot 0 + 32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ vyvolání a to v nejhorším případě. Pro $n = 2^b$ je horní hranice počtu porovnání definována jako

$$\sum_{k=1}^{b} (k-1)2^{b-k} = 2^b - b - 1 < n$$

Obdobně lze tvrzení dokázat pro n která nejsou mocninami 2. ∎

Věta 5.3 *HeapSort potřebuje k setřídění* n *prvků méně než* $2 \log n$ *porovnání.*

Důkaz. O něco vyšší hranici, přibližně $3 \log n$, dostáváme bezprostředně z věty 5.1. Nižší horní hranici uvedená v této větě plyne z věty 5.2. ∎

5.4.9 Třídění rozdělováním

Při **třídění rozdělováním** se v souladu s metodou divide-et-impera nejprve tříděná množina rozdělí na dvě disjunktní podmnožiny podle možnosti přibližně stejné velikosti tak, že všechny prvky jedné množiny jsou menší než prvky druhé množiny.

Každá množina se potom rekurzivně dotřídí. Závěrečný syntetizační krok je triviální – spočívá v konkatenaci setříděných posloupností. Rozdělení množiny v prvním kroku se realizuje tak, že se zvolí jeden prvek z množiny – zvaný **pivot** – a jedna podmnožina potom obsahuje všechny prvky menší než pivot, a druhá všechny ostatní. Tento na první pohled přirozený způsob třídění objevil v roce 1962 C. A. R. Hoare a nazval jej QuickSort.

Příklad:

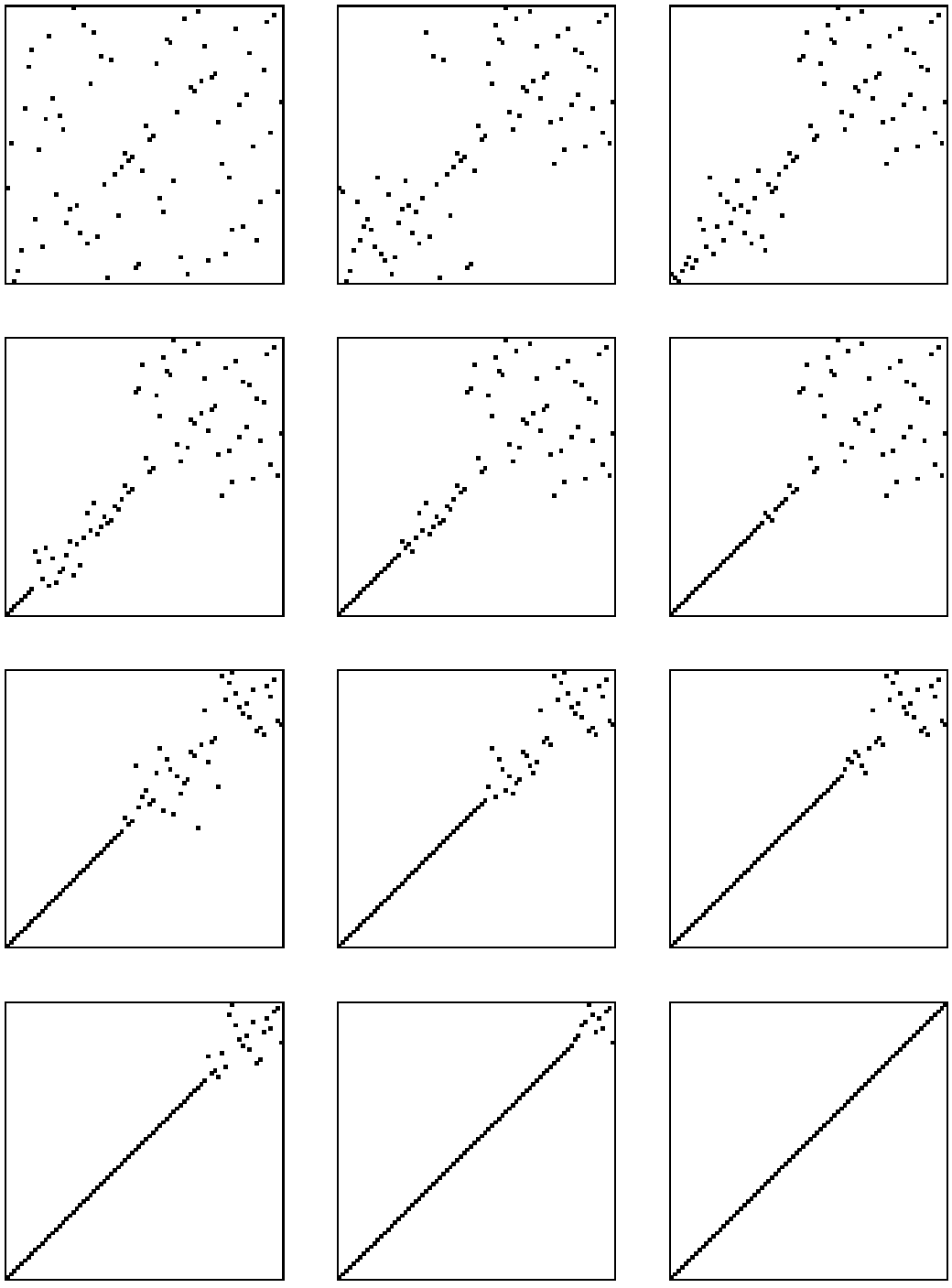
```
6  55  12  42  94  18  44  67
6  18  12  42  94  55  44  67
6  18  12  42  94  55  44  67
6  12  18  42  94  55  44  67
6  12  18  42  94  55  44  67
6  12  18  42  44  55  94  67
6  12  18  42  44  55  94  67
6  12  18  42  44  55  67  94
```

```
void QuickSort(int a[], int l, int r)
{
  int i, j, t, v;
  i = l;
  j = r;
  v = a[(l+r)/2];
  do
  {
    while (a[i] < v)
      i += 1;
    while (v < a[j])
      j -= 1;
    if (i <= j)
    {
      t = a[i]; a[i] = a[j]; a[j] = t;
      i++;
      j--;
    }; // if
  } while (i <= j);
  if (l < j)
    QuickSort(l, j);
  if (i < r)
    QuickSort(i, r);
} // QuickSort
```

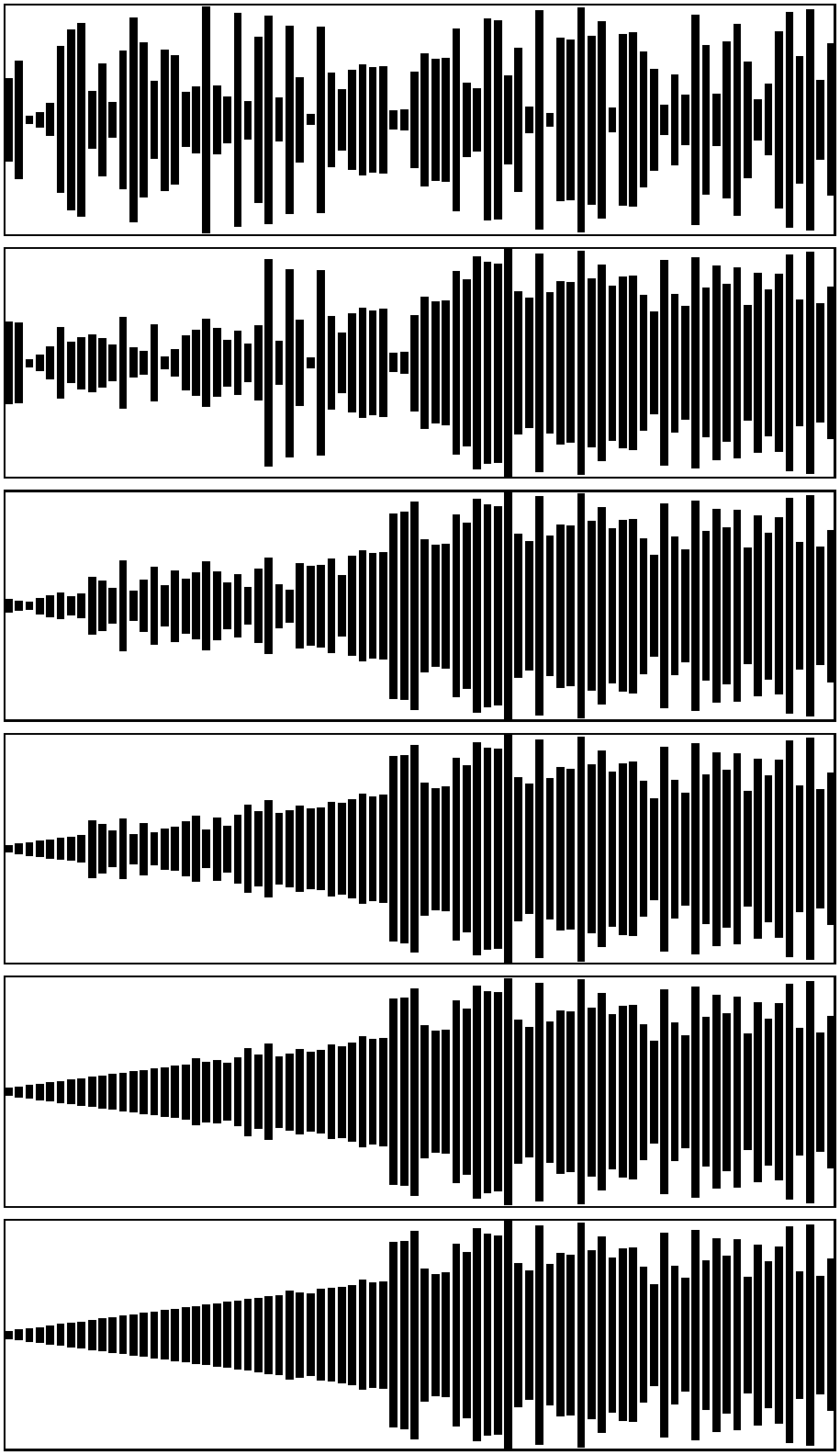
ANIAMCE

QuickSort se nejčastěji uvádí v rekurzivní variantě. Lze ovšem napsat nerekurzivní (iterační) verzi, transformací na iteraci s pomocnou pamětí. V průběhu iterace dělíme pole na dva úseky, ale v následujícím okamžiku jsme schopni zpracovat jen jednu část pole a druhou bude nutné uložit do pomocné paměti. Je zřejmé, že úseky by se měly z pomocné paměti vybírat v opačném pořadí, než tam byly uloženy. Z těchto úvah plyne, že pomocná paměť se chová jako zásobník. V našem ukázkovém kódu předpokládáme, že máme k dispozici třídu CStack, realizující zásobník přirozených čísel.

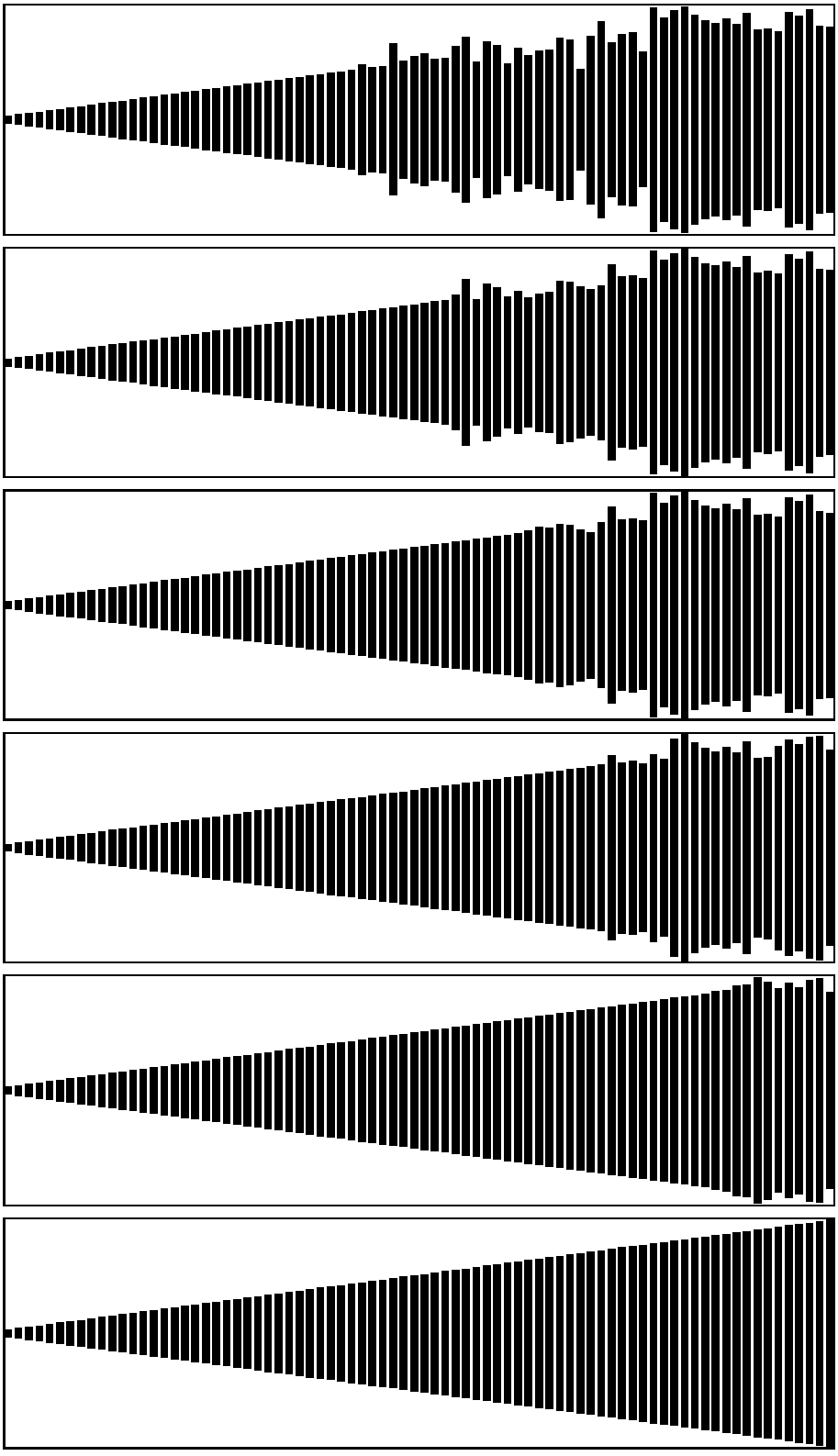
```
void QuickSortN(int a[], int n)
{
  CStack stack;
  int i, j, l, r, w, x;
  stack.Push(0);
  stack.Push(n-1);
```



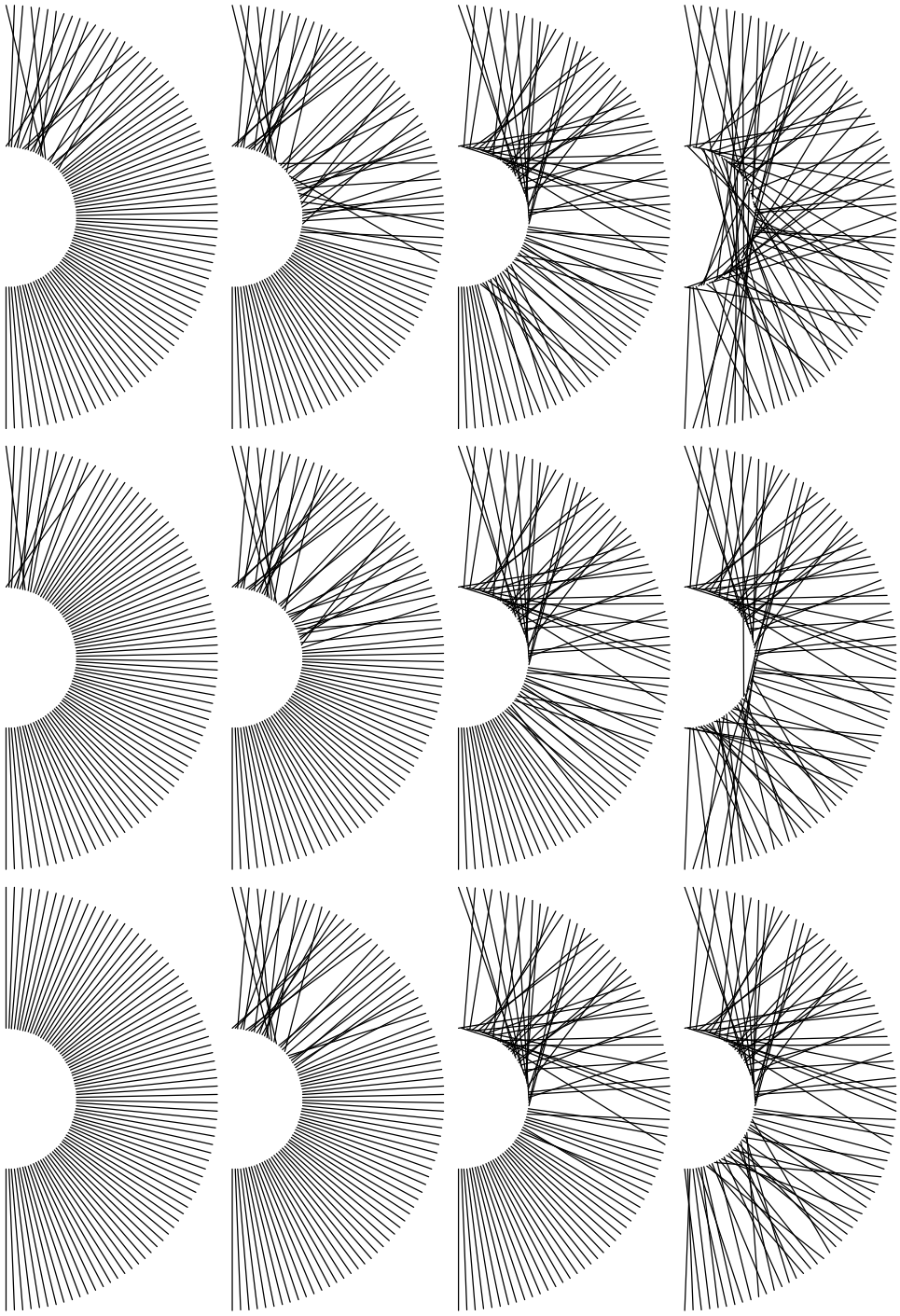
Obrázek 5.33: QuickSort – průběh třídění I



Obrázek 5.34: QuickSort – průběh třídění IIa



Obrázek 5.35: QuickSort – průběh třídění IIb



Obrázek 5.36: QuickSort – průběh třídění III

```
do
{
  r = stack.Pop();
  l = stack.Pop();
  do
  {
    i = l;  j = r;
    x = a[(l+r)/2];
    do
    {
      while (a[i] < x) i += 1;
      while (x < a[j]) j -= 1;
      if (i <= j)
      {
        w = a[i];  a[i] = a[j];  a[j] = w;
        i += 1;
        j -= 1;
      };  // if
    } while (i <= j);
    if (i < r)
    {// uložíme pravý úsek pole
      stack.Push(i);
      stack.Push(r);
    };  // if
    r = j;
  } while (l < r);
} while (!stack.Empty());
}  // QuickSortN
```

Zbývá odpovědět na otázku, jak velký zásobník budeme potřebovat? Nejhorší případ nastane, pokud pravý úsek, odkládaný do zásobníku, bude tvořen jediným prvkem. Potom rozsah zásobníku bude n, což je nepřijatelné. Zlepšení dosáhneme tím, že do zásobníku budeme odkládat delší úsek tříděného pole a pokračujeme úsekem kratším. V tomto případě bude velikost zásobníku ohraničena $\log n$. Mohli bychom doplnit kód nerekurzivního QuickSortu o následující test:

```
if (j-l < r-i)
{
  if (i < r)
  {// uložíme pravý úsek
    stack.Push(i);
    stack.Push(r);
  };  // if
  r = j;  // pokračujeme levým úsekem
}  // if
else
{
  if (l < j)
  {// uložíme levý úsek
    stack.Push(l);
    stack.Push(j);
  };  // if
```

```
    l = i;  // pokračujeme pravým úsekem
  };  // else
```

Praktické testy ukázaly, že QuickSort je velice rychlý při třídění rozsáhlých polí, ale zaostává oproti přirozeným algoritmům třídění (InsertSort, SelectSort) při třídění malých polí. Ukazuje se, že složitost QuickSortu má jistou minimální hodnotu, pod kterou i při třídění malého počtu prvků neklesne. Kdežto složitost přirozených algoritmů roste úměrně s počtem tříděných prvků. Jinými slovy do jistého počtu prvků má QuickSort větší režii (třídí pomaleji) než např. InsertSort. Tato hranice byla experimentálně stanovena na asi 12 prvků. Vyplatilo by se tedy QuickSortem třídit rozsáhlé pole, ale jakmile úseky na než se pole dělí budou kratší než zvolená mez (řekněme zmíněných 12 prvků), tento krátký úsek dotřídit InsertSortem.

```
void Insertion (int a [], int l, int r)
{
  int i, j, v;
  for(i = l; i < r; i++)
  {
    v = a[i];  j = i;
    while ((a[j-1] > v) && (j > 0))
    {
      a[j] = a[j-1];
      j--;
    }  // while
    a[j] = v;
  };  // for
}  // Insertion


void QuickSort12(int a [], int l, int r)
{
  const int  m = 12;
  int  i, j, t, v;
  if (r - l < m)
    Insertion (l, r);
  else
  {
    i = l;  j = r;  v = a[(l+r)/2];
    do
    {
      while (a[i] < v) i += 1;
      while (v < a[j]) j -= 1;;
      if (i <= j)
      {
        t = a[i];  a[i] = a[j];  a[j] = t;
        i += 1;  j -= 1;
      };  // if
    } while (i <= j);
    if (l < j)
      QuickSort12(l, j);
    if (i < r)
      QuickSort12(i, r);
```

```
    };  // else
  }  // QuickSort12
```

Analýza

Analýza QuickSortu patří mezi extrémně obtížné matematické problémy a nebyla dodnes úplně vyřešena. Uvedeme zde jen několik základních výsledků. Podrobnější rozbor lze nalézt v [7].

Nejlepší případ QuickSortu nastane, pokud se každým dělením tříděná posloupnost rozdělí přesně na poloviny. Potom počet porovnání lze vypočítat podle formule (stejnou formuli splňují i ostatní algoritmy založené na strategii divide-et-impera)

$$C_n = 2C_{n/2} + n$$

Výraz $2C_{n/2}$ pokrývá třídění dvou podposloupností, n je počet porovnání všech prvků v průběhu rozdělování posloupnosti. Výše uvedenou rekurzivní formuli lze vyřešit a dostáváme

$$C_n \approx n \log n$$

Je jasné, že rozdělení posloupnosti nedopadne vždy tak dobře, ale lze říci, že tomu tak je v průměru. Jestliže budeme postupovat precizně a vezmeme v úvahu pravděpodobnosti všech rozdělení tříděné posloupnosti, vzorec pro vyjádření rekurze se stane mnohem komplikovanějším, ale konečný výsledek bude podobný.

Věta 5.4 *QuickSort vyžaduje přibližně* $2n \log n$ *porovnání v průměrném případě.*

Důkaz. Přesný rekurzivní vzorec pro počet porovnání v QuickSortu pro náhodnou permutaci n prvků je

$$C_n = n + 1 + \frac{1}{n} \sum_{k=1}^{n} (C_{k-1} + C_{n-k}) \quad \text{pro } n \geq 2, C_1 = C_0 = 0.$$

Výraz $n + 1$ zahrnuje cenu porovnání pivota se všemi ostatními prvky (dvě porovnání navíc jsou potřeba při „překřížení“ posunovaných indexů ve tříděném poli). Zbytek formule vychází z předpokladu, že každý prvek k je možno považovat za pivot s pravděpodobností $1/k$. Tříděná posloupnost se nám tímto rozdělí na dvě podposloupnosti o velikosti $k - 1$ resp. $n - k$.

Ačkoli tento vzorec vypadá složitě, lze jej poměrně snadno ve třech krocích vyřešit. Za prvé, $C_0 + C_1 + \cdots + C_{n-1}$ je to samé jako $C_{n-1} + C_{n-2} + \cdots + C_0$, takže dostáváme

$$C_n = n + 1 + \frac{2}{n} \sum_{k=1}^{n} C_{k-1}.$$

Za druhé se pokusíme eliminovat sumu vynásobením obou stran vztahu n a odečtením stejné formule pro $n-1$:

$$nC_n - (n-1)C_{n-1} = n(n+1) - (n-1)n + 2C_{n-1}$$

Tím se rekurze zjednoduší na

$$nC_n = (n+1)C_{n-1} + 2n.$$

Za třetí, dělením obou stran výrazem $n(n+1)$ dostáváme vztah:

$$\begin{aligned} \frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2}{n+1} \\ &= \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_2}{3} + \sum_{k=3} n\frac{2}{k+1} \end{aligned}$$

Tento přesný vzorec je velice blízký sumě, kterou lze jednoduše aproximovat integrálem:

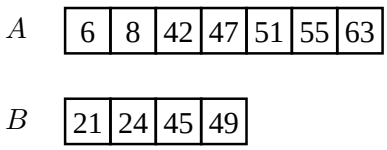
$$\frac{C_n}{n+1} \approx 2\sum_{k=1} n\frac{1}{k} \approx 2\int_1^n \frac{1}{x}dx = 2\ln n$$

Poznamenejme, že $2n\ln n \approx 1,38n\log n$, tudíž průměrný počet porovnání je pouze o 38 procent vyšší než počet porovnání v nejlepším případě. ■

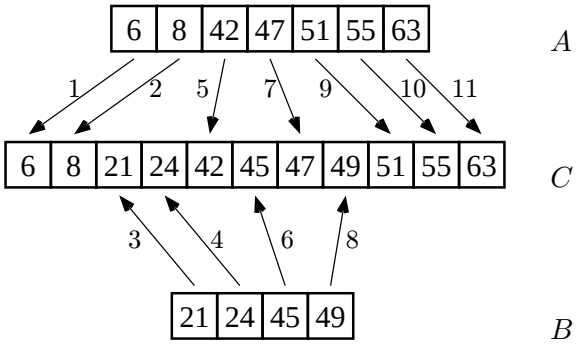
5.5 Třídění slučováním (Mergesort)

Třídění slučováním (česky také někdy označované jako třídění sléváním) představuje efektivní třídící techniku v případě sekvenčního přístupu k tříděným datům. Na rozdíl od již uvedených metod vnitřního třídění se složitostí $O(n^2)$, třídění slučováním má časovou složitost $O(n * \log n)$. Při velikosti vstupních dat 10000 položek ke třídění potřebuje třídící technika s n^2 10^8 časových jednotek, třídění slučováním postačí pouhých 4.10^4. Pro představu - vezmeme-li za časovou jednotku jednu tisícinu sekundy, dostaneme pro třídění slučováním hodnotu 40 sekund, kdežto pro třídící metody se složitostí $O(n^2)$ necelých 28 hodin.

V případě, že máme dostatek místa pro uložení dvou setříděných posloupností položek, je vhodné použít merge sort.



Obrázek 5.37: Vstupní posloupnosti A a B



Obrázek 5.38: Postupné slučování prvků do posloupnosti C

5.5.1 Princip slučování

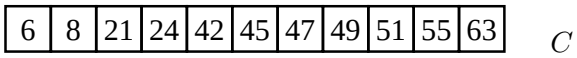
Nejprve si ukážeme princip slučování. Předpokládejme, že chceme spojit dvě posloupnosti, přičemž každá z nich je již sama o sobě setříděná. Výsledkem má být posloupnost, která bude obsahovat všechny prvky setříděné dohromady.

Mějme tedy dvě vzestupně setříděné posloupnosti A a B. Chceme vytvořit vzestupně setříděnou posloupnost C.

Načteme prvek a z posloupnosti A a prvek b z posloupnosti B. Porovnáním hodnot a a b zjistíme, zda $a \leq b$. Platí-li tato nerovnost, do výsledné posloupnosti C zapíšeme hodnotu a a znovu načteme nový prvek z posloupnosti A. Jinak do výsledné posloupnosti zapíšeme hodnotu b a načteme nový prvek z posloupnosti B. Popsané kroky opakujeme tak dlouho, dokud nevyčerpáme všechny prvky z jedné posloupnosti. Zbylé prvky z neprázdné posloupnosti pak už bez dalšího porovnávání můžeme přepsat do výsledné posloupnosti C.

Pořadí označených kroků na obrázku 5.5.1 koresponduje s postupným zařazováním prvků do výsledné posloupnosti C. V tabulce 5.1 jsou pro jednotlivé kroky popsána potřebná porovnávání pro zařazení jednotlivých prvků a je zde také patrné, že po kroku 9 se mohou prvky z posloupnosti A přepsat do posloupnosti C.

Pořadí označených kroků na obrázku 5.5.1 koresponduje s postupným zařazováním prvků do výsledné posloupnosti C. V tabulce 5.1 jsou pro jednot-



Obrázek 5.39: Výsledná posloupnost C

Krok	Porovnání	Zápis
1	6 a 21	Zápis 6 z A do C
2	8 a 21	Zápis 8 z A do C
3	42 a 21	Zápis 21 z B do C
4	42 a 24	Zápis 24 z B do C
5	42 a 45	Zápis 42 z A do C
6	47 a 45	Zápis 45 z B do C
7	47 a 49	Zápis 47 z A do C
8	51 a 49	Zápis 49 z B do C
9		Zápis 51 z A do C
10		Zápis 55 z A do C
11		Zápis 63 z A do C

Tabulka 5.1: Porovnávání a zápis prvků v jednotlivých krocích slučování

livé kroky popsána potřebná porovnávání pro zařazení jednotlivých prvků a je zde také patrné, že po kroku 9 se mohou prvky z posloupnosti A přepsat do posloupnosti C.

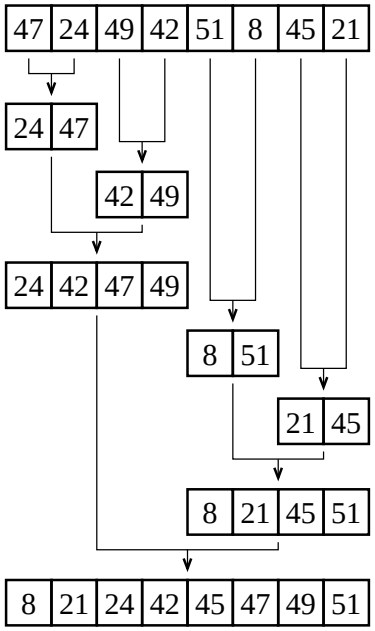
5.5.2 Třídění pomocí slučování

Základní idea třídění pomocí slučováním spočívá v dělení původní posloupnosti na dvě části (nejlépe o polovičním počtu prvků), jejich setřídění a poté použití metody slučování. Výsledkem je setříděná posloupnost o stejném počtu prvků jako byl v původní posloupnosti.

Jak dojde k setřídění dvou rozdělených částí? Opětovným rozdělením na dvě části - v ideálním případě to budou části se čtvrtinovým počtem prvků. Z nich pak pomocí slučování dostaneme setříděné části s poloviním počtem prvků a následně setříděnou celou posloupnost. Rekurentně takto můžeme postupovat dál - z rozdělených „osmin“ obdržíme slučováním setříděné „čtvrtiny“, z těch dalším slučováním „poloviny“ a posledním slučováním celou setříděnou posloupnost.

Kdy bude dělení posloupnosti na menší a menší části končit? Až dojdeme k takovému počtu prvků, které již budou setříděny. Nejmenší setříděnou posloupností je posloupnost jednoprvková[2], tím jsme tedy nalezli i podmínku pro ukončení rekurentního dělení posloupnosti na menší a menší části. Každé

[2]Dělení se obvykle ukončuje při takovém počtu prvků, který je možné setřídit některou metodou vnitřního třídění.



Obrázek 5.40: Princip třídění pomocí slučování

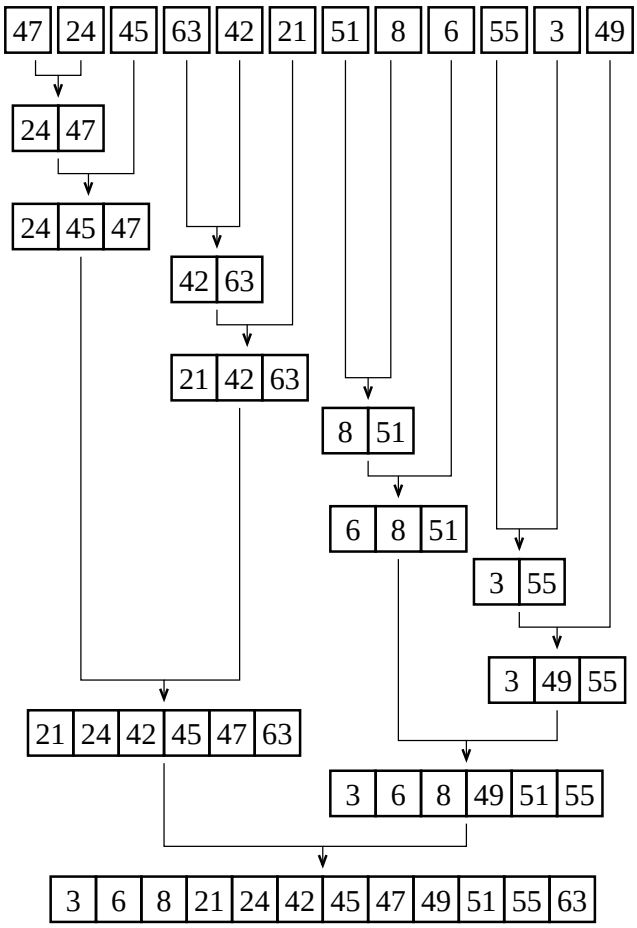
rekurentní volání znamená rozdělení posloupnosti na dvě části a návrat zpět znamená slučování rozdělených (již setříděných) částí do jedné pomocí metody slučování.

Na obrázku 5.40 je naznačen princip výše popsaného postupu. V prvním kroku vezmeme dvě posloupnosti o jednom prvku a sloučením dostaneme setříděnou dvouprvkovou posloupnost. V druhém kroku opět ze dvou jednoprvkových posloupností jednu setříděnou dvouprvkovou a tyto dvě dvouprvkové v dalším kroku sloučíme do setříděné čtveřice. Opakováním postupu dostaneme seřazenou osmiprvkovou posloupnost původních prvků.

Jak bude vypadat případ, kdy počet prvků nebude rekurentně dělitelný dvěma? Na obrázku 5.41 je zobrazen způsob dělení původní posloupnosti na poloviny a v krocích 2, 4, 6 a 8 je patrný postup při nestejném počtu prvků v částech, které se slévají dohromady.

Popsaný algoritmus vystačí s pamětí pro původní posloupnost. Po každém kroku slučování jsou setříděné části posloupnosti ukládány na původní místo. V případě, že množství vstupních dat přesahuje velikost pracovní paměti, je použití popsáno v části **??**.

```
#ifndef MERGESORT_H
#define MERGESORT_H

#include "sort.h"
```

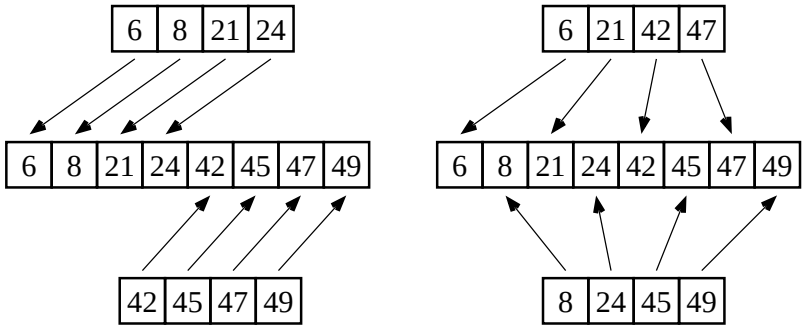


Obrázek 5.41: Mergesort - počet prvků není mocninou 2.

```
void mergesort( Item* to, int n, ItemCompare cmp );

#endif


#include "mergesort.h"

void merge(
  Item* from1, int m1,
  Item* from2, int m2,
  Item* to,
  ItemCompare cmp
)
{
  int i1 = 0;
  int i2 = 0;
  int j = 0;
  while (i1 < m1 && i2 < m2)
    to[j++] = cmp( from1[i1], from2[i2] ) != GREATER ? from1[i1++] : from2[i2
        ++];
  while (i1 < m1) to[j++] = from1[i1++];
  while (i2 < m2) to[j++] = from2[i2++];
}

void mergesort_rec( Item* from, Item* to, int n, ItemCompare cmp )
{
  if (n >= 2)
  {
    int m1 = n/2;
    int m2 = n-m1;
    mergesort_rec( to,     from,     m1, cmp );
    mergesort_rec( to+m1, from+m1, m2, cmp );
    merge( from, m1, from+m1, m2, to, cmp );
  }
}

void mergesort( Item* to, int n, ItemCompare cmp )
{
  Item* from = (Item*)calloc( n, sizeof( Item ) );
  int i;
  for (i = 0; i < n; i++)
    from[i] = to[i];
  mergesort_rec( from, to, n, cmp );
  free( from );
}
```

Analýza

Hodnocení třídění slučováním si ukážeme na konkrétním případu třídění několika prvků a poté zobecněné hodnoty uvedeme v tabulce.



Obrázek 5.42: Mergesort - nejlepší a nejhorší případ pro operace porovnání.

Operace přesunu prvku a porovnání budeme sledovat pro určení časové složitosti algoritmu. Rekurentní volání a návrat nebudeme zahrnovat, neboť jejich náročnost na čas je ve srovnání se sledovanými operacemi menší.

Počet přesunů prvků je možné určit z uvedeného příkladu, zobrazeného na obrázku 5.40. Pro tříděných 8 prvků potřebujeme 24 operací přesunu. Pro každý z prvků jsme potřebovali jeden přesun na třech úrovních. Je patrné, že po každé úrovni se velikost setříděných částí zdvojnásobí, takže pro 8 prvků stačí zopakovat slučování třikrát. V první úrovni máme 8 jednoprvkových částí, ve druhé 4 dvouprvkové a ve třetí jednu osmiprvkovou setříděnou posloupnost. Odtud tedy vzorec pro výpočet počtu přesunů

$$n \log n = 8 \log_2 8 = 24.$$

Do celkového počtu přesunů je nutno připočítat stejný počet operací, neboť tolikrát zapisujeme zpět na původní místo vstupních dat. V tabulce 5.2 je patrný výpočet počtu porovnání pro velikost vstupních dat, která je násobkem 2.

Počet porovnání prvků určuje vzájemné pořadí prvků v částech, které se postupně porovnávají. Kolik operací porovnání může proběhnout? Vezmeme-li náš příklad o 8 prvcích, může nastat situace, kdy všechny prvky z jedné posloupnosti budou menší než nejmenší prvek z druhé posloupnosti a tento případ povede k pouhým čtyřem operacím porovnání. Můžeme tedy říct, že nejmenší možný počet porovnání je shodný s počtem prvků posloupnosti, která na vstupu obsahuje prvky s menšími hodnotami.

Naproti tomu, jak bude vypadat nejhorší případ? Taková situace nastane, když zápis prvků bude probíhat střídavě z posloupností A a B. Dostáváme tedy maximální počet porovnání, který bude roven $n - 1$. Pro uvedený příklad je to 7 operací porovnání. Průběh porovnávání v nejlepším a nejhorším případě je znázorněn na obrázku 5.42.

n	Počet přesunů na prac. místa	Počet přesunů celkem	Počet porovnání minimum/maximum
2	2	4	1/1
4	8	16	4/5
8	24	48	12/17
16	64	128	32/49
32	160	320	80/129
64	384	768	192/321
128	896	1792	448/769

Tabulka 5.2: Počet sledovaných operací (n je násobkem 2)

Počet porovnání C_{max} pro sloučení dvou setříděných posloupností v jedné fázi v nejhorším případě je

$$C_{max} = n - 1$$

a počet porovnání C v nejhorším případě při vstupu n

$$C = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

Důkaz indukcí lze nalézt v [13].

