INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA

FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

prof. Ing. Radim Farana, CSc.

Ostrava 2013

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

Název: APLIKOVANÁ INFORMATIKA

Autor: prof. Ing. Radim Farana, CSc.

Vydání: první, 2013

Počet stran: 163

Náklad: 5

Jazyková korektura: nebyla provedena.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Tyto studijní materiály vznikly za finanční podpory Evropského sociálního fondu a rozpočtu České republiky v rámci řešení projektu Operačního programu Vzdělávání pro konkurenceschopnost.

Název: Modernizace výukových materiálů a didaktických metod

Číslo: CZ.1.07/2.2.00/15.0463

Realizace: Vysoká škola báňská – Technická univerzita Ostrava

© prof. Ing. Radim Farana, CSc.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3017-9

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

1. LEKCE – ÚVOD

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

1 ÚVOD 3

1 ÚVOD

Education is what survives when what has been learnt has been forgotten.

B.F. Skinner, „Education in 1994“, New Scientist, 484, 28 May 1984

Již od počátků éry programování platí pravidlo, že základem kvalitního programového produktu je dobrá analýza a volba vhodného algoritmu pro řešení daného problému. Předložená skripta se proto zabývají vybranou skupinou základních algoritmů doplněnou množinou některých speciálních algoritmů. Algoritmy jsou rozděleny do čtyř kapitol, zabývajících se postupně metodami interního třídění, externího třídění, vyhledávání a algoritmy pro řešení kombinatorických úloh. Je zřejmé, že nebylo možné podat zcela vyčerpávající popis exitujících metod pro řešení problémů v uvedených oblastech a dále uvést i některé další skupiny algoritmů. Jejich skladba byla pečlivě volena za účelem využití ve výuce odborných předmětů na katedře Automatizační techniky a řízení fakulty strojní. Skripta mají tedy primárně sloužit jako studijní materiál pro předměty Základy informatiky a Aplikovaná informatika a dále pak jako doplňkový text pro další předměty, zejména v oblasti programování v jazyce C a C++. Skripta však mohou být využita jako samostatný studijní materiál, neboť jsou velice intenzivně doplněna praktickými příklady a výpisy zdrojových kódů.

Skripta jsou také doplněna několika kapitolami, které popisují různé způsoby uložení dat v operační paměti počítače. Rozvaha způsobu uložení dat je totiž dalším z podstatných kroků, při sestavování algoritmu a datové analýze.

Jako významnou otázku hodnotíme také způsob zápisu algoritmu. Je nutno volit přístup umožňující dobrou čitelnost algoritmu a současně snadnou realizaci algoritmu ve zvoleném programovacím jazyce. Proto byl jako první kapitola zařazen krátký přehled nejčastějších způsobů zápisu algoritmu, zejména s ohledem na grafické zápisy. Ve vlastních skriptech je pro zápis algoritmů zvolen pseudokód sestavený tak, aby vyhovoval přístupům strukturovaného programování a přitom umožňoval zápis algoritmu v co největším množství programovacích jazyků.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

2. LEKCE – ALGORITMUS A JEHO POPIS

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.
© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

2 ALGORITMUS A JEHO POPIS ... 3
2.1 Popis algoritmu ... 5
2.1.1 Slovní popis ... 5
2.1.2 Vývojový diagram ... 6
2.1.3 Diagram aktivit UML ... 7
2.1.4 Kopenogram ... 8
2.1.5 N-S diagram ... 9
2.1.6 Strukturogram ... 10
2.1.7 Pseudojazyk ... 11
2.2 Základní struktury algoritmu ... 12
2.3 Členění algoritmu ... 14
2.4 Proměnné ... 14
2.5 Hodnocení algoritmu ... 15
POUŽITÁ LITERATURA ... 18

2 ALGORITMUS A JEHO POPIS

OBSAH KAPITOLY:

Popis algoritmu

Základní struktury algoritmu

Členění algoritmu

Proměnné

Hodnocení algoritmu

MOTIVACE:

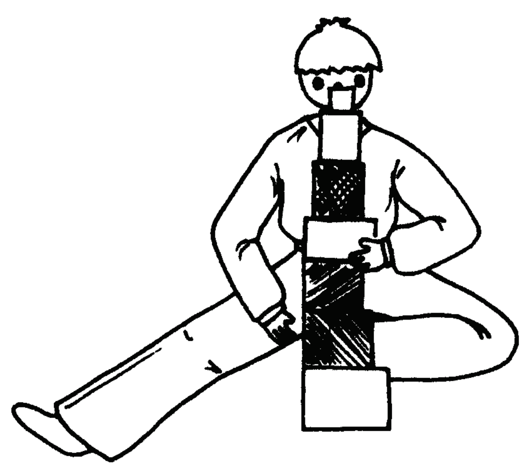
Základem dobré činnosti programů, informačních a řídicích systémů je kvalitně zpracovaný algoritmus jejich činnosti. Aby popis algoritmu snadno pochopili všichni členové realizačního týmu, je vhodné dokumentoval algoritmy v grafické podobě. To je důležité také pro možnost jejich dalšího rozvoje a úprav činnosti programu.

CÍL:

Pochopit jaké vlastnosti musí mít algoritmus a seznámit se s obvyklými metodami dokumentace algoritmů od různých verzí slovního popisu ke grafickým metodám jako jsou vývojové diagramy, diagramy aktivit, strukturogramy, nebo méně používané nástroje – kopenogramy a NS-diagramy. Seznámit se se základními strukturamy a členěním algoritmů. Umět používat správné druhy proměnných a předávání parametrů mezi jednotlivými programovými rutinami. Naučit se hodnotit algoritmy pro možnost jejich vzájemného porovnání a vyhodnocení jejich efektivity. Pochopit rozdělení složitosti algoritmů podle časové složitosti na polynomiální – P-problémy, nepolynomiální NP-problémy a NP-úplné problémy.

K tomu, aby mohl počítač provádět jisté manipulace s informací (daty) je potřeba určit jaké operace, v jakém pořadí a za jakých okolností mají být provedeny. Počítači je tedy třeba určit **postup práce** s daty, který samozřejmě může být závislý na tom, jaká data jsou zpracovávána. Tento postup práce je většinou obsažen v programu, který realizuje daný postup práce s využitím konkrétního **programovacího jazyka**, tedy jeho příkazů. Protože existuje řada různých programovacích jazyků, znamená to, že může existovat také řada různých programů, které realizují stejný postup práce. Převod programu do prostředí jiného programovacího jazyka je srovnatelný s překladem knihy do jiného jazyka. Duševní bohatství (chcete-li tedy informační obsah) díla se nemění. To je u programu uloženo právě v postupu práce, který program realizuje. V oblasti zpracování informace tento postup práce obvykle nazýváme **algoritmu**s. Definice algoritmu se různí, přijměme definici A. A. Markova [61]:

"Algoritmus je přesný předpis definující výpočtový proces vedoucí od měnitelných výchozích údajů až k žádaným (vždy správným) výsledkům. Tento předpis se skládá z jednotlivých výpočtových kroků, které jsou zapsány v určitém pořadí. Počet výpočtových kroků musí být konečný."



Obrázek 2.1 Algoritmus = postup práce

Abychom odlišili často zaměňované pojmy **algoritmus** a **program**, můžeme zjednodušeně uvést:

program = posloupnost příkazů,

algoritmus = postup práce.

Poznámka:

Slovo algoritmus pochází od uzbeckého matematika z IX. století Mohameda ibn Musa Al-Chórezmiho, který definoval základní pravidla pro provádění početních úkonů v desítkové soustavě.

Algoritmus musí splňovat několik základních požadavků:

1. **determinovanost** - shrnuje v sobě požadavky jako přesnost, srozumitelnost a jednoznačnost. To znamená, že v každém okamžiku řešení musí být jasné, jakou operaci má algoritmus provádět. (I když algoritmus čeká na příchod informace, která rozhodne o další činnosti, je jasné co má dělat.)
2. **hromadnost (masovost)** - algoritmus nesmí popisovat jen zpracování jedné sady konkrétních vstupních dat, ale celé skupiny příbuzných hodnot.
3. **rezultativnost** - algoritmus musí vždy dospět ke správnému výsledku, a to pomocí konečného počtu kroků.

4. **opakovatelnost** - při stejných hodnotách vstupních dat musí algoritmus vždy dospět ke stejnému výsledku.

2.1 POPIS ALGORITMU

Zatímco **popisem programu** je jeho **výpis**, s **popisem algoritmu** je to složitější. Postupně se vyvíjel a nabýval různých podob, které byly více či méně poplatné některému konkrétnímu programovacímu jazyku. Podle toho se také liší množina základních struktur, které je možno použít. V dalším textu nejprve na jednoduchém příkladu představíme několik typů popisu algoritmů a na závěr uvedeme přehled významných struktur, používaných v popisu algoritmů.

2.1.1 Slovní popis

Slovní popis vyjadřuje algoritmus prostředky přirozeného jazyka.

Příklad 2.1:

Z klávesnice čteme celá čísla a vypisujeme je na obrazovku doplněná o informaci, zda je číslo sudé nebo liché. Práce končí po vstupu čísla 0, které se nezpracovává.

U složitějších algoritmů je slovní popis komplikovaný, proto se často dělí do jednotlivých částí (bodů) nejčastěji očíslovaných vzestupnou řadou.

Příklad 2.2:

1 přečti číslo ze vstupu

2 když je číslo 0 jdi k bodu 9

3 vyšli číslo na výstup

4 když je číslo sudé, jdi k bodu 7

5 vyšli na výstup "liché"

6 jdi k bodu 1

7 vyšli na výstup "sudé"

8 jdi k bodu 1

9 konec

Slovní popis se tak velmi přiblížil k popisu **programu** pomocí **příkazů** [41][46][55][57]aj. Jednotlivé body se zpracovávají v přirozeném pořadí, pokud není proveden skok na jiný bod. U rozsáhlých popisů se poněkud ztrácí souvislosti při neustálých přeskocích. Při tvorbě algoritmu se navíc špatně vkládají další body do souvislé řady čísel. Aby se předešlo nutnosti opravovat čísla bodů a odskoků na ně, používá se číslování nesouvislou vzestupnou řadou (typicky s krokem 10). Tím se zápis algoritmu dále přibližuje zápisu jeho realizace, zejména v programovacím jazyku **BASIC** (Beginners All-purpose Symbolic Instruction Code). Tento jazyk vznikl v USA v roce 1964. Jeho snahou je co největší přiblížení příkazů přirozenému jazyku, samozřejmě anglickému. Postupně ovládl především mikropočítače, zejména díky své jednoduchosti. Později byl jeho vliv přehlušen vyššími programovacími jazyky (PASCAL, C, aj.) neboť základní množina jeho příkazů neumožňuje jednoduše realizovat některé běžné postupy (opakování s testem na začátku apod.). V současné době (pět let do konce tisíciletí) se BASIC opět vrací, samozřejmě výrazně rozšířený.

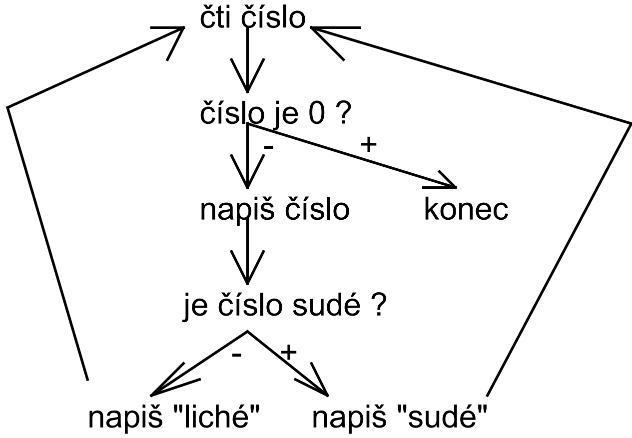
Zápis algoritmů přímo pomocí příkazů jazyka BASIC je špatně přenositelná do jiných programovacích jazyků a je rovněž špatně čitelný.

Příklad 2.3:

```
10 Read Cislo
20 If Cislo=0 Then Goto 90
30 Write Cislo
40 If Int(Cislo/2)*2=Cislo Then Goto 70
50 Write "liché"
60 Goto 10
70 Write "sudé"
80 Goto 10
90 End
```

Protože návaznosti jednotlivých částí algoritmu jsou při slovním popisu špatně patrné, používají někteří autoři **grafický zápis** [47]. V něm vhodně rozmístí jednotlivé činnosti a jejich návaznost označí šipkami. Zápis je pak podobný orientovanému grafu, kde uzly představují činnosti a hrany návaznost činností.

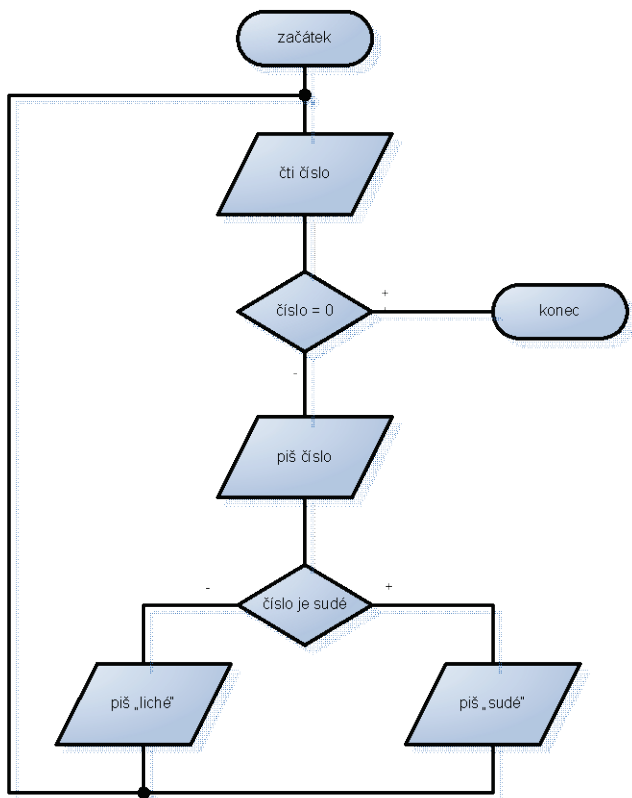
Příklad 2.4:



2.1.2 Vývojový diagram

Vývojový diagram můžeme chápat jako více formalizovaný grafický zápis algoritmu [7][14][27][61]. Pro jednotlivé druhy činností (výkonná činnost, rozhodování, vstup dat apod.) jsou zavedeny speciální grafické symboly, do kterých se zapisuje konkrétní prováděná činnost. Vývojové diagramy vznikly především pro popis algoritmů zpracovávaných v jazyce **FORTRAN** a nese s sebou také řadu omezení s tím spojených (programovací jazyk FORTRAN byl vyvinut v r. 1954 vývojovou skupinou IBM, kterou vedl John Backus, název je převzat ze slov FORmula TRANslator) Používání vývojových diagramů se později natolik rozšířilo, že tvar jednotlivých značek byl formalizován ČSN 36 9030, která rozeznává přes 30 značek. Část značek je obecná, (zpracování, rozhodování aj.), jiné jsou vysoce specializované (magnetický buben, disketa aj.).

Příklad 2.5:



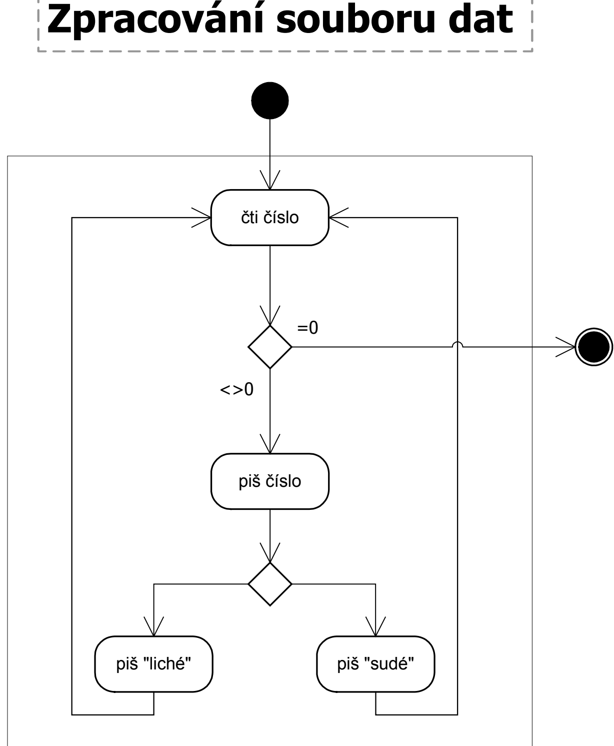
Mezi výhody blokových diagramů patří zejména možnost volby různé úrovně podrobnosti (hrubý vývojový diagram algoritmu se slovním popisem činnosti nebo podrobný popis odpovídající použitým příkazům). Vývojový diagram umožňuje poměrně snadno realizovat popsaný algoritmus prostředky různých programovacích jazyků.

Jako nevýhody jsou chápány skutečnosti, že vývojový diagram nevede uživatele k dodržování zásad **strukturovaného programování**, jak bude popsáno dále. Navíc se některé běžné struktury (např. opakování s udaným počtem cyklů) v blokovém diagramu realizují poněkud těžkopádně.

2.1.3 Diagram aktivit UML

Diagramy aktivit se podobají vývojovým diagramům, ale jsou nástrojem komplexního CASE (Computer-Aided Software Engineering) systému s názvem Unified Modeling Language, ve zkratce UML. UML představuje vizuální modelovací jazyk, který je v současné době standardem v oblasti analýzy a návrhu softwarových systémů, zejména objektově orientovaných. UML umožňuje vytvořit komplexní model celého systému, na který pohlížíme jednotlivými pohledy, z nichž právě diagram aktivit nejlépe popisuje činnost systému. Je určen k modelování řídicích a datových toků, tedy k posloupnosti aktivit, podmínek jejich vykonávání a toku dat mezi jednotlivými aktivitami. Jako jediný z popsaných grafických systémů zápisu algoritmu podporuje masivní paralelismus

Příklad 2.6:

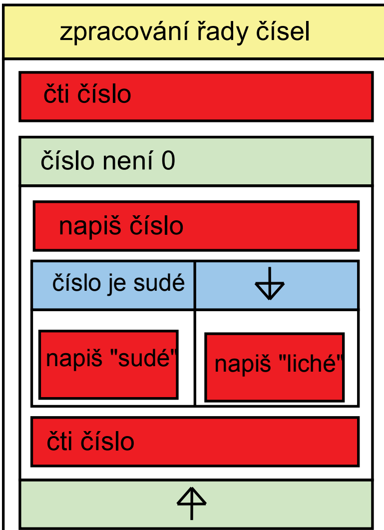


2.1.4 Kopenogram

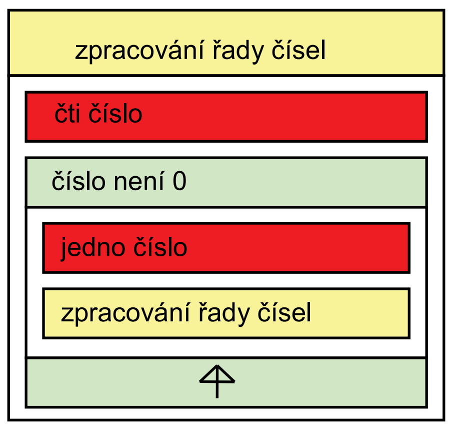
Název metody popisu algoritmů nazvané kopenogram je odvozen od jmen jejích tvůrců, kterými jsou Kofránek, Pecinovský a Novák. Vznikly především pro zápis programů dodržujících zásady **strukturovaného programování**, ve kterém je striktně předepsáno, že každá ucelená struktura algoritmu (opakování, rozhodování apod.) je vždy buď celá vnořena do jiné struktury, je s ní na stejné úrovni, nebo jinou strukturu obsahuje jako vnořenou [26][51].

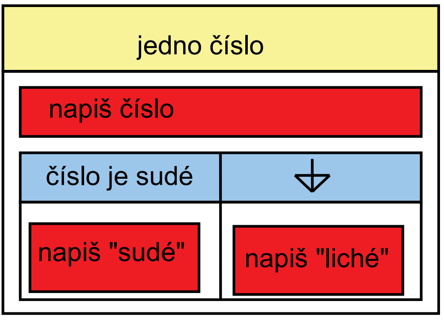
Kopenogramy se používaly zejména při tvorbě algoritmů ve výukovém programovacím jazyku **KAREL**. Tento jazyk striktně vyžaduje správné vnořování struktur. Např. násilné opuštění cyklu opakování (oblíbený příkaz nepodmíněného skoku známý z jazyka BASIC) zde vůbec neexistuje. Čitelnost složitějších algoritmů popsaných kopenogramy není nejlepší, což vede k nutnosti rozkládat algoritmus na jednotlivé ucelené činnosti realizované samostatnými **procedurami**. Tato skutečnost může být chápána jako výhoda, či jako nevýhoda. Do značné míry to závisí na osobních názorech hodnotícího. Pro zvýraznění jednotlivých typů struktur (opakování, větvení apod.) je doporučováno je vybarvovat vždy stejnou barvou. To však v našich ukázkách nemůžeme využít.

Příklad 2.7



Příklad 2.8:





Jedním ze silných nástrojů strukturovaného programování je **rekurzivní volání** (**rekurze**), při kterém procedura volá sama sebe. Její využití ukazuje příklad 2.7. Rozlišujeme dva druhy rekurze:

nepravá rekurze - za rekurzivním voláním již nenásleduje žádná struktura algoritmu, ale jen konce struktur, do kterých je rekurzivní volání vnořeno.

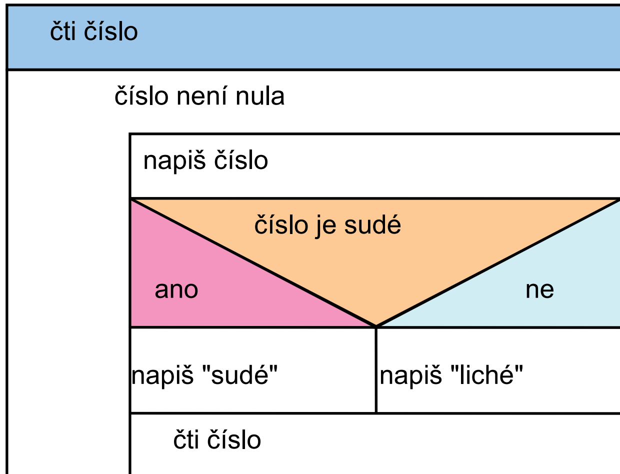
pravá rekurze - za rekurzivním voláním následuje alespoň jedna struktura na stejné nebo nižší úrovni vnoření.

S využitím rekurze souvisí některé nepříjemné problémy, zejména nebezpečí **přetečení zásobníku**. Budeme se jim podrobněji věnovat v kapitole věnované **dynamickým datovým strukturám**.

2.1.5 N-S diagram

Název **N-S diagram** je opět odvozen od jmen autorů, kterými jsou Isaac „Ike“ Nassi a Ben Schneiderman. Jsou velmi podobné kopenogramům a mají v zásadě shodné vlastnosti. Liší se pouze grafickým ztvárněním jednotlivých typů struktur [26].

Příklad 2.9:



Výsledný zápis N-S diagramu působí oproti kopenogramu kompaktním dojmem. Navíc zatímco kopenogramy jsou českou specialitou, byť je považujeme za přehlednější (při využití barevného zvýraznění), výhodou N-S diagramů je skutečnost, že byly publikovány v předním světovém časopise.

2.1.6 Strukturogram

Konstrukce **strukturogramů** vychází z postupu definovaného Michaelem Anthony Jacksonem v roce 1975. Vychází z poznatků strukturovaného programování a zjištění, že v algoritmu se nachází jen dva základní typy struktur [17][36].

sekvence (posloupnost operací) - tedy postupné provedení operací na stejné úrovni vnoření. Přitom je nevýznamné, zda jsou operace složeny z vnořené struktury dílčích operací či nikoliv.

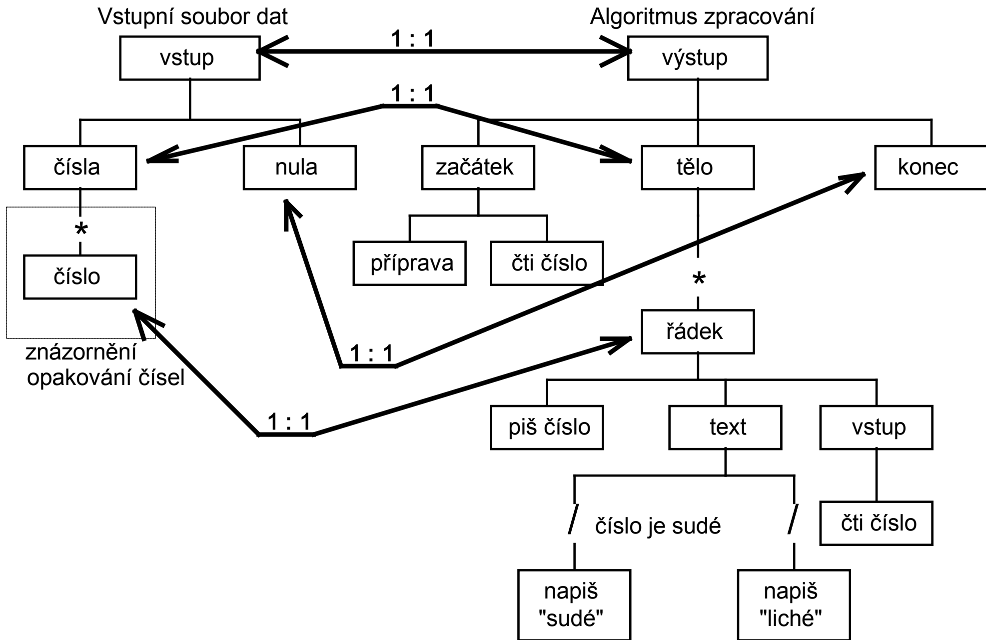
selekce (větvení) - provede se vybraná operace podle vyhodnocené podmínky.

Pokud vám v tomto výčtu schází **opakování** (cyklus, iterace), pak vězte, že je chápáno pouze jako zvláštní případ sekvence.

Kromě výrazně jednoduššího pohledu na základní struktury operací se Jacksonova technologie snaží pohlížet stejným způsobem na popis vstupního souboru dat i algoritmus pro jejich zpracování. Mezi těmito popisy jsou navíc jednoznačné vztahy, které usnadňují kontrolu správnosti algoritmu. Strukturogram autora navíc přímo vede k postupnému upřesňování algoritmu od hrubého návrhu až po velmi podrobný popis, odpovídající požadavkům použitého programovacího jazyka. Operace na stejné úrovni vnoření jsou umístěny na společné vodorovné linii. Upřesnění (rozvinutí) operace je znázorněno posloupností operací, které ji tvoří, zobrazenou níže. Tím se strukturogram liší od dříve uvedených popisů algoritmů, které čteme shora dolů. Strukturogram je nutno číst v zásadě zleva doprava, směr shora dolů udává postupné zpřesňování popisu činností.

Zápis strukturogramu, který budeme v dalším textu používat, vychází z Jacksonova zápisu, ale poněkud jej upravuje.

Příklad 2.10:



Jak vidíme z ukázky, algoritmus zpracování je chápán pouze jako popis převodu vstupního souboru dat na výstupní soubor.

Jacksonova technologie je aplikována v programu SGP [50], který umožňuje zápis algoritmu až do podrobností nutných pro vytvoření zdrojového textu pro překlad ve zvoleném programovacím jazyku. Jeho výhodou je skutečnost, že vůbec nepřipouští předčasné ukončení cyklu a skok do jiného místa algoritmu. Pokud je to potřeba, je možno jedině násilně opustit celý algoritmus (EXIT PROCEDURE). Tím je systém snadno aplikovatelný zejména pro jazyky podporující strukturovaný přístup, jako **PASCAL** (jazyk PASCAL vytvořil Niklaus Wirth původně pro výuku programování, velmi rychle si však získal širokou popularitu, jeho název je zkratkou z Programme Apliqué á la Sélection et la Compilation Automatique de la Littérature, která dává jméno francouzského filozofa Blaise Pascala 1623 * 1662) nebo C (univerzální programovací jazyk původně navržený Denisem Ritchiem v roce 1975 pro operační systém UNIX, dnes je známa spíše jeho nadmnožina C++ navržená Bjarnem Stroustrupem v Bellových laboratořích r. 1983).

Jako výhody strukturogramů jsou uváděny především snadná údržba algoritmů, a to i jiných autorů a přehledná dokumentace algoritmu. Zatím nenacházejí velké rozšíření, nejspíš z důvodů konzervativního přístupu programátorů, kteří si již zvykli využívat jiný způsob popisu algoritmu.

2.1.7 Pseudojazyk

Použití **pseudojazyka** je vlastně návratem k textovému popisu algoritmu, ale na vyšší úrovni. Principy strukturovaného programování jsou zajištěny tím, že vnořenou strukturu algoritmu znázorňujeme odsazením jejího zápisu doprava. Každá dílčí struktura vždy začíná a končí speciálními příkazy. Jejich znění je zvoleno tak, aby se co nejvíce blížilo co největšímu počtu programovacích jazyků. Hlavní inspirací přitom byl **Strukturovaný BASIC**, který rozšiřuje známý programovací jazyk BASIC o programové struktury, které odpovídají zásadám **strukturovaného programování** a přitom zachovávají jednoduchost přístupu. V současné době (pět let do konce tisíciletí) získává velkou popularitu, zejména u produktů v prostředí OS-Windows. Pseudojazyk budeme v dalším textu využívat i my.

Příklad 2.11:

```
čti číslo
while číslo <> 0
  tiskni číslo
  if číslo je sudé
        tiskni "sudé"
  else
        tiskni "liché"
  end if
  čti číslo
end while
```

2.2 ZÁKLADNÍ STRUKTURY ALGORITMU

V této kapitole uvádíme srovnání základních struktur a jejich vyjádření vybranými způsoby zápisu.

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Programový celek (rutina, podrogram, procedura a pod.) - definice			

`název (parametry)`
.
.
.
`end název`
	začátek
	⋮
	konec
			jeden celý strukturogram

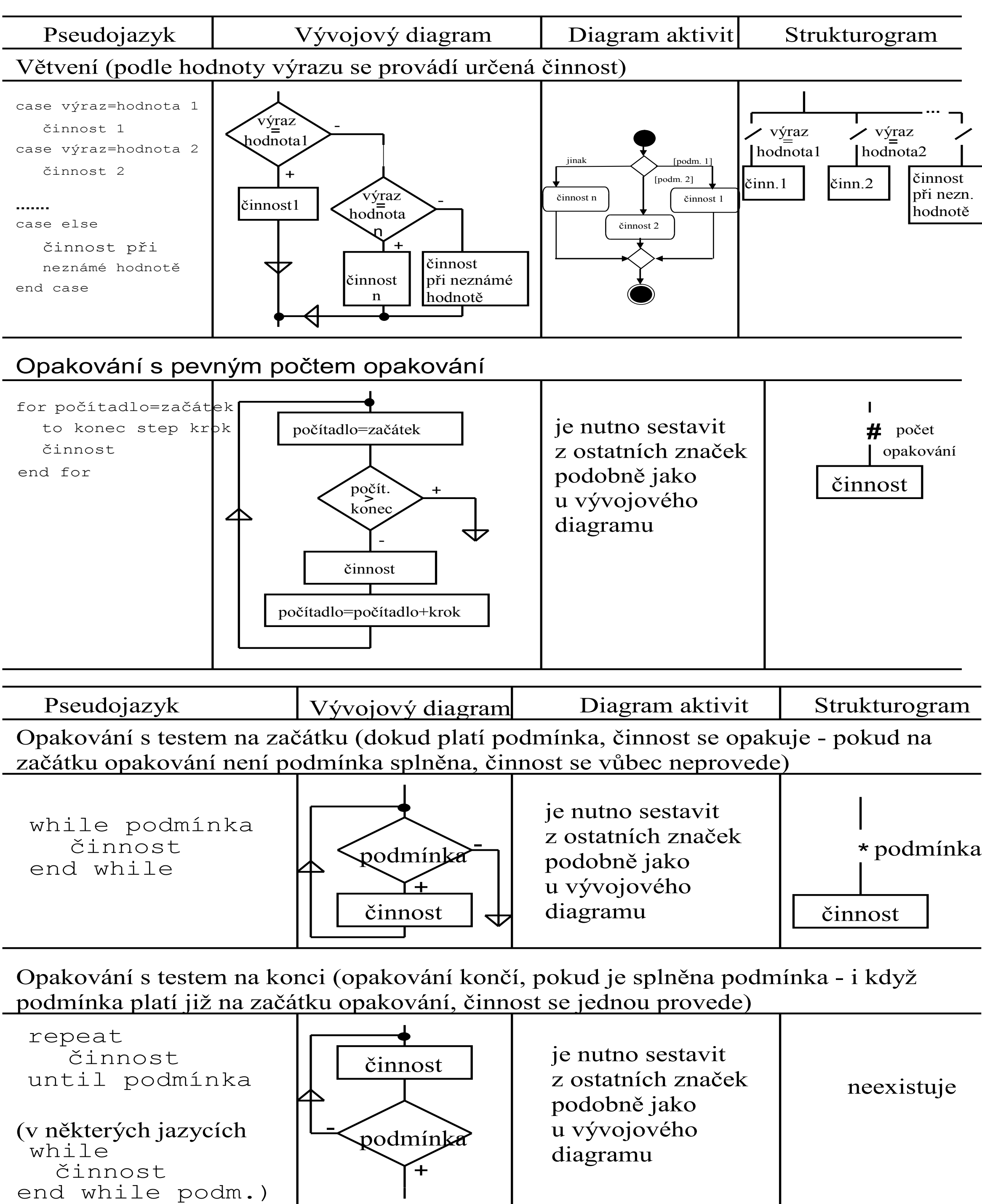
programový celek - použití (volání)			
`název (parametry)`	název(par.)	název (par.)	název(par.)

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Provedení konkrétní činnosti			
`popis činosti`	popis činnosti	popis činnosti	popis činnosti
Podmíněná činnost (provádí se pouze pokud je splněna určitá podmínka)			

`if podmínka`
`podmíněná činnost`
`end if`
	podmínka
	-
	+
	činnost
		smysl má jen rozhodování	podmínka
			činnost

Rozhodování (pokud platí určená podmínka, provede se činnost 1, jinak činnost 2)			

`if podmínka`
`činnost 1`
`else`
`činnost 2`
`end if`
	+ podmínka -
	činnost 1
	činnost 2
		podmínka
		jinak
		[podmínka
		činnost 2
		činnost 1
		sloučení
			podmínka
			činnost 1
			činnost 2



Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Větvení (podle hodnoty výrazu se provádí určená činnost)			

`case výraz=hodnota 1`
`činnost 1`
`case výraz=hodnota 2`
`činnost 2`
....
`case else`
`činnost při neznámé hodnotě`
`end case`
	výraz = hodnota1; -; +; činnost1; výraz = hodnota n; -; +; činnost n; činnost při neznámé hodnotě	jinak; [podm. 1]; [podm. 2]; činnost n; činnost 1; činnost 2	výraz = hodnota1; výraz = hodnota2; ...; činn.1; činn.2; činnost při nezn. hodnotě

Opakování s pevným počtem opakování			

`for počítadlo=začátek to konec step krok`
`činnost`
`end for`
	počítadlo=začátek; počít. > konec; +; -; činnost; počítadlo=počítadlo+krok	je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	# počet opakování; činnost

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Opakování s testem na začátku (dokud platí podmínka, činnost se opakuje - pokud na začátku opakování není podmínka splněna, činnost se vůbec neprovede)			

`while podmínka`
`činnost`
`end while`
	podmínka; -; +; činnost	je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	* podmínka; činnost

Opakování s testem na konci (opakování končí, pokud je splněna podmínka - i když podmínka platí již na začátku opakování, činnost se jednou provede)			

`repeat`
`činnost`
`until podmínka`
 	
(v některých jazycích
`while`
`činnost`
`end while podm.`)
	činnost; podmínka; -; +	je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	neexistuje

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Vstupní nebo výstupní operace			
Jako každá jiná činnost	popis činnosti	jako každá jiná aktivita	Jako každá jiná činnost
Přípravná činnost			
Jako každá jiná činnost	popis činnosti	jako každá jiná aktivita	Jako každá jiná činnost
Spojka (činnost končí v jedné části algoritmu a pokračuje v jiné části)			
Neexistuje	číslo číslo	Neexistuje	Neexistuje

2.3 ČLENĚNÍ ALGORITMU

Po uvedení výčtu základních struktur algoritmů se ještě chvíli zabývejme tvorbou dílčích celků algoritmů. Pro lepší přehlednost algoritmu je vhodné zpracovat algoritmus pro ucelenou část činnosti samostatně. Výsledkem je pak obvykle **rutina** realizující tuto konkrétní činnost (procedura, podprogram aj.).

Jako **rutinu** označujeme programový celek, který je po svém spuštění proveden celý až do konce, při opětovném spuštění jeho činnost začíná opět od začátku. Kromě toho existují programové celky, které provedou určenou činnost a v určitém místě ji ukončí, při dalším spuštění pokračují tam, kde přestaly. Označujeme je jako korutiny, v dalším textu se jimi nebudeme zabývat.

Většina programovacích jazyků, které podporují strukturované programování, odlišuje dva typy rutin:

funkce, které svým názvem vracejí specifikovanou vypočtenou hodnotu,

procedury, které žádnou hodnotu nevracejí.

2.4 PROMĚNNÉ

V algoritmech, které dále uvádíme, budeme pracovat s daty (informací) různých **datových typů**, jak budou popsány v následující kapitole včetně vysvětlení pojmů data a datový typ. Data ukládáme do proměnných, které mají určený datový typ a jejich obsahem je pak konkrétní informace. Základní operací s proměnnou je přiřazení hodnoty proměnné **přiřazovacím příkazem**. Tento příkaz můžeme také chápat jako binární operátor (budeme ho označovat "="). Jeho levý operand je název proměnné, pravý operand operace (výraz) jejíž výsledek se uloží jako obsah této proměnné, např.:

```
p = 1 + 1
Karel = Jméno + Příjmení + Titul
```

V programu a také v proceduře se můžeme setkat s několika typy proměnných:

globální proměnné - existují nezávisle na činnosti procedury, tedy již před jejím spuštěním a také po jejím ukončení. Představují jednu z možností, jak může informace vstoupit do procedury.

lokální proměnné - existují jen po dobu činnosti procedury, vytváří se při spuštění procedury, po jejím ukončení neexistují. Používají se jako pomocné proměnné pro realizaci činnosti procedury.

Pro předání informace proceduře je tedy možno použít jen globální proměnné. Nepříjemným důsledkem jejich použití je skutečnost, že každá změna obsahu globální proměnné zůstává v platnosti po skončení procedury. Tento problém je řešen zavedením pojmu **parametr** procedury. Je to lokální proměnná, která je uvedena v seznamu parametrů za názvem procedury. Při volání procedury současně určíme hodnoty všech jejích parametrů, takže procedura může při každém volání zpracovávat jinou informaci. Rozlišujeme přitom dva způsoby **předávání hodnot parametrů**:

předání **hodnotou** - proměnná má charakter **lokální proměnné**. Po opuštění procedury parametr zaniká. Touto cestou je tedy možno předat informaci do procedury, ale nikdy ne ven. Výhodou je skutečnost, že předáváním obsahu (hodnoty) proměnné je zaručeno, že obsah této proměnné se nezmění. Např.: hledej("Karel"), hledej(Jmeno+Příjmení).

předání **odkazem** - Tento způsob některé programovací jazyky vůbec nepřipouštějí a je nutno ho obcházet předáním hodnoty ukazatele na proměnnou. Předávaná proměnná je ztotožněna s parametrem, takže všechny operace s parametrem se projevují přímo na obsahu předávané proměnné. Výsledek činnosti tedy zůstává i po skončení činnosti procedury. Předání proměnné odkazem budeme zdůrazňovat slovem REF (reference) před jejím názvem v seznamu parametrů. Je zřejmé, že při volání procedury musí být parametru přidělena vždy proměnná, nikdy ne konstanta ani výsledek výrazu. Např.: hledej(Jméno).

Poznámka:

Aby nedošlo k nedorozumění, upozorňujeme, že pojmy globální a lokální proměnná jsou chápány z hlediska celého programu. Odkazem do procedury může být předána proměnná, která je v nadřazené proceduře lokální. Např.:

```
najdi (jméno)
  deklarace lokální proměnné počet
  hledej (jméno, počet)
  napiš jméno a počet výskytů.
end najdi

hledej (jm,REF poč)
  pro určené jméno (parametr jm) najde počet výskytů a
uloží do proměnné poč
end hledej.
```

Z hlediska uložení v paměti má rozlišení globálních a lokálních proměnných také velký význam. Blíže to bude uvedeno na začátku kapitoly věnované dynamickým datovým strukturám.

2.5 HODNOCENÍ ALGORITMU

Ke srovnání dvou algoritmů řešení stejného problému potřebujeme nějakým způsobem vyjádřit jejich výkonnost. Nejčastěji se k tomuto účelu využívá vyjádření **složitosti algoritmu**. [43][62]. Velmi zjednodušeně můžeme říci, že složitost algoritmu (**časová složitost algoritmu**) udává, jak roste počet operací algoritmu v závislosti na růstu množství

zpracovaných dat. Podobně se můžeme setkat také s **paměťovou složitostí algoritmu**, kterou se dále zabývat nebudeme.

Při vyjádření složitosti algoritmu můžeme vyjít od binární (bitové) složitosti operací. Za bitovou operaci přitom považujeme sčítání dvou jednobitových čísel (ať již s přenosem, nebo bez přenosu).

Sčítání dvou k-bitových čísel má binární složitost úměrnou délce k. Například sečtení následujících dvou čísel má binární složitost 8 :

```
   10101100        172
 + 11101010      + 234
 ----------      -----
  110010110        406
```

Podobně je tomu u jiných operací. Násobení dvou k a j bitových čísel představuje v nejhorším případě sčítání j sčítanců, po doplnění nulami dlouhých nejvýše $(k + j - 1)$ bitů. Pro dosažení výsledku musíme sčítat první s druhým, výsledek se třetím sčítancem atd. Tedy celkem nejvýše **(j - 1)** součtů **(k + j – 1)**-místných čísel. Pro odhad binární složitosti pak zjednodušíme :

$(j - 1)*(k + j - 1) < j*(k + 1) < 2*k*j.$

Binární složitost násobení je tedy úměrná **2*k*j**.

```
    101101        45 * 13 = 585
      1110
 ----------
    101101
    101101
    101101
 ----------
 1001001001
```

Pro desítkovou soustavu platí, že n-místné dekadické číslo bude mít nejvýše k binárních znaků, kde je $\mathbf{k \le \log_2(n) + 1}$. Sčítání dvou takových čísel má tedy binární složitost úměrnou hodnotě $\mathbf{\log_2(n) + 1}$.

Násobení n a m místného dekadického čísla má binární složitost úměrnou:

$$2*k*j < 2*[\log_2(n) + 1]*[\log_2(m) + 1] =$$

$$= 2*[\log_2(n)*\log_2(m) + \log_2(n*m) + 1] < 4*\log_2(n)*\log_2(m)$$

Binární složitost budeme zapisovat zkráceně: **f(n) = O(g(n))**, jestliže pro funkce **f** a **g** definované na množině přirozených čísel existuje konečná limita:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \tag{2.1}$$

Můžeme tedy zapsat:

1. **n**-místné dekadické číslo má $\mathbf{O(\log_2(n))}$ binárních číslic,
2. součet dvou **n**-místných dekadických čísel reprezentuje **nejvýše** $\mathbf{O(\log_2(n))}$ bitových operací,
3. součin (podíl) **n** a **m** místného dekadického čísla vyžaduje $\mathbf{O(\log_2(n)*\log_2(m))}$ bitových operací.

U algoritmů jsou významné některé typy složitostí. Především je to polynomiální složitost. Pojem **polynomiální složitost** budeme používat u operací, u kterých výpočet funkce **f(n)** dekadického čísla o **n** místech, které má k dvojkových cifer vyžaduje:

$$O(k^d) \text{ bitových operací,} \tag{2.2}$$

kde je *d* přirozené číslo.

Operace + - * /, tedy mají polynomiální složitost: pro + a - je $d = 1$,

pro * a / je $d = 2$.

Problémy, které jsou řešitelné pomocí algoritmů s polynomiální složitostí označujeme jako problémy třídy **složitosti P** nebo také **P-problémy** (případně PTIME) a považujeme je za efektivně řešitelné, nebo také řešitelné v polynomiálním čase.

Výpočet čísla **n! = 1*2*....*n** (faktoriál) má při použití postupného násobení binární složitost

$$O(n^2 * \log_2^2(n)) = O(2^{2k} * k^2) \tag{2.3}$$

Tento algoritmus tedy má **nepolynomiální složitost**. To mimo jiné znamená, že jeho složitost s rostoucím n roste velmi rychle. Takové algoritmy jsou v praxi velmi nepříjemné, vyžadují pro svoji realizaci velký výpočetní výkon. U problémů, které vyžadují k řešení algoritmy s nepolynomiální složitostí, hovoříme o **třídě složitosti NP** nebo také o NP-problémech (případně také exponenciálních problémech), jestliže správnost nalezeného řešení je možno ověřit v polynomiálním čase:

Speciálním případem jsou situace, kdy není možno ani ověřit správnost nalezeného řešení pomocí algoritmu s polynomiální složitostí. **Pak hovoříme o NP-úplných problémech**.

Známý **Eukleidův algoritmus** pro nalezení největšího společného dělitele celých čísel **(a, b)** má bitovou složitost $\mathbf{O(\log_2^3(a))}$ **pro a > b**. Můžeme ho zapsat např.:

```
nejvetsi_delitel(a, b, REF d)
  a = ABS(a)
  b = ABS(b)
  while a > 0
        x = a
        a = b mod a
        b = x
  end while
  d := b;
end nejvetsi_delitel
```

Stejnou bitovou složitost má nalezení čísel u, v splňujících rovnost: **u*a + v*b = (a, b)**, nebo algoritmus na zjištění nesoudělnosti čísel, když je **(a, b) = 1**.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ĺ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ĺ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5* – Uživatelská příručka. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

3. LEKCE – UKLÁDÁNÍ INFORMACE

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

3 UKLÁDÁNÍ INFORMACE ... 3

3.1 Jednoduché datové typy ... 4

3.1.1 Logická informace ... 5

3.1.2 Celá čísla ... 6

3.1.3 Reálná čísla ... 8

3.1.4 Přímý kód ... 9

3.1.5 Textová informace ... 14

3.1.6 Ukazatel ... 15

3.2 Složené datové typy ... 15

3.2.1 Pole ... 15

3.2.2 Struktura ... 15

3.2.3 Množina ... 16

3.2.4 Objekt ... 16

3.2.5 Soubor ... 17

POUŽITÁ LITERATURA ... 19

3 UKLÁDÁNÍ INFORMACE

OBSAH KAPITOLY:

Jednoduché datové typy

Složené datové typy

MOTIVACE:

Pro efektivní činnost programů je nutné ukládat co nejlépe vstupní, pomocná i výstupní data. K tomu slouží různé datové typy. Pro jejich správné používání je potřeba znát jejich možnosti a zejména jejich omezení. Z nich nejvýznamnější je pak otázka přesnosti uložení čísel a kumulace takto vzniklých chyb ve výpočtech. Při sestavování algoritmů je třeba tyto skutečnosti brát v úvahu.

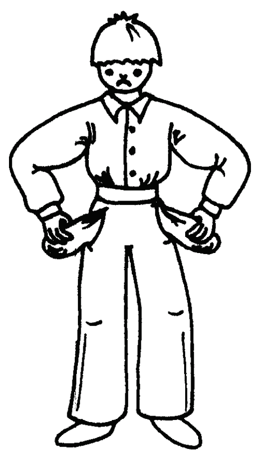
CÍL:

Seznámit se s používanými datovými formáty, jednoduchými i složenými. Umět správně používat datové formáty pro ukládání dat textových, číselných, datumů a časů, logických údajů i množin hodnot. U číselných datových formátů umět správně používat formáty pro celá čísla, i desetinná čísla jak s pevnou tak s pohyblivou řádovou čárkou. Znát jejich rozsahy a zejména zaokrouhlovací chyby a jejich akumulaci při výpočtech. Rozumět používání ukazatelů pro realizaci dynamických datových struktur i objektů a souborů. Chápat správně používání složených datových typů, jednorozměrných i vícerozměrných polí a záznamů, i jejich kombinací.