Počet přesunů při třídění slučováním je určen počtem fází a počtem prvků na vstupu. Počet fázíje roven počt dělení vstupní posloupnosti na nejmenší část, která je vstupem pro fázi slučování.

$$M = n \log n$$

Z předešlých úvah vidíme, že třídění slučováním patří mezi metody s časovou složitostí $O(n \log_2 n)$.

5.5.3 Použití třídění slučováním u sekvenčního zpracování dat

Princip slučování setříděných úseků dat je vhodné využít v případech, kdy vstupní data jsou uložena na vnějších paměťových médiích a přistupovat k datům je možné pouze sekvenčně, případně se princip mergesortu využívá při zpracování velkého objemu dat[3]. Mohou nastat situace, kdy není možné všechna data najednou setřídit ve vnitřní paměti, případně slučovaných posloupností je více než dvě.

Ukážeme si způsob použití třídění slučováním pro dva vstupní streamy a různý počet pomocných streamů k ukládání setříděných úseků. Počet pomocných streamů ovlivňuje způsob fázování při běhu algoritmu (obrázek

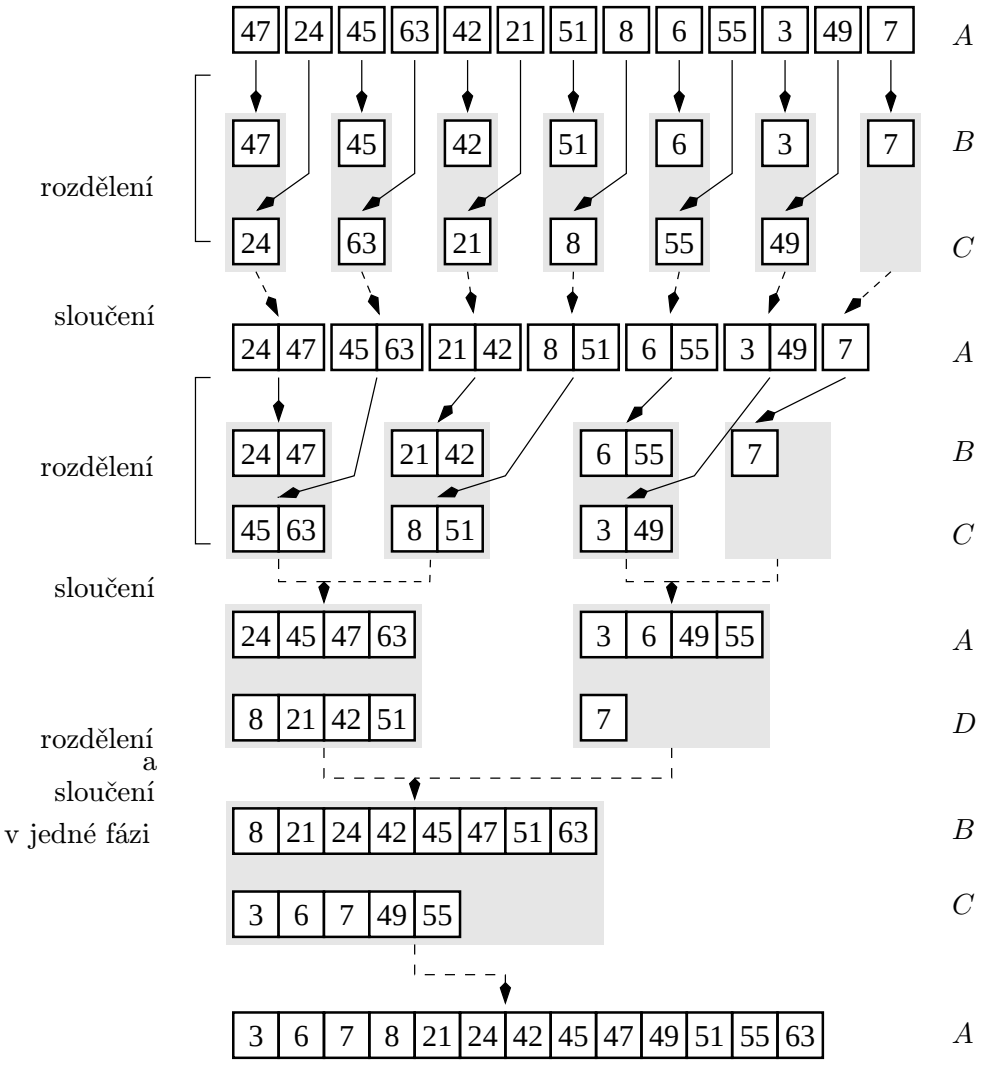
[3]Objem dat přesahuje velikost operační paměti.

5.43). Na obrázku je naznačena první fáze *rozdělování*, kdy prvky streamu A při načítání zapisujeme do dvou pomocných streamů B a C. Protože dopředu nevíme, kolik prvků na vstupu bude, naskýtá se použití techniky rozdělení na dvě poloviny naznačeným způsobem - prvky na lichých pozicích zapisujeme do prvního pomocného streamu, prvky na sudých pozicích do druhého. Tím docílíme rozdělení prvků na polovinu nebo se počet rozdělených prvků bude lišit maximálně o jeden.

Poté následuje fáze *slučování*, kterou již známe. Každý prvek v B a C představuje setříděnou posloupnost. Přistoupíme k jejich sloučení porovnáním a do streamu A zapisujeme setříděné dvojice. Situace se opakuje — opět nastane fáze rozdělení, tentokrát již střídavě do B a C zapisujeme *dvojice* a v další fázi slučování dojde ke sloučení dvouprvkových uspořádaných posloupností do *čtveřic*. Zde je ukázán druhý přístup, kdy fáze rozdělení a slučování provádíme v jednom kroku. První setříděná čtveřice je zapsána do streamu A, druhá pak do streamu D, třetí do A, čtvrtá do D. S poslední čtveřicí se zachází stejně i přesto, že není úplná — obsahuje pouze jeden prvek, ale princip slučování do dvojnásobně velkých *n-tic* po každá fázi slučování zůstává zachován. Obě fáze dohromady nazýváme prostě *fází*.

Ve zbývajících dvou fázích sloučíme čtveřice do osmiprvkových setříděných posloupností a poté do „neúplné“ šestnáctiprvkové posloupnosti, která již obsahuje všechny prvky ze vstupního streamu a tím algoritmus končí. Použití fází odděleně nebo rozdělování a slučování v jedné fázi závisí na možnostech v konkrétních případech. Zde byl pouze předveden dvojí přístup použití mergesortu v případě tří nebo čtyř pomocných streamů.

Složitost se ani u tohoto způsobu použití nemění, neboť během každé fáze potřebujeme $2n$ krát číst a $2n$ krát zapisovat do streamu. Celkem jde tedy o $4n$ operací přístupu. Protože po každé fázi získáme dvojnásobnou délku setříděného úseku, potřebujeme pro setřídění celého streamu fáze opakovat $\log_2 n$ - krát. Odtud plyne, že složitost algoritmu je opět $O(n \log_2 n)$.



Obrázek 5.43: Mergesort - verze pro tři nebo čtyři streamy.

Kapitola 6

Nelineární datové struktury

6.1 Volné stromy

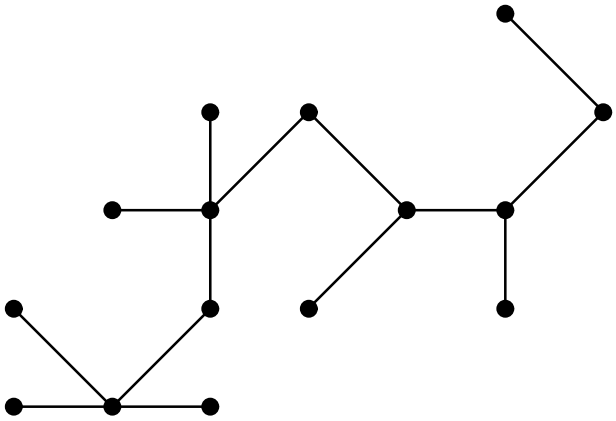
Definice 6.1 *Souvislý, acyklický, neorientovaný graf nazýváme* ***volným stromem*** *(angl. free tree).*

Často vynecháváme adjektivum „volný“, a říkáme jen, že daný graf je strom. Příklad volného stromu je na obrázku 6.1. Následující věta popisuje mnoho důležitých vlastností volných stromů.

Věta 6.1 *Nechť $G = (V, E)$ je neorientovaný graf, potom následující tvrzení jsou ekvivalentní:*

1. *G je volný strom.*
2. *Každé dva uzly v G jsou spojeny právě jednou cestou.*
3. *G je souvislý, ale pokud odebereme libovolnou hranu, získáme nesouvislý graf.*
4. *G je souvislý, a $|E| = |V| - 1$.*
5. *G je acyklický, a $|E| = |V| - 1$.*
6. *G je acyklický. Přidáním jediné hrany do množiny hran E bude výsledný graf obsahovat kružnici.*

Důkaz. (1)⇒(2): Jelikož strom je souvislý, libovolné dva uzly v G jsou spojeny nejméně jednou jednoduchou cestou. Nechť u a v jsou dva vrcholy, které jsou spojeny dvěmi různými cestami p_1 a p_2. Nechť w je první vrchol ve kterém se cesty rozdělují, jinými slovy w je první vrchol na cestách p_1 a p_2 jehož následovník na p_1 je x a následovník na p_2 je y, kde $x \neq y$. Nechť z je první vrchol ve kterém se cesty p_1 a p_2 znovu sbíhají. Vrchol z je tedy první společný následovník w na obou cestách p_1 a p_2. Nechť p' je podcesta z w skrze x do z a p'' je podcesta z w skrze y do z. Cesty p'



Obrázek 6.1: Volný strom

a p'' mají společné jedině své koncové body w a z. Cesta, kterou dostaneme spojením p' a p'' v obráceném pořadí obsahuje kružnici. Což je spor s definicí volného stromu. Proto, je-li G strom, každé dva vrcholy jsou spojeny jednou jednoduchou cestou.

(2)⇒(3): Jestliže libovolné dva vrcholy v G jsou spojeny jednou jednoduchou cestou, potom G je souvislý. Nechť (u, v) je libovolná hrana z E. Tato hrana je zároveň i cestou z u do v a tudíž tato cesta musí být jedinou z u do v. Jestliže cestu (u, v) vyjmeme z G, pak z u do v nepovede žádná cesta, to jest zrušením hrany se graf G stane nesouvislým.

(3)⇒(4): Za předpokladu, že graf G je souvislý platí vztah $|E| \geq |V|-1$. Nyní indukcí dokážeme, že $|E| \leq |V| - 1$. Spojitý graf s $n = 1$ nebo $n = 2$ vrcholy má $n - 1$ hran. Předpokládejme, že graf G má $n \geq 3$ vrcholů a že všechny grafy s menším počtem vrcholů splňují jednak vlastnost (3) a jednak $|E| \leq |V| - 1$. Odstranění libovolné hrany z G rozdělí graf na $k \geq 2$ souvislých komponent (v tomto okamžiku $k = 2$). Každá komponenta splňuje (3), jinak by celý G nesplňoval (3). Indukcí, počet hran ve všech komponentách je nejvýše $|V| - k \leq |V| - 2$. Připočtením odstraněné hrany dostáváme $|E| \leq |V| - 1$.

(4)⇒(5): Předpokládejme, že graf G je souvislý a platí $|E| = |V| - 1$. Musíme dokázat, že graf G je acyklický. Důkaz provedeme sporem. Předpokládejme, že graf G obsahuje kružnici tvořenou k vrcholy $v_1, v_2, \ldots, v_k$. Nechť $G_k = (V_k, E_k)$ je podgraf G obsahující pouze tuto kružnici. Poznamenejme, že platí $|V_k| = |E_k| = k$. Jestliže $k \leq |V|$, pak musí existovat vrchol $v_{k+1} \in V - V_k$, který je připojen k některému vrcholu $v_i \in V_k$, protože graf G je souvislý. Definujme $G_{k+1} = (V_{k+1}, E_{k+1})$ jako podgraf G, kde $V_{k+1} = V_k \cup \{v_{k+1}\}$ a $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Platí $|V_{k+1}| = |E_{k+1}| = k+1$. Je-li $k+1 \leq n$ můžeme pokračovat definováním G_{k+2} stejným způsobem dokud nedostaneme $G_n = (V_n, E_n)$, kde $n = |V|$, $V_n = V$ a $|E_n| = |V_n| = |V|$.

Jelikož G_n je podgraf G, platí $E_n \subseteq E$ a odtud $|E| \geq |V|$, což je spor s předpokladem, že $|E| = |V| - 1$. Proto graf G je acyklický.

(4)⇒(5): Předpokládejme, že graf G je acyklický a platí $|E| = |V| - 1$. Nechť k je počet souvislých komponent grafu G. Podle definice je každá souvislá komponenta volný strom a jelikož (1) implikuje (5), součet všech hran ve všech souvislých komponentách grafu G je roven $|V| - k$. Protože graf G je strom, musí platit, že $k = 1$. Ze (2) plyne, že každá dva vrcholy v G jsou spojeny jednou jednoduchou cestou. Proto přidáním hrany se v grafu G vytvoří kružnici.

(6)⇒(1): Předpokládejme, že G je acyklický graf, ale přidáním libovolné hrany do E vznikne v grafu kružnice. Musíme dokázat, že G je souvislý graf. Nechť u a v jsou libovolné dva vrcholy z G. Jestliže u a v dosud nejsou spojeny cestou, přidáním hrany (u, v) vznikne kružnici, ve kterém všechny hrany vyjma (u, v) patří do G. Proto, existuje cesta z u do v a protože u a v byly vybrány libovolně, G je souvislý. ■

6.2 Kořenové stromy a seřazené stromy

Kořenový strom (angl. rooted tree) je volný strom, který obsahuje jeden odlišný uzel. Tento odlišný uzel se nazývá **kořen**.

Uvažujme uzel x v kořenovém stromu T s kořenem r. Libovolný uzel y na jednoznačné cestě od kořene r do uzlu x se nazývá **předchůdce** uzlu x. Jestliže y je předchůdce x, potom x se nazývá **následovník** uzlu y. Každý uzel je pochopitelně předchůdcem a následovníkem sama sebe. Jestliže y je předchůdce x a zároveň $x \neq y$, potom y je **vlastní předchůdce** uzlu x a x je **vlastní následovník** uzlu y.

Jestliže poslední hrana na cestě z kořene r do uzlu x je hrana (y, x), potom se uzel y nazývá **rodič** uzlu x a uzel x je **potomek** uzlu y. Kořen stromu je jediným uzlem ve stromu bez rodiče. Dva uzly mající stejného rodiče se nazývají **sourozenci**. Uzel bez potomků se nazývá **externí uzel** nebo-li **list**. Nelistový uzel se je **vnitřním** uzlem.

Počet potomků uzlu x v kořenovém stromu se nazývá **stupeň** uzlu x. Poznamenejme, že stupeň uzlu závisí na tom, zda strom T uvažujeme jako volný strom nebo jako kořenový strom. V prvním případě, je stupeň počet sousedních uzlů. V kořenových stromech je stupeň definován jako počet potomků – tedy rodič uzlu se nebere v úvahu. Délka cesty od kořene k uzlu x se nazývá **hloubka** uzlu x ve stromu T. Největší hloubka libovolného uzlu se nazývá **výška** stromu T.

Seřazený strom (angl. ordered tree) je kořenový strom, ve kterém jsou potomci každého uzlu seřazeni. Tudíž, pokud uzel má k potomků, lze určit prvního potomka, druhého potomka, až k-tého potomka.

6.3 Binární stromy

Binární strom lze nejlépe definovat rekurzivně:

Definice 6.2 ***Binární strom*** *je struktura definovaná nad konečnou množnou uzlů, která:*

- *neobsahuje žádný uzel*
- *je složena ze tří disjunktních množin uzlů: kořene, binárního stromu zvaného* ***levý podstrom*** *a binárního stromu tzv.* ***pravého podstromu****.*

Binární strom, který neobsahuje žádný uzel se nazývá **prázdný** strom. Jestliže levý podstrom je neprázdný, jeho kořen je **levým potomkem** kořene celého stromu. Stejně tak, kořen neprázdného pravého podstromu se nazývá **pravým potomkem** kořene daného stromu.

Binární strom není jen seřazený strom, ve kterém má každý uzel stupeň nejvýše dva. Například, v binárním stromu, jestliže uzel má pouze jednoho potomka, potom fakt, že potomek je levý nebo pravý je důležitý. V seřazeném stromu není u jediného potomka možnost rozlišit, zda je pravý nebo levý.

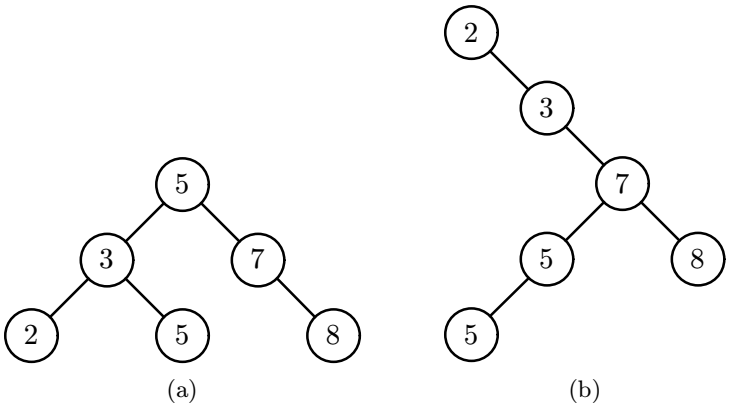
Umístění informace v binárním stromu lze reprezentovat pomocí vnitřních uzlů. Chybějící potomci se nahradí uzly bez potomků. Takto vzniklý strom se nazývá **úplný binární strom**. Každý uzel (včetně listů) v takovém stromu má stupeň právě dva.

Umísťování informace odlišující binární stromy od seřazených stromů, lze rozšířit na stromy s více než dvěma potomky uzlu. V **pozičním** stromu (angl. positional tree) jsou jednotliví potomci očíslováni kladným celým číslem. Říkáme, že i-tý potomek chybí, jestliže neexistuje potomek označen číslem i. n-ární strom je poziční strom, kde v každém uzlu, všichni potomci s číslem vyšším než n chybějí. Proto binární strom je n-ární strom s $n = 2$.

Úplný n-ární strom je n-ární strom, ve kterém všechny listy mají stejnou hloubku a všechny vnitřní uzly mají stejný stupeň n. Počet listů se dá určit jednoduše. Kořen stromu má n potomků s hloubkou 1, z nichž každý má n potomků s hloubkou 2 atd. Tedy, počet listů v hloubce h je n^h. Následně, hloubka úplného n-árního stromu s m listy je $\log_n m$. Počet interních uzlů úplného n-árního stromu výšky h je

$$1 + n + n^2 + \cdots + n^{h-1} = \sum_{i=0}^{h-1} n^i = \frac{n^h - 1}{n - 1}$$

Odtud je zřejmé, že binární strom má $2^h - 1$ interních uzlů.



Obrázek 6.2: Binární vyhledávací stromy

Pro každý uzel x jsou klíče v jeho levém podstromu menší nebo rovny klíči v x a klíče v jeho pravém podstromu jsou větší nebo rovny klíči v x. Shodnou množinu prvků lze reprezentovat různými stromy. Časová složitost vyhledávání v nejhorším případě je úměrná výšce stromu. **(a)** Binární strom se 6 uzly výšky 2. **(b)** Méně efektivní binární strom s výškou 4, který obsahuje tytéž hodnoty.

6.4 Binární vyhledávací stromy

Binární vyhledávací strom[1] je organizován, jak název napovídá, jako binární strom, například jako na obrázku 6.2.

Binární vyhledávací stromy se nejčastěji implementují pomocí dynamických struktur. Každý uzel potom obsahuje nějaký klíč, na jehož doméně je definováno nějaké uspořádání. Dále každý uzel obsahuje ukazatele *left* a *right* na svého levého a pravého potomka. Jestliže některý potomek chybí, je patřičná položka nastavena na NULL. Uzel samozřejmě může obsahovat i další data v závislosti na povaze aplikace.

Klíče v binárním vyhledávacím stromu jsou vždy uspořádány tak, že splňují vlastnost binárních vyhledávacích stromů:

Definice 6.3 *Nechť x je uzel v binárním stromu. Jestliže y je z levého podstromu uzlu x, potom* klíč$[y] \leq$ klíč$[x]$. *Jestliže y je z pravého podstromu uzlu x, potom* klíč$[x] \leq$ klíč$[y]$.

Proto na obrázku 6.2(a), klíč kořene je 5, klíče 2, 3 a 5 v levém podstromu nejsou větší než 5, klíče 7 a 8 v pravém podstromu nejsou menší než 5. Shodná vlastnost platí pro každý uzel ve stromu. Například klíč 3 na obr. 6.2 není menší než klíč 2 v jeho levém podstromu a není větší než klíč 5 v pravém podstromu.

[1] V dalším textu budeme mluvit jen o binárních stromech, ale budeme uvažovat vždy binární vyhledávací strom.

6.4.1 Vyhledávání v binárním stromu

Nejčastěji prováděnou operací na binárních stromech je vyhledání prvku, zda je či není ve stromu, jinými slovy jde o test přítomnosti prvku v množině reprezentované stromem. Mimo tuto operaci *Search* lze na binárních stromech implementovat mimo jiné například i operace *Minimum*, *Maximum*. Všechny zmíněné operace pracují v čase $O(h)$, kde h je výška stromu.

Vyhledávání

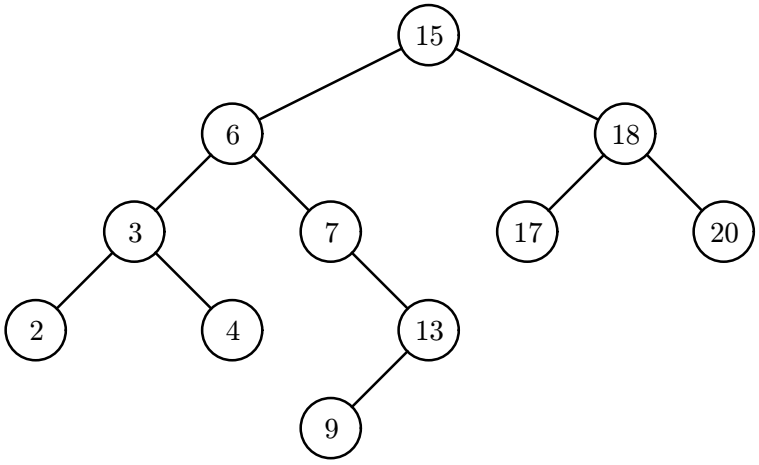
Metoda vyhledání daného prvku v binárním stromu, lze nejjednodušeji realizovat, jak je u stromů obvyklé, rekurzivně. Mějme dánu množinu M reprezentovanou binárním stromem (viz B.1) a jeho kořen r. Dále mějme dán prvek s klíčem k, který hledáme. Základem vyhledávací procedury je uspořádání klíčů ve stromu. Hledání zahájíme v kořeni stromu r. Potom mohou nastat tyto možnosti:

1. Strom s kořenem r je prázdný ($r = NULL$), potom tento strom nemůže obsahovat prvek s klíčem k a hledání končí neúspěchem. Platí tedy $k \notin M$.
2. V opačném případě srovnáme klíč k s klíčem kořene právě zkoumaného stromu resp. jeho podstromu r. V případě, že
   (a) $k =$ klíč(r) strom obsahuje prvek s klíčem k. Platí $k \in M$. Hledání končí úspěšně;
   (b) $k <$ klíč(r) vzhledem k vlastnostem binárních vyhledávacích stromů, jsou všechny prvky s klíči menšími než je klíč r v jeho levém podstromu, pokračujeme rekurzivně v levém podstromu;
   (c) $k >$ klíč(r) na rozdíl od předchozího případu jsou všechny prvky s klíči většími než je klíč r v pravém podstromu, pokračujeme pravým podstromem.

Stručně lze tento algoritmus vyjádřit následujícím pseudokódem:

```
bool Search(CNode* p, T k)
{
  if (p == NULL)
    return false;
  if (k == p->key)
    return true;
  if (k < p->key)
    Search(p->left, k);
  else
    Search(p->right, k);
}
```

Stejnou proceduru lze napsat iterativně pomocí cyklu `while`. Na většině počítačů bude nerekurzivní verze efektivnější díky režii nutné pro volání funkce. Finální kódy obou funkcí jsou uvedeny v části B.1.



Obrázek 6.3: Vyhledávání v binárním stromu

Při hledání prvku 13 jsou postupně navštíveny uzly 15, 6, 7 a 13. Minimální prvek 2 lze najít sestupováním po stromu podél ukazatelů na levý podstrom. Obdobně maximum je v našem stromu 20, které lze najít sledováním ukazatelů *right*.

```
bool IterativeSearch (CNode* p, T k)
{
  while (p != NULL)
  {
    if (k == p->key)
      return true;      // nalezeno
    if (k < p->key)
      p = p->left;
    else
      p = p->right;
  };  // while
  return false;          // p == NULL nenalezeno
}
```

Minimum a maximum

Prvek jehož klíč je minimem z množiny reprezentované daným stromem, lze v binárním stromu velice lehce najít sledováním pointerů *left* od kořene až k listu. Prvek v nejlevějším listu je pak hledaným minimem.

```
T Minimum()
{
  p = m_root;
  while (p->left != NULL)
    p = p->left;
  return p->key;
}
```

Binární vyhledávací strom zaručuje, že postup vyhledání minima je korektní. Jestliže uzel x nemá levý podstrom, potom všechny klíče v pravém podstromu musí být větší nebo rovny než klíč x. Minimum stromu s kořenem x je pak klíč kořene x. Jinak jestliže x má levý podstrom (pravý podstrom viz předchozí případ), potom klíče v levém podstromu musí být menší nebo rovny než klíč x, tudíž minimum stromu s kořenem x se nachází v jeho levém podstromu.

Pro maximum je kód symetrický. Obě funkce pracují v čase $O(h)$, kde h je výška stromu. Kompletní kódy obou funkcí jsou uvedeny v části B.1.

6.4.2 Vkládání do binárního stromu

Vkládání do binárního vyhledávacího stromu probíhá obdobně jako vyhledávání v takovém stromu. Nejprve je nutno určit kam vkládaný prvek přijde. Vzhledem k uspořádání klíčů ve stromu je takové místo určeno jednoznačně. Stejně jako při vyhledávání sestupujeme rekurzivně od kořene dolů směrem k listům. Vkládaný prvek x porovnáme s kořenem r zkoumaného podstromu. Mohou nastat tyto případy:

1. strom s kořenem r je prázdný ($r = NULL$), potom tento strom nemůže obsahovat prvek s klíčem k a hledání by v tomto okamžiku skončilo neúspěchem. Zároveň jsme však našli místo, kam lze prvek x vložit. Vytvoříme proto nový uzel (který je pochopitelně listem) a tento uzel připojíme k předchozímu uzlu.

2. V opačném případě, srovnáme klíč k s klíčem kořene právě zkoumaného stromu resp. jeho podstromu r. V případě, že

   (a) klíč(x) < klíč(r) pokračujeme rekurzivně levým podstromem;

   (b) klíč(x) = klíč(r) prvek x byl nalezen ve stromu. Záleží na konkrétní aplikaci, jak naloží s duplicitními výskyty prvků. Pokud například počítáme, kolik různých slov se vyskytuje v textu, lze duplicitní výskyty ignorovat. V případě, že bychom počítali například četnosti slov v textu, udržovali bychom ke každému slovu počítadlo a v tomto případě by se počitadlo inkrementovalo;

   (c) klíč(x) > klíč(r) pokračujeme rekurzivně pravým podstromem.

Stručně lze tento algoritmus vyjádřit následujícím pseudokódem:

```
void TreeInsert (CNode*& p, T k)
{
  if (p == NULL)
  {
    p = new Node;
    p->key = k;
    p->left = p->right = NULL;
    return;
```

```
    };
    if (k == p->key)
    {// duplicitní klíč
      return;
    };
    if (k < p->key)
      TreeInsert (p->left, k);
    else
      TreeInsert (p->right, k);
  }
```

Jak je patrno, vkládání se od vyhledávání liší jen činností při $p == NULL$. Je třeba připomenout, že při vkládání je nutno parametr p předávat odkazem (referencí) nikoliv hodnotou, protože se v průběhu výpočtu může změnit (vytvořením nového uzlu) a tuto změnu je nutno po ukončení volání funkce zachovat. Uvědomme si, že tímto způsobem je vyřešeno navázání nového uzlu na jeho rodiče. Jinými slovy: jestliže jsme rekurzí sestoupili do nějakého uzlu a a pokračujeme dále například pravým podstromem, potom je jako parametr p použit ukazatel na pravý podstrom uzlu a a předpokládejme, že tento ukazatel je NULL. Potom v následujícím vyvolání TreeInsert, se alokuje nový uzel a ukazatel na něj je uložen do p, které je díky mechanismu volání odkazem totožný s ukazatelem $a \rightarrow right$. Tudíž se automaticky mění i pravý ukazatel rodiče nového uzlu. Kdybychom použili volání hodnotou, ukazatel na nový uzel by se automaticky „zapomněl“.

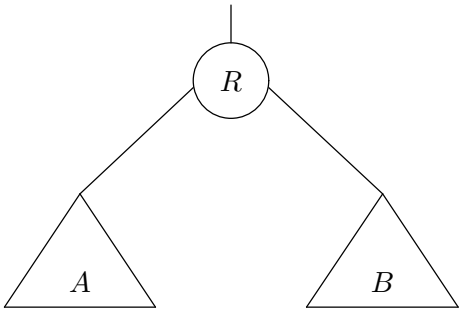
Iterativní verze téhož algoritmu je možná, ale je složitější než iterativní verze vyhledávání, protože je nutné udržovat dva ukazatele: jeden na aktuální uzel a druhý na jeho rodiče, právě kvůli navázání nového uzlu do stromu.

Finální kódy obou funkcí jsou uvedeny v části B.1.

6.4.3 Rušení uzlů v binárním stromu

Rušení uzlů v binárním stromu je vhodné řešit opět pomocí rekurze. Nejdříve je nutné rušený prvek nalézt ve stromu. Postupujeme obdobně jako v případě vyhledávání, tj. sestupujeme rekurzivně od kořene dolů, směrem k listům. Pokud rušený prvek ve stromu není, procedura končí bez jakékoliv činnosti. Jinak předpokládejme, že rušíme uzel x. Naše další činnost bude záviset na počtu potomků uzlu x. Pokud má x

- **nula potomků** – to znamená, že uzel x je list a lze jej snadno od stromu „odříznout“;
- **jednoho potomka** – uzel x již nelze snadno od stromu oddělit, protože odříznutím uzlu x bychom ztratili i jeho potomka. Vezmeme tedy potomka uzlu x, přičemž je celkem lhostejné, zda je to potomek levý nebo potomek pravý a napojíme na něj ukazatel z rodiče uzlu x. Jinými slovy uzel x „obejdeme“, čímž se bezpečně vypojí ze stromu a můžeme jej uvolnit z paměti.



Obrázek 6.4: Binární strom

- **dva potomky** – v tomto nejsložitějším případě musíme nahradit uzel x jeho předchůdcem resp. následovníkem, ve smyslu uspořádání klíčů uzlů. Předpokládejme, že budeme nahrazovat předchůdcem. Předchůdce uzlu x je nutno hledat v jeho levém podstromu, kde jsou uloženy všechny prvky s klíči menšími než je klíč x. Jelikož předchůdcem rozumíme nejbližší menší prvek než x, nejblíže prvku x bude právě maximum ze všech prvků v levém podstromu uzlu x. Maximum ve stromu se nalézá v jeho nejpravějším uzlu. Stručně řečeno, hledáme nejpravější uzel z levého podstromu uzlu x. Tímto uzlem nahradíme uzel x.

Kód funkce pro rušení uzlu v binárním stromu je uveden v kapitole B.1.

6.4.4 Další operace nad binárním stromem

Existuje mnoho úloh, které lze řešit na stromové struktuře; nejběžnější je uskutečnění dané operace P na každém uzlu stromu. Operaci P chápeme jako parametr všeobecnější úlohy navštívení každého uzlu stromu, což se obvykle nazývá **průchod stromem**.

Jestliže se na tuto úlohu díváme jako na jednoduchý sekvenční proces, je jasné, že jednotlivé uzly ve stromu se budou navštěvovat v určitém specifickém pořadí a je možné si představit, jako kdyby uzly stromu byly lineárně uspořádané.

Rozlišujeme tři základní uspořádání, která jsou přirozeným důsledkem stromové struktury. Podobně jako samotný strom se i tato uspořádání dají jednoduše vyjádřit rekurzivně. Podle obrázku 6.4 (kde R označuje kořen stromu, A a B jeho levý resp. pravý podstrom) lze tři uspořádání psát takto:

1. **Přímé** (*preorder*): R, A a B — nejprve byl navštíven kořen pak jeho podstromy

2. **Vnitřní** (*inorder*): A, R a B — nejprve levý podstrom, kořen a nakonec pravý podstrom

3. **Zpětné** (*postorder*): A, B a R — kořen se navštíví až po podstromech.

Tato schémata se rekurzivně aplikují na celý strom. Je například zřejmé, že pokud použijeme průchod inorder budou jednotlivé uzly navštíveny v souladu s uspořádáním klíčů ve stromu. Průchod postorder je vhodný například v destruktoru třídy při dealokaci celého stromu z paměti, kde je zcela evidentně potřeba dealokovat jednotlivé podstromy a teprve potom je možno zrušit daný uzel Při jiném pořadí by se ukazatele na části stromu ztratily.

Příklad 6.1
Mějme napsat funkci, která spočítá uzly ve stromu. Předpokládejme, že binární strom je definován způsobem uvedeným v sekci B.1. Naše úloha se výrazně zjednoduší uvědomíme-li si její rekurzivní charakter (použijeme opět obrázek 6.4 a předpokládáme, že aktuální uzel je R):

- Je-li R prázdný strom (tj. $R = NULL$), pak počet jeho uzlů je pochopitelně nula. Tím máme problém vyřešen.

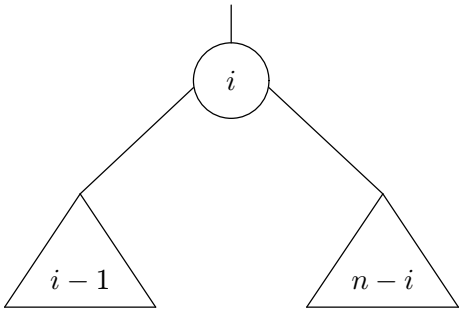
- V opačném případě víme, že ve stromu určitě jeden uzel existuje (R) a počty uzlů v levém a pravém podstromu se dají určit obdobným způsobem rekurzivně. To znamená, že počet uzlů ve stromu s kořenem R je $1 +$ počet_uzlů$(A) +$ počet_uzlů(B)

Počty uzlů pro jednotlivé podstromy se předávají jako výsledky volání funkcí prostřednictvím zásobníku programu, nejsou tudíž potřeba žádné pomocné proměnné. Výsledný kód v *C++* je uveden v kapitole B.1.

6.4.5 Analýza vyhledávání a vkládání

Starosti při použití binárního stromu způsobuje především skutečnost, že se dost dobře neví, jak bude strom narůstat; neexistuje v podstatě žádná přesná představa o jeho tvaru. Jediné co se dá předpokládat je, že to nebude dokonale vyvážený strom (viz kapitola 6.5). Průměrný počet porovnání potřebných pro lokalizaci klíče v dokonale vyváženém stromu je přibližně $h = \log n$. Průměrný počet porovnání našem případě bude určitě větší než h. Je otázkou o kolik bude větší.

Není těžké najít nejhorší případ. Předpokládejme, že všechny klíče jsou již setříděny (ať už sestupně nebo vzestupně). Každý klíč je při budování stromu připojen nalevo (resp. napravo) ke svému rodiči a výsledný strom bude degenerovaný, jinými slovy stane se z něj lineární seznam. Vyhledávání v takovém případě bude vyžadovat $n/2$ porovnání. V dalším textu budeme zkoumat průměrnou délku cesty vyhledávání vzhledem ke všem n klíčům a všem $n!$ stromům, které lze vygenerovat z $n!$ permutací n klíčů.



Obrázek 6.5: Rozdělení vah v podstromech

Mějme n různých klíčů s hodnotami $1, 2, \ldots, n$. Předpokládejme, že klíče nejsou uspořádané. Pravděpodobnost, že první klíč, který se pochopitelně stane kořenem stromu, bude mít hodnotu i je $1/n$. Jeho levý podstrom bude obsahovat $i - 1$ uzlů, pravý podstrom $n - i$ uzlů.

Označme průměrnou délku cesty v levém podstromu symbolem a_{i-1}, v pravém podstromu symbolem a_{n-i}. Předpokládejme dále, že všechny možné permutace zbývajících $n - 1$ klíčů budou stejně pravděpodobné.

Průměrná délka cesty ve stromě s n uzly je daná součtem součinů čísla úrovně každého uzlu a pravděpodobnosti přístupu k jednotlivým uzlům. Jestliže předpokládáme, že tato pravděpodobnost bude shodná pro všechny uzly, potom platí

$$a_n = \frac{1}{n} \sum_{i=1}^{n} p_i \tag{6.1}$$

kde p_i je délka cesty pro i-tý uzel stromu.

Uzly stromu, znázorněného na obrázku 6.5, můžeme rozdělit na tři třídy:

1. Průměrná délka cesty pro $i - 1$ uzlů v levém podstromu je $a_{i-1} + 1$.
2. Délka cesty kořene stromu je 1.
3. Průměrná délka cesty pro $n - i$ uzlů v pravém podstromu je $a_{n-i} + 1$.

Vztah (6.1) můžeme vyjádřit součtem tří výrazů:

$$a_n^{(i)} = (a_{i-1} + 1)\frac{i-1}{n} + 1\frac{1}{n} + (a_{n-i} + 1)\frac{n-i}{n} \tag{6.2}$$

Hledanou průměrnou délku cesty a_n je možno odvodit jako průměrnou hodnotu $a_n^{(i)}$ pro všechna $i = 1, 2, \ldots, n$. to jest pro všechny stromy s klíči $i = 1, 2, \ldots, n$ v jejich kořenech.

$$\begin{aligned} a_n &= \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n}+\frac{1}{n}+(a_{n-i}+1)\frac{n-i}{n}\right] = \\ &= 1+\frac{1}{n^2}\sum_{i=1}^{n}\left[(i-1)a_i+(n-i)a_{n-i}\right] = \qquad (6.3) \\ &= 1+\frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = \\ &= 1+\frac{2}{n^2}\sum_{i=1}^{n-1}ia_i \end{aligned}$$

Rovnice 6.3 představuje rekurentní vztah pro $a_n = f_1(a_1, a_2, \ldots a_{n-1})$. Z něj můžeme odvodit jednodušší rekurentní vztah $a_n = f_2(a_{n-1})$ následujícím způsobem.