Pro práci s **informací** v počítači je potřeba tuto informaci nějakým vhodným způsobem interpretovat. K tomu využíváme **kódování** informace. Musíme tedy vytvořit vhodný kód, neboli přiřazení **kódových** slov jednotlivým zprávám. Možných zpráv je nepřeberné množství, o pravděpodobnosti jejich výskytu nic nevíme, takže vytvořit **optimální kód** nebude možné. Přesto nechceme zbytečně plýtvat prostorem pro uložení informace. Vytvořit vhodný kód pro uložení jakékoliv informace zřejmě nebude jednoduché.

V praxi se osvědčilo vytvořit speciální kódy pro jednotlivé typy informace, které se často vyskytují. Je zřejmé, že kód pro uložení čísel bude vypadat jinak, než kód pro uložení textů. Postupným vývojem se ustálilo několik základních kódů, které jsou vhodné pro ukládání informace v počítačích. Takto uloženou informaci obvykle označujeme jako **data**.

Pojem **data** můžeme definovat jako informaci, která má určitou vypovídací schopnost, je určitým způsobem uspořádána, seřazena apod. Způsob jejího uložení pak obvykle nazýváme datový typ.



Obrázek 3.1 Ukládání informace

Rozlišujeme **jednoduché datové typy** (primitivní) pro ukládání informace stejného typu (čísla, text apod.). Z nich pak vytváříme **složené datové typy** pro ukládání rozsáhlejší informace. Způsob uložení souhrnné informace o určitém objektu obvykle označujeme jako **datovou strukturu**.

Pokud se tato struktura během zpracování nemění, to znamená, že je vytvořena při spuštění programu a zaujímá stále stejný prostor pro uložení, nazýváme ji **statická datová** struktura. Pokud se tato struktura během zpracování mění (přibývá jejích prvků apod.), nazýváme ji **dynamická datová struktura**.

V následujících kapitolách budou rozebrány některé významné postupy zpracování informace v počítači.

3.1 JEDNODUCHÉ DATOVÉ TYPY

V této kapitole rozebereme typicky používané **jednoduché datové typy** (primitivní, základní). Základní vlastností datového typu je počet hodnot, kterých může nabývat typ T. Nazývá se **kardinalita** typu **T**. Jak uvidíme dále, kardinalita všech datových typů je vždy konečné velikosti, k dispozici máme konec konců paměť omezené velikosti. Co z této skutečnosti plyne, bude podrobně vysvětleno při ukládání reálných čísel.

Datové typy přiřazujeme jednotlivým **proměnným**, se kterými pracujeme v programu. Při práci s proměnnými mohou nastat dva zvláštní případy:

1. Proměnná nebyla **vytvořena**. Tato skutečnost má při manipulaci s proměnnou obvykle za následek havárii programu. Řada programovacích jazyků přitom vůbec neumožňuje vytvořit program, který používá proměnnou, která nebyla vytvořena (FORTRAN, PASCAL, C, C++...).
2. Proměnná byla **vytvořena**, ale nemá přiřazenu žádnou hodnotu. Různé programovací jazyky se v této situaci chovají různě. Některé umožňují implicitní inicializaci proměnných hodnotou 0 nebo speciální hodnotou **NULL** (Access BASIC). Jiné tuto inicializaci neprovádějí pouze u jistých paměťových tříd proměnných (C, C++).

Pokud jazyk nepodporuje implicitní inicializaci, pak má proměnná nějakou náhodnou hodnotu, obvykle určenou tím, co obsahovala paměť v době vytvoření proměnné. Obecně se doporučuje při vytvoření proměnné ihned definovat její obsah (provést inicializaci proměnné).

Pokud jazyk používá speciální hodnotou **NULL**, pak je dobré si uvědomit následující skutečnost. Při operaci, do níž vstoupí alespoň jeden z argumentů hodnoty **NULL**, bude výsledkem operace rovněž hodnota **NULL**.

Dále se již těmito problémy zabývat nebudeme a přejdeme přímo k jednotlivým datovým typům. Pro vytvoření proměnné požadovaného datového typu budeme, tam kde to bude potřebné, používat následující syntaxi:

```
seznam proměnných : datový typ
```

3.1.1 Logická informace

Logická informace nabývá dvou hodnot Pravda/Nepravda, True/False, Yes/No, 0/-1, 1/0. Obvykle jsou definovány následující základní Booleovské operátory. Jejich definice je uvedena v následné tabulce:

- unární not - ¬, negace,

- binární and - $\wedge$, logický součin, konjunkce,

or - $\vee$, logický součet, disjunkce.

Tabulka 3.1 Pravdivostní hodnoty

p	**q**	**p or q**	**p and q**	**not p**
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

Pro porovnání hodnot proměnných slouží binární operátory jejichž výsledkem je rovněž logická informace (označení odpovídá jazykům jako BASIC, PASCAL, aj.):

=	- rovnost, shoda obsahu, (jazyk C používá = =),
<	- je menší,
>	- je větší,
>=	- je větší nebo rovno (≥),
<=	- je menší nebo rovno (≤),
<>	- není rovno (≠), (jazyk C používá ! =).

3.1.2 Celá čísla

Pro ukládání celých čísel se používá několik datových typů, které se liší velikostí zabírané paměti a tím i **kardinalitou**. Tabulka č. 3.2 uvádí vybrané datové typy.

V tabulce č.3.2 jsou uvedeny rozsahy jednotlivých datových typů pro různé typy operačních systémů. U šestnáctibitových OS (DOS, Windows) jsou kardinality menší než u třicetidvoubitových OS (Windows NT, Windows 95).

Čísla ukládáme v binární podobě. Vyjadřujeme je přitom v pozičním tvaru pomocí základu číselné soustavy polynomem, který ukládáme po jednotlivých bitech.

$$39 \Rightarrow 3\cdot 10^1 + 9\cdot 10^0 \Rightarrow 1\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0$$

⇓

0	0	1	0	0	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Tabulka 3.2 Vybrané celočíselné datové typy

jazyk C, C++	jazyk PASCAL	v šestnáctibitovém OS	ve třicetidvoubitovém OS
char	-----	-128 ÷ 127	-128 ÷ 127
unsigned char	Byte	0 ÷ 255	0 ÷ 255
short	ShortInt	-128 ÷ 127	-32 768 ÷ 32 767
unsigned short	-----	0 ÷ 255	0 ÷ 65 535
int	Integer	-32 768 ÷ 32 767	-2147483648 ÷ 2147483647
unsigned int	Word	0 ÷ 65 535	0 ÷ 4 294 967 295
long	LongInt	-2147483648 ÷ 2147483647	-2147483648 ÷ 2147483647
unsigned long	-----	0 ÷ 4 294 967 295	0 ÷ 4 294 967 295

Uložení kladného čísla nebo nuly je tedy velmi jednoduché. Potíže nastávají až při ukládání záporných čísel. Používají se tři způsoby kódování celých čísel (**číselné kódy**), které se liší právě ukládáním záporných čísel.

3.1.2.1 Přímý kód

Přímý kód ukládá absolutní hodnotu čísla spolu se **znaménkovým bitem**, který určuje, zda se jedná o číslo kladné, či záporné.

39	⇒	0	0	1	0	0	1	1	1
- 39	⇒	1	0	1	0	0	1	1	1

Při použití tohoto kódu musíme rozlišovat sčítání dvou kladných čísel, jednoho kladného a jednoho záporného atd. Znaménkový bit se operace neúčastní, pouze rozhoduje o druhu použité operace.

3.1.2.2 Doplňkový kód

Záporné číslo v **doplňkovém kódu** ukládáme zvětšené o $\mathbf{2^n}$, kde **n** je počet bitů datového typu. Např. pro uložení čísla na 8 bitů:

39	⇒	0	0	1	0	0	1	1	1
- 39	⇒ 2^8-39	1	1	0	1	1	0	0	1

Při sčítání dvou čísel sčítáme čísla včetně **znaménkových bitů**. Pokud dojde k přenosu na nejvyšším místě, ignoruje se. Např. sečtení čísel 39 a -39 v doplňkovém kódu:

```
          znaménko
 39=        0|0 1 0 0 1 1 1
-39=        1|1 0 1 1 0 0 1
         ---------------------
          1|0|0 0 0 0 0 0 0
   přenos|
```

Jiným problémem je **přetečení rozsahu** zobrazitelných čísel (přeplnění). Ukažme si ho na součtu čísel 127 a 1 uložených v doplňkovém kódu v 1 Byte :

```
          znaménko
127=        0|1 1 1 1 1 1 1
  1=        0|0 0 0 0 0 0 1
         ---------------------
          0|1|0 0 0 0 0 0 0
   přenos|
```

Jak vidíme, sečtením dvou kladných čísel jsme získali číslo záporné. Pokud chceme při sčítání čísel kontrolovat, zda nedošlo k přetečení rozsahu, můžeme využít zdvojení znaménkového bitu. Ukažme si to na dříve uvedených příkladech:

```
     znaménko                                 znaménko
 39=  0 0|0 1 0 0 1 1 1                 127=  0 0|1 1 1 1 1 1 1
-39=  1 1|1 0 1 1 0 0 1                   1=  0 0|0 0 0 0 0 0 1
     -------------------                      -------------------
   1|0 0|0 0 0 0 0 0 0                      0|0 1|0 0 0 0 0 0 0
přenos| stejná hodnota znaménka          přenos| nestejná hodnota znaménka
        ukazuje na správný výsledek              ukazuje na přetečení rozsahu
```

Jednoduchost aritmetických operací s čísly v doplňkovém kódu je vyvážena složitějším převodem záporných čísel. Provádí se tak, že u kladného čísla zaměníme hodnotu všech bitů a

k poslednímu přičteme 1. Tento kód je dnes standardně využíván (např. v jazyku PASCAL, C).

3.1.2.3 Inverzní kód

U **inverzního kódu** je záporné číslo vyjádřeno jako inverze (změna hodnot všech bitů) kladného čísla stejné absolutní hodnoty. Záporné číslo pro uložení zvětšíme o $\mathbf{2^n - 1}$. Opět pracujeme se všemi bity čísla.

39 ⇒		0	0	1	0	0	1	1	1
- 39 ⇒	2^8-1-39	1	1	0	1	1	0	0	0

Jak vidíme, záporné číslo v inverzním kódu je vždy o 1 větší než v doplňkovém kódu. Tuto skutečnost musíme uvažovat při aritmetických operacích.

Poznámka:

Pozorný čtenář si jistě uvědomil, že v inverzním kódu existují dva způsoby uložení hodnoty 0:

3.1.2.4 Operace s celými čísly

S celými čísly je možno provádět všechny běžné operace

+ - sčítání,

- - odečítání,

* - násobení,

/ - dělení, jeho výsledek závisí na programovacím jazyku. V jazyku C, C++ bude výsledkem opět celé číslo, v jiných jazycích je však výsledkem **reálné číslo**,

div - celočíselné dělení, např. **22 div 5 = 4**,

mod - zbytek po celočíselném dělení, např. **22 mod 5 = 2**.

Některé programovací jazyky umožňují provádět operace s bity celých čísel, jako jsou logické operace s jednotlivými bity, nebo **rotace bitů**:

- doleva (**nahoru**) - posunutí bitů na vyšší pozici, nejvyšší bit se ztrácí, nebo přesouvá na nejnižší pozici,

- doprava (**dolů**) - posunutí bitů na nižší pozici, nejnižší bit se ztrácí, nebo přesouvá na nejvyšší pozici.

3.1.3 Reálná čísla

Množina **reálných čísel** je sjednocením množin:

- **racionálních čísel** - ty je možno vyjádřit jako podíl dvou přirozených čísel a znaménkem. Tuto vlastnost využívaly některé speciální číselné kódy.

- **iracionálních čísel** - ty není možno vyjádřit jako podíl dvou přirozených čísel, patří sem např.: **π, e**.

Při ukládání reálných čísel si musíme uvědomit, že máme k dispozici jen omezený paměťový prostor, proto zejména iracionální čísla není možno uložit celá. Při ukládání reálných čísel existují dva přístupy.

- **kódy s pevnou řádovou čárkou**, kdy pozice řádové čárky je stále stejná. Nevýhodou je omezení rozsahu uložitelných čísel, výhodou vyšší rychlost operace s čísly.

	zn.	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}		
2,625 ⇒	0	0	1	0	1	0	1	0	=	2,625
2,6 ⇒	0	0	1	0	1	0	0	1	=	2,5625

poloha řádové čárky

Tyto kódy se dnes prakticky neužívají.

- **kódy s proměnnou řádovou čárkou**, kdy provádíme **normování** čísla do podoby:

$$m \cdot 2^e, \tag{3.1}$$

kde je m - mantisa, obvykle je $|m| < 1$,

e - exponent, který je celým číslem.

Z ukázky je zřejmé, že o **přesnosti** uloženého čísla rozhoduje **mantisa** (obvykle je kódována v přímém kódu), o **rozsahu** čísla rozhoduje **exponent** (obvykle je kódován tak, že je nutno od uložené hodnoty nutno odečíst $\mathbf{2^{n-1}}$, abychom získali jeho skutečnou hodnotu, přitom n je počet bitů pro uložení exponentu).

Exponent je celé číslo, o jejich kódování již byla řeč. Číselné kódy pro ukládání desetinného čísla jsou velmi podobné a mají také obdobné vlastnosti.

3.1.4 Přímý kód

Přímý kód můžeme definovat vztahem:

$$x_{př} = \begin{cases} x & x \geq 0 \\ 1 - x & x < 0 \end{cases} \quad |x| < 1, \tag{3.2}$$

Pokud čtenáře udivuje vztah pro vyjádření záporného čísla, pak je potřeba si uvědomit, že platí:

$$1\text{-}x = 1 + |x| \quad pro \quad |x| < 1 \tag{3.3}$$

Z ukázky je zřejmé, že bit před desetinnou čárkou má význam znaménka.

Příklad:

$$0{,}65625_{pø} \Rightarrow 0{,}10101$$

$$-0{,}65625_{pø} \Rightarrow 1{,}10101$$

Nula má v přímém kódu dvě vyjádření 1,0000 a 0,0000.

3.1.4.1 Doplňkový kód

Doplňkový kód můžeme definovat vztahem:

$$x_{dop} = \begin{cases} x & x \geq 0 \\ 2 + x & x < 0 \end{cases} \quad -1 \leq x < 1, \tag{3.4}$$

Kladná čísla se zobrazují přímo, záporná jako **doplněk do** čísla 2 (základu použité soustavy). Nula má přitom jediné vyjádření, neboť platí:

$$-1_{dop} = 1{,}0000$$

$$0{,}65625_{dop} = 0{,}10101$$

$$-0{,}65625_{dop} = 1{,}01011$$

3.1.4.2 Inverzní kód

Inverzní kód můžeme definovat vztahem:

$$x_{př} = \begin{cases} x & x \geq 0 \\ 2 - 2^{-n} + x & x < 0 \end{cases} \quad |x| < 1, \tag{3.5}$$

kde je n - řád poslední zobrazené dvojkové cifry (počet bitů pro uložení čísla bez znaménka).

Příklad uložení čísel v doplňkovém kódu:

$$0{,}65625_{\mathrm{inv}} = 0{,}10101$$

$$-0{,}65625_{\mathrm{inv}} = 1{,}01010$$

3.1.4.3 Operace s reálnými čísly

S reálnými čísly je možno provádět základní operace, jako

+	-	sčítání
-	-	odečítání
*	-	násobení
/	-	dělení.

Při jejich provádění si musíme uvědomit skutečnost, že reálná čísla lze uložit jen na omezený počet **platných cifer**. Jejich počet je určen počtem bitů pro uložení mantisy. **Přesnost ukládání čísel** vyjadřujeme pojmem **počítačové epsilon** (ε). Je to nejmenší číslo zobrazitelné v daném číselném kódu, pro které platí:

$$1{,}0 + \varepsilon > 1{,}0. \tag{3.6}$$

Pokud neznáme přesně použitý **číselný kód**, můžeme určit $\boldsymbol{\varepsilon}^*$, splňující podmínku, že je to největší zobrazitelné číslo, pro které platí:

$$1{,}0 + \boldsymbol{\varepsilon}^* = 1{,}0. \tag{3.7}$$

Pomocí jednoduchého postupu s použitím proměnných **e** a **e1** rozebíraného číselného kódu zjistíme $\boldsymbol{\varepsilon}^*$ následujícím algoritmem:

```
e = 1
e1 = 1+e
while e1 > 1
  e = 0.5 *e
  e1 = 1 + e
end while
tiskni e
```

Z dosavadních znalostí snadno určíme, že $\boldsymbol{\varepsilon} = \mathbf{2}\boldsymbol{\varepsilon}^*$.

Příklad:

V programovacím jazyku PASCAL zaujímá reálné číslo (datový typ real) 6 Byte. Testem zjistíme, že

$$\varepsilon^* = 9{,}0949470177 * 10^{-13} = 2^{-40},$$

$$\varepsilon = 1{,}8189894035 * 10^{-12} = 2^{-39}.$$

Aby bylo možno uložit číslo **1 + ε**, tedy konkrétně číslo $2^0 + 2^{-39}$, potřebujeme k tomu 40 bitů, jeden bit zabere znaménko, na exponent tedy zbývá 7 bitů. Z toho usuzujeme, že exponent

tedy nabývá rozsahu -64 ÷ 63, neboli číslo 2^{64} by již nemělo být možno uložit (to je přibližně $1{,}8.10^{19}$). Můžeme si však ověřit, že uložíme i větší čísla. Kde je v našich propočtech chyba? Odpověď je jednoduchá – nikde. Celý problém je spojen s využitím následující úvahy. Díky **normování** mantisy je první uložený bit vždy roven jedné. Takže tento bit vůbec neukládáme (říkáme mu **skrytý bit**). Mantisa pak zabírá 39 bitů + znaménko, na exponent zbývá 8 bitů. Nesmíme přitom opomenout uložení čísla 0. To je uloženo pomocí samých nul, tedy včetně exponentu. U ostatních čísel je exponent uložen zvětšen o 128 (27).

Konkrétně tedy např.:

0,1 = 01001100 11001100 11001100 11001100 11001101 01111011

0,5 = 00000000 00000000 00000000 00000000 00000000 10000000

1,0 = 00000000 00000000 00000000 00000000 00000000 10000001

1,5 = 01000000 00000000 00000000 00000000 00000000 10000001

Z ukázky je zřejmé, že zatímco třeba číslo 0,5 je uloženo přesně, číslo 0,1 nikoliv, tedy již při jeho ukládání vzniká **zaokrouhlovací chyba**, která se pak jeho opakovaným přičítáním akumuluje. Proto není divu, že následující algoritmus neskončí pro hodnotu 10 000, ale bude vesele pokračovat dále:

```
x = 0
repeat
  provedení potřebné činnosti
  x = x + 0,1
until x = 10 000
```

Pokud necháme do proměnné x 100 000 krát přičíst 0,1 zjistíme, že bude mít hodnotu pouze 9999,9999784. Nejlepšího výsledku dosáhneme následující úpravou algoritmu:

```
x = -0,1
repeat
  x = x + 0,1
  provedení potřebné činnosti
until x >= 10 000
```

Jistě nás nepřekvapí, že činnost uvnitř smyčky proběhla celkem 100 002 krát. Po skončení algoritmu má proměnná x hodnotu 10 000,099978. Pokud se nám to nelíbí, můžeme využít jako počítadlo pro opakování proměnnou **celočíselného typu**:

```
for i = 0 to 100 000 step 1
  x = i*0,1
  provedení potřebné činnosti
end for
```

Pokud se proměnná x nezvětšuje vždy o stejnou hodnotu, pak nelze využít popsaný postup. V záloze však máme jiný postup. Nazývá se **Kahanův trik** (vymyslel ho již v roce 1965 W. Kahan). Vychází ze zjištění, že při výpočtech na počítači neplatí základní matematické zákony (distributivní a asociativní zákon), pak se může stát že:

$$(a + b) + c \neq a + (b + c)$$

Např. při výpočtech s čísly na 7 platných cifer získáme:

A = 0,1234567

B = 2381,325

A+B= 2381,448

Přitom se ztratila informace 0,0004567, neboť při sčítání musely být exponenty obou čísel stejné. Vhodným postupem jsme schopni tuto ztracenou informaci získat. Pro **B > A** platí:

B =	2381,325
-(A+B) =	- 2381,448
B -(A+B) =	- 0,123
A =	0,1234567
(B-(A+B))+A =	0,0004567

Upozorněme ještě, že pro rozebíraný případ, kdy **B > A**, nám vyjde **(A-(A+B))+B = 0**. Pořadí operandů tedy není zvoleno náhodně.

Uplatněním Kahanova triku získáme algoritmus:

```
x = 0 c = 0
for i=1 to 100000 step 1
  y = 0,1 + c
  t = x + y
  c = (x - t) + y
  x = t
end for
```

Výsledkem bude hodnota x = 10000,000000. Přesněji to již vskutku nejde.

Pokud využijeme informaci o ukládání reálných čísel, pak se budeme snažit přičítat hodnoty, které jsou uloženy přesně, např. místo 0,1 použijeme $0,109375 = 2^{-4} + 2^{-5} + 2^{-6}$. Pokud je potřeba tuto hodnotu měnit, pak nejlépe násobením, či dělením dvěma. Způsobí totiž změnu exponentu o 1, mantisa a tedy ani přesnost uložení čísla se nezmění. Nesmíme ale zapomenout na další zdroj nepřesnosti. Tím je odříznutí desetinných míst při zarovnání sčítanců na stejnou velikost exponentu. Tento problém se vyskytuje zejména při součtu řad čísel.

Obdobným problémem je testování **rovnosti dvou reálných čísel**, zejména pokud jejich hodnoty byly vypočteny. Je zřejmé že přímé srovnání jejich hodnot použít nelze.

První, co začátečník obvykle udělá je určení, zda jsou si testovaná čísla "hodně blízká", např. určením, zda absolutní hodnota rozdílu čísel je velmi malé číslo, např. :

```
x = -0,1
repeat
  x = x + 0,1
  if ABS(x - 1000) < 10-26
        hlášení shody
  end if
until x >= 10 000
```

Shoda hodnot však nebude hlášena. Z přesnosti ukládaných čísel (na asi 11 **platných cifer**) plyne, že shoda bude hlášena až při nastavení "malého čísla" na hodnotu asi 0,0000011. Jenže jeho hodnota závisí na hodnotě testované, nelze tedy najít všeobecně platnou srovnávací hodnotu.

Řešení problému je přitom nasnadě. Řešitel může s úspěchem použít právě krok řešení, např. úpravou testu :

```
  if ABS(x - 1000) < krok řešení/2
        hlášení shody
  end if
```

Pokud se však krok řešení mění, může vzniknout jiný problém, a to případ, kdy se krok řešení zmenší natolik, že jeho přičtení k aktuální hodnotě proměnné x její hodnotu nezmění.

Další problém spojený s **reálnými čísly** je nestejnosměrnost pokrytí číselné osy zobrazitelnými čísly. Zkusme si tuto skutečnost dokumentovat na datovém typu, kde mantisa v přímém kódu a se skrytým bitem zaujímá 5 bitů a exponent 3 bity. Je možno zobrazit tato čísla:

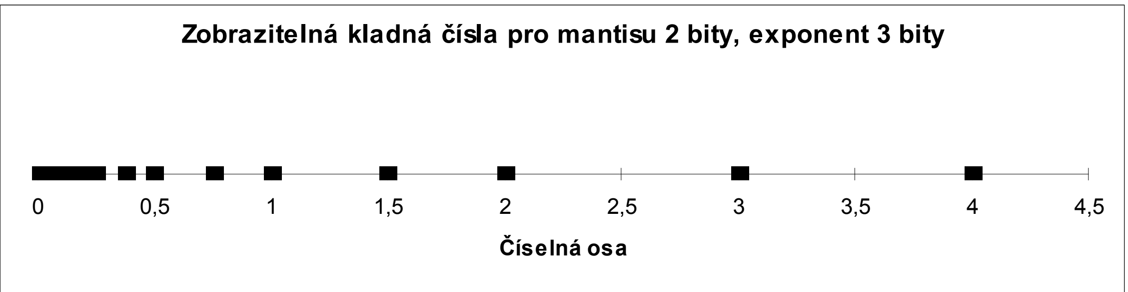
Tabulka 3.3 Zobrazitelná čísla pro mantisu 4bity (+ skrytý bit) a exponent 3 bity

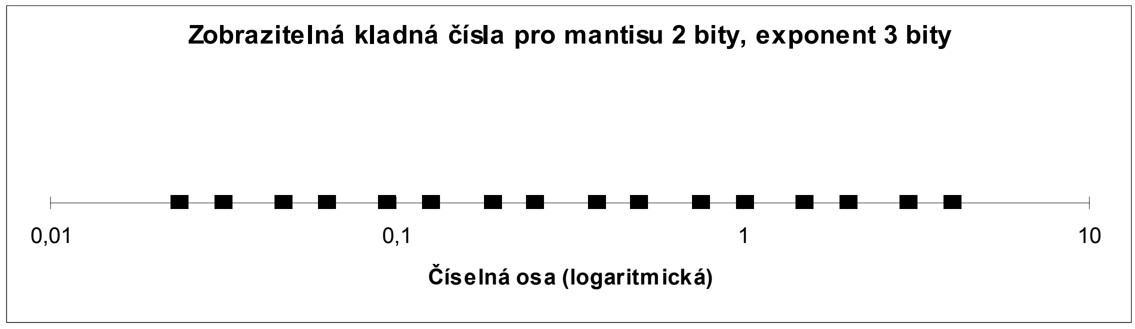
	Exponent (nahoře bitové vyjádření, dole číselné vyjádření)							
	000	001	010	011	100	101	110	111
Mantisa	-4	-3	-2	-1	0	1	2	3
0000	0	0.0625	0.125	0.25	0.5	1	2	4
0001	0.033203	0.066406	0.132813	0.265625	0.53125	1.0625	2.125	4.25
0010	0.035156	0.070313	0.140625	0.28125	0.5625	1.125	2.25	4.5
0011	0.037109	0.074219	0.148438	0.296875	0.59375	1.1875	2.375	4.75
0100	0.039063	0.078125	0.15625	0.3125	0.625	1.25	2.5	5
0101	0.041016	0.082031	0.164063	0.328125	0.65625	1.3125	2.625	5.25
0110	0.042988	0.085975	0.17195	0.3439	0.6878	1.3756	2.7512	5.5024
0111	0.044922	0.089844	0.179688	0.359375	0.71875	1.4375	2.875	5.75
1000	0.046875	0.09375	0.1875	0.375	0.75	1.5	3	6
1001	0.048828	0.097656	0.195313	0.390625	0.78125	1.5625	3.125	6.25
1010	0.050781	0.101563	0.203125	0.40625	0.8125	1.625	3.25	6.5
1011	0.052734	0.105469	0.210938	0.421875	0.84375	1.6875	3.375	6.75
1100	0.054688	0.109375	0.21875	0.4375	0.875	1.75	3.5	7
1101	0.056641	0.113281	0.226563	0.453125	0.90625	1.8125	3.625	7.25
1110	0.058594	0.117188	0.234375	0.46875	0.9375	1.875	3.75	7.5
1111	0.060547	0.121094	0.242188	0.484375	0.96875	1.9375	3.875	7.75

Pokud nám výčet čísel není zcela jasný, můžeme vypsat zobrazitelná kladná čísla pro mantisu 2 bity (bez skrytého bitu) a exponent 3 bity. Všimněte si zejména skutečnosti, že nepoužití skrytého bitu vede k tomu, že řada čísel je zobrazitelná více způsoby. Celkem je takto možno uložit 17 kladných čísel + nulu.

Tabulka 3.4 Zobrazitelná kladná čísla pro mantisu 2 bity a exponent 3 bity

	Exponent (nahoře bitové vyjádření, dole číselné vyjádření)							
	000	001	010	011	100	101	110	111
Mantisa	-4	-3	-2	-1	0	1	2	3
00	0	0	0	0	0	0	0	0
01	0.015625	0.03125	0.0625	0.125	0.25	0.5	1	2
10	0.03125	0.0625	0.125	0.25	0.5	1	2	4
11	0.046875	0.09375	0.1875	0.375	0.75	1.5	3	6



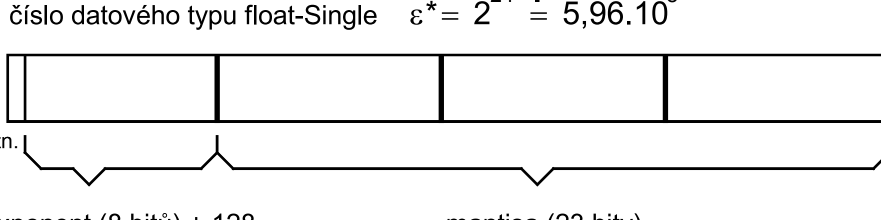


Na číselné ose je dobře patrné, že kolem nuly je soustředěna většina čísel, s rostoucí hodnotou se vzdálenost mezi zobrazitelnými čísly zvětšuje.

Tabulka 3.5 Vybrané neceločíselné datové typy

jazyk C, C++	jazyk PASCAL	rozsah	délka	des. míst
float	Single	$\pm 3{,}4.10^{\pm 38}$	4 Byte	7
-----	Real	$\pm 1{,}2.10^{\pm 39}$	6 Byte	11
double	Double	$\pm 1{,}797.10^{\pm 308}$	8 Byte	15
long double	Extended	$\pm 1{,}2.10^{\pm 4932}$	10 Byte	19

číslo datového typu float-Single $\varepsilon^* = 2^{-24} \doteq 5{,}96.10^{-8}$



3.1.5 Textová informace

Základním prvkem textu je **znak** *(character)*. Pro kódování znaků se zpočátku užíval pětibitový kód (32 znaků), nejvíce se však rozšířil kód definovaný Mezinárodní organizací pro standardizaci (ISO), zejména jeho americká verze **ASCII** – Americký standardní kód pro výměnu informace. Kód je sedmibitový, obsahuje 33 řídicích znaků a 95 tisknutelných znaků. Byl vytvořen zejména pro tisk na tiskárně. Brzy však začala chybět kódová slova pro další znaky (grafické znaky, znaky jiných národních abeced apod.). Nejprve začaly být využívány řídicí znaky také pro zobrazení (na monitoru apod.), pak byl kód rozšířen na osmibitový pod označením **ASCII - 2**. Tento kód se značně rozšířil. V dnešní době se používá jeho zpracovaná verze podle normy **ANSI**. Potřeba velkého množství znaků pro národní abecedy je řešena použitím různých kódových stránek. Například čeština používá kódovou stránku 852. Problém s jednotlivými národními abecedami zcela odstranilo až použití šestnáctibitového kódu Unicode).

Spojením několika znaků vzniká řetězec (string). Někdy je jeho délka omezena (typicky na 255 znaků). Některé programovací jazyky přímo podporují operace pro práci s řetězci. Např. jazyk PASCAL pracuje s řetězci maximálně délky 255 znaků. Jazyky C, C++ naproti tomu typ řetězec (string) vůbec nepodporují. S daty typu řetězec se pracuje vždy jako s **polem znaků**, takže by měl být tento typ zařazen do následující kapitoly věnované složeným datovým typům.

3.1.6 Ukazatel

Mezi základní datové typy náleží také ukazatel. Je určen k tvorbě **dynamických datových struktur**. Obsahuje adresu paměti na které se nacházejí vlastní data. Jeho kardinalita závisí na několika hlediscích. Adresy (modely adresování) mohou být tzv. **blízké** (NEAR) nebo **vzdálené** (FAR). V závislosti na tom může ukazatel zabírat 2 Byte nebo 4 Byte. Manipulaci s ukazateli je věnována samostatná kapitola věnovaná dynamickým datovým strukturám.

3.2 SLOŽENÉ DATOVÉ TYPY

Větší množství informace obvykle ukládáme pomocí složitějších datových struktur, které vzniknou skládáním **základních datových typů**.

3.2.1 Pole

Často používanou datovou strukturou je **pole** *(array)* sestávající z prvků stejného datového typu (jednoduchého nebo také složeného, např. typu záznam, který bude představen později). Jednotlivé **prvky** jsou uspořádány v paměti za sebou a vzestupně očíslovány. V některých programovacích jazycích má první prvek vždy číslo 0, jindy 1, někdy je možno jeho číslo určit, vždy to však musí být číslo celé.

Strukturu pole použijeme také pro uložení **vektoru** hodnot. Pro uložení **matice** potřebujeme **dvourozměrné pole**. První rozměr bude udávat číslo řádku a druhý číslo sloupce, na kterém leží prvek v matici. Maximální použitelný rozměr pole se liší podle použitého programovacího jazyka. Na pole se odkazujeme názvem a číslem prvku, nejčastěji v hranaté závorce:

- jednorozměrné pole x[1], x[i], ...
- dvourozměrné pole x[1,1], x[i,j] ...

I v případě **vícerozměrného pole** jsou prvky uloženy v paměti za sebou, a to buď po řádcích, nebo po sloupcích. K nalezení konkrétního prvku slouží **mapovací funkce**.

Pro uložení matice po řádcích platí:

adresa prvku = adresa prvního prvku matice +

+ $[(i - imin)*(jmax - jmin + 1) +$

+ $(j - jmin)]$*velikost paměti pro jeden prvek.

Vyhodnocování mapovací funkce je programátor většinou ušetřen.

Zvláštní datové struktury jsou někdy tvořeny pro ukládání **řídkých matic**, tedy matic, které mají většinu prvků nulových. Jednou z cest je využití tří jednorozměrných polí. Jedno obsahuje číslo řádku, druhé číslo sloupce a třetí hodnotu prvku matice vždy na stejné pozici v polích.

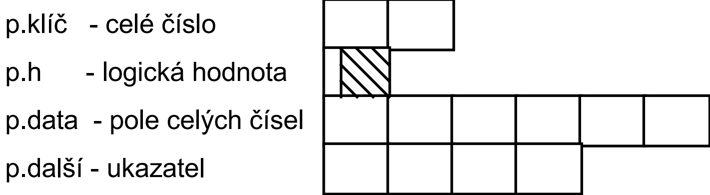
3.2.2 Struktura

Struktura (structure, v některých jazycích také record) je složený datový typ, který umožňuje sdružit informaci složenou z prvků různých datových typů (základních nebo složených) do jednoho celku. Informace je rozdělena na jednotlivé **položky** označené jejich názvy. Přístup k položkám je možný přes název proměnné typu záznam, za který píšeme tečku a název položky, např.:

`p.klíč, data.n[i], copy.a.data` apod.

Záznam má v paměti vymezen kompaktní prostor. Jeho velikost je dána součtem nároků jednotlivých položek. Obvykle se však paměťové prostory jednotlivých položek zarovnávají na celistvý počet datového typu Byte nebo Word (slov o 2 Byte).

Paměťový prostor proměnné typu záznam



V dalším textu budeme používat následující syntaxi pro definici datového typu struktura:

```
structure
  seznam proměnných : datový typ
  seznam proměnných : datový typ
  ....
end structure
```

3.2.3 Množina

Některé programovací jazyky (PASCAL) podporují následující datový typ, zatímco většina jiných ho nezná. **Množina** (set) je skupina prvků určeného datového typu. Množina všech prvků, které je možno do určené množiny zařadit se nazývá **potenční množina**. Může být definována některým základním datovým typem (pak je počet prvků potenční množiny roven kardinalitě základního datového typu), nebo **výčtem hodnot**. Výčet hodnot je někdy chápán jako samostatný datový typ (výčtový typ), je definován seznamem hodnot, případně rozsahem např. 1 ÷ 100. Kardinalita množinového datového typu je dána vztahem:

$$kardinalita = 2^{kardinalita\ potenční\ množiny} \quad (3.8)$$

3.2.3.1 Operace s množinami

*	-	průnik množin,
+	-	sjednocení množin,
-	-	rozdíl množin,
in	-	operátor příslušnosti prvku k množině (oznamuje, zda se prvek v množině nachází).

3.2.4 Objekt

Když mluvíme o objektově orientované technologii, musíme si ujasnit pomocí jakých nástrojů probíhá vývoj. Jsou to tedy: **Objektově Orientovaná Analýza**, **Objektově Orientovaný Návrh**, **Objektově Orientované Programování** a **Objektově Orientované Databáze**. Principielně, objektově orientované technologie modelují chování reálného světa tak, že rozdělí komplexní problémy na základní části. Objektová orientace je pak základem pro čtyři klíčové koncepce:

Datová Abstrakce (Data Abstraction)

Zaobalení (Encapsulation)

Ukrytí Informací (Information Hiding)

Dědičnost (Inheritance)

Datová abstrakce definuje datové typy na vyšší úrovni tak, aby byly pokryty speciální potřeby programovaného problému. Společně s datovou abstrakcí vytváříme zaobalení a ukrytí informací. Zaobalení vzniká, když jsou datové položky a způsoby práce s nimi umístěny do jednoho logického datového typu - **třídy** (class), jehož konkrétní proměnná se jmenuje **Objekt**.

Ukrytí informace se vztahuje k myšlence ukrýt datové položky od okolního světa a poskytnou pouze metody, které umí s nimi pracovat. Těmto metodám se potom říká **Veřejné Rozhraní** (public interface). Tímto velice jednoduchým způsobem mohou být data chráněna před poškozením a navíc nemusí být známo, jak jsou interně implementovány.

Dědičnost spočívá ve vytváření hierarchie tříd. Použitím koncepce předků, což znamená, že např. třída A je předkem třídy B, která je jeho potomkem, můžeme pak definovat, že třída B dědila všechny atributy třídy A a definuje své další atributy.

Abychom využili plnou sílu objektově orientovaných technologií, musíme použít při vývoji různé metody systémové analýzy a návrhu, které byly již vyvinuty.

V této publikaci se objekty zabývat nebudeme, čtenáře odkazujeme na specializovanou literaturu zabývající se objektově orientovaným programováním (OOP – Object Oriented Programming).

3.2.5 Soubor

Pro uložení větších objemů dat, zejména mimo paměť počítače (na disketě apod.) slouží **soubory** (*file*). Soubor je posloupnost za sebou uložených **prvků**, ke kterým je možno se dostat jen postupným průchodem souboru od jeho začátku. Tento postup nazýváme **sekvenční prohledávání**, odtud také soubory nazýváme jako **sekvenční soubory**.

Pokud mají všechny prvky souboru stejnou datovou strukturu - stejný datový typ, nazýváme je jako **typové soubory**. Při známé velikosti datové struktury prvku umožňují rychlejší přístup k i-tému prvku, protože dokáží rychle projít prvky předchozí.

V poslední době se začínají ve velké míře využívat **textové soubory**. V zásadě bychom je mohli chápat jako typové soubory, neboť je do nich uložena posloupnost **znaků**. Jak víme, jeden znak = 1 Byte. Tyto znaky však tvoří ucelené **řetězce**, vzájemně oddělené dvojicí řídicích znaků CR a LF (Carriage Return & Line Feed = návrat vozíku & posuv o řádku), které pochází z doby, kdy sloužily pro řízení tiskárny a podobných zařízení. Řetězce znaků tedy představují jednotlivé řádky textu. Délka řádků je různá, může být omezená, např. 255 znaky. Výhodou textových souborů je možnost ruční opravy obsahu, pokud došlo k jeho porušení a možnost přečtení jeho obsahu v jakémkoliv textovém editoru.

Pro ukládání speciálních typů informace se vyvíjejí různé **formáty souborů**. Soubory ve stejném formátu jsou zpravidla označeny stejnou **příponou** v názvu souboru. Jako ukázku uvádíme formát "**BMP**" pro ukládání obrázků složených z jednotlivých **bodů** (pixelů) umístěných v matici. Někdy jsou takové obrázky nazývány jako **bodová grafika**.

Soubor ve formátu BMP lze rozdělit do čtyř oblastí. Ukážeme si je přímo na rozboru souboru obsahujícího bílou plochu velikosti 10 × 20 bodů. Obsah jednotlivých Byte je zapsán hexadecimálně v (šestnáctkové soustavě):

1. **BitMapFileHeader** (14 Byte) - záhlaví souboru

42 4D = "BM" - typ souboru, tedy BMP

8E 00 00 00 - délka souboru (142 Byte)

00 00 - rezervováno, musí být nulové

00 00 - rezervováno, musí být nulové

3E 00 00 00 - délka všech záhlaví a palety barev (62 Byte)

2. **BitMapInfoHeader** (40 Byte) - informace o obrázku

28 00 00 00 - počet Byte vyhrazených pro toto záhlaví (40)

0A 00 00 00 - šířka obrázku v bodech (10)

14 00 00 00 - výška obrázku v bodech (20)

01 00 - počet vrstev, musí být jedna

01 00 - počet bitů na uložení jednoho bodu (1 ⇒ obrázek je černobílý, 4 ⇒ 16 barev)

00 00 00 00 - použitý typ komprese

00 00 00 00 - velikost obrazu v Byte, pro nekomprimovaný obrázek 0

00 00 00 00 - horizontální rozlišení zdrojového zařízení v bodech na metr

00 00 00 00 - vertikální rozlišení zdrojového zařízení v bodech na metr

00 00 00 00 - počet barev v tabulce barev, pokud počet odpovídá počtu bitů k zobrazení bodu, je zde 0

00 00 00 00 - počet importovaných barev, 0 znamená, že všechny barvy byly importovány

3. **RGBQuadArray** - určení kódů barev v pořadí modrá, zelená, červená, čtvrtý byte je rezervovaný, musí být 0 (Velikost oblasti je v našem případě 8 Byte = dvě barvy)

00 00 00 00 - černá barva

FF FF FF 00 - bílá barva (svítí vše)

4. **Vlastní data** (80 Byte) - obrázek je uložen po řádcích, počet Byte pro uložení řádku je vždy zarovnán na nejblíže vyšší celistvý násobek 4 Byte. Informace o bodech je uložena od začátku řádku, nepotřebné bity na konci řádku mohou obsahovat cokoli.

FF C0 54 20

FF C0 6A 65

FF C0 28 2A

FF C0 43 58

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ľ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ľ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms.* 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

4. LEKCE – DYNAMICKÉ DATOVÉ STRUKTURY

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

4 DYNAMICKÉ DATOVÉ STRUKTURY ... 3
4.1 Dynamický seznam ... 5
4.1.1 Vkládání prvků ... 5
4.1.2 Vyhledávání a vkládání prvků ... 8
4.1.3 Práce se seznamem ... 12
4.1.4 Rušení prvků ... 14
4.1.5 Kruhový zásobník ... 16
4.2 Stromy ... 18
4.2.1 Binární vyhledávací stromy ... 20
4.2.2 Vyvážené stromy ... 23
4.2.3 Tisk struktury stromu ... 30
4.2.5 B-stromy ... 36
POUŽITÁ LITERATURA ... 40

4 DYNAMICKÉ DATOVÉ STRUKTURY

OBSAH KAPITOLY:

Dynamický seznam

Stromy

MOTIVACE:

Pro efektivní řešení řady složitějších problémů je potřeba dobře pracovat s dostupnou pamětí. K tomu se s výhodou používají dynamické datové struktury, zejména, pokud předem nevíme, jaký objem dat budeme zpracovávat. Mezi nejjednodušší dynamické datové struktury patří lineární seznamy a stromy. Pro jejich využití je třeba se s nimi dobře obeznámit a umět je správně používat.

CÍL:

Naučit se znát vlastnosti a využití jednotlivých dynamických datových struktur. Umět používat lineární seznamy a vědět jak s jejich pomocí realizovat datové struktury jako je fronta, zásobník a setříděný seznam. Umět vkládat i vyjímat prvky z lineárního seznamu i setřídit prvky lineárního seznamu. Rozumět pojmům z oblasti stromů jako je stupeň stromu, hloubka (výška) stromu, rozhodovací a vyhledávací stromy a další. Znát základní algoritmy pro práci s binárními stromy, umět vkládat prvky do stromu i vyjímat prvky ze stromu, umět zkonstruovat vyvážený strom, AVL strom i optimální strom. Rozumět konstrukci B-stromu a umět ho efektivně využívat.

Jako **dynamické datové struktury** chápeme takové struktury, které během zpracování mění svoji strukturu (velikost zabírané paměti). Tím se výrazně liší od **datových struktur statických**. Výhodou dynamických struktur je skutečnost, že nezabírají paměť stále, ale jen pokud je jich potřeba.

Základem dynamických datových struktur je speciální **datový typ** nazvaný **ukazatel** (*pointer*). Zabírá 4 Byte (případně 2 Byte), které mají význam adresy v paměti, na které se nacházejí vlastní data. Při práci s proměnnou typu ukazatel musíme rozlišit dva případy:

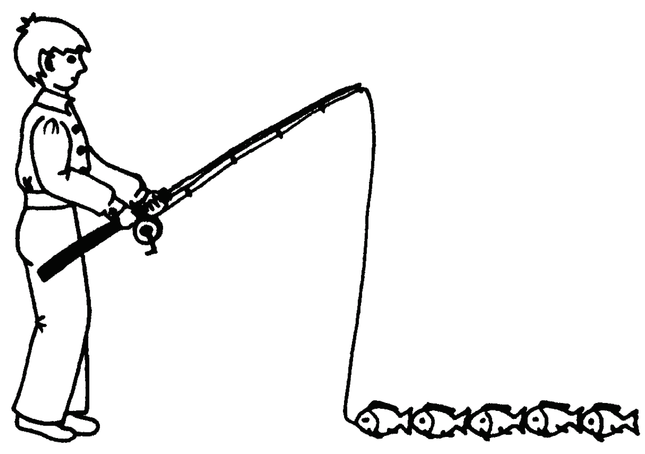
Jednak je to práce přímo s obsahem této proměnné. Odkazujeme se na něj názvem této proměnné, např. p.

Druhým případem je práce s daty, na která proměnná ukazuje. Odkazujeme se na ně názvem proměnné doplněným **znakem ukazatele** (znázorňuje se obvykle stříškou „^“ nebo šipkou nahoru „↑“, pro větší zřetelnost budeme dále používat znak šipky), např. p↑

Data, na která proměnná ukazuje, mohou mít jakoukoliv strukturu. Pomocí ukazatelů můžeme vytvořit pomocné proměnné pro algoritmus, které po jeho skončení opět uvolníme z paměti. Pomocí statických proměnných bychom pomocné proměnné mohli vytvořit dvěma způsoby.

pomocí **globálních proměnných**, pak budou zabírat paměť po celou dobu práce programu,

pomocí **lokálních proměnných**, pak budou zabírat paměť po dobu práce rutiny, která je deklarovala,



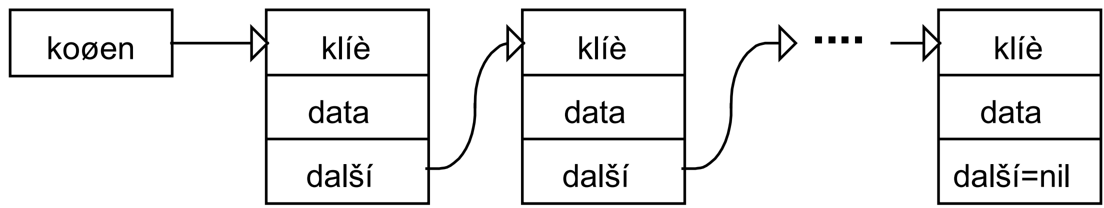
Obrázek 4.1 Dynamický seznam (lineární seznam)

Lokální proměnné se však alokují v paměti pro **zásobník**. Sem se ukládá také informace o volání rutin, takže této paměti není k dispozici příliš mnoho. Hlášení o **přetečení zásobníku** (*stack overflow error*), patří k velmi nepříjemným, neboť si vynucuje změnu přístupu k řešení problému. Použití dynamických proměnných může výrazně pomoci.

Proměnné typu ukazatel obvykle nepoužíváme pro základní datové typy, ale pro **datové typy složené**, ať již jsou to pole, nebo proměnné typu **struktura** (*record*). Pro nás bude zajímavější využití složených datových typů. Umožní nám tvořit datové struktury, které mají skutečně zcela dynamický charakter. Obvykle je možno rozdělit vlastní data na tři části:

1. **klíč** (*index*) – data, podle kterých třídíme informaci,
2. další data, často ukazatel na další datovou strukturu,
3. ukazatel na další proměnnou stejného datového typu.

Zřetězením těchto dat vzniká dynamická datová struktura – **dynamický seznam** (lineární seznam, vázaný seznam). K jeho vytvoření a samozřejmě také k přístupu k seznamu potřebujeme proměnnou typu ukazatel na danou datovou strukturu, nazveme ji **kořen**. Jejím obsahem bude adresa první dynamické proměnné. Ta bude ukazovat na další **prvek seznamu** atd.



Obrázek 4.2 Dynamický seznam (lineární seznam, vázaný seznam)

Velmi důležité je rozpoznání konce struktury. Poslední ukazatel totiž již nemá kam ukazovat. Přitom není možné, aby tento ukazatel měl libovolnou hodnotu, nepoznali bychom, že již neukazuje na další proměnnou. Pro tyto účely je v jazyku PASCAL definována hodnota **nil** pro ukazatel, který je prázdný, tj. neukazuje na žádná data. Toto označení budeme nadále používat i my.