Ze vztahu 6.3 přímo vyplývá:

$$\begin{aligned} a_n &= 1+\frac{2}{n^2}\sum_{i=1}^{n-1}ia_i = \\ &= 1+\frac{2}{n^2}(n-1)a_{n-1}+\frac{2}{n^2}\sum_{i=1}^{n-2}ia_i \qquad (6.4) \\ a_{n-1} &= 1+\frac{2}{(n-1)^2}\sum_{i=1}^{n-2}ia_i \qquad (6.5) \end{aligned}$$

Vynásobením vztahu 6.5 výrazem $((n-1)/2)^2$ dostáváme

$$\frac{2}{n^2}\sum_{i=1}^{n-2}ia_i = \frac{(n-1)^2}{n^2}(a_{n-1}-1) \qquad (6.6)$$

Jestliže dosadíme do rovnice 6.4 vztah 6.6, dostaneme

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1}+2n-1) \qquad (6.7)$$

Ukazuje se, že průměrnou délku cesty a_n lze vyjádřit v nerekurzívním tvaru pomocí harmonických čísel (viz 2.1)

$$a_n = 2\frac{n+1}{n}+H_n-3$$

Použitím Eulerovy formule a Eulerovy konstanty dostáváme pro velké n vztah

$$a_n \cong 2[\ln(n)+\gamma]-3 = 2\ln(n)-c$$

Protože průměrná délka cesty v dokonale vyváženém stromu je přibližně

$$a'_n = \log(n)-1 \qquad (6.8)$$

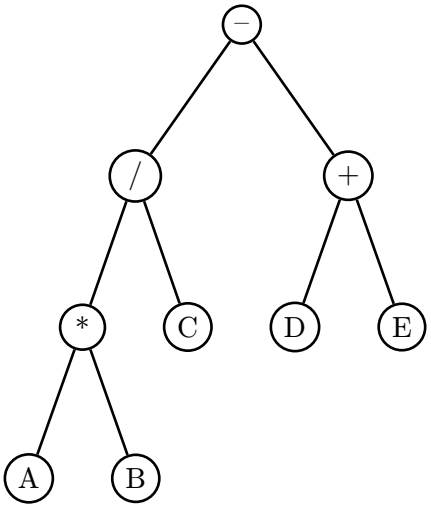
dostáváme vztah

$$\lim_{n\to\infty} \frac{a_n}{a'_n} = \frac{2\ln n}{\log n} = 2\ln 2 = 1,386 \tag{6.9}$$

Významné na vztahu[2] 6.9 je, že budeme-li se snažit za každou cenu zkonstruovat dokonale vyvážený strom, místo „náhodného“, můžeme za předpokladu shodné pravděpodobnosti vyhledání všech klíčů, očekávat průměrné zlepšení délky vyhledávání nejvíce o 39 %. Zdůrazněme slovo průměrný, protože zlepšení může být v konkrétních případech daleko větší.

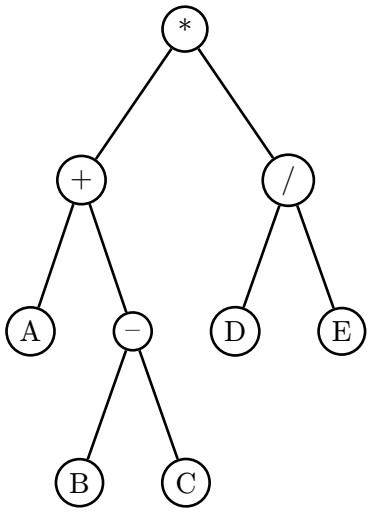
Cvičení

1. Aritmetický výraz je reprezentován výrazovým stromem. Tento výraz zapište v prefixové a v postfixové notaci.



2. Aritmetický výraz je reprezentován výrazovým stromem. Tento výraz zapište v prefixové a v postfixové notaci.

[2]Zanedbali jsme konstantní výrazy, které jsou pro velké n bezvýznamné.



6.5 Dokonale vyvážené stromy

V kapitole o binárních stromech jsme se dozvěděli, že časové složitosti operací nad binárním stromem závisí na jeho výšce. Bude tedy naší snahou tuto výšku pro daný počet uzlů uzlů ve stromu minimalizovat, to znamená budeme se snažit sestrojit takový strom, který je nějakým způsobem **vyvážený**. Vyvážený strom lze intuitivně chápat jako strom jehož pravý podstrom je přibližně stejně velký jako levý podstrom, čili mají zhruba stejnou výšku.

Jestliže chceme vytvořit z n uzlů strom, který má minimální výšku, bude nutno abychom na každou úroveň, vyjma poslední, umístili co největší počet uzlů. Toho se dá lehce dosáhnout tím, že jednotlivé uzly umístíme rovnoměrně na pravou a levou stranu právě vytvářeného uzlu.

Pravidlo rovnoměrné distribuce známého počtu n uzlů se dá nejlépe formulovat rekurzivně takto:

1. Zvolíme jeden uzel za kořen stromu.
2. Vytvoříme levý podstrom s počtem uzlů $n_l = n \operatorname{div} 2$.
3. Vytvoříme pravý podstrom s počtem uzlů $n_r = n - n_l - 1$.

Aplikací tohoto pravidla na posloupnost prvků dostaneme tzv. dokonale vyvážený binární strom.

Definice 6.4 *Strom se nazývá **dokonale vyvážený**, jestliže pro každý uzel stromu platí, že počet uzlů v jeho levém a pravém podstromu se liší nejvýše o jeden.*

Tato definice nám zaručuje, že délka všech cest z kořene do listů se bude lišit nejvíce o jeden uzel. Dokonalá vyváženost stromu má však obrovskou

nevýhodu v tom, že každé přidání uzlu nebo jeho smazání naruší vyváženost stromu, který je nutno zkonstruovat znovu. Je jasné, že pro dynamicky se měnící množiny klíčů je tato struktura nevhodná, vzhledem ke své obrovské programové režii. Naším dalším cílem tedy bude nalézt taková kritéria, která částečně sleví z přísného požadavku dokonalé vyváženosti, aby za tuto cenu získala metody pro pružnější obnovení vyváženosti.

6.6 AVL stromy

Z předcházející diskuse je jasné, že procedura vkládání prvků do stromu, která vždy způsobuje restrukturalizaci stromu za účelem dokonalého vyvážení se těžko může stát efektivnější, protože obnova dokonalého vyvážení po náhodném přidání je dost složitá operace. Možné zlepšení spočívají ve formulování méně přísných definicí vyváženosti. Tato nedokonalá kritéria vyváženosti by měla vést k jednodušším procedurám stromové reorganizace za cenu pouze minimálního zhoršení průměrné výkonnosti vyhledávání.

Jednu z definic vyváženosti zformulovali **Adelson-Velskii** a **Landis** [1, 23, 2]. Kritérium vyváženosti zní takto:

Definice 6.5 ***Strom je vyvážený*** *tehdy a jen tehdy, je-li rozdíl výšek každého uzlu nejvýše 1.*

Stromy, které splňují toto kritérium, se často nazývají **AVL–stromy**. Definice je nejen jednoduchá, ale vede i k proceduře znovuvyvážení a k průměrné délce cesty vyhledávání, která je prakticky identická s délkou cesty dokonale vyváženého stromu.

Na AVL-stromech je možno v čase $O(\log n)$ vykonávat následující operace:

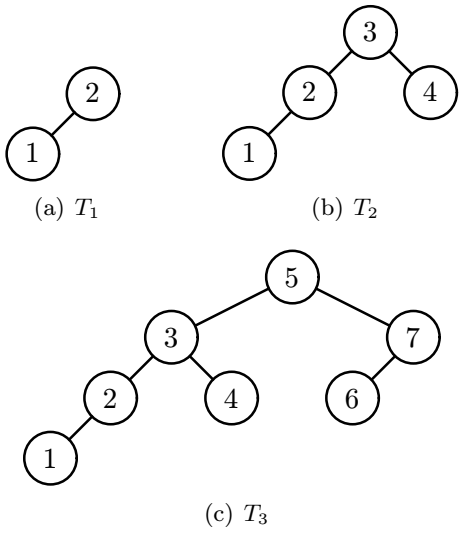
1. Vyhledání uzlu s daným klíčem
2. Vložení uzlu s daným klíčem
3. Zrušení uzlu s daným klíčem

Tato tvrzení jsou přímé důsledky věty, kterou dokázali Adelson-Velskii a Landis, a která zaručuje, že AVL-strom bude maximálně o 45% (nikdy ne víc) vyšší než jeho dokonale vyvážený „dvojník“ bez ohledu na počet uzlů. Jestliže symbolem $h_b(n)$ označíme výšku AVL-stromu s n uzly, potom

$$\log(n+1) \leq h_b(n) \leq 1,4404 \log(n+2) - 0,328$$

Optimální hodnotu získáme, pochopitelně, v případě dokonale vyváženého stromu, kde $n = 2^k - 1$. Jak ale vypadá struktura nejhoršího AVL-stromu?

Abychom našli maximální výšku h pro všechny vyvážené stromy s n uzly, uvažujme o pevné výšce h a pokusme se sestrojit vyvážený strom s minimálním počtem uzlů. Označme tento strom s výškou h symbolem T_h. Přirozeně



(a) T_1 (b) T_2

(c) T_3

Obrázek 6.6: Fibonacciho stromy výšky 2, 3 a 4

T_0 je prázdný strom, T_1 je strom s jediným uzlem. Abychom mohli sestrojit strom T_h pro $h > 1$, připojíme ke kořeni dva podstromy opět s minimálním počtem uzlů. Je jasné, že jeden podstrom musí mít výšku $h-1$, přičemž ten druhý ji může mít o jednu menší, tj. $h-2$. Na obrázku 6.6 jsou znázorněny stromy s výškou 2, 3 a 4.

Protože princip kompozice těchto stromů se velmi podobá principu definice Fibonacciho čísel, nazýváme je *Fibonacciho stromy*.

Definice 6.6 *1. Prázdný strom je Fibonacciho strom s výškou 0.*

2. Jediný uzel je Fibonacciho strom s výškou 1.

3. Jestliže T_{h-1} a T_{h-2} jsou Fibonacciho stromy s výškou $h-1$ a $h-2$, potom $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ je Fibonacciho strom s výškou h.

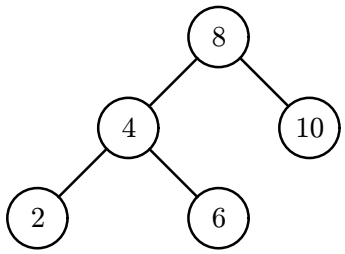
4. Žádné jiné stromy nejsou Fibonacciho stromy.

Počet uzlů stromu T_h můžeme definovat pomocí jednoduchého rekurentního vztahu:

$$\begin{aligned} N_0 &= 0 \\ N_1 &= 1 \\ N_h &= N_{h-1} + 1 + N_{h-2} \end{aligned}$$

6.6.1 Vkládání do AVL-stromů

Uvažujme nyní, co se může stát, jestliže přidáme uzel do AVL-stromu. Pro strom s kořenem r a dvěma podstromy levým L a pravým R mohou nastat



Obrázek 6.7: Vyvážený strom

tři případy. Předpokládejme dále, že se nový uzel přidá do levého podstromu L a tím způsobí zvýšení jeho výšky o 1.

1. $h_L = h_R$: L a R budou mít rozdílné výšky, ale kritérium vyváženosti zůstává neporušené.

2. $h_L < h_R$: L a R budou mít stejnou výšku tj. vyváženost se dokonce ještě zlepší.

3. $h_L > h_R$: kritérium vyváženosti se poruší, strom bude potřeba znovu vyvážit.

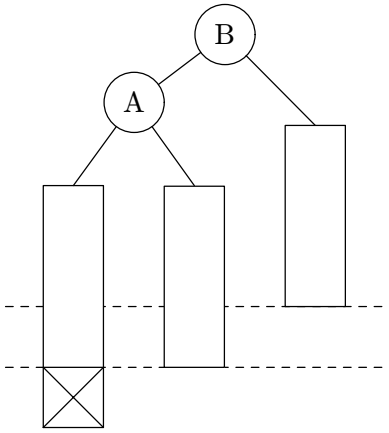
Uvažujme nyní strom na obrázku 6.7. Uzly s klíči 9 a 11 můžeme přidat bez nutnosti znovuvyvážení stromu; strom s kořenem 10 se stane jednostranným (případ 1); ve stromu s kořenem 8 dojde ke zlepšení vyváženosti (případ 2). Vložení uzlů s klíči 1, 3, 5 a 7 znamená nutnost vyvážení stromu.

Důkladným studiem situace zjistíme, že existují v podstatě dva různé případy vyžadující individuální řešení. Třetí případ jsme schopni odvodit na základě symetrie ze dvou předcházejících případů. Případ 1 charakterizuje přidání uzlů s klíči 1 a 3, případ 2 přidání uzlů 5 nebo 7 do stromu na obrázku 6.7.

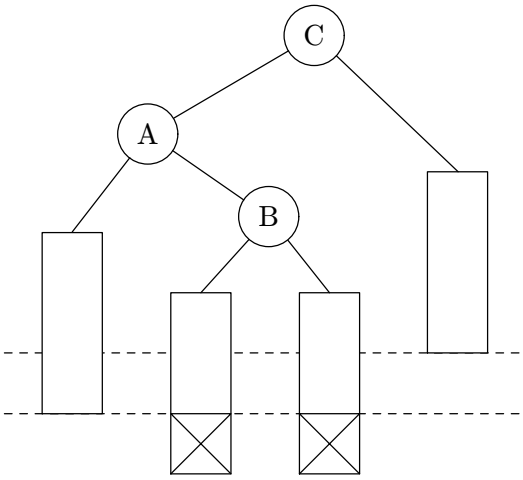
Výše zmiňované dva případy jsou schematicky znázorněny obrázkem 6.8. Obdélníky jsou označeny jednotlivé podstromy a křížkem výška, o kterou se příslušný podstrom vložením nového uzlu zvětšil. Vyváženost obou struktur obnovíme provedením jednoduchých transformací.

Výsledek transformací je na obrázku 6.9. Je dobré si uvědomit, že jediné možné pohyby uzlů jsou vertikálním směrem, relativní horizontální pozice musí zůstat nezměněny, protože tyto jsou určeny uspořádáním klíčů ve uzlech.

Algoritmus vkládání a rušení uzlů ve stromu závisí především na způsobu, kterým budeme uchovávat informace o vyváženosti ve stromové struktuře (této informaci se říká **vyvažovací faktor**). Extrémní řešení by spočívalo v implicitním uchovávání ve struktuře stromu. V takovém případě by se po každém přidání uzlu musel zjišťovat příslušný vyvažovací faktor uzlu,

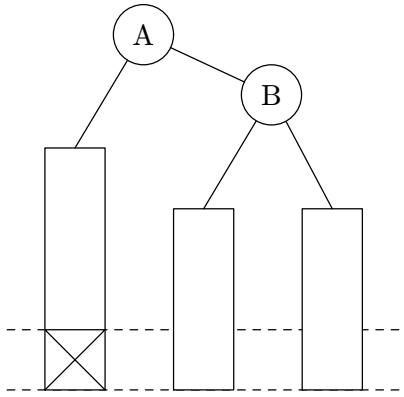


(a) Případ 1

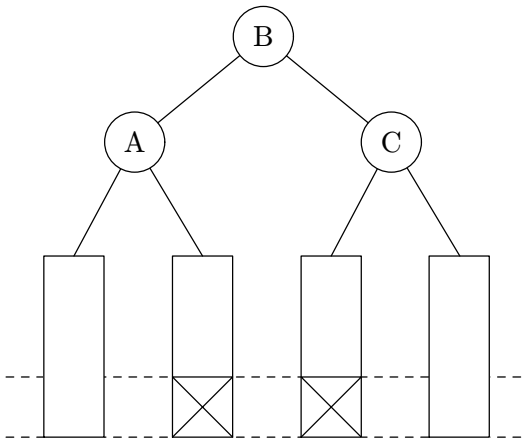


(b) Případ 2

Obrázek 6.8: Nevyváženost způsobená přidáním nového uzlu



(a) Případ 1



(b) Případ 2

Obrázek 6.9: Obnovení vyváženosti

což není nejjednodušší operace. Opačným extrémem je uchovávání informace o vyváženosti v každém uzlu. Deklarace uzlu ve stromu by potom vypadala následovně:

```
struct CNode
{
  TKey   key;      // klíč uzlu
  TData  data;     // data ve uzlu
  CNode* left;     // levý potomek
  CNode* right;    // pravý potomek
  int    bal;      // vyvažovací faktor −1, 0, +1
};  // CNode;
```

Vyvažovací faktor daného uzlu budeme interpretovat jako rozdíl výšky jeho pravého podstromu a výšky jeho levého podstromu.

Proces přidávání uzlu do stromu se v podstatě skládá ze tří bodů:

1. prohledání stromu abychom zjistili, zda se ve stromu daný uzel již nenachází,
2. přidání nového uzlu a určení výsledného vyvažovacího faktoru,
3. zkontrolování vyvažovacího faktoru, každého uzlu na cestě opačným směrem tj. na cestě od uzlu ke kořeni.

Přestože tato metoda způsobuje některé nadbytečné kontroly (jestliže jednou dosáhneme vyváženosti uzlu, není nutné ji znovu ověřovat u jeho předchůdců), budeme se této metody vkládání držet, protože je evidentně správná a je ji možno implementovat jednoduchým rozšířením vkládání do binárního stromu. Prezentovaná metoda popisuje algoritmus vkládání pro jeden uzel a vzhledem k jejímu rekurzívnímu charakteru je ji možné jednoduše přizpůsobit tak, aby obsahovala doplňkovou operaci „na cestě opačným směrem“. V každém kroku je potřeba vrátit informaci o tom, zda se výška podstromu(ve kterém se vkládání uskutečnilo) zvětšila nebo ne. Tuto funkci plní boolovský parametr h, který indikuje zvětšení resp. nezvětšení výšky podstromu. Pochopitelně je nutno parametr h předávat odkazem, protože jeho prostřednictvím se vrací výsledek.

Předpokládejme, že proces vkládání uzlu se vrátil do uzlu p z jeho levé větve (viz obr. 6.8) s informací, že výška podstromu se zvětšila. Nyní musíme rozlišit tři situace, v závislosti na výšce podstromů před přidáním nového uzlu:

1. $h_L < h_R$, `p->bal = +1`, předešlá nevyváženost ve uzlu p se vyrovnala;
2. $h_L = h_R$, `p->bal = 0`, váha se nyní nakloní doleva;
3. $h_L > h_R$, `p->bal = -1`, je nutné znovuvyvážení.

Ve třetím případě zjistíme, na základě prozkoumání vyvažovacího faktoru kořene levého podstromu, jestli jde o případ 1 nebo 2 z obrázku 6.8. Je-li výška levého podstromu větší než pravého podstromu, jde o případ 1, jinak jde o případ 2. Lze se přesvědčit, že se za této situace nemůže vyskytnout levý podstrom s vyvažovacím faktorem 0 v jeho kořeni.

Operace potřebné pro znovuvyvážení jsou realizovány cyklickými záměnami pointerů. V důsledku cyklických záměn pointerů může dojít k jednoduché (RR-rotace, LL-rotace) nebo dvojité rotaci (LR-rotací, RL-rotaci) dvou nebo tří uzlů. Kromě rotací je nutné současně nastavovat na patřičné hodnoty - vyvažovací faktory rotovaných uzlů.

Princip algoritmu vkládání je znázorněn na obrázku 6.10. Uvažujme binární strom *(a)*, který se skládá ze dvou uzlů. Přidání klíče 7 způsobí nevyváženost stromu (vznikne lineární seznam). Vyvážení se dosáhne provedením **RR-rotace**, čímž dostaneme dokonale vyvážený strom *(b)*. Dalším přidáním uzlů 2 a 1 nastane nevyváženost podstromu s kořenem 4. Na jeho vyvážení potřebujeme použít jednoduchou **LL-rotaci** *(d)*. Následujícím přidáním klíče 3 se naruší kritérium vyváženosti kořene 5. Znovuvyvážení dosáhneme složitější **LR-rotací**; výsledkem je strom *(e)*. Přidání uzlu 6 způsobí čtvrtý případ vyvažování, **RL-rotaci** okolo uzlu 5. Výsledkem je strom *(f)*.

V souvislosti s výkonností algoritmu vkládání do AVL-stromu nás zajímají především dvě otázky:

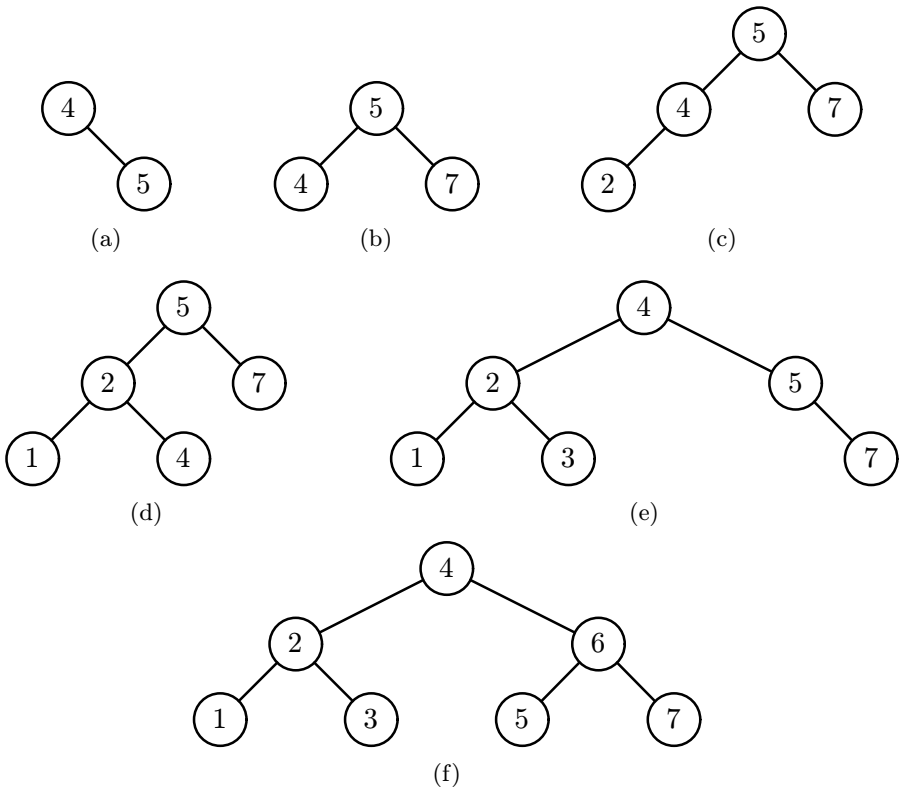
1. Jaká bude očekávaná výška AVL-stromu, jestliže se všech $n!$ permutací n klíčů vyskytuje se stejnou pravděpodobností?

2. Jaká je pravděpodobnost, že přidání uzlu způsobí znovuvyvažování?

Matematická analýza tohoto problému patří mezi otevřené otázky. Empirické testy potvrzují domněnku, že očekávaná výška AVL-stromu je $h = \log n + c$, kde c je konstanta s malou hodnotou ($c \cong 0,25$). To znamená, že AVL-stromy vykazují podobné chování jako dokonale vyvážené stromy, přičemž je s nimi jednodušší manipulace. Empirické testy ukazují, že v průměru je potřeba jedno vyvažování na přibližně každé dvě přidání nových uzlů. Jednoduché a dvojité rotace jsou stejně pravděpodobné. Příklad na obrázku 6.10 byl pečlivě zkonstruován tak, aby se vyskytly všechny možné rotace při minimálním počtu přidání do stromu.

6.6.2 Rušení uzlů v AVL-stromech

Základním schématem realizace procedury na rušení uzlu v AVL-stromu je procedura rušení uzlu v binárním vyhledávacím stromu. Jednoduché jsou opět případy listů a uzlů, které mají jediného potomka. Pokud má uzel, který chceme zrušit, dva podstromy nahradíme jej opět nejpravějším z levého podstromu. Podobně jako v případě vkládání, zavedeme boolovský parametr h označující zmenšení výšky. Pouze v případě, že h má hodnotu *true*, budeme



Obrázek 6.10: Vkládání do AVL-stromu

uvažovat o znovuvyvážení. Parametr h nabude hodnoty *true* jestliže najděme a zrušíme příslušný uzel nebo v případě, kdy samotné znovuvyvážení způsobí zmenšení výšky podstromu. V algoritmu (viz B.2) zavádíme dvě (symetrické) vyvažovací operace ve formě metod třídy (`Balance1` a `Balance2`), protože jsou volány z více míst algoritmu rušení uzlů. Poznamenejme, že metoda `Balance1` se volá v případě zmenšení výšky levého podstromu, metoda `Balance2` v opačném případě.

Operaci rušení uzlů v AVL-stromu znázorňuje obrázek 6.11. Z původního stromu *(a)* se postupně odebírají uzly s klíči $4, 8, 6, 5, 2, 1$ a 7; výsledkem jsou stromy *(b)*, ...,*(h)*.

Zrušení uzlu s klíčem 4 je jednoduché, protože tento uzel je listem. Zrušením tohoto uzlu se však poruší vyváženost uzlu 3. Operace znovuvyvážení vyžaduje jednoduchou LL-rotaci.

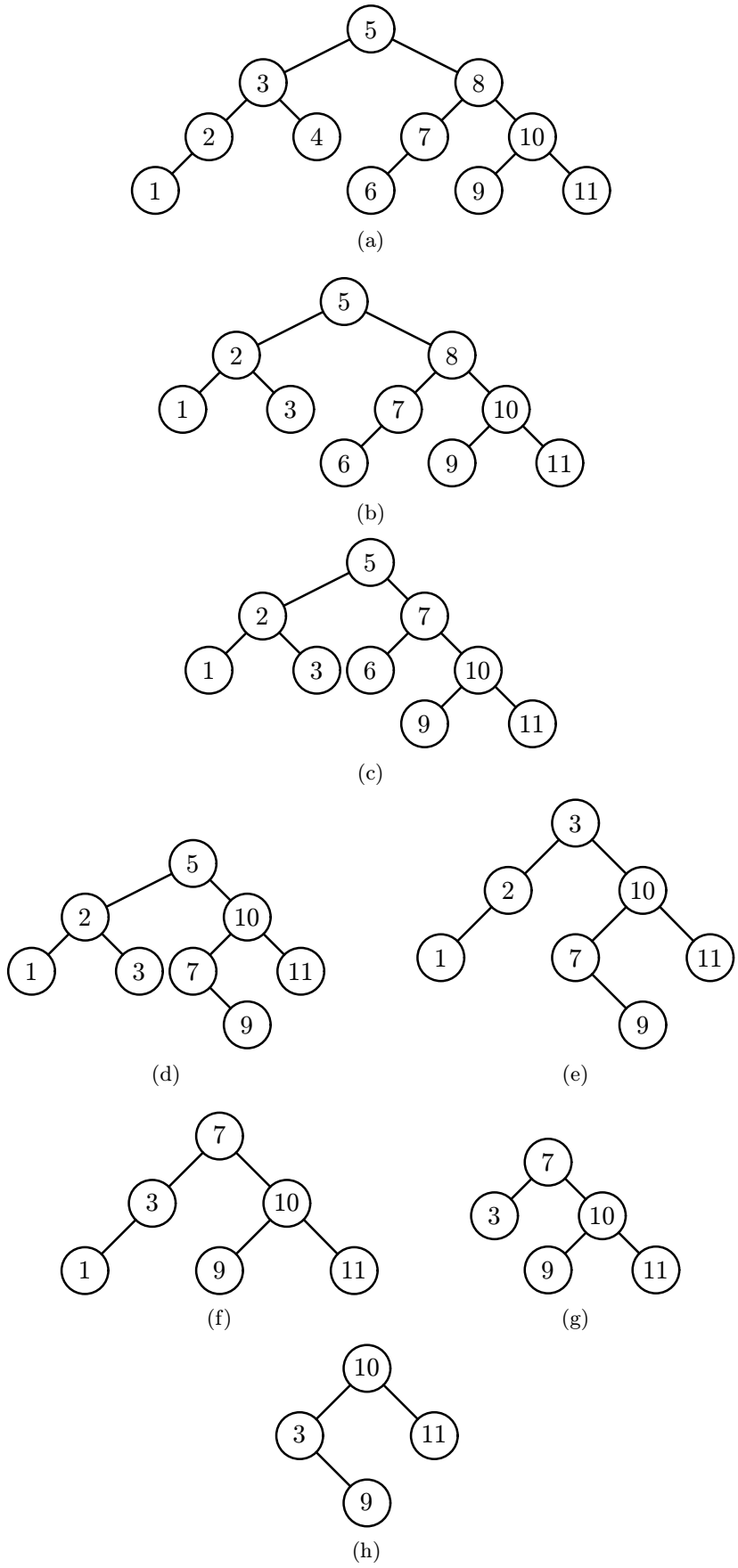
Znovuvyvážení bude opět zapotřebí při zrušení uzlu 6. V tomto případě je nutno vyvážit pravý podstrom kořene 7 a to pomocí jednoduché RR-rotace. Zrušení uzlu 2 je sice opět přímočaré, protože tento uzel má jen jediného potomka, způsobí však složitou dvojitou RL-rotaci. Předtím než zrušíme uzel 7, je třeba nahradit jej nejpravějším uzlem jeho levého podstromu, tj. uzlem s klíčem 3. Následující dvojitá LR-rotace způsobí znovuvyvážení stromu a jeho závěrečnou podobu *(h)*.

Je jasné, že zrušení prvku v AVL-stromu je možné vykonat – v nejhorším případě – prostřednictvím $O(\log n)$ operací. Nepřehlédněme však podstatný rozdíl mezi chováním algoritmu přidávání a rušení uzlů. Zatímco přidání uzlu může způsobit nejvýše jednu rotaci (dvou nebo tří uzlů), rušení může vyžadovat rotaci všech uzlů absolvované cesty. Uvažujme například o zrušení nejpravějšího uzlu Fibonacciho stromu. V tomto případě zrušení libovolného uzlu způsobí zmenšení výšky stromu; navíc zrušení nejpravějšího uzlu vyžaduje maximální počet rotací. Tento případ představuje nejhorší výběr uzlu v nejhorším případě vyváženosti stromu.

S jakou pravděpodobností se ale vyskytují rotace v průměrném případě? Výsledky empirických testů ukazují, že zatímco pro přibližně každé druhé přidání uzlu je potřeba jedna rotace, až každé páté zrušení vyvolá rotaci. Proto lze považovat rušení uzlů v AVL-stromech za stejně složité jako přidávání.

6.7 2-3-4 stromy

Abychom eliminovali nejhorší stavy při prohledávání binárních stromů potřebujeme vytvořit u námi používaných datových struktur jakousi flexibilitu. Zatím jsme se zabývali, uvolňováním striktnosti kritéria dokonalé vyváženosti. Nyní se zaměříme na uzly binárního stromu. Dejme tomu, že uzly stromu mohou obsahovat více než jeden klíč. Přesněji řečeno vytvoříme **2-3-4 strom** se třemi novými typy uzlů **3-uzel** a **4-uzel**, které mají tři resp.



Obrázek 6.11: Rušení uzlů ve vyváženém stromu

Obrázek 6.12: 2-3-4 strom

Obrázek 6.13: Vložení do 2-3-4 stromu

čtyři ukazatele ukazující na potomky. 3-uzel resp. 4-uzel obsahuje dva resp. tři klíče. První ukazatel v 3-uzlu ukazuje na uzel s klíči menšími než oba klíče aktuálního uzlu, druhý ukazuje na uzel s hodnotami klíčů mezi oběma klíči aktuálního uzlu a třetí na uzel s klíči vyššími. Obdobná situace nastává u 4-uzlu. (Uzly ve standardním vyhledávacím binárním stromu pak můžeme nazývat **2-uzly**; jeden klíč, dva ukazatele). Později si ukážeme jak definovat a implementovat některé základní operace nad těmito rozšířenými uzly; nyní si je třeba pouze uvědomit, že s nimi můžeme normálně pracovat a ukážeme si jak je skládat dohromady do stromů.

Například obrázek 6.12 ukazuje 2-3-4 strom obsahující položky A S E A R C H I a N. Na první pohled je jasné, jak bychom v takovémto stromu hledali. Například při hledání písmene G bychom z kořene sledovali prostřední ukazatel, protože G je mezi E a R. Pak by vyhledávání neúspěšně skončilo na nejlevějším ukazateli uzlu obsahující H I a N.

Při připojení nového uzlu do 2-3-4 stromu by bylo nejjednodušší provést neúspěšné hledání a poté navázat nový uzel na posledně vyhledaný uzel. Velmi jednoduché je to, pokud dojdeme nakonec do 2-uzlu. Pouze ho zaměníme za 3-uzel. Například X bychom do stromu na obr. 6.12 vložili (a připojili další ukazatel) do uzlu obsahujícího S. Obdobně 3-uzel zaměníme za 4-uzel. Ale co uděláme, když je potřeba vložit nový prvek do 4-uzlu? Například bychom chtěli vložit písmeno G. Jedna možnost je navázat nový uzel na již existující uzel obsahující H I a N, ale lepší řešení je na obrázku 6.13: nejprve rozdělíme 4-uzel na dva 2-uzly a přesuneme jeden z klíčů do rodičovského uzlu. Nejdříve tedy rozdělíme H I a J 4-uzel na dva 2-uzly (jeden bude obsahovat H a druhý N) a „prostřední klíč“ I přesuneme do 3-uzlu obsahujícího E a R, čímž z něho vytvoříme 4-uzel. Tím se pro klíč G vytvoří místo ve 2-uzlu obsahujícím H.

Co když ale potřebujeme rozdělit 4-uzel, jehož rodičem je také 4-uzel? Jednou z možností by bylo rozdělit i rodiče, ale i prarodič může být 4-uzel a i jeho rodič atd. : nakonec bychom mohli rozdělovat všechny uzly až po kořen. Jednodušší cestou je zajistit, že žádný rodič jakéhokoliv uzlu není 4-uzel tím, že cestou „dolů“ rozdělíme všechny 4-uzly, na které narazíme. Takto lze jednoduše do 2-3-4 stromu přidávat nové uzly. Jak je ukázáno na obr. 6.14 pokaždé, když narazíme na 2-uzel na nějž je napojený 4-uzel měli bychom jej transformovat na 3-uzel na nějž jsou napojeny dva 2-uzly. Stejně tak, když narazíme na 3-uzel k němuž je připojený 4-uzel změníme jej na 4-uzel uzel na nějž jsou napojeny dva 2-uzly. Procházíme-li strom shora dolů máme jistotu, že uzel, který opouštíme není 4-uzel.

Obrázek 6.14: Dělení 4-uzlů

Kdykoliv se stane, že by kořen byl 4-uzlem rozdělíme ho do tří 2-uzlů stejně tak, jak jsme to udělali v předchozích případech. Rozdělení kořene je jedinou operací která způsobí nárůst výšky stromu o jednu.

Dělení uzlů je založeno na přesouvání klíčů a ukazatelů. Dva 2-uzly mají stejný počet napojení jako jeden 4-uzel, takže rozdělení můžeme provést bez jakéhokoliv dalšího zásahu. A 3-uzel může být na 4-uzel změněn pouhým přidáním jednoho klíče (v tomto případě vznikne jeden nový ukazatel). Hlavní význam této transformace je, že je vše čistě lokální: nemusíme modifikovat žádnou část stromu, vyjma případů na obrázku 6.14. Každá z těchto transformací přemísťuje jednu položku ze 4-uzlu jeho rodiči a v závislosti na tom reorganizuje ukazatele k potomkům.

Takto navržený algoritmus ukazuje jak ve 2-3-4 stromu vyhledávat a jak do něj vkládat. Jednoduše je třeba dělit 4-uzly na menší při cestě shora dolů. Proto se anglicky těmto stromům také říká *top-down 2-3-4 stromy*. Zajímavé je, že ačkoliv jsme se vůbec nestarali o vyvažování stromu, je vždy dokonale vyvážený!

Věta 6.2 *Předpokládejme, že je dán 2-3-4 strom s n uzly. Vyhledávací algoritmus navštíví nejvýše* $\log n + 1$ *uzlů.*

Důkaz. Vzdálenost od kořene ke všem listům je stejná: transformace, jak jsme si ukázali, nemají na vzdálenost uzlů od kořene žádný vliv. Pouze pokud dělíme kořen mění se výška stromu a v tom případě se hloubka všech uzlů zvýší o jedna. Pokud jsou všechny uzly 2-uzly je výška stromu stejná jako u úplného binárního stromu, pokud jsou přítomny i 3-uzly a 4-uzly může být výška vždy jenom menší. ∎

Věta 6.3 *Nechť je dán 2-3-4 strom s n uzly. Vkládací algoritmus potřebuje méně než* $\log n + 1$ *rozdělení uzlů a předpokládá se, že využívá průměrně méně než jedno rozdělení.*

Důkaz. Nejhorším případem je dělení všech uzlů které po cestě dolů navštívíme. U stromu postaveného z náhodné permutace n prvků je tato situace krajně nepravděpodobná, čímž lze ušetřit mnoho dělení, protože ve stromu není mnoho 4-uzlů a drtivá většina z nich jsou listy. Výsledky analýzy průměrného chování 2-3-4 stromu mohou odrazovat, ale empirickým měřením bylo zjištěno, že se dělí jen velmi málo uzlů. ∎

Strom, který jsme na předchozích stránkách definovali je vhodný pro definici vyhledávacího algoritmu se zaručenou nejhorší variantou. Jsme ale na půli cesty k vlastní implementaci. Je sice možné napsat algoritmy na změny mezi různými typy uzlů, ale manipulace se složitějšími datovými strukturami

Obrázek 6.15: Červeno-černá reprezentace 3-uzlů a 4-uzlů

způsobí, že vyhledávací algoritmus bude pomalejší než standardní vyhledávací algoritmus na binárním stromu. Základní požadavek na vyvažování stromů je strom „pojistit“ proti nejhoršímu případu. Bylo by však plýtvání platit příliš vysokou cenu za násilné udržení takového stavu při každém průchodu stromem. Naštěstí existuje relativně jednoduchá reprezentace 2-uzlů, 3-uzlů a 4-uzlů, která umožňuje provádět vzájemné transformace standardní cestou s pouze malou ztrátou oproti standardnímu vyhledávání v binárním stromu.

2-3-4 strom je možno reprezentovat jako standardní binární strom (pouze 2-uzly) použitím speciálního bitu v každém uzlu. Myšlenka spočívá v tom, že 3-uzly a 4-uzly převedeme do malých binárních stromů spojených „červenými“ ukazateli. Tyto pak kontrastují s „černými“ ukazateli spojujícími dohromady vlastní 2-3-4 strom. Obrázek 6.15 ukazuje jak jsou spojeny jednotlivé uzly. (Pro 3-uzly jsou možné obě reprezentace.) Tomu ekvivalentní možností je obarvení jednotlivých uzlů. Uzel z něhož vedou červené hrany se obarví červeně a naopak.

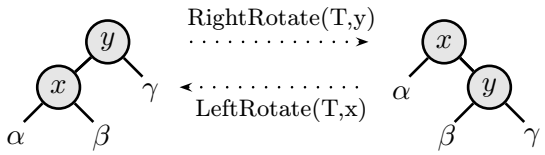
6.8 Red-Black stromy

Red–Black strom [7, 19] je binární strom s jedním dvouhodnotovým příznakem ve uzlu navíc. Tento příznak představuje barvu uzlu, která může být **červená** nebo **černá**. Red–Black strom zajišťuje, že žádná cesta z kořene do libovolného listu stromu nebude dvakrát delší než kterákoli jiná, to znamená, že strom je přibližně vyvážený.

Každý uzel se skládá z položek: *key, color, left, right a p*. Jestliže potomek nebo rodič uzlu neexistují, příslušný ukazatel je nastaven na NULL. Je vhodné uvažovat ukazatele NULL jako listy binárního stromu a „normální“ uzly s klíči jako vnitřní uzly binárního stromu.

Definice 6.7 *Binární vyhledávací strom je **Red–Black strom**, jestliže splňuje následující kritéria:*

1. *Každý uzel je buď černý nebo červený.*
2. *Každý list (NULL) je černý.*
3. *Jestliže je daný uzel červený, pak jeho potomci jsou černí.*
4. *Každá cesta z libovolného uzlu do listu (NULLu) obsahuje stejný počet černých uzlů.*



Obrázek 6.16: Rotace na binárním vyhledávacím stromu

Operace RightRotate(T, x) transformuje sestavu dvou uzlů na levé straně na sestavu uzlů na pravé straně obrázku výměnou konstantního počtu ukazatelů. Sestavu na pravé straně lze převést na sestavu na levé straně inverzní operací LeftRotate(T, y). Za uvedené dva uzly je možno považovat libovolné dva uzly v binárním vyhledávací stromu. Písmena α, β a γ reprezentují příslušné podstromy uzlů x a y. Rotace pochopitelně zachovávají pořadí klíčů ve stromu: klíče$[\alpha] <$ klíč$[x] <$ klíče$[\beta] <$ klíč$[y] <$ klíče$[\gamma]$.