Kromě toho je třeba definovat alespoň dvě základní procedury pro **dynamické přidělování paměti**. Opět využíváme funkcí známých z jazyka PASCAL:

> **new(p)** – Přidělí dynamické proměnné paměť pro její datovou strukturu a adresu této paměti uloží do proměnné **p** typu ukazatel.
>
> **dispose(p)** – Uvolní paměť přidělenou proměnné **p**. Obsah proměnné p se ale obvykle nemění.

Pozorný čtenář si jistě uvědomil, že obě procedury musí znát potřebnou velikost paměti pro datovou strukturu. Obvykle je problém řešen tak, že při deklaraci proměnné definujeme její typ jako ukazatel na určenou datovou strukturu. Některé jazyky pak nedovolují přiřadit proměnné obsah jiného datového typu. Pro některé aplikace to není vhodné a požadujeme proměnné, které mohou ukazovat na jakoukoliv datovou strukturu. Pro přidělení a uvolnění paměti pak musíme použít procedury, kterým sdělíme také velikost přidělované (uvolňované) paměti, tuto velikost si ovšem musíme stále pamatovat. V jazyku PASCAL jsou to:

> **GetMem (p, size)**
>
> **FreeMem (p, size)**

V dalším textu budeme pracovat s procedurami `new(p) a dispose(p).`

4.1 DYNAMICKÝ SEZNAM

Nejjednodušší dynamickou strukturou je **lineární seznam**, viz obr.4.2.

Datová struktura je přístupná pomocí proměnné **kořen**. Tím je lineární seznam podobný struktuře **souboru**. Do souboru však obvykle musíme nové prvky ukládat jen na konec souboru a rušit celý soubor najednou. U lineárního seznamu máme možností více. Nyní rozebereme základní operace s lineárním seznamem.

4.1.1 Vkládání prvků

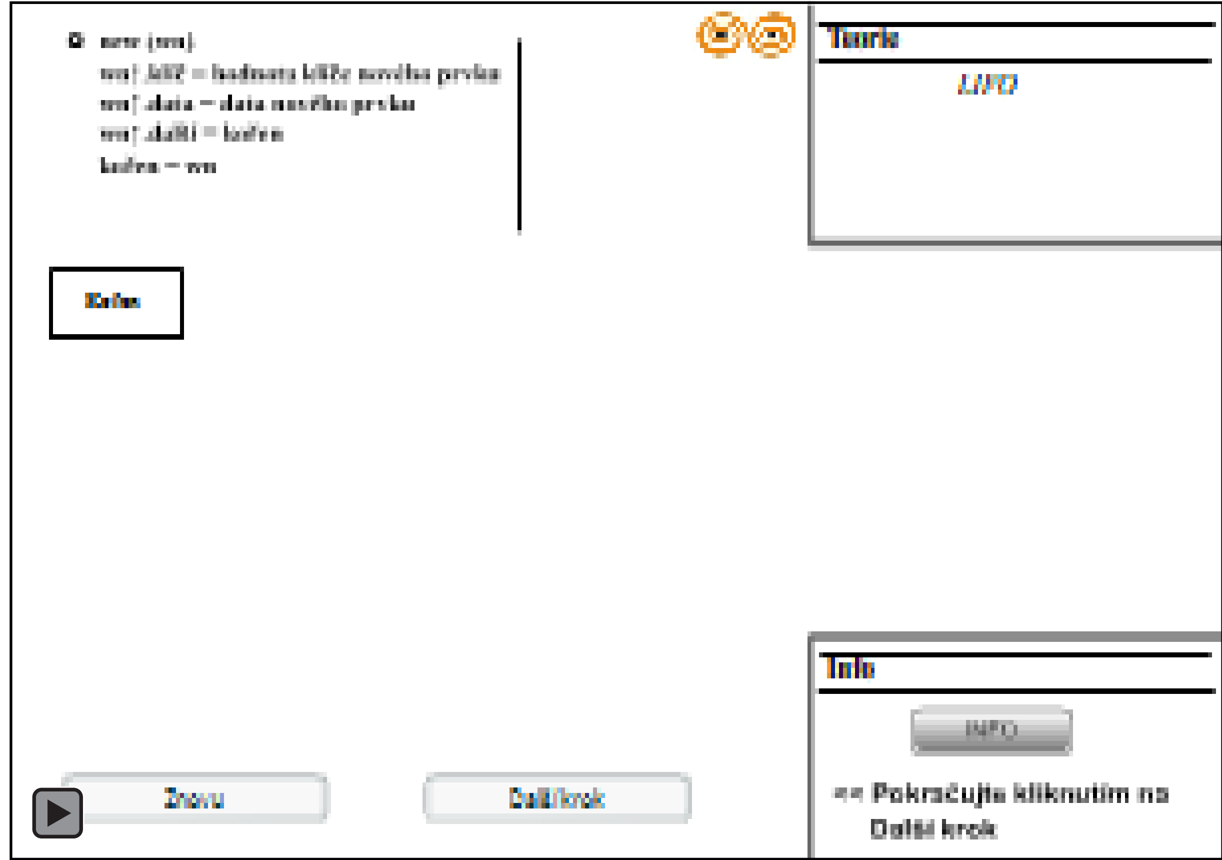
Nejjednodušší operací je vložení nového prvku na začátek seznamu. Kromě proměnné **kořen** potřebujeme jednu pomocnou proměnnou, např. **wn** stejného datového typu.

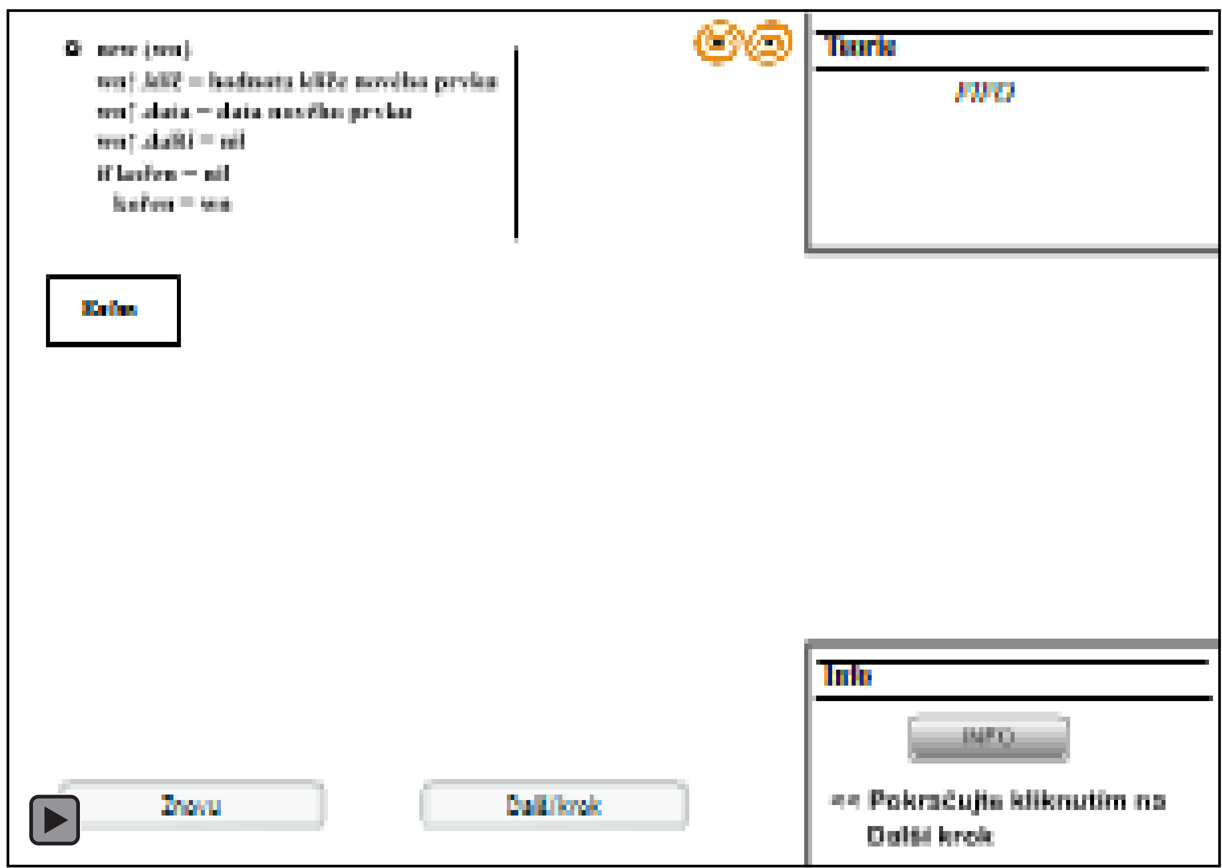
```
new (wn)
```

```
wn↑.klíč = hodnota klíče nového prvku
wn↑.data = data nového prvku
wn↑.další = kořen
kořen = wn
```

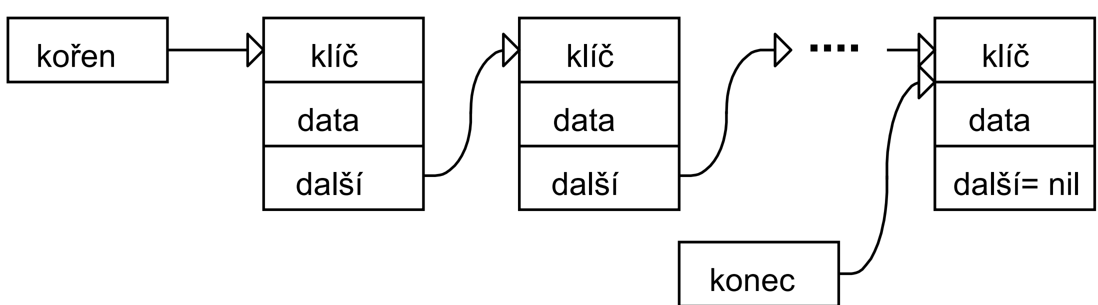
Výsledkem je lineární seznam obsahující prvky seřazené v opačném pořadí, než byly vkládány. K seznamu je přístup jen přes proměnnou **kořen**, takže poslední vložený prvek je první přístupný. Výsledná struktura se nazývá **zásobník**, neboli struktura **LIFO** (z anglického „last in, first out“). Druhou možností je vkládat data na konec seznamu, pak je první vložený prvek také první přístupný. Tato struktura se nazývá **fronta** neboli **FIFO** (first in, first out). Při vkládání na konec seznamu musíme řešit samostatně vkládání prvního prvku do seznamu, neboť má výlučné postavení dané odkazem proměnné **kořen**.

```
new (wn)
wn↑.klíč = hodnota klíče nového prvku
wn↑.data = data nového prvku
wn↑.další = nil
if kořen = nil
  kořen = wn
else
  w = kořen
  while not w↑.další = nil
        w = w↑.další
  end while
  w↑.další = wn
end if
```





Z výpisu je zřejmé, že je nutné vždy dojít na konec seznamu, abychom mohli vložit nový prvek, k tomu jsme použili pomocnou proměnnou **w**. Celý postup výrazně zrychlíme zavedením proměnné **konec**, která bude stále ukazovat na poslední prvek.



Obrázek 4.3 Dynamický seznam s ukazateli na začátek a konec

Postup ukládání nového prvku na konec seznamu se pak výrazně zjednoduší a zrychlí.

```
new (wn)
wn↑.klíč = klíč
wn↑.data = data
wn↑.další = nil
if kořen = nil
  kořen = wn
  konec = wn
else
  konec↑.další = wn
  konec = wn
end if
```

4.1.2 Vyhledávání a vkládání prvků

V řadě aplikací nejprve hledáme prvek v seznamu a vkládáme ho, jen pokud v seznamu dosud není. Obvykle nový prvek vkládáme na konec seznamu. Při vyhledávání ukončíme činnost pokud nastane jedna ze dvou skutečností

nalezení hledaného prvku, kdy w↑.klíč = nový klíč,

nalezení konce seznamu, kdy w = nil.

V druhém případě budeme přidávat nový prvek na konec seznamu, jenže nám chybí odkaz na poslední prvek seznamu. Nejsnáze ho získáme zavedením pomocné proměnné **wk**. Celý postup bude vhodné rozdělit do dvou fází:

Nejprve budeme hledat, zda se prvek v seznamu nachází. Pro tuto informaci přidáme logickou proměnnou **h**. Pokud prvek nenajdeme, uložíme do proměnné **wk** odkaz na poslední prvek seznamu.

Podle výsledku hledání buď provedeme manipulaci s existujícím prvkem seznamu (zvýšení počtu výskytů apod.) nebo přidáme nový prvek do seznamu. Nesmíme zapomenout, že přidáváme prvek na konec seznamu, takže je nutno respektovat zvláštní způsob vložení prvního prvku do seznamu.

```
w = kořen
h = true
while w<>nil and h
  if w↑.klíč = klíč
        h = false
  else
        wk = w
        w = w↑.další
  end if
end while
if h
  new (wn)
  wn↑.klíč = klíč
  wn↑.data = data
  wn↑.další = nil
  if kořen = nil
        kořen = wn
  else
        wk↑.další= wn
  end if
else
  práce s daty w↑.data
end if
```

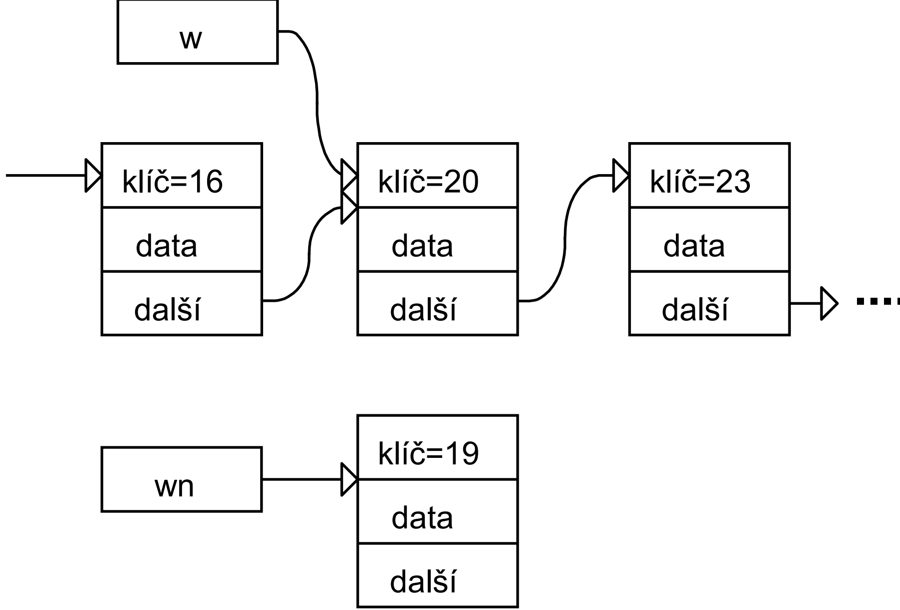
Pokud chceme zjednodušit vkládání, můžeme nový prvek vkládat na začátek seznamu. Vkládání na začátek či konec seznamu ale není právě typické. Daleko častější bude vkládání nového prvku tak, aby indexy prvků v seznamu byly seřazeny (ať již vzestupně nebo sestupně). Přitom může nastat jeden z případů:

je vkládán první prvek do prázdného seznamu,

je vkládán nový prvek na konec seznamu,

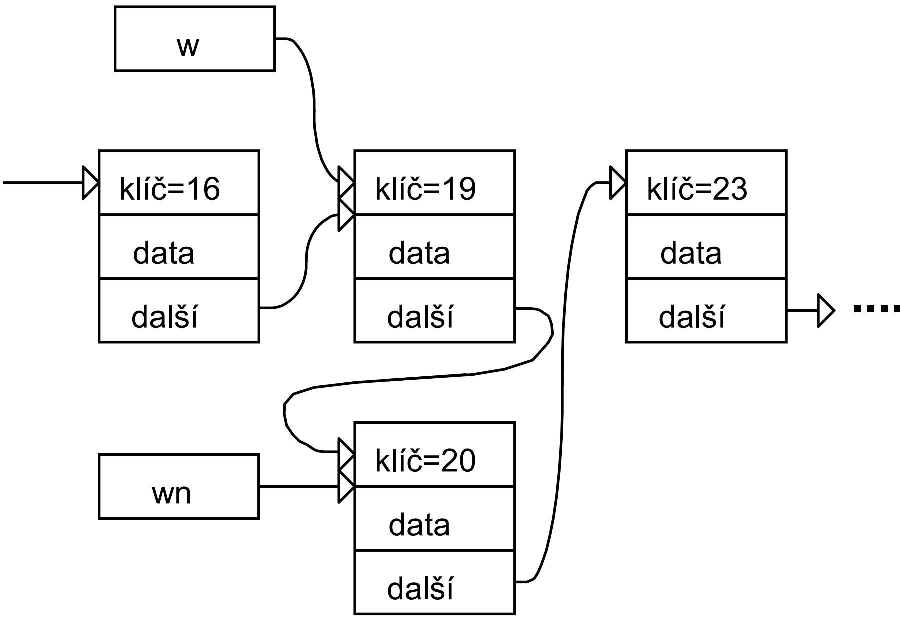
je vkládán nový prvek před nalezený prvek seznamu, neboť nalezený prvek má klíč vyšší hodnoty.

Obsluhu prvních dvou případů známe. Zajímavý je třetí případ. Musíme si uvědomit, že vkládáme nový prvek **před aktuální**! K tomu nám chybí odkaz na předchozí prvek, jehož složka **další** by se měla ukazovat na nově vložený prvek.



Obrázek 4.4 Situace před vložením nového prvku před prvek, na který ukazuje proměnná *w*

To samozřejmě není možné. Zařazení nového prvku dosáhneme jednoduchým trikem. Data proměnné **wn** naplníme daty z proměnné **w**, k proměnné **w** pak vložíme data nového prvku a upravíme ukazatele do výsledné podoby.



Obrázek 4.5 Situace po vložení nového prvku před prvek, na který ukazuje proměnná *w*

Postup vložení nového prvku před prvek w↑:

```
new (wn)
wn↑= w↑              REM přiřazení celé datové struktury
w↑.klíč = klíč
w↑.data = data
```

```
w↑.další = wn
```

Rádi bychom nějak zjednodušili vkládání nového prvku, nejlepší by bylo, kdybychom se dokázali zbavit nutnosti rozeznat jednotlivé případy vkládání (prázdný seznam, dovnitř a na konec seznamu). Na první pohled se to zdá nemožné. Přece však existuje vcelku jednoduchý trik, jak toho dosáhnout. Jmenuje se **zarážka**. Na počátku vložíme do prázdného seznamu ihned jeden prvek zcela libovolného (tedy i nedefinovaného) obsahu, na který bude ukazovat proměnná **zarážka**. Při prohledávání seznamu skončíme nalezením prvku, nebo dosažením situace, kdy prohledávací proměnná w = zarážka.

Postup založení prázdného seznamu se zarážkou

```
new(zarážka)
kořen = zarážka
```

Postup prohledávání a vkládání do seznamu se zarážkou

```
w = kořen
h = true
k = true
while h and w<>zarážka
  if w↑.klíč = klíč
        h = false
        k = false
  else
        if w↑.klíč<klíč
              w = w↑. další
        else
              h = false
        end if
  end if
end while
if k
  new (wn)
  wn↑ = w↑
  w↑.klíč = klíč
  w↑.data = data
  w↑.další = wn
  if w = zarážka
        zarážka = w↑.další
  end if
else
  práce s daty w↑.data
end if
```

Jak vidíme, zarážka výrazně zjednodušila algoritmus vkládání nového prvku. Výsledkem po vložení všech prvků je setříděný **indexsekvenční seznam**. Pro vyhledání prvku postupujeme od začátku seznamu až k prvku nebo (pokud neexistuje) na konec seznamu.

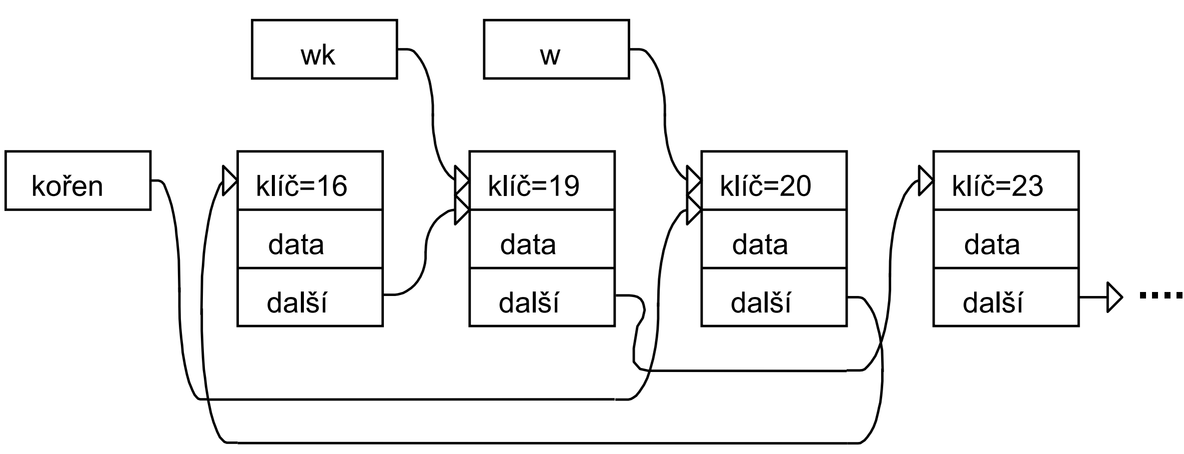
V některých aplikacích bychom rádi vyhledávání v průběhu vkládání zkrátili. Můžeme k tomu využít skutečnosti, že obvykle se jednotlivé prvky vyskytují s různou četností (různě často), v textech mají navíc jednotlivá slova tendenci ke shlukování, tedy opakují se v textu blízko sebe. Zrychlení vyhledávání pak dosáhneme, pokud při zpracování nalezených dat provedeme přesun těchto dat na začátek seznamu. Pokud tam náhodou jsou, tak se nic neděje. Nový prvek vkládáme rovněž na začátek seznamu.

```
w = kořen
```

```
h = true
while w<> nil and h
  if w↑.klíč = klíč
        h = false
  else
        wk = w
        w = w↑.další
  end if
end while
if h
  new (wn)
  wn↑.klíč = klíč
  wn↑. data = data
  wn↑. další = kořen
  kořen = wn
else
  zpracování dat w↑.data
  if w<>kořen
        wk↑.další = w↑.další
        w↑.další = kořen
        kořen = w
  end if
end if
```

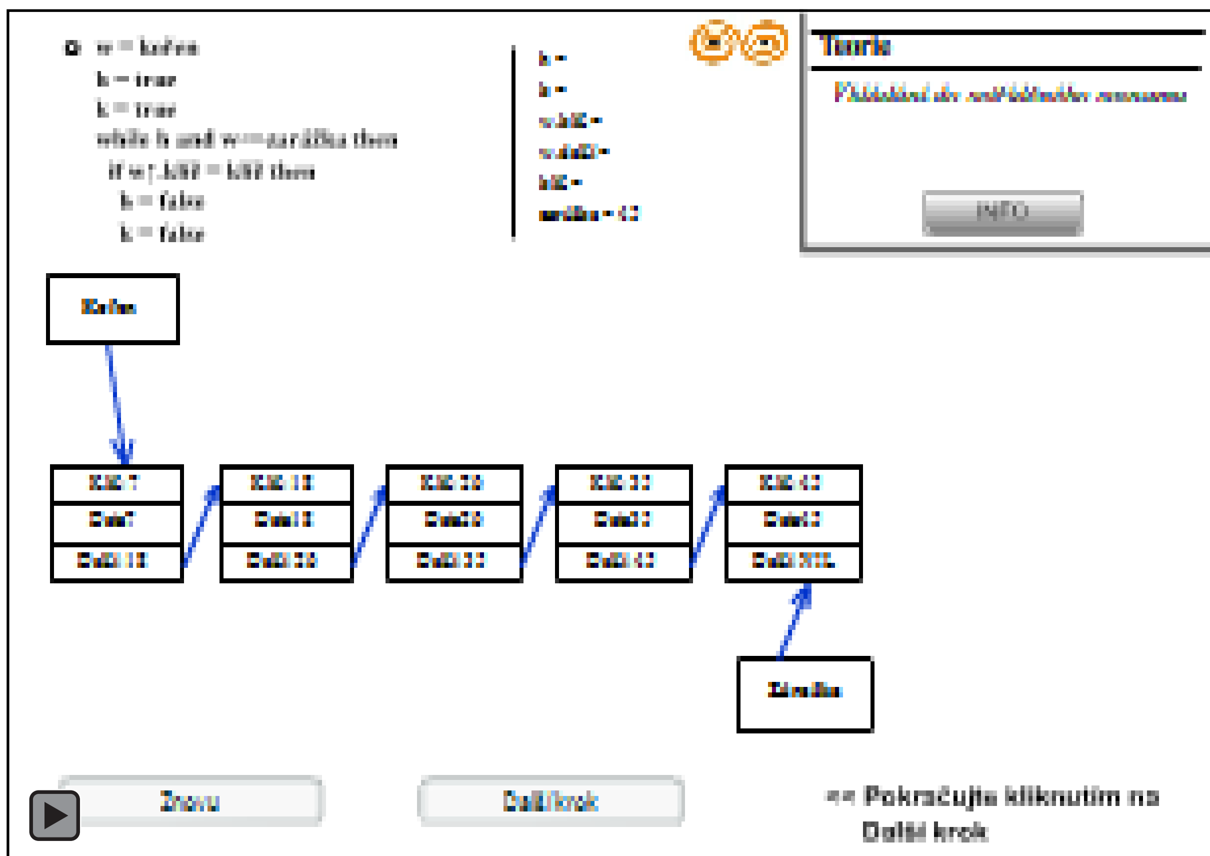
wk
w
kořen
klíč=16 data další
klíč=19 data další
klíč=20 data další
klíč=23 data další ....

Obrázek 4.6 Situace před přesunem prvku w na začátek seznamu



Obrázek 4.7 Situace po přesunu prvku *w* na začátek seznamu

Budeme-li chtít dále zrychlit vyhledávání prvku, pak již musíme opustit **lineární seznamy** a přejít k **vyhledávacím stromům**, ale o těch se píše o kousek dále.



4.1.3 Práce se seznamem

Zpracování vytvořeného seznamu je možné prakticky jen postupným procházením seznamu od jeho kořene. Jednotlivé prvky tedy budou zpracovány v pořadí svého umístění v seznamu. Pokud bychom chtěli např. zpracovat prvky od posledního směrem k prvnímu, mohli bychom s výhodou využít **proceduru** (programovou rutinu) s **rekurzívním voláním**, kdy procedura volá sama sebe. Některé programové jazyky takové volání nepřipouštějí a jak záhy ukážeme, pro lineární seznam není tento postup vhodný.

```
průchod(w)
  if w<> nil
        průchod (w↑.další)
        zpracování dat w↑.data
  end if
end průchod
```

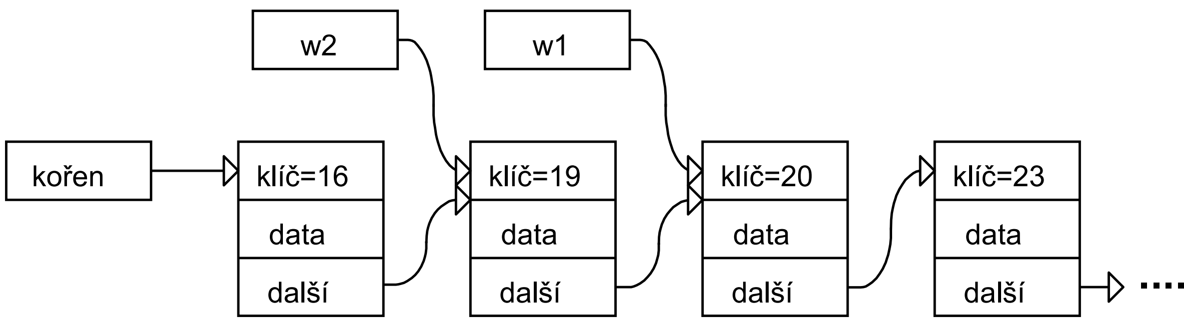
Proceduru budeme volat **průchod(kořen)**. Vidíme, že postupným voláním dojde procedura na konec seznamu a při návratu začne zpracovávat data jednotlivých prvků seznamu.

Vše vypadá velmi krásně, leč rekurzívní algoritmus v sobě nese skryté a to zákeřnější nebezpečí. Tím je **programový zásobník** (*stack*). Je to vyhrazená část paměti, do které se ukládají informace nutné pro návrat z procedury (obsah registrů procesoru, návratová adresa) a vytvářejí se v ní také lokální proměnné procedury a parametry předávané hodnotou. Pokud je seznam delší, hrozí nebezpečí, že vyhrazený paměťový prostor pro zásobník bude nedostatečný, takže dojde k již několikrát zmiňovanému **přetečení zásobníku**. To je závažná chyba, která má obvykle za následek havárii programu. Proto raději používáme **iterační algoritmus** průchodu seznamem, který zpracuje prvky v pořadí od začátku seznamu.

```
w = kořen
while w<>nil
  zpracování dat w↑.data
```

```
    w = w↑.další
  end while
```

Může se stát, že seznam setříděný podle daného klíče potřebujeme setřídit podle jiného **klíče** (jiné složky dat). Přitom máme jen omezený přístup k prvkům seznamu – vždy přes kořen seznamu. Proto můžeme použít jen nejjednodušší a nejméně efektivní metody třídění, jako je **třídění přímým vkládáním** a **třídění přímým výběrem**. Při jejich aplikaci je vhodné brát prvky z jednoho seznamu a vkládat do druhého. Pokud chceme třídit přímo v seznamu, pak můžeme použít **třídění přímou výměnou** známé také jako **bublinkové třídění** (*buble sort*). Při něm porovnáváme hodnoty dvou sousedních prvků a v případě potřeby je vyměníme. Následující program je aplikací tohoto algoritmu při setřídění lineárního seznamu podle hodnoty klíče. Pracuje se dvěma ukazateli **w1** a **w2**, výměna obsahu prvků probíhá pomocí proměnné **wp**. Nesmíme zapomenout, že k tomu, abychom mohli vyměnit pouze odkazy na oba prvky, nám chybí přístup k prvku, který předchází zkoumanou dvojici v seznamu.



Obrázek 4.8 Situace před výměnou obsahu dvojice prvků v seznamu

K ukončení algoritmu je použita logická proměnná **h**, která oznamuje, zda při průchodu seznamem došlo k výměně.

```
new(wp)
h = true
while h
  h = false
  w1 = kořen
  if w1 <> nil
        w2 = w1
        w1 = w1↑.další
  end if
  while w1<> nil
        if w2↑.klíč>w1↑.klíč
              wp↑.klíč = w2↑.klíč
              wp↑.data = w2↑.data
              w2↑.klíč = w1↑.klíč
              w2↑.data = w1↑.data
              w1↑.klíč = wp↑.klíč
              w1↑.data = wp↑.data
              h = true
        edn if
        w2 = w1
        w1 = w1↑. další
  end while
end while
dispose(wp)
```

Pokud bychom chtěli realizovat podobné **třídění přetřásáním** (*shake sort*) při kterém procházíme seznam střídavě odpředu a odzadu, pak narazíme na zásadní problém. Pro postup odzadu nám chybí informace o předešlém prvku seznamu. Pokud tuto informaci přidáme, získáme **obousměrný seznam**. Vkládání prvků je daleko složitější, nejčastěji se v průběhu vkládání odkazy na předchozí prvky ignorují a nastaví se až po skončení vkládání najednou následujícím algoritmem:

```
wp = nil
w = kořen
while w<>nil
  w↑.minulý = wp
  wp = w
  w = w↑.další
end while
```

Obrázek 4.9 Obousměrný seznam

Výkonnější algoritmy třídění nelze realizovat ani při použití obousměrného seznamu, vyžadují totiž přímý přístup ke všem prvkům seznamu. Z tohoto pohledu je lineární seznam více podobný **sekvenčnímu souboru**, takže je možno na něj aplikovat metody třídění známé ze sekvenčních souborů. Výhody dynamických datových struktur se projevují zejména ve speciálních aplikacích. Široce se využívají např. u metod z teorie grafů, u topologických třídění apod., viz např. [8].

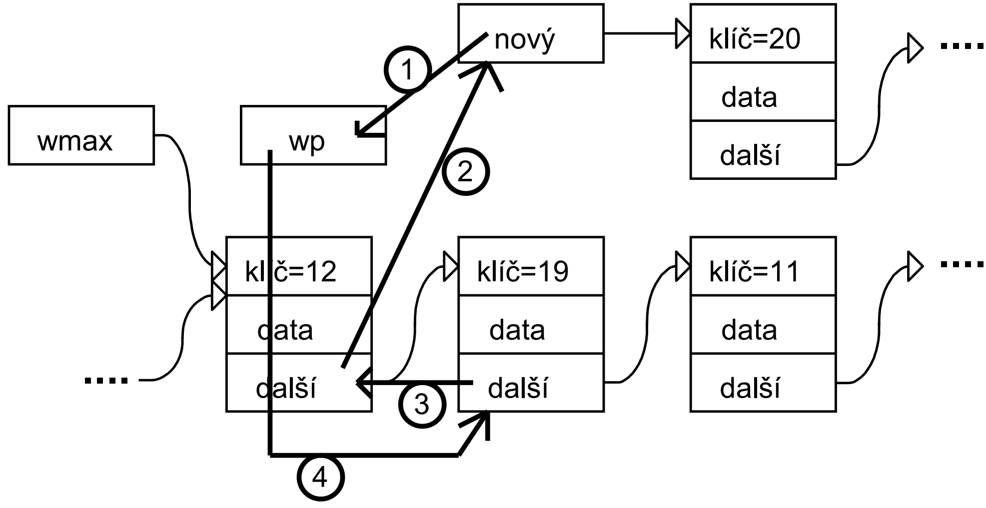
4.1.4 Rušení prvků

Lineární seznam obvykle rušíme celý. Nejjednodušší je rušení prvků na začátku seznamu.

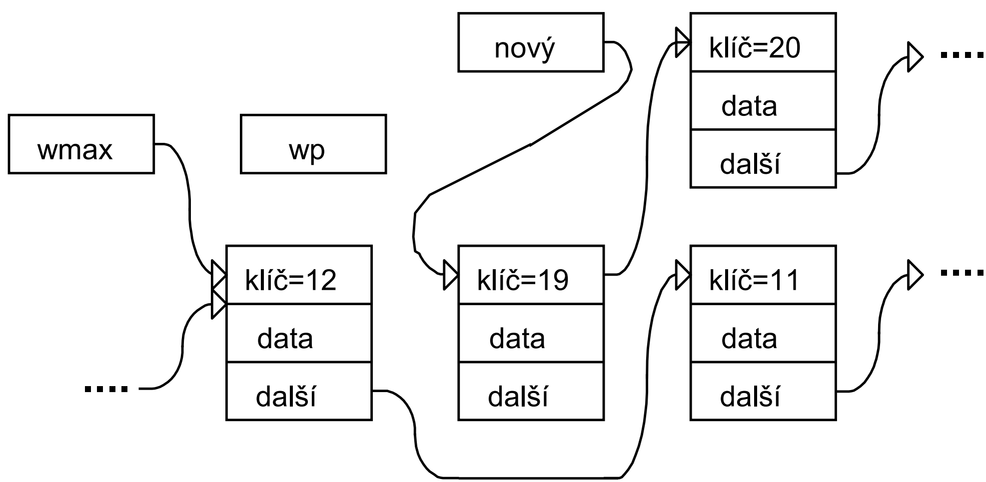
```
while kořen <> nil
  wp = kořen
  kořen = kořen↑.další
  dispose(wp)
end while
```

Všimněte si, že poprvé provádíme průchod seznamem s využitím proměnné **kořen**, u dříve uvedených algoritmů to nebylo přípustné, neboť bychom ztratili přístup k začátku seznamu.

Rušení prvků uvnitř seznamu je složitější, protože musíme zachovat vazby posloupnosti prvků seznamu. Nejjednodušší operací je vynětí následníka prvku, na který ukazuje proměnná **wmax** a jeho přesunutí na začátek jiného seznamu. K přesunu adres přitom použijeme pomocnou proměnnou **wp**.



Obrázek 4.10 Situace před přesunutím následníka prvku, na který ukazuje *wmax*, na začátek seznamu proměnné *nový*, včetně pořadí přesunu informace



Obrázek 4.11 Situace po přesunu prvku na začátek nového seznamu

Při aplikaci uvedeného postupu nesmíme zapomenout, že takto nelze přesouvat první prvek původního seznamu. Upravit algoritmus po přesunu prvního prvku jistě nebude pro čtenáře náročnou operací. Následující algoritmus používá popsaný postup pro třídění seznamu **metodou přímého výběru**. V původním seznamu najdeme prvek s maximální hodnotou klíče (zapamatujeme si adresu jeho předchůdce) a přesuneme ho na začátek nového seznamu. Nový seznam pak bude seřazen vzestupně.

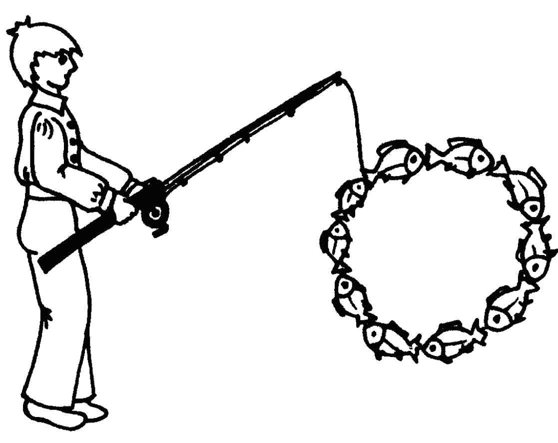
```
nový = nil
while kořen <> nil do
  w1 = kořen
  w2 = nil
  wmax = w2
  kmax = w1↑.klíč
  while w1<>nil
        if w1↑.klíč >= kmax
              wmax = w2
              kmax = w1↑.klíč
        end if
        w2 = w1
        w1 = w1↑.další
```

```
        end while
        if wmax <>nil
              wp = nový
              nový = wmax↑.další
              wmax↑.další = nový↑.další
              nový↑.další = wp
        else
              wp = nový
              nový = kořen
              kořen = nový↑.další
              nový↑.další = wp
        end if
      end while
      kořen = nový
```

Pokud chceme rušit přímo prvek, na který ukazuje proměnná wmax, pak můžeme využít stejný trik, jako při vkládání nového prvku před existující prvek. Musíme si ale uvědomit, že tímto způsobem nelze zrušit poslední prvek seznamu. Tento nedostatek můžeme opět vyřešit použitím zarážky na konci seznamu.

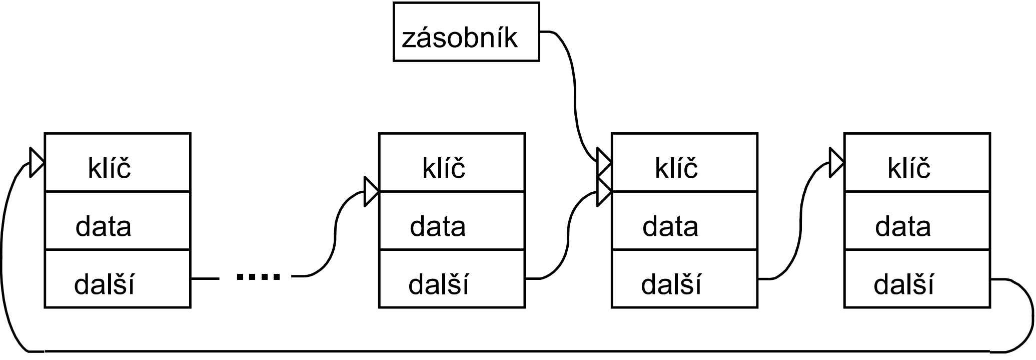
```
wp = wmax↑.další
wmax↑ = wp↑
if wp = zarážka
  zarážka = wmax
end if
dispose(wp)
```

4.1.5 Kruhový zásobník



Obrázek 4.12 Kruhový zásobník

Jak již bylo uvedeno, **zásobník** je datová struktura, do které vkládáme prvky na začátek seznamu a čteme opět od začátku. Poslední vložený prvek je tedy zpracován první (struktura **LIFO**). Pod pojmem **kruhový zásobník** se však skrývá trochu jiná datová struktura. Má obvykle pevný počet prvků, u nichž postupujeme tak, že přečteme hodnotu prvku, nahradíme ji novou a přesuneme se na následující prvek. Protože jsou prvky seřazeny v kruhu, má následující prvek vždy aktuálně nejstarší hodnotu.



Obrázek 4.13 Kruhový zásobník

Kruhový zásobník pro n prvků (n ≥ 1) vytvoříme algoritmem

```
new (zásobník)
w1 = zásobník
inicializace dat w1.data
i = 1
while i< n
  new(w1↑.další)
  w1 = w1↑.další
  inicializace dat w1↑.data
  i = i+1
end while
w1↑.další = zásobník
```

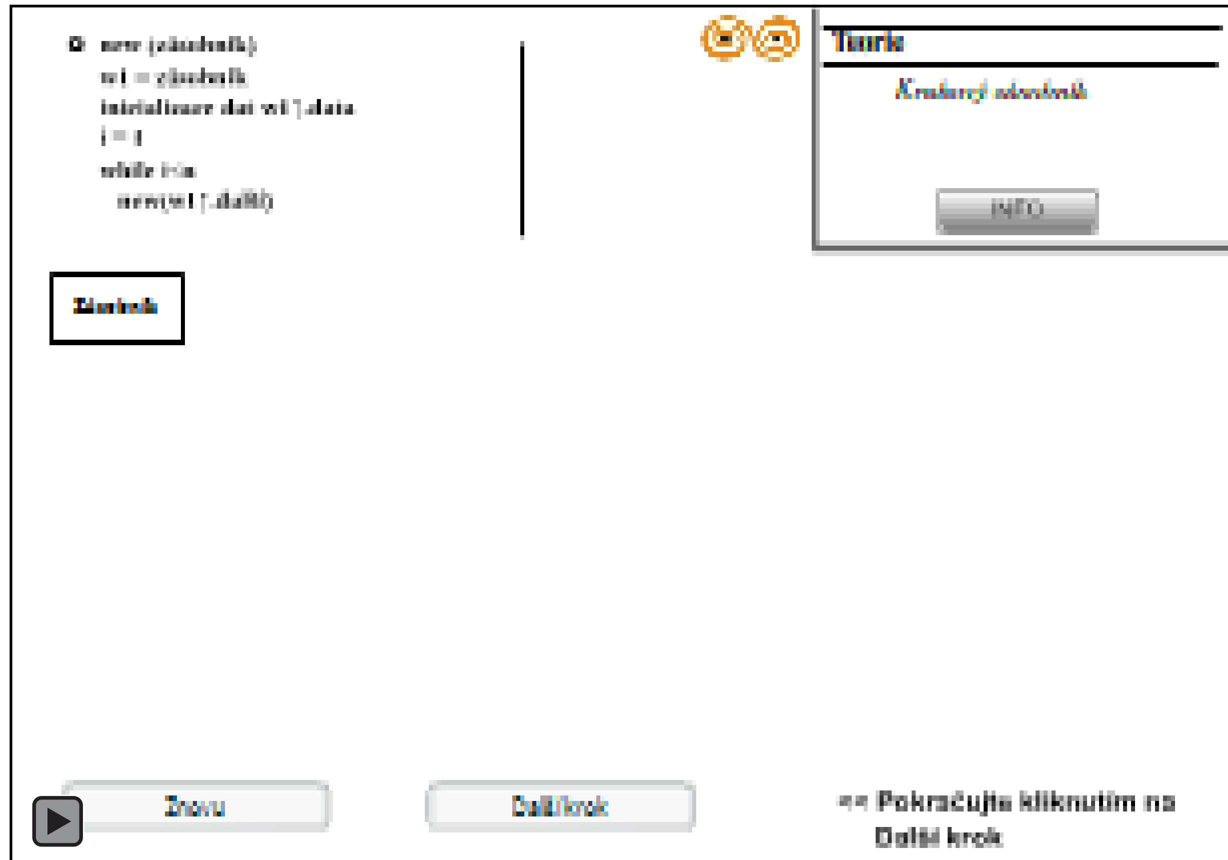
vlastní průchod zásobníkem pak probíhá následovně

```
přečtení dat zásobník↑.data
uložení nových dat zásobník↑.data
zásobník = zásobník↑.další
```

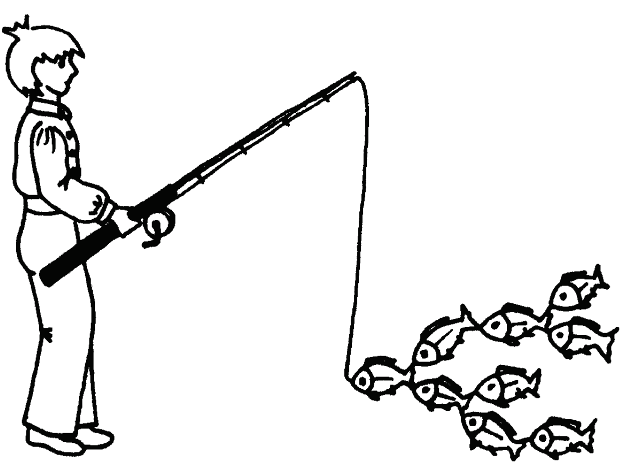
Zrušení kruhového zásobníku provedeme takto (pokud má zásobník alespoň jeden prvek). Test na konci opakování je důležitý pro správnou činnost pokud má zásobník právě jeden prvek. Pokud máme jistotu, že kruhový zásobník má alespoň dva prvky, můžeme test přesunout na začátek opakování:

```
w1 = zásobník
repeat
  wp = w1
  w1 = w1↑.další
  dispose(wp)
until w1 = zásobník
```

Kruhový zásobník se využívá v některých speciálních algoritmech, jako je komprese dat nebo realizace zpoždění o daný počet kroků apod.



4.2 STROMY



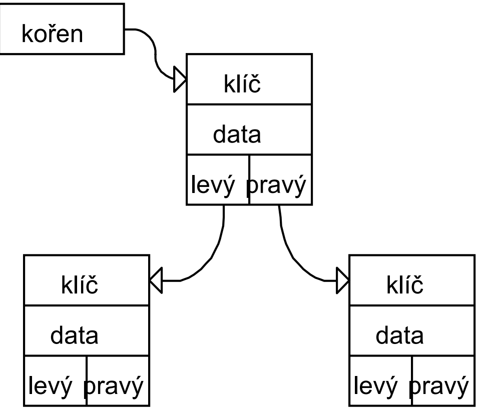
Obrázek 4.14 Stromová struktura

Název datové struktury ***strom*** byl zvolen pro grafickou analogii se skutečnými stromy, s jejich rozvětvováním od kmene přes větve až po listy. Základním kamenem stromu je datová struktura ***prvek*** (uzel, vrchol), která kromě vlastní informace obsahuje konečný počet připojených stromových struktur stejné datové struktury (datového typu) – ***podstromů***.