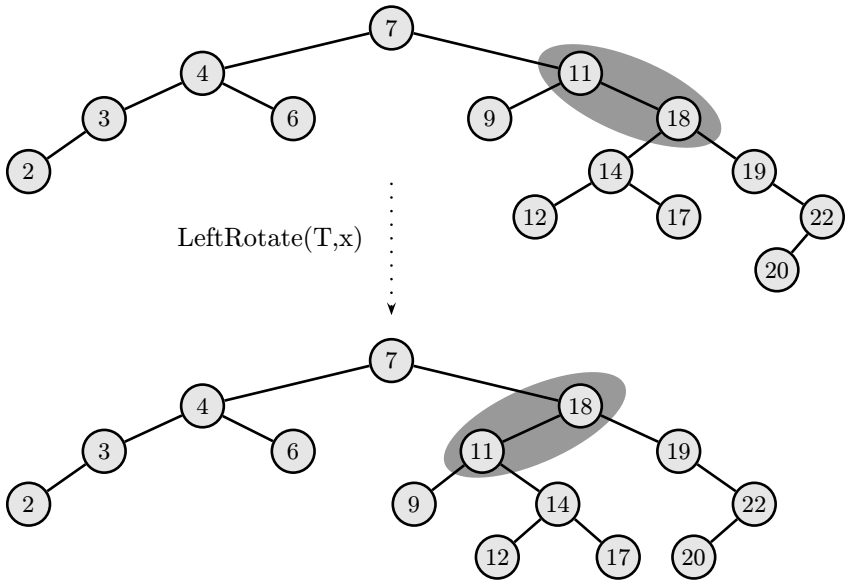
Počet černých uzlů na cestě z uzlu x do listu (mimo uzel x) černou výškou uzlu x, píšeme $bh(x)$. Dále definujeme černou výšku stromu jako černou výšku kořene stromu.

Věta 6.4 *Výška RB stromu s n vnitřními uzly je nejvýše* $2\log(n+1)$.

6.8.1 Rotace

Operace Insert a Delete pracují na Red–Black stromu s n klíči v čase $O(\log n)$. Protože strom modifikují, mohou narušit vlastnosti Red–Black stromu (viz definice na straně 162). Abychom tyto nežádoucí modifikace napravili, musíme změnit barvu některých uzlů a přestavět strukturu ukazatelů. Tuto strukturu měníme pomocí operací zvaných **rotace**, což jsou lokální operace nad stromem, které zachovávají pořadí klíčů. Rozeznáváme dva druhy rotací: levou a pravou (viz obrázek 6.16). Když provádíme pravou rotaci uzlu x, předpokládáme, že pravý potomek y není NULL. Levá rotace se točí okolo ukazatele z x do y. Uzel y se stane novým kořenem stromu, s uzlem x jako svým levým potomkem a levý potomek y se napojí jako pravý potomek x.

```
void CRedBlackTree::LeftRotate(CNode* x)
{
  CNode* y = x->right;
  x->right = y->left;
  if (y->left != m_z)
    y->left->parent = x;
  y->parent = x->parent;
  if (x->parent == m_z)
    m_root = y;
  else
    if (x == x->parent->left)
      x->parent->left = y;
    else
```



Obrázek 6.17: Příklad užití LeftRotate
Obrázek ukazuje jak levá rotace LeftRotate(T,x) modifikuje binární vyhledávací strom. Inorder průchod stromem před a po provedení rotace vytvoří stejný seznam uzlů tj. nejsou narušeny vlastnosti binárního vyhledávacího stromu.

```
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}  // CRedBlackTree::LeftRotate
```

Na obrázku 6.17 je ukázka levé rotace. Kód pravé rotace je obdobný (symetrický). Obě rotace pracují v čase $O(1)$. Mění se pouze ukazatele, ostatní položky uzly zůstávají nezměněny.

6.8.2 Vložení uzlu

Vložení nového uzlu do Red–Black stromu s n uzly lze provést v čase $O(\log n)$. Uzel se vkládá stejným způsobem jako v normálním binárním vyhledávacím stromu. Nově přidaný uzel je obarven červeně. Pro zachování vlastností Red–Black stromu je nutno strom po přidání opravit přebarvením uzlů a provedením rotací.

```
void CRedBlackTree::RBInsert(T new_item)
{
1  x = TreeInsert(new_item);
2  x->color = red;
3  while ((x != m_root) && (x->parent->color == red))
4     if (x->parent == x->parent->parent->left)
      {
```

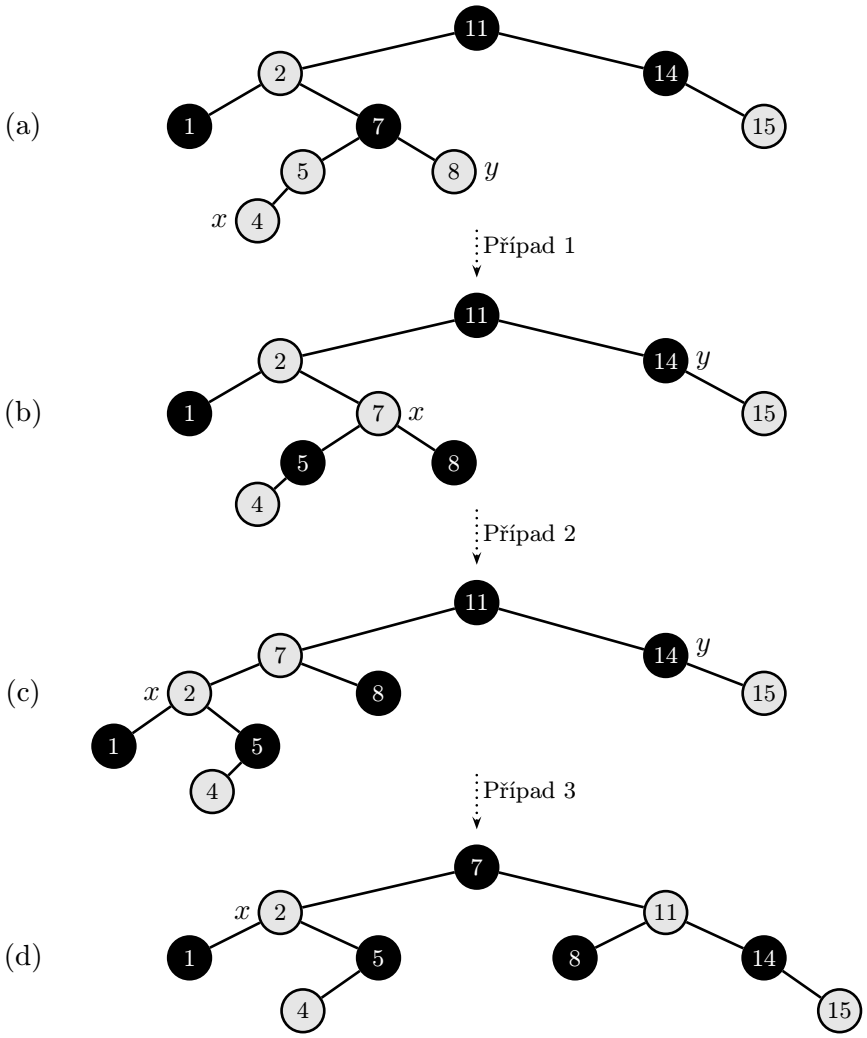
```
5       y = x->parent->parent->right;
6       if (y->color == red)
        {
7         x->parent->color = black            // případ 1
8         y->color = black;                    // případ 1
9         x->parent->parent->color = red; // případ 1
10        x = x->parent->parent;              // případ 1
        }
        else
        {
11        if (x == x->parent->right)
          {
12          x = x->parent;                    // případ 2
13          LeftRotate(x);                     // případ 2
          }
14        x->parent->color = black;           // případ 3
15        x->parent->parent->color = red; // případ 3
16        RightRotate(x->parent->parent);    // případ 3
        } // else
      } // if
17    else // symetrické k větvi if left a right jsou přehozeny
18    m_root->color = black;
}  // CRedBlackTree::RBInsert
```

Kód metody RBInsert je méně imponující než vypadá. Rozdělme zkoumání kódu do tří kroků. Za prvé je nutno určit jak se poruší vlastnosti Red–Black stromu na řádcích 1 a 2, když přidáme uzel a obarvíme jej červeně.. Za druhé musíme prozkoumat celkový záměr (cíl) cyklu `while` na řádcích 3–17. Nakonec prostudujme všechny tři případy uvnitř cyklu `while`.

Které z vlastností Red–Black stromu se mohou porušit na řádcích 1 a 2? Vlastnost 1 je určitě splněna, stejně jako vlastnost 2, protože nově vložený uzel je červený a potomci jsou NULL (černí). Vlastnost 4, která říká, že počet černých uzlů na libovolné cestě z daného uzlu musí být stejný, je splněna, jelikož vložený uzel x nahradil (černý) NULL a uzel x je červený se dvěmi černými potomky NULL. Proto lze porušit jedině vlastnost 3, která vyžaduje, že červený uzel nesmí mít červeného potomka. Speciálně, vlastnost 3 se poruší pokud rodič uzlu x je červený a uzel x je obarven na červeno na řádku 2. Obrázek 6.18 ukazuje jaké situace nastanou při vložení uzlu x.

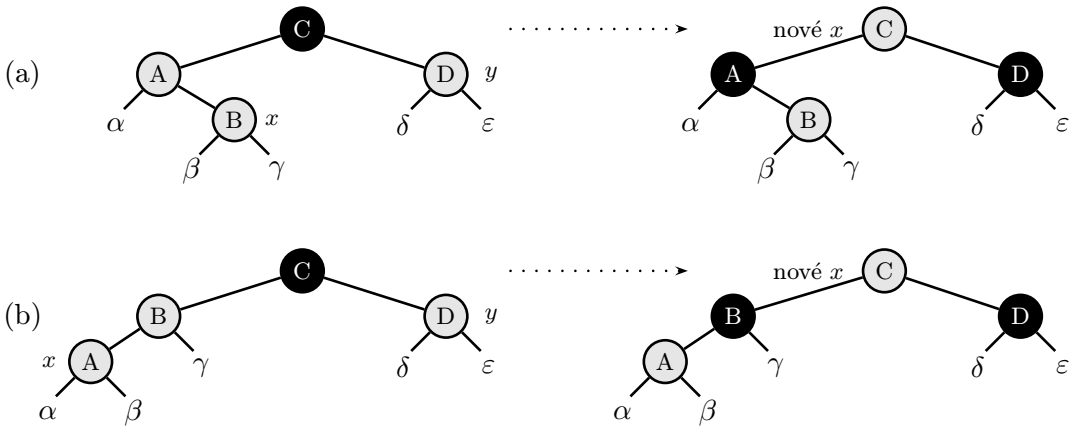
Účelem cyklu `while` na řádcích 3 až 17 je přesunování narušení třetí vlastnosti nahoru po stromu při zachování vlastnosti 4 jako invariantu. Na počátku každého průchodu cyklem, x ukazuje na červený uzel s červeným rodičem – jedinou anomálii ve stromu. Existují pouze dvě možnosti ukončení cyklu: ukazatel x se dostal až na kořen stromu nebo se provedla rotace a vlastnost 3 je splněna.

V cyklu `while` může nastat celkem 6 případů z nichž tři jsou symetrické v závislosti na tom, zda rodič $p[x]$ uzlu x je levým nebo pravým potomkem prarodiče $p[p[x]]$ uzlu x, což se detekuje na řádku 4. Důležitým předpokladem



Obrázek 6.18: Fáze operace RBInsert

(a) Uzel x po vložení. Protože x a jeho rodič $p[x]$ mají červenou barvu poruší se vlastnost 3. Jestliže rodič uzlu x je červený, lze aplikovat případ 1. Uzly jsou přebarveny a ukazatel x se přesune nahoru po stromu; výsledek činnosti je stav **(b)**. Ještě jednou x a jeho rodič jsou červení, ale strýc uzlu x je černý. Jelikož x je pravým potomkem $p[x]$ nastane případ 2. Provede se levá rotace a výsledný strom je zobrazen jako **(c)**. Nyní x je levým potomkem svého rodiče, tudíž se provede kód případu 3. Pravá rotace převede strom do tvaru **(d)**, který splňuje požadavky na Red-Black strom.



Obrázek 6.19: První případ při vkládání do Red-Black stromu

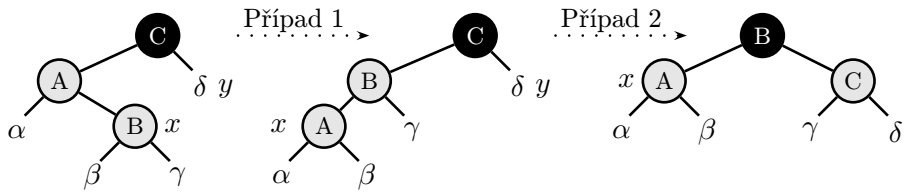
Je porušena vlastnost 3, protože x a jeho rodič $p[x]$ mají oba červenou barvu. Jak v případě **(a)**, kde x je pravým potomkem, tak i v případě **(b)**, kde x je levým potomkem se provede stejná akce. Všechny podstromy $\alpha, \beta, \gamma, \delta$ a ε mají černé kořeny a stejnou černou výšku. Kód případu 1 změní barvu několika uzlů, dodržujíc vlastnost 4. Cyklus `while` pokračuje s prarodičem $p[p[x]]$ jako s novým uzlem x. Jediné možné narušení vlastnosti 3 se může nyní objevit jedině mezi novým uzlem x, který je červený, a jeho rodičem je-li ovšem také červený.

je, že kořen stromu je černý — toto je zaručeno na řádku 18, vždy když se ukončuje cyklus — takže, $p[x]$ není kořen stromu a $p[p[x]]$ existuje.

Případ 1 se liší od případů 2 a 3 barvou „strýce“ uzlu x. Řádek 5 nastavuje ukazatel y na strýce x tj. $right[p[p[x]]]$. Jestliže y je červený provedou se příkazy případu 1, jinak se provede případ 2 nebo 3. Ve všech případech je prarodič uzlu x černý, zatímco rodič $p[x]$ je červený, takže vlastnost 3 je porušena jen mezi x a $p[x]$.

Situace pro případ 1 (řádky 7 až 10)je zobrazena na obrázku 6.19. Případ 1 se provede pokud oba uzly $p[x]$ a y jsou červené. Protože $p[p[x]]$ je černý můžeme obarvit $p[x]$ a y na černo, čímž vyřešíme problém dvou červených barev po sobě a zachováme vlastnost 4. Jediný problém může nastat pokud $p[p[x]]$ má také červeného rodiče. Z toho plyne nutnost opakovat cyklus `while` s $p[p[x]]$ jako novým ukazatelem x.

V případech 2 a 3 má „strýc“ y uzlu x barvu černou. Případy se liší podle toho, je-li x levý nebo pravý potomek $p[x]$. Řádky 12 a 13 představují druhý případ (viz obrázek 6.20), ve kterém je x pravý potomek. Použitím levé rotace lze tento případ převést jednoduše na případ 3. (řádky 14–16), ve kterém je x levým potomkem. Protože x i $p[x]$ mají červenou barvu, rotace neovlivní ani černou výšku bh ani vlastnost 4. Kdykoliv se dostaneme do třetího případu, ať už přímo nebo přes případ druhý, strýc uzlu x je černý, jinak bychom měli provést případ první. Provedeme několik změn barev a pravou rotaci čímž zachováme vlastnost 4 a tím cyklus `while` skončí,



Obrázek 6.20: Druhý a třetí případ při vkládání do Red-Black stromu
Stejně jako v případě 1, vlastnost 3 je narušena v obou případech 2 a 3, protože x a jeho rodič $p[x]$ mají červenou barvu. Všechny podstromy $\alpha, \beta, \gamma, \delta$ a ε mají černé kořeny a stejnou černou výšku. Případ 2 lze transformovat na případ 3 levou rotací, která zachovává vlastnost 4 Red-Black stromu. Případ 3 vyvolá změny několika barev a následnou pravou rotaci, opět s dodržením vlastnosti 4. Cyklus `while` potom končí, protože již nejsou dva červené uzly za sebou (vlastnost 3).

poněvadž již nemáme za sebou v řadě dva červené uzly (rodič $p[x]$ je nyní černý).

Jaká je složitost operace RBInsert? Výška Red–Black stromu s n uzly je úměrná $\log n$, přidání pomocí TreeInsert proběhne v čase $O(\log n)$. Cyklus `while` se opakuje jen pokud se provede případ 1. Tudíž cyklus `while` se provede také v čase $O(\log n)$. Celkem je složitost vložení uzlu do Red–Black stromu $O(\log n)$. Je zajímavé, že se nikdy neprovedou více než dvě rotace (případ 2 a 3).

6.8.3 Rušení uzlu

Stejně jako jiné základní operace nad stromy je možno i rušení uzlu provést v čase $O(\log n)$. Zrušení uzlu je trochu komplikovanější než vložení. V dalším textu předpokládáme reprezentaci ukazatele NULL jako zvláštního uzlu s černou barvou. Tento speciální uzel stromu T budeme označovat $NULL[T]$. Takto můžeme NULL považovat za potomka uzlu x a x za rodiče uzlu NULL. Možnost přidat pro každý ukazatel NULL samostatný uzel je nepraktická kvůli značnému ztrátovému prostoru. Místo toho se používá jediný speciální uzel pro celý strom.

Metoda RBDelete je odvozena od procedury na rušení uzlů v binárním vyhledávacím stromu. Po zrušení uzlu je potřeba zavolat pomocnou metodu RBDeletFixUp, která změní některé barvy a provede rotace nutné pro zachování vlastností Red–Black stromu.

```
void CRedBlackTree::RBDelete(CNode* z)
{
1   if (z->left == m_z || z->right == m_z)
2     y = z;
    else
3     y = TreeSuccessor(z);
4   if (y->left != m_z)
```

```
5       x = y->left;
      else
6       x = y->right;
7     x->parent = y->parent;
8     if (y->parent == m_z)
9       m_root = x;
      else
10      if (y == y->parent->left)
11        y->parent->left = x;
        else
12        y->parent->right = x;
13    if (y != z)
      {
14      z->key = y->key;
15      // kopie dalších složek záznamu
      }; // if
16    if (y->color == black)
17      RBDeleteFixUp(x);
}  // CRedBlackTree::RBDelete(CNode* z)
```

Všimněme si, že přiřazení na řádku 7 se provede bez testu na NULL. Jestliže x je $NULL[T]$, jeho ukazatel *parent* ukazuje na rušený uzel y. Metoda RBDeleteFixUp se volá jen pokud je y černý, v opačném případě se černá výška stromu nemění a vlastnosti Red–Black stromu zůstávají zachovány. Uzel x předaný do metody RBDeleteFixUp je potomek uzlu y, než byl uzel y ze stromu vyjmut nebo je to $NULL[T]$, jestliže y neměl žádného potomka. Přiřazení na řádku 7 garantuje, že rodič x je nyní uzel, který byl předtím rodičem uzlu y, pokud x je normální uzel nebo je to $NULL[T]$.

Nyní budeme zkoumat jak metoda RBDeleteFixUp obnovuje vlastnosti Red–Black stromu.

```
void CRedBlackTree::RBDeleteFixUp(CNode* x)
{
1   while (x != m_root && x->color == black)
2     if (x == x->parent->left)
      {
3       w = x->parent->right;
4       if (w->color == red)
        {
5         w->color = black;            // případ 1
6         x->parent->color = red;  // případ 1
7         LeftRotate(x->parent);      // případ 1
8         w = x->parent->right;    // případ 1
        }; // if
9       if (w->left->color == black && w->right->color == black)
        {
10        w->color = red;              // případ 2
11        x = x->parent;               // případ 2
        } // if
        else
        {
12        if (w->right->color == black)
          {
```

```
13          w->left->color = black;      // případ 3
14          w->color = red;               // případ 3
15          RightRotate(w);               // případ 3
16          w = x->parent->right;         // případ 3
          }; // if
17        w->color = x->parent->color; // případ 4
18        x->parent->color = black;    // případ 4
19        w->right->color = black;     // případ 4
20        LeftRotate(x->parent);        // případ 4
21        x = m_root;                   // případ 4
        }; // else
      } // if
22    else // symetrické ke větvi if
} // CRedBlackTree::RBDeleteFixUp
```

Jestliže rušený uzel y je černý, jeho vyjmutí ze stromu způsobí, že některá cesta ve stromu, která předtím obsahovala uzel y má o jeden černý uzel méně tím ovšem předchůdce uzly y narušuje vlastnost 4. Tento problém je možno odstranit, představíme-li si, že x má jednu černou barvu „navíc“. Tím přidáme 1 k počtu černých uzlů na cestách přes x a vlastnost 4 je splněna. Takže, když rušíme černý uzel „přehodíme“ jeho černou barvu na jeho potomka. Jediným problémem zůstává, že takhle narušíme vlastnost 1, protože uzel x má dvě barvy najednou.

Metoda RBDeleteFixUp se snaží obnovit platnost první vlastnosti. Cílem cyklu `while` na řádcích 1–22 je posunovat nadbytečnou černou barvu nahoru po stromu dokud

1. x ukazuje na červený uzel, který se jednoduše přebarví na černo, nebo
2. x ukazuje na kořen stromu, černou barvu navíc zapomeneme, nebo
3. se provede nějaká rotace a přebarvení uzlů.

Uvnitř cyklu, x vždy ukazuje na černý uzel, který není kořenem, a má jednu černou barvu navíc. Na řádku 2 určujeme, je-li x levým nebo pravým potomkem svého rodiče $p[x]$. Jako ukázku uvedeme jen kód pro levého potomka, pro pravého je kód symetrický. Zavedeme ukazatel na uzel w, což je sourozenec uzlu x. Protože x má dvě černé barvy, uzel w nemůže být $NULL[T]$; jinak by počet černých uzlů na cestě od $p[x]$ do null-ového uzlu w byl menší než na cestě od $p[x]$ do x.

Čtyři možné případy v cyklu `while` jsou zobrazeny na obrázku 6.21. Před detailní analýzou, prozkoumejme jak je při transformacích zachována vlastnost 4. Klíčovou myšlenkou je zachování počtu černých uzlů na cestách od kořene (včetně) do jednotlivých podstromů $\alpha, \beta, \ldots, \zeta$. Například na obrázku 6.21 (a) ilustrující případ 1, počet černých uzlů do podstromů α a β je 3 (x má dvě černé), před i po transformaci. Podobně počet černých uzlů na cestě do podstromů $\gamma, \delta, \varepsilon, \zeta$ je 2, před i po transformaci. Na obrázku 6.21 (b) musíme brát v potaz barvu c, která může být jak červená tak i černá.

Definujeme-li $count(red) = 0$ a $count(black) = 1$, potom počet černých uzlů od kořene do uzlu α je $2 + count(c)$, před i po transformaci. Ostatní případy se dokáží obdobně.

Případ 1 (řádky 5–8, obrázek 6.21 (a)) nastane, když uzel w je červený. Jelikož w musí mít černé potomky, můžeme uzly w a $p[x]$ přebarvit opačně a provést levou rotaci okolo $p[x]$ bez narušení vlastností Red–Black stromu. Nový uzel x, jeden z potomků w, je nyní černý, a případ 1 jsem předělali na jeden z případů 2,3 nebo 4.

Případy 2, 3 a 4 nastanou pokud w je černý; případy se odlišují barvou potomků w. V případě 2 (řádky 10–11, obrázek 6.21 (b)) jsou oba potomci černí. Protože w je také, vezmeme jednu černou barvu jak z x tak z w, v x ponecháme jen jednu černou a ve w se barva změní na červenou. Černou barvu navíc přidáme do $p[x]$. Cyklus `while` opakujeme s $p[x]$ jako novým uzlem x. Jestliže na případ 2 narazíme skrze případ 1, barva c nového uzlu x je červená, protože původní $p[x]$ byl červený a tudíž cyklus skončí.

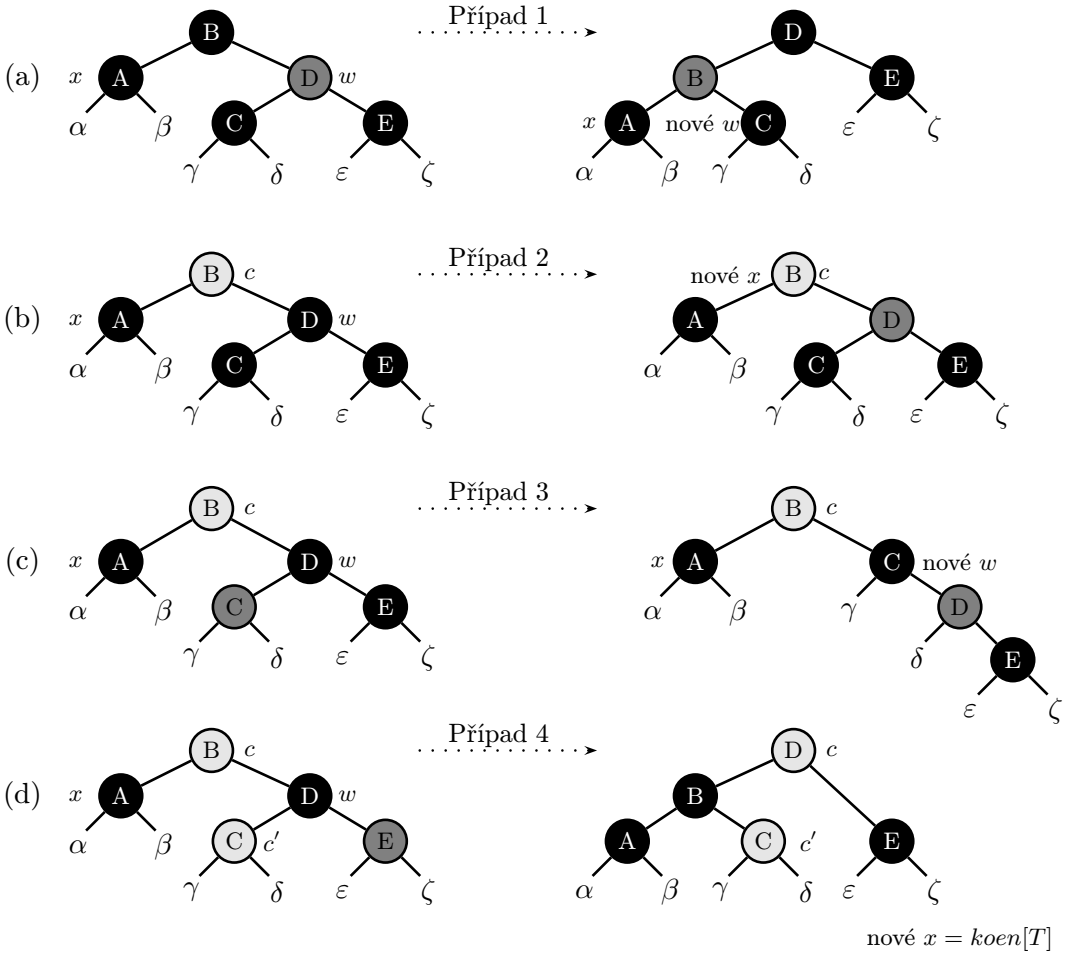
Případ 3 (řádky 13–16, obrázek 6.21 (c)) nastane, když w je černý, jeho levý potomek je červený a pravý potomek černý. Bez narušení vlastností Red–Black stromu můžeme přehodit barvy u w a jeho levého potomka a potom provést pravou rotaci okolo w. Nový potomek w uzlu x má nyní černou barvu s pravým černým potomkem a případ jsme transformovali na případ 4.

Případ 4 (řádky 17–20, obrázek 6.21 (d)) nastává, pokud potomek w uzlu x je černý a pravý potomek w má červenou barvu. Změnou několika barev a provedením levé rotace okolo $p[x]$ odstraníme nadbytečnou černou barvu z x. Přiřazením kořene stromu do x se v následujícím testu cyklus ukončí.

Časová složitost: jelikož výška Red–Black stromu s n uzly je $O(\log n)$, metodu RBDelete lze vykonat v čase $O(\log n)$. Uvnitř metody RBDeleteFixUp případy 1,3 a 4 končí po provedení konstantního počtu změn barev a nejvýše tří rotací. Jedině v případě 2 se x posunuje po stromu nahoru bez provádění rotací v čase nejvýše $O(\log n)$. Proto metoda RBDeleteFixUp pracuje v čase $O(\log n)$ s provedením nejvýše tří rotací. Celková časová složitost je tedy $O(\log n)$.

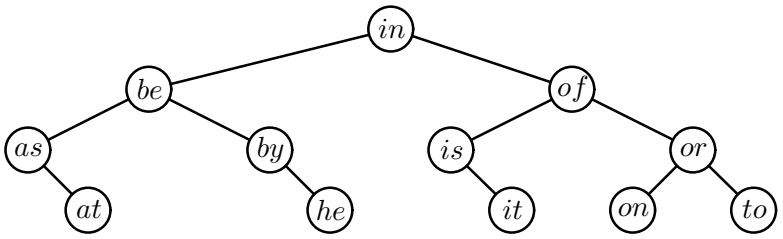
6.9 Ternární stromy

Pro uložení množiny řetězců si můžeme vybrat z několika datových struktur. Jednou z možností je použití hashovacích tabulek. Jejich výhodou je rychlý přístup k datům, ale nevýhodou je ztráta informace o relativním pořadí. Jinou možností je uložení řetězců do binárního vyhledávacího stromu, jehož výhodou je malá prostorová složitost. Dále můžeme použít tzv. vyhledávací trie, jsou rychlé ale mají velkou prostorovou složitost.



Obrázek 6.21: Možné případy ve funkci RBDelete

Nejtmavší uzly mají černou barvu, tmavě šedé červenou a světlé mohou mít buď černou nebo červenou barvu (jsou označeny c a c'). Písmena $\alpha, \beta, \ldots, \zeta$ představují jednotlivé podstromy. I v každém z případů je strom vlevo transformován na strom vpravo provedením změn barev nebo/a rotacemi. Uzel označený x má jednu černou barvu navíc. Jedině případ 2 vede k pokračování cyklu `while`. **(a)**Případ 1 se transformuje na případy 2, 3 a 4 výměnou barev uzlů B a D a levou rotací. **(b)** V případě 2 se černá barva navíc představovaná uzlem x posune nahoru po stromu obarvením uzlu D na červeno a nastavením x na B. Jestliže se do případu 2 dostaneme přes případ 1, cyklus `while` se ukončí, protože c je červené. **(c)** Případ 3 je převeden na případ 4 výměnou barev C a D a pravou rotací. **(d)** V případě 4 lze černou barvu navíc reprezentovanou x zlikvidovat přebarvením několika barev a levou rotací. Cyklus tím končí.



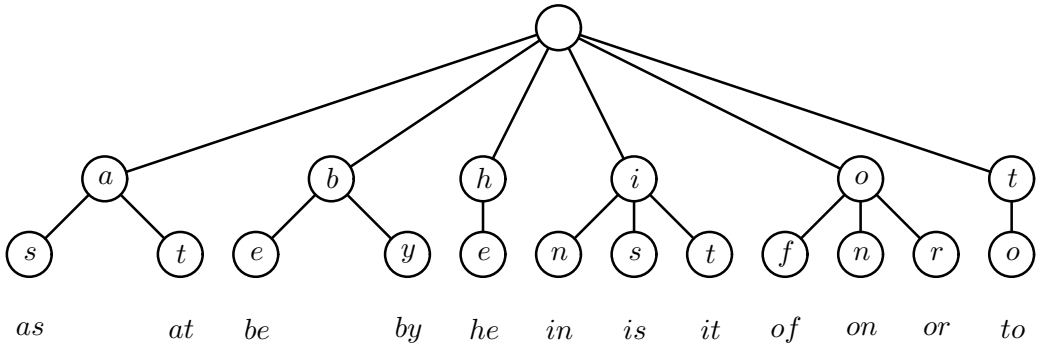
Obrázek 6.22: Binární strom pro 12 slov

Na obrázku 6.22 je binární vyhledávací strom, který reprezentuje 12 obvyklých dvouhláskových anglických slov. Pro každý uzel platí, že jeho leví následovníci mají hodnotu menší než je hodnota tohoto uzlu a všichni praví následovníci mají hodnotu větší než je hodnota tohoto uzlu. Vyhledávání začíná v kořenu tohoto stromu. Abychom například našli řetězec „*on*“, porovnáme jej s „*in*“ a pokračujeme doprava, porovnáme s „*of*“, pak pokračujeme doprava a porovnáme jej s „*or*“, jdeme doleva a porovnáme jej s „*on*“. Řetězec byl nalezen. Pro každé porovnání jsou přístupny všechny znaky řetězce.

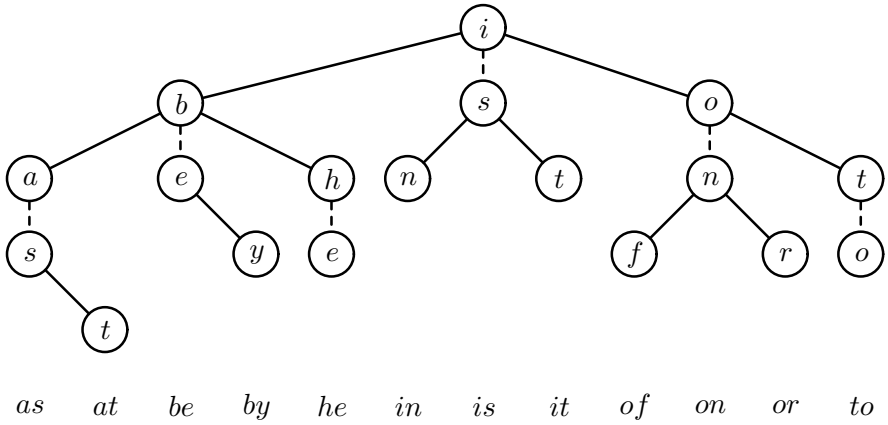
Vyhledávací trie je stromová struktura, jejíž historie sahá do roku 1959. Digitální vyhledávací trie ukládají řetězce znak po znaku. Na obrázku 6.23 je strom, reprezentující stejnou množinu 12 slov jako na předchozím obrázku 6.22. Každé vstupní slovo je zobrazeno pod uzlem, který jej reprezentuje. Dvouhlásková slova vypadají na obrázku nejnázorněji (pozn.:všechny struktury samozřejmě slouží pro uložení libovolně dlouhého slova). Ve stromě, který reprezentuje např. slova skládající se jen z malých písmen, má každý uzel 26 následovníků (anglická abeceda) - na obr. 6.23 nejsou pro přehlednost zobrazeny všechny větve. Vyhledávání je velmi rychlé, v každém uzlu vlastně přistupujeme k prvku pole (jednomu z 26), testujeme na null a vybíráme větev. Trie bohužel mají nadměrnou prostorovou složitost. Uzel, ze kterého vychází 26 větví typicky vyžaduje 104 bytů, uzel s 256 větvemi spotřebuje $1kB$.

Ternární strom (dále TS) kombinuje přednosti trií a binárních vyhledávacích stromů - časovou efektivitu a prostorovou efektivitu. Stejně jako trie postupují znak po znaku. Stejně jako binární stromy jsou prostorově efektivní, každý uzel má 3 potomky (oproti dvěma u binárních stromů). Při vyhledávání se porovnává aktuální znak řetězce se znakem v uzlu. Pokud hledaný znak je menší než aktuální uzel, pokračujeme levým potomkem. Je-li větší pokračujeme pravým potomkem. Pokud jsou si znaky rovny, pokračujeme prostředním potomkem a vyhledáváme následující znak v řetězci.

Na obrázku 6.24 je vyváženývyvážený ternární strom pro stejnou množinu 12 slov. Ukazatele na menší a větší následovníky jsou reprezentovány plnou čarou, zatímco ukazatele na následovníka, který je roven aktuálnímu



Obrázek 6.23: Trie pro 12 slov



Obrázek 6.24: Ternární strom pro 12 slov

znaku řetězce, je reprezentován čárkovaně. Každé vstupní slovo se nachází pod příslušným konečným uzlem. Vyhledávání slova „*is*“ začíná v kořenu stromu, pokračuje dolů a nalezne uzel ohodnocený „*s*“ a zastavuje se po dvou porovnáních. Při vyhledávání řetězce „*ax*“ provede tři porovnání pro první znak „*a*“ a dvě porovnání pro druhý znak „*x*“, hledání pak končí neúspěchem neboť slovo není ve stromě.

Idea ternárních stromů vznikla již kolem roku 1964, bylo dokázáno mnoho teoretických poznatků o TS, např. že vyhledání řetězce délky k ve stromě o n uzlech bude vyžadovat v nejhorším případě $O(k+n)$ porovnání.

Každý uzel ternárního stromu může být reprezentován takto:

```
typedef struct tnode *Tptr;
typedef struct tnode {
    char splitchar;
    Tptr lokid, eqkid, hikid;
} Tnode;
```

Hodnota uzlu je označena jako *splitchar* a tři pointery *lokid*, *eqkid* a *hikid* ukazují na jeho tři potomky. Kořen je deklarován jako *Tptr root*. Bude reprezentován každý znak řetězce včetně ukončovacího symbolu *null*.

6.9.1 Vyhledávání

Nejprve rekurzivní verze vyhledávací funkce *rsearch*. Vrací 1 pokud řetězec s je v podstromu s kořenem p a jinak vrací 0. Funkce se volá $rsearch(root, s)$:

```
int rsearch(Tptr p, char *s)
{   if (!p) return 0;
    if (*s < p->splitchar)
        return rsearch(p->lokid, s);
    else if (*s > p->splitchar)
        return rsearch(p->hikid, s);
    else {
        if (*s == 0) return 1;
        return rsearch(p->eqkid, ++s);
    }
}
```

První příkaz if vrací 0 jestliže vyhledávání skončí neúspěchem. Další dva podmíněné příkazy pokračují prohledáváním levého resp. pravého podstromu a poslední else vrací 1 jestliže aktuální uzel je ukončovacím znakem řetězce, jinak se posuneme na následující znak v řetězci a prohledáváme prostřední podstrom.

Iterační verze (funkce *search*s jedním argumentem):

```
int search(char *s)
{   Tptr p;
    p = root;
    while (p) {
        if (*s < p->splitchar)
            p = p->lokid;
        else if (*s == p->splitchar) {
            if (*s++ == 0)
                return 1;
            p = p->eqkid;
      } else
            p = p->hikid;
    }
    return 0;
}
```

Většinou je doba běhu rekurzivní funkce kolem 5% času iterační verze.

6.9.2 Vkládání nového řetězce

Funkce *insert* vloží nový řetězec do stromu, pokud tento řetězec již ve stromě není (pak se neprovede nic). Funkci pro vložení řetězce s voláme pomocí $root = insert(root, s)$;. První příkaz if inicializuje nový uzel. Pak se pokračuje obvyklým způsobem.

```
Tptr insert (Tptr p, char *s)
{ if (p == 0) {
        p = (Tptr) malloc(sizeof(Tnode));
        p->splitchar = *s;
        p->lokid = p->eqkid = p->hikid = 0;
  }
  if (*s < p->splitchar)
        p->lokid = insert(p->lokid, s);
  else if (*s == p->splitchar) {
        if (*s != 0)
            p->eqkid = insert(p->eqkid, ++s);
  } else
        p->hikid = insert(p->hikid, s);
  return p;
}
```

Existuje několik způsobů vkládání nového řetězce do ternárního stromu. Jedním ze způsobů je vyvážený TS, kdy jako kořen podstromu vybíráme vždy medián z příslušné množiny. Další možností je ta, že nejprve setřídíme vstupní množinu a jako kořen stromu vložíme prostřední řetězec, obdobně postupujeme dále.

6.9.3 Porovnání s ostatními datovými strukturami

Ternární vyhledávací stromy jsou viditelně rychlejší než hashing v případě neúspěšného hledání. Mohou odhalit neshodu po porovnání jen několika znaků, zatímco hashovací tabulky zpracují celý klíč. V případě množiny dat s velmi dlouhými klíči a neshodách v prvních znacích potřebují TS pouze jednu pětinu času oproti hashování.

Alternativní reprezentace TS je prostorově efektivnější: pokud každý podstrom obsahuje jen jeden řetězec, uložíme pointer řetězce na sebe a každý uzel bude potřebovat tři bity, které určí zda jejich potomek ukazuje na uzel nebo nebo na řetězec. Tento kód je méně úsporný, ale redukuje počet uzlů ternárního stromu tak, že se prostorová složitost tohoto TS blíží prostorové složitosti potřebné pro hashování (jinak je větší u TS).