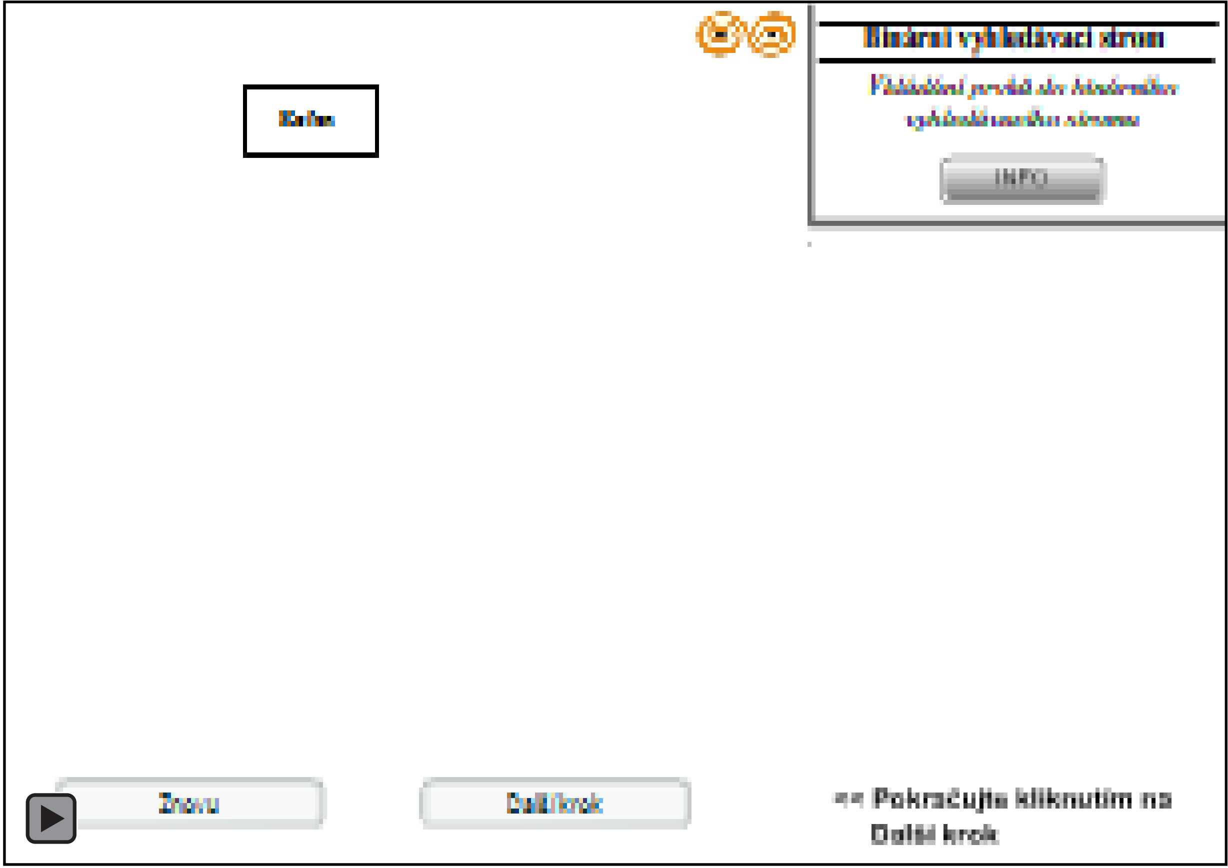
Z uvedené definice je zřejmé, že ***lineární seznam*** můžeme chápat také jako strom, ve kterém má každý prvek nejvýše jeden podstrom. Většinou ho však chápeme jako ***degenerovaný strom***, jak uvidíme dále, může k jeho vytvoření vlivem nepříznivé situace dojít. My se však budeme dále zabývat stromy s více než jedním podstromem. Maximální počet podstromů, který se u prvků stromu vyskytuje, přitom nazýváme ***stupeň stromu***. Pokud tedy žádný prvek

nemá více než dva podstromy, jedná se o strom druhého stupně. Tento typ stromu má v praxi velmi časté použití, proto v dalším textu této kapitoly budeme popisovat výhradně pouze stromy druhého stupně. Pro jejich označení se často používá pojem ***binární stromy***.

Vlastní data každého prvku rozdělíme na informaci o ***klíčové položce*** (klíči, indexu), podle které budeme prvky řadit a zbytek dat. Odkazy na podstromy budeme v souladu se zvyklostmi a analogií grafického znázornění stromu označovat jako levý a pravý podstrom.



Obrázek 4.15 Strom



Základním prvkem stromu je ***kořen***. Liší se od ostatních prvků stromu tím, že nemá žádného ***předchůdce***, ale jen nejvýše dva přímé ***následníky*** (levý a pravý podstrom). Tedy prvky, které na něj bezprostředně navazují. Obecně je ***následník*** takový prvek, který leží v některém podstromu daného prvku. Prvky, které nemají žádné následníky, označujeme za ***koncové prvky***, nebo také ***listy***. Ostatní prvky, které nejsou ani koncové ani kořen, nazýváme ***vnitřní***

prvky. Jako ***prázdný strom*** pak označujeme takový strom, který nemá ani kořen (proměnná ***kořen*** = ***nil)***

Stromy většinou zobrazujeme tak, že se rozvíjejí od kořene směrem dolů (označení koncových prvků jako listy pak působí poněkud překvapivě, ale označení kořínky či podobné je matoucí a nepoužívá se). Prvky, které mají stejný počet předchůdců, kreslíme ve společné vodorovné linii. Říkáme, že tyto prvky leží na stejné ***úrovni*** (úrovni vnoření do stromu). Kořen je přitom na 1. úrovni, jeho přímí následníci pak na 2. úrovni atd. Maximální úroveň, které dosahuje některý prvek stromu, nazýváme ***hloubka stromu*** (méně často výška stromu).

Dalším významným pojmem je ***délka cesty*** k prvkům. Je to počet spojnic prvků (hran) nebo prvků (uzlů, vrcholů), které musíme projít na nejkratší cestě ke kořenu stromu. Pro co nejrychlejší přístup k prvku je třeba, aby každý měl co nejkratší cestu. Pokud sečteme délku cest všech prvků, dostaneme ***délku cesty stromu*** (délku vnitřní cesty stromu). Je zřejmé, že ***minimální délku*** bude mít strom tehdy, jestliže na každé úrovni (kromě poslední) bude mít maximální možný počet prvků. Takový strom má současně ***minimální hloubku*** (výšku) takže při použití rekurzívních algoritmů je nejmenší nebezpečí přetečení zásobníků.

Základní manipulace se stromy je opět vkládání, vyhledávání, a rušení prvků. I když své uplatnění mají i statické stromy, budeme se zabývat především dynamickými stromy, jejichž struktura se během zpracování mění. Vzhledem k vlastnostem stromů, které lze definovat rekurzívně, je možno velmi úspěšně řešit operace na stromech pomocí rekurzívních algoritmů. Na známé nebezpečí ***rekurze*** (přetečení zásobníku) přitom samozřejmě nezapomínáme.

Stromy mají řadu aplikací, například ***rozhodovací stromy***, u kterých uzly představují dílčí řešení. Jejich aplikací je strom Huffmanova kódu, který se používá jako jedna z metod komprese dat. V dalším textu budeme rozebírat ***vyhledávací stromy,*** které se využívají zejména při třídění dat.

4.2.1 Binární vyhledávací stromy

Vyhledávací stromy jsou stromy, u kterých zařazujeme prvky podle klíčové položky tak, že levý následník má vždy nižší hodnotu klíče než prvek a pravý následník hodnotu vyšší. Zařazení prvku je obvykle spojeno s jeho vyhledáním a zařazuje se jen tehdy, pokud nebyl nalezen.

Algoritmus ***vyhledání a vložení prvku*** do stromu budeme realizovat rekurzívní procedurou. Předávat jí budeme odkaz na aktuální prvek stromu. Protože budeme do stromu také vkládat, musíme předávat skutečně odkaz na prvek a ne pouze jeho hodnotu.

```
hledej (klíč, REF prvek)
  w = prvek
  if w = nil
        new(w)
        w↑.klíč = klíč
        w↑.data = data
        w↑.levý = nil
        w↑.pravý = nil
  else
        if klíč < w↑.klíč
              hledej (w↑.levý)
        else
              if klíč > w↑.klíč
                    hledej (w↑.pravý)
              else
                    zpracování dat w↑.data
```

```
                  end if
            end if
      end if
    end hledej
```

Vyhledávání i vkládání bude probíhat rychleji než u lineárního seznamu. Pokud však budou prvky vstupovat již seřazené (ať vzestupně nebo sestupně) degeneruje strom v lineární seznam. Okamžitě se objevuje také nebezpečí přetečení zásobníku při rekurzívním volání procedury ***hledej***. Jak toto nebezpečí odstranit budeme řešit v kapitole věnované ***vyváženým stromům***.

Nalezení konkrétního prvku v binárním stromu, je možno realizovat také iteračním algoritmem. Stejně jako u lineárních seznamů používáme logickou proměnnou ***h*** k ukončení vyhledávání. Po skončení algoritmu obsahuje proměnná ***w*** odkaz na hledaný prvek nebo hodnotu ***nil***, pokud ve stromu hledaný prvek není.

```
    h = true
    w = kořen
    while w <> nil and h
      if w↑.klíč = klíč
            h = false
      else
            if w↑.klíč > klíč
                  w = w↑.levý
            else
                  w = w↑.pravý
            end if
      end if
    end while
```

Uvedený postup je možno použít i při převodu rutiny ***hledej(REF prvek)*** z rekurzívního na iterační tvar.

Sestavený strom budeme často ***procházet*** a ***zpracovávat***. Opět je vhodný rekurzívní algoritmus. Pokud chceme zpracovávat prvky v jejich seřazeném pořadí, můžeme použít následující proceduru.

```
    projdi (prvek)
      if prvek <> nil
            projdi (prvek↑.levý)
            zpracování dat prvek↑.data
            projdi (prvek↑.pravý)
      end if
    end projdi
```

Tento algoritmus snadno upravíme pro ***zrušení celého stromu***. Je však nutno nejprve projít a zrušit oba následníky (a také části stromu, které na ně navazují, neboli oba podstromy) a pak zrušit tento prvek. Opět nesmíme zapomenout předávat prvky odkazem.

```
    zruš (REF prvek)
      if prvek <> nil
            projdi (prvek↑.levý)
            projdi (prvek↑.pravý)
            dispose(p)
      end if
    end zruš
```

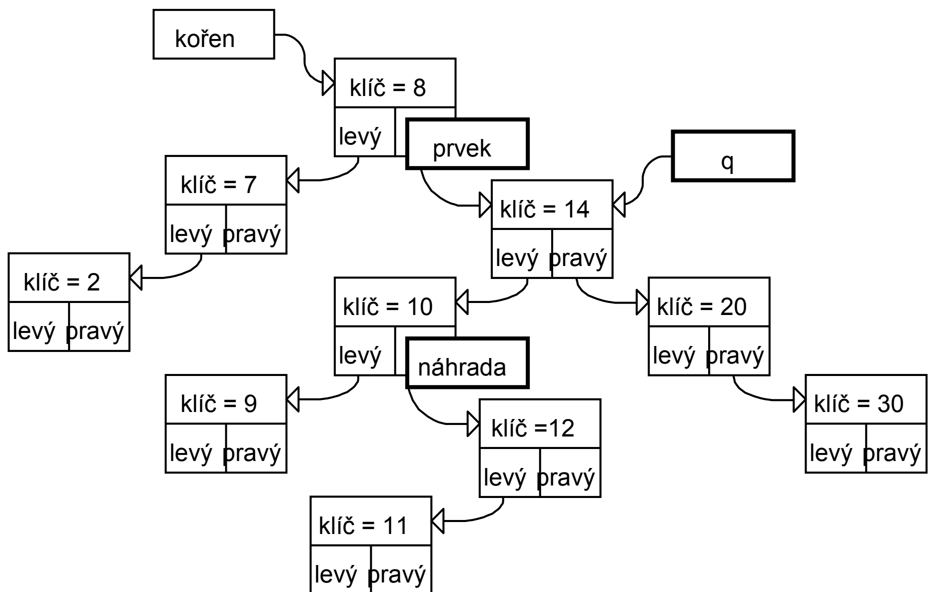
Inverzní problém k vyhledávání a vkládání prvků je ***vyhledávání a rušení prvků***. Jeho řešení je výrazně složitější. Musíme rozlišovat následující případy:

prvek s hledaným klíčem ve stromě není, pak vyhlásíme chybu, nebo prostě neuděláme nic,

prvek s hledaným klíčem má nejvýše jednoho následníka, pak na místo nalezeného prvku přesuneme tohoto následníka (a s ním spojený podstrom),

prvek s hledaným klíčem má dva následníky, pak nalezený prvek nahradíme buď nejpravějším prvkem jeho levého podstromu (nebo nejlevějším prvkem pravého podstromu), viz obr. 4.16.

Všimněte si zejména toho, že díky předávání proměnných odkazem způsobí změna obsahu proměnné také změnu struktury stromu, protože proměnná je totožná s konkrétním ukazatelem některého prvku stromu. Pro zvýraznění byly proměnné algoritmu orámovány tučnou čarou.



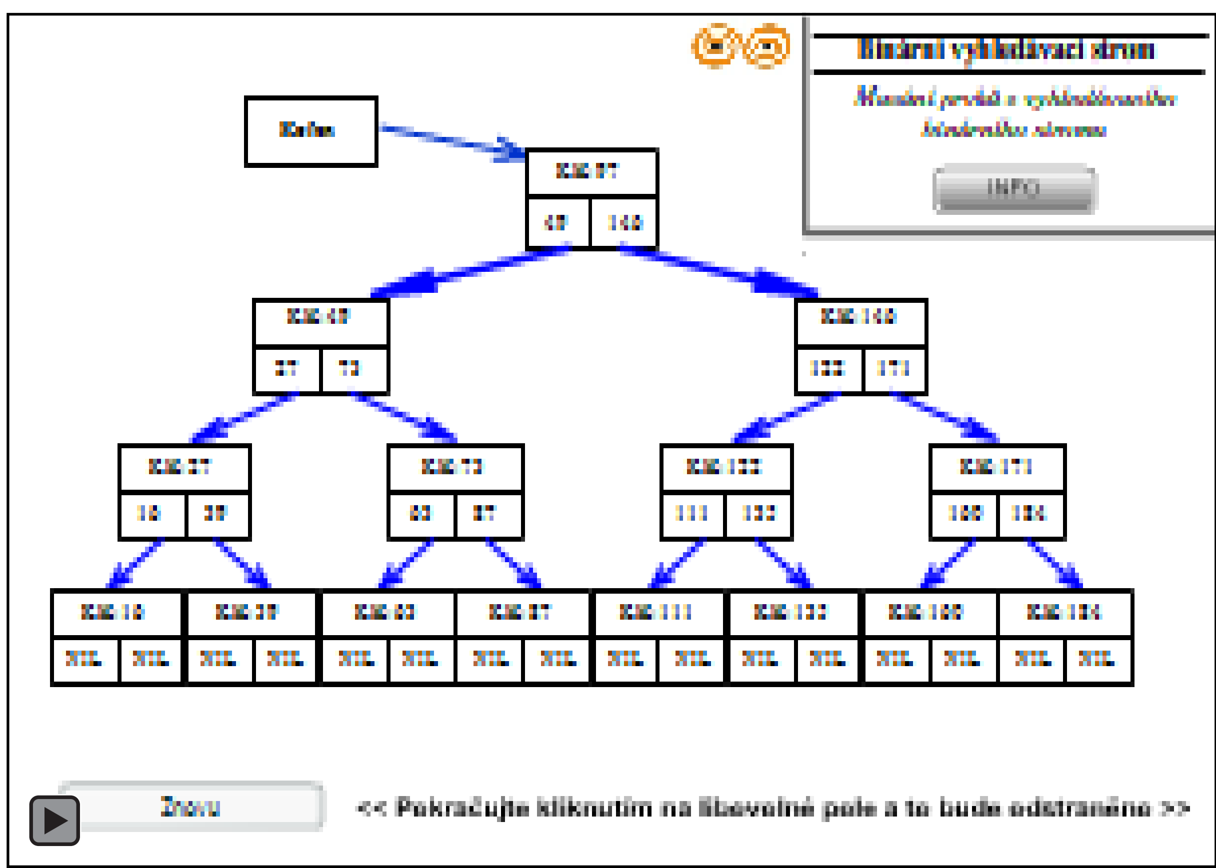
Obrázek 4.16 Stav před zrušením prvku 14 a jeho náhradou prvkem 12

```
vezmi (klíč, REF prvek)
  if prvek = nil
        hledaný prvek ve stromu není
  else
        if klíč<prvek↑.klíč
              vezmi (klíč, prvek↑.levý)
        else
              if klíč>prvek↑.klíč
                    vezmi (klíč, prvek↑.pravý)
              else
                    hledaný prvek byl nalezen
                    q = prvek
                    if q↑.pravý = nil
                          prvek = q↑.levý
                    else
                          if q↑.levý = nil
                                prvek = q↑.pravý
                          else
                                najdi (q↑.levý)
```

```
                    end if
                 end if
                 dispose (q)
              end if
           end if
        end if
     end vezmi
```

Jak vidíme, pro nejsložitější případ je použita procedura ***najdi***, která do prvku ***q*** dosadí nejpravější koncový prvek z levého podstromu prvku ***q*** (proměnná ***q*** tedy musí být deklarována jako globální, jinak ji musíme předávat jako parametr procedury, samozřejmě odkazem).

```
najdi (REF náhrada)
  if náhrada↑.pravý<>nil
        najdi (náhrada↑.pravý)
  else
        q↑.klíč = náhrada↑.klíč
        q↑.data = náhrada↑.data
        q = náhrada
        náhrada = náhrada↑.levý
  end if
end najdi
```



4.2.2 Vyvážené stromy

V předchozí kapitole bylo upozorněno na nebezpečí degenerování ***binárního vyhledávacího stromu*** na ***linenární seznam*** a s tím spojené nebezpečí havárie rekurzívního algoritmu.

Nejlepším řešením by byla konstrukce stromu s ***minimální hloubkou***, neboť hloubka stromu určuje také počet vnoření při volání rekurzívního algoritmu.

Nejjednodušeji toho dosáhneme, když budeme jednotlivé prvky umisťovat rovnoměrně na levou i pravou stranu stromu. Tento postup nazýváme ***strukturování stromu***. Pro známý počet vrcholů n můžeme definovat strom s minimální hloubkou pomocí tří pravidel:

1. zvolíme jeden z prvků jako kořen stromu,
2. vytvoříme levý podstrom s počtem ***nl = n div 2*** prvků.
3. vytvoříme pravý podstrom s počtem ***nr = n - nl -1*** prvků.

Výsledný strom bude ***dokonale vyvážený,*** neboť pro každý prvek platí, že počet prvků v jeho levém a pravém podstromu se liší nanejvýš o 1. V praxi je použití dokonale vyváženého stromu dosti problematické. Obvykle předem neznáme počet prvků a navíc požadujeme tvorbu ***vyhledávacího stromu***, jehož prvky nejsou umístěny náhodně. Pak bychom při každém vložení nového prvku do vyhledávacího stromu (tzv. ***náhodném vkládání***) museli provést restrukturalizaci stromu, která je velmi neefektivní.

Ze slepé uličky se dostaneme, pokud přijmeme méně přísnou definici ***vyváženého stromu***, jak ji formulovali Adelson-Velskij a Landis. Zní: „***Strom je vyvážený právě tehdy, když se výšky dvou podstromů každého prvku liší nejvýše o 1***“. Takto konstruované stromy nazýváme ***AVL-stromy***, my je budeme dále označovat jako vyvážené stromy, neboť se jinými zabývat nebudeme.

Autoři dokázali, že AVL-strom bude v nejhorším případě o 45% vyšší, než dokonale vyvážený strom se stejnými prvky. Přitom je možno se složitostí ***O(log n)*** provést všechny základní operace, tedy vložení, vyhledání i zrušení prvku s daným klíčem.

4.2.2.1 Vkládání prvku do AVL-stromu

Při tvorbě vyváženého stromu budeme postupovat tak, že po vložení nového prvku vždy, když je to nutné, opravíme vyváženost stromu. Vložením nového prvku do levého podstromu konkrétního prvku mohou nastat následující případy:

1. vyváženost se zlepší, pokud výška levého byla menší než u pravého podstromu,
2. vyváženost se zhorší, ale neporuší, pokud byly obě výšky stejné,
3. vyváženost se poruší, pokud výška levého podstromu byla větší než u pravého, vyváženost je třeba upravit.

K opravě vyváženosti stromu slouží ***rotace prvků***. Při nich samozřejmě nesmí dojít k porušení návaznosti klíčů u prvků vyhledávacího stromu.

K rozhodnutí o tom, která situace nastala, potřebujeme informaci o předchozím stavu vyváženosti stromu. Zjišťovat ji přímo ze struktury stromu je značně nepraktické, výhodnější bude doplnit datovou oblast prvků o složku ***vaz*** nabývající hodnot {-1, 0, 1} s následujícím významem:

-1 - levý podstrom je delší,

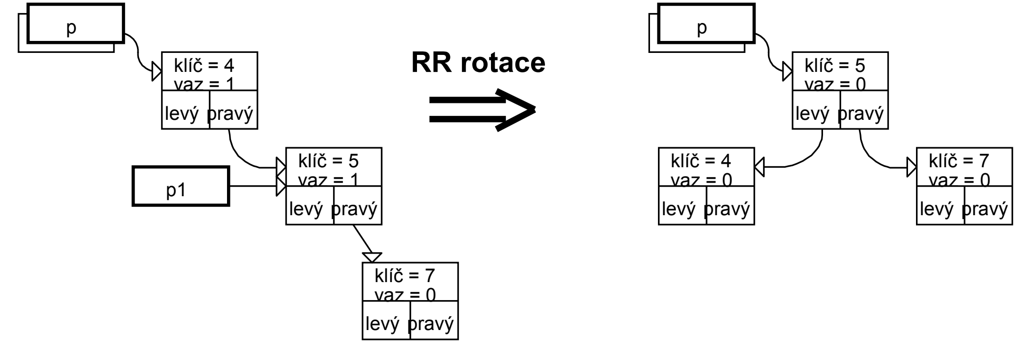
0 - oba podstromy mají stejnou délku,

1 - pravý podstrom je delší.

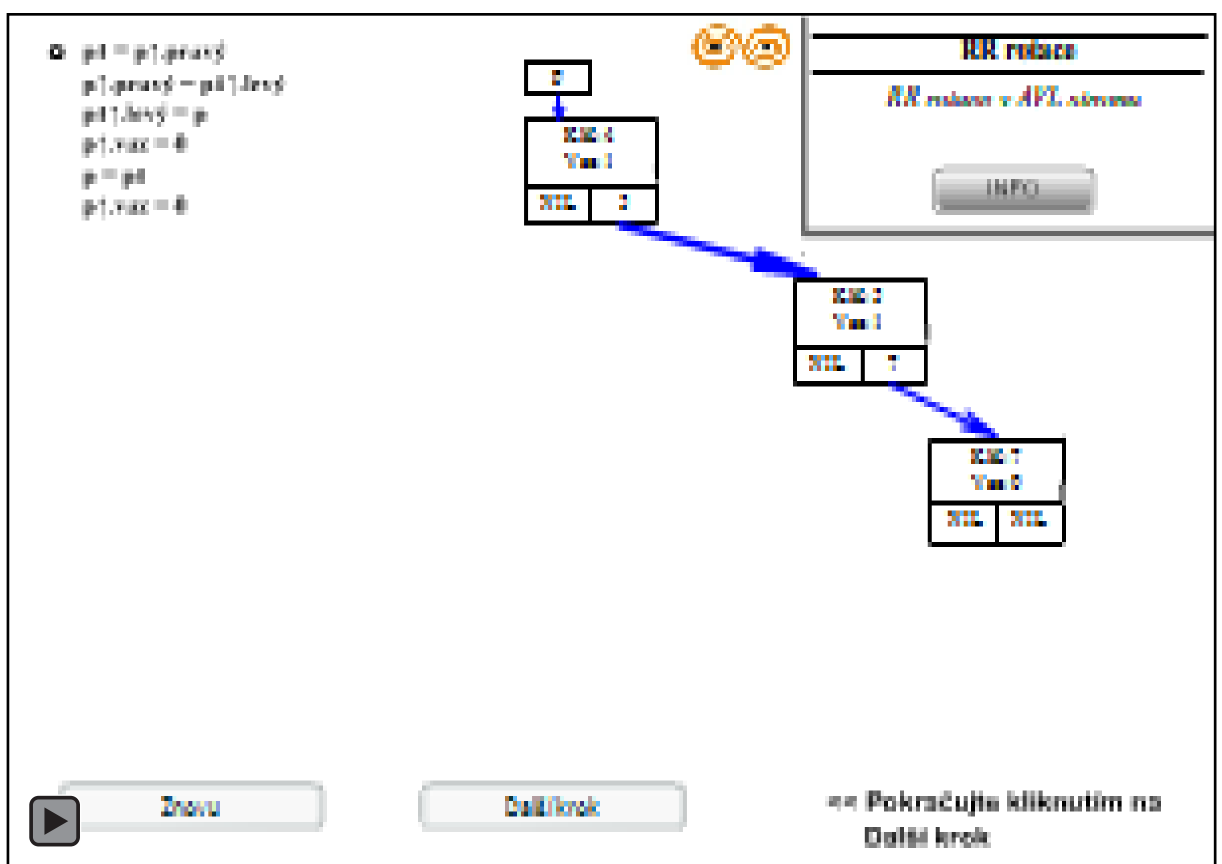
Po vložení nového prvku opravíme hodnotu proměnné ***vaz*** a v případě potřeby opravíme vyváženost stromu. Nesmíme zapomenout na rekurzívní volání, a proto musí rutina vracet informaci o tom, zda se vložením prvku zvětšila délka podstromu nebo ne. Využijeme k tomu logickou proměnnou ***h***, kterou budeme předávat odkazem, stejně jako aktuální prvek.

Ukažme si nyní, jaké případy nevyváženosti mohou nastat a jak je budeme řešit. Pro zjednodušení budeme kreslit jen vazby mezi prvky a u prvků hodnoty jejich klíčů a proměnné ***vaz***. Na prvek, u kterého došlo k porušení vyvážení, bude ukazovat proměnná ***p***, předávaná odkazem, ostatní proměnné budou pomocné.

Do prázdného stromu vložíme nejprve prvek s klíčem 4, ten bude kořenem, pak prvek s klíčem 5. Vložením prvku s klíčem 7 dojde k porušení vyvážení u prvku s klíčem 4. Úpravy dosáhneme jednoduchou RR rotací pravého podstromu.

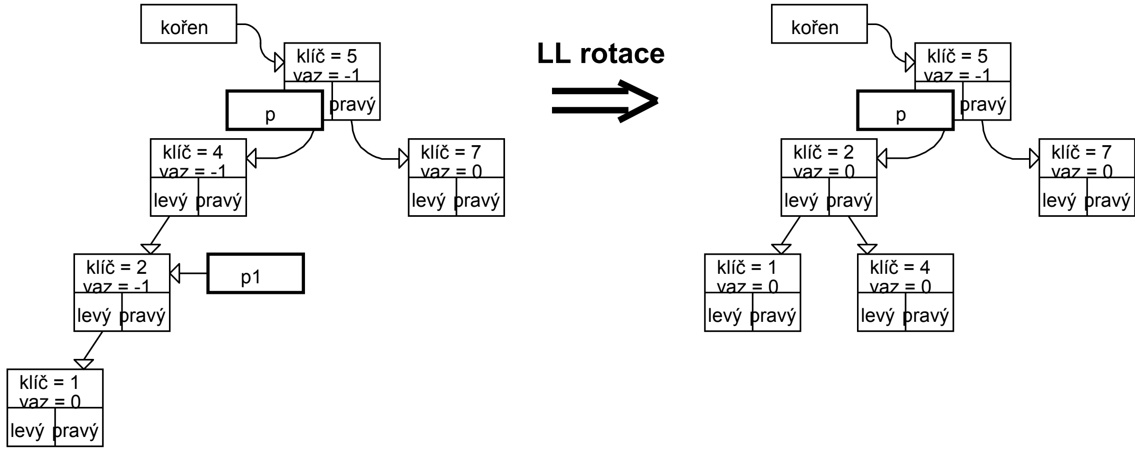


Obrázek 4.17 Rotace RR

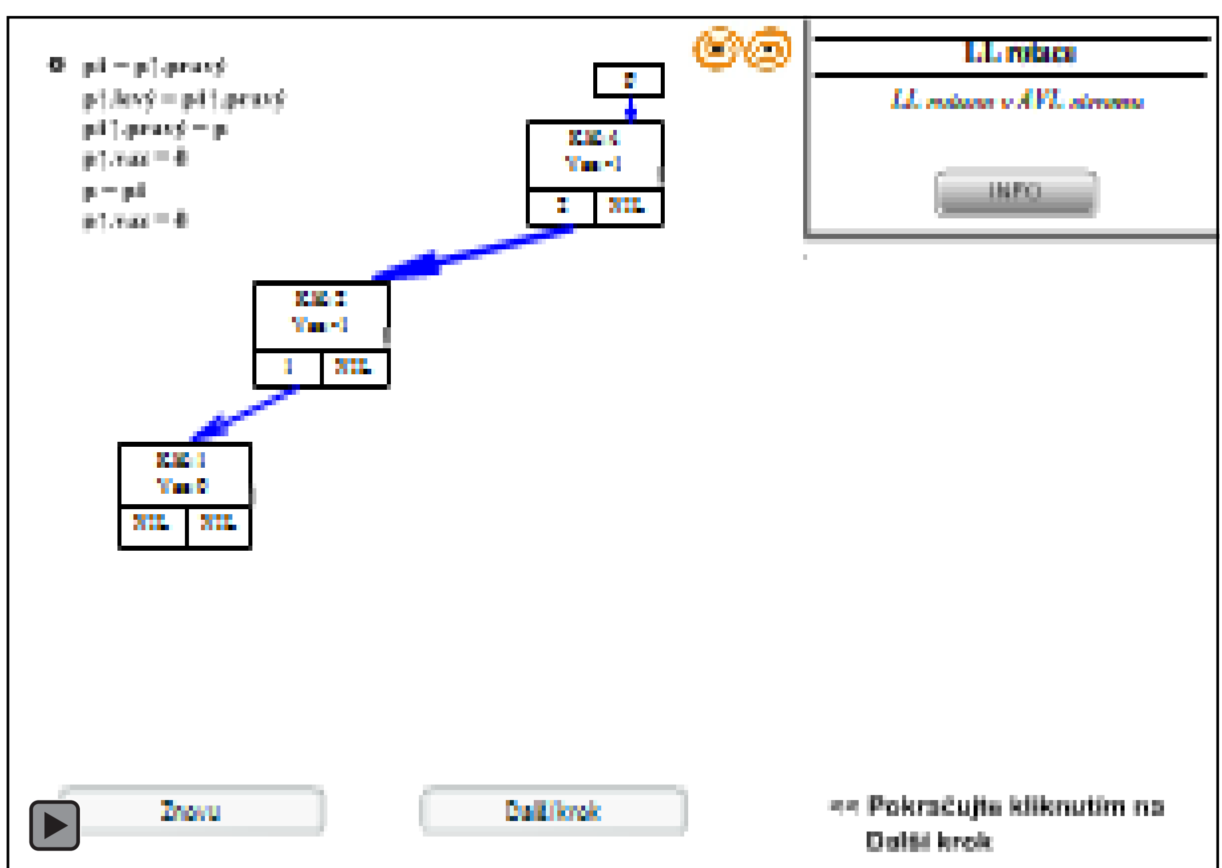


```
p1 = p↑.pravý
p↑.pravý = p1↑.levý
p1↑.levý = p
p↑.vaz = 0
p = p1
p↑.vaz = 0
h = false
```

Dalším vložením prvku s klíčem 2 a pak 1 dojde k podobné situaci v levé větvi opět u prvku s klíčem 4. Opravu dosáhneme jednoduchou rotací LL.

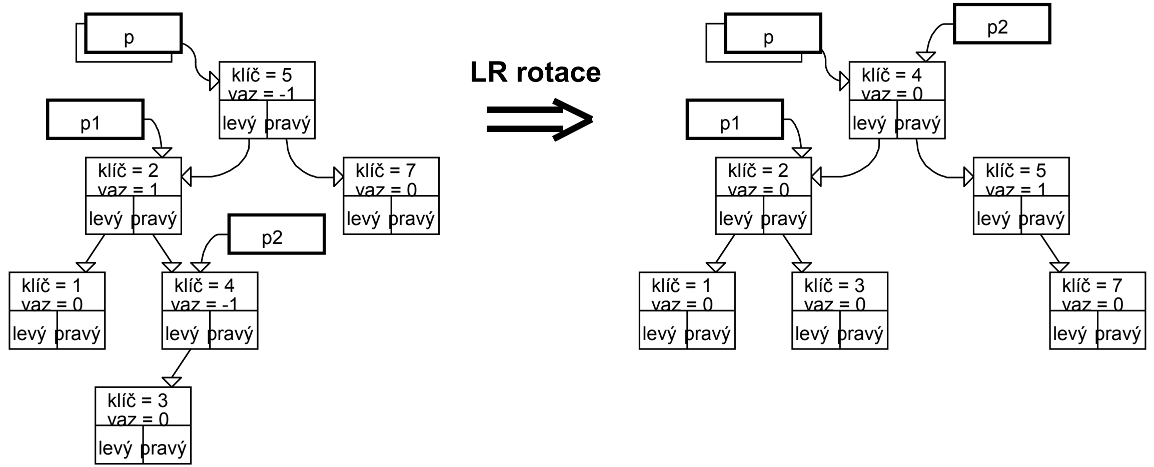


Obrázek 4.18 Rotace LL

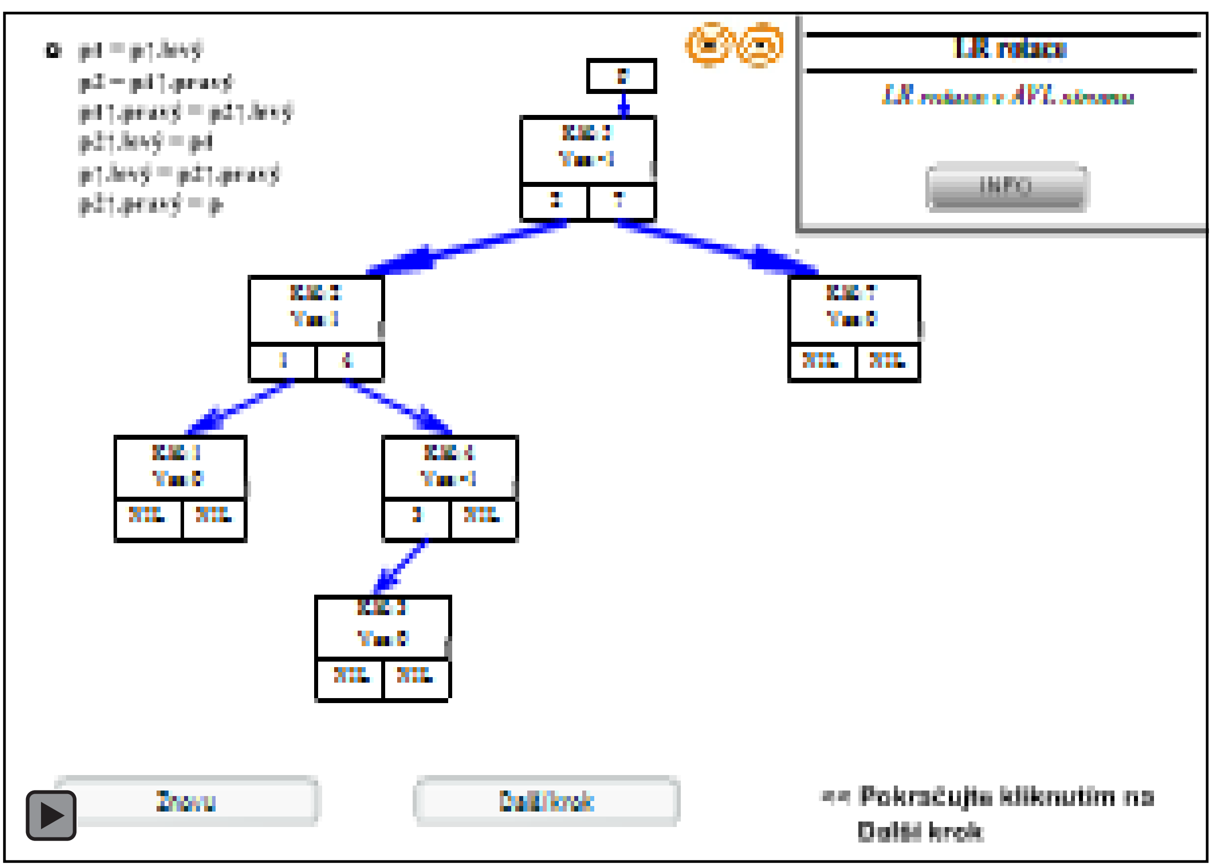



```
p1 = p↑.pravý
p↑.levý = p1↑.pravý
p1↑.pravý = p
p↑.vaz = 0
p = p1
p↑.vaz = 0
h = false
```

Jako další vložíme prvek s klíčem 3. Tím se poruší vyváženost u prvku s klíčem 5. Tentokrát je situace složitější. K opravě je nutná dvojitá LR rotace



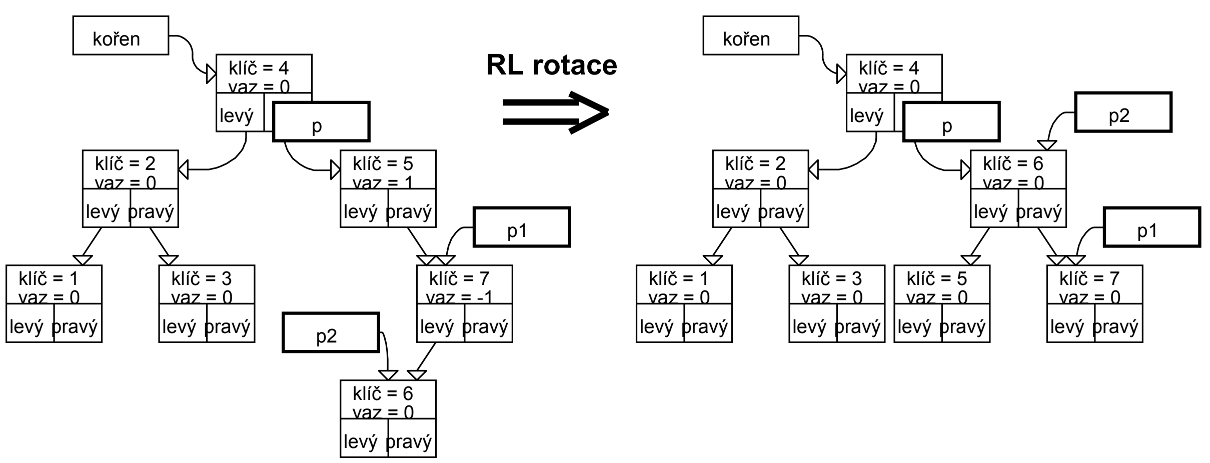
Obrázek 4.19 Dvojitá rotace LR



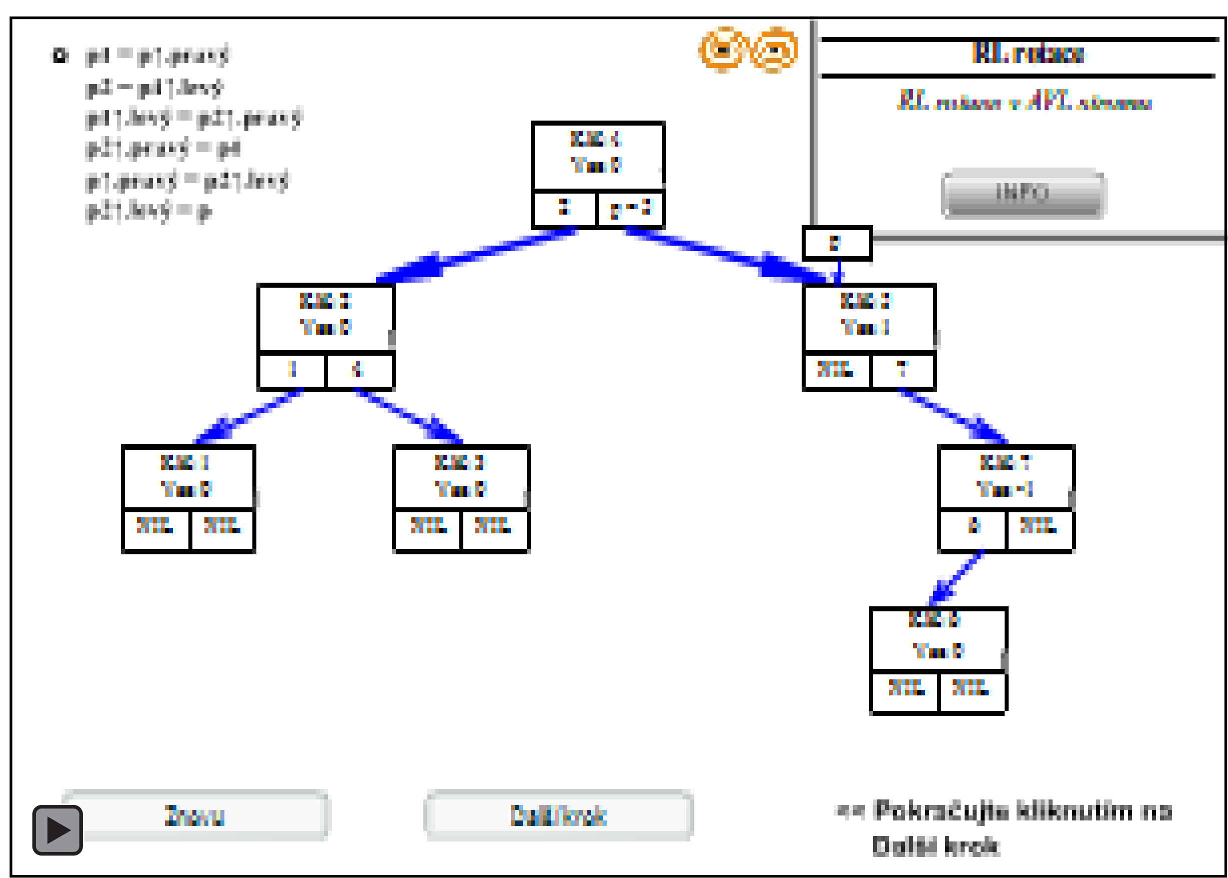

```
p1 = p↑.levý
p2 = p1↑.pravý
p1↑.pravý = p2↑.levý
p2↑.levý = p1
p↑.levý = p2↑.pravý
p2↑.pravý = p
if p2↑.vaz = -1
  p↑.vaz = 1
else
  p↑.vaz = 0
```

```
  end if
  if p2↑.vaz = 1
    p1↑.vaz = -1
  else
    p1↑.vaz = 0
  end if
  p = p2
  p↑.vaz = 0
  h = false
```

Poslední úpravou bude vložení prvku s klíčem 6, které způsobí porušení vyvážení u prvku s klíčem 5. Nutná je dvojitá rotace RL.



Obrázek 4.20 Dvojitá rotace RL



```
p1 = p↑.pravý
p2 = p1↑.levý
p1↑.levý = p2↑.pravý
p2↑.pravý = p1
p↑.pravý = p2↑.levý
p2↑.levý = p
if p2↑.vaz = 1
  p↑.vaz = -1
else
  p↑.vaz = 0
end if
if p2↑.vaz = -1
  p1↑.vaz = 1
else
  p1↑.vaz = 0
end if
p = p2
p↑.vaz = 0
h = false
```

Z rozboru jednotlivých situací je zřejmé také rozpoznání jednoduché a dvojité rotace. Pokud mají na počátku rotace prvky ***p*** a ***p1*** stejnou hodnotu ***vaz***, jedná se o jednoduchou rotaci, jinak je potřebná dvojitá rotace. Nyní již můžeme sestavit celou rekurzívní rutinu pro vyhledávání a vkládání prvků do vyváženého stromu. U popisů jednotlivých typů rotací byly orámovány ty části algoritmů, které nyní uvádíme pouze názvem rotace.

```
hledej1 (REF p, REF h)
if p = nil
  vložení nového prvku do proměnné p, h = true
else
  if klíč < p↑.klíč
        hledej1 (p↑.levý,h)
        if h
              case p↑.vaz = 1
                    p↑.vaz = 0
                    h = false
              case p↑.vaz = 0
                    p↑.vaz = -1
              case p↑.vaz = -1
                    p1 = p↑.levý
                    if p1↑.vaz = -1
                          rotace LL
                    else
                          rotace LR
                    end if
                    p↑.vaz = 0
                    h = false
              end case
        end if
  else
        if klíč > p↑.klíč
              hledej1 (p↑.pravý,h)
```

```
                if h
                        case p1↑.vaz = -1
                                p1↑.vaz = 0
                                h = false
                        case p1↑.vaz = 0
                                p↑.vaz = 1
                        case p1↑.vaz = 1
                                p1↑.vaz = pravý
                                if p1↑.vaz = 1
                                        rotace RR
                                else
                                        rotace RL
                                end if
                                p1↑.vaz = 0
                                h = false
                        end case
                end if
        else
                zpracování dat p↑.data, h = false
        end if
  end if
end if
end hledej1
```

4.2.2.2 *Práce s AVL – stromy*

Vyhledávání prvků, stejně jako ***průchod vyváženým stromem*** je zcela shodné se stromy nevyváženými. Obdobně jako u nich je pak operace ***vyhledávání*** a ***rušení*** prvků výrazně náročnější než ***vyhledávání*** a ***vkládání***. Kromě zrušení nalezeného prvku je třeba provést nové vyvážení stromu.

4.2.3 Tisk struktury stromu

Jako samostatnou kapitolu nyní vkládáme řešení speciálního problému, kterým je zobrazení struktury stromu. Přitom chceme, aby bylo zřejmé, na jaké ***úrovni*** prvek leží a nejlépe také ***vazby*** mezi prvky. Nejčastější výstupní zařízení bude zřejmě monitor počítače, tiskárna či textový soubor. Úroveň na jaké se prvek nachází, můžeme naznačit jeho odsazením o příslušný počet mezer. Pro jejich počítání zavedeme proměnnou ***u***. Jednotlivé prvky budeme zobrazovat pod sebou na řádcích. Proto bude vhodné začít nejpravějším prvkem stromu a postupovat směrem k nejlevějšímu

```
zobraz (w, u)
  if w <>nil
        zobraz (w↑.pravý, u+1)
        tiskni u mezer a klíč w↑.klíč
        zobraz (w↑.levý, u+1)
  end if
end zobraz
```

Složitější situace nastane, pokud chceme graficky znázornit také ***vazby mezi prvky*** pomocí spojovacích čar. Při tisku informace o aktuálním prvku potřebujeme znát také počet přechodů na levý či pravý podstrom, kterými jsme se k prvku dostali. Ty musíme jednak předávat při volání procedury, k tomu použijeme znakovou proměnnou ***s***. Potřebujeme však znát všechny předchozí hodnoty. Pro předem neznámou hloubku stromu by bylo vhodné je ukládat do

lineárního seznamu. Pokud známe hloubku stromu, můžeme použít pole ***uk*** se znakovými prvky, označenými 1, 2 až po maximální hloubku stromu.

Pro volbu vykreslovaných znaků znázorňujících vazby prvků platí následující pravidla:

Na poslední úrovni budeme volit pro

P - pravý podstrom - znak ´┌´

C - kořen stromu - znak ´—´

L - levý podstrom - znak ´└´

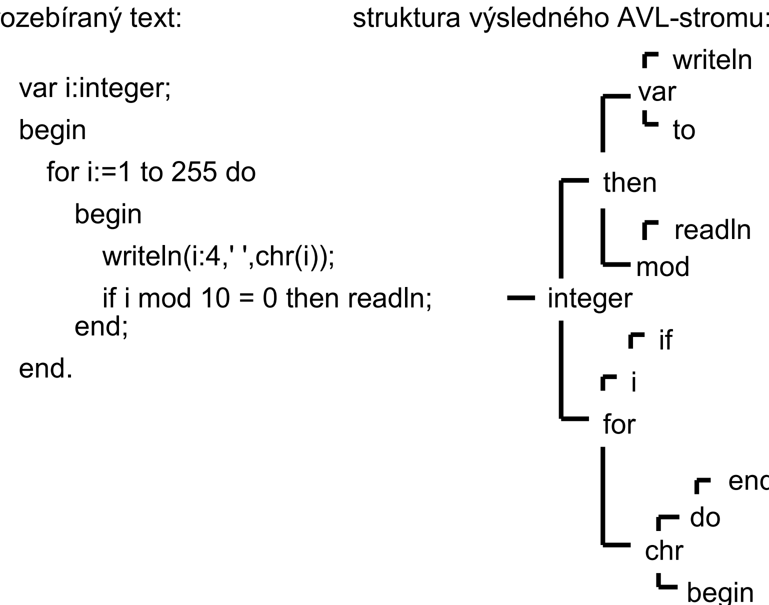
Na vyšších úrovních budeme volit mezi znakem mezera a ´|´, který prodlužuje spojení podstromu podle toho, jak jdou jednotlivá volání za sebou. Čáru kreslíme, pokud za Pravým podstromem následuje Levý a za Levým následuje Pravý, jinak kreslíme mezeru.

Proceduru pro zobrazení struktury celého stromu budeme volat ***zobraz(kořen, 1, ´C´)***

```
zobraz (w, u, s)
  if w <>nil
        uk[u] = s
        zobraz (w↑.pravý, u+1,´P´)
        s1 = ´ ´
        for i = 1 to u - 1 step 1
              if uk[i] = uk[i+1]
                    s1 = s1 + ´ ´
              else
                    s1 = s1 + ´|´
              end if
        end for
        case s = ´L´
              s1 = s1 + ´└´
        case s = ´P´
              s1 = s1 +´┌´
        case s = ´C´
              s1 = s1 + ´—´
        end case
        tiskni řetězec s1 a w↑.klíč
        zobraz (w↑.levý, u+1, ´L´)
  end if
end zobraz
```

Úpravu algoritmu pro ukládání informace o postupu stromem do ***lineárního seznamu*** ponecháváme na čtenáři. Je třeba si uvědomit, že při vstupu do procedury přidáváme prvek na konec seznamu a před jejím opuštěním ho opět vyjímáme. Protože víme, že zrušení posledního prvku vyžaduje přístup k předposlednímu prvku, bude vhodnější vkládat prvky na začátek lineárního seznamu, neboť rušení prvního prvku je velmi jednoduché. Jen nesmíme zapomenout, že informaci musíme zpracovávat od konce seznamu, což při použití rekurzívního algoritmu pro tvorbu řetězce ***s1*** nebude problém.