TS jsou účinné a snadno implementovatelné. Poskytují podstatné výhody jak binárních stromů tak trií. Zdá se, že jsou lepší než hashování neboť TS nezpůsobují další režie pro vkládání nebo úspěšné vyhledávání. Změna velikosti ternárního stromu není problematická narozdíl od hashovacích tabulek, které musí být při změně velikosti přestavěny.

TS byly užívány několik let pro reprezentaci Anglických slovníků v komerčním OCR (Optical Character Rrecognition) systému v Bellových laboratořích (Bell Labs).

6.9.4 Další operace nad ternárními stormy

Průchod ternárním stromem

Můžeme například vytisknout řetězce v setříděném pořadí pomocí rekurzivního průchodu ternárním stromem.

```
void traverse (Tptr p) {     if (!p) return;
     traverse (p->lokid);
     if (p->splitchar)
          traverse (p->eqkid);
     else
          printf ("%s/n", (char *) p->eqkid);
     traverse (p->hikid);
}
```

Jednoduchá rekurzivní vyhledávání mohou nalézt předchůdce a následovníka daného prvku nebo seznam prvků v daném rozmezí. Pokud přidáme čítač ke každému uzlu, můžeme rychle spočítat prvky v daném rozmezí, spočítat kolik slov začíná daným podřetězcem nebo vybrat m-tý největší prvek. Mnoho z těchto operací vyžaduje logaritmický čas v ternárním stromě, ale lineární v hashovací tabulce.

Vyhledávání na částečnou shodu

Vyhledávaný řetězec může obsahovat jak běžné znaky, tak nevýznamné znaky „.“. Prohledávání slovníku na řetězec „*.u.u.u*“ nalezne slovo *auhuhu*, zatímco vzor „*.a.a.a*“ nalezne 94 slov, včetně *banana*, *casaba*, a *pajama*. (Tento vzor nenalezne samozřejmě *abracadabra*.)

Funkce *pmsearc* ukládá pointery na nalezená slova do $srcharr[0..srchtop-1]$ a volá se například takto: $srchtop = 0; pmsearch(root, ".a.a.a");$

```
void pmsearch(Tptr p, char *s)
{    if  (!p) return;
     nodecnt++;
     if (*s == '.' || *s < p->splitchar)
        pmsearch(p->lokid, s);
     if (*s == '.' || *s == p->splitchar)
        if (p->splitchar && *s)
           pmsearch(p->eqkid, s+1);
     if (*s == 0 && p->splitchar == 0)
           srcharr [srchtop++] =
              (char *) p->eqkid;
     if (*s == '.' || *s > p->splitchar)
           pmsearch(p->hikid, s);
}
```

Vyhledávání nejbližšího souseda, prohledávání okolí

Máme nalézt všechna slova ve slovníku, která leží v Hammingově vzdálenosti od klíčového slova. Například hledání všech slov ve vzdálenosti 2 od *Dobbs* najde *Debby*, *hobby*, a 14 dalších slov.

```
void nearsearch(Tptr p, char *s, int d)
{   if (!p || d < 0) return;
    if (d > 0 || *s < p->splitchar)
        nearsearch(p->lokid, s, d);
    if (p->splitchar == 0) {
        if ((int) strlen(s) <= d)
            srcharr[srchtop++] = (char *) p->eqkid;
    } else
        nearsearch(p->eqkid, *s ? s+1:s,
            (*s == p->splitchar) ? d:d-1);
    if (d > 0 || *s > p->splitchar)
        nearsearch(p->hikid, s, d);
}
```

6.10 B-stromy

V předchozích kapitolách jsme si ukázali některá kritéria vyváženosti stromů. Nyní zaměříme pozornost na konstrukci vícecestných vyhledávacích stromů a definici přiměřených kritérií vyváženosti.

Velmi vhodná kritéria navrhl R. Bayer v roce 1970 (např. [16]). Strom sestrojený podle jeho návrhu se nazývá B-strom. Základní myšlenka spočívá v tom, že uzly stromu (dále je budeme označovat jako **stránky**) mohou obsahovat n až $2n$ klíčů – položek. Potom složitost operace vyhledávání v takovémto stromu bude v nejhorším případě řádu $\log_n(N)$, kde N je počet položek ve stromu. Dalším důležitým hlediskem je faktor využití paměti[3] který v případě B-stromu je minimálně 50 procent. To vše při zachování relativně malé složitosti operací potřebných na údržbu této struktury. Definujme nejprve přesně, co máme na mysli pod pojmem B-strom.

Definice 6.8 ***B-strom*** *řádu n je $(2n+1)$-ární strom, který splňuje následující kritéria:*

1. *Každá stránka obsahuje nejvýše $2n$ položek (klíčů).*
2. *Každá stránka, s výjimkou kořenové obsahuje alespoň n položek.*
3. *Každá stránka je buď listovou tj. nemá žádné následovníky nebo má $m+1$ následovníků, kde m je počet klíčů ve stránce.*
4. *Všechny listové stránky jsou na stejné úrovni.*

[3] Z předchozího je patrné, že ne všechno místo zabrané stránkami B-stromu musí být naplněno daty - položkami. Faktorem využití paměti rozumíme poměr mezi obsazeným místem ve stránkách a celkovým místem.

6.10.1 Vyhledávání v B-stromu

Další otázkou je uspořádání klíčů ve stránce. Z tohoto pohledu je B-strom přirozeným zobecněním binárního vyhledávacího stromu. Klíče jsou ve stránce udržovány v uspořádaném pořadí, od nejmenšího po největší. Potom m klíčů definuje $m+1$ intervalů, kterým odpovídá $m+1$ následovníků stránky p_0, p_2, až p_m. Vyhledávání v B-stromu pak probíhá následujícím způsobem. Označme klíče ve stránce symboly $k_1, k_2, \ldots, k_m$. V případě, že hledaná hodnota x se nerovná žádnému z klíčů k_i, pokračujeme prohledáváním stránky následovníka, kterého určíme takto:

1. Jestliže $k_i < x < k_{i+1}$ pro $1 \leq i < m$, pokračujeme zpracováním stránky následovníka p_i.

2. Jestliže $k_m < x$, vyhledávání pokračuje na stránce p_m.

3. Jestliže $x < k_1$, vyhledávání pokračuje na stránce p_0.

Jestliže následovník, vybraný tímto způsobem, neexistuje, položka s klíčem x se ve stromu nevyskytuje.

6.10.2 Vkládání do B-stromu

Přidávání do B-stromu je poměrně jednoduché. Nejjednodušší případ nastane, když se má přidat položka do stránky která ještě není zaplněná, tzn. obsahuje méně než $2n$ položek. V tomto případě se prostě začlení nová položka do této stránky, při zachování uspořádání položek v rámci stránky.

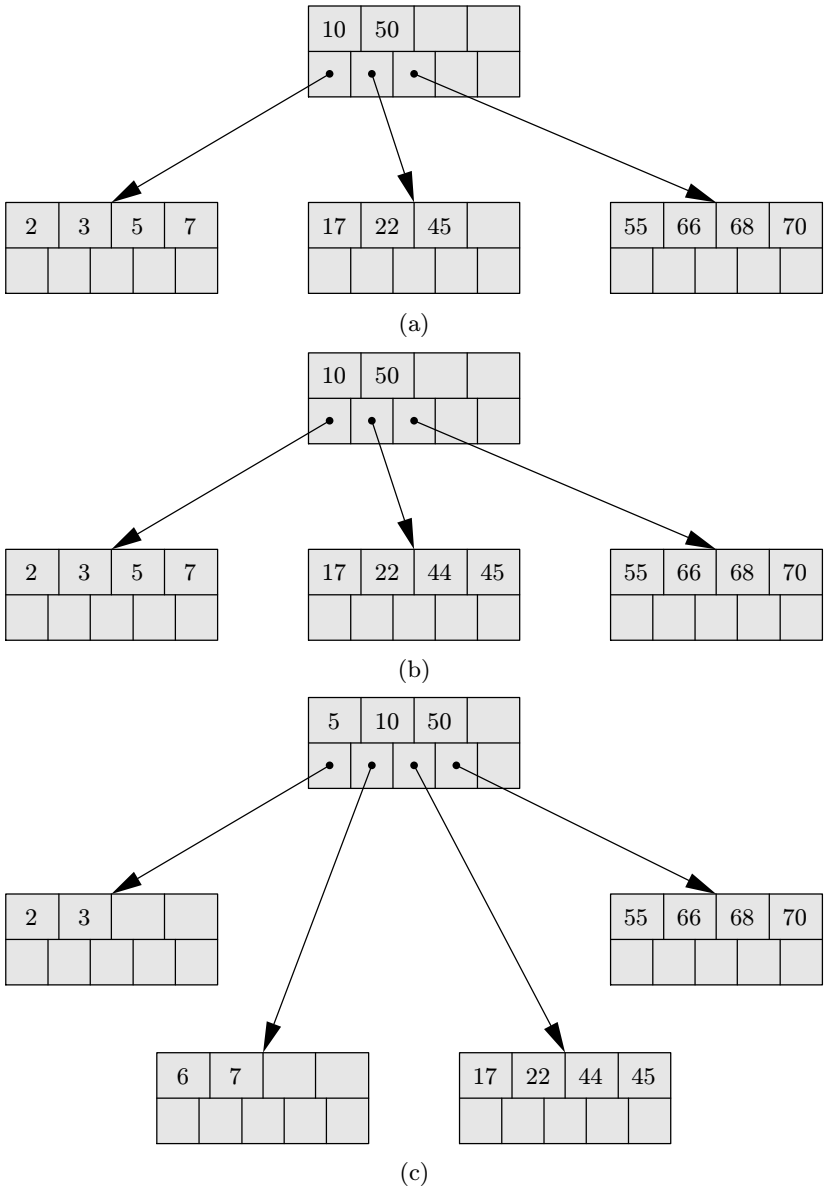
V případě, že stránka obsahuje $2n$ položek, je potřeba provést určité úpravy struktury stromu, které vedou k vytvoření jedné nebo více nových stránek. Popišme proces přidávání nové položky s klíčem x do stránky C.

1. Stránka C se rozdělí na dvě stránky C a D, tzn. vytvoří se jedna nová stránka D.

2. Všech $m+1$ položek se rozdělí rovnoměrně mezi tyto stránky, přičemž zůstane jedna položka s klíčem K nezařazená. Stránka C obsahuje všechny položky s klíči $k_i \leq K, 1 \leq i \leq n$ a stránka D všechny položky s klíči $l_i \geq K, 1 \leq i \leq n$.

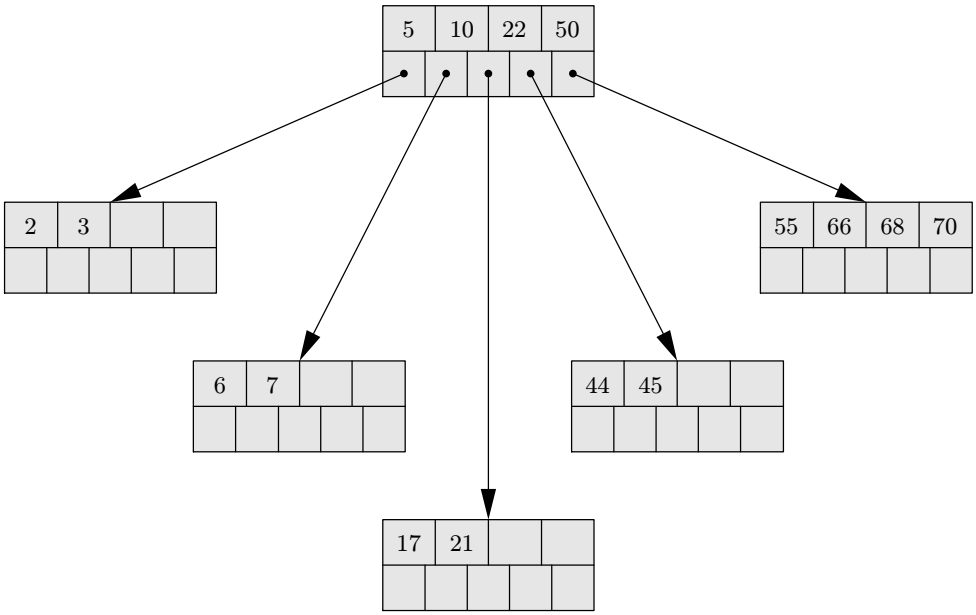
3. Zbývá začlenit položku s klíčem K do stránky předchůdce, což můžeme chápat jako přidávání nové položky do této stránky.

Z popsaného algoritmu vyplývají následující úvahy. Rozdělené stránky obsahují přesně n položek. V případě, že stránka předchůdce je také zaplněna, pokračuje proces štěpení do dalších úrovní, v extrémním případě se může zastavit až rozštěpením stránky kořenové V tom případě se zvětší výška B-stromu a je to také jediný možný způsob růstu výšky B-stromu.

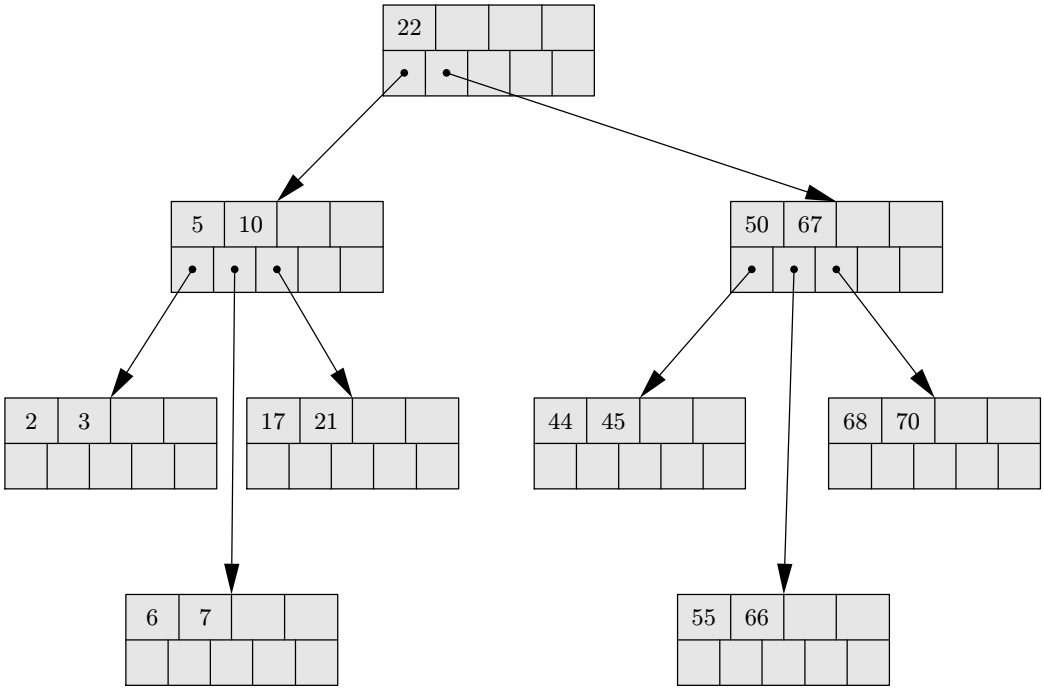


Obrázek 6.25: Vkládání do B-stromu I.

Na prvním obrázku shora je původní B-strom. Na druhém obrázku je B-strom po vložení 44. Klíč 44 byl vložen mezi klíče 22 a 45. B-strom na třetím obrázku byl modifikován vložením klíče 6. Tento klíč způsobil dělení stránky. Klíče 2 a 3 setrvaly v původní stránce, klíče 6 a 7 se přesunuly do nové stránky. Prostřední klíč 5 byl přemístěn do rodičovské stránky.



Obrázek 6.26: Vkládání do B-stromu II.
Na čtvrtém obrázku B-stromu je znázorněno jak by vypadal náš B-strom po vložení klíče 21. Jak je vidět nejenže se rozdělila stránka s klíči 17, 22, 44 a 45, ale došlo i k zaplnění kořenové stránky vlivem přesunutí klíče 22.



Obrázek 6.27: Vkládání do B-stromu III.

Poslední obrázek o vkládání do B-stromu demonstruje situaci po vložení klíče 67. Došlo k štěpení příslušné listové stránky, ale protože je kořen již zaplněn došlo i ke štěpení kořenové stránky a vzniku nového kořene s jediným klíčem 22. Zároveň se zvětšila výška B-stromu.

6.10.3 Odebírání z B-stromu

Myšlenka algoritmu odebírání z B-stromu je v podstatě stejně jednoduchá jako u přidávání do B-stromu, vyžaduje ale vyřešení většího množství detailů. Současně je celý postup analogií odebírání z binárního vyhledávacího stromu, pouze je potřeba dbát na dodržení pravidel definovaných pro B-strom.

V zásadě rozlišujeme dvě situace.

1. Položka, kterou chceme odebrat, se nachází v listové stránce. V tom případě je způsob odebrání zřejmý.

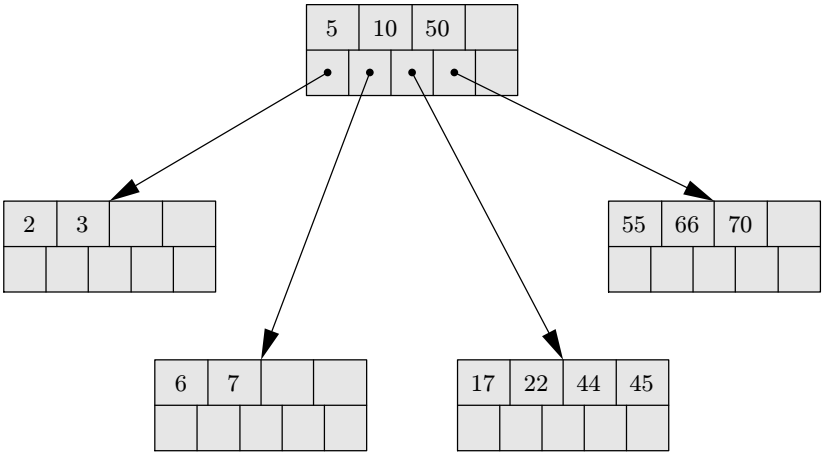
2. Položka není v listové stránce. V tomto případě je potřeba ji nahradit jedním ze dvou sousedních prvků, ve smyslu uspořádání, které je možné snadno odebrat, pokud se nachází v listové stránce. Máme na výběr buď nejbližší menší prvek nebo nejbližší větší prvek.

Při nahrazení položky v druhém případě můžeme postupovat podobně jako u binárního stromu. Budeme hledat nejbližší větší prvek. Je-li odebíraná položka umístěna ve stránce na pozici i zahájíme sestup i-tým odkazem. Dále sestupujeme ve směru nejlevějších odkazů stránky – nultých odkazů – až dosáhneme listové stránky P. Nahradíme položku, která se má odebrat, nejlevější položkou stránky P a snížíme počet položek v P.

V obou případech musíme po zmenšení stránky provádět kontrolu počtu položek m v této stránce. Pokud by nastal případ, že $m < n$, je potřeba provádět určité úpravy. Jedním z možných řešení je připojit prvek z jedné ze sousedních stránek. Tato operace ale vyžaduje umístění stránky Q do operační paměti, což je nákladná operace zejména v případě jejího načítání z disku. Je vhodné při této příležitosti provést současně připojení více prvků z Q, a to tak, aby prvky byly ve stránkách P a Q rozděleny rovnoměrně. Této činnosti můžeme říkat vyvažování.

Pokud se stane, že nelze ze stránky Q odebrat žádný prvek, tzn. počet prvků v Q je roven n, rovná se celkový počet prvků ve stránkách P a Q hodnotě $2n - 1$. Z toho vyplývá, že tyto stránky můžeme sloučit do stránky jediné, přičemž do ní přidáme jeden prvek (prostřední) ze stránky předchůdce. Prvky sloučíme do stránky P a stránky Q se zbavíme. Odebrání prvku ze stránky předchůdce ovšem může vést k poklesu počtu prvků v této stránce pod hodnotu n. To má za následek provádění úprav na další úrovni. V extrémním případě se může slučování stránek šířit až ke stránce kořenové V případě, že se kořenová stránka úplně vyprázdní, odstraní se a výška B-stromu se sníží. To je současně jediný možný způsob zmenšení výšky stromu.

Příklad 6.2
Ukažme si na následujícím příkladě rušení položek v B-stromu. Vyjděme ze stromu na obrázku 6.25(c). Nejdříve zrušíme položku s klíčem 68. Tato



Obrázek 6.28: B-strom po odebrání 68

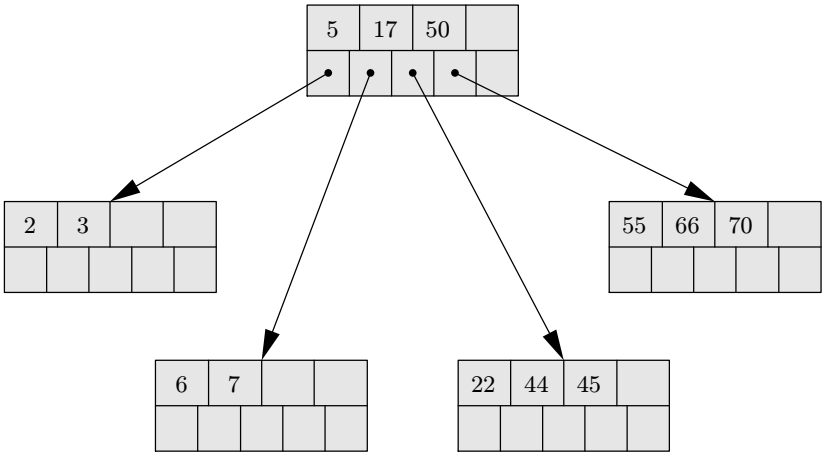
položka se nachází v listové stránce a příslušná stránka obsahuje dostatečný počet položek, tudíž můžeme tuto položku jednoduše odstranit (viz obrázek 6.28).

Jako další zrušíme položku s klíčem 10. Tato položka se nachází ve vnitřní stránce (v našem případě, shodou okolností i v kořenové, což na věci nic nemění), takže musíme najít za tuto položku náhradu. Jak bylo popsáno výše, budeme hledat položku s nejbližším větším klíčem. Touto položkou je 17. Vyjmeme 17 z listové stránky a nahradíme jí rušenou položku 10 (viz obrázek 6.29).

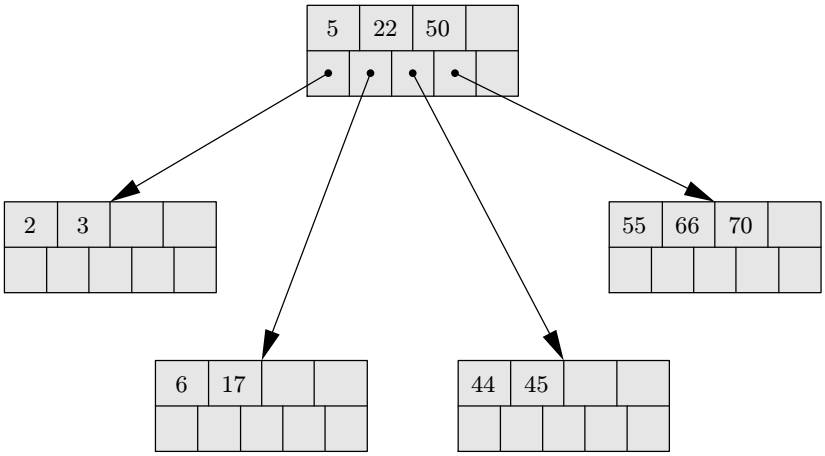
Dále budeme rušit položku s klíčem 7. Vzhledem k tomu, že stránka obsahuje jen dvě položky (tj. přesně n položek), nelze ji jednoduše odstranit, protože bychom porušili vlastnosti B-stromu. Problém můžeme vyřešit přenosem položky ze sousední stránky. Ze sousedních stránek budeme volit tu, která obsahuje více položek – v našem případě pravá sousední. Při přesunech nesmíme zapomenout na klíč v rodičovské stránce, který leží „mezi“ oběma stránkami! Dostáváme posloupnost klíčů $[6, 7, 17, 22, 44, 45]$. Položku s klíčem 7 zrušíme a posloupnost $[6, 17, 22, 44, 45]$ rozdělíme rovnoměrně mezi obě stránky ($[6, 17]$ a $[44, 45]$), přičemž prostřední klíč (22) náleží do rodičovské stránky. Výsledek operace je na obrázku 6.30.

Tento způsob „výpůjčky“ položek ze sousední stránky můžeme použít jen v případě, že sousední stránka obsahuje aspoň $n+1$ položek. Stránka ve které rušíme obsahuje $n-1$ položek. Spolu s jednou položkou z rodičovské stránky dostáváme $(n-1)+1+(n+1) = 2n+1$ položek. Tento počet, lze bez problémů rozdělit mezi dvě stránky a jednu položku pro rodičovskou stránku.

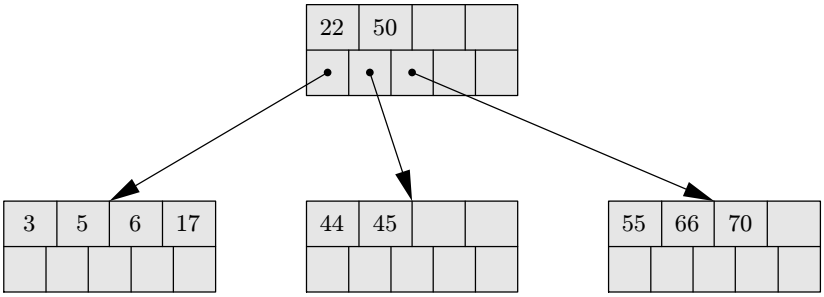
Co se stane v případě, že obě sousední stránky budou přesně n položek? V tomto případě bychom podle výše zmíněného postupu dostali jen $(n-1)+$



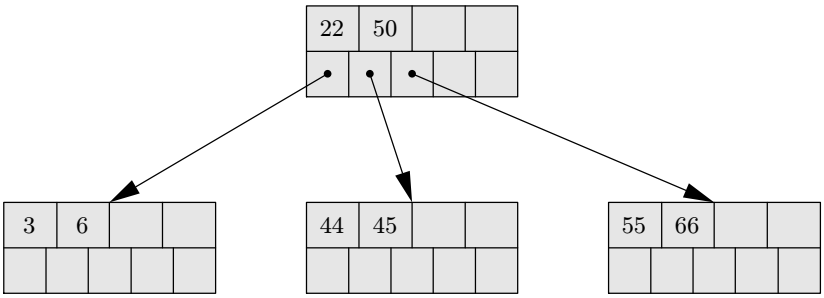
Obrázek 6.29: B-strom po odebrání 10



Obrázek 6.30: B-strom po odebrání 7



Obrázek 6.31: B-strom po odebrání 2

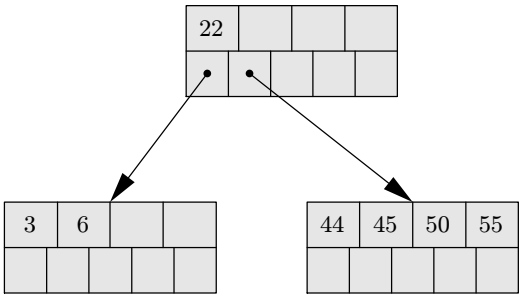


Obrázek 6.32: B-strom po odebrání 5, 17, 70

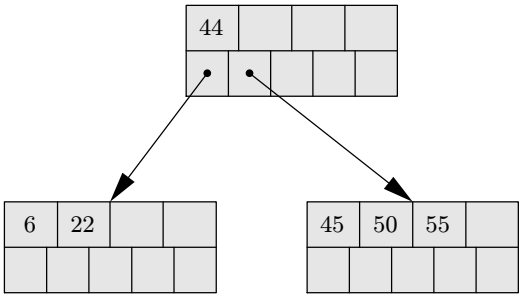
$1 + n = 2n$ položek a tudíž nejsme schopni sestavit dvě stránky plus navíc jednu položku pro rodičovskou stránku. Je jasné, že v tomto případě musí dojít ke sloučení stránek. Výsledná sloučená stránka bude potom obsahovat všech $2n$ položek. V tomto případě není nutné přesouvat jednu položku do rödiče, protože jedna stránka zanikla, z čehož plyne, že v rodičovské stránce stačí o jeden ukazatel méně, a tím i o odpovídající položku méně. V našem příkladě tato situace nastane rušením položky s klíčem 2. Spolu se sousední stránkou a rodičem dostáváme posloupnost položek $[2, 3, 5, 6, 17]$. Zrušením 2 máme posloupnost $[3, 5, 6, 17]$ jen o čtyřech položkách, kterou zaplníme jen jednu stránku. Výsledný strom je na obrázku 6.31.

Bez větších obtíží lze zrušit položky s klíči 5, 17, a 70 (viz obrázek 6.32). Odebrání položky 66 způsobí zánik další stránky (viz obrázek 6.33). Zrušením položky s klíčem 3 dojde k přesunu položek mezi stránkami (viz obrázek 6.34), zrušení položky 55 je triviální (viz obrázek 6.35).

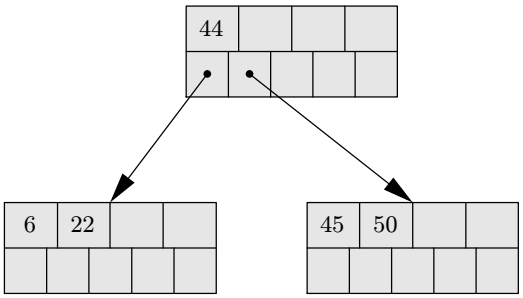
Jako poslední budeme rušit položku s klíčem 22. Je jasné, že bude nutné přesunovat položky ze sousední stránky. Dostáváme posloupnost klíčů $[6, 22, 44, 45, 50]$. Zrušením klíče 22 dostaneme posloupnost $[6, 44, 45, 50]$ právě na jednu stránku. Odstraněním položky 44 z kořene se dostáváme do situace, kdy je možné kořen zrušit, výška B-stromu se sníží o jedničku a úlohu kořene přebírá jiná stránka. Výsledný B-strom bude v našem případě



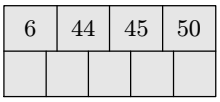
Obrázek 6.33: B-strom po odebrání 66



Obrázek 6.34: B-strom po odebrání 3



Obrázek 6.35: B-strom po odebrání 55



Obrázek 6.36: B-strom po odebrání 22

obsahovat jedinou stánku, která je zároveň kořenem stromu i jeho listem. Konečná podoba stromu je vyobrazena na obrázku 6.36.

6.10.4 Hodnocení B-stromu

B-strom představuje velice efektivní strukturu pro uchovávání a vyhledávání hodnot. Její použití je výhodné zejména v případě, že se hodnoty nevejdou do operační paměti a musejí se uchovávat v sekundární paměti - např. na pevném disku. Potom se snažíme omezit na minimum počet přístupů na disk, protože právě přístup na disk je v tomto případě časově nejnáročnější operace. Srovnáme-li B-strom s binárním vyhledávacím stromem jeho zřejmá výhoda spočívá ve větším základu u logaritmu určujícího třídu složitosti a tím pádem i menším počtu přístupů na disk. (Srovnej například $\log_2(10^6)$ a $\log_{100}(10^6)$.)

Kapitola 7

Hashování

Mnohé aplikace nepotřebují ke svému provozu celou škálu operací podporovaných v dynamických strukturách (např. stromech), ale vystačí jen s operacemi *Insert, Search, Delete*. Například kompilátory programovacích jazyků potřebují spravovat tabulky identifikátorů v překládaném programu a dá se předpokládat, že kompilátor nebude potřebovat operace výběru nejmenšího identifikátoru a podobné.

Hashovací tabulky nabízí nástroje jak vytvářet velice efektivní tabulky, kde složitost vyhledávání je, za několika rozumných předpokladů, rovna $O(1)$. I když nejhorší případ je stále $\Theta(n)$.

Přímo adresovatelné tabulky a hashovací tabulky jsou rozšířením standardních polí. Přímo adresovatelné tabulky používají přímo klíče jako indexy v poli, hashovací tabulky transformují prostor klíčů o velmi velké mohutnosti pomocí hashovací funkce do relativně malého prostoru indexů pole.

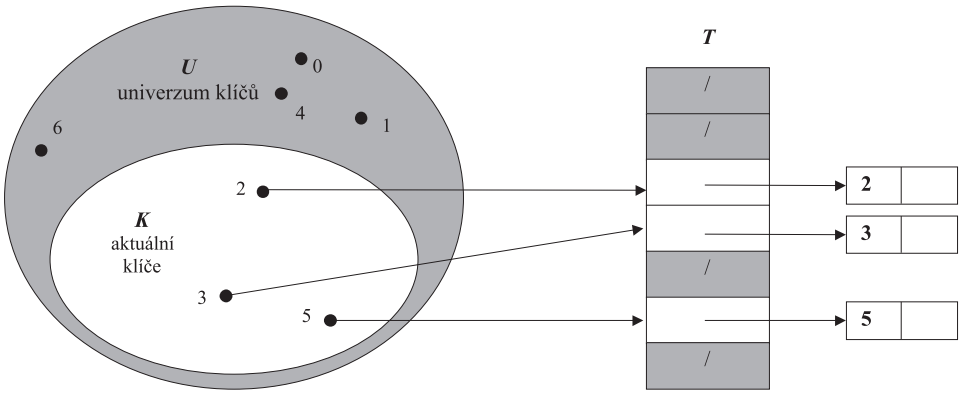
7.1 Přímo adresovatelné tabulky

Přímé adresování je jednoduchá technika, která dobře funguje, pokud univerzum klíčů U má malou mohutnost. Předpokládejme, že aplikace potřebuje ke své činnosti dynamicky se měnící množinu a klíče prvků množiny náleží do univerza $U = \{0, 1, \ldots, m-1\}$, kde m není velké. Dále předpokládejme, že žádné dva prvky nemají shodné klíče.

Pro reprezentaci takové množiny použijeme pole nebo **přímo adresovatelnou tabulku** $T[0 \ldots m-1]$. Každá pozice (**slot**) koresponduje s nějakým klíčem univerza U. Z obrázku 7.1 je patrno, že slot k ukazuje na prvek s klíčen k. Jestliže množina neobsahuje prvek s klíčem k, pak tento slot má hodnotu $NULL$.

Implementace je velice triviální:

```
t_Item* Search(t_Key k)
{
    return T[k];
```



Obrázek 7.1: Přímo adresovatelná tabulka

```
}

void Insert (t_Item* item)
{
    T[item->key] = item;
}

void Delete(t_Item* item)
{
    T[item->key] = NULL;
}
```

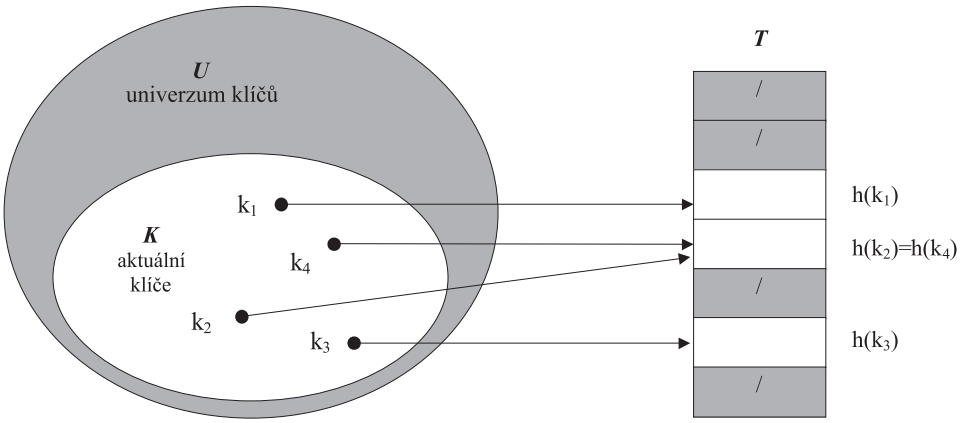
Všechny tyto operace jsou velice rychlé – pracují v čase $O(1)$.

V mnoha případech je možné uchovávat prvky množiny přímo v tabulce. Není tedy nutné mít v tabulce jen klíče a pointery na prvky množiny. Stejně tak lze vynechat vlastní klíč prvku. Máme-li totiž index prvku v tabulce, máme zároveň i klíč prvku. Musíme však být schopni nějakým mechanismem poznat, že daný slot je nebo není obsazen.

7.2 Hashovací tabulky

Hlavní problém s přímým adresováním je zřejmý: jestliže univerzum U je velké, udržování tabulky T velikosti $|U|$ je nepraktické, na většině počítačů ne-li přímo nemožné. Avšak množina všech aktuálně uložených klíčů K ($K \subset U$) může být relativně malá vzhledem k množině U.

Jestliže množina klíčů K je mnohem menší než univerzum U všech možných klíčů, hashovací tabulka spotřebuje mnohem méně místa než přímo adresovatelná tabulka. Paměťové nároky mohou být redukovány na $\Theta(|K|)$, při zachování původní složitosti vyhledání prvku, totiž $O(1)$. Jediným zhoršením je fakt, že pro hashovací tabulku platí uvedená složitost v průměrném případě, kdežto pro přímo adresovatelnou tabulku i v nejhorším případě.



Obrázek 7.2: Hashovací tabulka (ukázka kolize)

V přímo adresovatelné tabulce je prvek s klíčem k uložen ve slotu k. V hashovací tabulce je uložen ve slotu $h(k)$, kde h je hashovací funkce. **Hashovací funkce**h zobrazuje univerzum klíčů U na sloty hashovací tabulky $T[0 \ldots m-1]$:

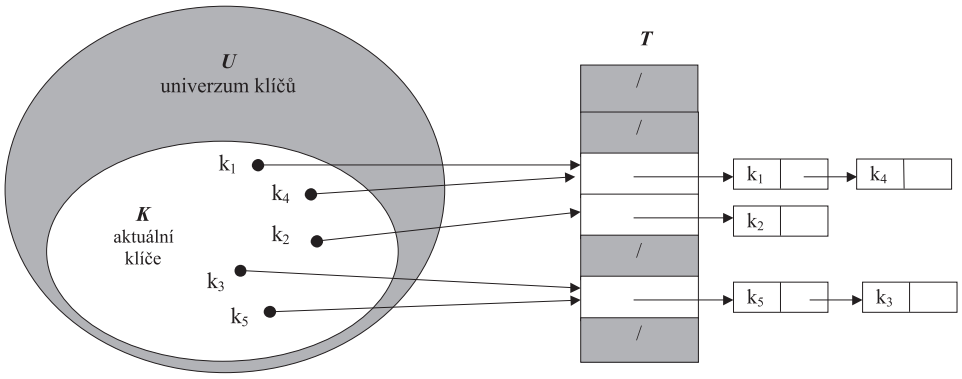
$$h : U \to \{0, 1, \ldots, m-1\}$$

Říkáme, že prvek s klíčem k je **hashován** do slotu $h(k)$, říkáme také, že $h(k)$ je **hashovací hodnota** klíče k. Hlavním účelem hashovací funkce je transformace klíčů z univerza U do jednotlivých slotů. Tím se také zmenšují nároky na paměť. Místo původních $|U|$ klíčů stačí udržovat jen m hodnot.

Je jasné, že celá tato konstrukce má jednu vadu. Dva klíče se mohou hashovací funkcí zobrazit na tentýž slot – dojde ke **kolizi** (viz obrázek 7.2). Naštěstí existují účinné techniky jak kolize prvků řešit.

Samozřejmě by bylo nejlepší najít takovou funkci, která kolizím zabrání úplně nebo aspoň minimalizuje počet kolizí. Toho by šlo dosáhnout pomocí hashovací funkce s „náhodným“ chováním. Sloveso „to hash“ znamená rozemletí, rozmělnění, vzbuzuje tedy představu různého rozdělování, přeskupování a jiných transformací klíčů. Pochopitelně funkce h musí být deterministická – pro každý klíč musí vždy spočítat stejnou hodnotu $h(k)$. Jelikož $|U| > m$ musí nevyhnutelně existovat dva kolidující klíče a úplné odstranění kolizí není možné. Kvalitním návrhem hashovací tabulky a hashovací funkce lze výrazně zmenšit počet kolizí.