Výsledkem činnosti algoritmu bude zobrazení stromu, na které je třeba se dívat z pravé strany, aby pojmy levý a pravý podstrom odpovídaly zobrazení. Použitý přístup vychází ze zkušenosti, že obvykle mají stromy velké množství prvků při relativně malé ***hloubce*** (zvláště AVL – stromy):



V literatuře se můžeme setkat také se zobrazením stromu, při kterém jsou prvky na stejné úrovni vnoření umístěny na jedné vodorovné linii. K tomu je třeba zřetězit prvky na stejné úrovni a určit polohu na vodorovné linii. Výsledné zobrazení roste velmi rychle do šířky, což je nepraktické, proto tento algoritmus neuvádíme.

4.2.4 Optimální stromy

V úvodní kapitole jsme definovali ***délku cesty stromu*** (délku vnitřní cesty stromu) jako součet délek cesty ke všem prvkům stromu. Pro co nejrychlejší přístup k náhodně zvolenému prvku pak konstruujeme stromy s ***minimální délkou***. Přitom vycházíme z předpokladu, že pravděpodobnost výskytu je pro všechny prvky stejná.

V praxi se často setkáváme s aplikacemi, ve kterých se jednotlivé prvky vyskytují různě často, tedy ***pravděpodobnost výskytu*** prvků je různá. Jestliže pravděpodobnost přístupu k i-tému prvku stromu označíme p_i, pak platí

$$p_i \geq 0; \quad i = 1,2 \ldots n \,, \tag{4.1}$$

$$\sum_{i=1}^{n} p_i = 1 \,, \tag{4.2}$$

kde je n - počet prvků ve stromu.

Pravděpodobnosti výskytů prvků můžeme chápat také jako ***váhy*** těchto prvků. S jejich znalostí je vhodné konstruovat stromy s ***minimální váženou délkou cesty***:

$$P_I = \sum_{i=1}^{n} p_i h_i \longrightarrow \min \,, \tag{4.3}$$

kde je h_i - úroveň i-tého prvku stromu,

p_i - pravděpodobnost přístupu k i-tému prvku,

P_I - vážená délka cesty stromu.

V řadě aplikací určujeme pravděpodobnost výskytu prvku jako poměr počtu výskytů tohoto prvku k počtu všech prvků v souboru prvků. Je zřejmě nutné, aby celkový počet prvků v souboru byl konečný.

$$p_i = \frac{m_i}{\sum_{i=1}^{n} m_i} \qquad (4.4)$$

kde je m_i - počet výskytů i-tého prvku,

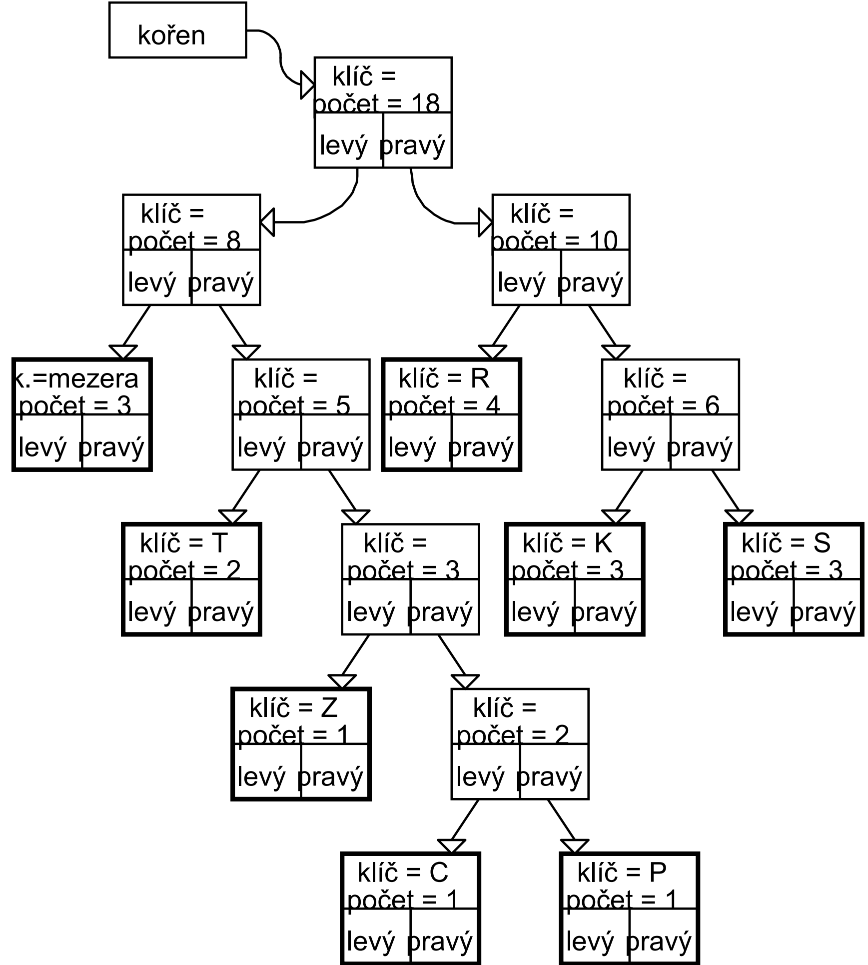
n - počet různých prvků v souboru.

Ze vztahu (4.4) je zřejmé, že beze ztráty obecnosti můžeme pracovat přímo s počtem výskytů prvku, pro který platí pouze podmínka (4.1), tedy $\boldsymbol{m_i \geq 0}$ pro $\boldsymbol{i = 1, 2, \ldots n}$, takže ***váženou délku cesty stromu*** můžeme v těchto aplikacích vyjadřovat upraveným vztahem (4.3) v podobě:

$$P_I = \sum_{i=1}^{n} m_i h_i \ . \qquad (4.5)$$

Strom, u kterého dosahuje ***vážená délka cesty*** minimální hodnoty, nazýváme ***optimální strom***. Jeho konstrukce je dosti komplikovaná a časově náročná, složitost algoritmu je řádově $\boldsymbol{O(n^2)}$ a velikost potřebné paměti (paměťová složitost) rovněž $\boldsymbol{O(n^2)}$[8](tento algoritmus při konstrukci uvažuje také pravděpodobnosti výskytu prvků, které se ve stromu nevyskytují). Pokud se spokojíme s kvazioptimálním stromem, můžeme využít velmi efektivní algoritmus, který má složitost ***O(n.log n)*** a paměťové nároky úměrné ***O(n)***.

Jako ukázku aplikace ***optimálního stromu*** nyní uvedeme algoritmus tvorby stromu pro konstrukci ***Huffmanova kódu***. Jedná se o ***prefixový kód*** (žádné klíčové slovo není začátkem jiného kódového slova) minimální délky. Kódová slova mají tím kratší délku, čím mají větší pravděpodobnost výskytu. Huffmanův kód je vhodné definovat pomocí optimálního stromu (pro binární kód rovněž binárním). Informaci přitom nesou jen ***koncové prvky***, všechny vnitřní prvky jsou jen pomocné, jinak by kód nebyl prefixový. Kódové slovo pak tvoříme podle cesty ke koncovému prvku - postup k levému následníku zastupujeme znakem 0, k pravému 1. Následující obrázek znázorňuje optimální strom pro konstrukci Huffmanova kódu pro soubor znaků: "STRC PRST SKRZ KRK"



Obrázek 4.21 Strom Huffmanova kódu

Každý koncový prvek stromu (je nakreslen tučnou čarou) nese informaci o znaku a počtu jeho výskytů, vnitřní prvek pak součet výskytů všech jeho následníků. Tabulka uvádí počty výskytů jednotlivých prvků a přiřazená kódová slova.

Tabulka 4.1

prvek	poč. výskytů	kód. slovo	prvek	poč. výskytů	kód. slovo
C	1	01110	S	3	111
K	3	110	T	2	010
P	1	01111	Z	1	0110
R	4	10	mezera	3	00

Využití Huffmanova kódu je jednou z metod ***komprese dat***. Znakům souboru přidělíme kódová slova Huffmanova kódu a s jejich pomocí zakódujeme. Protože pro každý soubor konstruujeme speciální kód, nesmíme opomenout informaci o vytvořeném souboru uložit do ***komprimovaného souboru***. Postup tvorby stromu Huffmanova kódu můžeme rozdělit do tří částí. Pro uložení informace o znacích a počtu jejich výskytu využijeme dynamickou datovou strukturu. Každý prvek bude obsahovat informaci:

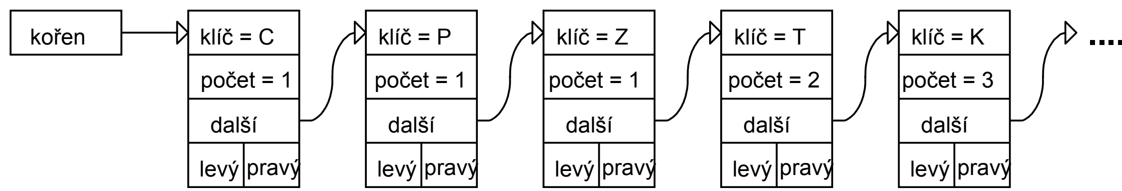
klíč = znak souboru (u koncových prvků),

počet = počet výskytů v souboru,

další = ukazatel na další prvek lineárního seznamu,

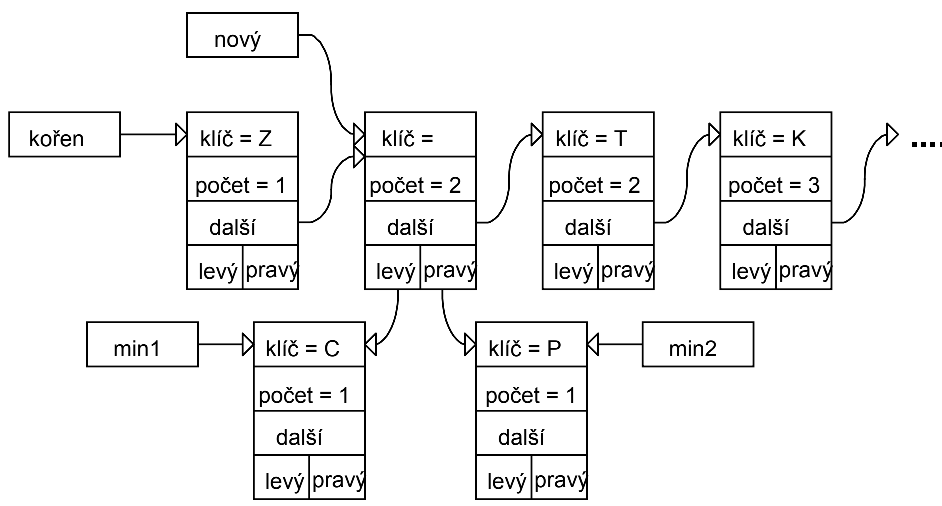
levý, pravý = ukazatele na následníky ve stromu.

4. Určíme, které znaky se v souboru vyskytují a kolikrát. Pokud je maximální počet různých znaků známý a nepříliš velký, můžeme využít statické ***pole*** (znaky obvykle kódujeme ASCII kódem, který obsahuje 256 kódových slov). Pro větší počet prvků (např. pro celá slova) bude vhodnější dynamická datová struktura. Můžeme využít ***lineární seznam***, nebo ***vyhledávací strom*** organizovaný podle klíče. (znaků).
5. Jednotlivé prvky seřadíme do ***lineárního seznamu*** podle počtu výskytů od nejmenšího k největšímu. Pokud jsme pro určení počtu výskytů prvků využili lineární seznam, provedeme jejich nové setřídění. Nejlépe pomocí stabilního algoritmu třídění. Tak zajistíme, že prvky se stejným počtem výskytů budou setříděny v definovaném pořadí. Nesmíme zapomenout položkám levý a pravý přiřadit hodnotu nil.



Obrázek 4.22 Lineární seznam seřazený podle počtu výskytů znaků

6. Pokud má lineární seznam více než jeden prvek (kořen↑.další <> nil), pak vyjmeme ze seznamu první dva prvky (označíme je ukazateli ***min1*** a ***min2***), vytvoříme nový prvek, prvky ***min1*** a ***min2*** uložíme jako jeho následníky. Počet výskytů nového prvku určíme jako součet výskytů jeho přímých následníků. Nakonec nový prvek zařadíme do lineárního seznamu a celou činnost zopakujeme.



Obrázek 4.23 Lineární seznam po vložení prvního vnitřního prvku budoucího stromu Huffmanova kódu

Algoritmus tvorby stromu Huffmanova kódu z lineárního seznamu:

```
if kořen <> nil
  while kořen↑.další <> nil
        min1 = kořen
        kořen = kořen↑.další
        min2 = kořen
        kořen = kořen↑.další
        new (nový)
        nový↑.levý = min1
```

```
            nový↑.pravý = min2
            nový↑.počet = min1↑.počet + min2↑.počet
            zařazení prvku nový do seznamu
        end while
    end if
```

4.2.5 B-stromy

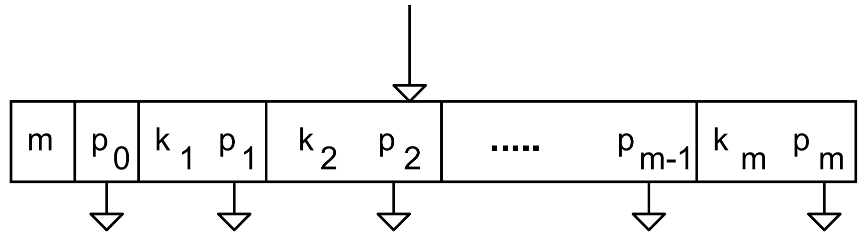
B-stromy jsou speciálním případem stromů ***vyššího stupně*** (každý prvek může mít více následníků). Uvádíme je pro jejich dobré vlastnosti, zejména jednoduché kritérium, usměrňující růst stromu. Navrhli je Rudolf Bayer & Ed McCreight v roce 1972, s názvem B jako „*ballanced*“, ale možná jako „*Bayer*“ nebo „*Boeing*“, neboť pracovali pro „*Boeing Scientific Research Labs*“.

Strom je složen ze ***stránek***. Každá stránka musí obsahovat ***n*** až ***2n*** prvků pro danou konstantu ***n***. (Kromě kořenové stránky, jak uvidíme dále.) Za těchto podmínek bude strom s ***N*** prvky a maximální velikostí stránky ***2n*** vyžadovat maximálně $\log_n N$ přístupů ke stránce pro vyhledání prvků. Přitom máme jistotu, že paměť bude využita nejméně z 50%.

I za uvedených podmínek budou algoritmy vyhledávání, vkládání a rušení prvků poměrně jednoduché. Datovou strukturu nazývanou B-strom definujeme následujícími pravidly. Parametr n přitom označujeme jako ***řád*** stromu.

7. Každá stránka obsahuje nejvýše ***2n*** prvků (klíčů).
8. Každá stránka kromě kořenové obsahuje alespoň ***n*** prvků.
9. Každá stránka je buď koncová (nemá následníky), nebo má ***m + 1*** následníků, kde ***m*** je počet prvků stránky.
10. Všechny koncové stránky jsou na stejné úrovni.

Pro uložení informace o jedné stránce zvolíme následující datovou strukturu. Informaci o jednotlivých prvcích zatím zúžíme na uložení hodnoty klíče, podle kterého je B-strom organizován.



Obrázek 4.24 Datová struktura stránky B-stromu

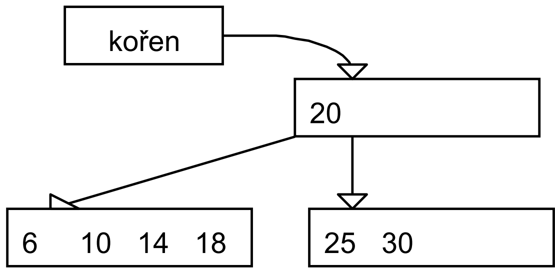
Vyhledávání prvku je rozšířením vyhledávání prvku v binárním stromu. Protože stránka obsahuje více než dva prvky, můžeme využít kromě ***sekvenčního prohledávání*** také rychlejší ***binární prohledávání***, zejména pro větší ***řády*** B-stromu. Pokud prvek najdeme v aktuální stránce, pak nastane jeden ze tří případů:

11. ***klíč*** $< k_1$ - vyhledávání pokračuje ve stránce p_0↑,
12. $k_i <$ ***klíč*** $< k_{i+1}$ - vyhledávání pokračuje ve stránce p_i↑,
13. $k_m <$ ***klíč*** - vyhledávání pokračuje ve stránce p_m↑.

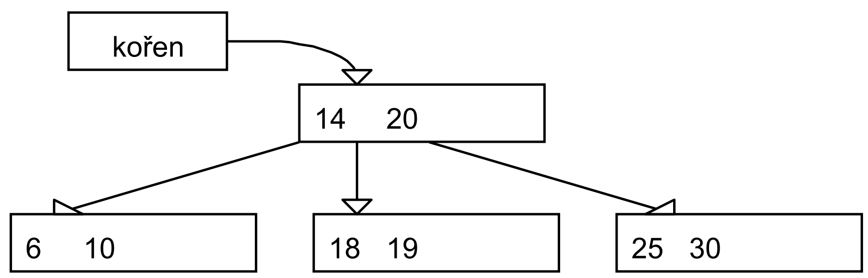
V případě, že ukazatel na další stránku má hodnotu nil, znamená to, že prvek ve stromu není a je nutno ho přidat do B-stromu. Přitom mohou nastat dva případy:

14. ***m < 2n*** - datová struktura není plná, pak se prvek přidá do této stránky.

15. ***m*** = ***2n*** - stránka je plná, rozdělí se na dvě stránky po ***n*** prvcích (jedna obsahuje prvních ***n*** prvků, druhá posledních ***n*** prvků). Zbývající prvek (ze středu posloupnosti prvků) je přesunut do vyšší stránky. Pokud dojde k přeplnění vyšší stránky, opět dojde k jejímu rozdělení. Tento proces může pokračovat až ke ***kořenové stránce***. Jejím rozdělením vzroste hloubka stromu (výška) a vznikne nová ***kořenová stránka*** obsahující jediný prvek. Jak vidíme, B-strom roste zvláštním způsobem, od koncových prvků ke kořenům.



Obrázek 4.25 B-strom před uložením prvku s klíčem 19



Obrázek 4.26 B-strom po vložení prvku s klíčem 19 - došlo k rozdělení plné stránky

Algoritmus vyhledávání a přidávání prvků do B-stromu je realizován jako procedura hledej. Vstupují do ní parametry ***klíč*** a aktuální stránka (***a***). Odkazem je předávána logická proměnná ***h***, která informuje o tom, že došlo k rozdělení stránky a v proměnné ***u*** je pak předáván prvek, který je přesouván do vyšší stránky.

Pro uložení stránky B-stromu využijeme následující datové struktury:

stránka - structure

m - počet prvků ve stránce

p0 - ukazatel na prvního následníka

l [1 až 2n] -prvky stránky

end structure

prvek - structure

klíč - položka klíče

data - ostatní data spojená s prvkem

p - ukazatel na další stránku

end structure

```
hledej (klíč, a, REF h,REF u)
if a = nil
  klíč není v B-stromu, prvku u přiřadíme nový klíč
  u.klíč = klíč nového prvku
```

```
    u.data = data nového prvku
    h = true
    prvek u bude putovat nahoru v B-stromu
  else
    hledáme prvek s udaným klíčem ve stránce a pomocí
  binárního prohledávání
    if prvek nalezen
          manipulace s daty nalezeného prvku a↑.e[i]
    else
          hledej (klíč, následník, h, u)
          if h
                prvek postupuje nahoru
                if a↑.m < 2n
                      prvek u přidáme do stránky a↑
                      h = false
                else
                      stránku a↑ rozdělíme na dvě stránky
                      střední prvek dosadíme do u
                      h = true
                end if
          end if
    end if
  end if
  end hledej
```

Již bylo zmíněno, že kořenová stránka má výhradní postavení, její dělení je nutno naprogramovat samostatně. Nejjednodušší bude zařadit proceduru ***hledej*** do procedury ***vkládání***. V případě, že proměnná ***h*** bude mít po návratu z procedury ***hledej*** hodnotu true, znamená to, že je nutno rozdělit kořenovou stránku. K tomu použijeme ukazatel ***q***. Spolu s proměnnými ***h*** (logická) a ***u*** (prvek stránky) může být definována v proceduře ***vkládání*** jako lokální.

```
  vkládání (klíč)
    hledej (klíč, kořen, h, u)
    if h
          q = kořen
          new (kořen)
          kořen↑.m = 1
          kořen↑.p0 = q
          kořen↑.e[1] = u
    REM prvek u obsahuje informaci o klíči a ukazatel
    REM na stránku následníků
    end if
  end vkládání
```

Průchod B-stromem je opět rozšířením průchodu lineárním stromem. Následující procedura zpracuje prvky B-stromu v pořadí hodnot jejich klíčů.

```
  průchod (p)
    if p <> nil
          průchod (p↑.p0)
          for i = 1 to p↑.m step 1
                zpracování dat p↑.e[i].data
                průchod (p↑.e[i].p)
          end for
```

```
    end if
  end průchod
```

Nejčastěji budeme rušit celý B-strom najednou. K tomuto účelu upravíme proceduru pro průchod stromem tím, že po zpracování celé stránky dojde k jejímu zrušení.

```
  zrušení (p)
    if p <> nil
          zrušení (p↑.p0)
          for i = 1 to p↑.m step 1
                zrušení (p↑.e[i].p
                zrušení dat p↑.e[i].data)
          end for
          dispose (p)
  end if
  end zrušení
```

Na závěr kapitoly věnované B-stromům se ještě zmíníme o řešení jednoho problému. Tím je konstrukce vyhledávacích stromů, které svým rozsahem přesahují ***velikost dostupné operační paměti***. V tomto případě je nutno strom ukládat na jiné paměťové médium, které obvykle neumožňuje současný přístup ke všem částem informace (typickým příkladem je ***sekvenční soubor***). B-stromy nám nabízejí dobré řešení. Do souboru budeme ukládat jednotlivé stránky B-stromu. Přitom využijeme skutečnost, že stránka má známou velikost. Jako hodnoty ukazatelů na stránky pak budeme používat pozici stránky v souboru. Nové stránky budou zřejmě ukládány na konec souboru, jinak by bylo nutno upravovat hodnoty ukazatelů na odsouvané stránky při vložení nové stránky do souboru. Velikost stránky (***řád*** B-stromu) budeme volit tak, aby bylo možno celou stránku přesunout do operační paměti. V proceduře ***hledej*** je orámována část algoritmu, při níž je potřeba mít v paměti aktuální stránku. Při tomto řešení můžeme pro vyhledání prvku ve stránce využít rychlejší ***binární prohledávání***. Pokud bychom i s aktuální stránkou pracovali v souboru, museli bychom využívat ***sekvenční prohledávání***.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ĺ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ĺ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms.* 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

5. LEKCE - TŘÍDĚNÍ

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

5 DYNAMICKÉ DATOVÉ STRUKTURY ... 3

5.1 Algoritmus přímého vkládání (Insertion sort) ... 5

5.2 Algoritmus přímého výběru (Selection sort) ... 8

5.3 Algoritmus třídění přímou výměnou (bublinkového třídění) a třídění přetřásáním (Bubble sort, Shake sort) ... 9

5.4 Třídění zmenšováním kroku (Shell sort) ... 11

5.5 Třídění distribučním čítáním (Distributing couting) ... 13

5.6 Algoritmus třídění rozdělováním (Quick sort) ... 15

5.7 Třídění hromadou (Heapsort) ... 18

5.8 Vyhledání *n*-tého nejmenšího prvku, vyhledání *n* nejmenších prvků ... 23

5.9 Třídění slučováním (Merge Sort) ... 24

5.9.1 Nerekurzivní verze algoritmu Merge sort ... 28

POUŽITÁ LITERATURA ... 31

5 DYNAMICKÉ DATOVÉ STRUKTURY

OBSAH KAPITOLY:

Algoritmus přímého vkládání (Insertion sort)

Algoritmus přímého výběru (Selection sort)

Algoritmus třídění přímou výměnou (bublinkového třídění) a třídění přetřásáním (Bubble sort, Shake sort)

Třídění zmenšováním kroku (Shell sort)

Třídění distribučním čítáním (Distributing couting)

Algoritmus třídění rozdělováním (Quick sort)

Třídění hromadou (Heapsort)

Vyhledání n-tého nejmenšího prvku, vyhledání n nejmenších prvků

Třídění slučováním (Merge Sort)

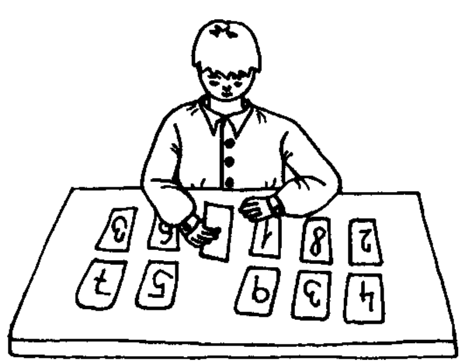
MOTIVACE:

Jednou z nejčastějších operací na seznamu prvků je jejich vyhledávání a třídění. Proto bylo postupně vyvinuto mnoho různých algoritmů vyhledávání a třídění. To nám umožňuje ukázat nejen samotné algoritmy, ale také ukázat na hodnocení jejich efektivity pomocí časové složitosti.

CÍL:

Seznámit se s nároky na algoritmy vyhledávání a třídění, umět je zařadit podle jejich vlastností mezi interní a externí, rozhodnout zda jsou stabilní a zda jsou přirození. Umět zapsat stabilní algoritmy interního třídění přímých vkládáním, přímým výběrem a přímou výměnou, nestabilní algoritmy – třídění zmenšováním kroku, třídění rozdělováním, třídění hromadou. Pro speciální případy umět použít specifické algoritmy jako je algoritmus rozdělování distribučním čítáním. Umět upravit známě algoritmy pro plnění specifických požadavků jako je hledání průměru nebo mediánu. Umět aplikovat algoritmy externího třídění po interní třídění.

Jednou z nejčastějších operací na seznamu prvků je jejich setřídění. V každém takovém seznamu však musí být nejdříve zadána hodnota, podle které bude třídění probíhat. Tato hodnota, často celé číslo, se nazývá ***klíč.*** Obecně je prvek seznamu typu struktura a klíč je jednou z položek této struktury. Jindy však může být klíč vypočítán na základě položek struktury prvků seznamu. V následujících algoritmech však budeme předpokládat, že klíč prvku je již vypočten a umístěn do jedné z položek struktury a bude mít celočíselnou hodnotu. To proto, aby byly na první pohled jasné operace <, >, <=, >=, ≠, =.



Obrázek 5.1 Třídění

Hovoříme-li o třídění seznamu prvků, můžeme použité algoritmy rozdělit do dvou skupin podle toho, jak pracují s pamětí počítače. V případě, že objem tříděných dat je větší než kapacita operační paměti počítače, může být v paměti vždy umístěna jistá část seznamu a zbytek je dočasně uchován na externím paměťovém médiu. V tomto případě se jedná o tzv. ***vnější třídění (externí)***.

V případě, že se všechna data vejdou do operační paměti počítače, jde o ***třídění vnitřní (interní)***. Následující kapitoly se budou zabývat oběma případy uložení dat. Většina algoritmů však bude věnována vnitřnímu třídění. U všech algoritmů budeme sledovat dvě měřítka efektivity algoritmů:

počet porovnání klíčů

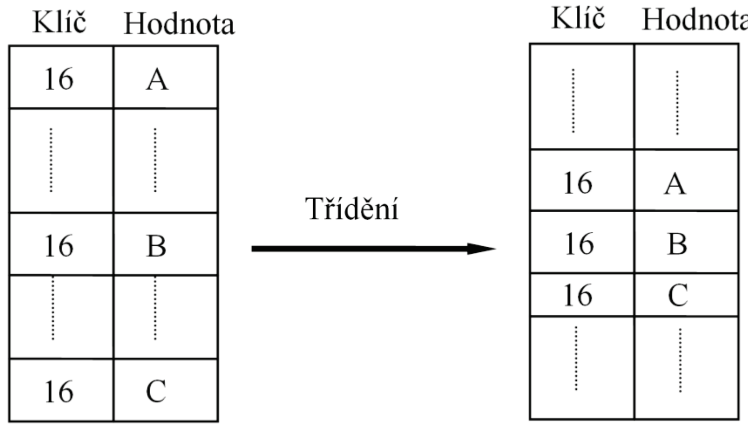
počet přesunů prvků seznamu

Obě operace totiž velice významně ovlivňují délku běhu programu. Vypočtená složitost algoritmu však nebude odrážet prolog a epilog algoritmu, který může být časově náročný, avšak není součástí popisovaného algoritmu.

Ve všech algoritmech budeme uvažovat následující stejné podmínky:

1. uvažujeme, že klíče jsou celá čísla,
2. zaměříme se na manipulaci s těmito klíči,
3. předpokládáme, že seznam klíčů je uložen na pozicích **[1 .. *n*]** pole ***a*** (někdy na pozicích **[0 .. (*n* - 1)]**.

U každého algoritmu budeme dále sledovat, zda zachovává vzájemnou pozici prvků totožných klíčů. Pokud algoritmus tuto pozici zachovává, označujeme jej za ***stabilní***.



Obrázek 5.2 Stabilní algoritmus třídění

Na obr. 5.2 je zobrazen příklad stabilního algoritmu třídění, který zachovává pořadí prvků s hodnotami A, B, C. Prakticky je nutné stabilitu sledovat při třídění seznamů se ***složeným klíčem***. Tím může být např. seznam lidí, kdy prvním klíčem je příjmení a druhým jméno. Po setřídění podle jmen musí dojít ještě k setřídění podle příjmení. Pokud by algoritmus neuchovával vzájemnou pozici totožných klíčů, výsledek třídění by byl nekorektní.

5.1 ALGORITMUS PŘÍMÉHO VKLÁDÁNÍ (INSERTION SORT)

Základní myšlenka: ze seznamu postupně vybíráme prvky, které postupně zařazujeme na správná místa. Zařazování se provádí tak, že postupně porovnáváme vybraný prvek s prvky vlevo od jeho pozice. Jakmile je nalezeno místo pro vložení, všechny prvky seznamu od místa vložení směrem doprava jsou posunuty o jednu pozici, na takto uvolněnou se vloží právě jmenovaný prvek.

Prakticky se algoritmus realizuje nejlépe následujícím způsobem: Vkládaný prvek je na ***i***-tém místě v seznamu. Porovnání se provádí postupně s prvky ležícími vlevo od pozice ***i***; levá část seznamu je již setříděná a proto je možno do ní prvek zařadit. Je-li při porovnání klíč vkládaného prvku větší nebo roven svému levému sousedovi, je vkládaný prvek na správném místě. V opačném případě prohodíme prvky na pozicích ***i*** a **(*i* - 1)** a porovnání provedeme s dalším prvkem vlevo.

Při tvorbě algoritmu je nutné také uvažovat případ, kdy vkládaný prvek bude umístěn na začátek seznamu. Pro tento případ musí být na začátku seznamu vytvořena jistá zarážka, která zamezí podtečení pole. Tato situace se dá řešit dvěma způsoby:

4. umístěním prvku s klíčem o nejmenší možné hodnotě na pozici ***a*****[0]**
5. umístěním vkládaného prvku na pozici ***a*****[0]**.

Při prohledávání pak program skončí svou práci při nalezení správné pozice nebo doiterováním na pozici ***i* = *0***. Následující algoritmus předpokládá, že tříděné prvky jsou umístěny v poli ***a*** na pozicích **[1 .. *n*]** a na pozici ***a*****[0]** je umístěn prvek s nejmenším možným klíčem.

```
for i = 2 to n
  v = a[i]
  j=i
  while a[j-1] > v
        a[j] = a[j-1]
        j = j-1
```

```
    end while
    a[j] = v
  end for
```

Analýza algoritmu přímého vkládání.

Algoritmus přímého vkládání má jak nejhorší případ, tak nejlepší případ. Nejhorší nastane, pokud zdrojová posloupnost prvků bude uspořádána v opačném pořadí. Pak v ***i***-tém kroku bude nutné provést ***i* - 1** porovnání a ***i*** přesunů včetně uložení vkládaného prvku.

Celkový počet porovnání pro nejhorší případ je tedy:

$$C_{WC} = \sum_{i=2}^{n} C_i = \sum_{i=2}^{n} (i-1) = \frac{n^2 - n}{2} \tag{5.1}$$

a počet přesunů je

$$M_{WC} = \sum_{i=2}^{n} M_i = \sum_{i=2}^{n} i = \frac{n^2 + 3n}{2} \tag{5.2}$$

Nejlepší případ nastane, pokud je zdrojová posloupnost již uspořádána. Pak je zapotřebí vždy 1 porovnání a 2 přesuny.

Celkový počet porovnání je tedy

$$C_{BC} = \sum_{i=2}^{n} 1 = n - 1 \tag{5.3}$$

a počet přesunů

$$M_{BC} = 2(n-1) \tag{5.4}$$

Budeme-li chtít vyčíslit průměrný případ, kdy všechny permutace zdvojené posloupnosti mají stejnou pravděpodobnost, pak lze říci, že počty porovnání a přesunů budou poloviční, než je tomu u nejhoršího případu.

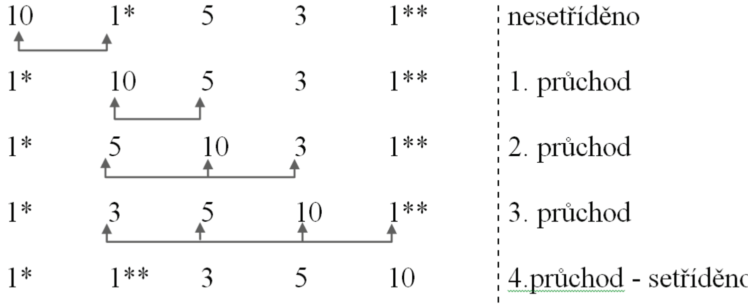
$$C_{AC} = \frac{n^2 + n - 2}{4} \tag{5.5}$$

$$M_{AC} = \frac{n^2 + 9n - 10}{4} \tag{5.6}$$

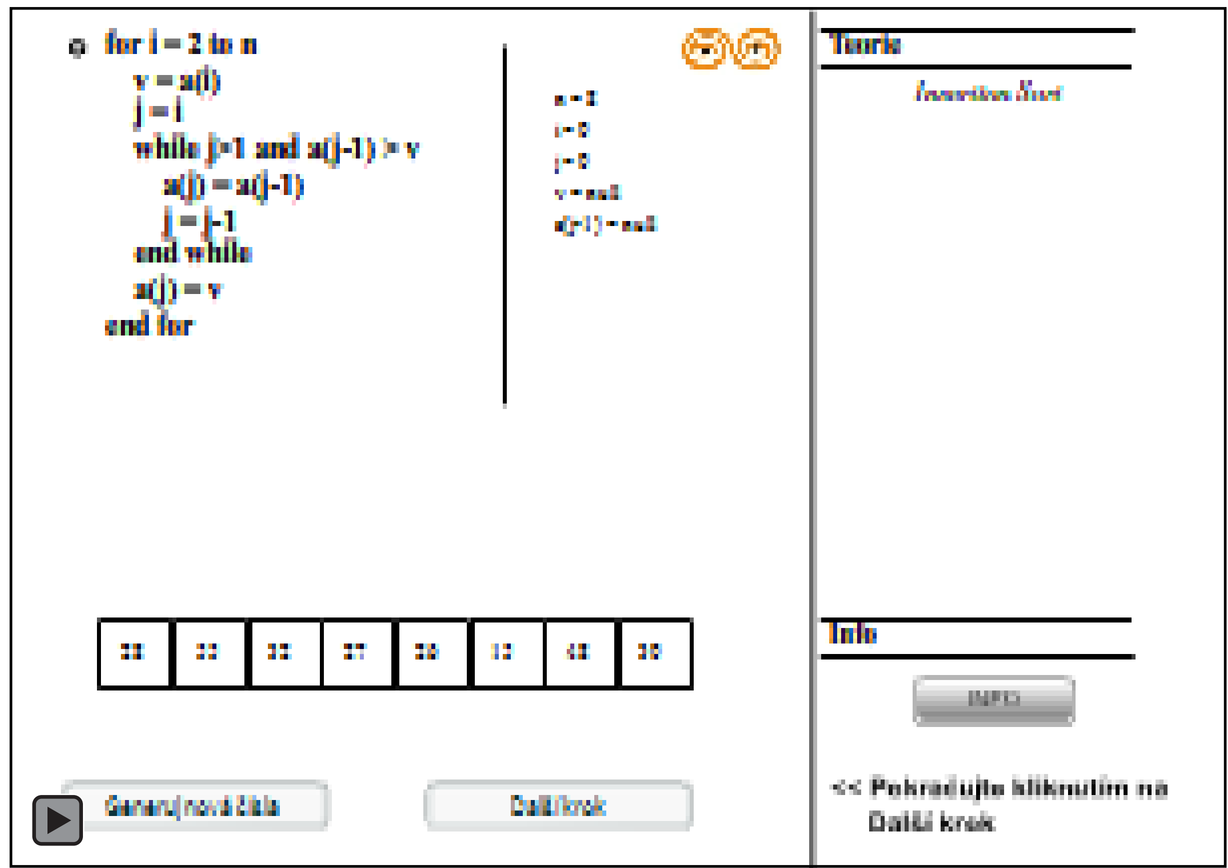
Průměrný počet přesunů a porovnání je tedy O(n^2). Zaměříme-li se na posouzení stability algoritmu, pak prvky se stejnými klíči zůstávají ve stejném vzájemném pořadí, tedy algoritmus je stabilní. Algoritmus je vhodný pro třídění posloupnosti náhodných hodnot klíčů.

Příklad:

Uvažujme následující nesetříděnou posloupnost čísel: 10 1* 5 3 1**.[1] Její setřídění proběhne ve čtyřech průchodech:



Obrázek 5.3 Příklad přímého vkládání



[1] Prvky pole se stejnými klíči mají své hodnoty rozlišeny hvězdičkami tak, aby bylo zřetelné, jestli je algoritmus stabilní.

5.2 ALGORITMUS PŘÍMÉHO VÝBĚRU (SELECTION SORT)

Základní myšlenka: V posloupnosti prvků nalezneme nejmenší prvek a ten vyměníme s prvkem na první pozici v seznamu. Potom vybereme druhý nejmenší prvek a ten vložíme na 2. pozici. Tento postup aplikujeme na všechny prvky seznamu.

Algoritmus pro třídění přímým výběrem může vypadat následovně a předpokládá, že prvky seznamu jsou uloženy v poli ***a* [1 .. *n*]**. Třídění se provádí pro ***n* - 1** prvků:

```
for i = 1 to n-1
  min = i
  for j = i+1 to n
        if a[j] < a[min]
              min=j
        end if
  end for
  x=a[min]
  a[min]=a[i]
  a[i]=x
end for
```

Analýza algoritmu přímého výběru

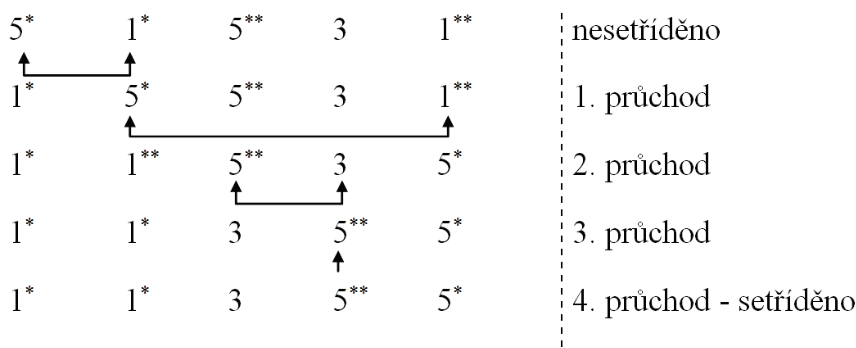
Uvedený algoritmus nemá nejhorší a nejlepší případ a tak počet porovnání a přesunů není závislý na počátečním způsobu uspořádání posloupnosti prvků. Počet porovnání je dosti vysoký, neboť vždy je provedeno porovnávání od pozice ***i* + 1** až do konce (***n***). Počet porovnání je tedy totožný s nejhorším případem u algoritmu přímého vkládání.

$$C = \frac{n}{2}(n-1) \tag{5.7}$$

Co se týče počtu přesunů, pak jejich počet je velice nízký a je roven $\boldsymbol{M = 3\ (n\ -\ 1)}$, tedy 3 přesuny na každý prvek. Metoda je vhodná tam, kde porovnání je časově málo náročná operace, zatímco přesun operace časově velmi náročná. Počet přesunů je roven $\boldsymbol{0(n)}$ a počet porovnání $\boldsymbol{0(n^2)}$. Algoritmus přímým výběrem nezachovává pořadí prvků se stejnými klíči, proto je nestabilní. Variantou algoritmu, kdy výměnu prvků nahradíme odsunutím všech prvků počínaje pozicí při vložení nalezeného nejmenšího prvku dozadu, bude algoritmus stabilní. Stabilita však bude vyvážena zvýšením počtu přesunů.

Příklad:

uvažujeme posloupnost čísel: 5^{*} 1^{*} 5^{**} 3 1^{**}. Její setřídění proběhne ve čtyřech průchodech:



Obrázek 5.4 Příklad algoritmu přímého výběru

5.3 ALGORITMUS TŘÍDĚNÍ PŘÍMOU VÝMĚNOU (BUBLINKOVÉHO TŘÍDĚNÍ) A TŘÍDĚNÍ PŘETŘÁSÁNÍM (BUBBLE SORT, SHAKE SORT)

Základní myšlenka: postupně porovnáváme sousední prvky posloupnosti zezadu dopředu. V případě, že prvky nejsou ve stejném pořadí, jsou vyměněny. Tak „vybublá" prvek s nejnižší hodnotou klíče na levý konec seznamu, který se při každém průchodu posouvá směrem doprava.

Následující algoritmus není příliš dobře optimalizován na počet průchodů, avšak dobře odráží princip metody třídění přímou výměnou (bublinkového třídění). Opět předpokládáme, že prvky jsou uloženy v poli ***a*****[1 ..** ***n*****]**.

```
for i = 2 to n
  for j = n to i step -1
        if a[j-1] > a[j]
              x = a[j-1]
              a[j-1] = a[j]
              a[j] = x
        end if
  end for
end for
```

Algoritmus třídění přímou výměnou není příliš efektivní a existuje několik způsobů, jak jej vylepšit. Prvním vylepšením může být eliminace průchodů v případě setříděného seznamu. Jak je zřejmé z algoritmu, vnější smyčka ***for i* = *2 to n*** probíhá **(*n -1*)**-krát nezávisle na tom, zda je setříděn či nikoliv. Setříděnost se dá identifikovat podle počtu výměn prvků uvnitř testu ***if***. Neproběhne-li žádná výměna, znamená to, že seznam je setříděn a je tedy možné vnější cyklus ukončit.

Dalším vylepšením algoritmu může být snaha o zavedení symetrie při zařazování prvků s velkými a malými klíči. Při krokování algoritmu třídění přímou výměnou se dá dokázat, že prvky s nižšími hodnotami klíčů jsou zařazovány rychleji, zatímco prvky s vyššími hodnotami se do svých pozic dostávají velice pomalu. Kdybychom třídění prováděli v opačném pořadí, tedy zařazovali nejdříve prvky s vyššími hodnotami klíčů, pak by byla nesymetrie opačná. Proto se lze domnívat, že pokud budeme při třídění měnit směry průchodu, dojde ke zrychlení algoritmu.

Poslední změnou může být uchování hodnoty indexu prvku, jehož se týkala poslední výměna. Prvků ležících za tímto prvkem se již určitě nebudou týkat žádné další změny, protože tato posloupnost je již uspořádána a tak tuto část seznamu již není nutné příště procházet. Provedeme-li všechny uvedené modifikace, získáme algoritmus nazývaný ***třídění přetřásáním*** (***Shake sort***):

```
el = 2; r = n; k = n
repeat
  for j = r to 1 step -1
        if a[j-1] > a[j]
              swap(a[j], a[j-1])[2]
              k = j
        end if
  end for
  el = k+1
  for j = el to r
        if a[j-1] > a[j]
```

```
                swap(a[j], a[j-1])2
                k = j
          end if
    end for r = k-1
    r = k-1
  until el>r
```

Analýza algoritmů:

U třídění přímou výměnou je počet porovnání nezávislý na počátečním uspořádání prvků. Neexistuje tedy nejhorší a nejlepší případ. Vždy je tedy proveden stejný počet testů, který je roven

$$C = \sum_{i=1}^{n-1}(n-1) = \frac{1}{2}(n^2 - n) \tag{5.8}$$

Při posuzování počtu přesunů zjistíme, že algoritmus je citlivý na počáteční uspořádání. Nejlepší případ tedy nastane, pokud je posloupnost již setříděna a pak nedojde ani k jedné výměně. Tedy

$$M_{BC} = 0 \tag{5.9}$$

Nejhorší případ nastane na opačně setříděné posloupnosti prvků, kdy počet přesunů bude roven

$$M_{WC} = \frac{3}{2}(n^2 - n) \tag{5.10}$$

(k přehození dvou sousedních prvků potřebujeme 3 přesuny; proto trojka v čitateli zlomku). Průměrný počet přesunů za známých předpokladů o pravděpodobnosti jednotlivých permutací bude poloviční.

U třídění přetřásáním je zřejmě, že dojde k redukci počtu průchodů testem, zatímco počet přesunů bude určitě stejný jako u metody třídění přímou výměnou. Nejlepší případ, případ setříděného seznamu, bude mít počet testů roven

$$C_{BC} = n - 1 \tag{5.11}$$

To proto, že první cyklus ***for*** proběhne celý, tedy pro ***i = n .. 1***, druhý cyklus již neproběhne a cyklus ***repeat*** skončí, protože ***el = n+ 1*** a ***r = n - 1***. Oba algoritmy, jak Bubble sort, tak Shake sort jsou stabilní.