V dalších částech se budeme věnovat nejjednodušší technice řešení kolizí nazývané separátní řetězení. Dále uvedeme alternativní metodu a sice otevřené adresování.



Obrázek 7.3: Ošetření kolizí pomocí separátního řetězení

7.2.1 Separátní řetězení

Technika **separátního řetězení** řeší kolize velice jednoduše. V hashovací tabulce je v každém slotu pointer na seznam a prvky se stejnou hodnotou hashovací funkce se vkládají do příslušného seznamu (viz obrázek 7.3).

Implementace operací je velice jednoduchá a přímočará.

```
void Insert (t_Item item)
{
    T[h(item.key)]. InsertToList (item);
}

bool Search(t_Key k)
{
    return T[h(k)]. SearchInList (k);
}

void Delete(t_Item item)
{
    T[h(item.key)]. DeleteFromList(item);
}
```

Časová složitost vkládání je v nejhorším případě $O(1)$ za předpokladu, že vkládaný prvek není dosud v hashovací tabulce. Pokud chceme jeho přítomnost v tabulce ověřit musíme jej vyhledat, což zvyšuje časovou složitost vkládání. Časová složitost vyhledávání a rušení prvku je úměrná délce seznamu ve slotu, kam se prvek zobrazil hashovací funkcí.

Analýza separátního zřetězení

Nechť je dána hashovací tabulka T s m sloty ve které je uloženo n prvků. Číslo α, kde

$$\alpha = \frac{n}{m}$$

se nazývá **faktor naplnění** hashovací tabulky. V našem případě je zřejmé, že toto číslo udává zároveň i průměrnou délku seznamu ve slotu. Číslo α může být menší než jedna, rovno jedné nebo větší než jedna. Při dalších úvahách budeme α považovat za konstantní a hodnoty m a n necháme růst k nekonečnu.

Efektivita pro nejhorší případ u separátního řetězení je děsivá: všech n klíčů se hashuje do jednoho slotu a vytváří tak seznam délky n. Složitost pro nejhorší případ je tedy $\Theta(n)$ plus čas nutný pro výpočet hashovací funkce. Situace je stejná jako bychom použili jednoduchý seznam. Naštěstí se hashovací tabulky nekonstruují pro tuto mizivou výkonnost.

Složitost průměrného případu závisí na tom, jak hashovací funkce rozptýlí jednotlivé klíče do prostoru slotů. Předpokládejme, že libovolný klíč je hashován do všech slotů stejně pravděpodobně, nezávisle na tom kam se hashovaly ostatní klíče. Takové hashování nazýváme **jednoduché uniformní hashování**.

Dále předpokládejme, že výpočet hodnoty hashovací funkce h pro klíč k jsme schopni provést v čase $O(1)$ a čas nutný pro prohledání seznamu ve slotu $T[h(k)]$ tabulky T je lineárně závislý na délce tohoto seznamu. Zbývá zjistit jaký je průměrný počet klíčů, které musíme porovnat s klíčem k abychom zjistili, jestli se klíč k v tabulce vyskytuje. Mohou nastat dva případy: buď budeme při hledání úspěšní nebo neúspěšní.

Věta 7.1 *Průměrná časová složitost neúspěšného vyhledání v hashovací tabulce se separátním zřetězením je $\Theta(1+\alpha)$, za předpokladu jednoduchého uniformního hashování.*

Důkaz. Za předpokladu jednoduchého uniformního hashování se každý klíč k hashuje se stejnou pravděpodobností do libovolného z m slotů tabulky. Průměrný čas neúspěšného hledání klíče k je proto průměrný čas prohledání jednoho z m seznamů. Průměrná délka každého takového seznamu je rovna faktoru naplnění $\alpha = n/m$. Tudíž lze očekávat, že budeme nuceni prozkoumat α prvků. Z toho plyne, že celkový čas pro neúspěšné hledání (plus navíc konstantní čas pro výpočet $h(k)$) je $\Theta(1+\alpha)$. ∎

Věta 7.2 *Průměrná časová složitost úspěšného vyhledání v hashovací tabulce se separátním zřetězením je $\Theta(1+\alpha)$, za předpokladu jednoduchého uniformního hashování.*

Důkaz. Opět předpokládejme jednoduché uniformní hashování, kdy se klíč k hashuje stejně pravděpodobně do všech slotů tabulky. Dále předpokládejme, že nové prvky jsou připojovány na konce příslušných seznamů ve slotech (lze dokázat, že časová složitost je shodná pro vkládání nových prvků na konec resp. na začátek seznamu). Očekávaný počet porovnání prvků je o jednu vyšší než při vkládání hledaného prvku, poněvadž vkládaný prvek

se připojí na konec seznamu bez porovnání, kdežto u hledání jsme nuceni porovnat všechny předchozí prvky a navíc ještě tento poslední prvek, u kterého indikujeme shodu. Abychom určili počet testovaných prvků, spočítáme průměr přes všech n prvků z výrazu „1 + průměrná délka seznamu ve slotu, když byl vložen i-tý prvek vložen“. Délka takového seznamu je $(i-1)/m$. Očekávaný počet otestovaných prvků je tedy roven

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

Proto celkový čas potřebný pro úspěšné vyhledání (včetně času pro výpočet hodnoty hashovací funkce) je $\Theta(2+\alpha/2-1/2m) = \Theta(1+\alpha)$. ■

Jinými slovy nám tento výpočet říká následující: jestliže velikost hashovací tabulky je úměrná počtu prvků v tabulce tj. $n = O(m)$, potom $\alpha = n/m = O(m)/m = O(1)$. Proto v průměru lze vyhledávání realizovat v konstantním čase. Jestliže používáme obousměrné seznamy u kterých je časová složitost pro nejhorší případ $O(1)$ jak pro vkládání, tak pro rušení prvků, je možno všechny operace nad hashovací tabulkou provést v konstantním čase.

7.2.2 Otevřené adresování

Při použití **otevřeného adresování** jsou všechny prvky uloženy přímo v hashovací tabulce. Každý slot tabulky obsahuje buď nějaký prvek nebo je prázdný. Při hledání prvku v tabulce systematicky prohledáváme sloty tabulky dokud nenajdeme hledaný prvek nebo najdeme prázdný slot. Na rozdíl od separátního řetězení nejsou ke slotům připojeny žádné seznamy, tabulka je jen průběžně plněna a z tohoto důvodu faktor naplnění nemůže nikdy překročit 1.

Pochopitelně lze při separátním řetězení ukládat seznamy kolidujících prvků přímo do tabulky a nebudovat dynamické seznamy zvlášť, ale hlavní výhodou otevřeného adresování je úspora místa, poněvadž místo pointerů tato metoda **vypočítává** posloupnost slotů, které je nutno prozkoumat. Seznamy kolidujících prvků jsou jakoby počítány za běhy programu. Paměť nutnou k uložení pointerů v seznamech prvků můžeme věnovat na zvýšení počtu slotů hashovací tabulky, čímž dosáhneme zmenšení počtu kolizí a vyššího výkonu.

Při vkládání prvku do hashovací tabulky provádíme takzvané **pokusy** dokud nenajdeme hledaný prvek nebo prázdný slot. Otázkou je jak volit posloupnost slotů, které budeme prozkoumávat. Místo fixní posloupnosti

$0, 1, \ldots, m-1$, což by vedlo na složitost hledání $\Theta(n)$, vybereme takovou posloupnost slotů, která závisí na vkládaném klíči. Pro určení posloupnosti rozšíříme definici hashovací funkce tak, aby zahrnovala i pořadí pokusu (počínaje 0) jako druhý parametr. Rozšířená definice bude vypadat následovně:

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

Pro každý klíč k obdržíme posloupnost pokusů (angl. probe sequence)

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

která je permutací množiny $\{0, 1, \ldots, m-1\}$. To znamená, že každý slot v tabulce bude eventuálně použit pro uložení prvku s klíčem k. V ukázkovém pseudokódu předpokládáme, že prvky jsou totožné se svými klíči.

```
void HashInsert(t_Key k)
{
    int j, i = 0;
    do
    {
        j = h(k, i);     // určení dalšího pokusu
        if (T[j] == NULL)
        {// nalezen volný slot
            T[j] = k;
            return;
        } // if
        else
            i += 1;
    }
    while (i < m);
    error "overflow"
}
```

Algoritmus vyhledávání klíče k prochází stejnou posloupnost slotů jako algoritmus pro vložení klíče k. Vyhledávání skončí buď úspěšně – klíč k je nalezen v některém z testovaných slotů. A naopak vyhledávání končí neúspěšně, pokud jsme narazili na prázdný slot. Protože kdyby hledaný klíč byl v tabulce, byl by uložen v tomto prázdném slotu.

```
bool HashSearch(t_Key k)
{
    int j, i = 0;
    do
    {
        j = h(k, i);     // určení dalšího pokusu
        if (T[j] == k)
            return true;
        i += 1;
    }
    while ((T[j] != NULL) && (i != m));
    return false;
}
```

Smazání prvku je velice obtížné. Když smažeme prvek ze slotu i, nelze tento slot jednoduše označit za prázdný. Tímto bychom mohli narušit posloupnost pokusů pro jiný klíč, který testoval slot i a způsobil v něm kolizi a tudíž musel pokračovat v pokusech až do nějakého jiného slotu j. Jednou možností jak řešit tento problém je označit slot speciálním příznakem *deleted*, který by vyhledávací procedura interpretovala jako obsazený slot a naopak vkládací procedura jako volný slot. Takovým postupem už ale nebude záviset složitost vyhledávání jen na faktoru naplnění tabulky α. Proto, když se požaduje mazání prvků z hashovací tabulky, volí se obvykle metoda separátního řetězení.

V dalších analýzách činnosti hashovacích tabulek předpokládáme tzv. **uniformní hashování**, které tvrdí, že pro každý klíč jsou všechny permutace $\{0, 1, \ldots, m-1\}$ posloupnosti pokusů stejně pravděpodobné. Uniformní hashování je zobecněním jednoduchého uniformního hashování, které pro libovolný klíč generovalo se stejnou pravděpodobností jedno číslo, kdežto uniformní hashování generuje celou posloupnost čísel. Dosáhnout v praxi uniformního hashování je obtížné, ale existují použitelné aproximace.

V dalším probereme tři nejpoužívanější techniky pro generování posloupnosti pokusů: metodu lineárních pokusů (linear probing), metodu kvadratických pokusů (quadratic probing) a dvojité hashování (double hashing). Žádná z těchto metod nesplňuje přesně požadavky kladené na uniformní hashování, protože nejsou schopny generovat více než m^2 posloupností pokusů (místo $m!$ různých posloupností). Dvojité hashování je schopno vygenerovat nejvíce různých posloupností, lze tedy od něj očekávat nejlepší výsledky.

Lineární pokusy

Mějme dánu jednoduchou hashovací funkci$h' : U \to \{0, 1, \ldots, m-1\}$. **Metoda lineárních pokusů** používá rozšířenou hashovací funkci:

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

pro $i = 0, 1, \ldots, m-1$. Pro klíč k se nejprve prozkoumá slot $T[h'(k)]$. Dále se zkoumá slot $T[h'(k) + 1]$. Postupujeme až ke slotu $T[m-1]$. V tomto okamžiku se indexy „přetočí“ a pokračujeme sloty $T[0]$, $T[1]$ až nakonec prozkoumáme slot $T[h'(k) - 1]$. Jelikož počáteční hodnota hashovací funkce určuje sekvenci pokusů, lze vygenerovat jen m pokusných sekvencí.

Metoda lineárních pokusů je snadno implementovatelná, ale vyvolává problém s tzv. **primárním shlukováním** (angl. primary clustering) prvků, které mají tendenci se shlukovat v řetězcích. Tyto řetězce vznikají při řešení kolizí tím, že kolidující záznamy vkládáme na další a další sloty, jeden za druhým. Jestliže navíc budeme vkládat prvek jehož hashovací hodnota, za jiných okolností nekolidující, padne dovnitř řetězce obsazených slotů musíme i tento prvek zařadit na konec řetězce.

Kvadratické pokusy

Metoda kvadratických pokusů používá hashovací funkci tvaru

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

kde h' je pomocná Metoda kvadratických pokusů, $c_1 \neq 0$ a $c_2 \neq 0$ jsou pomocné konstanty a $i = 0, 1, \ldots, m-1$. Prvně je otestován slot $T[h'(k)]$; další sloty jsou testovány v pořadí určeném funkcí h. Vzhledem k jejímu kvadratickému charakteru nenastává zde tak silné shlukování jako v případě lineárních pokusů, ale jen tzv. **sekundární shlukování** (angl. secondary clustering), které se podstatně méně projevuje na počtu potřebných pokusů k nalezení hledaného prvku.

Dvojité hashování

Dvojité hashování je patrně nejlepší metodou pro určení sekvence pokusů při otevřeném adresování, protože permutace slotů poskytované touto metodou mají nejblíže k náhodně voleným permutacím. **Dvojité hashování** používá hashovací funkci tvaru:

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

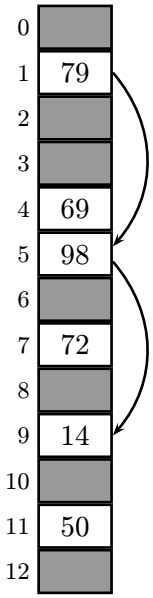
kde h_1 a h_2 jsou pomocné hashovací funkce. Na počátku je testován slot $T[h_1(k)]$; následující pokusy jsou určeny posunem o $h_2(k)$ pozic modulo m. Je jasné, že dvojité hashování umožňuje větší rozptyl pro výběr sekvencí pokusů, protože na klíči k závisí nejen počáteční pozice, ale i velikost kroku o který se v tabulce posunujeme. Na obrázku 7.4 je uveden příklad vkládání s dvojitým hashováním.

Hodnota funkce $h_2(k)$ musí být prvočíslo přibližně stejně velké jako m. Jinak, jestliže m a $h_2(k)$ mají největší společný dělitel $d > 1$ nějaký klíč k, potom je prohledáno pouze $1/d$ slotů hashovací tabulky. Velikost tabulky m se většinou volí prvočíselná a funkce h_2 se navrhuje tak, aby vždy vracela kladné číslo menší než m. Například se dají použít tyto funkce:

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod m') \end{aligned}$$

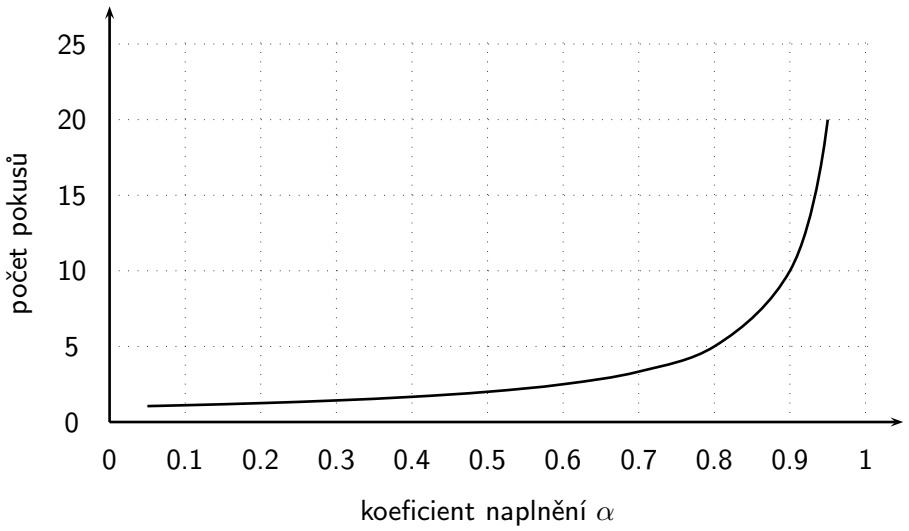
kde m' je „o něco menší“ než m (např. $m-1$, $m-2$).

Dvojité hashovánípředstavuje zlepšení oproti lineárním nebo kvadratickým pokusům, protože je schopno generovat $\Theta(n^2)$ posloupností pokusů místo $\Theta(m)$. Je to dáno tím, že hodnoty $h_1(k)$ a $h_2(k)$ z nichž se tvoří výsledná hodnota hashovací funkce se mohou měnit nezávisle na sobě. Výsledkem je, že použitím dvojitého hashování se přibližujeme ideálu uniformního hashování.



Obrázek 7.4: Vkládání dvojitým hashováním

Je dána hashovací tabulka velikosti 13, hashovací funkce $h_1(k) = k \bmod 13$ a $h_2(k) = 1 + (k \bmod 11)$. Protože $14 \equiv 1 \bmod 13$ a $14 \equiv 3 \bmod 11$, klíč 14 bude vložen do prázdného slotu 9, po prozkoumání obsazených slotů 1 a 5.



Obrázek 7.5: Nejvyšší počty pokusů při neúspěšném vyhledání jako funkce faktoru naplnění α

Analýza otevřeného adresování

Při analýze se budeme snažit vyjádřit počet pokusů při neúspěšném a úspěšném hledání jako funkci faktoru naplnění α, s tím že hodnoty n a m jdou k nekonečnu. Při použití otevřeného adresování může v každém slotu být nejvýše jeden prvek, proto $n \leq m$ a z toho plyne že $\alpha \leq 1$.

Věta 7.3 *Mějme dánu hashovací tabulku s otevřenou adresací s faktorem naplnění $\alpha = n/m < 1$. Očekávaný počet pokusů při neúspěšném hledání je nejvýše $1/(1-\alpha)$ za předpokladu uniformního hashování.*

Důkaz. Při neúspěšném hledání, všechny prohledané sloty, vyjma posledního, neobsahují hledaný klíč a posledně prohledaný slot je prázdný. Definujme

$$p_i = \text{Pravděpodobnost } \{\text{přesně } i \text{ pokusů testovalo obsazený slot}\}$$

pro $i = 0, 1, 2, \ldots$. Pro $i > n$ je $p_i = 0$, protože lze otestovat jen n aktuálně obsazených slotů. Očekávaný počet pokusů je

$$1 + \sum_{i=0}^{\infty} i p_i \tag{7.1}$$

Pro vyhodnocení výrazu 7.1 definujme

$$q_i = \text{Pravděpodobnost}\{\text{nejméně } i \text{ pokusů testovalo obsazený slot}\}$$

pro $i = 0, 1, 2, \ldots$. Dostáváme

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

Jaká je hodnota q_i pro $i \geq 1$? Pravděpodobnost, že se první pokus trefí do obsazeného slotu je n/m, proto

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

Za předpokladu uniformního hashování, druhý pokus, je-li potřeba, směřuje do jednoho ze zbylých $m - 1$ slotů, ze kterých je $n - 1$ obsazených. Tento druhý pokus provádíme jen tehdy, když je první testovaný slot obsazený, tedy:

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

Obecně, i-tý pokus provedeme jenom tehdy, bylo-li prvních $i - 1$ slotů obsazeno a další testovaný slot je, se stejnou pravděpodobností, kterýkoliv ze zbývajících $m - i + 1$ slotů, z nichž $n - i + 1$ je obsazených. Tudíž

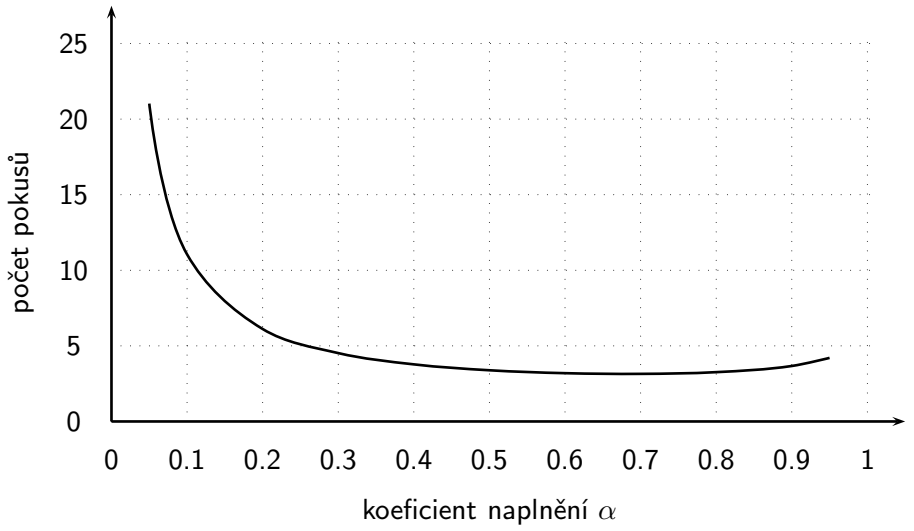
$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\ &= \frac{1}{1-\alpha} \end{aligned} \tag{7.2}$$

Intuitivní interpretace rovnice 7.2 je velice jednoduchá: první pokus se provede vždy, s pravděpodobností přibližně α se provede druhý pokus, s pravděpodobností přibližně α^2 třetí pokus a tak dále. ■

Jestliže je α konstantní, věta 7.3 tvrdí, že neúspěšné hledání lze provést v čase $O(1)$. Například, jestliže hashovací tabulka je zaplněna z poloviny ($\alpha = 0,5$), očekávaný počet pokusů při neúspěšném hledání bude nejvýše $1/(1 - 0,5) = 2$. Vývoj počtu pokusů v závislosti na α je zobrazen na grafu 7.5.

Věta 7.4 *Vložení prvku do hashovací tabulky s otevřenou adresací při faktoru naplnění α průměrně vyžaduje nejvýše $1/(1-\alpha)$ pokusů za předpokladu uniformního hashování.*

Důkaz. Prvek se dá do tabulky vložit jen pokud není plná, čili $\alpha < 1$. Před vložením prvku do tabulky se musí nejprve provést neúspěšné hledání následované vložením prvku do prvního volného slotu. Proto očekávaný počet pokusů je $1/(1 - \alpha)$. ■



Obrázek 7.6: Nejvyšší počty pokusů při úspěšném vyhledání jako funkce faktoru naplnění α

Věta 7.5 *V hashovací tabulce s faktorem naplnění $\alpha < 1$, očekávaný počet pokusů při úspěšném hledání je nejvýše*

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

za předpokladu uniformního hashování a za předpokladu, že každý klíč se bude vyhledávat se stejnou pravděpodobností.

Důkaz. Vyhledání klíče k sleduje stejnou posloupnost pokusů jako při vložení klíče k do tabulky. Jestliže klíč k byl vložen do tabulky v pořadí jako $(i+1)$-ní, očekávaný počet pokusů při vyhledání klíče k je, podle věty 7.4 nejvýše $1/(1-i/m) = m/(m-i)$. Zprůměrováním přes všech n klíčů v tabulce nám dá průměrný počet pokusů při úspěšném vyhledání:

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

kde H_i je i-té harmonické číslo (viz kapitola 2.1). Užitím nerovnosti $\ln i \leq H_i \leq \ln i + 1$ dostáváme

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

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

jako hranici očekávaného počtu pokusů při úspěšném hledání. ■

Jestliže je tedy například hashovací tabulka z poloviny plná, očekávaný počet pokusů je menší než $3,387$. Vývoj počtu pokusů v závislosti na faktoru naplnění α je zobrazen grafem 7.6.

7.2.3 Hashovací funkce

Dobře navržená hashovací funkce splňuje (přibližně) předpoklad uniformního hashování tj. každý klíč se hashuje se stejnou pravděpodobností do libovolného z m slotů. Formálně, předpokládejme, že klíče jsou vybírány z nějakého univerza U podle pravděpodobností distribuční funkce P. Potom $P(k)$ značí pravděpodobnost výběru klíče k. Předpoklad uniformního hashování lze zapsat jako

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ pro } j = 0, 1, \ldots, m-1$$

Bohužel podmínku není vždy dost dobře možné ověřit pro konkrétní situaci, protože zpravidla nebývá známa distribuční funkce P.

Někdy ovšem známa je, pak toho lze využít při konstrukci hashovací funkce. Například předpokládejme, že klíče jsou reálná čísla. Dále předpokládejme, že pravděpodobnost výskytu jednotlivých klíčů je na intervalu $\langle 0, 1\rangle$ distribuována rovnoměrně. Potom můžeme použít jednoduchou hashovací funkci

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

V praxi se hashovací funkce většinou navrhují heuristickým způsobem. Využívá se přitom částečná představa o funkci P. Představme si například, že vyvíjíme tabulku symbolů pro kompilátor. Klíče v tomto případě budou řetězce znaků reprezentující identifikátory v programu. Lze předpokládat, že identifikátory nebudou vybírány rovnoměrně z množiny všech možných identifikátorů (tj. z množiny všech n-znakových řetězců), ale budou se pravděpodobně shlukovat, jako třeba identifikátory `Item` a `Items`. Dobře navržená hashovací funkce by měla tyto dva případy rozlišit.

Rozsah hodnot hashovací funkce musí být přirozené číslo v rozsahu $0, \ldots, m-1$, kde m je velikost hashovací tabulky. Z toho plyne, že nejobvyklejší tvar hashovací funkce je

$$h(k) = f(k) \bmod m,$$

kde funkce $f(k)$ vypočítá z klíče k číselnou hodnotu.

Jak volit hodnotu m? Předně záleží na počtu hodnot, které chceme v tabulce ukládat, jestli několik set, několik tisíců. Dále záleží na technice řešení

kolizí, abychom dosáhli optimální hodnoty faktoru naplnění α, při které je potřeba minimální počet pokusů. Tím je určena přibližná hodnota m.

Volba $m = 2^p$ nám sice umožní velice jednoduše vypočítat zbytek po dělení, ale tato volba není příliš vhodné, neboť v tomto případě bereme v úvahu jen nejnižších p bitů čísla $f(k)$. Je vhodnější, aby hodnota hashovací funkce závisela na všech bitech čísla $f(k)$. Nejvhodnější se obecně jeví volit m jako prvočíslo.

Volba funkce $f(k)$ (předpokládáme, že klíč k je tvořen posloupností bytů $k = k_1 k_2 \dots k_r$):

1. Použití několika posledních nebo prostředních bytů $f(k) = k_{r-2}k_{r-1}k_r$.
2. Druhá mocnina několika prostředních bytů $f(k) = (k_{r/2-1}k_{r/2}k_{r/2+1})^2$.
3. Součet nebo součin všech bytů v klíči. $f(k) = \sum_{i=1}^{r} k_i$ nebo $f(k) = \Pi_{i=1}^{r} k_i$.
4. Polynom s koeficienty k_i $\sum_{i=1}^{r} c^{i-1}k_i$, kde c je konstanta např. $c = 3$.

V posledně dvou zmiňovaných případech se operace modulo počítá při sčítání resp. násobení členů. Protože algebraická struktura $(Z_m, +, *)$ je těleso, máme tím zaručenu neexistenci dělitelů nuly (viz kapitola 2), kteří by nepříznivě ovlivňovali výpočet hodnoty hashovací funkce.

Příklad 7.1
Ukážeme si praktickou realizaci hashovací funkce. Klíče budou představovat znakové řetězce. Pro výpočet použijeme posledně zmiňovaného schématu tj. polynom.

```
const int   m = 1009  // velikost  hashovací tabulky
int  hash(const char *s)
{
    int  i, h;
    for(i = h = 0; i < strlen(s); i++)
        h = (3*h + s[i]) % m;
}  // hash
```

Cvičení

1. Implementujte přímo adresovatelnou tabulku. Klíče v této tabulce jsou textové řetezce délky tři. Řetězce obsahují jen malá písmena anglické abecedy tj. `a ...z`.
2. Implementujte hashovací tabulku s využitím technologie separátního řetězení.

3. Implementujte hashovací tabulku s využitím otevřeného adresování. Otestujte metodu lineárních pokusů, kvadratických pokusů a dvojitého hashování.

4. Navrhněte vlastní hahsovací funkci pro textové řetězce.

Kapitola 8

Vyhledávání v textu

Vyhledávání v textu je, neformálně řečeno, operace, při které se zjišťuje, zda daný text obsahuje hledaná slova – **vzorky**. Úloha nalézt v nějakém textu **výskyty** zadaných textových vzorků patří v počítačové praxi k nejfrekventovanějším. Algoritmy, které ji řeší, se používají mimo jiné:

- v textových editorech (pohyb v editovaném textu, záměna řetězců),
- v utilitách typu *grep* (OS Unix), které umožní najít všechny výskyty zadaných vzorků v množině textových souborů (což programátor ocení např. při hledání všech modulů, které se odkazují na danou globální proměnnou),
- v rešeršních systémech (výběr anotací podle klíčových slov),
- při studiu DNA,
- při analýze obrazu, zvuku apod.

Typická velikost prohledávaných dat (např. textů) se pohybuje od jednotek kilobytů (v případě editorů) až po tisíce megabytů v případě rešeršních systémů, textových informačních systémů. V těchto případech může efektivnost vyhledávacího algoritmu velmi podstatně ovlivnit celkovou efektivitu systému.

8.1 Rozdělení vyhledávacích algoritmů

Obecně lze dělit algoritmy vyhledávání v textu podle mnoha kritérií. Všechna tato kritéria jsou nějakým způsobem vztažena ke dvěma daným skutečnostem – hledanému vzorku a prohledávanému textu.

Metoda vyžaduje		Předzpracování textu	
		ne	ano
Předzpracování vzorku	ne	I	III
	ano	II	IV

Tabulka 8.1: Klasifikace podle předzpracování

8.1.1 Předzpracování textu a vzorku

Metody vyhledávání můžeme klasifikovat podle toho, zda vyžadují **předzpracování** textu nebo předzpracování vzorku nebo obojí, do čtyř kategorií podle tabulky 8.1.

Do skupiny **I** patří elementární algoritmus (viz kapitola 8.3), který nevyžaduje ani předzpracování textu ani předzpracování vzorku.

Do skupiny **II** patří metody, nejdříve pro daný vzorek vytvoří jistá pomocná data, která se následně využijí pro vyhledávání. Přesněji řečeno, vytvoří se **vyhledávací stroj**, který potom provádí vyhledávání ([14]).

Do skupiny **III** patří **indexové metody**, které pro text, ve kterém se má vyhledávat, vytvoří index. ***Indexem*** zde rozumíme uspořádaný seznam slov s odkazy na jejich umístění v textu.

Do skupiny **IV** patří **signaturové metody**, které jak pro daný vzorek tak pro daný text vytvoří řetězce bitů – ***signatury***. Tyto signatury charakterizují jak vzorek tak i text. Vyhledávání se provádí porovnáváním signatur.

V dalším textu se budeme zabývat algoritmy, které lze řadit do kategorie I a II. Algoritmy vyhledávání patřící do kategorie III a IV spadají do oblasti *dokumentografických informačních systémů* – DIS[1]. Těmito algoritmy se zde zabývat nebudeme a zájemce odkazujeme například na skripta [17].

8.1.2 Další kritéria rozdělení

Algoritmy pro vyhledávání v textu můžeme dále dělit podle mnoha kritérií. Uveďme si aspoň některá z nich:

počet hledaných vzorků – je možné hledat jeden vzorek, konečný počet vzorků nebo nekonečný počet vzorků,

počet výskytů – můžeme hledat jen první výskyt tj. ověření existence vzorku v textu nebo nás může zajímat počet všech výskytů, případně i jejich poloha,

způsob porovnávání – možnost, kdy vzorek přesně odpovídá části textu je nejjednodušší možností. V jiných případech můžeme definovat me-

[1]Někdy se též používá označení fulltextové systémy z anglického *fulltext systems*.

triku a maximální vzdálenost do které budeme považovat vzorek a nalezené místo v textu za shodné (viz kapitola 8.2).

důležitost jednotlivých znaků ve vzorku – můžeme trvat na výskytu všech znaků ve vzorku pro nalezení výskytu nebo můžeme prohlásit některé znaky za „méně“ důležité a netrvat na jejich přítomnosti,

směr vyhledávání – text se obvykle prohledává zleva doprava. **Souměrné** vyhledávací algoritmy porovnávají znaky ve vzorku ve stejném směru tj. zleva doprava. Naopak tomu **protisměrné** algoritmy porovnávají vzorky zprava doleva čili proti směru prohledávání textu.

Z výše uvedených kritérií je patrné, že vyhledávacích úloh a jim odpovídajících algoritmů řešení, je relativně značné množství. V našem výkladu se omezíme na několik dnes již klasických algoritmů.

8.2 Definice pojmů

Definice 8.1 ***Abeceda*** Σ *je konečná neprázdná množina symbolů.*

Definice 8.2 *Konečná posloupnost symbolů ze* Σ *se nazývá* ***řetězec*** *nad* Σ*. Prázdná posloupnost se nazývá prázdný řetězec a budeme ji značit* ϵ*. Délka řetězce* x *se značí* $|x|$ *a rovná se počtu výskytů symbolů v něm obsažených.*

Definice 8.3 *Množinu všech řetězců nad abecedou* Σ *bez prázdného řetězce budeme značit* Σ^+ *a množinu všech řetězců nad abecedou* Σ *budeme značit* Σ^**.*

Definice 8.4 *Řetězec* u *se nazývá* ***předponou*** *(prefixem) řetězce* w*, jestliže existuje řetězec* v *(i prázdný) takový, že* $w = uv$*.*

Definice 8.5 *Řetězec* v *se nazývá* ***příponou*** *(sufixem) řetězce* w*, jestliže existuje řetězec* u *(i prázdný) takový, že* $w = uv$*.*

Definice 8.6 *Řetězec* y *se nazývá* ***podřetězcem*** *(faktorem) řetězce* w*, jestliže existují řetězce* u *a* v *(i prázdné) tak, že* $w = uzv$*.*

Definice 8.7 *Číslo* p *se nazývá* ***perioda*** *řetězce* w*, jestliže platí*

$$p = min\{q : 0 \leq i < |w| - q, w[i] = w[i + q]\}$$

Řetězec w *se nazývá* ***periodický****, jestliže délka jeho periody je menší nebo rovna* $|w|/2$*. V opačném případě se řetězec nazývá neperiodický.*

Definice 8.8 *Řetězec* z *se nazývá* ***hranicí*** *řetězce* w*, jestliže existují dva řetězce* u *a* v *takové, že* $w = uz = zv$*.* z *je současně prefixem i sufixem* w*.*

8.2.1 Označení

V dalším textu budeme používat následující označení:

- x hledaný vzorek, $x = x_0x_1 \dots x_{m-1}$, kde m je délka vzorku,
- y prohledávaný text, $y = y_0y_1 \dots y_{n-1}$, kde n je délka vzorku,
- Σ – abeceda z níž je sestaven vzorek i text,
- σ – velikost abecedy Σ ($\sigma = |\Sigma|$),
- $\bar{C}_n$ – očekávaný počet porovnání potřebných k vyhledání vzorku v textu délky n.

Označení v implementaci

V ukázkách kódu budeme předpokládat tyto definice:

```
// velikost abecedy
const int AlphabetSize = 256;

// delka slova procesoru
const int WordSize = 8*sizeof(int);
```

8.3 Elementární algoritmus

Hlavní rysy

- bez předzpracování,
- konstantní paměťová složitost složitost,
- posun vzorku vždy o jednu pozici,
- porovnávání vzorku lze provádět v libovolném směru,
- Časová složitost $O(mn)$,
- očekávaná složitost $O(k_L n)$

Popis

Algoritmus **elementární** hledá vzorek jen pomocí „hrubé síly“, proto se mu také někdy říká vyhledávání hrubou silou, což odpovídá anglickému názvu *Brute force searching*. Hrubou silou je myšlen postup, kdy se vzorek postupně přikládá na všechny možné pozice v textu a testuje se, jestli nedošlo ke shodě. Prohledávaný text a zadaný vzorek se nijak nepředzpracovává.

Nevýhodou algoritmu je velká časová složitost v průměrném případě. Časová složitost tohoto triviálního algoritmu je $O(mn)$. Příkladem může

být vyhledání vzorku $a^m b$ v textu $a^n b$. Očekávaný počet porovnání $\bar{C}_N$ pro elementární algoritmus je podle [**?**]

$$\bar{C}_n = \frac{\sigma}{\sigma - 1}\left(1 - \frac{1}{\sigma^m}\right)(n - m - 1) + O(1) \tag{8.1}$$

kde $n \geq m$, což je zásadní rozdíl oproti nejhoršímu případu nm.

V přirozených jazycích nedochází ke shodě počátků slov příliš často, a proto je průměrná asymptotická složitost algoritmu redukována na $O(k_L n)$, kde k_L je konstanta závislá na jazyku. Pro angličtinu byla její hodnota stanovena experimentálně na $1,07$ (viz [14]), algoritmus se tedy prakticky chová jako lineární.

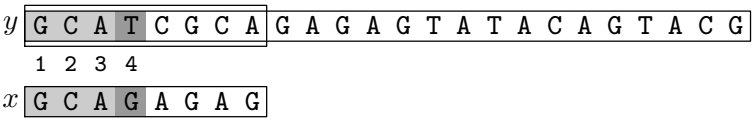
Jiné algoritmy se snaží docílit lepší asymptotické složitosti (v průměrném i nejhorším případě) než elementární algoritmus předzpracováním vzorku nebo textu samotného. Během předzpracování je prozkoumána struktura vzorku (textu) a na jejím základě vytvořen vyhledávací algoritmus (vyhledávací stroj), který již pracuje lineárním čase vzhledem k délce textu n.

Implementace

```
void BruteForce(const char *x, const int m, const char *y, const int n)
{
    int i, j;

    for(i = 0; i <= n - m; i++)
    {
        for(j = 0; j < m; j++)
            if (x[j] != y[j+i])
                break;
            if (j == m)
              printf ("Vzorek nalezen na pozici %d\n", i);
    }
} // BruteForce
```

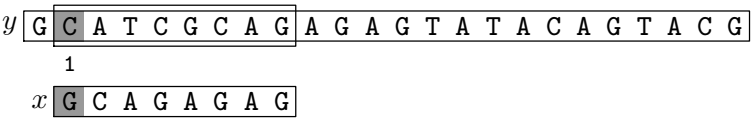
Příklad

První pokus:



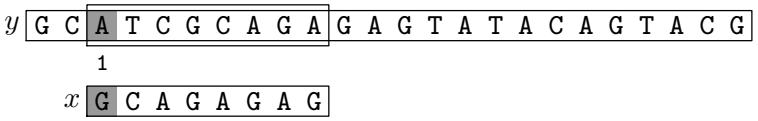
Posun o 1

Druhý pokus:



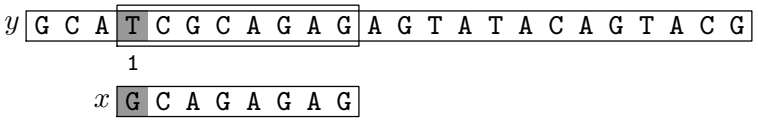
Posun o 1

Třetí pokus:



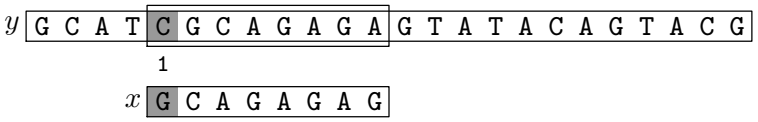
Posun o 1

Čtvrtý pokus:



Posun o 1

Pátý pokus:



Posun o 1

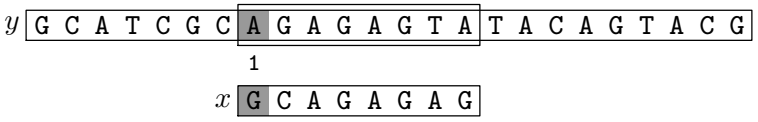
Šestý pokus:

Posun o 1

Sedmý pokus:

Posun o 1

Osmý pokus:



Posun o 1

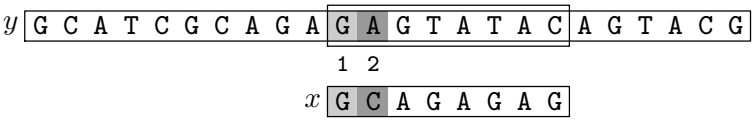
Devátý pokus:

Posun o 1

Desátý pokus:

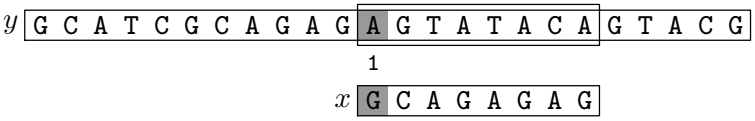
Posun o 1

Jedenáctý pokus:



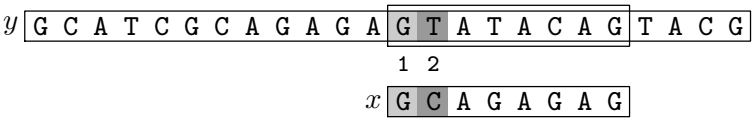
Posun o 1

Dvanáctý pokus:



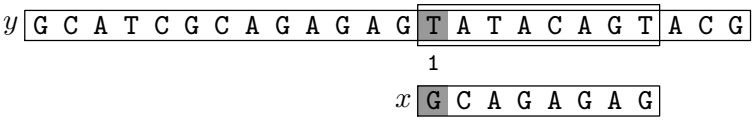
Posun o 1

Třináctý pokus:



Posun o 1

Čtrnáctý pokus:

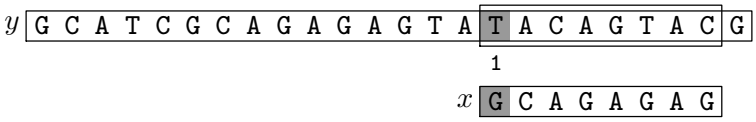


Posun o 1

Patnáctý pokus:

Posun o 1

Šesnáctý pokus:



Posun o 1

Sedmnáctý pokus:

Posun o 1

Algoritmus provedl celkem 30 porovnání znaků.