Příklad:

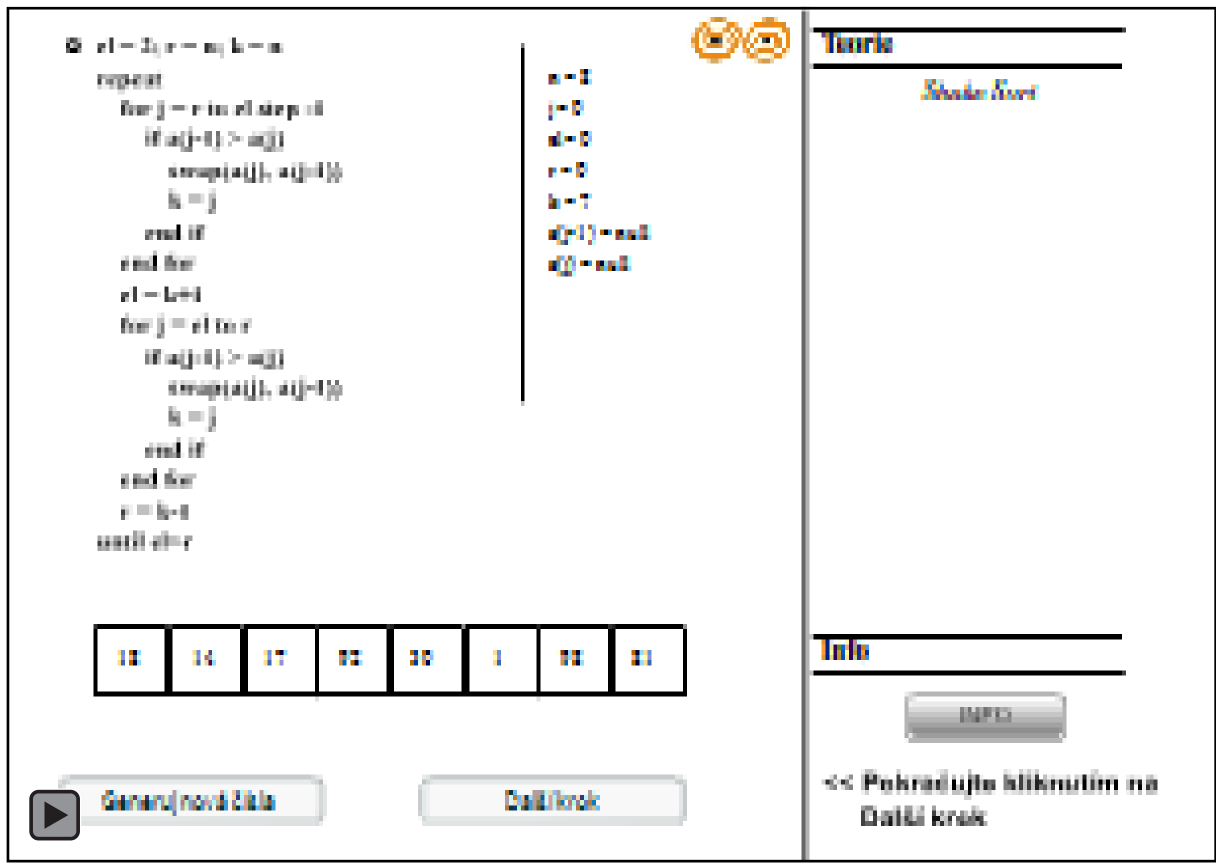
---

[2] Funkce ***swap*** vymění obsahy proměnných předaných jako parametry (hodnoty je nutné předat odkazem, nebo pomocí ukazatelů).

```
swap( a, b )
  pom = b
  b = a
  a = pom
end swap
```

12	30	2	28	44	1	nesetříděno, 1. průchod el=2, r=6, k=6
1	12	30	2	28	44	k=1, el=2
1	12	2	28	30	44	2. průchod k=5, r=5
1	2	12	28	30	44	setříděno k=2, el=3

Obrázek 5.5 Příklad algoritmu přetřásáním



5.4 TŘÍDĚNÍ ZMENŠOVÁNÍM KROKU (SHELL SORT)

Základní myšlenka: Shellův třídicí algoritmus spočívá ve vytváření tzv. setřídění posloupnosti s krokem ***h***. Vezmeme tedy každý ***h***-tý prvek v seznamu a provedeme na takto vzniklé množině prvků třídění metodou přímého vkládání. Pak vezmeme každý ***h* + 1** prvek a provedeme další setřídění atd. Tím získáme setříděnou posloupnost s krokem ***h***. Uvedeným způsobem se nám podaří vytvořit shluky prvků s blízkými klíči a tím zmenšit počet přesunů prvků na velké vzdálenosti, ***h*** – setříděná posloupnost se pak znovu přetřídí s menším krokem ***h***. Pro ***h* = 1** dojde tedy k vyvolání standardního algoritmu třídění přímým vkládáním. Obecně lze tedy říci, že Shellův algoritmus je mnohem úspornější, než algoritmus přímého vkládání. Toto tvrzení je doloženo tabulkou:

Uspořádání	Typ algoritmu	
	Přímé vkládání	Shellův algoritmus
opačné pořadí 25 prvků	324	129
náhodné pořadí 25 prvků	204	104

V následujícím algoritmu předpokládáme, že jednotlivé hodnoty kroku ***h*** jsou uloženy v poli ***h*[1 .. *t*]**. Diskuse o způsobech výpočtu hodnot kroku ***h*** je uvedena v odstavci věnovanému rozboru algoritmu. Dále předpokládáme, že tříděné prvky jsou uloženy opět v poli ***a*[1 .. *n*]**.

```
for m = 1 to t
  k = h[m]
  for i = k+1 to n
        x = a[i]
        j = i - k
        while (x < a[j]) and (j >= k)
              a[j + k] = a[j]
              j = j - k
        end while
        a[j + k] = x
  end for
end for
```

Analýza algoritmu:

Analýza Shellova třídění je dosti závislá na určené posloupnosti kroků h_i. Existuje několik způsobů, jakými se tato posloupnost určuje. Nebyla však prozatím vypracována žádná univerzální metoda optimalizující tuto posloupnost. Dobrých výsledků lze dosáhnout pro následující sekvenci kroků h_i:

$$h_i = 1 \quad \text{pro } i = 1$$
$$h_i = 3 * h_{i-1} + 1 \quad \text{pro } i > 2 \tag{5.12}$$
tedy posloupnost čísel: $1, 4, 13, 40, 121, \ldots$

Jiná doporučená posloupnost kroků má následující tvar:

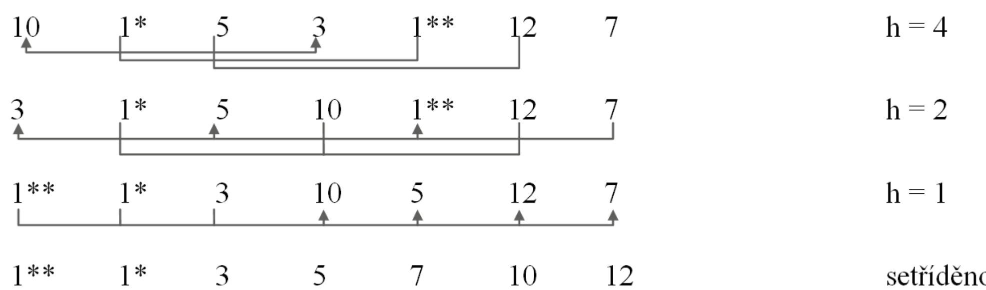
$$h_i = 1 \quad \text{pro } i = 1$$
$$h_i = 2 * h_{i-1} + 1 \quad \text{pro } i > 2 \tag{5.13}$$
tedy posloupnost čísel: $1, 3, 7, 15, 31, \ldots$

Odvození složitosti algoritmu je velice složité a proto uvádíme výsledky některých známých autorů:

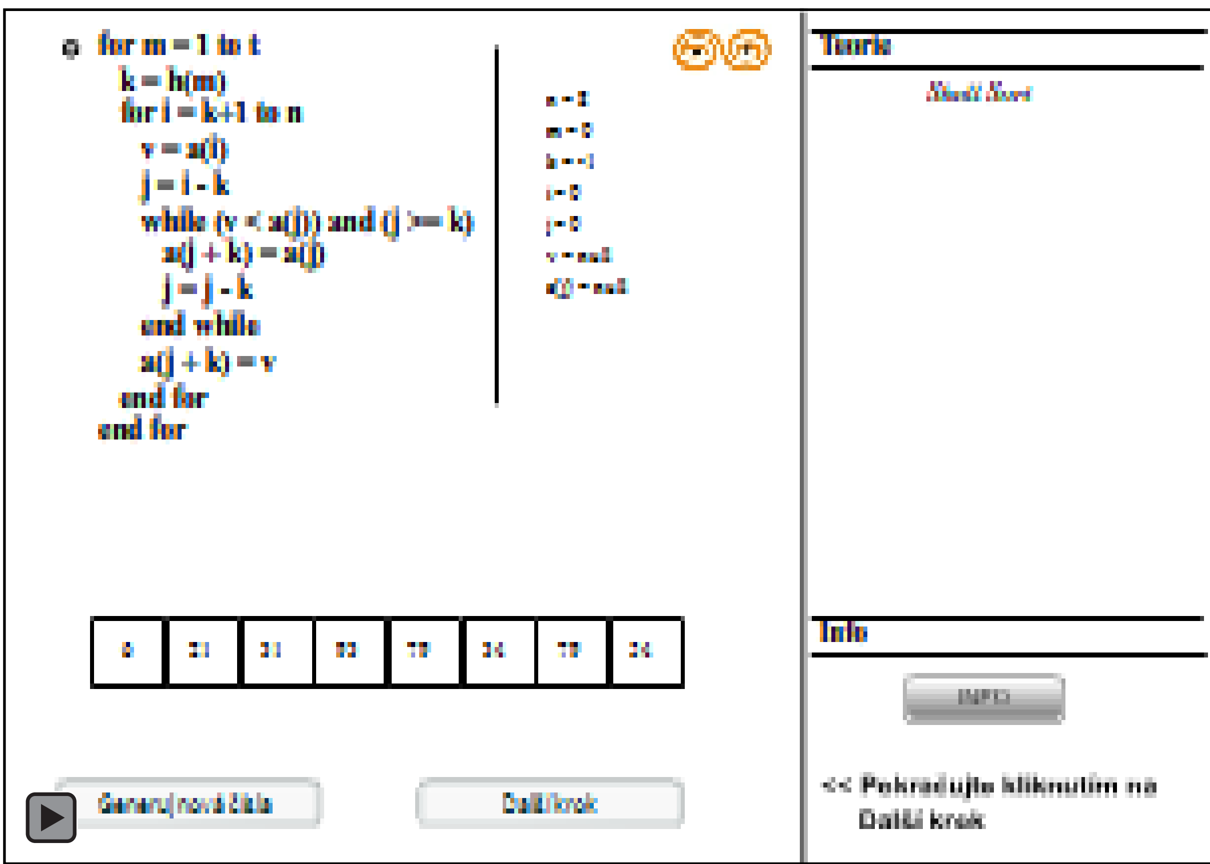
$O(N^{1.20})$

$O(N^{1.25})$

Příklad:



Obrázek 5.6 Příklad Shellova algoritmu třídění



V uvedeném příkladu byla zvolena sekvence kroků 4, 2, 1, která není moc výhodná, což se dá dokázat, ale pro ujasnění principu algoritmu je dostatečná. Z příkladu na obr. 5.6 je také zřetelné, že algoritmus není stabilní.

5.5 TŘÍDĚNÍ DISTRIBUČNÍM ČÍTÁNÍM (DISTRIBUTING COUTING)

Základní myšlenka: Pomocí tohoto třídícího algoritmu lze třídit prvky souboru, jejichž klíče jsou v rozsahu **[*0* .. (*M* - 1)]**, kde ***M*** je relativně malé číslo, a tak v souboru existuje mnoho prvků se stejnými klíči. Funkci algoritmu lze rozdělit do dvou kroků:

vypočteme frekvence výskytu jednotlivých klíčů a určíme diskrétní distribuční funkci na základě vypočtení distribuční funkce setřídíme prvky seznamu.

Při tvorbě programu předpokládáme, že prvky jsou umístěny v poli ***a*** **[1 ..** ***n*****]** a maximální hodnota klíče je ***M***.

```
REM - vytvoreni pocitadla vyskytu jednotlivych klicu
for j = 0 to M - 1
  count [j] = 0
end for
REM - vypocet frekvencni funkce
for j = 1 to N
  count [a [j]] = count [a [j]] + 1
end for
REM - vypocet distribucni funkce
for j = 1 to M -1
  count [j] = count [j - 1] + count [j]
end for
REM - vlastní setrideni, pole t je pomocne pole
```

```
for j = N to 1 step -1
  t [ count [a [j]]] = a[j]
  count [a [j]] = count [a [j]] -1
end for
```

Analýza algoritmu:

Složitost algoritmu je v tomto případě dána jak počtem prvků v seznamu, tak i maximální hodnotou klíče. Jak je viditelné ze zápisu algoritmu, neexistují zde žádná porovnání. Program je složen pouze ze čtyř cyklů ***for*** a několika přiřazení. Počet přiřazení do pole diskrétní distribuční funkce je:

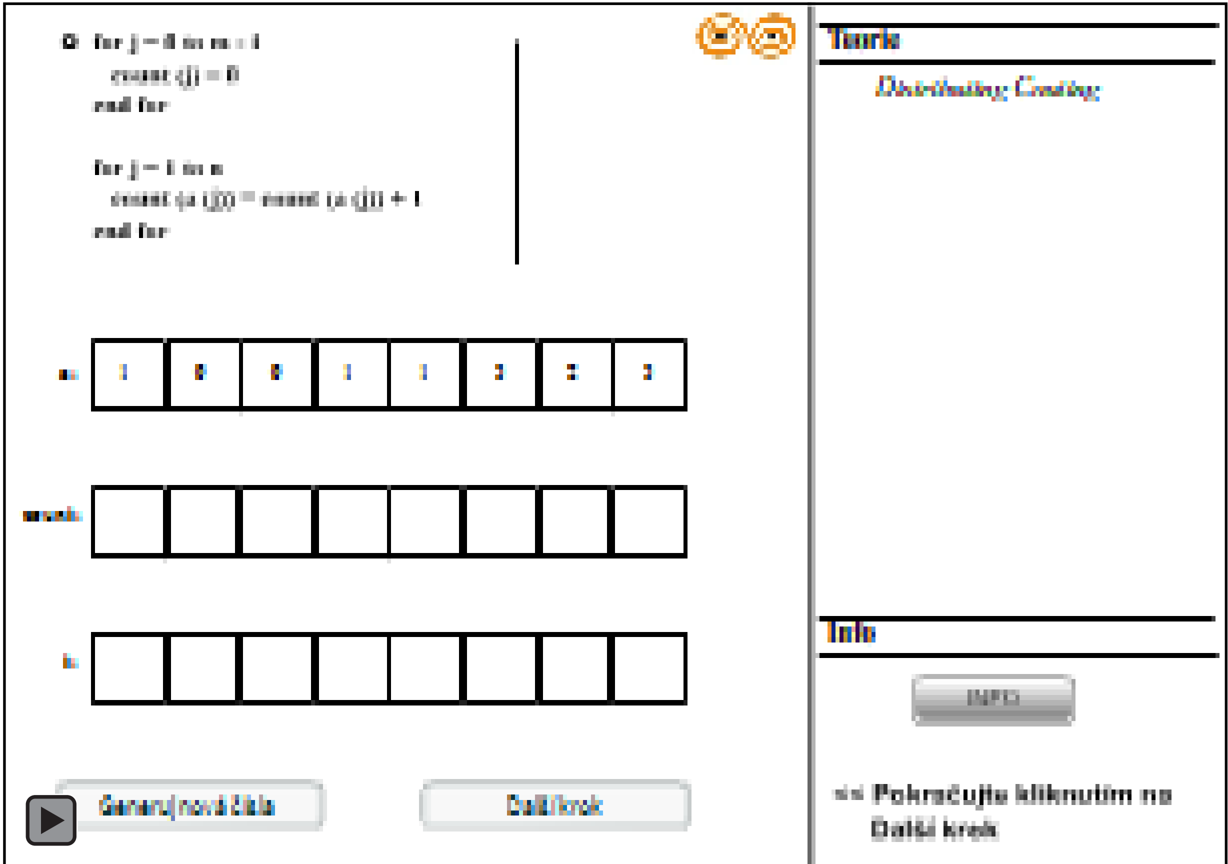
$$C_{DF} = M * N * M * N \tag{5.14}$$

a pro $M << N$

$$C_{DF} = N^2 \tag{5.15}$$

Počet přiřazení do pole prvků je: $C_E = N$.

Algoritmus distribučního čítání je stabilní.



Příklad:

0	1	2	3	4	5	6	7	8	9	
10	1^{*}	5	3	1^{**}	12	7^{*}	7^{**}	1^{***}	0	pole a

0	1	2	3	4	5	6	7	8	9	10	11	12	
1	3	0	1	0	1	0	2	0	0	1	0	1	pole count - 2. for cyklus

0	1	2	3	4	5	6	7	8	9	10	11	12	
1	4	4	5	5	6	6	8	8	8	9	9	10	pole count - 3. for cyklus

0	1	2	3	4	5	6	7	8	9	10	
-	0	1	1	1	3	5	7	7	10	12	pole t - setříděno

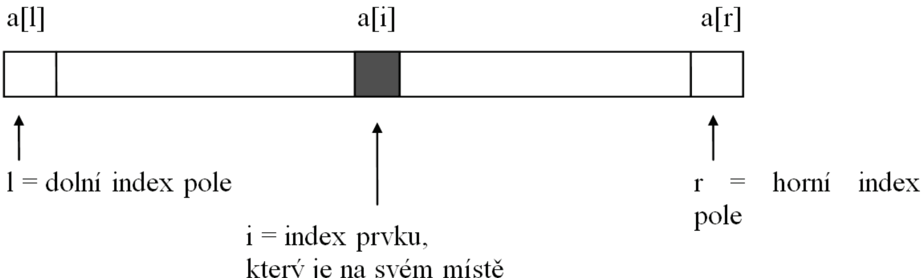
Obrázek 5.7 Příklad třídění metodou distribučního čítání

5.6 ALGORITMUS TŘÍDĚNÍ ROZDĚLOVÁNÍM (QUICK SORT)

Základní myšlenka: Algoritmus třídění rozdělováním (Quick sort) je algoritmus využívající metodu ***Rozděluj a panuj*** (***devide and conquer***). Metody typu Rozděluj a panuj postupují shora dolů, tedy složitější operaci rozdělíme na skupinu jednodušších, které mohou být dále rozděleny na ještě jednodušší operace.

Předpokládejme tedy, že potřebujeme v našem případě setřídit seznam ***N*** prvků. Tento seznam tedy rozdělíme na ***k*** disjunktních podseznamů, které setřídíme odděleně. Po rozdělení a setřídění můžeme podseznamy opět složit zpět do původního seznamu, jehož setřídění bude mnohem rychlejší. Jak je vidět z popisu metody rozděluj a panuj pro podseznamy, řešíme stejný problém jako pro seznam a tedy metoda vede na rekurzivní algoritmus. Jak uvidíme, tato přirozená rekurze se dá odstranit, jelikož pro velká ***N*** bychom pravděpodobně měli problémy s praktickou realizací.

Rekurzivní algoritmus je implementován funkcí ***QuickSort*** se dvěmi parametry, udávající dolní a horní index tříděného pole. Tato funkce dále volá funkci ***partition***, která dané pole rozdělí podle následujícího principu:



Obrázek 5.8 Princip metody rozdělení pole

a platí:

$a [k] \leq a [i]$ pro $k < i$

$a [k] \geq a [i]$ pro $k > i$

```
QuickSort (el, r)
  if (r > el)
        i = partition(el, r)
```

```
         QuickSort(el, i-1)
         QuickSort(i+1, r)
      end if
    end QuickSort
    partition (el, r)
      v = a[r]
      i = el-1
      j = r
      repeat
         repeat
            i = i+1
         until a[i] >= v
         repeat
            j = j-1
         until (a[j] <= v) OR (j = el)
         swap(a[i], a[j])
      until j <= i
      t = a[j]
      a[j] = a[i]
      a[i] = a[r]
      a[r] = t
    end partition   opravit algoritmus
```

Jak je viditelné ze zápisu algoritmu, je funkce ***QuickSort*** volána uvnitř svého těla dvakrát. Tedy dochází zde ke dvěma rekurzím. První rekurze je velice jednoduše odstranitelná tak, že operaci ihned provedeme. K odstranění druhé, přímo následující rekurze, se však musí použít jiná metoda, kterou může být na příklad vlastní druh zásobníku, do kterého budeme ukládat meze pro volání funkce ***QuickSort***. Tedy na začátku programu do zásobníku vložíme dvojici hodnot **(1, *n*)** a metoda ***QuickSort*** si tyto meze vybere a rozdělí. Jednu dvojici zpracujeme ihned a druhou uložíme opět na zásobník. Funkce ***QuickSort*** pak musí běžet tak dlouho, dokud jsou nějaké dvojice indexů pole na zásobníku.

```
QuickSortM
  vytvor_zasobnik
  vloz_na_zasobnik(1, n)
  repeat
        vyjmi_ze_zasobniku(el, r)
        repeat
              i = el;  j = r
              x = a[(el+r) div 2]
              repeat
                    while a[i] < x
                          i = i+1
                    end while
                    while a[j] > x
                          j = j-1
                    end while
                    if i <= j
                          swap(a[i], a[j])
                          i = i + 1
                          j = j - 1
                    end if
              until i > j
```

```
                if i< r
                      vloz_na_zasobnik(i, r)
                end if
                r = j
          until el >= r
    until zasobnik_je_prazdny
  end QuickSortM
```

Analýza algoritmu:

Odvození složitosti algoritmu třídění rozdělováním je trochu zdlouhavější, nikoli však nemožné. Nejdříve vyjádříme střední případ, kdy prvky umístěné v posloupnosti jsou náhodné a všechny permutace klíčů mají stejnou pravděpodobnost výskytu. Dále uvažujeme obecný případ, kdy jako dělící klíč seznamu jsme zvolili prvek s indexem ***i***. Tak lze počet výměn vyjádřit jako součin prvků v levé části pole a pravděpodobnosti výskytu klíče, který je větší než klíč s indexem ***i***.

Počet klíčů vlevo od prvku ***i = i -1***

Pravděpodobnost výskytu prvků s větším klíčem než má prvek ***i***:

$$p(i) = \frac{n-i+1}{n} \tag{5.16}$$

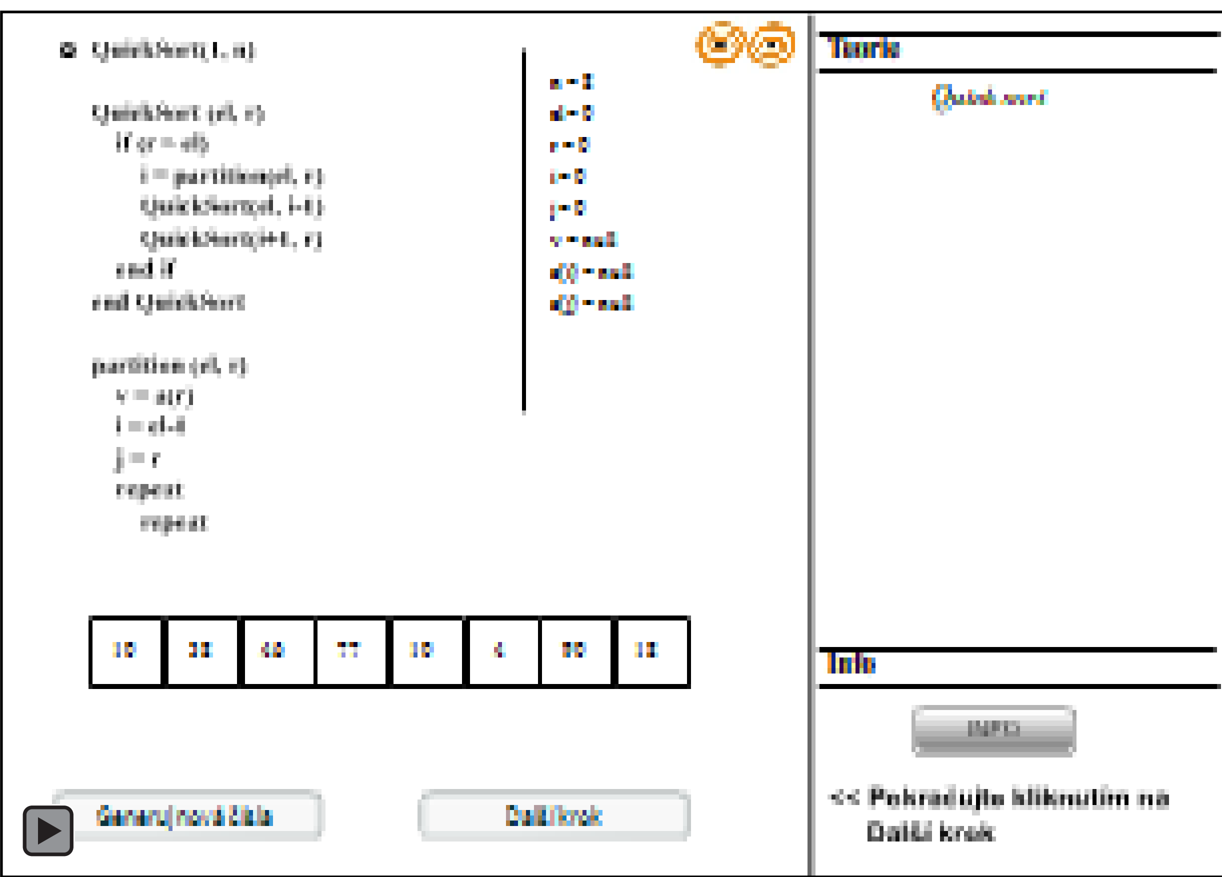
Střední počet výměn při dělení pole na úseky je:

$$M = \frac{1}{n}\sum_{i=1}^{n}(i-1)\frac{n-i-1}{n} = \frac{n}{6} - \frac{1}{6n} \approx \frac{n}{6} \tag{5.17}$$

pokud pole rozdělíme vždy na polovinu, pak budeme potřebovat $(\log_2 n)$ průchodů pro setřídění pole. Pak tedy

$$C_{BC} = n\log_2 n \tag{5.18}$$

$$M_{BC} = \left(\frac{n}{6}\right)\log_2 n \tag{5.19}$$



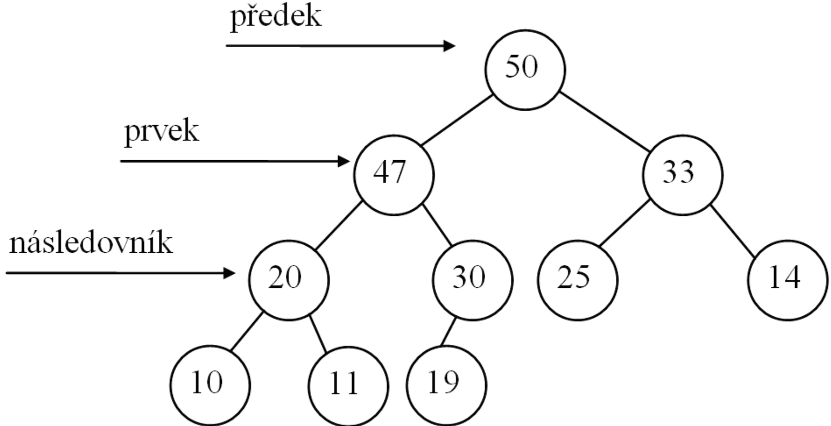
nejhorší případ nastane, pokud jako dělící prvek zvolíme nejmenší prvek v úseku. V tomto případě budeme potřebovat n^2 přesunů. Uvedený případ nastane prakticky, když pole je již setříděno a jako dělící prvek je zvolen jeden z krajních prvků. Prakticky se proto dělící prvek volí náhodně nebo jako medián úseku (viz program: ***x* = *a*[(l + *r*) *div* 2]**).

V případě, že budeme používat nerekurzivní verzi algoritmu, bude zajímavé odhadnout potřebnou velikost zásobníku, na který budeme ukládat dvojice indexů. Hloubku zásobníku musíme vždy dimenzovat pro nejhorší případ, tedy případ, kdy pravý úsek, jenž se vyhodnocuje nejdříve, má pouze jeden prvek a levý pak délku ***n* - 1**. V tomto extrémním případě se nám na zásobníku ocitne až ***n*** dvojic indexů.

5.7 TŘÍDĚNÍ HROMADOU (HEAPSORT)

Nejdříve si musíme definovat pojem ***hromada*** (***heap***). Hromada je úplný binární strom, který splňuje podmínku hromady. Podmínka hromady zní: klíč v každém uzlu má hodnotu menší nebo rovnu klíčové hodnotě jeho předka.

Příklad:



Obrázek 5.9 Hromada

Strom uvedený na obr. 5.9 je hromadou. V paměti počítače však bude reprezentován pomocí pole, což je sice nezvyklé pro binární stromy, ale v případě třídění hromadou efektivní co do rychlosti i nároků na paměť. Uzly hromady budou uspořádány postupně po jednotlivých úrovních zleva doprava, počínaje kořenem:

1	2	3	4	5	6	7	9	8	10
50	47	33	20	30	25	14	10	11	19

Obrázek 5.10 Reprezentace hromady pomocí pole

Z uvedeného uspořádání v poli se dají určit následující vlastnosti:

6. rodičovský uzel je vždy na pozici **(*j div* 2)**
7. uzel potomka je vždy na pozici **2*j*** [levý] a **(2*j* + 1)** [pravý]

Základní myšlenka: Vybudujeme hromadu z daných prvků, které mají být setříděny. Odebereme prvek umístěný v kořeni, o kterém můžeme s jistotou prohlásit, že je největší a na jeho místo umístíme prvek nacházející se v nejnižší úrovni stromu vpravo. Dále pak snížíme počet uzlů ve stromu o jeden. Uvedenou operací jsme však porušili podmínku hromady a hromadu je nutné přebudovat. Po opětovném vystavění hromady opět odebereme prvek umístěný v kořeni a celou úlohu opakujeme tak dlouho, dokud jsou ve stromu nějaké prvky. Používaná označení v algoritmu:

N - počet prvků umístěných na hromadě,

M - počet prvků, které se mají setřídit,

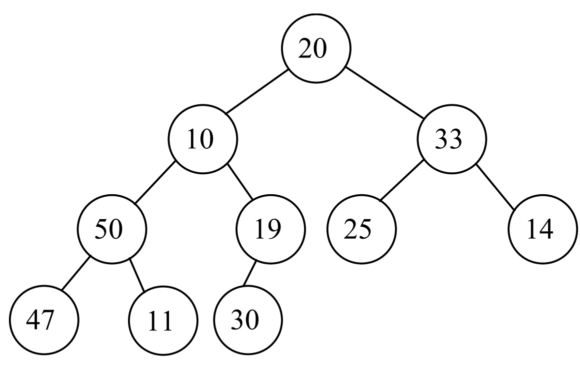
a - pole obsahující prvky hromady.

```
HeapSort
  N = M
  for  k = (M div 2) to 1 step -1
        downheap(k) REM - vytvoreni pocatecni hromady
  end for
  repeat
        swap(a[1], a[n])
```

```
            N = N - 1
            downheap(1) REM - znovu vybudovani hromady
      until N <= 1
    end HeapSort
    downheap (k)
      v = a[k]
      while k <= (N div 2)          REM - pokud uzel ma potomka
            j = 2 * k               REM - levy potomek
            if j < N                REM - ma leveho potomka ?
                  if a[j] < a[j+1]
                        j = j + 1
                  end if
            end if
            if v >= a[j]
                  goto konec_cyklu
            end if
            a[k] = a[j]
            k = j
      end while
    konec_cyklu:  a[k] = v
    end downheap
```

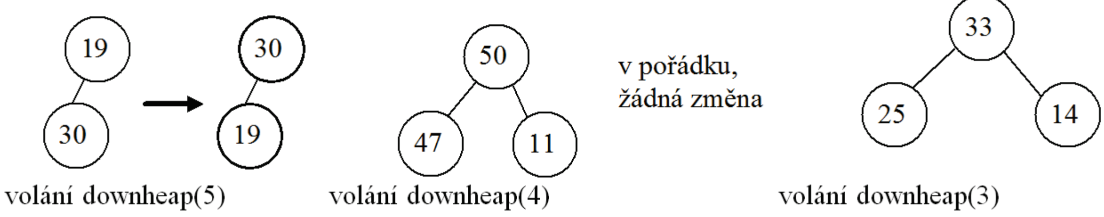
funkce ***HeapSort*** interně používá funkci nazvanou ***downheap***, která má jeden celočíselný argument. Funkce ***downheap*** umožňuje vybudovat hromadu od prvku s indexem daným argumentem (***j***) za předpokladu, že prvky na pozicích s indexy **[*j* .. *M*]** jsou již správně uspořádány. Postup vybudování hromady lze ilustrovat následujícím příkladem:

Příklad: předpokládejme, že máme setřídit posloupnost čísel: 20, 10, 33, 50, 19, 25, 14, 47, 11, 30.

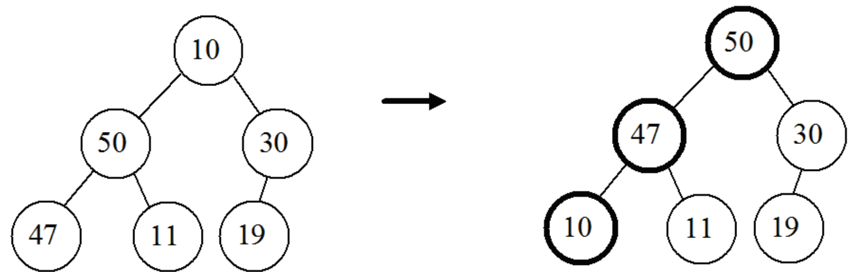


Obrázek 5.11 Počáteční stav stromu

Počáteční stav stromu je na obr. 5.11. Strom je naplněn jednotlivými prvky tak, jak jdou za sebou. Strom však nesplňuje podmínku hromady a tudíž je nutné postupně zavolat funkci ***downheap*** s argumenty **(5) (4) (3) (2) (1)**. Vždy stačí volat tuto funkci pro argumenty v rozsahu [**1 ..** ***n/2***], protože platí, že prvky s indexem [***n/2* ..** ***n***] jsou koncovými listy stromu.

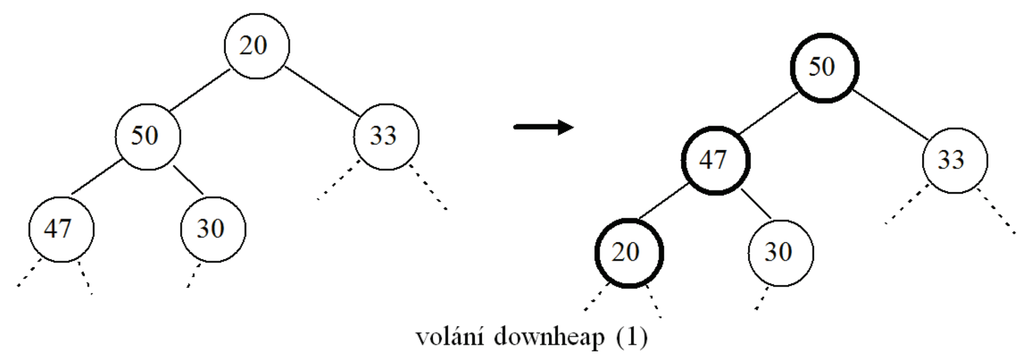


Obrázek 5.12 Vytvoření hromady a



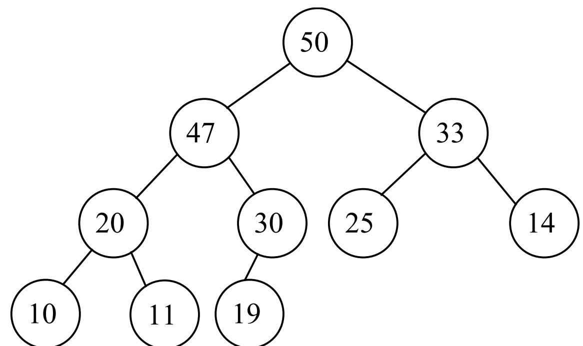
volání downheap(2)

Obrázek 5.13 Vytvoření hromady b



Obrázek 5.14 Vytvoření hromady c

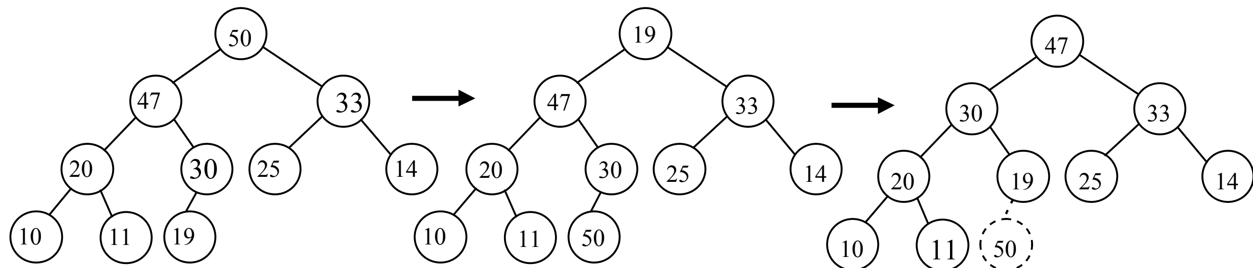
výsledný strom splňující podmínku hromady má tvar:



Obrázek 5.15 Výsledný strom splňující podmínku hromady

a v paměti má reprezentaci posloupnosti čísel: 50, 47, 33, 20, 30, 25, 14, 10, 11, 19.

Prvek s největší hodnotou klíče tedy leží v kořeni stromu. Provedeme jeho výměnu s posledním prvkem stromu, snížíme počet prvků a zavoláme funkci ***downheap*(1)** pro obnovení podmínky stromu:



Obrázek 5.16 Postupné odebírání největších prvků

uvedený postup opakujeme, dokud má strom prvky. Nakonec získáme vzestupně setříděnou posloupnost, která v paměti počítače, ve které je strom uložen v jednorozměrném poli, odpovídá posloupnosti: 10, 11, 14, 19, 20, 25, 30, 33, 47, 50.

Analýza algoritmu:

Pro výpočet složitosti algoritmu rozdělíme úlohu třídění na 2 části:

8. inicializaci a vytvoření hromady
9. vlastní třídění.

Vytvoření hromady se provede ***n/2***-násobným voláním funkce ***downheap***. Pro nejhorší případ bude vždy provedeno ($\log_2 n - \log_2 i$) přesunů pro i =1 .. n/2.

$$M_{1WC} = \sum_{1}^{n/2} (\log_2 n - \log_2 i) \approx \sum_{1}^{n/2} (\log_2 n) - \int_{1}^{n/2} \log_2 i di = \frac{n}{2} \log_2 n - \left[\frac{n}{2} \left(\log_2 \frac{n}{2} - s \right) + s \right] \quad (5.20)$$

kde $s = \log_2 e \approx 1{,}44269$.

Celkový počet přesunů při tvorbě hromady je tedy menší než $\frac{n}{2}\log_2 n$. Vlastní třídění je charakterizováno konstantním počtem cyklů rovným hodnotě **(n - 1)**. V každém průchodu je volána metoda ***downheap*(1)** vyžadující vždy **($\log_2 m$)** přesunů, kde m je počet prvků hromady. Tedy celkově bude třeba:

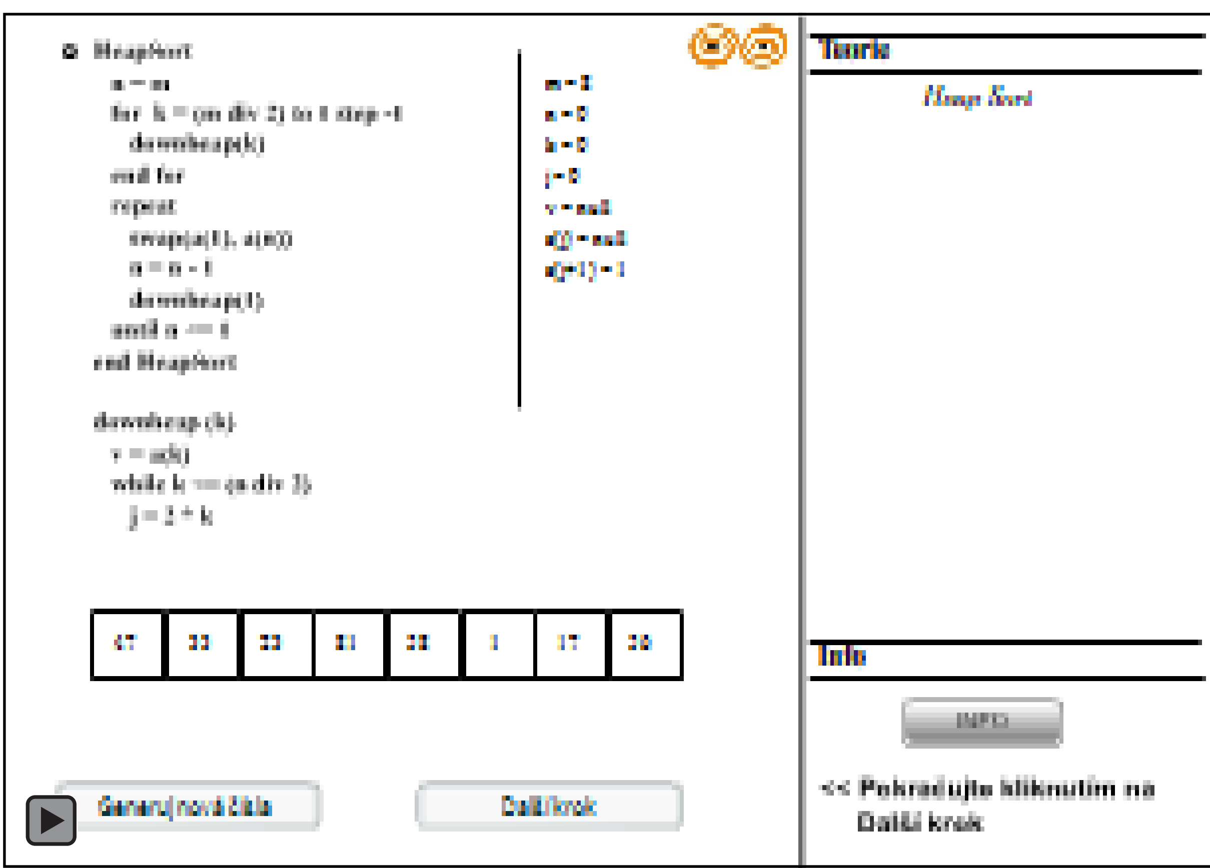
$$M_2 = \sum_{1}^{n-1} \log_2 m \approx \int_{1}^{n-1} \log_2 m dm = (n-1)(\log_2 (n-1) - s) + s \quad (5.21)$$

kde s = $\log_2$e = 1.44269...

v každém průchodu navíc potřebujeme vyměnit prvek kořene s posledním listem vpravo, tedy celkově další tři přesuny na cyklus:

$$M_2 = (n-1)(\log_2 (n-1) - s) + s + 3(n-1) \quad (5.22)$$

Počet přesunů je tedy **$O(n \,.\, \log_2 n)$.**



5.8 VYHLEDÁNÍ *N*-TÉHO NEJMENŠÍHO PRVKU, VYHLEDÁNÍ *N* NEJMENŠÍCH PRVKŮ

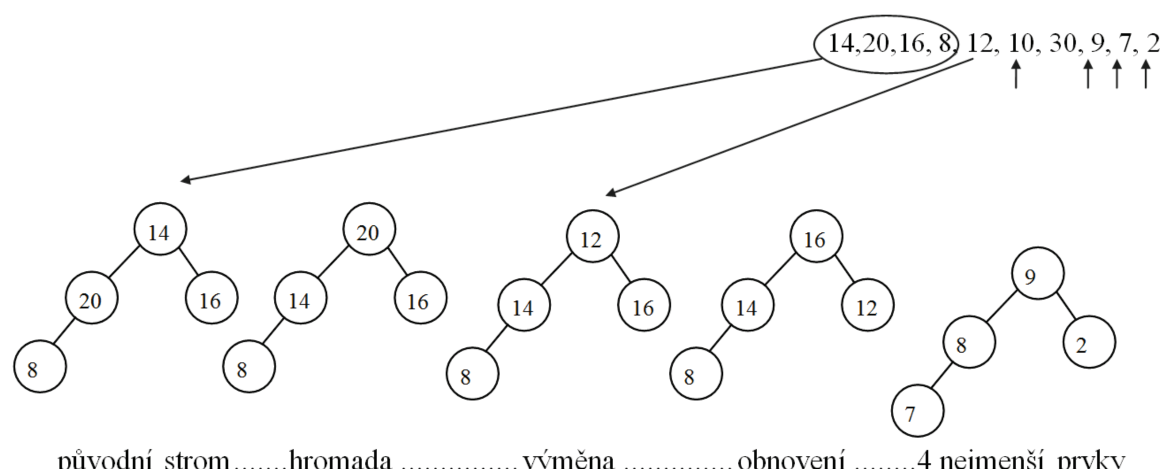
Jednou z velmi často požadovaných úloh je vyhledání ***n***-tého nejmenšího prvku nebo nalezení ***n*** nejmenších prvků v dané množině. Za tímto účelem je možné s výhodou použít algoritmu třídění hromadou popsaného v předchozí kapitole.

Základní myšlenka:

Předpokládejme, že máme ***n*** prvků, z nich hledáme ***k*** nejmenších. Vybudujeme tedy hromadu z prvních ***k*** prvků. Následně zaměníme prvek kořene stromu s dalším prvkem množiny, pokud má prvek kořene vyšší hodnotu klíče. Po záměně opět hromadu přebudujeme metodou ***downheap*****(1)**. Tak postupujeme pro všechny prvky množiny. Na konci pak hromada obsahuje k nejmenších prvků množiny.

Příklad:

uvažujeme, že chceme najít čtyři nejmenší prvky z posloupnosti čísel:



Obrázek 5.17 Vyhledání čtyř nejmenších prvků

Analýza algoritmu:

Odvození složitosti algoritmu vyplývá ze složitosti třídění hromadou. Uvažujeme-li pouze průchod vlastním algoritmem, pak počet přesunů je $O(n\ .\ \log_2 k)$. To proto, že musíme provést cyklus pro **(*n* - 4)** prvků a každé volání metody ***downheap*(1)** zabere $\log_2 k$ přesunů.

5.9 TŘÍDĚNÍ SLUČOVÁNÍM (MERGE SORT)

Metoda třídění slučováním (***Merge sort)*** je použitelná jak pro interní, tak pro externí třídění. Nejdříve si uvedeme její variaci pro interní třídění. Při interním třídění předpokládáme, že data jsou uložena v dynamickém seznamu jednostranně vázaném. Jelikož metoda třídění sekvenčně přistupuje k uloženým datům, je právě velmi vhodná pro toto uspořádání dat.

Základní myšlenka:

Metoda předpokládá, že dynamický seznam rozdělíme do dvou seznamů, které jistým způsobem setřídíme. Jde tedy o metodu typu rozděluj a panuj. Setříděné seznamy pak pomocí jiné metody sloučíme v jeden setříděný. Prakticky se dělení na podseznamy provádí tak dlouho, dokud nemáme ***n*** jednoprvkových podseznamů, které jsou již defacto setříděné. Následuje jejich slučování do seznamů o 2, 4, 8, atd. prvcích, dokud nevznikne výsledný setříděný seznam.

Podívejme se tedy z praktického hlediska na metodu slučování dvou setříděných dynamických seznamů. Pro sloučení využijeme metodu ***dvoucestného slučování*** **(2 - *way meeting*)**, jejíž princip je následující:

10. na začátek každého seznamu ukazuje ukazatel a každý seznam je ukončen prvkem se speciálním klíčem (zarážka), jeho hodnota je vyšší než u ostatních prvků,
11. porovnáme prvky obou seznamů, na které ukazují ukazatele ***a***, ***b*** a určíme menšího z nich. Na tento prvek začne ukazovat ukazatel ***c***,
12. ukazatel na menší z prvků se posune na jeho místo následovníka v seznamu,
13. prvek, na nějž původně ukazoval ukazatel ***c***, bude ukazovat na menší ze dvou srovnávaných (na ten, na nějž nyní ukazuje ***c***),
14. opakujeme od bodu 2) pro všechny prvky seznamu.

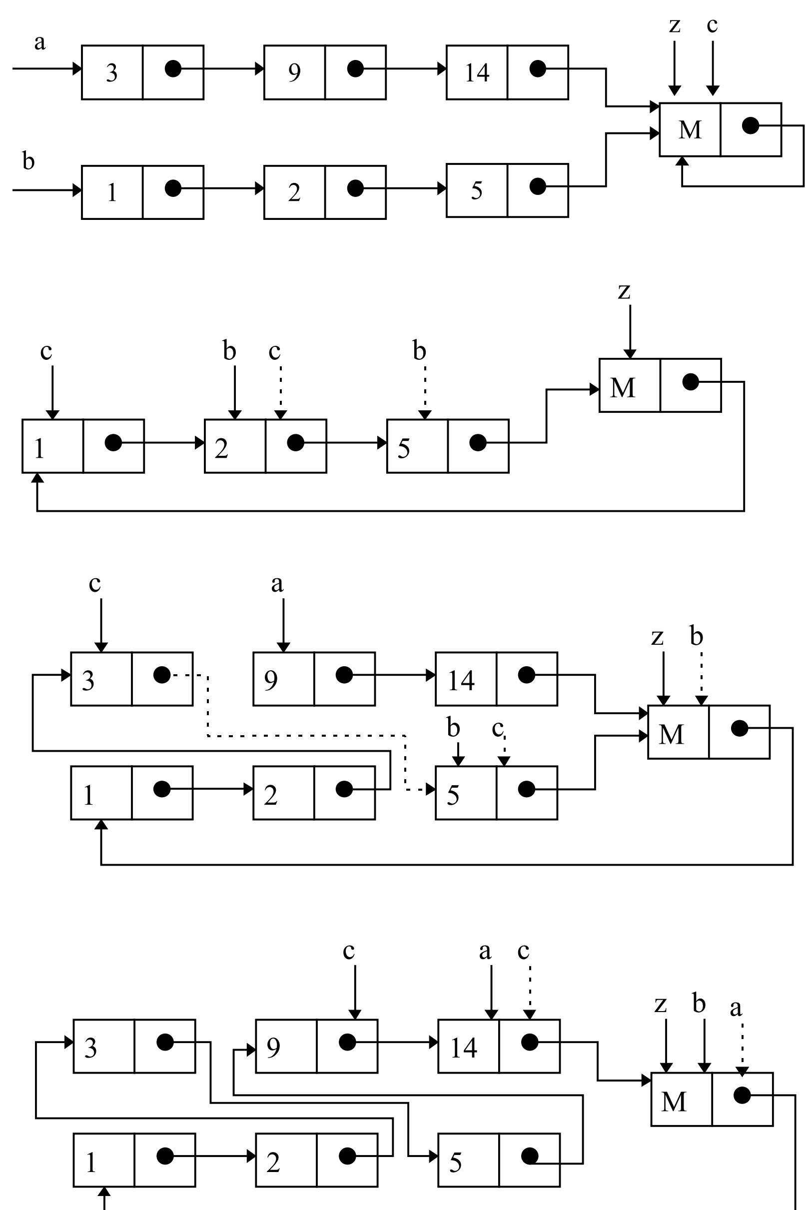
Funkce pro uvedený algoritmus dvoucestného slučování:

Použité proměnné: ***a***, ***b***, ***c***, ***head***, ***z*** - jsou ukazatelé na strukturu ***nodeType***

```
merge (a: nodePtr, b: nodePtr): nodePtr
  c = z
  repeat
        if(a↑.key <= b↑.key)
              c↑.next = a
              c = a
              a = a↑.next
        else
              c↑.next = b
              c = b
              b = b↑.next
        end if
  until (c↑.key = M)
  head = z↑.next
  z↑.next = z
  merge = head
end merge
```

Uvedený výpis programu předpokládá datovou strukturu pojmenovanou ***nodeType***, obsahující hodnotu klíče prvku a dále pak ukazatel na další prvek v seznamu. Pro zjednodušení zápisu se typ ukazatel na strukturu ***nodeType*** nazývá ***nodePtr***. Dále pak již z obrázku vyjadřující princip metody dvojcestného slučování plyne, že každý seznam je ukončen speciálním prvkem, jehož hodnota klíče má maximální hodnotu. Poslední prvek taktéž ukazuje vždy sám na sebe a na něj ukazuje vždy ukazatel nazvaný ***z***. Vybudování seznamu by tedy mohlo vypadat:

```
readlist(n: integer)
  new(z)
  z↑.key = M
  z↑.next = z
  p = z
  for i = 0 to (n-1)
        new(p↑.next)
        p = p↑.next
        nacti hodnotu
        p↑.key = hodnota
  end for
  p↑.next = z
  head = z↑.next
  z↑.next = z
  readlist = head
end readlist
```



Obrázek 5.18 Merge sort – nerekurzivní verze

Podívejme se nyní na vlastní algoritmus třídění. Ten je možné realizovat dvěma způsoby:

rekurzivní formou

nerekurzivní formou

Rekurzivní forma je jednodušší a pochopitelnější na první pohled, avšak při vysokém počtu tříděných prvků vznikají stejné problémy jako u všech rekurzivních metod – problém přetečení zásobníku. I přesto uveďme tuto variantu jako první:

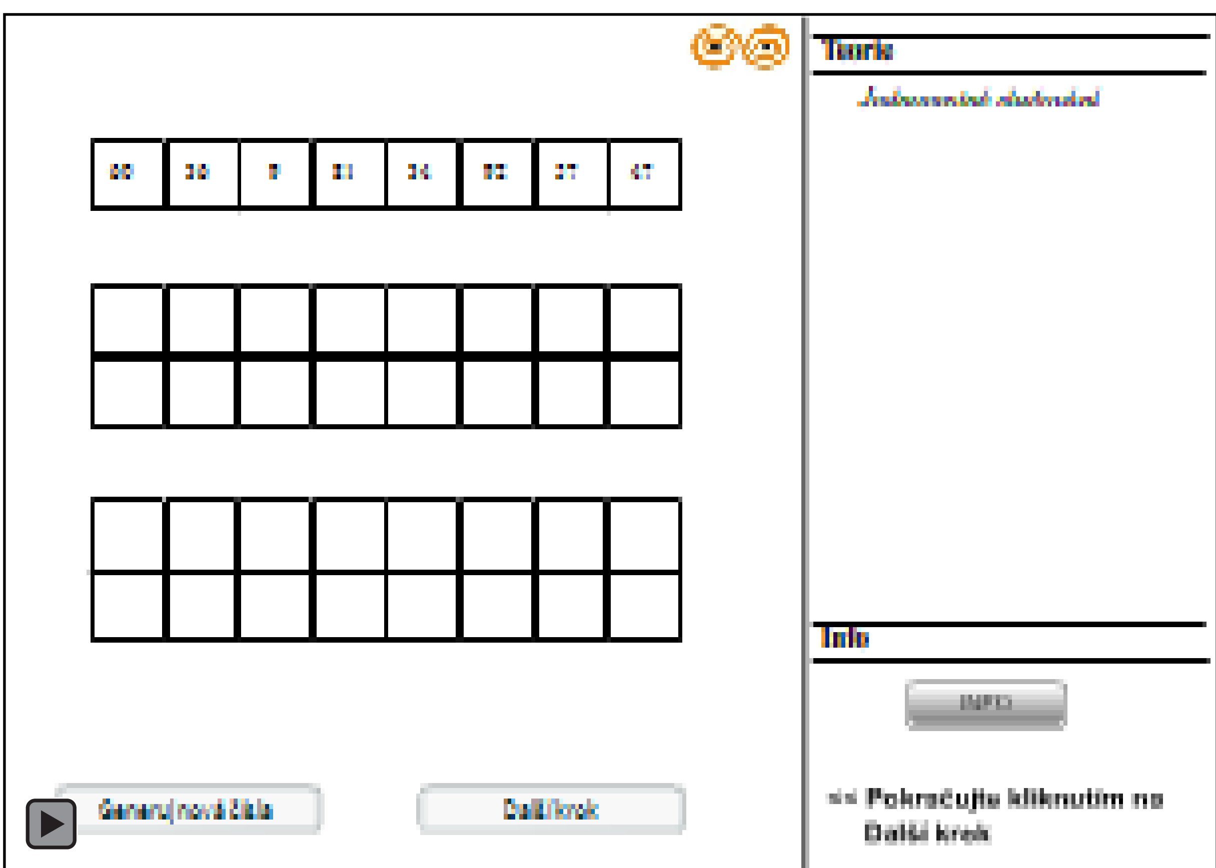
```
mergeSort(c: nodePtr): nodePtr
  if (c↑.next = z)
        mergeSort = c
  else
        a = c
        b = c↑.next
        b = c↑.next
```

```
            while (b <> z)
                  c = c↑.next
                  b = b↑.next
                  b = b↑.next
            end while
            b = c↑.next
            c↑.next = z
      mergeSort = merge(mergeSort(a), mergeSort(b))
      end if
    end mergeSort
```

Funkce předpokládá, že vstupním parametrem je ukazatel na začátek vázaného seznamu. Návratová hodnota je taktéž ukazatel, avšak na setříděný seznam. Uvnitř těla jsou použity tři ukazatele na prvek seznamu – ***a***, ***b***, ***c***. Ukazatel ***a*** ukazuje vždy na začátek první poloviny seznamu. Polovinu určíme tak, že postupně přesouváme ukazatele ***b*** a ***c*** po jednotlivých prvcích seznamu tak, že ukazatel ***b*** se pohybuje dvakrát rychleji, než ukazatel ***c***. Tato činnost se provádí tak dlouho, dokud ukazatel ***b*** nenarazí na konec seznamu. V této chvíli je ukazatel ***c*** právě v jeho polovině. První polovinu seznamu pak musíme zakončit prvkem s maximální hodnotou a následovně stačí zavolat metodu ***merge*** s parametry ***mergeSort (a)*** a ***mergeSort (b)***, čímž vytvoříme rekurzivní volání. Rekurze bude probíhat tak dlouho, dokud jednotlivé poloviny seznamů budou mít více než 1 prvek. Pak bude postupně docházet ke zpětnému slučování.

Analýza algoritmu.

Algoritmus třídění slučováním nemá nejhorší případ a tak jeho složitost není závislá na počátečním uspořádání prvků. Při fázi slučování budeme nejdříve slučovat ***N*** jednoprvkových seznamů do ***N/2*** dvouprvkových seznamů atd. Každé sloučení vyžaduje ***2i-1*** porovnání, kde ***i*** je počet prvků ve slučovaném seznamu. Počet porovnání potřebných pro sloučení je tedy $O(N \cdot \log_2 N)$. O počtu přesunů u tohoto algoritmu nelze hovořit, protože pro manipulaci s prvky se používají ukazatele a tak k vlastnímu přesunu dat vůbec nedochází. Uvedený algoritmus je stabilní.



5.9.1 Nerekurzivní verze algoritmu Merge sort

Nerekurzivní verze odstraňuje problémy s přetečením zásobníku. Změna spočívá pouze v modifikaci funkce ***mergeSort***. Funkce ***merge*** zůstává nezměněna. Podívejme se nejdříve na kód funkce ***mergeSort***, který budeme následovně demonstrovat na praktickém příkladu:

```
mergeSort(c: nodePtr): nodePtr
  n = 1
  new(head)
  head↑.next = c
  do
        todo = head↑.next
                              ← (1)
        c = head
        repeat
              t = todo
              a = t
              for i = 1 to (n-1)
                    t = t↑.next
              end for
              b = t↑.next
              t↑.next = z
              t = b
              for i = 1 to (n-1)
                    t = t↑.next
              end for
              todo = t↑.next          ← (2)
```

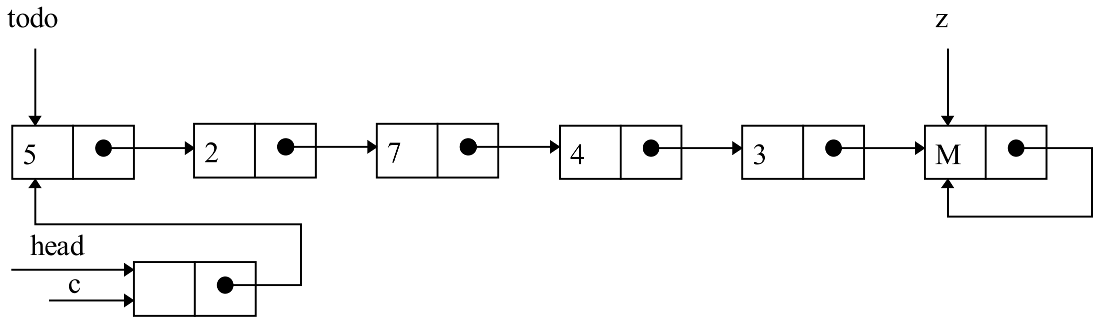
```
                tↁ.next = z

                c↑.next = merge(a, b)        ← (2)
                for i = 1 to 2*n
                      c = c↑.next
                end for
          until (todo = z)
                                  ← (3)
          n = n + n
    while (a <> head↑.next)
  end mergeSort
```

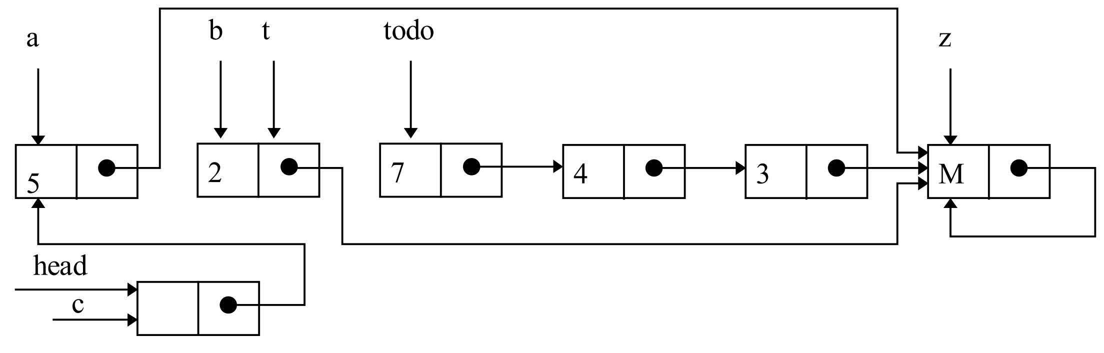
Nerekurzivní verze ***mergeSort*** nejdříve setřídí vybudovaný seznam po dvojicích, následovně po čtveřicích atd. dokud nezískáme setříděnou posloupnost. Proměnná ***n*** udává počet prvků vstupujících v jednotlivých seznamech do funkce ***merge***. Tedy při prvním průchodu je ***n* = 1**, v dalších 2, 4, 8 atd. V programu je dále zřízen ukazatel ***head***, který vždy ukazuje na pomocný prvek seznamu. Tento pomocný prvek vždy ukazuje na první prvek již setříděného seznamu. Dále zde existuje několik pomocných ukazatelů, které mají následující určení: ***todo*** – ukazuje na zbytek seznamu, který ještě nebyl v daném průchodu tříděn; ***a*** – ukazuje na první část seznamu, která bude vstupovat jako parametr do metody ***merge***; ***b*** – ukazuje na druhou část seznamu, která bude vstupovat jako parametr do metody ***merge***; ***c*** – ukazuje na poslední prvek již setříděné části seznamu v daném průchodu.

Příklad:

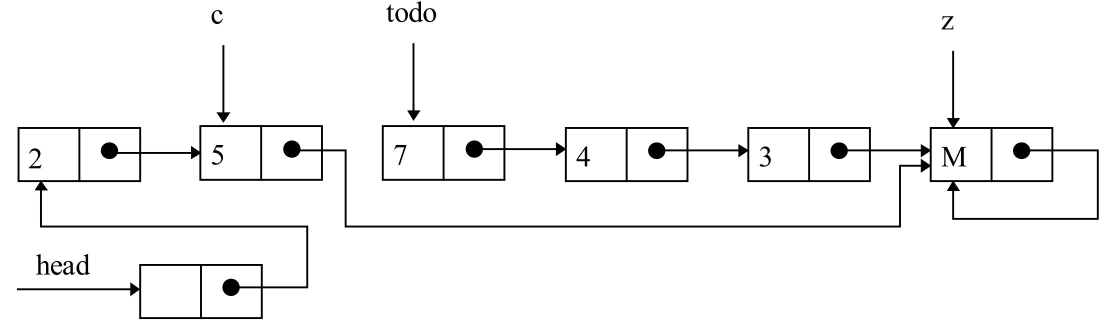
Setřiďte posloupnost čísel: 5, 2, 7, 4, 3.



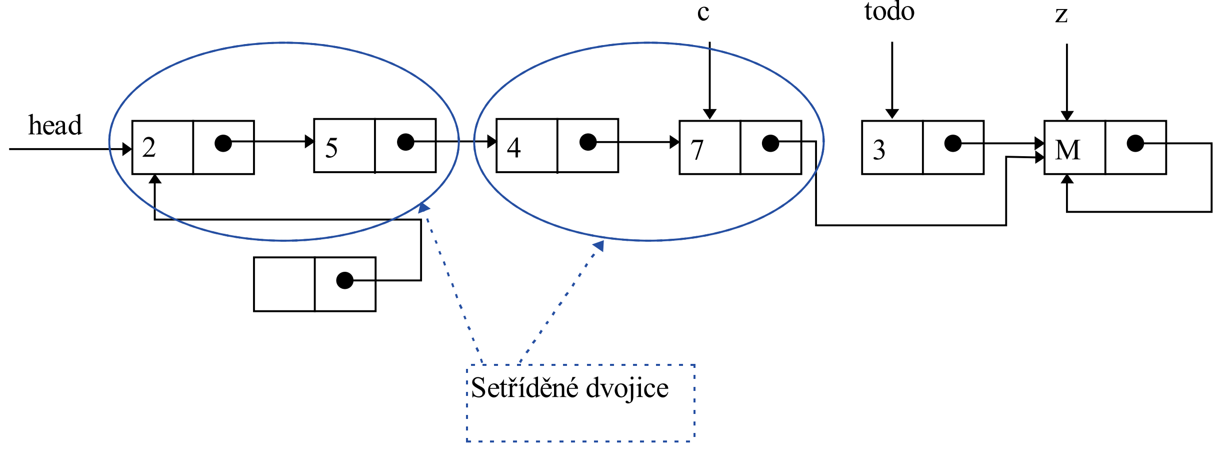
Obrázek 5.19 Stav seznamu v místě 1



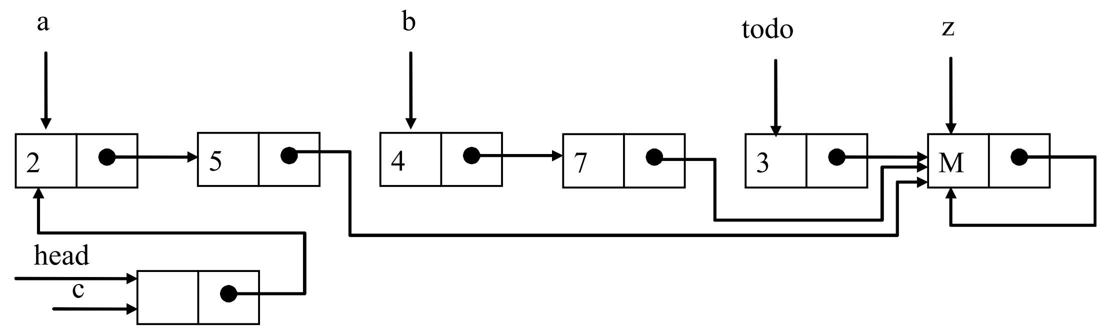
Obrázek 5.20 Stav seznamu v místě 2 pro *n* = 1, první průchod vnitřním cyklem *do*



Obrázek 5.21 Stav seznamu v místě 3 pro *n* = 1, první průchod vnitřním cyklem *do*



Obrázek 5.22 Stav seznamu v místě 3 pro *n* = 1, druhý průchod vnitřním cyklem *do*



Obrázek 5.23 Stav seznamu v místě 2 pro *n* = 2, první průchod vnitřním cyklem *do*

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ĺ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ĺ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

6. LEKCE – EXTERNÍ TŘÍDĚNÍ

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

6 EXTERNÍ ŘÍZENÍ ... 3

6.1 Algoritmus dvoucestného slučování (2 - way merging) ... 4

POUŽITÁ LITERATURA ... 9

6 EXTERNÍ ŘÍZENÍ

OBSAH KAPITOLY:

lgoritmus dvoucestného slučování (2 - way merging)

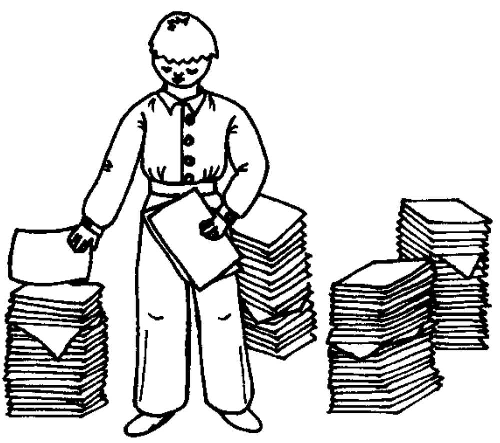
MOTIVACE:

Velké objemy dat není možno uložit do paměti, a přesto musíme být schopni je efektivně setřídit. K tomu je potřeba znát alespoň základní algoritmy externího třídění a umět je efektivně použít.

CÍL:

Pro externí algoritmy třídění umět použít algoritmus jednocestného slučování, dvoucestného slučování, přirozeného slučování a umět zhodnotit jejich efektivitu.

Metody třídění, kterými jsme se až doposud zabývali, předpokládaly, že se všechna tříděná data vejdou do operační paměti počítače. V mnoha případech však nelze tento základní předpoklad splnit. V tomto případě musíme použít metody využívající principy externího třídění. U externího třídění však vyvstává problém přístupu na paměťové médium, na němž jsou uložena data. Jde o externí paměťová média, nejčastěji pevné disky, avšak mohou to být i různá magnetopásková zařízení. V této chvíli je si nutné také uvědomit způsob přístupu k souborům na těchto médiích. Některá média umožňují jak sekvenční, tak náhodný přístup. Jiná média umožňují pouze sekvenční přístup. U algoritmů externího třídění bude používat pouze ***sekvenční přístup***, protože i u zařízení s náhodným přístupem je rychlejší. Při posuzování složitosti algoritmu externích třídících metod se budeme věnovat pouze sledování počtu přístupů k externím paměťovým médiím. Počty porovnání dvou prvků v paměti nebo jejich přesuny jsou naprosto zanedbatelné oproti počtům načtení souboru z média. Navíc i sledování I/O operací není zcela vypovídající, protože řešení je velice silně závislé na použitém hardware (buffery zařízení, propustnosti sběrnic,...) a konfiguraci operačního systému (použití vyrovnávacích pamětí apod.)



Obrázek 6.1 Externí třídění

6.1 ALGORITMUS DVOUCESTNÉHO SLUČOVÁNÍ (2 - WAY MERGING)

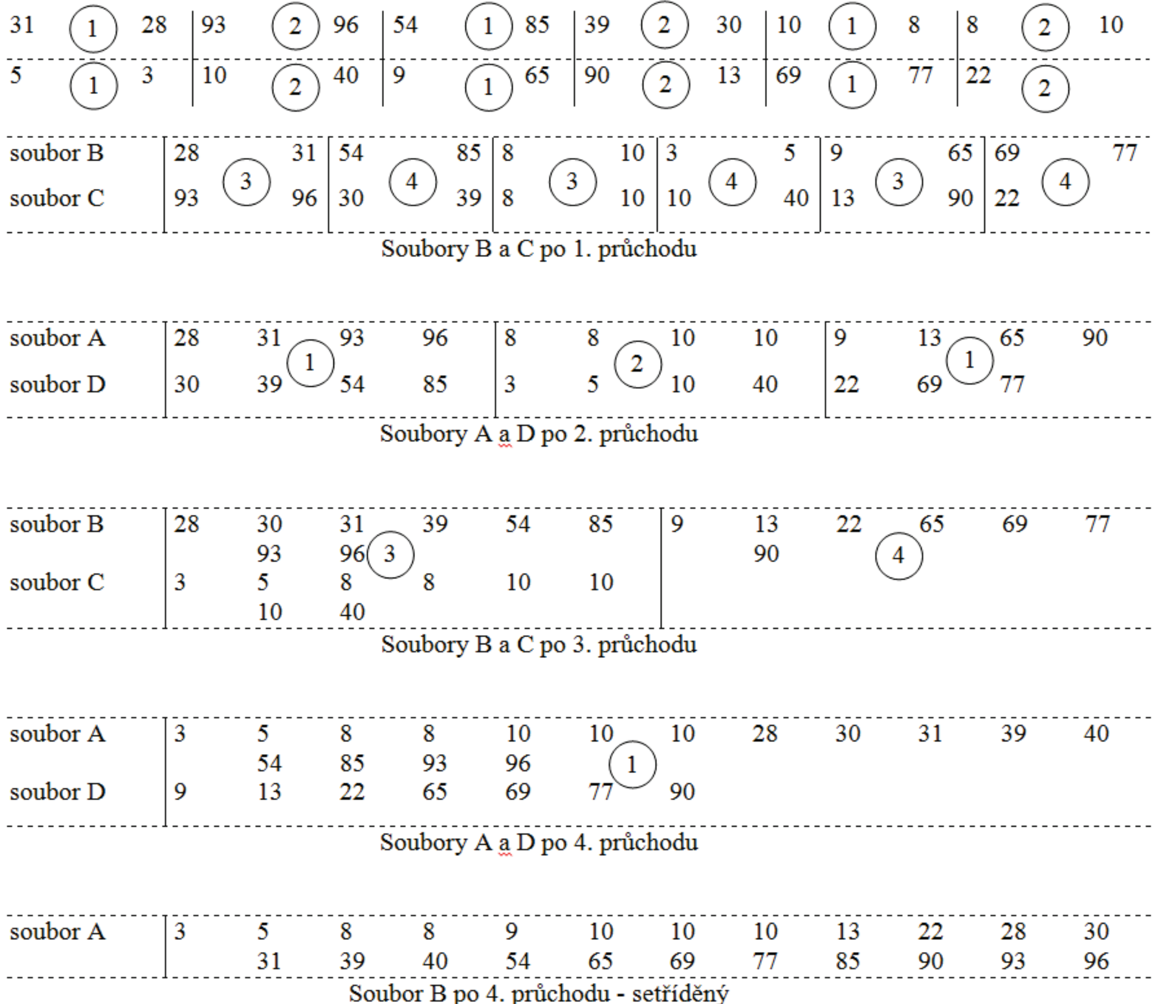
Algoritmus dvoucestného slučování vychází z metody slučování (merge sort) popsané v předchozí kapitole.

Základní myšlenka:

Každý průchod programem lze rozdělit do fáze rozdělování a do fáze slučování. Ve fázi rozdělování jde o rozdělení souboru, řekněme *A*), do dvou dalších souborů tak, že do souboru ***B*** budou uloženy liché prvky souboru ***A*** a do souboru ***C*** sudé. Následovně je provedeno sloučení souborů ***B*** a ***C*** do souboru ***A*** tak, že jsou vždy složeny jednotlivé prvky ze souboru ***B*** a ***C*** tak, že soubor ***A*** je nyní tvořen setříděnými dvojicemi prvků. V dalším průchodu je soubor ***A*** rozdělen do souborů ***B*** a ***C*** tak, že každého jsou zapsány liché respektive sudé dvojice prvků. Následuje opět slučování, nyní však po dvojicích ze souborů ***B*** a ***C*** do souboru ***A***. Celá operace se opakuje, dokud nejsou v souboru ***A*** data setříděna. Uvedený algoritmus však není moc dobře navržen, protože fáze rozdělování je neefektivní – neprobíhá během ní třídění a znamená jedno čtení a jeden zápis celého souboru dat.

Algoritmus lze proto zefektivnit tím, že použijeme ještě jeden pomocný soubor – ***D***. To může být, jak uvidíme, výhodnější z hlediska počtu přístupů k souboru, avšak někdy nemožné z hlediska kapacity dostupných paměťových médií. Podívejme se však, v čem bude modifikace spočívat. Na počátku rozdělíme soubor ***A*** do souboru ***B*** a ***C*** podle známého klíče. Nyní však budeme provádět slučování tak, že každou sloučenou dvojici střídavě ukládáme do souborů ***A*** a ***D***. Tím jsme eliminovali fázi rozdělování, která měla následovat a může proto ihned spustit fázi slučování, kdy jednotlivé dvojice ze souborů ***A*** a ***D*** slučujeme a ukládáme střídavě do souborů ***B*** a ***C***. Zamyslíme-li se nad uvedeným algoritmem, zjistíme, že úpravou jsme zkrátili potřebný čas k setřídění souboru na polovinu.

Příklad: počáteční soubor je dán následující tabulkou, ve které je přímo označeno, jak budou jednotlivé dvojice sloučených prvků umísťovány do souborů ***B*** a ***C***:



Obrázek 6.2 Příklad dvoucestného slučování

Analýza algoritmu:

Pro jednoduchost předpokládejme, že máme množinu o $n = 2^k$ prvků, kde k je přirozené číslo. V prvním průchodu budeme slučovat 2^k jednoprvkových úseků, ve druhém 2^{k-1} atd. V posledním k-tém průchodu pak sloučíme dva úseky o 2^{k-1} prvcích. Celkově tedy musíme provést $k = \log_2 n$ průchodů. Pro libovolné n pak platí:

$$C_p = \lceil \log_2 n \rceil \tag{6.1}$$

Poznámka:

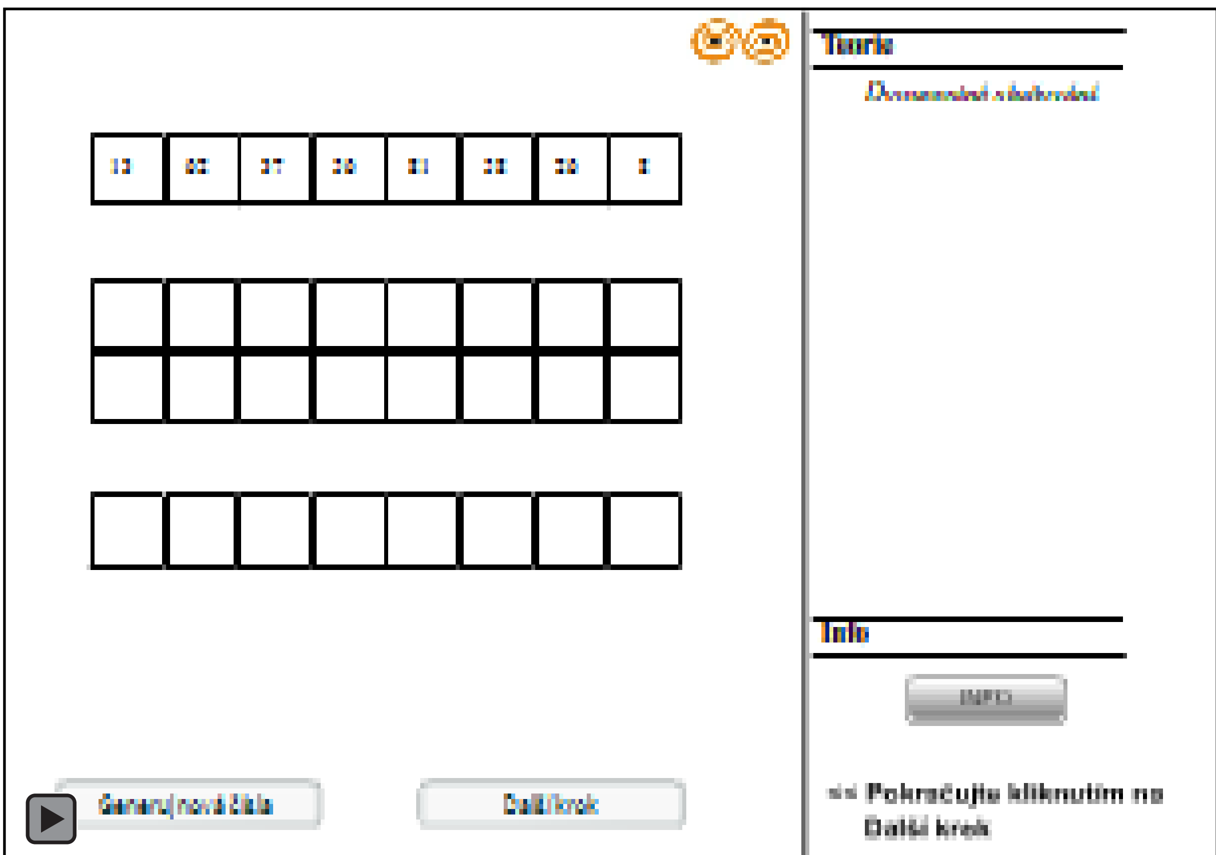
$\lceil \ \rceil$ - stropní funkce

protože, je-li $\boldsymbol{n}$ v intervalu $(\mathbf{2}^{k-1}, \mathbf{2}^{k})$, pak bude třeba vždy $\boldsymbol{k}$ průchodů. Počet průchodů se tedy vždy zaokrouhlí směrem nahoru. U první verze algoritmu se využívají dva pomocné soubory a každý průchod byl složen z fáze rozdělování a fáze slučování. Fáze rozdělování znamená načtení a uložení celého souboru a fáze slučování taktéž. Tedy při každém průchodu dojde ke dvěma načtením a dvěma uložením celého souboru dat. Pak tedy:

$$C_{ps} = 4n(\lceil \log_2 n \rceil) \tag{6.2}$$

udává celkový počet přístupů do souboru. Přístupem se rozumí načtení nebo zápis jednoho prvku. U druhé verze algoritmu byla eliminována fáze rozdělování a tak počet přístupů do souborů bude poloviční.

Dalšího zrychlení algoritmu lze dosáhnout, pokud budeme slučovat $\boldsymbol{n}$-tice nekonstantní délky. To proto, že v každém souboru dat se mohou vyskytovat úseky (shluky) dat, které jsou již podle daného klíče setříděny. Pak je určitě zbytečné tyto setříděné shluky rozdělovat dříve uvedeným algoritmem, aby následovně došlo k jejich opětovnému sloučení. Algoritmus, který umožňuje pružně reagovat na uvedenou situaci je označován jako metoda ***Přirozeného slučování***.



Příklad:

původní soubor (setříděné úseky jsou odděleny svislou čarou)

17 31 | 5 59 | 13 41 43 67 | 11 23 29 47

5 17 31 59 | 11 13 23 29 41 43 47 67

5 11 13 17 23 29 31 41 43 47 59 67

Obrázek 6.3 Metoda přirozeného slučování

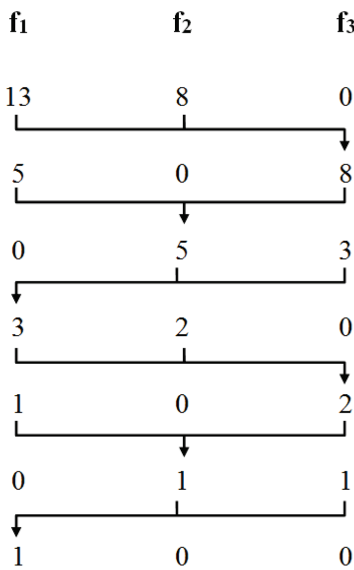
Při provádění rozdělování ***n***-tic (nestejné délky), je důležité, aby obě dílčí posloupnosti měly stejný počet ***n***-tic. Stejná délka je druhořadá. Při slučování jednotlivých ***n***-tic – ***běhů*** – může dojít k jistým komplikacím. Ty vyplývají především z faktu, že počet běhů může být v jednotlivých souborech rozdílný, i když v rozdělovací fázi zapisujeme střídavě do obou souborů. Počty běhů by se tedy mohly teoreticky lišit pouze o hodnotu ***1*** tak, jak tomu bylo v původní verzi algoritmu. V souborech však může dojít k automatickému sloučení dvou po sobě jdoucích běhů. K tomu dojde, pokud poslední prvek ***i-1*** běhu je menší než první prvek běhu ***i***-tého.

Dalším zrychlením může být použití více souborů (pásek), což vede na metodu ***Polyfázového slučování***. Princip spočívá ve slučování z ***n-1*** souborů do ***n***-tého souboru. Jakmile je některý ze vstupních souborů prázdný, přepíná se výstup do tohoto souboru a původní výstupní soubor se stává vstupním.

Uvedený algoritmus je výhodný zejména při použití více souborů (pásek). Důležitá je u tohoto algoritmu především počáteční distribuce *n*-tic. Pro tři soubory počty *n*-tic tvoří ***Fibonacciho posloupnost*** (1. řádu)

0 1 1 2 3 5 8 13 21 34 55

kdy každé číslo je součtem dvou předcházejících.



Obrázek 6.4 Polyfázové třídění se třemi soubory

Pro více souborů platí, že počáteční počet n-tic se má rovnat součtu (n – 1), (n – 2), ..., 1 po sobě jdoucích čísel Fibonacciho čísel (n – 2) řádu. Obecně je Fibonacciho posloupnost čísel definována pro řád ***p***:

$$\begin{aligned} & f_{i+1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + \ldots + f_{i-p}^{(p)} \qquad \text{pro i} \geq \text{p} \\ & \left. \begin{aligned} f_p^{(p)} &= 1 \\ f_i^{p} &= 0 \end{aligned} \right\} \quad \text{pro } 0 \leq \text{i} < \text{p} \end{aligned} \tag{6.3}$$

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ľ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ľ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms.* 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

7. LEKCE - VYHLEDÁVÁNÍ

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

7 VYHLEDÁVÁNÍ ... 3

7.1 Hešování (Hashing) ... 4

7.1.1 Lineární zkoušení (linear probing) ... 5

7.1.2 Dvojité hešování (double hashing) ... 6

7.2 Stromy s více potomky ... 8

7.2.1 2-3-4 stromy ... 9

7.3 Vyhledávání textových řetězců ... 10

7.3.1 Brute-Force algoritmus ... 10

7.3.2 Boyer-Moore algoritmus ... 11

POUŽITÁ LITERATURA ... 13

7 VYHLEDÁVÁNÍ

OBSAH KAPITOLY:

Hešování (Hashing)

Stromy s více potomky

Vyhledávání textových řetězců

MOTIVACE:

Vyhledávání ve velkých objemech strukturovaných i nestrukturovaných dat je velmi častá činnost, proto je dobré znát alespoň základní a přitom efektivní algoritmy vyhledávání.

CÍL:

Seznámit se s několika efektivními algoritmy vyhledávání ve strukturovaných i nestrukturovaných datech. Umět používat algoritmus vyhledávání hešováním, jak v podobě lineárního zkoušení, tak dvojitého hešování. Umět pro vyhledávání používat stromy s více potomky, jako jsou 2-3-4 stromy. Umět používat základní algoritmy pro vyhledávání v textech, jako je Brute-Force algoritmus nebo Boyer-Moore algoritmus

7.1 HEŠOVÁNÍ (HASHING)

Hlavní myšlenka:

U metody ***hešování*** je cílem najít umístění záznamu v tabulce tak, že provedeme aritmetickou transformaci klíče tak, abychom získali adresu v tabulce. Pravidla určující transformaci se nazývají ***hešovací funkce***.

Předpokládejme, že máme prvky s klíči obsahujícími celočíselné unikátní hodnoty v rozmezí **1 ..** ***N***. Pak můžeme použít tabulku s indexy v rozsahu **1 ..** ***N*** a hešovací funkci $\boldsymbol{h(k) = k}$. V tomto extrémním případě vyvstává problém pro velká *N*. Je totiž nutné vytvořit pole o ***N*** prvcích, přičemž mnoho prvků pole může být neobsazeno. Tedy v tomto případě je možno hledaný prvek velice rychle nalézt v poli, avšak za velmi vysoké provize v paměti počítače. Druhým extrémem může být např. následující hešovací funkce :

$$h(k) = d_n + d_{n-1} + d_{n-2} + \cdots + d_1 + d_0 \tag{7.1}$$

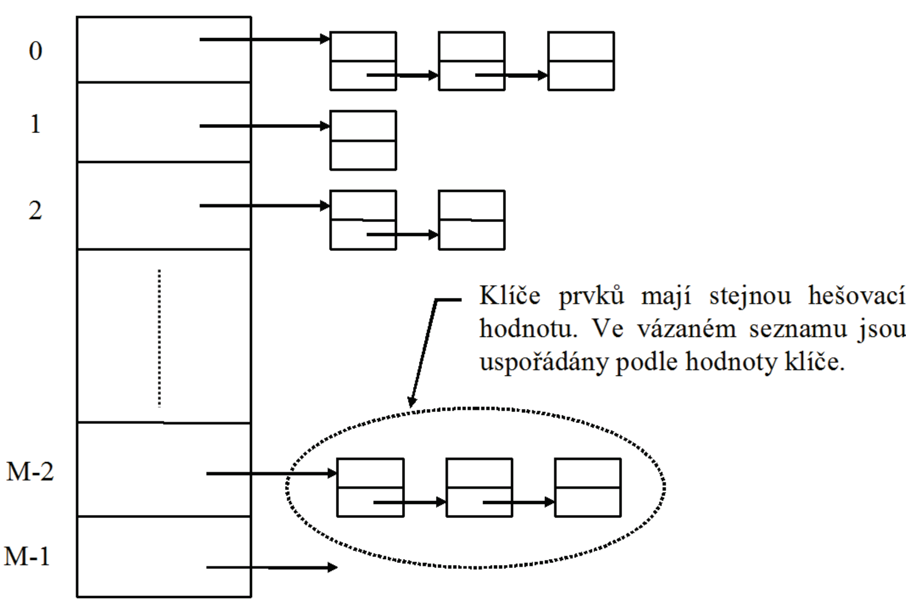
kde $d_n, d_{n-1}, \ldots, d_1, d_0$ jsou jednotlivé číslovky klíče.

Pak např.

h(1234) = h(4321) = h(2134) = h(1000432) = h(91)

V tomto druhém extrému se vytváří mnoho tzv. ***kolizí*** v hešovací tabulce. Jinými slovy mnoho různých klíčů má stejnou mapovací adresu do hešovací tabulky, danou hešovací funkcí. Dobrá hešovací funkce leží někde mezi uvedenými extrémy. Prakticky se např. používá hešovací funkce $\boldsymbol{h(k) = k \bmod M}$, kde ***k*** je celé kladné číslo a ***M*** je prvočíslo.

Samozřejmě, vždy může nastat kolize pro dva nebo více klíčů a tak je nutné použít tzv. ***kolizní metodu***. Jednou z metod může být použití např. dynamicky vázaného setříděného seznamu pro uchování prvků se stejnou mapovací adresou.



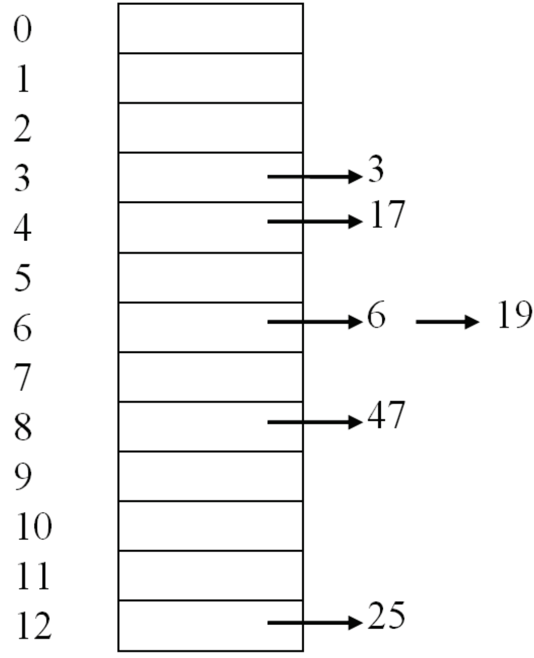
Obrázek 7.1 Použití dynamického seznamu pro řešení kolizí v hešovací tabulce

Hešovací tabulka pak má ***M*** ukazatelů na typ prvek.

Uvažujeme následující posloupnost klíčů: 3, 17, 25, 19, 47, 6 a $M = 13$

Rozbor algoritmu.

Pro náhodně vygenerované klíče je délka vázaného seznamu v průměru ***N/M***. Jelikož každý seznam je setříděn, pak musíme v průměru prohlédnout polovinu prvků pro nalezení konkrétního prvku. Tedy pro nalezení prvku potřebujeme v průměru ***N/2M*** iterací.



Obrázek 7.2 Příklad tabulky s dynamickým seznamem

Dalším způsobem, jak vyřešit kolize mezi adresami, je použití tzv. ***otevřeného adresování (open addressing)***. Tato metoda vychází z předpokladu, že při zaplňování hešovací tabulky jsou některé její prvky nezaplněny a jejich adresy lze právě použít pro vyřešení nastalých kolizí. Otevření adresování lze realizovat dvěmi algoritmy:

1. **lineární zkoušení (linear probing)**
2. **dvojité hešování (double hashing)**

7.1.1 Lineární zkoušení (linear probing)

Když vytváříme hešovací tabulku a dojde ke kolizi, vyhledáme nejbližší volnou pozici a vložení prvku provedeme na tuto pozici. Prohledávání začíná vždy od vypočtené adresy směrem k vyšším adresám.

Příklad: předpokládejme prvky s následujícími klíči: 1*, 19, 5*, 1**, 18, 3, 8, 9, 14, 7, 5**, 24, 1***, 13, 16, 12, 5***.

Tedy ***N = 17*** a dále zvolme ***M = 19*** (prvočíslo)

Pro hešování použijeme funkci ***h(k) = k mod M***

Tabulka 7.1 Plnění tabulky metodou lineárního zkoušení

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	1*	1**			5*													
19	1*	1**	3		5*	5**	7	8	9					14				18
19	1*	1**	3	1***	5*	5**	7	8	9	24				14				18
19	1*	1**	3	1***	5*	5**	7	8	9	24	5***	12	13	14		16		18

Tabulka ukazuje postup vkládání jednotlivých prvků do hešovací tabulky. Řádky postupně promítají obsah tabulky vždy do místa kolize. Další pokračování vkládání se pak děje, pro větší přehlednost, v dalším řádku.

Pokud nyní potřebujeme vyhledat prvek v tabulce, musíme nejdříve použít hešovací funkci k nalezení indexu (adresy) v tabulce a pokud hledaný prvek zde není, musíme použít metodu lineárního prohledávání směrem doprava.

7.1.2 Dvojité hešování (double hashing)

U metody dvojitého hešování je zvolen následující postup při kolizi dvou prvků v tabulce :

3. vypočti adresu v hešovací tabulce a v případě, že nedošlo ke kolizi, použij ji,
4. pokud došlo ke kolizi, použij druhou hešovací funkci pro výpočet inkrementu, který je přičten k adrese vypočtené v prvním kroku,
5. pokud dojde opět ke kolizi, opakuj krok 2.

Prakticky se jako první hešovací funkce používá nám již známá:

$$h1(k) = k \bmod M, \tag{7.2}$$

zatím co pro druhé hešování funkce:

$$h2(k) = M - 2 - (k \bmod (M - 2)) \tag{7.3}$$

Uvažujeme stejnou posloupnost prvků s klíči jako v minulém příkladu a dále uvažujeme následující hešovací funkci:

$$h1(k) = k \bmod 19$$

a

$$h2(k) = 17 - (k \bmod 17)$$

hodnoty hešovacích funkcí pro jednotlivé prvky jsou uspořádány do tabulky:

Tabulka 7.2 Tabulka hodnot hešovacích funkcí pro jednotlivé tříděné prvky

klíč	1*	19	5*	1**	18	3	8	9	14	7	5**	24	1***	13	16	12	5***
h1	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5
h2	16	15	12	16	16	14	9	8	3	10	12	10	16	4	1	5	12

Hešovací tabulka pak bude vypadat následovně:

Tabulka 7.3 Hešovací tabulka pro metodu dvojitého hešování

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	1*		3		5*	5***	7	8	9	5**	1***	12	13	14	24	16	1**	18

Poznámka:

Pro umístění prvku s klíčem 5*** je nutné pětinásobného použití hešovací funkce h2.

Rozbor algoritmu:

Algoritmus může mít na jedné straně velice dobrou výkonnost, avšak na straně druhé i velice nedostatečnou. To zvláště v případě, že nastává mnoho kolizí a metoda řešení kolizí klíčů není zrovna nejlepší. Taková situace nastane např. v případě, kdy je zvolena metoda ***kvadratických pokusů***, kdy hešovací funkce použitá pro řešení kolize má tvar $h_i = i^2 \bmod N$, kde i je číslo kolize. Při použití této metody může nastat situace, kdy kolize není vyřešena a volné místo v tabulce není nalezeno i přesto, že existuje.

Podívejme se nyní, jak odvodit průměrný počet pokusů o vložení nového klíče nebo jeho vyhledání v existující tabulce. Předpokládejme, že všechny klíče mají stejnou pravděpodobnost výskytu a hešovací funkce tyto klíče distribuuje rovnoměrně do buněk tabulky, jejíž velikost je N a obsahuje již k prvků. Pravděpodobnost, že zařazovaný prvek se podaří umístit na první pokus, je $1 - k/N$. Pravděpodobnost, že bude nutné použít pouze jeden sekundární pokus, je rovna:

$$P_2 = \frac{k}{N} \cdot \frac{n-k}{N-1} \tag{7.4}$$

Pravděpodobnost zásahu volného místa v tabulce na i-tý pokus lze vyjádřit vztahem:

$$P_i = \frac{k}{N} \cdot \frac{k-1}{N-1} \cdot \frac{k-2}{N-2} \cdot \ldots \cdot \frac{k-i+2}{N-i+2} \cdot \frac{N-k}{N-i+1} \tag{7.5}$$

Počet pokusů k přidání $(k + 1)$ klíče do tabulky o N prvcích je tedy :

$$E_{k-1} = \sum_{i=1}^{k+1} ip_i = 1\frac{N-k}{N} + 2\frac{k}{N}\frac{N-k}{N-1} + \ldots + (k+1)\left(\frac{k}{N} \cdot \frac{k-1}{N-1} \cdot \frac{k-2}{N-2} \cdot \ldots \cdot \frac{k-i+2}{N-i+2} \cdot \frac{1}{N-i+1}\right)$$
$$E_{k-1} = \frac{N-1}{N-k+1} \tag{7.6}$$

Budeme-li chtít vyčíslit střední hodnotu pokusů potřebných pro vyhledání klíče v tabulce, můžeme použít stejného vztahu jako pro výpočet středního počtu pokusů na přidání prvku do tabulky. Potom

$$E = \frac{1}{M}\sum_{k-1}^{M} E_k = \frac{N-1}{M}\sum_{k=1}^{M}\frac{1}{N-k+2} = \frac{N+1}{M}\left(H_{N+1} - H_{N-M+1}\right) \tag{7.7}$$

kde

$$H_N = 1 + \frac{1}{2} + \ldots + \frac{1!}{N} \tag{7.8}$$

Aproximací HN získáme vztah $H_N \cong \ln(N) + \gamma$,kde γ je ***Eulerova konstanta***. Dále pak substitucí

$$\alpha = \frac{M}{N+1} \tag{7.9}$$

dostaneme vztah:

$$E = \frac{1}{2}\left(\ln(N+1) - \ln(N-M+1)\right) = \frac{-1}{\alpha}\ln(1-\alpha) \tag{7.10}$$

Koeficient α vyjadřuje tzv. koeficient zaplnění tabulky a vyjadřuje poměr obsazených a volných míst v tabulce. Vyčíslíme-li pro několik hodnot α počet pokusů o vyhledání prvku a umístíme-li výsledky do tabulky, získáme:

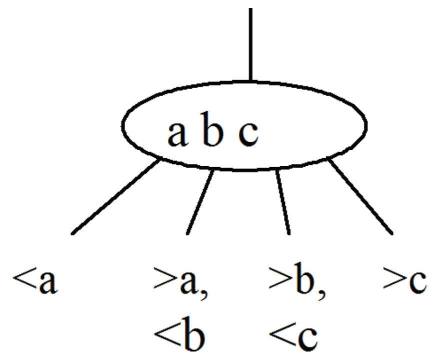
Tabulka 7.4 Závislost počtu pokusů na koeficientu zaplnění tabulky

α	E
0.10	1.05
0.25	1.15
0.50	1.39
0.75	1.85
0.90	2.56
0.95	3.15
0.99	4.66

Na základě získaných výpočtů lze prohlásit, že docela uspokojivých výsledků získáme i při 95% naplnění tabulky, kdy bude potřeba v průměru 3,15 pokusů o naplnění nebo získání hodnoty. Navíc je nutné si uvědomit, že uvedený počet pokusů není závislý na počtu prvků v tabulce, ale na koeficientu naplnění tabulky.