8.4 Morris-Prattův algoritmus

Hlavní rysy

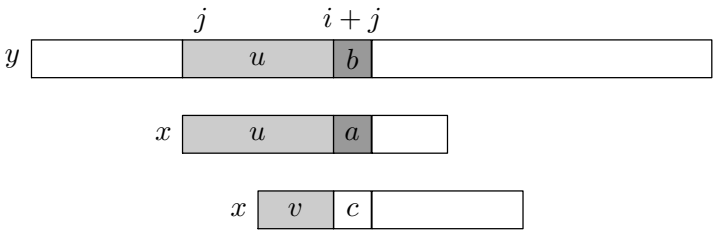
- hledání zleva doprava,
- časová a paměťovou složitost předzpracování vzorku $O(m)$,
- hledání s časovou složitostí $O(m+n)$, nezávisle na velikosti abecedy,
- nejvíce $2n-1$ porovnání znaků v průběhu hledání,
- jeden znak porovnáván nejvíce m-krát.

Popis

Návrh **Morris-Prattova** algoritmu [15] vychází z přesné analýzy činnosti elementárního algoritmu (viz kapitola 8.3), především se snaží analyzovat ztráty informace shromažďované v průběhu hledání v textu.

Pokud se podíváme podrobněji na elementární algoritmus, je možné zvětšit délku posunu vzorku po textu a zároveň si zapamatovat která část textu odpovídala částečně vzorku. Tyto informace nám ušetří porovnávání znaků, čímž se následně urychlí celý algoritmus.

Předpokládejme, že hledáme vzorek v textu na pozici j, tzn. testujeme podřetězec $y[j \ldots j+m-1]$. Předpokládejme dále, že první neshoda nastane



Obrázek 8.1: Posun v Morris-Prattově algoritmu: v je hranicí u.

mezi znaky $x[i]$ a $y[i+j]$, pro $0 < i < m$. Potom se části textu a vzorku rovnají, $x[0 \ldots i-1] = y[j \ldots i+j-1] = u$, a $a \neq b$, kde $a = x[i]$, $b = y[i+j]$. Když posuneme vzorek, lze očekávat, že jistá předpona v vzorku x bude odpovídat jisté příponě (sufixem) části textu u. Nejdelší takovou předponu nazveme hranicí řetězce u (vyskytuje se na obou koncích u). Můžeme vytvořit tabulku $Next$, kde hodnota $Next[i]$ bude označovat délku nejdelší možné hranice řetězce $x[0 \ldots i-1]$, pro všechna $i = 1, 2, \ldots, m$. Proto po posunu porovnávání znaků může pokračovat mezi znaky $c = x[Next[i]]$ a $y[i+j] = b$ aniž bychom se museli obávat ztráty některého výskytu vzorku x v textu y. Zároveň tím zamezíme opakovanému porovnávání již prozkoumaných částí (viz obrázek 8.1). Hodnota $Next[0]$ je nastavena na -1. Tabulka $Next$ má velikost m položek. Její výpočet je možné provést s časovou i prostorovou složitostí $O(m)$. Výpočet probíhá aplikací vyhledávacího algoritmu na vzorek sám, jako kdyby $y = x$.

Fáze vyhledávání pak má časovou složitost $O(m+n)$. Morris-Prattův algoritmus provede během vyhledávání nejvýše $2n-1$ porovnání znaků. Jeden znak je testován nejvýše m-krát.

Implementace

```
void PreprocessMorrisPratt(const char *x, const int m, int Next[])
{
    int i, j;

    i = 0;
    j = Next[0] = -1;
    while (i < m)
    {
        while (j > -1 && x[i] != x[j])
            j = Next[j];
        Next[++i] = ++j;
    }
}

void MorrisPratt(const char *x, const int m, const char *y, const int n)
{
    int i, j;
```

```
    int* Next = new int[m+1];

// Predzpracovani
    PreprocessMorrisPratt(x, m, Next);

// Vyhledavani
    i = j = 0;
    while (j < n)
    {
        while (i > -1 && x[i] != y[j])
            i = Next[i];
        i++;
        j++;
        if (i >= m)
        {
            printf("Vzorek nalezen na pozici %d\n", j - i);
            i = Next[i];
        }
    }
    delete [] Next;
} // MorrisPratt
```

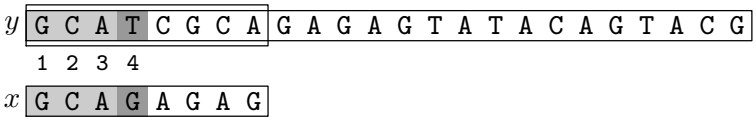
Příklad

Fáze předzpracování

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$Next[i]$	-1	0	0	0	1	0	1	0	1

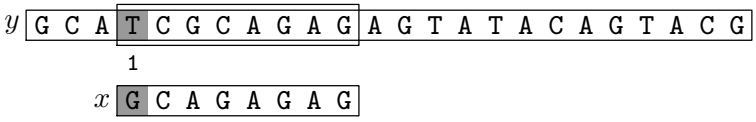
Fáze vyhledávání

První pokus:



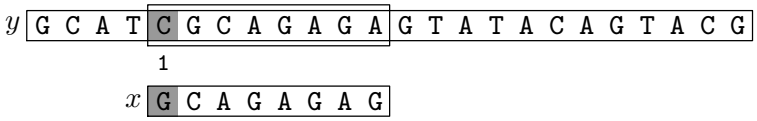
Posun o 3 ($i - Next[i] = 3 - 0$)

Druhý pokus:



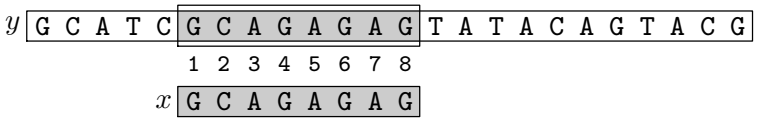
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Třetí pokus:



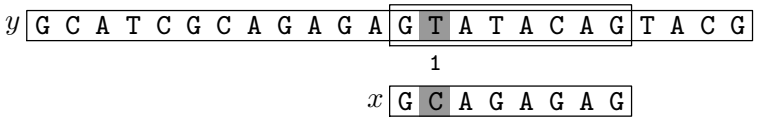
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Čtvrtý pokus:



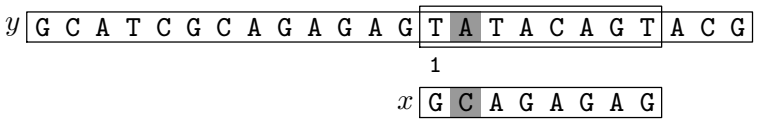
Posun o 7 ($i - Next[i] = 8 - 1$)

Pátý pokus:



Posun o 1 ($i - Next[i] = 1 - 0$)

Šestý pokus:

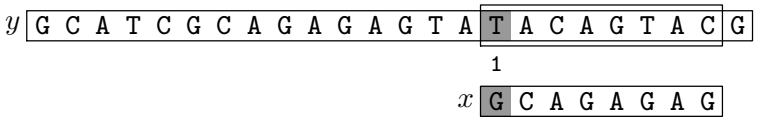


Posun o 1 ($i - Next[i] = 0 - (-1)$)

Sedmý pokus:

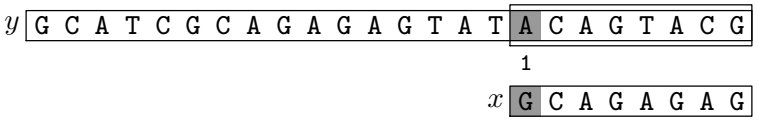
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Osmý pokus:



Posun o 1 ($i - Next[i] = 0 - (-1)$)

Devátý pokus:



Posun o 1 ($i - Next[i] = 0 - (-1)$)

Morris-Prattův algoritmus provedl celkem 19 porovnání znaků.

8.5 Knuth-Morris-Prattův algoritmus

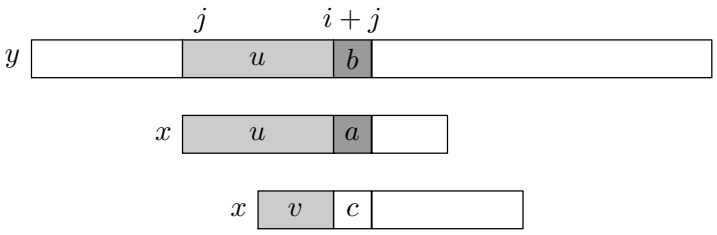
Hlavní rysy

- hledání zleva doprava,
- časová a paměťová složitost předzpracování vzorku $O(m)$,
- hledání s časovou složitostí $O(m + n)$, nezávisle na velikosti abecedy,
- nejvíce $2n - 1$ porovnání znaků v průběhu hledání,
- počet porovnání jednoho znaku shora ohraničeno $\log_\Phi(m)$, kde Φ je zlatý řez, $\Phi = \frac{1+\sqrt{5}}{2}$ (viz kapitola **??**).

Popis

Knuth-Morris-Prattův algoritmus [12] vychází z analýzy Morris-Prattova algoritmu (viz kapitola 8.4). Zlepšení spočívá v prodloužení délek posunů.

Předpokládejme, že hledáme vzorek v textu na pozici j, tzn. testujeme podřetězec $y[j \dots j+m-1]$. Předpokládejme dále, že první neshoda nastane mezi znaky $x[i]$ a $y[i + j]$, pro $0 < i < m$. Potom se části textu a vzorku rovnají, $x[0 \dots i-1] = y[j \dots i+j-1] = u$, a $a \neq b$, kde $a = x[i]$, $b = y[i+j]$. Když posuneme vzorek, lze očekávat, že jistá předpona v vzorku x bude odpovídat jisté příponě části textu u. Navíc pokud chceme zabránit okamžité neshodě, znak který následuje po předponě v ve vzorku musí být různý od a. Vytvoříme tabulku $Next$, kde hodnota $Next[i]$ bude označovat délku nejdelší možné značené hranice řetězec $x[0 \dots i-1]$ nebo -1 pokud taková značená hranice neexistuje, pro všechna $i = 1, 2, \dots, m$. Proto po posunu porovnávání znaků může pokračovat mezi znaky $c = x[Next[i]]$ a $y[i + j] = b$ aniž bychom se museli obávat ztráty některého výskytu vzorku x v textu y. Zároveň tím zamezíme opakovanému porovnávání již prozkoumaných částí (viz obrázek 8.2). Hodnota $Next[0]$ je nastavena na -1. Tabulka $Next$ má velikost m položek. Její výpočet je možné provést s časovou i prostorovou složitostí $O(m)$. Výpočet probíhá aplikací vyhledávacího algoritmu na vzorek sám, jako kdyby $y = x$.



Obrázek 8.2: Posun v Knuth-Morris-Prattově algoritmu: v je hranicí u a zároveň $a \neq c$.

Fáze vyhledávání pak má časovou složitost $O(m + n)$. Knuth-Morris-Prattův algoritmus provede během vyhledávání nejvýše $2n - 1$ porovnání znaků. Jeden znak je testován nejvýše $\log_\Phi(m)$-krát.

Implementace

```
void PreprocessKMP(const char *x, const int m, int Next[])
{
    int i, j;

    i = 0;
    j = Next[0] = -1;
    while (i < m)
    {
        while (j > -1 && x[i] != x[j])
            j = Next[j];
        i++;
        j++;
        if (x[i] == x[j])
            Next[i] = Next[j];
        else
            Next[i] = j;
    }
}

void KMP(const char *x, const int m, const char *y, const int n)
{
    int i, j;
    int* Next = new int[m+1];

// Predzpracovani
    PreprocessKMP(x, m, Next);

// Vyhledavani
    i = j = 0;
    while (j < n)
    {
        while (i > -1 && x[i] != y[j])
            i = Next[i];
```

```
            i++;
            j++;
            if (i >= m)
            {
                printf ("Vzorek nalezen na pozici %d\n", j - i);
                i = Next[i];
            }
        }
        delete [] Next;
    }
```

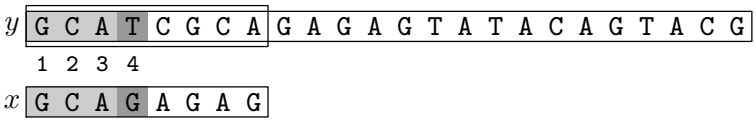
Příklad

Fáze předzpracování

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$Next[i]$	-1	0	0	-1	1	-1	1	-1	1

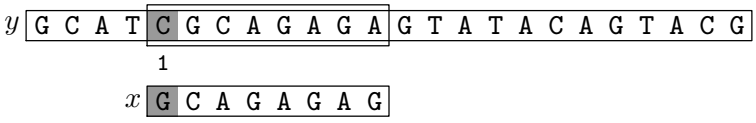
Fáze vyhledávání

První pokus:



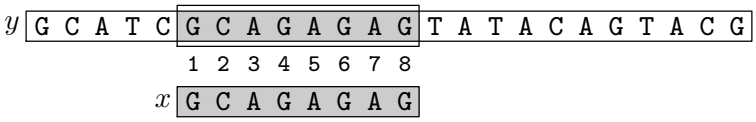
Posun o 4 ($i - Next[i] = 3 - (-1)$)

Druhý pokus:



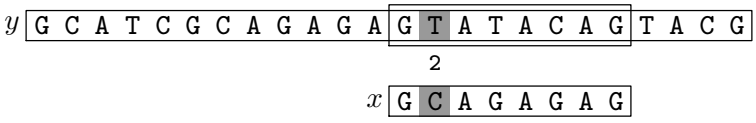
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Třetí pokus:



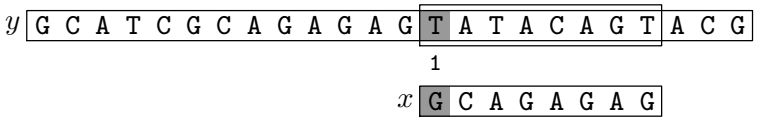
Posun o 7 ($i - Next[i] = 8 - 1$)

Čtvrtý pokus:



Posun o 1 ($i - Next[i] = 1 - 0$)

Pátý pokus:

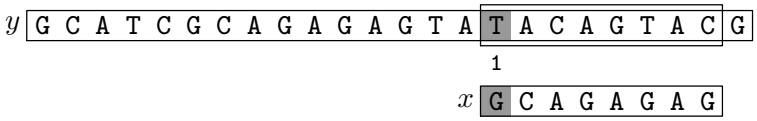


Posun o 1 ($i - Next[i] = 0 - (-1)$)

Šestý pokus:

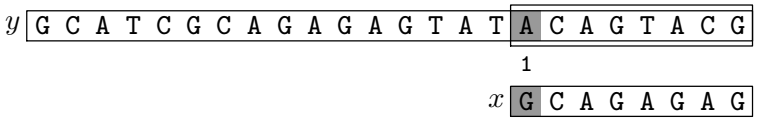
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Sedmý pokus:



Posun o 1 ($i - Next[i] = 0 - (-1)$)

Osmý pokus:



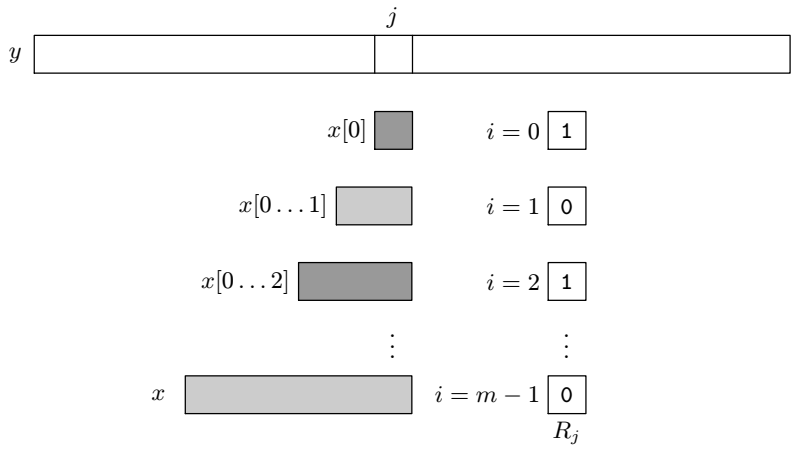
Posun o 1 ($i - Next[i] = 0 - (-1)$)

Knuth-Morris-Prattův algoritmus provedl celkem 18 porovnání znaků.

8.6 Shift-Or algoritmus

Hlavní rysy

- používá bitové operace,
- efektivní, pokud délka vzorku nepřesahuje délku slova procesoru (typicky 16, 32 nebo 64 bitů),
- předzpracování s časovou a paměťovou složitostí $O(m + \sigma)$,
- hledání se složitostí $O(n)$ nezávisle na velikosti abecedy a délce vzorku,
- lze jej snadno přizpůsobit pro přibližné vyhledávání řetězců.



Obrázek 8.3: Význam vektorů R_j v Shift-Or algoritmu

Popis

Shift-Or algoritmus [3, 9] je založen na bitových operacích. Nechť R je pole bitů délky m, potom vektor R_j bude označovat hodnoty pole R po zpracování znaku $y[j]$ (viz obrázek 8.3). Tento vektor obsahuje informace o všech shodách prefixů vzorku x, které *končí* na pozici j. Pro $0 \leq i \leq m-1$ platí

$$R_j[i] = \begin{cases} 0 & \text{jestliže } x[0 \ldots i] = y[j-i \ldots j] \\ 1 & \text{jinak.} \end{cases}$$

Vektor R_{j+1} lze spočítat na základě vektoru R_j následujícím způsobem. Pro všechna $R_j[i]$ která jsou nulová se vypočte

$$R_{j+1}[i+1] = \begin{cases} 0 & \text{jestliže } x[i+1] = y[j+1] \\ 1 & \text{jinak,} \end{cases}$$

a také

$$R_{j+1}[0] = \begin{cases} 0 & \text{jestliže } x[0] = y[j+1] \\ 1 & \text{jinak.} \end{cases}$$

Jestliže se $R_{j+1}[m-1]$ potom byl vzorek úspěšně nalezen.

Přechod od vektoru R_j k vektoru R_{j+1} se dá spočítat snadno a rychle tímto způsobem. Pro všechna $a \in \Sigma$ nechť S_a je pole bitů délky m takové, že pro všechna $0 \leq i \leq m-1$ platí

$$S_a[i] = \begin{cases} 0 & \text{právě když } x[i] = a \\ 1 & \text{jinak.} \end{cases}$$

Pole S_a určuje pozice znaku a ve vzorku x. Hodnoty ve všech polích S_a se počítají během předzpracování vzorku. Výpočet vektoru R_{j+1} se potom

redukuje na dvě operace – bitový posuv a bitový součet:

$$R_{j+1} = Shift(R_j)\ Or\ S_{y[j+1]}$$

Za předpokladu, že vzorek není delší než délka slova procesoru, časová a paměťová složitost předzpracování je $O(m + \sigma)$. Časová složitost hledání vzorku v textu je $O(n)$ nezávisle na velikosti abecedy a délce vzorku.

Implementace

```
unsigned int PreprocessShiftOr(const char *x, const int m, unsigned int S[])
{
    unsigned int j, lim;
    int i;

    for (i = 0; i < AlphabetSize; ++i)
        S[i] = ~0u;
    for (lim = i = 0, j = 1; i < m; ++i, j <<= 1)
    {
        S[x[i]] &= ~j;
        lim |= j;
    }
    lim = ~(lim>>1);
    return lim ;
}


void ShiftOr(const char *x, const int m, const char *y, const int n)
{
    unsigned int lim, state;
    unsigned int S[AlphabetSize];
    int j;

    if (m > WordSize)
    {
        printf("Vzorek je delsi nez delka slova procesoru!\n");
        return;
    }

// Predzpracovani
    lim = PreprocessShiftOr(x, m, S);

// Vyhledavani
    for (state = ~0u, j = 0; j < n; ++j)
    {
        state = (state << 1) | S[y[j]];
        if (state < lim)
            printf("Vzorek nalezen na pozici %d\n", j - m + 1);
    }
}
```

Příklad

	$S_{\mathtt{A}}$	$S_{\mathtt{C}}$	$S_{\mathtt{G}}$	$S_{\mathtt{T}}$
G	1	1	0	1
C	1	0	1	1
A	0	1	1	1
G	1	1	0	1
A	0	1	1	1
G	1	1	0	1
A	0	1	1	1
G	1	1	0	1

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
0	G	0	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0
1	C	1	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	A	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	G	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	A	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	G	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	A	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
7	G	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

Hodnota $R_{12}[7] = 0$ znamená, že vzorek x byl úspěšně nalezen na pozici $12 - 8 + 1 = 5$.

8.7 Karp-Rabinův algoritmus

Hlavní rysy

- využívá hashovací funkce,
- předzpracování s časovou složitostí $O(m)$,
- paměťovou složitost konstantní,
- hledání s časovou složitostí $O(mn)$,
- očekávaná časová hledání $O(m + n)$.

Popis

Hashování nabízí jednoduchou možnost jak se ve většině případů vyhnout kvadratické složitosti vyhledávání. Místo toho abychom pro každou možnou pozici v textu zkoumali, jestli se zde vzorek vyskytuje nebo ne, by bylo lepší zkoumat jen ty pozice které „vypadají jako“ vzorek. K vyjádření podobnosti mezi vzorkem a částí textu (shodné délky jako vzorek) použijeme hashovací

funkci. V textu budeme hledat pouze ty úseky, které mají shodnou hashovací hodnotu jako hledaný vzorek. Algoritmus si můžeme představit tak, že máme pomyslnou hashovací tabulku do které vkládáme části textu. Ty části které se hashovaly do jiného slotu než vzorek nemusíme vůbec zkoumat. Zkoumat musíme jen ty části, které se hashovaly do téhož slotu jako vzorek tj. sledujeme jen kolidující části textu. Pokud objevíme kolidující část textu nezbývá nic jiného než tuto část znak po znaku porovnat se vzorkem. Jak bylo řečeno v kapitole o hashování, dobře navržená hashovací funkce by měla počet kolizí minimalizovat. Shrňme si vlastnosti hashovací funkce vhodné pro vyhledávání řetezců:

- efektivně vypočitatelná,
- citilivá na změny v řetězci,
- hashovací hodnota řetězce $hash(y[j+1 \dots j+m])$ musí být lehce vypočitatelná z hodnoty $hash(y[j \dots j+m-1])$ a znaků $y[j])$, $y[j+m])$. To znamená, že pro m znakový řetězec na pozici j který posuneme o jeden znak doprava musí být možné lehce spočítat hashovací hodnotu posunutého řetězce na základě předešlé hashovací hodnoty, znaku který zprava přibyl a znaku, který zleva ubyl. Formálně zapsáno:

  $$hash(y[j+1 \dots j+m]) = rehash(y[j], y[j+m], hash(y[j \dots j+m-1]))$$

Pro řetězec w délky m můžeme definovat například tuto hashovací funkci:

$$hash(w[0 \dots m-1] = (w[0] \times 2^{m-1} + w[1] \times 2^{m-2} + \cdots + w[m-1] \times 2^0) \bmod q$$

kde q je velké přirozené číslo. Potom pro tuto hashovací funkci lze definovat funkci $rehash$:

$$rehash(a, b, h) = ((h - a \times 2^{m-1}) \times 2 + b) \bmod q \ .$$

Předzpracování vzorku v **Karp-Rabinově** algoritmu [10] znamená výpočet hashovací hodnoty vzorku x. Paměťová složitost tohoto výpočtu je konstantní, časová $O(m)$.

Během hledání vzorku stačí porovnat jeho hashovací hodnotu s hashovacími hodnotami $hash(y[j \dots j+m-1])$, pro $j = 0, 1, \dots, n-m$. V případě rovnosti je nutné otestovat rovnost $x = y[j \dots j+m-1]]$.

Časová složitost vyhledávací fáze je $O(mn)$ (Například při hledání a^m v textu a^n). Očekávaný počet porovnání znaků je úměrný $O(m+n)$.

Implementace

Implementace Karp-Rabinova algoritmu se liší od popisu několika detaily:

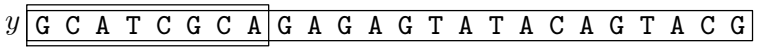
1. násobení 2 je realizováno jako bitový posun doleva,
2. číslo q bylo zvoleno jako $2^{32} - 1$ tj. největší 32-bitové číslo bez znaménka. Tím odpadá výpočet zbytku po dělení; zbytek je počítán automaticky zanedbáváním přenosu nejvyššího bitu.

```
inline unsigned int Rehash(const unsigned int a, const unsigned int b,
                           const unsigned int h, const unsigned int d)
{
    unsigned int iRet;
    iRet = ((h - (a << d)) << 1) + b;
    return iRet;
}


void KarpRabin(const char *x, const int m, const char *y, const int n)
{
    unsigned int hx, hy;
    int i, j;

    hx = hy = 0;
    for(i = 0; i < m; i++)
    {
        hx = (hx << 1) + x[i];
        hy = (hy << 1) + y[i];
    }

    i = 0;
    while (i <= n-m)
    {
        if (hx == hy)
        {// potencialni nalez vzorku
            for(j = 0; j < m; j++)
                if (x[j] != y[i+j])
                    break;
            if (j == m)
                printf ("Vzorek nalezen na pozici %d\n", i);
        }
        hy = Rehash(y[i], y[i+m], hy, m-1);
        i++;
    }
} // KarpRabin
```

Příklad

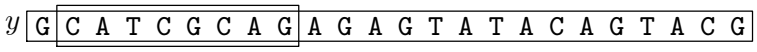
Hashovací hodnota vzorku: $hash(\texttt{GCAGAGAG}) = 17597$.

První pokus:



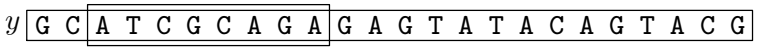
$hash(y[0 \dots 7]) = 17819$

Druhý pokus:



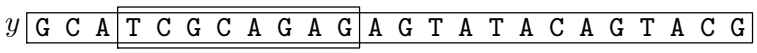
$hash(y[1 \dots 8]) = 17533$

Třetí pokus:



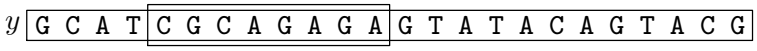
$hash(y[2 \dots 9]) = 17979$

Čtvrtý pokus:



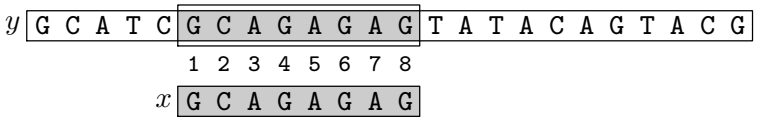
$hash(y[3 \dots 10]) = 19389$

Pátý pokus:



$hash(y[4 \dots 11]) = 17339$

Šestý pokus:



$hash(y[5 \dots 12]) = 17597 = hash(x)$

Protože se hashovací hodnota vzorku rovná hashovací hodnotě části textu je nutné pomocí n porovnání (v našem případě osmi) zjistit, zda se jedná skutečně o námi hledaný vzorek.

Sedmý pokus:

$hash(y[6 \ldots 13]) = 17102$

Osmý pokus:

$hash(y[7 \ldots 14]) = 17117$

Devátý pokus:

$hash(y[8 \ldots 15]) = 17678$

Desátý pokus:

$hash(y[9 \ldots 16]) = 17245$

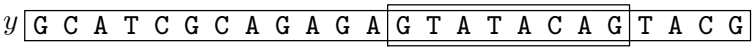
Jedenáctý pokus:

$hash(y[10 \ldots 17]) = 17917$

Dvanáctý pokus:

$hash(y[11 \ldots 18]) = 17723$

Třináctý pokus:

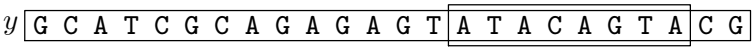


$hash(y[12 \ldots 19]) = 18877$

Čtrnáctý pokus:

$hash(y[13 \ldots 20]) = 19662$

Patnáctý pokus:

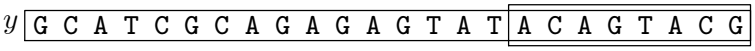


$hash(y[14 \ldots 21]) = 17885$

Šesnáctý pokus:

$hash(y[15 \ldots 22]) = 19197$

Sedmnáctý pokus:



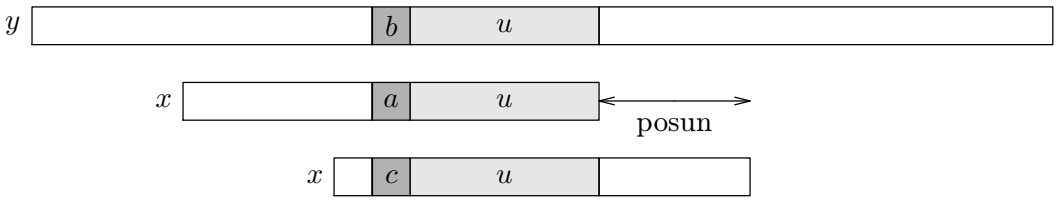
$hash(y[16 \ldots 23]) = 16961$

Karp-Rabinův algoritmus provedl 17 porovnání hashovacích hodnot a 8 porovnání znaků.

8.8 Boyer-Mooreův algoritmus

Hlavní rysy

- vzorek je porovnáván zprava doleva, tj. protisměrně,
- časová a paměťová složitost předzpracování vzorku $O(m + \sigma)$,
- časová složitost vyhledávání $O(mn)$,



Obrázek 8.4: Posun při nalezení vhodné přípony. u se v x vyskytuje celá znovu předcházena znakem c různým od a.

- v nejhorším případě $3n$ porovnání znaků, platí pro neperiodický vzorek,
- v nejlepším případě časová složitost vyhledávání $O(n/m)$.

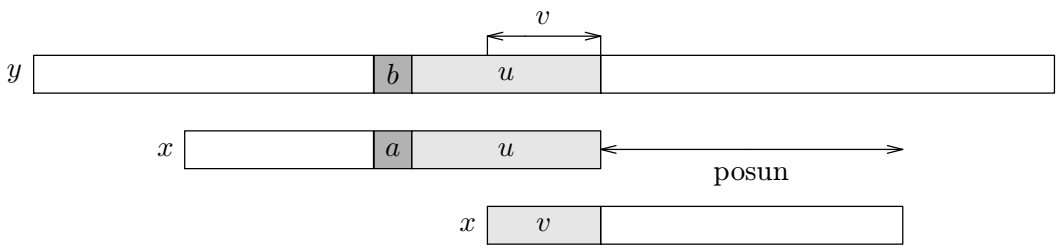
Popis

Boyer-Mooreův algoritmus [6] je považován za jeden z nejefektivnějším vyhledávacích algoritmů pro běžné aplikace. Zjednodušená verze tohoto algoritmu je základem implementace funkcí „najdi“ a „nahraď“ v textových editorech. Boyer-Mooreův algoritmus je představitelem proti směrných vyhledávacích algoritmů. To znamená, že prochází znaky ve vzorku zprava doleva.

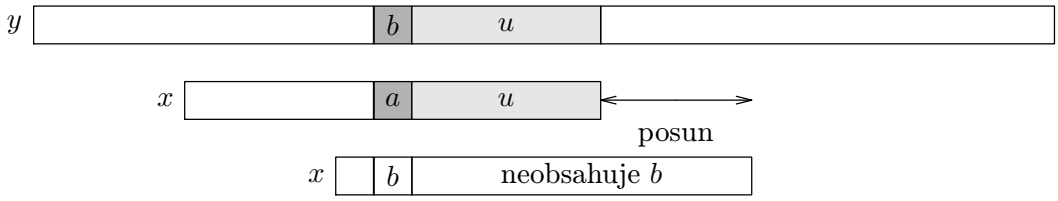
Algoritmus při hledání využívá dvě funkce (ve formě tabulek), které jsou vypočteny na základě vzorku během fáze předzpracování. Ve fázi vyhledávání je vzorek pomocí těchto funkcí posouván po textu doprava ať už v případě neshody znaku nebo shody celého vzorku. Jedna z funkcí posunuje vzorkem při nalezení vhodné přípony (angl. *good-suffix shift*), druhá při neshodě znaku ve vzorku a v textu (angl. *bad-character shift*). První funkci budeme značit Gs, druhou Bc.

Předpokládejme, že došlo k neshodě mezi znakem vzorku $x[i] = a$ a znakem textu $y[i + j] = b$ při testování pozice j. Potom se část vzorku $x[i + 1 \ldots m - 1]$ a část textu $y[i + j + 1 \ldots j + m - 1]$ shodují (označme ji u) a platí, že $x[i] \neq y[i + j]$. Úloha funkce Gs spočívá v posunutí části u tak aby byla zarovnána s jejím nejpravějším výskytem ve vzorku x. A navíc tomuto výskytu musí předcházet znak odlišný od $x[i]$ (viz obrázek 8.4). Jestliže taková část neexistuje, posuneme vzorek tak, aby se nejdelší přípona (sufix) části textu $v = y[i + j + 1 \ldots j + m - 1]$ shodovala s předponou (prefixem) vzorku x (viz obrázek 8.5).

Posun při neshodě znaku spočívá v zarovnání znaku $y[i + j]$ s jeho nejpravějším výskytem v části vzorku $x[0 \ldots m - 2]$ (viz obr. 8.6). Jestliže se znak $y[i+j]$ ve vzorku x nevyskytuje, potom žádný výskyt vzorku x nemůže



Obrázek 8.5: Posun při nalezení vhodné přípony. V x se znovu vyskytuje jen část u.



Obrázek 8.6: Posun při neshodě znaku. Znak a se vyskytuje v x.

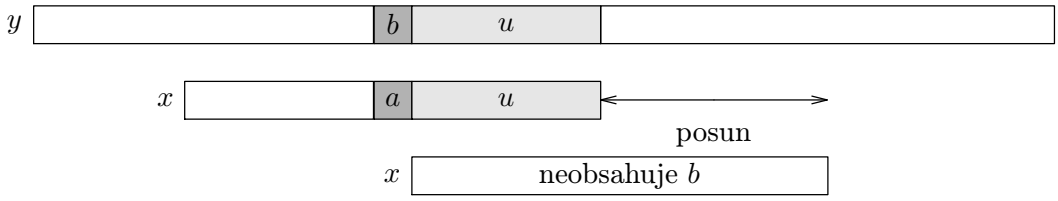
tuto pozici zahrnovat a můžeme celý vzorek posunout tak, aby začínal na znaku bezprostředně po $y[i+j]$ tj. $y[i+j+1]$ (viz obrázek 8.7).

Poznamenejme, že posun při neshodě znaku může být záporný, proto Boyer-Mooreův algoritmus bere při posunu maximum z hodnot funkcí Gs a Bc. Formálně můžeme obě funkce definovat následujícím způsobem.

Výpočet funkce Gs

Definujme nejdříve dvě podmínky:

$$Cs(i,s) \quad : \quad \forall k \text{ taková, že } i < k < m, (s \geq k) \vee (x[k-s] = x[k])$$



Obrázek 8.7: Posun při neshodě znaku. Znak a se nevyskytuje v x.

$$Co(i,s) \quad : \quad (s < i) \Rightarrow (x[i-s] \neq x[i]) \ .$$

Potom pro všechna $0 < i < m$:

$$Gs(i) = min\{s > 0 : \text{platí } Cs(i,s) \wedge Co(i,s)\}$$

$Gs(0)$ definujeme jako délku periody (viz definici 8.7) vzorku x. K výpočtu tabulky Gs se používá funkce $suff$ definovaná jako:

$$\forall i, 1 \leq i < m, suff[i] = max\{k : x[i-k+1 \ldots i] = x[m-k \ldots m-1]\}$$

Výpočet funkce Bc

Funkci Bc lze definovat předpisem $\forall c \in \Sigma$:

$$Bc(c) = \begin{cases} min\{i : (1 \leq i < m-1) \wedge (x[m-1-i] = c)\} & \text{jestliže } c \in x \\ m & \text{jinak} \end{cases}$$

Výpočet funkcí Bc a Gs probíhá v rámci předzpracování vzorku. Jeho časová a prostorová složitost je $O(m + \sigma)$. Složitost vyhledávání je obecně kvadratická, přičemž je porovnáno nejvýše $3n$ znaků při vyhledávání neperiodického vzorku. Pro rozsáhlé abecedy (vzhledem k délce vzorku) je algoritmus velice rychlý. Při hledání vzorku $a^{m-1}b$ v textu a^n algoritmus porovná jen $O(n/m)$ znaků, což je absolutní minimum pro vyhledávací algoritmy s předzpracováním vzorku – algoritmy skupiny II.

Implementace

```
void PreprocessBc(const char *x, const int m, int Bc[])
{
    int i;

    for (i = 0; i < AlphabetSize; ++i)
        Bc[i] = m;
    for (i = 0; i < m - 1; ++i)
        Bc[x[i]] = m - i - 1;
}


void Suffixes(const char *x, const int m, int suff[])
{
    int f, g, i;

    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i)
    {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else
```

```
        {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && x[g] == x[g + m - 1 - f])
                --g;
            suff[i] = f - g;
        }
    }
}


void PreprocessGs(const char *x, const int m, int Gs[])
{
    int i, j;
    int* suff = new int[m];

    Suffixes(x, m, suff);
    for (i = 0; i < m; ++i)
        Gs[i] = m;
    j = 0;
    for (i = m - 1; i >= -1; --i)
        if (i == -1 || suff[i] == i + 1)
            for (; j < m - 1 - i; ++j)
                if (Gs[j] == m)
                    Gs[j] = m - 1 - i;
    for (i = 0; i <= m - 2; ++i)
        Gs[m - 1 - suff[i]] = m - 1 - i;
    delete [] suff;
}


void BoyerMoore(const char *x, const int m, const char *y, const int n)
{
    int i, j;
    int Bc[AlphabetSize];
    int* Gs = new int[m+1];

// Predzpracovani
    PreprocessGs(x, m, Gs);
    PreprocessBc(x, m, Bc);

// Vyhledavani
    j = 0;
    while (j <= n - m)
    {
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
        if (i < 0)
        {
            printf("Vzorek nalezen na pozici %d\n", j);
            j += Gs[0];
        }
        else
            if (Gs[i] > (Bc[y[i + j]] - m + 1 + i))
```

```
                j += Gs[i];
            else
                j += Bc[y[i + j]] - m + 1 + i;
        }
        delete [] Gs;
    }
```

Příklad

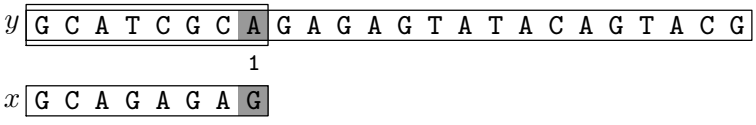
Fáze předzpracování

c	A	C	G	T
$Bc[c]$	1	6	2	8

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$suff[i]$	1	0	0	2	0	4	0	8
$Gs[i]$	7	7	7	2	7	4	7	1

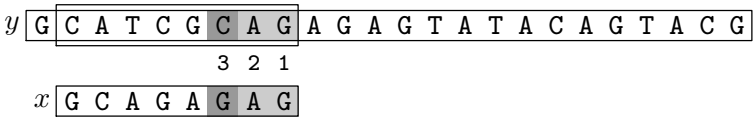
Fáze vyhledávání

První pokus:



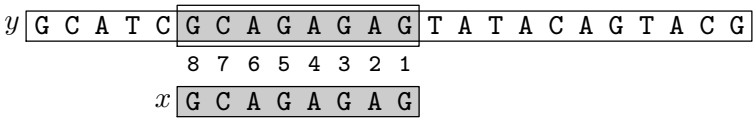
Posun o 1 ($Gs[7] = Bc[\texttt{A}] - 7 + 7$)

Druhý pokus:



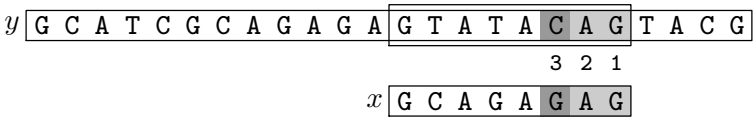
Posun o 4 ($Gs[5] = Bc[\texttt{C}] - 7 + 5$)

Třetí pokus:



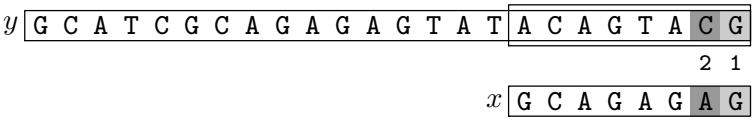
Posun o 7 ($Gs[0]$)

Čtvrtý pokus:



Posun o 4 ($Gs[5] = Bc[\texttt{C}] - 7 + 5$)

Pátý pokus:



Posun o 7 ($Gs[6]$)

Boyer-Mooreův algoritmus provedl celkem 17 porovnání znaků.