Výhodou principu hešování je relativně vysoká výkonnost algoritmu. Hešování však není možné vždy dobře implementovat. To zvláště v případech, kdy neznáme dopředu počet ukládaných prvků a tedy nemůžeme dobře odhadnout statický rozměr tabulky. Dále pak uvedený algoritmus není vhodný, když potřebujeme odebírat prvky z tabulky. Rušení prvků z tabulky je velmi složité, kromě metody řešící kolize zřetězením stejných klíčů pomocí lineárního dynamického seznamu.

7.2 STROMY S VÍCE POTOMKY



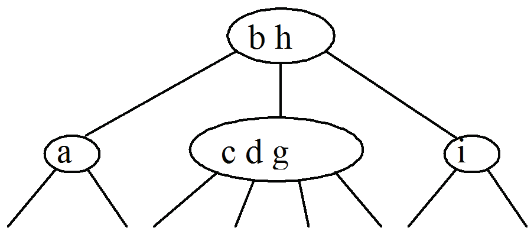
Obrázek 7.3 B-strom, m=4

Další velice často používanou metodou pro vyhledávání prvků je použití binárních vyhledávacích stromů. Binární stromy, jak bylo již uvedeno v kapitole věnované stromům, nejsou vhodné, protože může při jejich výstavbě dojít k vysoké nevyváženosti nebo dokonce k degradace v jednosměrně vázaný seznam. Jeden ze způsobů jak vyvážit binární strom je použít jeho modifikaci – ***AVL strom***. Třetím způsobem, jak vyvážit strom je umožnit, aby uzly mohly mít více než dva potomky. Místo binárních stromů se pak používají ***stromy s více potomky*** (***multi-way trees***). Ty pak obsahují uzly s ***n***-potomky a tedy ***n***-vazbami a uzel má pak ***n*** – **1** klíčů. Uzly s klíči menšími než ***a*** jsou vkládány do podstromu 1. Uzly s klíči mezi hodnotami ***a*** a ***b*** pak do podstromu 2. Podobně je tomu s prvky v rozmezí ***b*** a ***c***, které jsou vloženy do podstromu 3. Do podstromu 4 jsou pak vloženy prvky, které mají hodnotu klíče > než ***c***. Pro daný typ uzlů bude mít strom menší výšku než balancovaný binární strom, protože je schopen uchovat více klíčů v jednom uzlu. Stromy s více vazbami jsou tedy lehčeji vyvážitelné než binární stromy.

Jedním speciálním stromem je tzv. ***B-strom*** (***Bayerův strom***). B-strom stupně ***m*** má uzly, které mohou mít až ***m*** potomků. Nyní se podíváme na konkrétní případ, kdy ***m=4***, tedy B-stromy 4.stupně, které jsou často nazývány ***2-3-4 stromy***.

7.2.1 2-3-4 stromy

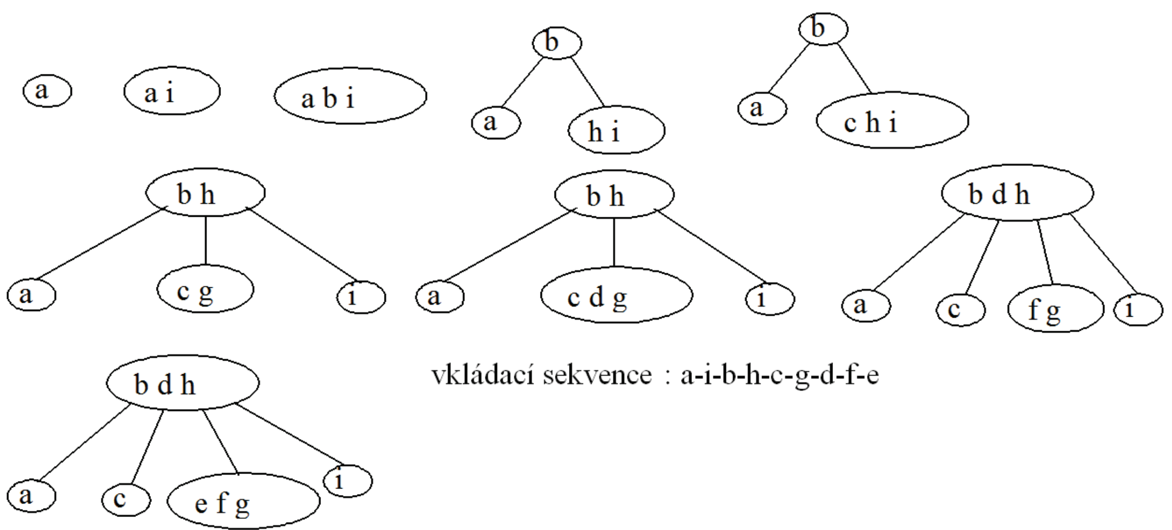
Název je odvozen od skutečnosti, že 2-3-4 strom může mít uzly s dvěmi, třemi nebo 4-mi potomky. Přidávání prvku do 2-3-4 stromu je odlišné od přidávání prvku do binárního stromu a



Obrázek 7.4 2-3-4 strom

vypadá následovně:

6. vyhledat místo, na které se má vložit nový uzel
7. pokud je místo vložení povrch (uzel) se dvěmi vazbami, stane se z něj uzel se 3-mi vazbami.
8. pokud je místo vložení uzel se 3-mi vazbami, stane se z něho uzel se 4-mi vazbami.
9. pokud je místo vložení uzel se 4-mi vazbami, je tento uzel už plný a je nutné jej rozdělit na dva uzly, každý se dvěma vazbami. Nový prvek je pak vložen do odpovídajícího uzlu, ze kterého se stane uzel se 3-mi vazbami.



Obrázek 7.5 Budování 2-3-4 stromu

Obrázek ukazuje, jak se rozděluje uzel se 4-mi vazbami. Prostřední klíč je vysunut směrem nahoru a stane se případně novým kořenem a další dva klíče jsou umístěny vpravo a vlevo od kořene a tvoří uzly se dvěmi vazbami. Rozložení uzlu se tedy děje vždy směrem nahoru a tím je zajištěno vyvážení stromu po celou dobu jeho výstavby.

Vkládání a tedy budování se sestává obecně ze dvou činností -

10. prohledání stromu seshora dolů
11. rozložení uzlů zespoda nahoru

Jak bylo vidět na obrázku, strom zůstává vyvážen i při použití známé patologické sekvence[1] klíčů.

7.3 VYHLEDÁVÁNÍ TEXTOVÝCH ŘETĚZCŮ

Jednou z častých úloh při zpracování informace je vyhledání tzv. vzoru v poli dat. Nejčastěji je uvedená úloha realizována jako vyhledání podřetězce (vzoru) v řetězci znaků. V uvedeném případě je pole dat jednorozměrné. Existují však úlohy, kdy pole dat je i vícerozměrné. Jako příklad můžeme uvést úlohu zpracování dvojrozměrného nebo dokonce trojrozměrného obrazu, kdy se snažíme například o nalezení jistého detailu na fotografii, nebo informaci z průmyslové kamery. Následovně uvedené algoritmy převážně řeší první typ úlohy, tedy úlohu vyhledání vzory v textovém řetězci.

7.3.1 Brute-Force algoritmus

Základní myšlenka tohoto algoritmu je velice jednoduchá a spočívá v prohledání každé případné pozice v daném textu na výskyt vzoru. Předpokládejme tedy, že textový řetězec, který se má prohledávat je uložen v poli znaků ***a*[1 .. *N*]** a hledaný vzor v poli ***p*[1 .. *M*]**. V následujícím funkci jsou použity dvě indexové proměnné ***i*** pro textový řetězec a ***j*** pro vzor. Funkce má dva formální parametry obsahující vzor a řetězec a dále pak vrací celočíselnou hodnotu určující nalezenou pozici vzoru v řetězci.

```
BruteForce(p : array of char, a : array of char): integer
  j=1
  i=1
  repeat
        if (a[i]=p[j])
              i=i+1
              j=j+1
        else
              i=i-j+2
              j=1
        end if
  until (j>M) OR (i>N)
  if (j>M)
        BruteForce = i-M
  else
        BruteForce = 0
  end if
end BruteForce
```

Rozbor algoritmu:

[1] Patologická sekvence způsobí vytvoření degenerovaného binárního stromu. Degenerovaný strom je takový, který je absolutně nevyvážený.

Brute-Force algoritmus je nejjednodušší ze série uvedených algoritmů, avšak také nejpomalejší. Algoritmus je citlivý na uspořádání vstupních dat. Jeho složitost je pro nejhorší případ (vzor není obsažen) velice jednoduše vyčíslitelná a je rovna ***O*(*M*N*)**, zatím co pro nejlepší případ (vzor je hned na počátku řetězce) je rovna ***O(M+N)***.

7.3.2 Boyer-Moore algoritmus

Základní myšlenka algoritmu spočívá ve vybudování tabulky indexů posunutí. Její velikost je dána množinou znaků, které se mohou vyskytnout v prohledávaném a hledaném řetězci. Všechny prvky tabulky jsou implicitně inicializovány na hodnoty ***N***, která udává délku vzoru. Prvky pole odpovídající znakům obsažených ve vzoru jsou následovně nahrazeny hodnotami ***N – j***, kde ***j*** je pozice znaku v poli vzoru.

Vyhledávání vzoru v řetězci se provádí v inverzním pořadí postupně od znaku s indexem ***N*** tak dlouho, dokud není nalezen rozdílný znak. Pokud je takový rozdílný znak nalezen, pak dojde k použití tabulky indexů k výpočtu posunutí vzoru vzhledem k prohledávanému řetězci. Nedochází tedy k posunutí o jeden znak, jak tomu bylo u Brute-Force algoritmu. Posunutí u Boyer–Moore algoritmu je dáno znakem na pozici ***N*** v prohledávaném řetězci. Pokud tento znak ve vzoru neexistuje, dojde k posunutí vzoru vzhledem k aktuální pozici o celou jeho délku – ***N***, protože je zcela jasné, že jakékoliv menší posutí by skončilo následovaným neúspěšným vyhledáním. V případě, že tento znak ve vzoru existuje, dojde k posunutí vzoru tak, aby se stejné znaky v prohledávaném řetězci a vzoru kryly.

Následující program pro zjednodušení předpokládá, že používáme pouze velká písmena abecedy, tedy A-Z. Tabulka indexů posunutí je uložena v poli ***skip***, prohledávaný text v poli znaků ***a*** a vzor v poli ***p***.

```
BoyerMoore(p : array of char, a : array of char): integer
  REM inicializuj tabulku
  for i=0 to 31 do
        skip[i]=N
  end for
  REM napln prvky odpovidajici znakum ve vzoru
  for i=1 to N-1 do
        skip[p[j]-'A']=M-i;
  end for
  i=N+1
  REM zacatek vyhledavani
  repeat
        j=N+1
        k=i
        repeat
              j=j-1
              if (j=0)
                    BoyerMoore = k    REM vzor nalezen
                    exit function
              end if
              k=k-1
        until (p[j]<>a[k])
        i=i+skip[a[i-1]]             REM posun vzor
  until (i>M+1)
  BoyerMoore = 0
end BoyerMoore
```

Příklad:

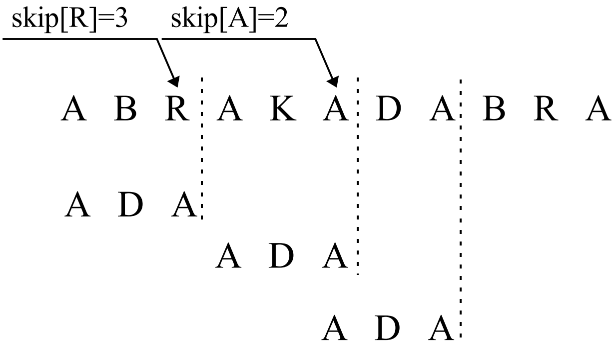
Vyhledejte slovo ADA ve slově ABRAKADABRA. (N = 3, M = 11.)

Tabulka skip bude obsahovat následující hodnoty:

Tabulka 7.5 Hodnoty

pozice (odpovídající znak abecedy)	hodnota
0 (A)	2
1 (B)	3
2 (C)	3
3 (D)	1
4 (E)	3
:	:
31 (Z)	3

Posuny slova ADA vzhledem k prohledávanému řetězci jsou pak:



Obrázek 7.6 Úpravy při odstranění uzlu – rotace

Rozbor algoritmu:

Boyer-Moore algoritmus je v současné době nejrychlejším algoritmem pro vyhledávání vzorů v řetězcích znaků. Proto je tak většinou implementován jako vyhledávací algoritmus pro textové řetězce v knihovnách různých programovacích jazyků. Jeho složitost je $O(M + N)$.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ĺ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ĺ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

8. LEKCE – ALGORITMY ŘEŠENÍ KOMBINATORICKÝCH ÚLOH

prof. Ing. Radim Farana, CSc.

Ostrava 2013

© prof. Ing. Radim Farana, CSc.
© Vysoká škola báňská – Technická univerzita Ostrava
ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

8 ALGORITMY ŘEŠENÍ KOMBINATORICKÝCH ÚLOH 3

8.1 Třída řešených problémů 4

8.2 Algoritmy neheuristické 6

8.2.1 Prohledávání do šířky 6

8.2.2 Prohledávání do hloubky 7

8.2.3 Vyzkoušení všech kombinací parametrů 8

8.3 Algoritmy heuristické 8

8.3.1 Heuristické prohledávání 9

8.3.2 Vyzkoušení všech slibných kombinací parametrů 10

8.4 Algoritmy pravděpodobnostní 10

8.4.1 Metoda pokusu a omylu 10

8.4.2 Genetický algoritmus 11

POUŽITÁ LITERATURA 14

8 ALGORITMY ŘEŠENÍ KOMBINATORICKÝCH ÚLOH

OBSAH KAPITOLY:

Třída řešených problémů

Algoritmy neheuristické

Algoritmy heuristické

Algoritmy pravděpodobnostní

MOTIVACE:

Řada úloh, které v technické praxi programově řešíme, je popsána jako výběr optimální varianty, která je závislá na konkrétních hodnotách parametrů. K určení, jak kvalitní jsou jednotlivá řešení, je definována ohodnocující funkce neboli účelová funkce (pokud známe pouze funkci, která popisuje kvalitu řešení přibližně, nazýváme ji heuristika). Pokud neznáme vztahy mezi jednotlivými parametry a přitom jak počet parametrů je konečný a jejich obor hodnot je dán množinou hodnot, řešíme úlohu jako kombinatorický problém.

CÍL:

Umět popsat kombinatorickou úlohu a definovat její omezení. Seznámit se se základními neheuristickými algoritmy – prohledáváním do šířky a do hloubky a prohledáváním do hloubky s omezenou hloubkou vnoření. Umět použít heuristické algoritmy prohledávání. Seznámit se pravděpodobnostními přístupy a základy genetických algoritmů. Znát principy a použití genetických algoritmů.

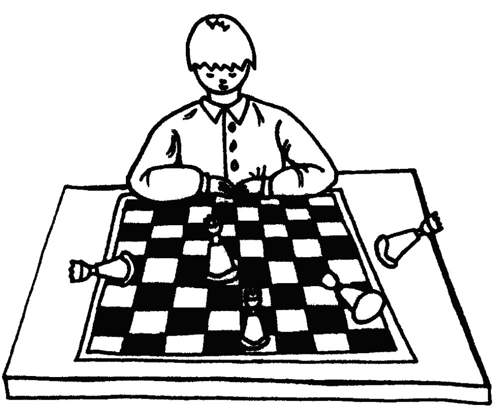
Řada úloh, které v technické praxi programově řešíme, je definována jako výběr ***optimální*** (nejlepší) ***varianty***, která je závislá na konkrétních hodnotách parametrů. K určení, jak kvalitní je které řešení, tedy můžeme definovat ***ohodnocující funkci*** neboli ***účelovou funkci*** (pokud známe pouze funkci, která popisuje kvalitu řešení přibližně, nazýváme ji ***heuristika***):

$$f\,(p_1, p_2, \dots, p_n), \qquad (8.1)$$

kde je f - ohodnocující funkce,

p_i - i-tý parametr.

Jak vidíme, konkrétní řešení je určeno konkrétními hodnotami jednotlivých parametrů. Kvalita řešení je pak dána hodnotou ***ohodnocující funkce*** (***heuristiky***) pro tyto parametry. Beze ztráty obecnosti můžeme předpokládat, že lepší řešení má vyšší hodnotu ohodnocující funkce. Při řešení problému manipulujeme s hodnotami parametrů tak, abychom dosáhli maximální (optimální) hodnoty ohodnocující funkce. Jedná se tedy o úlohu ***maximalizace ohodnocující funkce***. Tato problematika je široce řešena teoreticky, existuje řada optimalizačních metod (analytické, numerické). My se v dalším textu budeme zabývat vybranými speciálními metodami řešení uvedeného problému, které využívají postupů, uvedených v předchozích kapitolách.



Obrázek 8.1 Řešení kombinatorických úloh

8.1 TŘÍDA ŘEŠENÝCH PROBLÉMŮ

Pro naše potřeby provedeme následující zjednodušení:

1. **Omezení počtu parametrů**. Budeme pracovat jen s ohodnocujícími funkcemi, které mají známý počet parametrů. Dále budeme předpokládat, že hodnoty těchto parametrů jsou vzájemně nezávislé nebo se sice ovlivňují, ale nejsme schopni definovat funkční závislosti parametrů.
2. **Omezení oboru hodnot parametrů**. Budeme pracovat jen s parametry, jejichž hodnota je omezena povoleným rozsahem (oborem) hodnot. Dalším zjednodušením bude omezení oboru hodnot parametru na množinu hodnot. Budeme tedy pracovat s diskrétními parametry.

Na základě zavedených omezení se dostáváme do situace, kdy vlastně řešíme problém, jehož jednotlivá řešení jsou dána kombinací hodnot jednotlivých parametrů. Proto se tyto úlohy často nazývají ***kombinatorické***. V dalším textu rozebereme některé algoritmy řešení těchto problémů. Nejprve si však uvědomíme, kolik existuje různých řešení dané úlohy. Počet je dán následujícím vztahem:

$$r = \prod_{i=1}^{n} m_i \,, \tag{8.2}$$

kde je r - počet možných řešení,

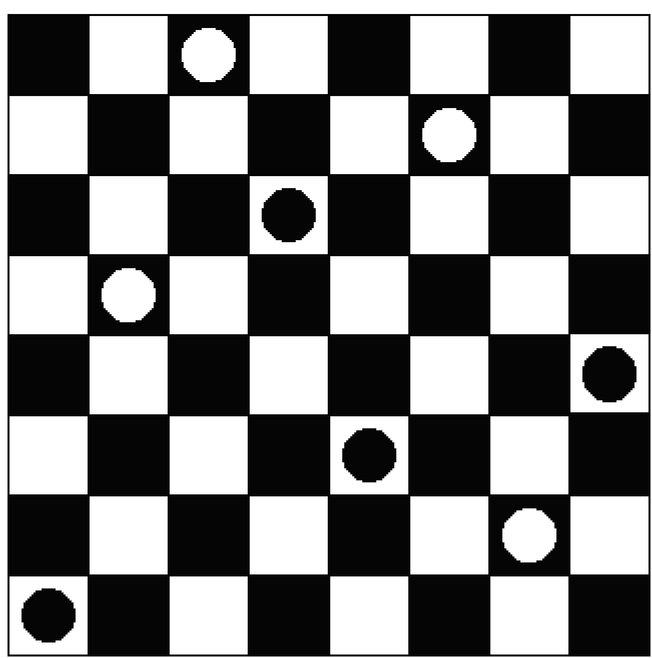
m_i - počet možných hodnot parametru p_i,

n - počet parametrů.

Počet různých řešení tedy s rostoucím počtem parametrů roste geometricky. Zde je základní potíž. Protože řešení jsou vzájemně nezávislá, měli bychom vyzkoušet všechna, abychom s plnou jistotou našli optimální řešení. V řadě algoritmů tomu tak naštěstí není. Abychom mohli ukázat zvolené metody řešení, vybereme nyní příklad, který budeme řešit.

Příklad:

Jedná se o úlohu rozestavení 8 dam na šachovnici tak, aby se vzájemně neohrožovaly. Při popisu úlohy vyjdeme ze skutečnosti, že v každém sloupci (1 ÷ 8) musí ležet právě jedna dáma.



Obrázek 8.2 Řešení úlohy rozložení osmi dam na šachovnici

Tato dáma musí ležet na jednom z řádků (1 ÷ 8). Úloha je tedy popsána pomocí osmi parametrů, které nabývají hodnot z množiny {1, 2, 3, ..., 8}. Ohodnocující funkci tedy můžeme definovat jako počet dam, které neohrožují jinou dámu a řešení nalezneme pro:

$$f(p_1, p_2, \ldots\ldots, p_8) = 8, \tag{8.3}$$

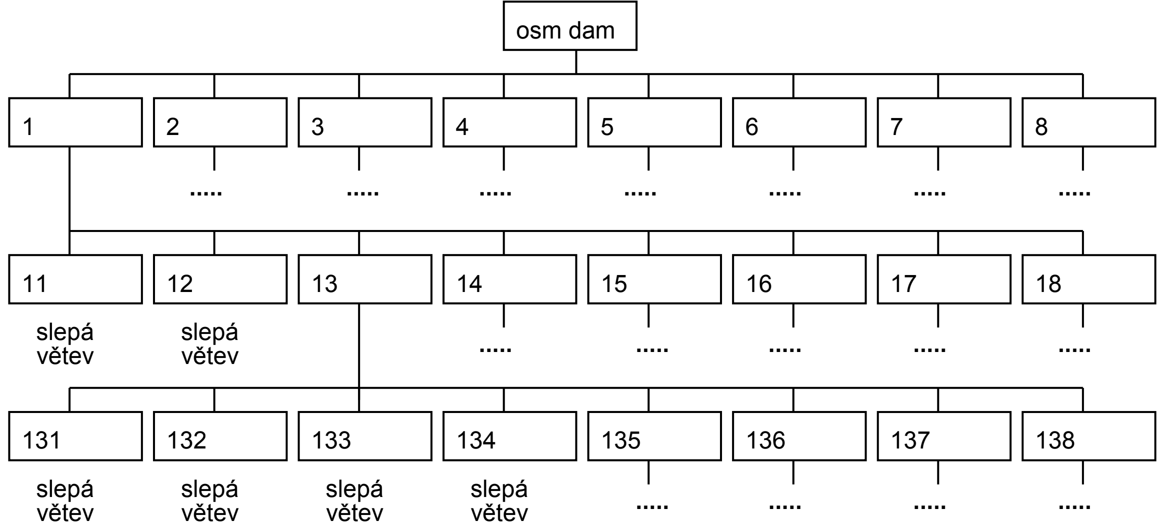
kde je f - ohodnocující funkce,

p_i - i-tý parametr (pozice dámy na i-tém sloupci).

Vyhodnocení ohodnocující funkce poněkud zjednodušíme tím, že budeme zkoumat dámy od sloupce 1 do 8 a jako správně umístěnou dámu považujeme tu, která neohrožuje žádnou dámu ležící na sloupci nižšího čísla. Počet všech možných rozložení dam určíme dle vzorce (8.4):

$$r = \prod_{i=1}^{8} 8 = 8^8 = 16777216 \doteq 17.10^6 \tag{8.4}$$

Jak vidíme, existuje skoro 17 miliónů různých rozložení. Z nich však jen některá jsou správná. Znázorněme si strom jednotlivých rozložení. Na každé nižší úrovni vždy přidáme jeden parametr. Je zřejmé, že tím rozvineme vždy 8 větví stromu. Hodnoty parametrů budeme pro jednoduchost zapisovat za sebou.



Obrázek 8.3 Strom všech možných rozložení dam

Jak vidíme na obr.8.3, ve stromu existuje řada slepých větví, které nemá smysl procházet, neboť v nich řešení nemůže existovat. Tuto skutečnost se pokusíme také využít.

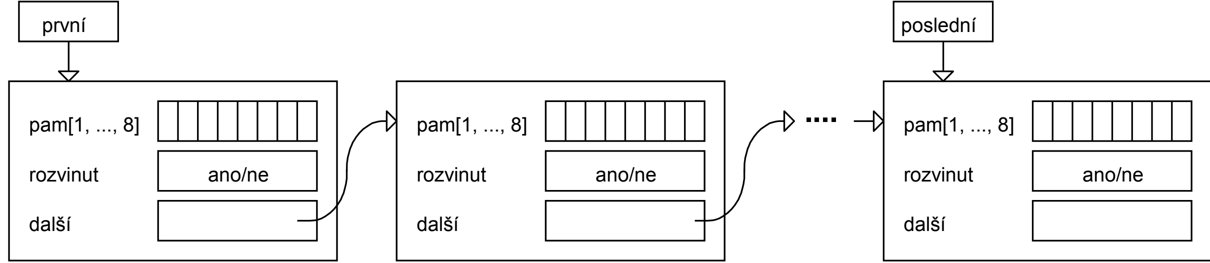
Nyní však již přejděme k jednotlivým algoritmům hledání řešení.

8.2 ALGORITMY NEHEURISTICKÉ

Tyto algoritmy vycházejí z předpokladu, že ohodnocující funkce (heuristika) jednoho řešení je zcela nezávislá na jeho poloze ve stromu řešení. To znamená, že řešení může ležet na jakékoliv úrovni vnoření do stromu a ve kterékoliv větvi. Jak víme, v námi řešeném příkladu to není pravda. Proto se budou dále uvedené metody jevit jako neefektivní. Uvádíme je však, neboť v jiných úlohách mohou mít své místo.

8.2.1 Prohledávání do šířky

Algoritmus ***prohledávání do šířky*** se snaží odstranit nebezpečí velmi hlubokého vnoření do stromu rozložení, pokud bychom se vydali po slepé větvi. Skutečnost, že je větev slepá, totiž zjistíme až vyzkoušením všech řešení, která na ní leží. Současně se snaží dosáhnout řešení co nejrychleji. Postupuje tak, že vždy rozvine všechny prvky na dané úrovni vnoření. Pokud se mezi nimi nenachází řešení, rozvine prvky další úrovně. Je zřejmé, že algoritmus má velké nároky na paměť, protože musíme pro každý prvek stromu rozložení, který použijeme, uložit informaci o parametrech řešení a pro rychlejší zpracování nejlépe také hodnotu ohodnocující funkce:



Obrázek 8.4 Datová struktura pro uložení vytvořených rozložení

V paměti budeme vytvářet ***lineární seznam***, ukazovátka ***první*** a ***poslední*** k němu umožňují přístup. Tím obcházíme nutnost zjistit předem řád stromu a vytvořit příslušnou stromovou strukturu. Je zřejmé, že pokud budeme chtít nalézt všechna řešení, dojde v konečném stavu k rozvinutí celého stromu rozložení. To obvykle není možné zajistit. Požadavky na paměť by byly příliš velké. Proto se tento algoritmus používá především tam, kde hledáme jedno řešení nebo úloha sama o sobě má jedno optimální řešení (maximum ohodnocující funkce), a to má nepříliš velkou hloubku vnoření do stromu. Postup práce pak můžeme zapsat:

```
počáteční nastavení
while není nalezeno řešení
  rozviň všechny nerozvinuté prvky a označ je jako
rozvinuté
  spočti ohodnocující funkci všech nových prvků
end while
vyhlas nalezené řešení
```

V paměti sice vytvoříme lineární seznam, ale budeme jím reprezentovat rozvíjení stromu rozložení. Pokud bychom chtěli vytvořit datovou strukturu odpovídající skutečně našemu stromu, pak by každý prvek musel obsahovat osm ukazovátek na další prvek, řada z nich by přitom nebyla využita (pokud nebyl prvek rozvinut) a hledání vhodného prvku k rozvinutí by bylo komplikovanější.

Z ukázky vidíme **výhody** algoritmu:

dobrá rychlost,

nehrozí nebezpečí nadměrného vnoření do stromu.

Musíme však upozornit také na **nevýhody**:

velké nároky na paměť,

složitější manipulace s datovou strukturou.

Poznámka:

Námi vytvořená datová struktura má zvláštní vlastnost, každý prvek nese informaci o tom, jak se k němu ve stromu rozložení dostat. Proto můžeme prvky, které již byly rozvinuty, z paměti odstranit. Manipulace se stromem bude složitější, ale ušetříme paměť.

8.2.2 Prohledávání do hloubky

Algoritmus ***prohledávání do hloubky*** se snaží odstranit velké paměťové nároky algoritmu prohledávání do šířky. V každém kroku je zvolen jeden prvek, ten je rozvinut a řešení je hledáno na nižší úrovni. Pokud není řešení nalezeno, je opět rozvinut některý z prvků s dosud největší hloubkou vnoření. Teprve pokud dorazíme k nejvyššímu možnému vnoření a řešení nenalezneme, vracíme se zpět a zkoušíme jinou větev stromu řešení. Pokud však řešení

neexistuje, dojdeme do stejného stavu jako při použití algoritmu prohledávání do šířky a tedy také ke stejným paměťovým nárokům.

Jako **výhody** algoritmu můžeme uvést:

postupujeme rychle do hloubky,

potřebujeme relativně málo paměti k provedení algoritmu.

Opět však nesmíme opomenout **nevýhody**:

pokud se řešení nachází blízko středu stromu, budeme ho hledat poměrně dlouho,

pokud nevíme, na jaké úrovni vnoření se řešení nachází a zejména pokud úroveň vnoření není omezena, může se stát, že budeme postupovat do velkých hloubek stromu bez úspěchu. Tento problém se někdy řeší ***prohledáváním do hloubky s omezenou hloubkou vnoření*** (s navracením). Jakmile dojdeme do určené hloubky, vracíme se zpět, bez ohledu na kvalitu nalezených řešení. V řešeném příkladu bychom nejspíš pracovali s maximální hloubkou = 8.

8.2.3 Vyzkoušení všech kombinací parametrů

Tento postup je na místě tam, kde nevíme nic o vlastnostech úlohy, a v plné míře platí, že neexistuje žádný vztah mezi hodnotami parametrů. Z vlastností stromu rozložení přitom víme, že řešení leží jedině na nejhlubší úrovni vnoření (na koncových prvcích). To znamená, že musíme vzít v úvahu všechny parametry. Přitom se budeme snažit zbavit velkých nároků na dostupnou paměť.

Postupně tedy zkoušíme všechna rozložení dam. V nejhorším případě vyzkoušíme všechna a teprve potom můžeme říci, že úloha nemá řešení. Pro vlastní realizaci využijeme vektor počítadel s hodnotami 1 ÷ 8. Vždy zvětšíme hodnotu posledního, pokud přesáhne povolený rozsah (8) nastavíme ho na 1 a zvětšíme počítadlo předchozí. Generování skončí přetečením rozsahu prvního počítadla.

Celý postup pak můžeme zapsat:

```
nastavení počátečního rozložení dam [1, 1, 1, 1, 1, 1, 1,
1]
while nepřeteklo první počítadlo
  if ohodnocující funkce aktuálního řešení = 8
        vyhlas řešení
  end if
  generuj další rozložení dam
end while
```

Uvedeným postupem se pohybujeme ve stromu možných řešení po koncových prvcích zleva doprava, až projdeme všechna rozložení. Přitom najdeme celkem 92 různých řešení této úlohy. Doba řešení však bude velmi dlouhá.

8.3 ALGORITMY HEURISTICKÉ

Heuristické algoritmy vycházejí ze zjištění, že nemá smysl postupovat při hledání řešení po stromu rozložení do prvku, který snižuje hodnotu ohodnocující funkce (heuristiky). To je samozřejmě možné jen tehdy, pokud se při cestě k prvku s řešením hodnota ohodnocující funkce nesnižuje. V našem příkladu víme, že musí při přechodu na další prvek vždy vzrůst. Abychom se pokud možno vyvarovali velkých požadavků na paměť, upravíme algoritmus prohledávání do hloubky na ***heuristické prohledávání***.

8.3.1 Heuristické prohledávání

Algoritmus heuristického prohledávání můžeme popsat následovně:

```
počáteční nastavení
while není nalezeno řešení
  najdi první dosud nerozvinutý prvek s maximální hodnotou
ohodnocující funkce
  rozviň nalezený prvek a označ ho jako rozvinutý
  spočti ohodnocující funkci všech nových prvků
end while
vyhlas nalezené řešení
```

Postup heuristického prohledávání ilustruje následující obrázek.

osm dam

1 2 3 4 5 6 7 8

vynecháno 7.8^7 možných řešení

11 12 13 14 15 16 17 18

vynecháno 7.8^6 možných řešení

131 132 133 134 135 136 137 138

vynecháno 7.8^5 možných řešení

1351 1352 1353 1354 1355 1356 1357 1358

vynecháno 7.8^4 možných řešení

13571 13572 13573 13574 13575 13576 13577 13578

vynecháno 7.8^3 možných řešení

135721 135722 135723 135724 135725 135726 135727 135728

vynecháno 7.8^2 možných řešení

1357241 1357242 1357243 1357244 1357245 1357246 1357247 1357248

vynecháno 7.8^1 možných řešení

13572461 13572462 13572463 13572464 13572465 13572466 13572467 13572468

řešení nebylo nalezeno, algoritmus se vrací na vyšší úroveň a pokračuje na další větev vpravo

Obrázek 8.5 Postup heuristického prohledávání stromu do hloubky

Z ukázky vidíme **výhody** algoritmu:

vynecháváme celé oblasti stromu rozložení, ve kterých nemůže řešení existovat,

potřebujeme dosti málo paměti k provedení algoritmu.

8.3.2 Vyzkoušení všech slibných kombinací parametrů

Tento algoritmus je na první pohled podobný předchozímu. Také se budeme pohybovat po stromu všech rozložení. Také budeme využívat skutečnosti, že existuje mnoho „slepých větví“, na kterých řešení ležet nemůže. Proto se jim budeme vyhýbat. K jejich nalezení využijeme ohodnocující funkci tak, jak byla definována dříve. Aby další řešení nebylo slepou větví, musí pro dílčí řešení na i-té úrovni stromu platit:

$$f\,(p_1, p_2, \ldots, p_i) = i. \tag{8.5}$$

Pak má smysl pokračovat ve stromu hlouběji rozvinutím tohoto prvku. Pro vlastní realizaci využijeme vektor počítadel:

pam[1 - 8] = pam[1], pam[2], ...pam[8]

a proměnnou i, obsahující číslo úrovně stromu rozložení, na které se nacházíme. Řešení tedy nalezneme tak, že platí podmínka (8.5) a současně $i = 8$. Celý postup pak můžeme zapsat:

```
vynulování pole počítadel
i = 1
while nepřeteklo první počítadlo
  pam[i] = pam[i] + 1
  if pam[i] > 8
        pam[i] = 0
        i = i - 1
  else
        if f(pam[1], ... pam[i]) = i
              if i = 8
                    vyhlas řešení
              else
                    i = i + 1
              end if
        end if
  end if
end while
```

Opět obdržíme jeho aplikací všech 92 řešení, oproti neheuristickému algoritmu vyzkoušení všech možných rozložení však výrazně rychleji.

8.4 ALGORITMY PRAVDĚPODOBNOSTNÍ

Uvedené algoritmy vycházejí z poznání, že pro náhodně zvolenou kombinaci hodnot parametrů existuje určitá pravděpodobnost, že tato kombinace určuje optimální řešení.

8.4.1 Metoda pokusu a omylu

Algoritmus ***náhodného přístupu*** je velmi neefektivní. Spočívá v náhodném generování rozložení, až dojde k vygenerování správného řešení. Jeho nízkou účinnost si snadno uvědomíme, pokud určíme pravděpodobnost náhodného nalezení řešení při jednom pokusu. Víme, že existuje 92 řešení při 16 777 216 možných rozloženích dam. Pravděpodobnost úspěchu jednoho pokusu tedy je:

$$P(1) = \frac{92}{16777216} \doteq 5{,}48.10^{-6} \tag{8.6}$$

Pravděpodobnost neúspěchu při jednom pokusu určíme snadno jako

$$\overline{P}(1) = 1 - P(1). \tag{8.7}$$

Samozřejmě budeme pokusy opakovat, a tak nás zajímá, jaká bude pravděpodobnost, že při ***n*** pokusech najdeme alespoň jedno řešení (nebo 2, 3,...,n). Nejprve určeme, jaká je pravděpodobnost, že řešení nenalezneme žádné. Jedná se o sjednocení ***n*** vzájemně nezávislých jevů (nenalezení při prvním pokusu, při druhém pokusu atd.):

$$\overline{P}(n) = \prod_{i=1}^{n} \left(1 - P(1)\right) = \left[1 - P(1)\right]^n . \tag{8.8}$$

Pravděpodobnost, že alespoň jedno řešení najdeme, je pak:

$$P(n) = 1 - \left[1 - P(1)\right]^n \tag{8.9}$$

Pro nás bude jistě zajímavé, kolik pokusů je třeba vykonat, abychom se zvolenou pravděpodobností našli alespoň jedno řešení. Využijeme vzorce (8.9) a zjistíme:

$$n = \frac{\log\left[1 - P(n)\right]}{\log\left[1 - P(1)\right]} . \tag{8.10}$$

Např. pro pravděpodobnost nalezení alespoň jednoho řešení 10 % je třeba vykonat 19 214 pokusů.

Algoritmus náhodného přístupu nevypadá nijak povzbudivě. Přesto může být v některých úlohách úspěšně používán. Určitě bychom ho doplnili o paměť nejlepšího dosaženého řešení, takže při ***n*** pokusech dosáhneme poměrně dobrého řešení.

Mimochodem, ze vzorců je zřejmé, že pro dosažení 100 % jistoty nalezení nejlepšího řešení musíme vykonat nekonečně mnoho pokusů.

8.4.2 Genetický algoritmus

Genetický algoritmus využívá pravděpodobnostního přístupu a velmi vhodně jej rozvíjí. Vychází z pozorování přírody, která dokáže úspěšně řešit i velmi složité problémy. Z analogie s živými organismy popišme naší úlohu následovně.

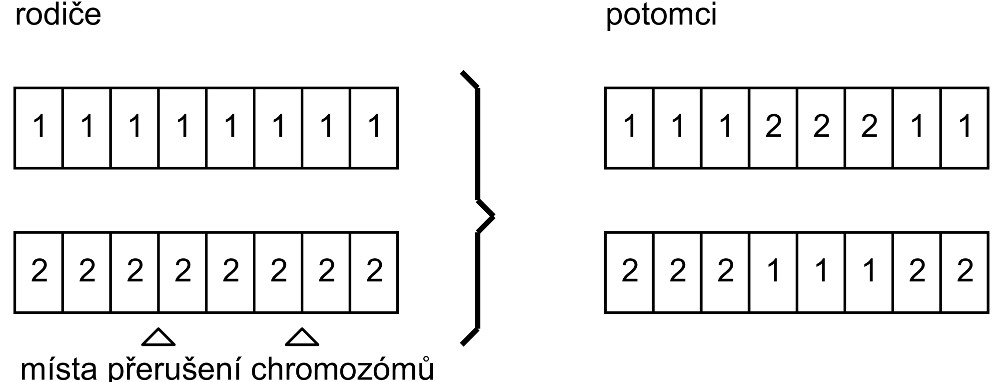
Vlastnosti řešení (rozložení dam) rozdělíme na samostatné prvky, v našem případě to bude umístění jednotlivých dam, a nazveme je ***geny***. Skupina genů, která plně popisuje úlohu, pak bude ***chromozóm***. Ten bude současně vyjadřovat vlastnosti jednoho konkrétního rozložení dam, konkrétního řešení, tedy jednoho ***jedince***. V přírodě se však současně vyskytuje více jedinců, celá ***populace*** jedinců různých vlastností. Tito jedinci spolu vytvářejí ***potomky***, tedy jedince následující ***generace***. Potomci přebírají vlastnosti svých ***rodičů*** do značné míry náhodně, ale pravidlo přirozeného výběru říká, že nejvíce se množí nejlepší jedinci, zatímco nejhorší hynou.

Uvedené principy využijeme v naší úloze. Začneme tím, že náhodně vytvoříme počáteční populaci. Jedince seřadíme podle jejich kvalit. K tomu využijeme již známé ohodnocující funkce. Nejhorší jedince zlikvidujeme a nahradíme potomky nejlepších jedinců. Přitom je vhodné, aby nejlepší jedinci přešli do následující generace beze změny, jinak bychom mohli ztratit nejlepší dosažené řešení.

Jak vznikají potomci. K jejich vytvoření se používají operace, které mají opět analogii v přírodě. Jsou to ***reprodukce*** a ***mutace***.

8.4.2.1 Reprodukce

Při reprodukci vzniká ze dvou jedinců (***rodičů***) nový jedinec (***potomek***), který přebírá části chromozómů svých rodičů. Kombinace vlastností probíhá tak, že chromozómy rodičů rozdělíme na několik úseků, které přebírají potomci. Místa rozdělení chromozómů jsou volena náhodně. Obvykle je jich 1 ÷ 5 v závislosti na počtu genů.



Obrázek 8.6 Průběh reprodukcí jedinců

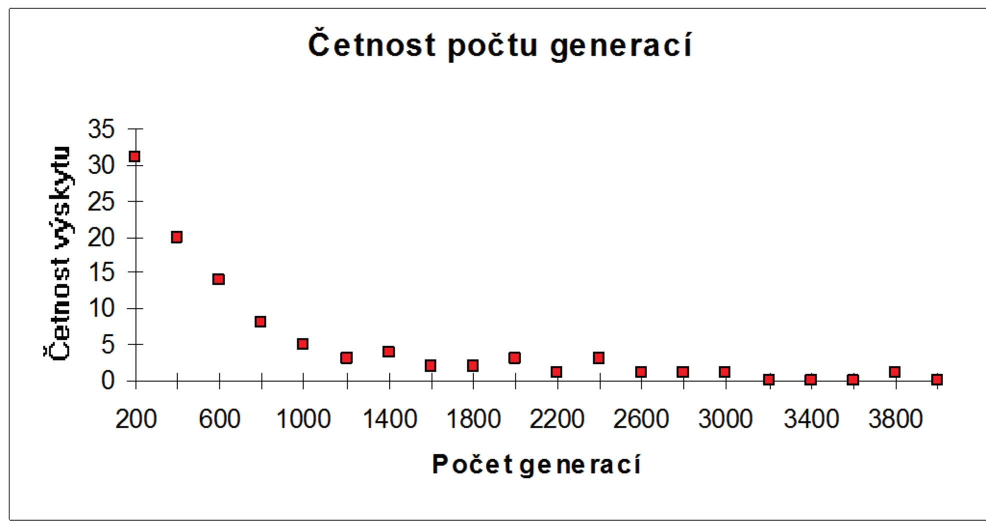
8.4.2.2 Mutace

Mutace jsou náhodné změny hodnot genů, které dovolí prohledávat okolí nového chromozómu. V algoritmu je nutné stanovit maximální počet změn (obvykle je také náhodný) a maximální změnu hodnoty genu.

Po naplnění nové populace je opět provedeno seřazení jedinců a vytvoření další generace. Algoritmus končí nalezením řešení. Protože konvergence algoritmu může být pomalá, bývá často omezen maximální počet populací. Po jejich provedení sice není zaručeno, že najdeme optimální řešení, ale při vhodném nastavení parametrů algoritmu obvykle najdeme velmi dobré řešení. Genetický algoritmus pracuje s náhodným přístupem, ale díky jeho paměti a variaci s nejlepšími dosaženými výsledky dosahuje výrazně lepší výsledky než metoda pokusu a omylu.

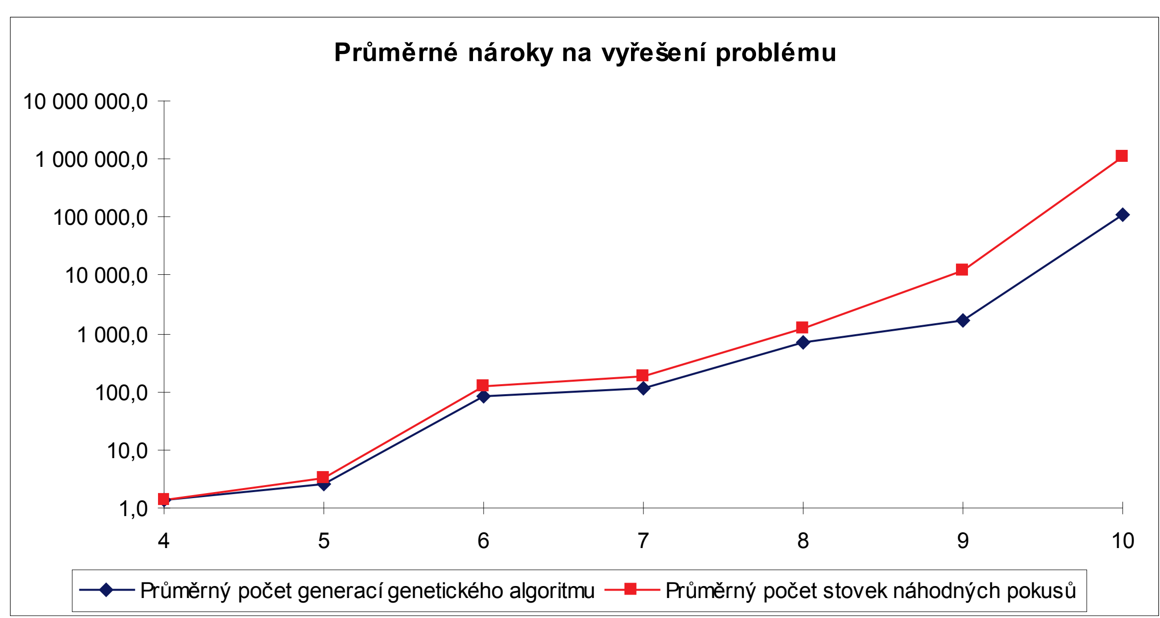
Celý postup je velice jednoduchý. Z pohledu algoritmu není nutná znalost zákonitosti systému (vztahy při umísťování jednotlivých dam), jen popis vlastností systému (ohodnocení kvality jedinců), tedy již několikrát použitá ***ohodnocující funkce*** (***heuristika***). Tato malá znalost je nahrazena velkým výpočetním výkonem. Bohužel nikde není zaručeno, že najdeme skutečně optimální řešení, pokud nevznikne nekonečně velký počet generací.

Testováním algoritmu na problému 8 dam (chromozóm má 8 genů s hodnotami 1, 2, ..., 8) s velikostí populace 100 jedinců, 2 místa přerušení při reprodukci, maximálně 3 mutace v rozsahu změny hodnoty genu maximálně -3 ÷ 3 byl ze 100 průběhů hledání řešení určen střední počet potřebných generací ve výši 685 a jeho rozptyl ve výši 742. Na průběhu četnosti vidíme, že počet generací je náhodná veličina s exponenciálním rozdělením.



Obrázek 8.7 Průběh četnosti počtu generací nutných k nalezení řešení

Z hlediska hodnocení složitosti genetického algoritmu je zajímavé srovnání průměrného počtu generací nutných k nalezení řešení. Na obr.8.8 vidíme v logaritmických souřadnicích srovnání průměrného počtu generací s průměrným počtem náhodných pokusů (po 100 pokusech). Z průběhu lze usuzovat, že genetický algoritmus má exponenciální složitost, nároky na dosažení řešení však rostou výrazně pomaleji než u náhodných pokusů.



Obrázek 8.8 Průměrné nároky na vyřešení problému různými metodami v závislosti na jeho složitosti

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ĺ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ĺ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5 – Uživatelská příručka*. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms.* 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.