8.9 Quick Search algoritmus

Hlavní rysy

- zjednodušení Boyer-Mooreova algoritmu,
- používá pouze posuny při neshodě znaku,
- jednoduchá implementace,
- předzpracování s časovou složitostí $O(m + \sigma)$, prostorovou $O(\sigma)$,
- časová složitost vyhledávání $O(mn)$,
- v praxi velice rychlý - platí pro krátké vzorky nad rozsáhlou abecedou.

Popis

Algoritmus **Quick Search** [21] vychází z Boyer-Moorova algoritmu (viz kapitola 8.8), který zjednodušuje. Funkce Bc, která v původním algoritmu nehrála příliš velkou roli se při praktickém použití ukazuje jako nečekaně efektivní. V běžných situacích (např. textové editory) má abeceda mnohem větší mohutnost ve srovnání s délkou vzorku. Typickým vzorkem v textovém editoru je slovo o délce několika znaků, kdeždo abeceda obsahuje v případě ASCII 256 znaků nebo dokonce 65536 v případě UNICODE. Abeceda proto obsahuje velký počet znaků, které se ve vzorku nevyskytují, k neshodám znaků dochází mnohem častěji než v našem ukázkovém příkladu. Naopak funkce Gs svého významu pozbývá a je vynechána.

Předpokládejme, že jsme porovnávali vzorek x s částí textu $y[j \dots j + m - 1]$. Pokud jsme zde vzorek nenašli, musíme jej posunout nejméně o jeden znak doprava. Z toho plyne, že se vzorek bude nutně porovnávat i se znakem $y[j + m]$. A dále z toho plyne, že znak $y[j + m]$ můžeme ihned použít pro výpočet funkce Bc. Funkci Bc lze potom definovat, obdobně jako v Boyer-Moorově algoritmu, předpisem $\forall c \in \Sigma$:

$$Bc(c) = \begin{cases} min\{i + 1 : (1 \leq i < m - 1) \wedge (x[m - 1 - i] = c)\} & \text{jestliže } c \in x \\ m + 1 & \text{jinak} \end{cases}$$

Vzorek je možné, během vyhledávání, porovnávat s textem v libovolném směru, čili jak sousměrně tak i protisměrně. Vyhledávání má v nejhorším případě kvadratickou časovou složitost, nicméně průměrná je nižší $O(n/m)$.

Implementace

```
void PreprocessBc(const char *x, const int m, int Bc[])
{
    int i;
    for (i = 0; i < AlphabetSize; i++)
        Bc[i] = m + 1;
    for (i = 0; i < m; i++)
        Bc[x[i]] = m - i;
}


void QuickSearch(const char *x, const int m, const char *y, const int n)
{
    int i, j;
    int Bc[AlphabetSize];

// Predzpracovani
    PreprocessBc(x, m, Bc);

// Vyhledavani
    i = 0;
    while (i <= n - m)
    {
        for(j = 0; j < m; j++)
            if (x[j] != y[i+j])
                break;
        if (j == m)
            printf ("Vzorek nalezen na pozici %d\n", i);
        i += Bc[y[i + m]];
    }
} // QuickSearch
```

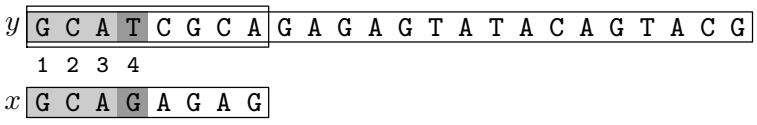
Příklad

Fáze předzpracování

c	A	C	G	T
$Bc[c]$	2	7	1	9

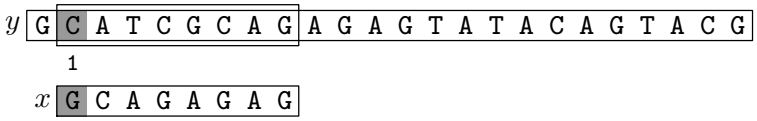
Fáze vyhledávání

První pokus:



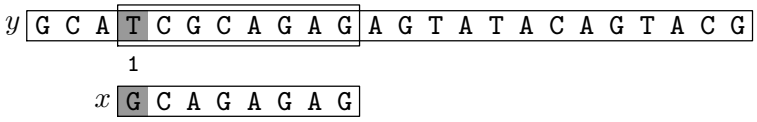
Posun o 1 (Bc[G])

Druhý pokus:



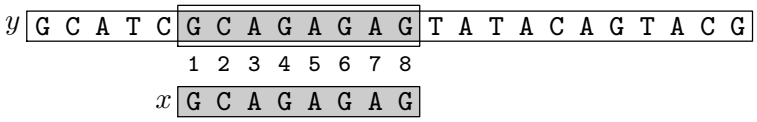
Posun o 2 (Bc[A])

Třetí pokus:



Posun o 2 (Bc[A])

Čtvrtý pokus:



Posun o 9 (Bc[T])

Pátý pokus:

Posun o 7 (Bc[C])

Quick Search algoritmus provedl 15 porovnání znaků.

Příloha A

Algoritmus, datové typy, řídící struktury

A.1 Základní pojmy

Procesor

Procesorem je objekt, který vykonává algoritmem popisovanou činnost (může jím být stroj (počítač) nebo člověk). Formulace algoritmu souvisí s tím, pro jaký typ procesoru se bude vytvářet.

Etapy řešení problému

K řešení problému potřebujeme vědět s jakými údaji budeme pracovat (vstupní, výstupní a vnitřní data - mezivýsledky) a podmínky, za jakých lze docílit správného výsledku. Kroky, které spolu logicky souvisejí, lze seskupit do *bloků, modulů* (množina kroků). Řešení problému probíhá v několika etapách:

- *specifikace (definice) problému*

  je nutné znát vstupy, se kterými bude řešení problému spojeno, a požadavky na výstupy,

- *analýza problému*

  volba vhodné metody řešení, rozsáhlejší problémy rozdělujeme na podproblémy,

- *sestavení algoritmu*

  posloupnost na sebe navazujících kroků (řídící struktury),

- *kódování*

  zápis algoritmu v jayzce, kterému rozumí procesor (např. v programovacím jazyce),

- *testování*

  ověření správnosti navrženého algoritmu.

Zápis algoritmu

- přirozený jazyk (slovní popis),
- grafické znázornění (např. vývojový diagram),
- speciální jazyk (pseudojazyk),
- programovací jazyk.

Datový typ

Datový typ určuje jakých hodnot může nabývat objekt daného datového typu a množinu přípustných operací nad tímto datovým typem. Datovým objektem rozumíme *konstantu, proměnnou, výraz a funkci*.

Identifikátory

Identifikátory jsou jména, která dáváme např. konstantám, proměnným, funkcím. Tato jména mohou být tvořena písmeny anglické abecedy, číslicemi a znakem podtržítko. Prvním znakem musí být písmeno nebo znak podtržítko, pak může následovat libovolná sekvence písmen, číslic a znak podtržítko. Délku identifikátoru můžeme považovat za prakticky libovolnou[1].

Konstanta

Konstanta je veličina, která nemění hodnotu během řešení problému. Může být použita dvěmi způsoby,

1. *přímo*, (63, 10^{-2}, 'ABC') nebo
2. *pojmenováním* (označení identifikátorem), (Pi jméno konstanty $3,14\ldots$).

Proměnná

Proměnná je veličina, která může měnit hodnotu během řešení problému. Proměnná se zavádí *definicí* (pojmenování a určení datového typu konkrétní proměnné).

[1]Délka identifikátoru je omezena jednak normou jazyka a jednak konkrétním kompilátorem daného jazyka. V obou případech je délka omezena desítkami, dokonce stovkami znaků

Výraz

Výraz je tvořen operátory, operandy a speciálními znaky. Operandem může být:

- konstanta,
- proměnná
- výraz
- a volání funkce,
- operátor a příslušné operandy, popřípadě závorky. Operand je tvořen opět výrazem.

Příklad A.1
Ukázka výrazů:

```
12          "abc"       a
b           sin(0.5)    12+9*3
(a+b)/2     a>=c
```

Příkazy

Příkazy (také ozn. jako řídící struktury) popisují jednotlivé kroky algoritmu a jejich návaznosti. Rozlišujeme *jednoduché* a *strukturované* příkazy. Celý algoritmus lze chápat jako jeden příkaz. (Pozn. pro zápis algoritmu v některém programovacím jazyce zpravidla platí, že z výrazu se stane příkaz teprve tehdy, až za něj vložíme středník.)

A.2 Datové typy

Datový typ (dále DT) určuje množinu hodnot, kterých může nabýt konstanta, proměnná, funkce nebo výraz a množinu operací nad těmito hodnotami. **Definice objektu** (konstanty, proměnné, funkce) znamená přiřazení jednoznačného jména – (identifikátoru) a určení jeho datového typu. Datový typ určuje, kolik místa (bajtů) bude v paměti pro např. proměnnou tohoto typu vyhrazeno.

Jednoduché datové typy

Logický typ (Boolean)

Objekt datového typu ***boolean*** může nabývat dvou hodnot – *nepravda* a *pravda*. Nad logickým datovým typem jsou definovány logické operace jejichž výsledkem je opět logická hodnota. Základní logické operace jsou negace, konjunkce (logický součin) a disjunkce (logický součet), pomocí nichž lze realizovat libovolnou logickou funkci.

Číselné datové typy

Objekt datového typu ***celé číslo*** nabývá hodnot z množiny celých čísel. Objekt datového typu ***reálné číslo*** nabývá hodnot z množiny reálných čísel. V obou případech závisí rozsah, případně přesnost s jakou jsou čísla reprezentována, na konkrétním operačním systému a použitém překladači.

Nad číselným DT definovámy tyto operace:

- Aritmetické operace,
- Relační operace.

Znak

Množina hodnot DT ***znak*** je tvořena znaky abecedy (malá, velká písmena), číslicemi a speciálními znaky. Každému znaku je přiřazena celočíselná hodnota, tzv. *kód znaku*. Přiřazení kódů jednotlivým znakům je voleno tak, aby odpovídalo jejich pořadí v tzv. kódovací tabulce. Mezi nejpoužívanější kódování patří kódování podle norem ASCII, Unicode. Takové přiřazení usnadňuje další práci se znaky, například řazení podle abecedy.

Strukturované datové typy

Pole

Pole je posloupnost prvků stejného DT. Jedinou operací nad DT pole je přístup k jednotlivým prvkům pole pomocí indexu, tj. celého čísla, které udává pozici prvku v poli. Prvkem pole může být opět pole – vznikají dvou a vícerozměrná pole. Speciálním případem pole je ***řetězec***, jehož prvky jsou typu *znak*.

Struktura

Struktura je tvořena několika elementy – *položkami*, obecně různého typu. Definice struktury znamená pojmenování struktury a určení datového typu jednotlivých položek a jejich pojmenování. Pomocí identifikátorů položek se v algoritmu přistupuje k hodnotě příslušné položky.

Ukazatel

Ukazatel (angl. pointer), podobně jako index u datového typu pole, neobsahuje přímo data uložená v proměnné, ale určuje pouze polohu této proměnné v paměti. Rozdíl mezi ukazatelem a indexem spočívá v tom, že index určuje polohu proměnné v poli – i-tý prvek, zatímco ukazatel obsahuje přímo adresu buněk paměti počítače, kde je proměnná uložena.

Ukazatele se používají v programovacích jazycích pro práci s proměnnými, které vytváříme v průběhu programu. Mnohdy předem neznáme

množství dat s nimiž budeme pracovat. Proto si tyto proměnné vytváříme až v okamžiku, kdy je jich zapotřebí. Na takovou dynamicky vytvořenou proměnnou se odkazujeme právě pomocí ukazatele.

Uživatelem definované datové typy

Většina jazyků umožňuje programátorovi definovat své datové typy. Můžeme vytvořit například pole, jehož prvky jsou typu struktura.

A.3 Řídící struktury

Jednoduché příkazy

Mezi jednoduché příkazy patří prázdný příkaz a volání funkce.

Strukturované příkazy

Sekvence, posloupnost

Sekvence je tvořena posloupností jednoho nebo více příkazů, které se provádějí v pevně daném pořadí. Příkaz se začne provádět až po ukončení předchozího příkazu.

Selekce

Provedení dalšího příkazu je závisí na splnění podmínky, tedy podmíněný příkaz určí, který z příkazů bude vykonán v závislosti na splnění či nesplnění podmínky. Existují dvě varianty ***podmíněného příkazu***:

1. úplný

```
if (výraz)
{
    příkaz1
}
else
{
    příkaz2
}
```

2. neúplný

```
if (výraz)
{
    příkaz1
}
```

Cyklus

Cyklus je část algoritmu, která je opakovaně prováděna za splnění *řídící podmínky*. Opakující se příkaz (příkazy) nazýváme *tělo cyklu*.

Rozlišujeme dva typy cyklů:

- *indukční* - řídící podmínka cyklu určuje, zda bude provedena posloupnost příkazů, která tvoří tělo cyklu, nebo dojde k předání řízení za tělo cyklu,
- *iterační* - počet opakování těla cyklu závisí na hodnotě řídící proměnné.

Druhy cyklů

Volba typu cyklu záleží na řešeném problému, převod cyklů mezi sebou je možný. Rozeznáváme následující druhy cyklů:

1. ***cyklus s podmínkou před vykonáním těla cyklu***

   ```
   while (výraz)
   {
       příkaz
   }
   ```

   U tohoto cyklu dochází k jeho ukončení v případě, že podmínka není splněna. Tělo cyklu se tedy nemusí vykonat ani jednou.

2. ***cyklus s podmínkou za tělem cyklu***

   ```
   do
   {
       příkaz
   }
   while (výraz)
   ```

   Tělo cyklu provede minimálně jednou, protože k prvnímu testování podmínky dojde až po prvním průchodu tělem cyklu.

3. ***cyklus s pevným počtem opakování***

   ```
   for(výraz1; výraz2; výraz3)
   {
       příkaz
   }
   ```

Zásady pro řízení cyklů:

1. před zahájením cyklu musí řídící proměnné nabývat smysluplných hodnot, umožňujících jeho ukončení
2. tělo indukčního cyklu musí zajistit změnu řídících proměnných cyklu

Cvičení

1. Navrhněte proceduru nebo funkci, která nalezne v matici typu $m \times n$ prvek s maximální hodnotou a určí pozici jeho posledního výskytu.
2. Navrhněte proceduru nebo funkci, která nalezne v matici typu $m \times n$ prvek s minimální hodnotou a určí pozici jeho posledního výskytu.
3. Navrhněte proceduru nebo funkci, která nalezne v poli celých čísel prvek s minimální hodnotou a určí počet jeho výskytů v poli.
4. Navrhněte proceduru nebo funkci, která vypočte skalární součin dvou N - prvkových vektorů.
5. Navrhněte proceduru nebo funkci, která určí počet cifer zadaného kladného celého čísla.
6. Navrhněte proceduru nebo funkci, která provede v matici typu $m \times n$ záměnu prvního a posledního sloupce matice.
7. Navrhněte proceduru nebo funkci, která slouží k výpočtu ciferného součtu daného přirozeného čísla (např. číslo 463 má ciferný součet 13).
8. Navrhněte proceduru nebo funkci, která provede zrcadlové obrácení vstupního řetězce a obrácený výstupní řetězec vypíše.
9. Navrhněte proceduru nebo funkci, která provede v matici typu $m \times n$ záměnu prvního a posledního sloupce matice.
10. Navrhněte logickou funkci, která určí, zda jsou si dva řetězce rovny.

Příloha B

Vybrané zdrojové kódy

B.1 Implementace binárního stromu

```
template<class T>class CBinaryTree
{
public:
  CBinaryTree();
  ~CBinaryTree();

  void  Insert (T x);      // vložení vrcholu s klíčem x
  void  Delete(T x);       // smazání vrcholu s klíčem x
  bool  Search(T x);       // rekurzivní hledání x ve stromu
  bool  SearchN(T x);      // nerekurzivní hledání
  int   Count();           // počet vrcholů ve stromu
  T     Minimum();
  T     Maximum();

  void  InOrder();         // in-order průchod stromem

protected:
// vrchol stromu
  struct CNode
  {
    T       key;
    CNode* left;
    CNode* right;
  };  // CNode

  void  FreeAll(CNode* p);              // zruší celý strom
  void  Ins(CNode*& p, T x);            // vložení x (rekurzivní prohledání)
  bool  Srch(CNode* p, T x);            // vyhledání x
  void  Del(CNode*& p, T x);            // vyhledání x a smazání
  void  Del1(CNode*& r, CNode*& q); // pomocná metoda pro mazání
  void  DoInOrder(CNode* p);            // rekurzivně projde strom
  int   CountIt(CNode* p);              // rekurzivní počítání vrcholů

  CNode* m_root;      // pointer na kořen stromu
};  // CBinaryTree
```

```
template<class T>CBinaryTree<T>::CBinaryTree()
{  m_root = NULL; }

template<class T>CBinaryTree<T>::~CBinaryTree()
{  FreeAll(m_root); }

template<class T> void CBinaryTree<T>::Insert(T x)
{  Ins(m_root, x); }

template<class T> void CBinaryTree<T>::Delete(T x)
{  Del(m_root, x); }

template<class T> bool CBinaryTree<T>::Search(T x)
{
  return Srch(m_root, x);
}  // CBinaryTree::Search

template<class T> bool CBinaryTree<T>::SearchN(T x)
{
  CNode* p = m_root;
  while (p != NULL)
  {
    if (x < p->key)
      p = p->left;
    else
      if (x > p->key)
        p = p->right;
      else
        return true;      // nalezeno
  };  // while
  return false;           // p == NULL nenalezeno
}  // CBinaryTree::SearchN

template<class T> int CBinaryTree<T>::Count()
{
  return CountIt(m_root);
}  // CBinaryTree::Count

template<class T> T CBinaryTree<T>::Minimum()
{
  CNode *p = m_root;
  while (p->left != NULL)
    p = p->left;
  return p->key;
}  // CBinaryTree::Minimum

template<class T> T CBinaryTree<T>::Maximum()
{
  CNode *p = m_root;
  while (p->right != NULL)
    p = p->right;
  return p->key;
}  // CBinaryTree::Maximum
```

```
template<class T> void CBinaryTree<T>::InOrder()
{
  DoInOrder(m_root);
}  // CBinaryTree::InOrder

template<class T> void CBinaryTree<T>::FreeAll(CNode* p)
{
  if (p != NULL)
  {
    FreeAll(p->left);        // zrušíme levý podstrom
    FreeAll(p->right);       // zrušíme pravý podstrom
    delete p;                // nakonec smažeme vrchol p
  };  // if
}  // CBinaryTree::FreeAll

template<class T> void CBinaryTree<T>::Ins(CNode*& p, T x)
{
  if (p == NULL)
  {// vytvoříme nový vrchol
    p = new CNode;
    p->key = x;
    p->left = p->right = NULL;
  }  // if
  else if (x < p->key)
    Ins(p->left, x);         // pokračujeme v levém podstromu
    else if (x > p->key, x)
      Ins(p->right, x);      // pokračujeme v pravém podstromu
      else
      {// duplicitní klíč
       // lze ignorovat, počítat výskyty atd.
      };  // else
}  // CBinaryTree::Ins

template<class T> bool CBinaryTree<T>::Srch(CNode* p, T x)
{
  if (p == NULL)
    return false;      // x nenalezeno
  if (x < p->key)
    return Srch(p->left, x);
  if (x > p->key)
    return Srch(p->right, x);
  return true;         // x == p->key nalezeno
}  // CBinaryTree::Srch

template<class T> void CBinaryTree<T>::Del(CNode*& p, T x)
{
  CNode    *q;
  if (p == NULL)
    return;    // x není ve stromu
  if (x < p->key)
    Del(p->left, x);
  else
    if (x > p->key)
```

```
      Del(p->right, x);
    else
    {// x == p->key
      q = p;
      if (q->right == NULL)
        p = q->left;          // žádný nebo jen levý potomek
      else
        if (q->left == NULL)
          p = q->right;       // existuje jen pravý potomek
        else
        {// existují oba potomci
          Del1(q->left, q);   // nejpravější z levého podstromu
        }; // else
      delete q;
    }; // else
} // CBinaryTree::Del

template<class T> void CBinaryTree<T>::Del1(CNode*& r, CNode*& q)
{
  if (r->right != NULL)
    Del1(r->right, q);  // hledáme nejpravějšího potomka
  else
  {
    q->key = r->key; // okopírujeme data
    q = r;
    r = r->left;
  }; // else
} // CBinaryTree::Del1

template<class T> void CBinaryTree<T>::DoInOrder(CNode* p)
{
  if (p != NULL)
  {
    DoInOrder(p->left);
    // zpracování dat v p
    // např. výpis pomocí
    cout << p->key << " ";
    DoInOrder(p->right);
  }; // if
} // CBinaryTree::DoInOrder

template<class T> int CBinaryTree<T>::CountIt(CNode* p)
{
  if (p == NULL)
    return 0;   // prázdný strom
  else
    return 1 + CountIt(p->left) + CountIt(p->right);
} // CBinaryTree::CountIt
```

B.2 Implementace AVL-stromu

```
template<class T> class CAVLTree
```

```
{
public:
  CAVLTree();
  ~CAVLTree();

  void Insert (T x);
  void Delete(T x);

protected:
  struct CNode
  {
    T       key;
    CNode* left;
    CNode* right;
    int     bal;
  }; // CNode;

  void  FreeAll(CNode* p);
  void  DoInsert(CNode*& p, T x, bool& h);
  void  DoDelete(CNode*& p, T x, bool& h);
  void  Balance1(CNode*& p, bool& h);
  void  Balance2(CNode*& p, bool& h);
  void  Del(CNode*& r, CNode*& q, bool& h);

  CNode* m_root;
}; // CAVLTree

template<class T> CAVLTree<T>::CAVLTree()
{ m_root = NULL; }

template<class T> CAVLTree<T>::~CAVLTree()
{ FreeAll(m_root); }

template<class T> void CAVLTree<T>::Insert(T x)
{
  bool h = false;
  DoInsert(m_root, x, h);
} // CAVLTree::Insert

template<class T> void CAVLTree<T>::Delete(T x)
{
  bool h = false;
  DoDelete(m_root, x, h);
} // CAVLTree::Delete

template<class T> void CAVLTree<T>::DoInsert(CNode*& p, T x,
  bool& h)
{ // h == false
  CNode *p1, *p2;
  if (p == NULL)
  {
    p = new CNode;
    p->key = x;
    p->left = p->right = NULL;
```

```
  p->bal = 0;
  h = true;
  return;
};  // if
if (x < p->key)
{
  DoInsert(p->left, x, h);
  if (h)
  {
    switch (p->bal)
    {
    case 1:
      p->bal = 0;
      h = false;
      break;
    case 0:
      p->bal = -1;
      break;
    case -1:
      p1 = p->left;
      if (p1->bal == -1)
      {// LL
        p->left = p1->right;
        p1->right = p;
        p->bal = 0;
        p = p1;
      }  // if
      else
      {//  LR
        p2 = p1->right;
        p1->right = p2->left;
        p2->left = p1;
        p->left = p2->right;
        p2->right = p;
        p->bal = (p2->bal == -1) ? +1 : 0;
        p1->bal = (p2->bal == +1) ? -1 : 0;
        p = p2;
      };  // else
      p->bal = 0;
      h = false;
      break;
    };  // switch
  };  // if
  return;
};  // if
if (x > p->key)
{
  DoInsert(p->right, x, h);
  if (h)
  {
    switch (p->bal)
    {
    case -1:
      p->bal = 0;
```

```
        h = false;
        break;
      case 0:
        p->bal = +1;
        break;
      case +1:
        p1 = p->right;
        if (p1->bal == +1)
        {// RR
          p->right = p1->left;
          p1->left = p;
          p->bal = 0;
          p = p1;
        } // if
        else
        {// RL
          p2 = p1->left;
          p1->left = p2->right;
          p2->right = p1;
          p->right = p2->left;
          p2->left = p;
          p->bal = (p2->bal == +1) ? -1 : 0;
          p1->bal = (p2->bal == -1) ? +1 : 0;
          p = p2;
        }; // else
        p->bal = 0;
        h = false;
        break;
      }; // switch
    }; // if
    return;
  }; // if
  // duplicitni klic
  h = false;
} // CAVLTree::DoInsert

template<class T> void CAVLTree<T>::FreeAll(CNode* p)
{
  if (p != NULL)
  {
    FreeAll(p->left);       // zrušíme levý podstrom
    FreeAll(p->right);      // zrušíme pravý podstrom
    delete p;               // nakonec smažeme vrchol p
  }; // if
} // CAVLTree::FreeAll

template<class T> void CAVLTree<T>::DoDelete(CNode*& p, T x,
  bool& h)
{
  if (p == NULL)
    h = false;     // klíč x není ve stromu
  else
    if (x < p->key)
    {
```

```
      DoDelete(p->left, x, h);
      if (h)
        Balance1(p, h);
    } // if
    else
      if (x > p->key)
      {
        DoDelete(p->right, x, h);
        if (h)
          Balance2(p, h);
      } // if
      else
      {// x == p->key
        CNode *q = p;
        if (q->right == NULL)
        {
          p = q->left;
          h = true;
        } // if
        else
          if (q->left == NULL)
          {
            p = q->right;
            h = true;
          } // if
          else
          {
            Del(q->left, q, h);
            if (h)
              Balance1(p, h);
          }; // else
        delete q;
      }; // else
} // CAVLTree::DoDelete

template<class T> void CAVLTree<T>::Balance1(CNode*& p, bool& h)
{// h = true, levá větev se zmenšila
  CNode  *p1, *p2;
  int    b1, b2;
  switch (p->bal)
  {
  case -1:
    p->bal = 0;
    break;
  case 0:
    p->bal = +1;
    h = false;
    break;
  case 1:
    p1 = p->right;
    b1 = p1->bal;
    if (b1 >= 0)
    {// jednoducha RR rotace
      p->right = p1->left;
```

```
      p1->left = p;
      if (b1 == 0)
      {
        p->bal = +1;
        p1->bal = -1;
        h = false;
      } // if
      else
      {
        p->bal = 0;
        p1->bal = 0;
      }; // else
      p = p1;
    } // if
    else
    {// RL
      p2 = p1->left;
      b2 = p2->bal;
      p1->left = p2->right;
      p2->right = p1;
      p->right = p2->left;
      p2->left = p;
      p->bal = (b2 == +1) ? -1 : 0;
      p1->bal = (b2 == -1) ? +1 : 0;
      p = p2;
      p2->bal = 0;
    }; // else
    break;
  }; // switch
} // CAVLTree::Balance1

template<class T> void CAVLTree<T>::Balance2(CNode*& p, bool& h)
{// h = true, pravá větev se zmenšila
  CNode *p1, *p2;
  int b1, b2;
  switch (p->bal)
  {
  case 1:
    p->bal = 0;
    break;
  case 0:
    p->bal = -1;
    h = false;
    break;
  case -1:
    p1 = p->left;
    b1 = p1->bal;
    if (b1 == -1)
    {// LL
      p->left = p1->right;
      p1->right = p;
      if (b1 == 0)
      {
        p->bal = -1;
```

```
        p1->bal = +1;
        h = false;
      } // if
      else
      {
        p->bal = 0;
        p1->bal = 0;
      }; // else
      p = p1;
    } // if
    else
    {// LR
      p2 = p1->right;
      b2 = p2->bal;
      p1->right = p2->left;
      p2->left = p1;
      p->left = p2->right;
      p2->right = p;
      p->bal = (b2 == -1) ? +1 : 0;
      p1->bal = (b2 == +1) ? -1 : 0;
      p = p2;
      p2->bal = 0;
    }; // else
  }; // switch
} // CAVLTree::Balance2

template<class T> void CAVLTree<T>::Del(CNode*& r, CNode*& q,
  bool& h)
{// h = false
  if (r->right != NULL)
  {
    Del(r->right, q, h);
    if (h)
      Balance2(r, h);
  } // if
  else
  {
    q->key = r->key;
    q= r;
    r = r->left;
    h = true;
  }; // else
} // CAVLTree::Del
```

B.3 Implementace Red-Black stromu

```
#ifndef _RedBlackTree_h_
#define _RedBlackTree_h_

#include <iostream.h>
#include <iomanip.h>
```

```
template<class T>class CRedBlackTree
{
public:
  CRedBlackTree();
  ~CRedBlackTree();

  void  Insert (T a);
  void  Delete(T a);

  void Report();

private:
  enum TColor {red, black};

  struct  CNode
  {
    T         key;
    TColor   color;
    CNode* left;
    CNode* right;
    CNode* parent;
  };  // CNode;

  void      FreeAll(CNode* p);
  CNode* TreeInsert(T x, CNode*& p, CNode* par);
  void      LeftRotate(CNode* x);
  void      RightRotate(CNode* y);
  CNode* TreeSuccessor(CNode* x);
  void      RBDeleteFixUp(CNode*& x);
  void      DoReport(CNode* p, int level);

  CNode* m_root;
  CNode* m_z;
};  // CRedBlackTree

template<class T> CRedBlackTree<T>::CRedBlackTree()
{
  m_z = new CNode;
  m_z->color = black;
  m_z->left = m_z->right = m_z->parent = m_z;
  m_root = m_z;
}  // CRedBlackTree::CRedBlackTree

template<class T> CRedBlackTree<T>::~CRedBlackTree()
{
  FreeAll(m_root);
  delete m_z;
}  // CRedBlackTree::~CRedBlackTree

template<class T> void CRedBlackTree<T>::Insert(T a)
{
  CNode* x;
  CNode* y;
  x = TreeInsert(a, m_root, m_z);
```

```
if (x != m_z)
{// nový uzel
  while ((x != m_root) && (x->parent->color == red))
    if (x->parent == x->parent->parent->left)
    {
      y = x->parent->parent->right;
      if (y->color == red)
      {
        x->parent->color = black;
        y->color = black;
        x->parent->parent->color = red;
        x = x->parent->parent;
      }  // if
      else
      {
        if (x == x->parent->right)
        {
          x = x->parent;
          LeftRotate(x);
        };  // if
        x->parent->color = black;
        x->parent->parent->color = red;
        RightRotate(x->parent->parent);
      };  // else
    }  // if
    else
      if (x->parent == x->parent->parent->right)
      {
        y = x->parent->parent->left;
        if (y->color == red)
        {
          x->parent->color = black;
          y->color = black;
          x->parent->parent->color = red;
          x = x->parent->parent;
        }  // if
        else
        {
          if (x == x->parent->left)
          {
            x = x->parent;
            RightRotate(x);
          };  //  if
          x->parent->color = black;
          x->parent->parent->color = red;
          LeftRotate(x->parent->parent);
        };  // else
      }  // if
      else
      {// dvouprvkový strom
        return;
      };  // else
  };  // if
}  // CRedBlackTree::Insert
```

```
template<class T> void CRedBlackTree<T>::Delete(T a)
{
  CNode *x, *y, *z;
  // nalezení uzlu s klíčem a
  z = m_root;
  while (z != m_z)
    if (a < z->key)
      z = z->left;
    else
      if (z->key < a)
        z = z->right;
      else
        break;    // a == z->key
  if (z == m_z)
    return;   // není ve stromu => není co rušit
  if (z->left == m_z || z->right == m_z)
    y = z;
  else
    y = TreeSuccessor(z);
  if (y->left != m_z)
    x = y->left;
  else
    x = y->right;
  x->parent = y->parent;
  if (y->parent == m_z)
    m_root = x;
  else
    if (y == y->parent->left)
      y->parent->left = x;
    else
      y->parent->right = x;
  if (y != z)
  {
    z->key = y->key;
    // kopie dalších složek uzlu
  };  // if
  if (y->color == black)
    RBDeleteFixUp(x);
}  // CRedBlackTree::Delete

template<class T> void CRedBlackTree<T>::Report()
{
  DoReport(m_root, 0);
}  // CRedBlackTree::Report

template<class T> void CRedBlackTree<T>::FreeAll(CNode* p)
{
  if (p != m_z)
  {
    FreeAll(p->left);
    FreeAll(p->right);
    delete p;
  };  // if
```

```
} // CRedBlackTree::FreeAll

template<class T> CRedBlackTree<T>::CNode*
  CRedBlackTree<T>::TreeInsert(T x, CNode*& p, CNode* par)
{
  if (p == m_z)
  {
    p = new CNode;
    p->key = x;
    p->color = red;
    p->left = p->right = m_z;
    p->parent = par;
    return p;
  }; // if
  if (x < p->key)
    return TreeInsert (x, p->left, p);
  if (p->key < x)
    return TreeInsert (x, p->right, p);
  // x == p->key => duplicitni klic
  return m_z;
} // CRedBlackTree::TreeInsert

template<class T> void CRedBlackTree<T>::LeftRotate(CNode* x)
{
  CNode* y;
  y = x->right;
  x->right = y->left;
  if (y->left != m_z)
    y->left->parent = x;
  y->parent = x->parent;
  if (x->parent == m_z)
    m_root = y;
  else
    if (x == x->parent->left)
      x->parent->left = y;
    else
      x->parent->right = y;
  y->left = x;
  x->parent = y;
} // CRedBlackTree::LeftRotate

template<class T> void CRedBlackTree<T>::RightRotate(CNode* y)
{
  CNode* x;
  x = y->left;
  y->left = x->right;
  if (x->right != m_z)
    x->right->parent = y;
  x->parent = y->parent;
  if (y->parent == m_z)
    m_root = x;
  else
    if (y == y->parent->right)
      y->parent->right = x;
```

```
    else
      y->parent->left = x;
  x->right = y;
  y->parent = x;
}  // CRedBlackTree::RightRotate

template<class T> CRedBlackTree<T>::CNode*
  CRedBlackTree<T>::TreeSuccessor(CNode* x)
{
  CNode* y;
  if (x->right != m_z)
  {
    y = x->right;
    while (y->left != m_z)
      y = y->left;
    return y;
  };  // if
  y = x->parent;
  while (y != m_z && x == y->right)
  {
    x = y;
    y = y->parent;
  };  // while
  return y;
}  // CRedBlackTree::TreeSuccessor

template<class T> void CRedBlackTree<T>::RBDeleteFixUp(CNode*& x)
{
  CNode* w;
  while (x != m_root && x->color == black)
    if (x == x->parent->left)
    {
      w = x->parent->right;
      if (w->color == red)
      {
        w->color = black;
        x->parent->color = red;
        LeftRotate(x->parent);
        w = x->parent->right;
      };  // if
      if (w->left->color == black && w->right->color == black)
      {
        w->color = red;
        x = x->parent;
      }  // if
      else
      {
        if (w->right->color == black)
        {
          w->left->color = black;
          w->color = red;
          RightRotate(w);
          w = x->parent->right;
        };  // if
```

```
        w->color = x->parent->color;
        x->parent->color = black;
        w->right->color = black;
        LeftRotate(x->parent);
        x = m_root;
      }; // else
    } // if
    else
    {
      w = x->parent->left;
      if (w->color == red)
      {
        w->color = black;
        x->parent->color = red;
        RightRotate(x->parent);
        w = x->parent->left;
      }; // if
      if (w->right->color == black && w->left->color == black)
      {
        w->color = red;
        x = x->parent;
      } // if
      else
      {
        if (w->left->color == black)
        {
          w->right->color = black;
          w->color = red;
          LeftRotate(w);
          w = x->parent->left;
        }; // if
        w->color = x->parent->color;
        x->parent->color = black;
        w->left->color = black;
        RightRotate(x->parent);
        x = m_root;
      }; // else
    }; // else
} // CRedBlackTree::RBDeleteFixUp


template<class T> void CRedBlackTree<T>::DoReport(CNode* p,
  int level )
{
  if (p == m_z)
    return;
  for(int i = 0; i < level; i++)
    cout << " ";
  cout << p->key;
  if (p->color == red)
    cout << " (red)";
  else
    cout << " (black)";
  cout << endl;
```

```
  DoReport(p->left, level + 2);
  DoReport(p->right, level + 2);
}  // CRedBlackTree::DoReport

#endif // __RedBlackTree_h__
```

Literatura

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] L. Ammeraal. *Algorithms and Data Structures in C++*. John Wiley, 1996.

[3] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[4] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12:403–417, 1980.

[5] G. Birkhoff and S. MacLane. *Prehľad modernej algebry*. ALFA Bratislava, 1979.

[6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[7] T. H. Cormen, C. E. Leiserson, and R. L.Rivest. *Introduction to Algorithms*. The MIT Press, 1991.

[8] J. Demel. *Grafy a jejich aplikace*. Praha, Academia, 2002.

[9] G. H. Gonnet and R. A. Baeza-Yates. *Text algorithms*, chapter 7, pages 251–288. Addison-Wesley, Wokingham, U. K., second edition, 1991.

[10] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[11] D. E. Knuth. *The art of computer programming*, volume 3 Sorting and Searching. Addison-Wesley Publishing Company, 1973.

[12] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[13] K. Mehlhorn. *Data Structures and Algoritms*, volume 1 Sorting and Searching. Springer-Verlag Berlin, 1984.

[14] B. Melichar. *Textové informační systémy.* Skriptum ČVUT Praha, 1994.

[15] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[16] J. Pokorný. *Základy implementace souborů a databází.* Skriptum MFF UK Praha, 1997.

[17] J. Pokorný, V. Snášel, and D. Húsek. *Dokumentografické informační systémy.* Karolinum Praha, 1998.

[18] R. Sedgewick. *Algorithms in C++.* Addison-Wesley Publishing Company, 1992.

[19] R. Sedgewick. *Algorithms in C.* Addison-Wesley Publishing Company, third edition, 1998.

[20] V. Snášel and M. Kudělka. Hanojské věže. Technical Report TR-CS-94-03, Univerzita Palackého Olomouc, 1994.

[21] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[22] J. Wiedermann. Algoritmy triedenia. *Informačné systémy*, 1,2:97–110, 205–234, ALFA 1986, Bratislava.

[23] N. Wirth. *Algoritmy a štruktúry údajov.* ALFA Bratislava, 1988. slovenský překlad.

[24] J. Švrček and J. Vanžura. *Geometrie trojúhelníka.* SNTL, 1988.

Rejstřík